DM

Realtime Image Noise Reduction FPGA Implementation With Edge Detection

MASTER DISSERTATION

Ricardo Jorge Ferreira Jardim

MASTER IN ELECTRICAL ENGINEERING - TELECOMMUNICATIONS

UNIVERSIDADE da MADEIRA

A Nossa Universidade

www.uma.pt

February | 2020

Realtime Image Noise Reduction FPGA Implementation With Edge Detection

MASTER DISSERTATION

Ricardo Jorge Ferreira Jardim
MASTER IN ELECTRICAL ENGINEERING - TELECOMMUNICATIONS

SUPERVISION
Fernando Manuel Rosmaninho Morgado Ferrão Dias

Abstract

The purpose of this dissertation was to develop and implement, in a Field Programmable Gate Array (FPGA), a noise reduction algorithm for real-time sensor acquired images. A Moving Average filter was chosen due to its fulfillment of a low demanding computational expenditure nature, speed, good precision and low to medium hardware resources utilization. The technique is simple to implement, however, if all pixels are indiscriminately filtered, the result will be a blurry image which is undesirable.

Since human eye is more sensitive to contrasts, a technique was introduced to preserve sharp contour transitions which, in the author's opinion, is the dissertation contribution. Synthetic and real images were tested. Synthetic, composed both with sharp and soft tone transitions, were generated with a developed algorithm, while real images were captured with an 8-kbit (8192 shades) high resolution sensor scaled up to 10×10^3 shades.

A least-squares polynomial data smoothing filter, Savitzky-Golay, was used as comparison. It can be adjusted using 3 degrees of freedom ─ the window frame length which varies the filtering relation size between pixels' neighborhood, the derivative order, which varies the curviness and the polynomial coefficients which change the adaptability of the curve. Moving Average filter only permits one degree of freedom, the window frame length. Tests revealed promising results with 2^{nd} and 4^{th} polynomial orders. Higher qualitative results were achieved with Savitzky-Golay's better signal characteristics preservation, especially at high frequencies.

FPGA algorithms were implemented in 64-bit integer registers serving two purposes: increase precision, hence, reducing the error comparatively as if it were done in floating-point registers; accommodate the registers' growing cumulative multiplications. Results were then compared with MATLAB's double precision 64-bit floating-point computations to verify the error difference between both. Used comparison parameters were Mean Squared Error, Signal-to-Noise Ratio and Similarity coefficient.

Keywords:

Denoising, FPGA, contour detection, Moving Average, Savitzky-Golay.

Resumo

O objetivo desta dissertação foi desenvolver e implementar, em FPGA, um algoritmo de redução de ruído para imagens adquiridas em tempo real. Optou-se por um filtro de Média Deslizante por não exigir uma elevada complexidade computacional, ser rápido, ter boa precisão e requerer moderada utilização de recursos. A técnica é simples, mas se abordada como filtragem monotónica, o resultado é uma indesejável imagem desfocada.

Dado o olho humano ser mais sensível ao contraste, introduziu-se uma técnica para preservar os contornos que, na opinião do autor, é a sua principal contribuição. Utilizaram-se imagens sintéticas e reais nos testes. As sintéticas, compostas por fortes e suaves contrastes foram geradas por um algoritmo desenvolvido. As reais foram capturadas com um sensor de alta resolução de 8-kbit (8192 tons) e escalonadas a 10×10^3 tons.

Um filtro com suavização polinomial de mínimos quadrados, Savitzky-Golay, foi usado como comparação. Possui 3 graus de liberdade: o tamanho da janela, que varia o tamanho da relação de filtragem entre os pixels vizinhos; a ordem da derivada, que varia a curvatura do filtro e os coeficientes polinomiais, que variam a adaptabilidade da curva aos pontos a suavizar. O filtro de Média Deslizante é apenas ajustável no tamanho da janela. Os testes revelaram-se promissores nas $2ª$ e $4ª$ ordens polinomiais. Obtiveram-se resultados qualitativos com o filtro Savitzky-Golay que detém melhores características na preservação do sinal, especialmente em altas frequências.

Os algoritmos em FPGA foram implementados em registos de vírgula fixa de 64-bits, servindo dois propósitos: aumentar a precisão, reduzindo o erro comparativamente ao terem sido em vírgula flutuante; acomodar o efeito cumulativo das multiplicações. Os resultados foram comparados com os cálculos de 64-bits obtidos pelo MATLAB para verificar a diferença de erro entre ambos. Os parâmetros de medida foram MSE, SNR e coeficiente de Semelhança.

Palavras-chave:

Ruído, FPGA, deteção de contorno, Média Deslizante, Savitzky-Golay.

Acknowledgements

This work represents another important stage of my life and is the culmination of a series of steps which may not have been achieved without the precious contribution of several people, made available in several forms, including their knowhow, psychological support, understanding and patience.

First, to my supervisor, Professor Fernando Morgado-Dias, for the opportunities given, and there were several, including conferences participation, for his guidance on this dissertation and his understanding. This acknowledgment extends since from under graduation course.

To my wife Anna, for her unconditional support, understanding, and coolness, especially on less reasonable moments.

To my exceptional parents that have been campaigning me since from the womb, for their support through the route of my life, particularly for their confidence and friendship.

To several friends related to the academic world and from outside of it, for their support and suggestions, that directly or indirectly have contributed to this Master Dissertation.

List of Acronyms

AI	–	Artificial Intelligence
AWGN	–	Additive White Gaussian Noise
AXI	–	Advanced eXtensible Interface
CLB	–	Configurable Logic Block
CPU	–	Central Processing Unit
DMA	–	Direct Memory Access
DSP	–	Digital Signal Processor
FWC	–	Factor Wheel Conversion
FPGA	–	Field Programmable Gate Array
FIFO	–	First Input First Output buffer
IEEE	–	Institute of Electrical and Electronics Engineers
I/O	–	Input/Output
IP	–	Intellectual Property
LUT	–	Look-Up Table
MSE	–	Mean Squared Error
PCIe	–	Peripheral Component Interconnect express
PDF	–	Probability Density Function
PSNR	–	Peak Signal-to-Noise Ratio
RAM	–	Random Access Memory
RGB	–	Red, Green, Blue
RTL	–	Register-Transfer Level
SMF	–	Sigma Multiplicative Factor
SNR	–	Signal-to-Noise Ratio
TCP/IP	–	Transmission Control Protocol/Internet Protocol
USB	–	Universal Serial Bus
VHDL	–	Very High-Speed Integrated Circuit Hardware Description Language

List of Symbols

μ	–	Mean
σ	–	Standard deviation
σ^2	–	Variance
MHz	–	Mega Hertz
N	–	Number of pixels (filter size)

Contents

1. Introduction

This Master Dissertation is the culmination of a research from which resulted two papers, one entitled "*Image Denoise FPGA Implementation using a Moving Average Filter with Contour Detection",* which was presented on the International Conference of Biotechnology and Engineering Applications (ICBEA) 2018 having been peer reviewed and published through Institute of Electrical and Electronics Engineers (IEEE) (https://ieeexplore.ieee.org/document/8471740) and a second paper, entitled "*Savitzky-Golay filtering as Image Noise Reduction with Sharp Color Reset",* which was submitted to the Evise ─ Microprocessors and Microsystems Journal paper (https://www.evise.com/), which was, as well, peer reviewed and accepted to be published on Volume 74, in April 2020, with the number 103006. Its DOI webpage is https://doi.org/10.1016/j.micpro.2020.103006.

The dissertation was initially projected, as was its purpose, to reflect the work developed on the first paper, a time-domain noise reducing filter for real-time acquired images by image sensors connected to low complex, low power, digital processing units such as those normally found in small cameras or smartphones, more specifically, on FPGAs. However, the work further extended leading, lately, to a second published paper, which scrutinizes a filter that better preserves the signal characteristics, especially on the high-frequency spectrum, the Savitzky-Golay filter.

Noise is a general and abroad concept which basically means a random uncorrelated statistical distribution which, at some point, can be approached by one or more processes such as Gaussian, Rayleigh, Beta, or higher order distributions such as Dirichlet. More complex noise patterns can follow yet other distributions or compositions of them difficult to model.

In the context of the signal processing and, especially of this dissertation, besides, following on how image acquisition areas have been traditionally dealt with, it will be referred as a Gaussian white random process, normally known by Additive White Gaussian Noise (AWGN). It is *white* because the range of signals that composes a specific bandwidth of interest have all the same amplitude. It is a *random* process because it cannot be predicted when a specific spectral component will appear at a specific time. Finally, it is *Gaussian* because it can be described as a probabilistic process that follows a distribution with the same name with.

Probabilistic processes can be described with a minimum of two parameters such as mean, μ, and standard deviation, σ. However, in Electrical Engineering, it is common to refer to the latter as variance, σ^2. Variance

enhances the visualization of the error growth comparatively to the standard deviation because it imputes more emphasis the larger it develops, instead of a linear growth as standard deviation does.

When light is captured by an image sensor, the process begins with photons-to-electrons conversion in semiconductor electronic components, as shown in Figure 1.1. This conversion process is made up to a rate which is proportional to the light conditions. The signals are diffused through the distribution channels and lanes, passing through passive components and are delivered with an amplitude which is resultant from the referred conversion.

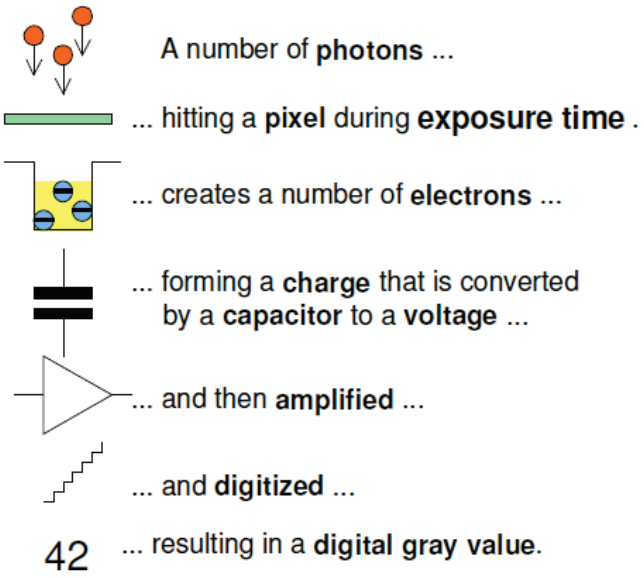

Figure 1.1- Photons to pixels process [1].

The final process is the gathering of all these individual values in a matrix which, after adjusted, will lead to the generation of pixels, the minimal picture fragment or unit of light that composes an image. If a pixel is monochromatic, then is only defined by gray shades and has one subpixel. If it is a color pixel, then it is normally defined by 3 subpixels, corresponding to the 3 basic colors Red, Green and Blue (RGB). Figure 1.2, shows an RGB image composed by pixels in Bayer pattern disposition.

This light capture process suffers oscillations which have consistently been verified to follow an AWGN curve with mean, $\mu = 0$, and a certain variance, σ^2. Higher variances mean noisier images. Under low light conditions noise becomes granularly more perceptible. During the development of this project this condition was verified and addressed with the creation and implementation of an inverse non-linear function.

Noise generated in an image can be composed by a contribution of both a signal-independent and a signal-dependent parts. The independent part,

which cannot be directly controlled and therefore can only be estimated, is due to the contribution of the photons, i. e., light intensity, weather conditions, scattering phenomena, etc.

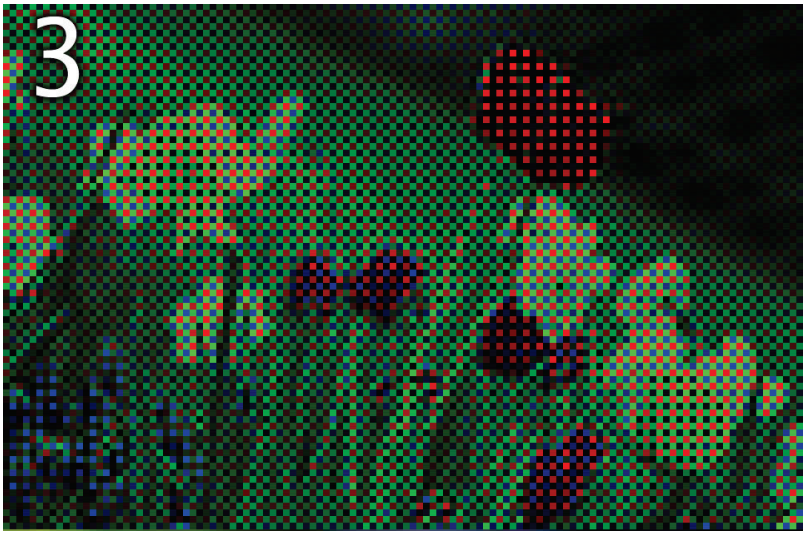

Figure 1.2 - Image composed by pixels in a Bayer pattern disposition [2].

On the other side, a wafer manufacture process destinated to make image sensors can itself introduce non-uniform linearities contributing ultimately to the noise generation. It is, however, a process that can be mitigated and corrected and up to a certain degree, it is a characteristic of the non-homogenic semiconductor doping process. This is also regarded as the dependent-signal noise contribution [3–6].

1.1. Motivation

Photosensors are ubiquitous our days, making it integral part of video and photo cameras. They are spread out all over and a constant presence of the modern's lifestyle. Embedded into personal smartphones ranging through standalone photo snappers, professional studio machines until the huge industrial, printing process controllers, pictures are taken anywhere, at any time, with any light conditions, from the dawn to the dusk, even during the night.

With acceptable brightness, the number of photons acquired by camera sensors are enough to be evenly converted by the devices' circuitry to electrons and distributed through the pixel matrix to compose and generate pictures given a good brightness/contrast ratio. However, when that is not the case and, below certain brightness levels, it can be perceptive a certain image granularity proportionally inverse to the light conditions that was gathered at the shooting time, due to this sparse photons-to-electrons conversion. All this process introduces some level of noise on the images they generate. However, the purpose of this work is to focus on noise reduction for devices of low-power consuming, low-processing capabilities.

Despite some of the high range performers' smartphones to include low precision, low bit depth, fast dedicated hardware oriented for Artificial Intelligence (AI) and self-learning capabilities targeting quick audio and video recognition, they often lack specialized Digital Signal Processors (DSPs) for immediate noise processing coming right out from the sensors, such as embedded FPGAs. This voids the parallelization spurs of the whole set of device's operations leaving main Central Processing Units (CPUs) overburden with all sorts of tasks, that could otherwise, have been split among generalists and dedicated processors.

Several techniques exist for noise reduction as this has been a hot topic from 20–30 years now. However, this is a non-trivial choice process and should mandatorily consider tradeoffs between, by one side, fast and simple techniques, even though, with less satisfactory results and, on the other side, more complex, time and processing resources consuming but qualitative ones. In situations where speed and quality cannot be achieved, preference, quite often, is given to speed over quality and this is particularly valid in real-time situations.

On all, noise reduction techniques, especially when applied to pictures, are interpreted as psychovisual outcomes which may not generally have an immediate connection with measurable parameters such as MSE, SNR and Local or Global Similarity. In addition, because of time limitations, it is very difficult to develop global and scalable solutions in order to exploit the immense assortment of DSPs and CPUs processing devices, including FPGAs which is the case of the current work's plan.

Meaningful outcomes are questionably connected with complex computations evolving transforms between physical domains, namely between time and/or space and frequency domains, use of trigonometric functions, calculus and heavy numbering computations. Acceptable precision results might not be achieved if floating point computations does not come into question, and consequently, real time results might be conditioned even though considering the use of Look Up Tables (LUTs) which can sky rock memory usage the more the precision is demanded.

These issues led to the development of two real-time algorithms, one being the focus of this dissertation initial work, which can reduce the noise and lead perceptually, up to a certain degree, to granular-free images. The first is a less computationally expensive algorithm, and the second one, a slightly more advanced and a more effective algorithm at the signal integrity preserving especially at higher frequencies, where noise reduction operations are critical. The development was done towards precision but without threatening the

expected ready availability of the results, thus maintaining low complexity with a minimum of power consumption.

Initially, the work of research started on the AWAIBA enterprise having interest in the development of a low computational algorithm looking forward minimizing the generated noise out of their sensors. In that sense, this dissertation can be regarded as an important contribution in order to parametrize and hence, take the most out of their sensors for matters of low noise image quality.

1.2. Objectives

The objective of the initial project was to study the noise generation in images on sensors in general, verifying the relationship on how varying the light conditions does that affect the granularity in the inverse proportion, that is, under low light conditions the photons are scarce and spreads through the image unevenly, leading to uncontrolled signal variations.

Always when possible, precautions were taken to detach these processes, making them independent from a specific architecture and focus on the process of the image formation and to the noise attached to it. Hence, although tests were done with images sourced from a particular image sensor, processed with a certain FPGA through Universal Serial Bus (USB) channels, the interest fall on techniques of noise reduction which can be extended to a multitude of sensors and platforms. Thus, during the execution of this project, a plan of work was respected that guaranteed the following objectives:

1. Research on photons-to-electrons conversion principle, including theory and equations, how does it lead to an image construction, and how and in what conditions can result in noise generation.

2. To study and understand the existing Very High-Speed Integrated Circuit Hardware Description Language (VHDL) source code for image capture, pre-processing and interface routing, identifying control signals pretraining to sensors image acquisition.

3. Image capture under several ambient light conditions and study of the noise generation under these conditions regardless of the underlying associated equipment used in the acquisition process, including, image sensors, signal processing platforms and signal routing used for propagation through related processing stages .

4. Study of image sensors exposition controlling methods to the light and

source light intensity dynamics control in order to mitigate and reduce noise generation, leading, lately, to signal processing solution complexity reduction.

5. Development and simulation of a fixed-point solution in parallel with a floating-point in MATLAB and comparing the error before porting to VDHL language and implemented on an FPGA.

6. Analysis and constraints of a real-time image acquisition and denoise processing implementation.

7. Development and integration of two algorithms, in VHDL code, for an FPGA platform, making comparisons between both and additional development of MATLAB support programs to test, as well as, to compare its efficiency.

1.3. Contents framework

The sequence of this dissertation, which resulted in two papers is structured in 6 chapters – Introduction, State of Art, Image Denoise FPGA Implementation using a Moving Average Filter with Contour Detection, Savitzky-Golay filtering as Image Noise Reduction with Sharp Color Reset, Testing Platform, Conclusion and Appendices.

Chapter one starts with the introduction, a quick reference of the noise concept and how it is generated, it expresses the motivation behind today's demanding for low processing computational algorithms using the power of FPGA's configurable blocks. These devices possess a parallelizing processing nature which unclutter the bottlenecks of the CPU's different paradigm. Finally, how this dissertation is structured through this contents' framework.

In Chapter two, a research is made through state of art with a quick introduction on noise generation on an image to help to understand what the tradeoffs are between choosing some determinate technique over another, particularly in a less expensive computational realm and immediately post-sensor delivering data.

Chapter three is the IEEE paper which resulted of the work done in the sequence of this dissertation, which was presented on ICBEA18 International Conference of Biomedical Engineering and Applications and the focus and purpose of this dissertation.

Chapter four is an evolution paper with a different filtering technique to

the previously work done. While the moving average filter is a constant coefficients filtering technique, the Savitzky-Golay filter is an adaptative least squares error curve fitting. This paper is now published by Elsevier, Journal of Microprocessors and Microsystems - Embedded Hardware Design.

Chapter five describes in detail the two developed solutions and extends the characteristics of the platform used to implement and to test these techniques.

Chapter six draws general conclusions and comments the results.

The Appendices state the VHDL source codes used on the testing FPGA and the C source codes used on the CPU side.

1.4. Own work

Although similar techniques exist, it is believed that this work contributes with an original computational FPGA's image processing with contour color preserving technique that is computationally speedy, with results comparable to those of the most complex processing devices and algorithms.

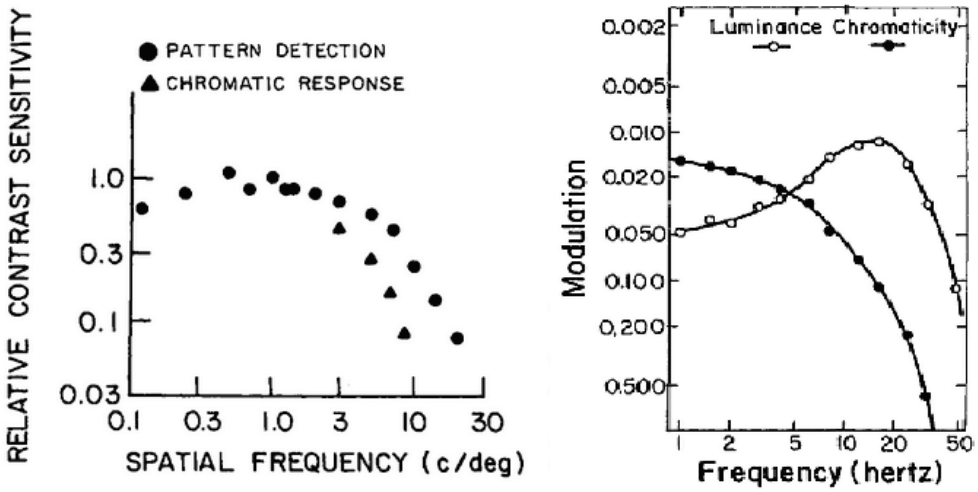

Figure 1.3 – Human eye contrast sensitivity tests for 2 different tones. On the left, the dots, correspondent to pattern detection, is 3x more sensitive than for chromatic detection. Spatial frequency is expressed in cycles/degree of modulation change. On the right, the temporal CSF modulation for the same 2 colors [7].

Several studies [7]–[8] point that Human eye is more sensitive to the Luminance, than Chrominance, that is, except for low modulation frequencies, humans can better spot changes in contrasts or shapes than in colors, as can be seen in Figure 1.3's study [7].

Therefore, the author of this dissertation proposes two real-time algorithms, which aim to denoise images acquired with low to mid-range image sensors, while preserving the images' contrasts. The two solutions already

exist. However, the author believes that the images' preservation technique, that is, the images' contour preservation is his own contribution.

The development is to be applied to a very fast-processing platform such an FPGA using the VHDL hardware description language.

2. *State of Art*

In this section the basics principles of the signal processing are quickly reviewed, since they are further analyzed in Chapters 3 and 4, the chapters which have resulted from the work developed for this dissertation. Thus, the aim of this work focusses particularly in computational techniques for real-time noise reduction applied to images, using FPGAs as main processing platform.

The most common used methods are referenced, from the simplest to the most complex ones pointing, and whenever necessary, it will be point out each one's strengths and weaknesses.

Additionally, it is referenced, the tradeoffs for opting for some determined architecture considering the complexity of the algorithm in question. Finally, it is referenced the algorithms' development platform for both, implementation and testing.

2.1 Noise generation

2.1.1 Introduction

Noise can be described as a random or stochastic, undesirable and uncorrelated signal that sums up to the signal of interest [9]–[11]. In turn, a stochastic signal can be part of a broader family of discreet-time Probability Density Functions (PDFs). It is understood that, at any given moment, a certain signal can be modeled by one or a combination of PDFs. Thus, a signal, can be interpreted as set of individual or jointly probabilities which are specified by a sequence of random variables distributed in the axis of time [9].

A physical observable signal can be a superimposition of a random, correlated signal we want to extract information from and an uncorrelated, stationary in average, disturbing signal, known as noise [12]–[13].

$$observable\_signal = interest\_signal + noise$$

in other words,

$$d_i = f(t_i) + \vartheta z_i \tag{2.1}$$

where t_i is the independent time variable defined as $t_i = \frac{i}{n}$, n is a number of random samples, i is the indexed sample, $z_i \overset{iid}{\sim} N(0,1)$ is an AWGN and ϑ is the attached noise [11].

This superimposition can happen at the moment of the creation or the transformation of the observable signal thus, the latter does not possess the

same integrity as of that the time of its creation [12]. Therefore, unless the attached noise is added deliberately with the purpose of scrambling or hide the relevant signal, or the signal is to be emerged below the noise threshold level for the same purpose, this uncorrelated noise is undesirable [14].

In turn, noise does not have a physical meaning itself, neither can be measurable isolatedly hence, it can be regarded as a disturbance to the model or to the system which is not part of [11].

In the imagery creation context, i. e., created by image sensors, signal's attached noise can be a contribution from two sources: one, from a dependent source of noise which is characterized as photonic noise; the other one, from an independent source which is generated in the device's circuitry. The later, has trendily been reducing over times along with technical progress [3].

It is then desired to filter the interest signal out of the noise in order to obtain, as much as possible, an $observable\_signal = interest\_signal$. Noise can be generated from several sources: thermal, shot, flicker, burst and others [15].

2.1.2 Noise generation in a signal

Considering the amount of types of noise that exist and assuming that they all are unwanted stochastic processes that sum up to the signal of interest, whatever its origin, the noise interferes with the information signal, distorts and degrades its quality [15].

Often, the noise is considered to be an information-carrier, with a proper pattern which is revealing of its origin, whether it is acoustic, thermal, electronic, shot, electromagnetic, cosmic and several others. In this sense, a distorted signal has some degree of certainty to carry noise attached to it, drifting the signal itself from its ideal conditions [13], [15]. For example, on a transmission line the noise is one of the main limitative factors of the signal integrity at the destiny given that the initial transmission conditions are ideal [12]–[13], [15]–[16].

Noise itself can be, and often is, modulated by a statistical model which, depending on its source, can follow one or more Probability Distribution Functions (PDFs). One of the mostly considered PDFs, in all engineering disciplines, particularly in Electrical Engineering, which includes the creation of pseudorandom noisy models for synthetic testing purposes, is the White Noise with a Gaussian PDF, that is, Additive White Gaussian Noise (AWGN). AWGN is known to have a consistent and flat signal spectrum density. Jamming, cryptography, communication and electronic calibration systems are examples

of such a deliberate utilization [17]– [18].

Figure 2.1 - An image, on the left, without noise, and on the right with AWGN [26].

2.1.3 Noise generation in an image

A pure informative signal is a generalized term which defines a set of random variables, strongly correlated, whose values or amplitudes can be defined, at any moment, by one or more sets of probabilistic functions, specifically PDFs. A pure image is also a signal which is subjected to the same random nature disturbances as the signal is [9].

Image sensor photon detection leading to a pixel generation is subject to some perturbations which are the result of both, the manufacture process, i. e., non-homogenic doping process in the whole surface of the semiconductor material and certain light conditions [17].

These perturbations might lead to a perceptible granularity in the image, as can be seen on the right side of the Figure 2.1, which is to say, a random tone variability, known as variance, within a similar tone area, known as *noise*. The aforementioned process is of an uncorrelated nature, and it has been verified to exhibit a probabilistic Gaussian normal behavior [9], [17].

2.1.4 Image denoising techniques

Several algorithms have been studied through the last 2 to 3 decades. On one side, there are simple time domain denoise techniques with quite good results but poor band stopping. On the other side, it exists a sheer amount of frequency domain denoising techniques, several of them with stellar results [4].

In the field of our interest, noise generation resulting from a photoelectron counting conversion is a hypothesized electromagnetic signal which is defined statistically to be deterministic, having a zero-mean, and a Gaussian shape-like which can be expanded by Karhunen–Loéve series of orthogonal functions.

This type of source generated noise can be modeled by Laguerre polynomials [19].

Discreet Cosines is a technique of low complexity which may or may not use orthogonal transforms. Can be applicable to random shape signals, thus to images, yet using relatively low computational resources [20].

A new wavelet approach of point despeckling or smoothing a high variability signal that can produce good results on low gradient transitions while still preserving most of the image's signal energy. It should not be considered for this dissertation purposes because it requires expensive frequency transforms computations [4]–[5].

To preserve most of the image's signal energy techniques exist based on undecimated wavelet decomposition with the need of having to work with time or space and frequencies domains [21].

A general denoising model for images, can be achieved by employing several mathematical fields such as numerical analysis, statistical models, weighted similarities between pixels and patches with several expectations, besides statistics, for example, in color multiscale analysis and graph-based data representation where the employment of quadratic Laplacian forms are used in order to obtain information about the energy distribution and from there the signal's filtering technique to apply. This is typically used jointly with wavelet transforms [20].

If the noise model is known to be uncorrelated, which in its majority is, a generalized similarity approach assumption among pixels and patches with dependency of the distribution model of the noise may be applied with the use of weights which can be tweaked with the last assessment similarity. This scheme can improve the SNR, with images affected by additive Gaussian and by multiplicative speckle noise models [23].

A shape-adaptative technique using Discreet Cosines Transforms (DCT) can be combined with local polynomial approximation and anisotropy to filter shape-adaptative blocks. It works by compressing an image in DCT blocks then to compute the confidence intervals to find the transform's support shape on a pointing adaptative method. With established appropriate threshold values, it can be found the estimation coefficients [24]–[25] that serve to reconstruct the signal's support shape. This technique somehow imposes restrictions in the luminance-chrominance space in order to increase the accuracy of the image's colors. This adaptative reconstruction enhances the color transitions (cleanses undesired artefacts) [20], [26].

Strictly time domain techniques such as moving averages [27] and polynomial curve fits using least squares errors [28] are among the most used with very good results and these are the main focus of this dissertation.

All image sensors, namely photo and video sensors add noise to the visual information (images) they acquire. This phenomena correlates several principles which will be further developed in the next chapters, but to get an idea they include but are not limited to, the scattering light conditions at the moment of the photo snapping, the photons-to-electrons conversion process on the circuitry upon the pixels acquisition on the sensor, known as Factor Wheel Conversion (FWC) and the thermal noise generated from two sources: the atmospheric and the devices' electronics.

Although several noise reduction techniques exist for any type of signal in general and for imagery in particular, for this dissertation's working thread proposed objectives, we aim to develop two different but comparable fast image processing denoising techniques with image contour preservation. As referred, they should be fast processing, relatively simple to implemented in nearly all hardware FPGA platforms and should give visible results to justify its implementation.

It is believed that those can be found amongst the simplest time domain constant, as well as adaptative polynomial curves moving average filtering techniques. These can be as fast as real-time and with a minimum possible latency. Although other possible solutions exist, such as frequency domain processing and they will be referred throughout this dissertation, they will not be considered due to their substantial implementation complexity, hence, the processing time required and computational resources used.

For the mentioned considerations two solutions were developed and implemented. The first, a moving average filter with image contour preservation which is described in Chapter 3. The second, an adaptative least squares error Savitzky-Golay filter, as well with image contour preserving, described on Chapter 4.

2.2 Conclusions

These chosen techniques have both advantages and constraints over others that should be referred:

Advantages

- Entirely processing in time domain. Transforming between time and

frequencies domains can bring a bit better results, by selectively filtering noisier bands but are time and computationally too expensive. Besides they require costly floating-point calculations.

- All computations can be done completely in fixed-point arithmetic.
- Low power consumption, hence suitable to be implemented in mobile, low-power image acquisition devices such as video and photo cameras, including those embedded on smartphones.
- Very low latency. Results can be obtained at a speed of one clock, thus to the level of pixel-in this clock, pixel-out next clock.
- Quick processing setup, including hardware optimized for parallel data processing. Few, if any, configuration parameters are required to activate/deactivate the denoising FPGA Intellectual Property (IP) module. For processing parallelization, the developed algorithm should create several channels.
- Selection of the averaging or the adaptative filter size for both solutions can give better results up to a certain point. A tradeoff should be considered.
- It probably does a better job in preserving the images' shapes than their frequency domain counterparts, although this was not tested. However, it is well known that frequency domain processing is done block after block of pixels, normally 8×8 or 16×16 and the final image "stitching" result has a somehow mosaicking effect.
- The adaptative polynomial curves such as the second algorithm proposed here, the Savitzky-Golay filter, use the least squares errors and is a bit more complex than simple moving average filters but better preserves high frequency signals.

Disadvantages

- Both proposed algorithms, the moving average and the Savitzky-Golay filters behave poorly in the frequency domain. This means that the secondary lobes are very frequency leaky. We want to filter the frequencies on the main lobe and cut completely all the others. Undesirably, this is a characteristic which all time domain filters possess. Though, there is no such perfect physical filter, even on frequency domain.
- The adaptative polynomial curves such as the second algorithm

proposed here, the Savitzky-Golay filter, use the least squares errors and are a bit more complex than simple moving average filters. However, as advantage, it better preserves high frequency signals.

- For Savitzky-Golay filter, due to its polynomial and hence multiple derivative nature, for each new derivative, additional information not immediately seen in the data being smoothed might be revealed. Depending on the case to which a certain derivative degree is to be applied on, it can have a pleasant or otherwise an undesirable image effect.

- Since certain operations, in frequency domain, are simpler than in time domain, i. e., a convolution in time domain is done as a multiplication on frequency domain, it is not of common agreement that time domain filtering will be always simpler than frequency domain filtering [9], [29].

3. *Image Denoise FPGA Implementation using a Moving Average Filter with Contour Detection*

This chapter is the first of two papers which resulted from this master dissertation work. It is an image contour detection technique using a Moving Average Filter placed here as a chapter belonging to this dissertation.

The paper was presented on the International Conference on Biomedical Engineering and Applications ICBEA held in Funchal - July 9th to 12th, 2018, jointly organized by institute of Knowledge and Development, the University of Madeira and the Madeira Interactive Technologies Institute.

It was published by IEEE, under the International Standard Book Number (ISBN) 978-1-5386-8058-2/18/$31.00 ©2018 IEEE

Ricardo Jardim, F. Morgado-Dias
University of Madeira and Madeira Interactive Technologies Institute, 9000-390 Funchal, Madeira, Portugal

rjjardim@gmail.com** , **morgado@uma.pt

***Abstract*—A Moving Average Filter should be among the first filters, if not the first, one should consider when speed, good precision and low to medium hardware resources are what is required for implementing image noise reduction on a physical device. The technique is fairly simple to implement, but if taken a straightforward approach, the result will be a blurry image which is undesirable. Since the human eye is most sensitive to the sharp tone transition, we introduced a contour detection technique. Because the inherently lack of floating-point representation, except for the most modern FPGA's versions, all the calculations were done in fixed point. Also, images used were scaled up to $10x10^3$ colour levels (a bit higher than 2^{13} bits). Real results of double precision 64-bit were achieved through MATLAB and then compared to a Zynq-7000 FPGA using 64-bit fixed point calculations. The error difference obtained between both implementations was $1x10^{-4}$.**

Keywords — color scaling denoising function, DSP, fixed-point arithmetic, FPGA, image contour/edge detection, moving average filter, noise reduction, signal processing.

3.1. Introduction

Noise reduction is not a trivial choice process without considering trade-offs between the quick, simple and less than stellar results to the more complex, time consuming and satisfying ones. In cases where both speed and quality

cannot be attained, prioritization, almost always, is given to the speed over quality and this is especially true on critical real-time scenarios. Although active research has been done from two to three decades now with quite good results, noise reduction techniques, particularly done on images, are psychovisual subjective processes which might not always have a direct correlation with quantifiable by parameters such as Mean Squared Error (MSE), Signal-to-Noise Ratio (SNR) and Local or Global Similarity. Besides, due to time constraints, it's almost impossible to develop completely scalable solutions to take advantage of the vast variety of Digital Signal Processors (DSP) and/or Central Processing Units (CPU) processing hardware, namely Field Programmable Gate Arrays (FPGA) as is the case of this work's proposal.

Satisfactory results are, arguably, neck to neck with complex processing, which might evolve bridging between different physical domains, as is the case of the use of the transforms between time/space and frequency domains, trigonometric functions implementation, and intensive numbering operation. There are obviously solutions, that pass, for example, through the use of look-up tables, but acceptable precision cannot be attained if floating point is not considered, besides, real time results might be conditioned.

We aim to implement a solution for noise reduction on an FPGA, which has advantages, over a CPU*:

- Optimize hardware for parallel data processing;
- Reduced dimensions and hence appropriate for mobility;
- Low energy consumption.

On the other hand, it lacks, in most cases, dedicated hardware for precision calculations, such as floating point.

If, however, simplicity and few utilization resources are desired yet with good results comparable to floating-point, it is possible to implement a fixed-point based solution through a moving average filter, which is certainly to be one of the simplest filters for noise reduction.

This type of filter is easy both to understand and to implement, and for better visual acuity it can be paired with a sharp colour transition detection technique while maintaining the smoothness on low gradient colour changes. This is convenient because, what catches immediately the human eye, when looking at an image, is the sharp contours. The moving average filter can be applied to any signal source acquired either in the time or space domains. It can come from an audio, video, imaging or from any electromagnetic nature. In this work, the source is stored images, hence the processing is done in the space domain, but can also be easily applied to real time image acquisition, in time

* Although the same FPGA solution was developed to run on MATLAB, hence, on a CPU (see the C and M codes in Appendix) it should not be considered a fair comparison due to the platforms' different processing conditions (FPGA at 200 MHz with parallel dedicated processing, while CPU at 2.4 GHz with serial processing through MATLAB)

domain.

An idiosyncrasy of the moving average filter is that while it can generate quite good results in the time and spatial domains, it has, however, a very poor degree of controllability in the frequency domain, which is to be considered as the worst filter to control in this domain, namely, and often useful, a poor flexibility in band separation [27]. Through this dissertation, we will refer to the frequency domain, as a comparison reference with the time domain.

MATLAB code was developed to generate all the graphics. The rest of this dissertation will be organized as follows. Section 3.2 reviews the theoretical fundaments, how noise can be generated on an image, what parameters are used to quantify noisy images, what is a moving average filter and how does it work, its behavior in the frequency domain, what results can be obtained with filter autoconvolution and finally the proposed technique of contour detection. Section 3.3 refers to the methodology and the used equipment, its configuration and the testing scenario. Section 3.4 presents the results and finally, Section 3.5 drives the conclusions.

3.2. Moving Average Filter

3.2.1 Noise Generation on an Image

Image sensor photon detection leading to a pixel generation is subject to some perturbations which is a result of both the manufacture process, i. e., the non-homogenic doping process, in the whole surface of the semiconductor material and the light conditions [19]. These perturbations might lead to a perceptible granularity in the image, that is, a tone variability within a similar tone area known as noise. The aforementioned process is of a random nature, and it has been verified to exhibit a probabilistic Gaussian normal behavior.

3.2.2 Quantifiable Used Parameters

Given two versions of the same image, one contaminated with Additive White Gaussian Noise (AWGN) being the other, the denoised processed version, it is possible to compare both versions using the MSE, SNR and the Peak SNR (PSNR) parameters, which are global values, i. e., unique values that quantifies the whole set of pixel differences between 2 images. Comparison between pixels is made on the same coordinate of each 2 images. Structural Similarity Index (SSIM) is yet another parameter we use both as local and global. Local values are made on a per-pixel comparison basis, resulting in an image generated with the same dimensions as the sources, containing the pixel differences between the 2 source images.

MSE can be calculated using Equation 3.1:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2 \quad (3.1)$$

where $\hat{Y}$ is the predictor of the reference image and Y is the noisy image.

The SNR can be calculated by Equation 3.2:

$$SNR = 10\ log_{10}\left(\frac{Y^2}{MSE}\right) \quad (3.2)$$

The PSNR can be calculated using Equation 3.3:

$$PSNR = 10\ log_{10}\left(\frac{maxPR^2}{MSE}\right) \quad (3.3)$$

where $maxPR$ is the image maximum pixel resolution.

3.2.3 Moving Average Filter

It is possible to express the moving average filter using Equation 3.4:

$$y_i = \frac{1}{N}\sum_{j=0}^{N-1} x_{i+j} \quad (3.4)$$

where x is the input pixel, y the output pixel, N the number of counting pixels for the average, i the index of the pixel being averaging and j the offset index of the pixel relative to i contributing to the average of the output pixel.

This can be best understood as a step function with an amplitude of $\frac{1}{N}$ being convoluted with the signal of interest. It is easy to verify that the larger the filtering window, N, the smoother will be the output signal, following the square root law, as it can be seen in Figure 3.1. It is an optimal solution both for solid and for low gradient colour transitions but otherwise for sharp colour changes, resulting in blurry image. The step function slope limit, for the left and the right sides is $\frac{N}{2}$ and $-\frac{N}{2}$, respectively.

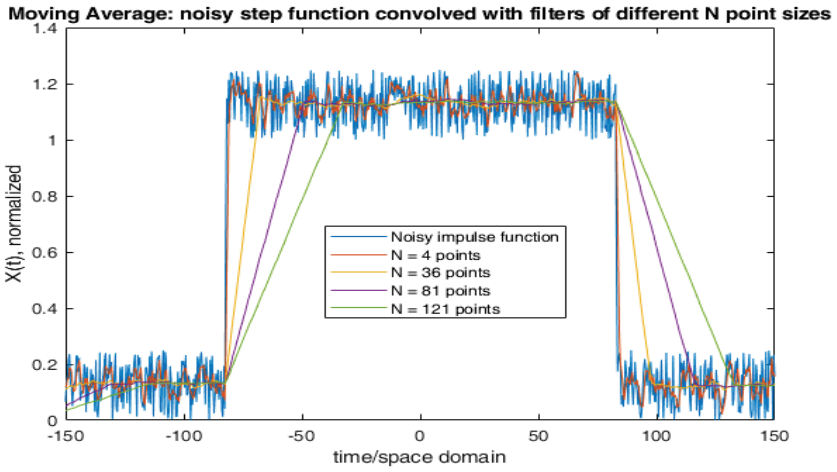

Figure 3.1 - Moving average filters of 4, 36, 81 and 121 points applied to a rectangular step function with random noise.

3.2.4 Frequency Domain

Complementarily, for better understanding with what we are dealing, the frequency response of a moving average filter's kernel seen in time domain as a rectangular pulse, can be given by its Fourier Transform $H(\omega)$:

$$h_H\left(t+\frac{N}{2}\right)-h_H\left(t-\frac{N}{2}\right) \quad \overset{\mathcal{F}}{\Leftrightarrow} \quad H(\omega)=\frac{sin(N\pi\omega)}{N\pi\omega} \tag{3.5}$$

Image acquisition on image sensors is done through channels in which the unavoidable added noise follows a normal Gaussian distribution curve with a zero-mean occurrence probability throughout the entire signal spectrum. However, when the signal is transformed into the frequency domain, the bulk of the signal energy is concentrated at the lowest frequencies shown by the main lobes, in Figure 3.2.

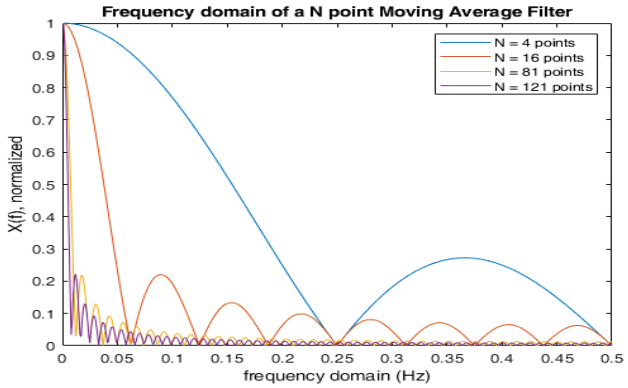

Figure 3.2 - Frequency response of 4 moving average filters.

This means that the noise added to the lower frequencies of the signal becomes diluted on it, hence turning the noise unnoticeable, a good feature. On other side, where the noise becomes noticeable is at higher frequencies. Noise elimination techniques in frequency make extensive use of this property at expense of processing power.

In time domain, the moving average is an optimal filter kernel because it symmetrically reduces the variance of a Gaussian shape with zero-mean random noise distributed all over the signal. It is, however, possible to choose where to eliminate specific noise contaminated frequencies by shortening or enlarging the number of points contributing for the averaging, bearing in mind that some of the signal information will also be eliminated along with the noise. In the frequency domain, sequenced high secondary filter lobes turn it poorly controllable due to the soft roll-off. It is desirable a sharp stopband decay for all filter lengths, N.

If all attempts fail to explain how the moving average filter works, one can simply resume referring to a basic principle of the Vector Signal: the larger the signal pulse in the time domain, in this case, a larger N, the narrower the range of the band in the frequency domain, which means a reduced noise variance

obtained.

3.2.5 Multiple Autoconvolution

Different results can be obtained if one or more convolution passes of the filter over itself are made.

The convolution can be given by Equation 3.6:

$$h_1 \star h_2(t) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} h_{H_1}(\tau) \cdot h_{H_2}(t-\tau)\, d\tau \tag{3.6}$$

where $h_1(t)$ and $h_2(t)$ were given by the left-hand side of $\mathcal{F}$ (the time-domain side), in Equation 3.5.

Figure 3.3 shows several curves, which represent sets of autoconvolution operations of the same filter.

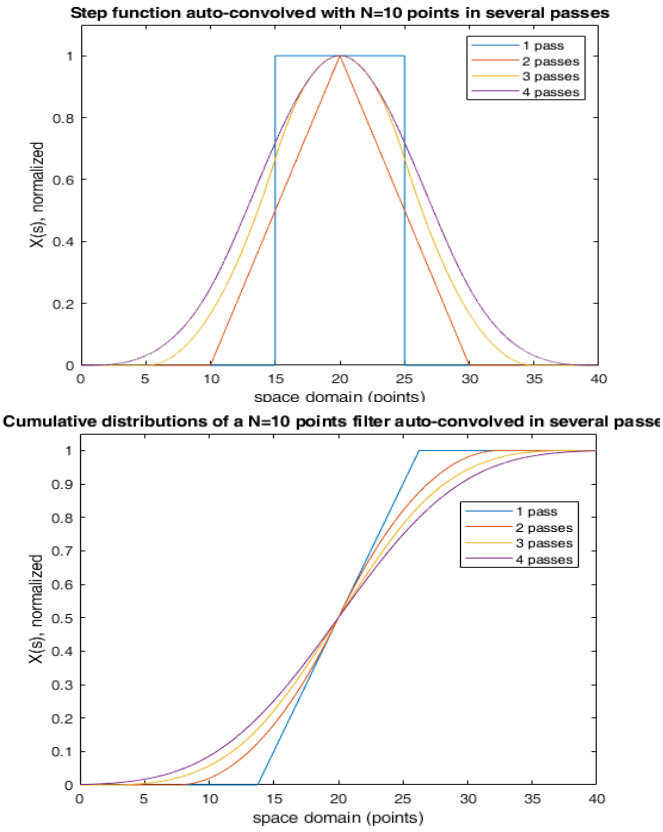

Figure 3.3 - – Autoconvolution. **Above**: shape curves from 1 to 4 passes. **Below**: cumulative distributions from the corresponding curves

As the number of passes increase, the filter shape changes from a step function to a Gaussian function.

The result of successive autoconvolutions, as expected, increases the noise attenuation but cumulatively decreases signal detail. Also, filter amplitudes increase as more passes are made due to the increasing integrated area, although this is not shown in Figure 3.3 because all curves have been

normalized.

While in one pass filtering the weight of the neighbor pixels under it is the same, on a multiple pass filtering, the averaging weight takes the shape of the curve being convoluted with the signal, higher on the center of the filter. With 2 passes the filter shape becomes triangular, and from 3 passes up, the curve gets a non-linear weight Gaussian distribution shape like.

The cumulative distribution below graphic in Figure 3.3, shows how the filter becomes less sharp inversely proportional to the number of passes, a non-desirable characteristic due to the non-linearities it introduces but on the other hand, in the frequency domain, as can be seen in Figure 3.4, the secondary lobes drastically drop because each new result is a multiplicative operation between the previous pass and a new function, as given by the right-hand side of $\mathcal{F}$ (the frequency domain-side), Equation 3.5. This is a welcome feature turning the Moving Average Filter a tradeoff choice between specific frequencies vs. a whole frequency band.

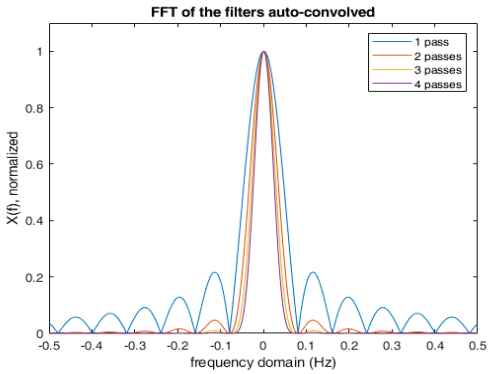

Figure 3.4 - Frequency domain multipass autoconvolution.

3.2.6 Contour Detection

The algorithm for implementing the Contour Detection can be developed starting with the Moving Average Filter given in Equation 3.4. If:

$$\bar{M} = y \tag{3.7}$$

being $\bar{M}$ the filtered averaged pixels, that is, to the pixels convolved with the filter, the decision for the pixel, p, being analyzed, to contribute to the average is decided if

$$p \subset [\bar{M} - \sigma \dots \bar{M} + \sigma] \tag{3.8}$$

in which

$$\sigma = \sqrt{\frac{\bar{M}}{N}} \tag{3.9}$$

otherwise, p restarts a new averaging

Visually, the difference between Moving Average Filters, one without (simple) and another with Contour Detection can be seen below in Figure 3.5. If a whole width horizontal line of pixels is randomly chosen from the top image (red) and moving average operations are made with several filter lengths, N, the result will be the middle image pixel profiles. Higher values of the curves correspond to lighter grey shades and the lower values to darker shades. In the middle is graphic without contour detection, if N is big enough, we end up losing the image sharpness since everything gets averaged. Conversely, as shown in the lower graphic, the steeply tone changes are pretty much preserved and only the soft transitions variability is smoothed out.

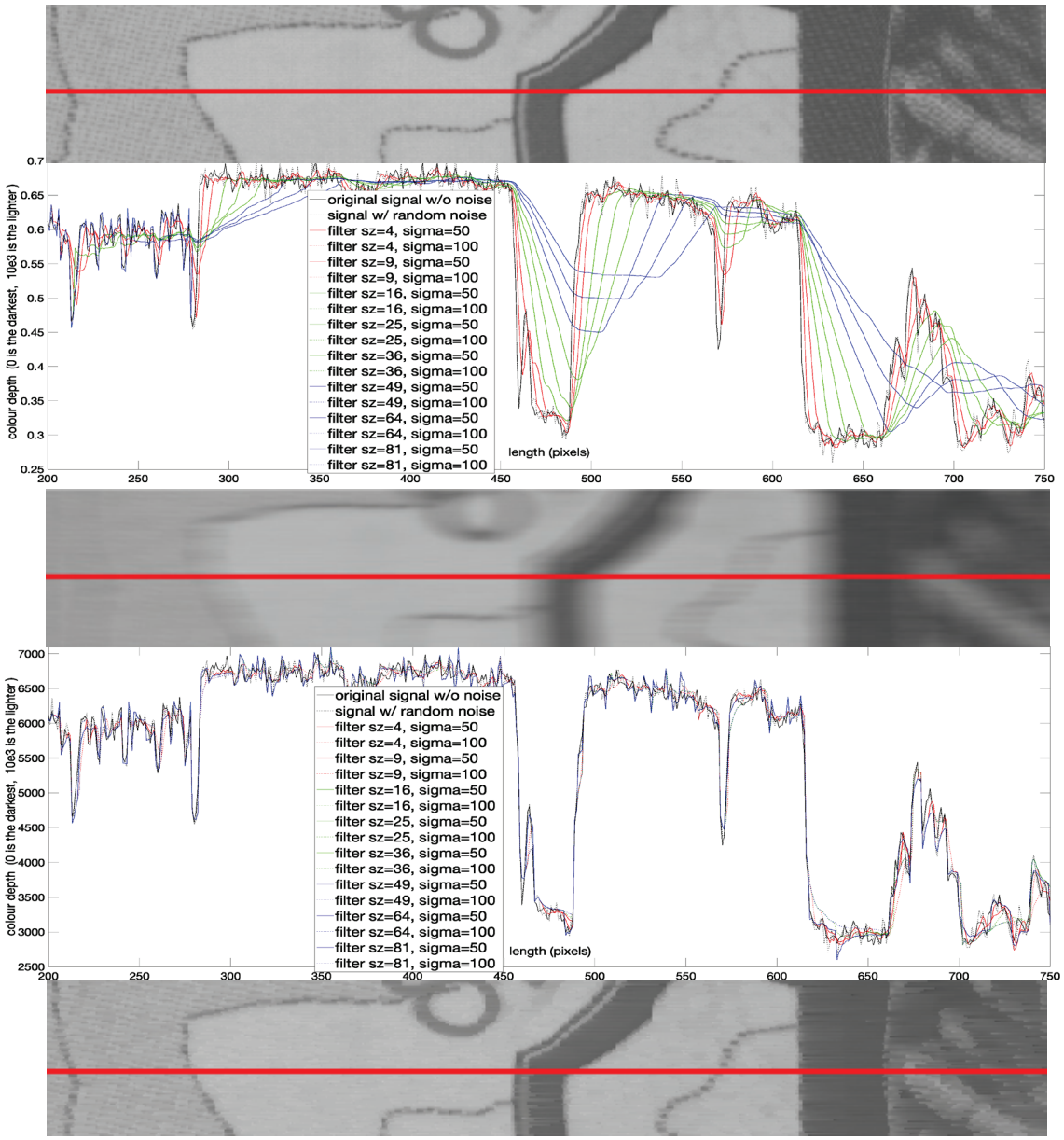

Figure 3.5 - Moving Average Filter (from top to bottom): original image; without contour detection (graphic and result); with contour detection (graphic and result).

3.3. Methodology and Used Equipment

An image acquired with an image sensor in proper light conditions, to minimize photon shot noise, was chosen and referenced as original image. The image's pixel depth is a 2^{13}-bit fixed-point (8192 tones). In MATLAB, the image was converted to normalized floating-point range $[0..1]$ with the purpose to add several AWGN variances noise, according to the Section 3.5. This is needed because almost all MATLAB's Toolboxes APIs work in floating-point notation. The noisy images to be processed in the FPGA were converted back to its original bit depth fixed-point while the ones to be processed on MATLAB remained in floating point.

For the real floating-point calculations, MATLAB scripts were used. For fixed-point calculations it was used a ZedBoard Zynq-7000 All Programmable SoC XC7Z020-CLG484-1 FPGA board, shown in Figure 3.6, along with programming environments Xilinx Vivado versions 2017.4 and 2018.1. The language for the FPGA programming was Very-High Speed Integrated Circuit Hardware Descriptive Language (VHDL) with some system libraries in Verilog. The images to be processed in the board were uploaded onto the same SD card which loads its OS.

Since the original image has a pixel depth of 2^{13}-bit, with a Factor Wheel Conversion (FWC) pixel (photon to electrons conversion) of 10×10^3 (as stated, slightly above 2^{13}, it should be level up, for consistency, to the nearest multiple of 8 and hence to 2^{16}-bit. More, because some pixel's calculations are same register cumulative, this easily overflows 32-bit registers if these were chosen, so 64-bit registers were implemented. This, not only, does not increase roundoff errors because they are done, as will be discussed, to the nearest zeroth decimal floor, but also makes room for calculations with higher bit-depth images without having to redesign the registers.

The hybrid capabilities of Zynq chip, working as a full integrated CPU+OS computer system with an FPGA easily accessed through an AMBA AXI3/AXI4 interface bus facilitate the communication process. From the board's computing side, images data pixels are loaded and sent in a column-by-column sequence to the FPGA to be processed, which are then collected, assembled and stored in the SD card as new images. It is also possible to use the board's Ethernet to exchange images between the working computer and the ZedBoard avoiding constant SD card's swap, however, network instability issues voided this option.

In the working computer, measurement parameters described in Section 3.2.2 were used using the MATLAB's Image Processing Toolbox. Each image, either fixed- or floating-point, was compared with the original in its respective

scale. These original-processed image pairs were used as MATLAB's measurement functions input parameters. The functions return a result which is a floating-point double value. Visual comparisons were also made.

In Section 3.4, each image has a header defining the attained results which were transcribed to Figure 3.7 through Figure 3.15 captions.

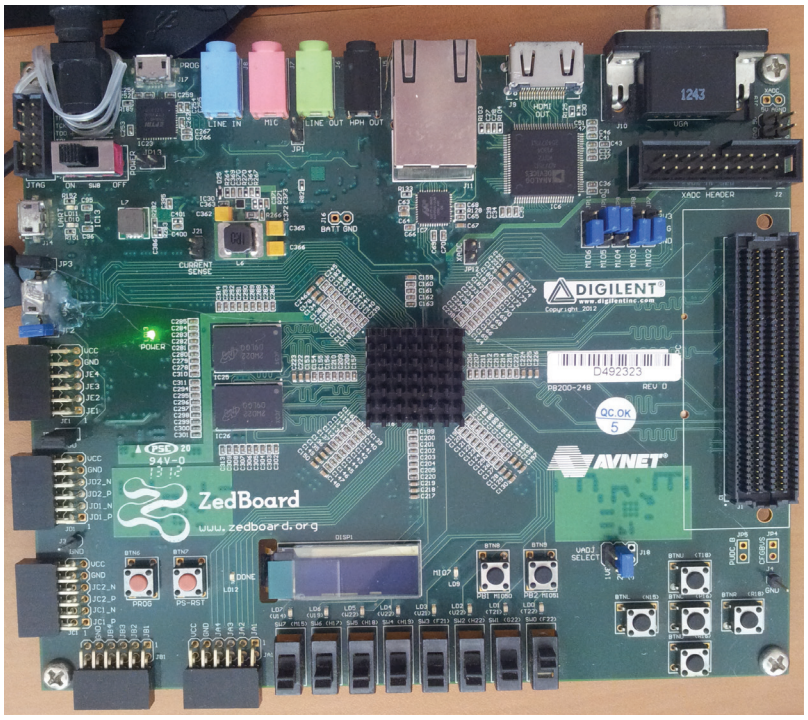

Figure 3.6 - Hardware platform FPGA ZedBoard Zynq-7000 used for the tests.

In Appendix 3.A, Figure 3.16 shows a timing diagram of the FPGA implemented solution for a filter length $N = 4$. Despite the volume of signals and variables, most starting with an "o_" which are internal registers and control signals, the relevant information is pointed out and is described here. Most important signals are the highlighted DataIn and DataOut. The first blue marker at 160 ps delimits the beginning of data sequence to be processed, DataIn, which starts with a four-value header: N, SMF, bit depth and the number of passes (how many loops this data sequence is to be processed). So, it can be seen for this header that $N = 4$, $SMF = 50$, bit depth $= 8192\ (2^{13})$ and $NumPasses = 1$. Right after, as pointed by the second blue marker, data starts to flow in. It can then also be seen that the processed data is coming out of the module, carried by DataOut, in just one complete clock cycle, as shows the yellow marker. Another signal of interest is the "o_rst_clk", below DataIn, which resets the pixel counter every time a colour sharp transition is detected. Appendix 3.B shows utilization of several FPGA's resources.

3.4. Results

Figure 3.7 is composed of a pair of images in which the top is the original version and the bottom is the noise contaminated version with 0.01 AWGN standard deviation. Figure.3.8 through Figure 3.15 show pairs of images processed using filter lengths $N = (4, 9, 16 \text{ and } 25)$, chosen according to the Equation 3.9, which says that the noise reduction factor is given by the squared root of the filter's length, N, so we can expect noise reduction factors of 2, 3, 4 and 5 folds. For filter lengths above $N = 25$, the image quality starts to degrade due to the fact that too much image detail is removed along with the noise.

For each filter length value, 2 multiplicative factors of σ (SMF), were chosen, 50 and 100. It is known that a Gaussian curve is almost entirely contained between -3σ and 3σ, so the question is, why to set σ equal to 50 and 100? Our experiments show that σ factors lower than these simply present negligible to none filtering results, hence the use of this inflating factor. It can be seen that there is a denoising increasing factor according to the increase of N and SMF.

The Table 3.1, in the Appendix 3.C, shows the quantifiable results, based on the already referred parameters MSE, SNR, PeakSNR and Global Similarity, associated to these generated images.

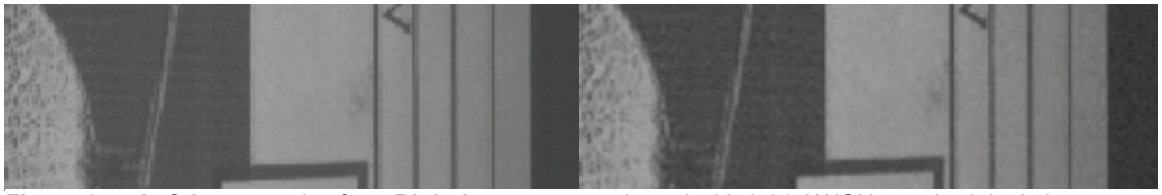

Figure 3.7 - **Left image:** noise free. **Right image:** contaminated with 0.01 AWGN standard deviation (σ).

Figure.3.8 - **Left image:** Floating-point version, **Right image:** Fixed-point version. Number of Passes=1, Filter size N=4, SMF=50.

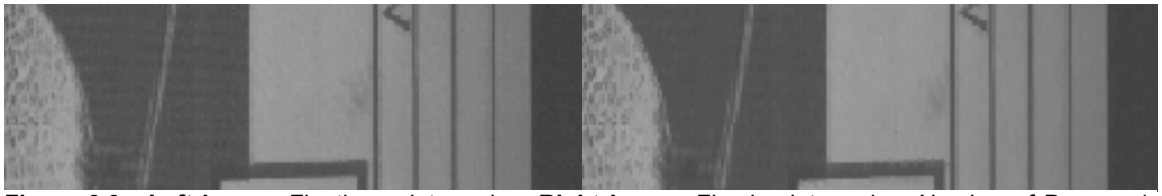

Figure 3.9 - **Left image:** Floating-point version, **Right image:** Fixed-point version. Number of Passes=1, Filter size N=4, SMF=100.

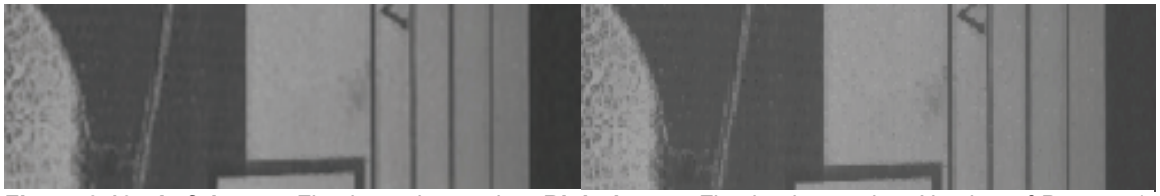

Figure 3.10 - **Left image:** Floating-point version, **Right image:** Fixed-point version. Number of Passes=1, Filter size N=9, SMF=50.

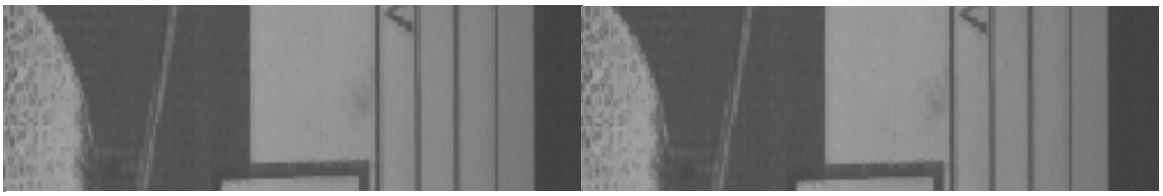

Figure 3.11 - **Left image:** Floating-point version, **Right image:** Fixed-point version. Number of Passes=1, Filter size *N*=9, SMF=100.

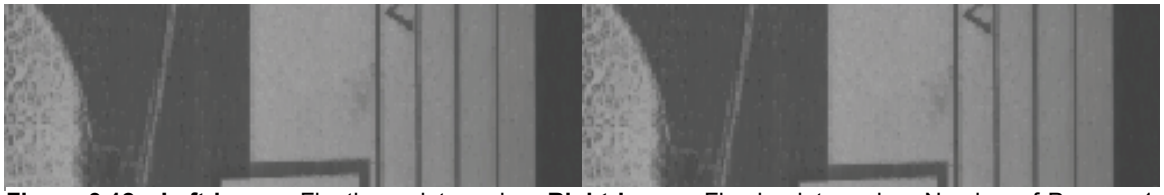

Figure 3.12 - **Left image:** Floating-point version, **Right image:** Fixed-point version. Number of Passes=1, Filter size *N*=16, SMF=50.

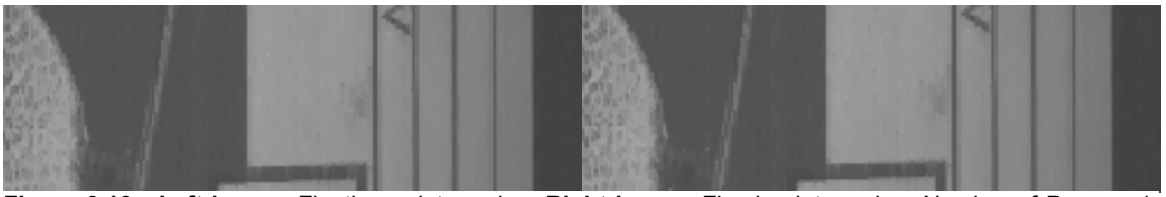

Figure 3.13 - **Left image:** Floating-point version, **Right image:** Fixed-point version. Number of Passes=1, Filter size *N*=16, SMF=100.

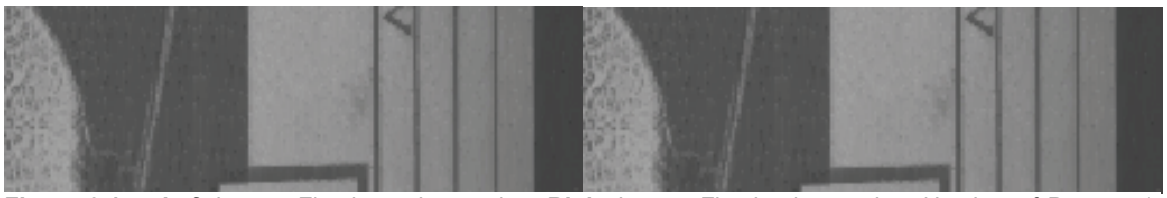

Figure 3.14 - **Left** image: Floating-point version, **Right** image: Fixed-point version. Number of Passes=1, Filter size N=25, SMF=50.

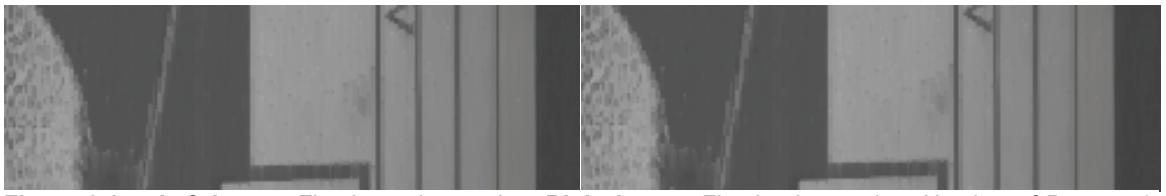

Figure 3.15 - **Left image:** Floating-point version, **Right image:** Fixed-point version. Number of Passes=1, Filter size *N*=25, SMF=100.

3.5. Conclusions

Table 3.1, in Appendix 3.C, show the test results obtained with added noise variances of $\sigma^2 = 1 \times 10^{-4}$ and $\sigma^2 = 1 \times 10^{-3}$ to an image that we assume to be noise free. Main focus was given to $\sigma^2 = 1 \times 10^{-4}$ because it shows a good tradeoff between a mild perceptible added noise to be processed and the presumed noise free image. The $\sigma^2 = 1 \times 10^{-3}$ was chosen for comparison because of its significative noise perceptibility.

A MATLAB pseudorandom generator function was used to add the indicated noise variances to the image. The rightmost column, *Change*, shows the differences, in percentage, including the SNR and PeakSNR logarithmic values converted to decimal, between the MATLAB's floating-point version and

the FPGA fixed-point calculations, taking the foremost as reference. Floating-point calculations were all done in the value's range between $[0..1]$, because it holds the most precision scale. According to MathWorks, MATLAB maximum rounding error, a machine epsilon, is 2^{-53} in a scale of values between $2n$ and $2n+1$, in this case $n=0$. This is the reason to consider it as the reference, while fixed-point calculations were done, as previously mentioned, in the range $[0..10\times10^{3}]$. Comparing both, as expected, all $\sigma^2=1\times10^{-4}$ tests presents better results than $\sigma^2=1\times10^{-3}$, with the highest values being $N=9$ and $SMF=50$, while the lowest being $N=16$ and $SMF=100$. SNR and PeakSNR depend inversely on MSE as given by Equations 3.1, 3.2 and 3.3, and Global Similarity follows this trend as well. The less the MSE error, the better the SNR and hence the better the Similarity between the original and the processed images.

Relatively to $\sigma^2=1\times10^{-4}$, it should be noted that better values are achieved when values of $SMF=50$. Also, from $N=4$ to $N=25$ the results degrade progressively among the same value of SMF. The exception to this is when $N=25$, which achieves better results for fixed-point in both SMF's values. Image's results reveal that progressive higher quality is achieved at N=16 and $SMF=100$. For $N=25$, images show some adjacent colour fusion, resembling patches, so beginning to lose some quality. SNR's magnitude orders of $31\,dB$ and PeakSNR's of $37\,dB$ for $SMF's=50$ seem reasonable for this value of noise variance ($\sigma^2=1\times10^{-4}$) while for $SMF=100$, lower SNR's of 28 or $29\,dB$ and PeakSNR's of 35 dB are more expected, given that a higher SMF increases the smoothness over a lower SMF for the same N, and so, visually, it seems to qualitatively reduce further the noise. However, as revealed numerically, image degrades because information details are, as well, processed along with the noise. We have made an additional test for comparison, which consisted in a simple moving average image noise reduction and we got SNR values between 26 and $19\,dB$, hence significantly lower than the ones of our interest. It is also noticeable that floating-point results are consistent with fixed-point results. The differences between both are minor but suffice to conclude that the floating-point, due to higher precision, generates better results. In relation to $\sigma^2=1\times10^{-3}$, the best values are obtained when $N=4$ and $SMF=100$. There is, however, a significant difference: while in $\sigma^2=1\times10^{-4}$, SNR results are better in SMF=50 than in SMF=100 (differences between 2 or $3\,dB$), in $\sigma^2=1\times10^{-3}$ they are practically the same around $SNR=24dB$. Given that $\sigma^2=1\times10^{-3}$ generates a perceptible noisier image, this is an acceptable result.

The most meaningful result here is that it is possible to make calculations in a simple fixed-point notation without any visible difference compared to the more computationally expensive floating-point counterpart, once a relative high

pixel bit depth is chosen. Our tests were made with images containing pixels with 10×10^3 colour tones, i.e., a little over $2^{13} = 8192$ shades. This further reduces roundoff errors to a level of precision of $\frac{1}{10 \times 10^3} = 1 \times 10^{-4}$, due to the fact that calculations made in fixed-point always ditch the rational part by flooring down the result to the immediate lower integer available. Additionally, attention must be given to the whole mathematical operation, with the purpose to spot where divisions and square roots occur and exert some effort to reduce it to a maximum of one square root and a maximum of one division, explained by the fact that round offs only occur on these math operations.

There is yet another feature that needs to be treated properly. A noise contaminated image is more noticeable at darker colour or shades than lighter ones and this deserves special care. A non-linear smoothing filtering function must be adjusted and multiplied by σ, letting darker colour shades have more filtering tolerance than the lighter ones.

A quick note on the results: the sparse difference between the measurement parameters does not quite really translates the visual perception of the denoising processing results. Furthermore, the different filter sizes and the multiplicative sigma factor can make a huge contribution when the image has large solid patch shades to process.

This work shows promising results on low power, low complexity, high portability, FPGAs utilization for real-time signal processing.

For future works and already working on it, we aim to make use of a more challenging approach in frequency domain, including Discrete Cosines (DCT), Fourier and Wavelet Transforms. This was one also of the reasons to include it as a reference on Section 3.2.

3.6 Acknowledgments

Acknowledgments to the Portuguese Foundation for Science and Technology for their support through Projeto Estratégico LA 9 - UID/EEA/50009/2013.

3.A. Timing diagram

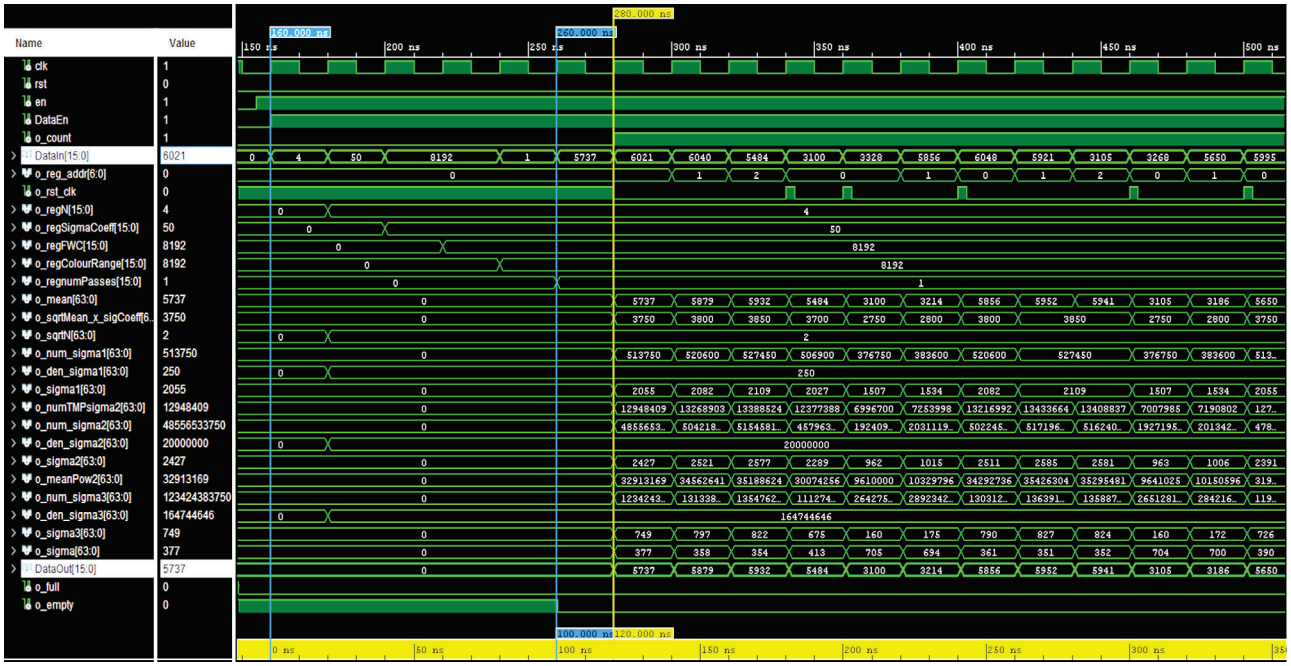

Figure 3.16 – Timing diagram referenced in Section 3.3.

3.B. Information of the FPGA's implemented logic

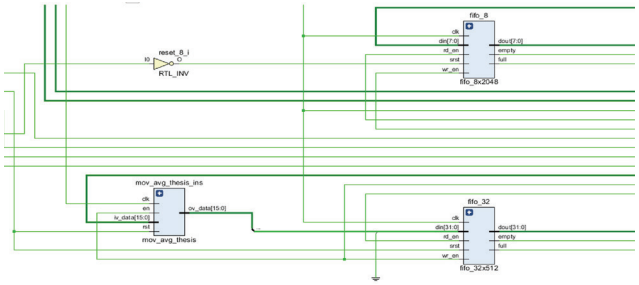

Figure 3.17 - Integrated 'mov_avg' IP block schematic.

Resource	Utilization	Available	Utilization %
LUT	15363	53200	28.88
LUTRAM	335	17400	1.93
FF	6851	106400	6.44
BRAM	4.50	140	3.21
DSP	14	220	6.36
IO	85	200	42.50
BUFG	3	32	9.38
PLL	1	4	25.00

Figure 3.18 - FPGA resources utilization.

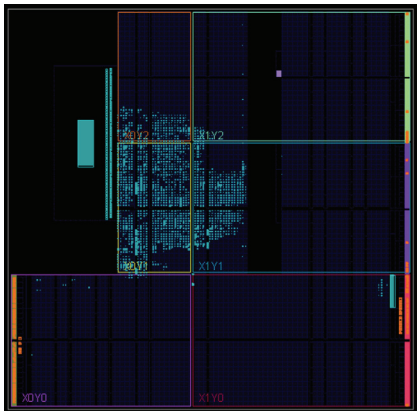

Figure 3.20 - FPGA logic layout.

Pulse Width

Worst Pulse Width Slack (WPWS):	3.000 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	5083

Figure 3.19 - Pulse width.

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.881 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**46.7°C**
Thermal Margin:	38.3°C (3.2 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

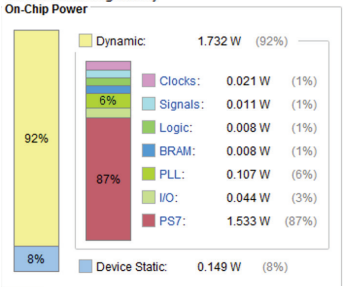

Figure 3.21 - FPGA power consumption.

3.C. Results table

Table 3.1 - Results relative to figures 3.8 to 3.15 based on the aforementioned parameters MSE, SNR, PeakSNR, Global Similarity and Noise

Figure	Parameter	Floating Point (Matlab)		Fixed Point (FPGA)		Change	
	Noise added (σ^2)	$1x10^{-4}$	$1x10^{-3}$	$1x10^{-4}$	$1x10^{-3}$	$1x10^{-4}$ (%)	$1x10^{-3}$ (%)
8	**Filter length (N)**	4					
	Sigma multiplicative factor	50					
	MSE	0.00016072	0.000846545	0.000160746	0.000846592	+0.016177	+0.005552
	SNR (dB)	31.64	24.43	31.64	24.43	-0.015887	-0.005526
	PeakSNR (dB)	37.94	30.72	37.94	30.72	-0.016117	-0.005526
	Global Similarity (%)	92.05	67.61	92.05	67.60	-0.000217	-0.000444
9	**Filter length (N)**	4					
	Sigma multiplicative factor	100					
	MSE	0.000254682	0.000667664	0.000254709	0.000667742	+0.010601	+0.011683
	SNR (dB)	29.64	25.46	29.64	25.46	-0.010131	-0.011512
	PeakSNR (dB)	35.94	31.75	35.94	31.75	-0.010361	-0.011742
	Global Similarity (%)	89.72	73.34	89.72	73.35	-0.000669	+0.000682
10	**Filter length (N)**	9					
	Sigma multiplicative factor	50					
	MSE	0.000160537	0.000906685	0.000160546	0.000906783	+0.005606	+0.010809
	SNR (dB)	31.65	24.13	31.65	24.13	-0.005987	-0.010822
	PeakSNR (dB)	37.94	30.43	37.94	30.42	-0.005987	-0.010822
	Global Similarity (%)	92.17	65.81	92.17	65.81	-0.000217	-0.002127
11	**Filter length (N)**	9					
	Sigma multiplicative factor	100					
	MSE	0.000296587	0.000736503	0.000296641	0.000736601	+0.018207	+0.013306
	SNR (dB)	28.98	25.03	28.98	25.03	-0.018189	-0.013124
	PeakSNR (dB)	35.28	31.33	35.28	31.33	-0.017959	-0.013354
	Global Similarity (%)	88.37	72.27	88.37	72.27	-0.002263	-0.000138
12	**Filter length (N)**	16					
	Sigma multiplicative factor	50					
	MSE	0.00016108	0.000910483	0.000161083	0.000910579	+0.001862	+0.010544
	SNR (dB)	31.63	24.11	31.63	24.11	-0.002533	-0.010361
	PeakSNR (dB)	37.93	30.41	37.93	30.41	-0.002303	-0.010591
	Global Similarity (%)	92.15	65.69	92.15	65.68	+0.000109	-0.002436
13	**Filter length (N)**	16					
	Sigma multiplicative factor	100					
	MSE	0.000297388	0.000784837	0.000297413	0.000784952	+0.008407	+0.014653
	SNR (dB)	28.97	24.75	28.97	24.75	-0.008519	-0.014735
	PeakSNR (dB)	35.27	31.05	35.27	31.05	-0.008289	-0.014735
	Global Similarity (%)	88.47	70.37	88.47	70.37	+0.000339	-0.001279
14	**Filter length (N)**	25					
	Sigma multiplicative factor	50					
	MSE	0.000161714	0.000910427	0.000161711	0.000910529	-0.001855	+0.011204
	SNR (dB)	31.62	24.11	31.62	24.11	+0.001842	-0.011282
	PeakSNR (dB)	37.91	30.41	37.91	30.41	+0.001612	-0.011282
	Global Similarity (%)	92.10	65.69	92.11	65.69	+0.000760	-0.002740
15	**Filter length (N)**	25					
	Sigma multiplicative factor	100					
	MSE	0.000293998	0.000799112	0.000293994	0.000799207	-0.001361	+0.011888
	SNR (dB)	29.02	24.68	29.02	24.68	+0.001612	-0.011742
	PeakSNR (dB)	35.32	30.97	35.32	30.97	+0.001612	-0.011742
	Global Similarity (%)	88.74	69.69	88.74	69.69	-0.000338	+0.001148

4. Savitzky-Golay filtering as Image Noise *Reduction with Sharp Color Reset*

This chapter is the second of two papers which resulted from this master dissertation work. It is a more elaborated filter than the one presented on the previous chapter, a Savitzky-Golay filter with sharp color detection and is placed here as a chapter belonging to this dissertation.

The paper was submitted to the Elsevier – Microprocessors and Microsystems Journals & Books, has been accepted to be published on the Volume 74, in April 2020, with the number 103006. It can be found on the DOI webpage https://doi.org/10.1016/j.micpro.2020.103006.

Ricardo Jardim, F. Morgado-Dias
University of Madeira and Madeira Interactive Technologies Institute, 9000-390 Funchal, Madeira, Portugal

rjjardim@gmail.com** , **morgado@uma.pt

Abstract — Images acquired through photosensors, known to be non-stationary correlated signals, due to its acquisition characteristics, often come attached with uncorrelated, zero-mean, variance σ^2, stationary processes. The latter can be built-up over the former, being sourced from two contributions, one dependently, known as photonic noise, while the other independently, if circuitry generated. Either way, a straightforward approach to reduce this noise would be the use of low-pass filters. We propose an improvement of the Moving Average Filter with a polynomial Savitzky-Golay filter. Based on the obtained results, we believe that, in most cases, this filter can produce better results than the standard Finite Impulse Response (FIR) filters since it does a better job in preserving the high-frequency signals. The challenge is to choose the best tradeoff between the window frame size, the derivative order and polynomial coefficients. The contour (color sharp) transitions are done through filter resetting every time the next pixel value is outside the variance markers of the filter. Since the processing is intended to be done in real time, we want to stick in time domain using fixed-point calculations. MATLAB is used to compare floating-point results with an integer processing platform, an FPGA. Error and Signal-to-Noise Ratio (SNR) improvements were achieved by several orders of magnitude using this new method.

Keywords — Savitzky-Golay filter; noise reduction; polynomial smoothing; least-squares; FPGA; DSP; signal processing; fixed and floating-point arithmetic; color transition detection.

4.1. Introduction

There is no easy way to properly make a choice about what method to apply to reduce the noise out of a signal. It is desirable to be quick and simple but at the same time as efficient as possible. Noise reduction techniques has been a hot topic for 20 to 30 years now and still continues improving, relying the bulk of its development with a strong basis of statistics and optimized with a proper numeric tool. Designing for mobile low power devices, it is required and desirable to make use of techniques that permit quick, almost instantaneous results.

When photon-to-electrons conversion takes place on a photosensor leading to an image generation, the result can be a noisy signal whose magnitude is inversely proportional to the efficiency of the conversion rate. The contained signal be sourced from two ways, as described by Alparone et al. [3]. This conversion was already studied, quite some time now, by Karp and Clark [19] and published in 1970. The work describes the photoelectron counting problem which is classical in noise generation where it is assumed that an electromagnetic signal resulting from this conversion is a deterministic, zero-mean, Gaussian random process which can be expanded by Karhunen-Loéve series of orthogonal functions. The functional similarity is shown by Laguerre polynomials as well as the MAP (maximum a posteriori) and the ML (maximum likelihood) constraints estimation functions.

Argenti et al. in [4] and [5], uses a new approach of point despeckling or smoothing a high variability signal that can produce good results on low gradient transitions while still preserving most of the image's signal energy. However, it is based on undecimated wavelet decomposition, which is better explained by Starck et al. [21]. It requires the use of transforms between the time and the frequencies' domains. It should be reminded that the purpose of this work is to maintain low computational processing resources and also within the realm of the time-domain, so avoiding high resourceful demanding filters.

Sikora [30], uses a relatively low complexity approach based on a shape-adaptive Discreet Cosines which in turn, is based on Gilge's [31] arbitrary shaped images processing algorithm with orthogonal transforms. It is a good algorithm for managing image segments of random shape yet using low computational resources but again, involves transforms between domains, a high demanding computational expense not practical on many of todays' FPGAs integer operational nature.

Block-matching 3D (BM3D) transform, proposed by Chen [32], has an augmented performance for denoising mild noisy images. It works by grouping similar 2D image's fragments into an array, that can be regarded as the addition

of an extra dimension, effectively, sets of 3D blocks from which it derivates its name. It has, however, a low efficiency when the noise increases because, along with it, the similarity mismatching also increases. Besides, this algorithm uses, as well, DCT transforms.

Another proposal of 2D block-grouping into 3D arrays which is based on an enhanced sparse representation is studied by Dabov et al [33]. The author tags Collaborative filtering to the manipulation of these similar 3D arrays, which includes 3D transformation, shrinkage of the signal's spectrum in order to reduce the noise, and inverse 3D transformation. These transforms use discreet cosines hence, another method that gives the power of the spectrum manipulation but increases the complexity and the computational resources.

A general mathematical denoising model, in comparison with other models and their classification, along with a proposal of a NL-means (Non-Local) algorithm is suggested by Buades et al. [34]. NL-means are Euclidean distances between patches. Patches are themselves sets of two pixels centered in the middle of their junction. In his proposal, he tries to preserve the structure of the image, a feature within the context of our previous paper [35] as well as this work's objective, although using other techniques. To achieve this, he makes use of numerical analysis, statistical models, weighted similarities between pixels and patches with certain limits' assumptions. Additionally, it is used anisotropy, a curvature motion technique, but this leads us to the frequencies' domain, which is out of scope of our proposal.

Deledalle [23] proposes a new approach for a known uncorrelated noise model which is an extension of the NL-means [34]. It is proposed a more generalized similarity approach among pixels and patches in which it has dependency with the distribution model of the noise. The selected weights can be refined based on the similarity taken out from the last estimation. This technique improves somehow the SNR, particularly, on SAR (synthetic aperture radar) images but also, in general, with images affected by additive Gaussian and by multiplicative speckle noise models. The author claims improvements on the latter.

Another numerical model in which it is defined color multiscale analysis and graph-based data representation is exploited by Malek [22]. It uses a quadratic Laplacian form to obtain information about the energy distribution as means to draw the new data representation. It also spots the parameters' influence on that distribution. This approach applies a psychovisual technique but does not consider the computational resources used as priority, besides, it makes used of wavelet transforms, which means having to work with floating-point complex numbering.

Right in the beginning of his shape-adaptative DCT paper, Katkovnik

[20], deliberately warns for the retained complexity pretraining to all comparable block DCT algorithms. This is yet, another research that combines shape-adaptative blocking with local polynomial approximation and with anisotropy. It first works by deblocking the image, previously compressed with blocking DCT. It then computes the confidence intervals to find the transform's support shape on a pointing adaptative method. By previously establishing the appropriate threshold values, it then finds the estimation coefficients that will serve to reconstruct the signal's support shape. Furthermore, the shapes' supports normally overlaps in the same pointwise neighborhood, leading to further computations which are required to make weight averaging. It also proposes some restrictions in the luminance-chrominance space in order to increase the accuracy of the image's colors. This adaptative reconstruction procedure enhances the color transitions, meaning it cleans the undesired artefacts, in fact one of the goals of our work's proposal, consequently, the results are visually catching. However, besides using transforms between domains, it is too computationally demanding to be considered within this paper's boundaries.

Given the above researches, seems that all high-quality results belong to the realm of the frequencies' transforms, However, we think there is still room for improvement not necessarily needing to leave and returning the time domain as is this paper's proposal.

Following the processing approach described by Jardim and Morgado-Dias [35], in which an adapted moving average technique, suitable for real time, mobile, low power devices was worked out, we introduce the polynomial flexibility towards noise reduction. It has a benefit to emboss the inflection signal points, and from there, take the appropriate decisions for further processing. A quick introduction of the moving average filtering can be consulted in Smith [27].

In a simple and carefully tweaked way, these techniques permit to obtain a real-time processing without much visual difference relatively to the most complex ones. This may lead to a one-fit-all general rule which can be especially true in the case of the photon dependency nature of the noise generation, given that, by the side of the circuitry independent noise generation, a previous study can be made in order to map and create a model to virtually wipe out the noise generated from this source.

Anyway, for quick, yet good visual results, good candidates' methods for noise estimation which seems to give better results are those that detect and apply a correction that less deviate from the noise free references.

Parameters that can be used to quantify the level of uncorrelated randomness in an image, although not always related with the visual perception are the Mean Squared Error, MSE, a parameter that parabolically magnifies the error the more it deviates from the reference. The ubiquitous Signal-to-Noise

Ratio, SNR, can be used to qualify the signal deterioration, for example, with the help of the former computed MSE. Similarity between two sets of pixels is another parameter that can be used to measure its differences either globally, a unique value which translates the whole difference between the pixels' sets, or locally, a map based on a per-pixel comparison.

Bearing in mind that the aim of this work is to study denoising solutions that target low power consumption, low processing demanding and ready to be viewed images, it is always possible to save resources by using fixed point calculation units such as Field Programmable Gate Arrays, FPGAs, Digital Signal Processors DSPs and the most recent mobile Central Processing Units, CPUs, in spite these last two possess the capability to make floating-point calculations. With these resources, it is tempting to use transforms between time/space and frequency domains. However, making these operations, the computational price to pay in order to obtain better visual results in the frequency domain will rise non-linearly and will end up revealing unfruitful due to the intensive use of the floating-point computations.

For those less familiar with floating point arithmetic, Nascimento, Jardim and Morgado-Dias [36] presented the standard approach defined in IEEE 745, which involves numbering decomposition in signal, exponent and significand or mantissa parts, only after, the operation takes place. While two numbers' addition is done roughly in a straightforward manner, multiplication is done first by adjusting the exponents for addition, then the product of mantissas is computed, and this requires a great load of bit shift manipulation. The result has yet to be coded in the floating-point format. A quicker approach evolves the use of lookup tables which introduce errors of higher magnitude orders.

If the proposed solution is to exploit the advantages of several FPGAs architectures, even if using fixed-point arithmetic at, for example, 64-bit resolution depths which were used in this work, it is possible to keep rounding errors to its minimum. Our tests reveal fixed- to floating-point differences roughly between 1.4×10^{-4} and 1×10^{-3}.

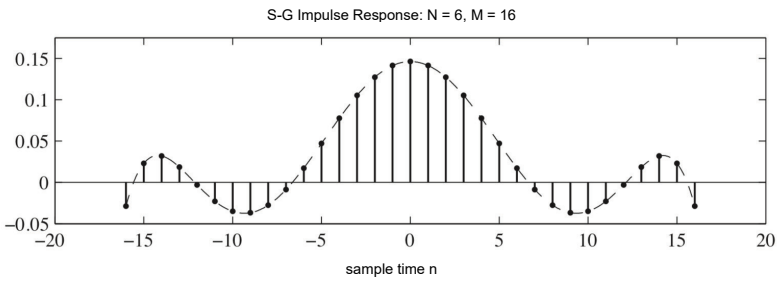

Figure 4.1 - Response of a Savitzky-Golay filter. The curve is the polynomial fit, $\widetilde{\boldsymbol{p}}(\boldsymbol{n})$ **[24]**.

Maintaining the low complexity of the Moving Average filter presented by Jardim and Morgado-Dias [35], we propose a polynomial Savitzky-Golay filter. Contrary to the flat averaging behavior of the former, the S-G filter is built with a

$$\tilde{p}(n) = \sum_{k=0}^{\lfloor N/2 \rfloor} \tilde{a}_{2k}\, n^{2k} \tag{4.1}$$

polynomial estimator $\tilde{p}$, of an arbitrary degree N, and coefficients $\tilde{a}$ that minimizes the least squares error difference, which in the case shown in Figure 4.1, responds to the unit impulse in the range of n integers $-M \leq n \leq M$. Due to its control's flexibility, compared to the former, it makes available more degrees of freedom. The challenge is to choose the best tradeoff between the window frame size, the derivative order and the polynomial coefficients. Color transitions are made through filter resetting, similarly to what was done by Jardim and Morgado-Dias [35] every time the pixel under scrutiny is outside the established variance markers. It is possible to obtain better results than the standard FIR filters, such as those previously obtained with the moving average filter due to a better interpolation adjustment, hence, resulting in better high-frequency signals preserving, critical in denoising processes.

The key principle of noise reducing is to seek for a low variability between adjacent values, a well-known characteristic of any signal's source. It is possible to use this feature to correlate adjacent pixels while identifying the random uncorrelated additive noise. Though, it is possible to substitute the correlated signal points with an average value, so reducing the high variability differences. If more surrounding values are compared, the further the ambiguity reduces. In this sense, it is easy to see that Savitzky-Golay filter follows the same line as the moving average filter. Coupled with contour color detection, most of the image's relevant information can be preserved.

Due to its polynomial nature, Savitzky-Golay filters can be easily derivate $d-1$ times, d being the polynomial degree. This often reveals hidden information, like the trend of the curve, the inflective points and gives the possibility to take appropriate decisions for data treatment. It has frequently been verified that degrees of $d=2$ and $d=4$ give the best results. This behavior contrasts with the flat nature (if no autoconvolution is applied) of the moving average filter. The lower the polynomial degree, the better it smooths the signal but less preserves the highs and the widths of the signal.

Arguably, the biggest advantage of making all the operations in time domain, within the context of this work proposed objectives, is that complex domain transforms' computations along with additional processing for selection and suppression of bands where noise is more noticeable, all together are avoided.

All floating-point data computation was done using MATLAB scripts and compared with a Zynq-7000 FPGA series for fixed-point results. The theory contents are reviewed in Section 4.2, including the process of noise addition to a signal, the parametrization to measure the noise in images, a brief explanation of the Savitzky-Golay filter and finally, the color sharp transition algorithm. In Section 4.3 a brief description of configuration of the testing platform is given. On Section 4.4 we describe the results, and in Section 4.5 we take the conclusion notes.

4.2. Proposed Savitzky-Golay filter

4.2.1 Uncorrelated Stationary Random Processes

Noise is a stationary, uncorrelated random process that is normally attached to images acquired from image sensors. When the acquisition channel is unknown or when the noise itself expresses an unknow behavior it can be assumed to possess a zero-mean, variance σ^2 and assumed, as well, as to be equally distributed all over its bandwidth, that is, white Gaussian noise. It can, however, culminate in estimation, leading to the creation of more appropriate models. Image features, such light, brightness and contrast should not change the noise model for the same sensor. Noise generation acquired from photonic contribution is a dependent process, while noise generated by the equipment underlying circuitry photon-to-electron conversion is an independent process. The contribution ratio of the latest, the deterministic model, and the foremost, the photon unpredictable model, has been reduced due to the progress being made in the technology.

4.2.2 Noise metrics

Most channels add noise spread all over its bandwidth that follows a Gaussian probability distribution known as AWGN or Additive White Gaussian Noise with a zero-mean and a variance σ^2. A feasible way to get the noise metrics is to determine how far a set of noise free pixels is from its counterpart contaminated version. The least square error, MSE, can be used for this purpose:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{Y}_i - Y_i\right)^2 \tag{4.2}$$

$\hat{Y}$ is the estimator of the noiseless image and Y the noisy image. The estimator is the unbiased probabilistic parameter acquired from a theoretical infinite number of trials, N, which will converge and will result into the Central Limit Theorem. Thereafter, as N grows, σ^2 reduces proportionally. Once determined, MSE can be used to obtain another important metric, the Signal-to-Noise Ratio, SNR:

$$SNR = 10 \log_{10}\left(\frac{Y^2}{MSE}\right) \tag{4.3}$$

The peak SNR, PSNR, can be obtained using the image maximum pixel resolution, *maxPR*:

$$PSNR = 10 \log_{10}\left(\frac{maxPR^2}{MSE}\right) \tag{4.4}$$

Structural Similarity Index, SSIM, is yet another parameter that can be used to quantify how far the original pixels detach from their noisy counterparts.

4.2.3 Savitzky-Golay filter

Savitzky-Golay smoothing filters, Schafer et al [24], are typically used to "smooth out" a noisy signal whose frequency span (without noise) is large. They are also called digital smoothing polynomial filters or least-squares smoothing filters. In essence, this kind of filters perform, in some applications, better than standard averaging FIR filters, which tend to filter the high-frequency content along with the noise. Additionally, these filters are more effective at preserving high frequency signal components but are less successful at rejecting noise.

Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to frames of noisy data.

They fall into a class of low-pass, time domain filters that smooth out data high variability taking advantage of the least-squares to minimize the errors, seen in Figure 4.2. On top, a synthetic signal with added Gaussian white noise. Below, the dashed line represents the noiseless signal and the solid line is filtered signal. These filters belong to the class or FIR filters, which are non-recursive and are defined as polynomials. Depending on its polynomial degree suitable for some particular applications, they can be set up to adjust a particular curve or signal. They are considered better than other similar filters in the sense that they can be tweaked to preserve high frequencies hence, either or rejecting less noise or increase the smoothness.

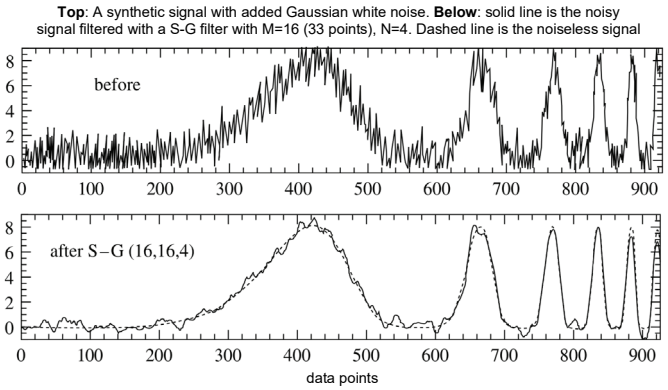

Figure 4.2 - S-G filter with a size of 33 points and a 4th order degree.

Similarly, to the moving average filters proposed by [35], Savitzky-Golay filters are completely described in the time domain. The interest in these filters came from the observation that its successive derivatives could give information pretraining to the trend of the curve, which is not always easily detectable, namely the heights and the widths and the inflective points. They constitute a useful tool in chemical spectrometric analysis and, in subject of this work's interest, noise reduction.

Savitzky-Golay filters can be described as:

$$g_i = \sum_{n=-n_L}^{n_R} c_n \, f_i + n \, c \tag{4.5}$$

The pixel f at position i to be filtered is linearly combined with adjacent pixels, to its left n_L and to its right n_R, to generate the pixel g. Parameters c_n are the weight contribution coefficients that have to be determined to take advantage of the curve fit.

4.2.4 Behavior on the frequency domain

Simply stated, the idea for building this filter is to find a polynomial equation p, with coefficients a, with variable n:

$$p(n) = \sum_{k=0}^{N} a_k \, n^k \tag{4.6}$$

that fits the data by the difference of the least squares error, ε,

$$\varepsilon_N = \sum_{n=-M}^{M} \left(\sum_{k=0}^{N} a_k n^k - x[n] \right) \tag{4.7}$$

where x is the data points. Figure 4.3 shows the behavior of the S-G filter with several polynomials' orders which are the frequencies' domain representation of the impulse response given in Figure 4.1. Care must be taken to properly adjust the curve fitting, that is, the filter size $2M + 1$, has to be equal or greater than the number of the coefficients of the polynomial order, $N + 1$. Additionally, N shall not be too large resulting, as such, in an improper curve fitting. However, given these unfavorable factors, if the order N and the number of points M are wisely chosen it is possible to take advantage of several frequency pass or rejecting band behaviors.

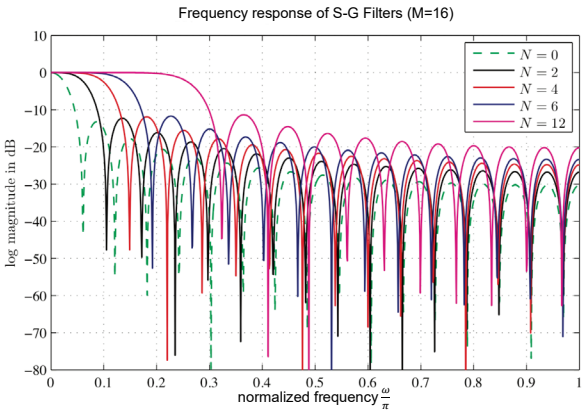

Figure 4.3 - S-G filters in frequency domain with $M = n_L = n_R = 16$ and polynomials orders N of 0, 2, 4, 6 and 12 [24].

The filter half decay, where the cutoff frequency happens at 3 dB, is determined by the order N and by its length M. Like the moving average filter, as expected, the longer the M the shorter the cutoff frequency.

The Z-domain unit circle where the zeros might appear, traces the sharpness of the cutoff decay.

It can be shown that if the system response, $H(e^{j\omega})\big|_{\omega=0} = 0$, it imputes a constant top limit horizontality during its passband. It continues this behavior through its r^{th} derivative, on the frequencies' domain, ω, for $r = 1, 2, \ldots,$ N:

$$\frac{d^r H(e^{j\omega})}{d\omega^r}\bigg|_{\omega=0} = (-j)^r \sum_{n=-M}^{M} n^r h[n] = 0 \tag{4.8}$$

where h is the system's impulse response.

Similar to the moving average filter, the S-G has a poor stopband. The secondary lobes have high amplitude in pretty much all of the fitting polynomial orders.

It is well established that the higher the polynomial order N, the higher the cutoff frequency, and for this matter, as stated above, the shorter the number of points M.

4.2.5 Color transitions

Different from the constant mean value of the moving average filter in Jardim and Morgado-Dias [35], Equation 4.8, the color transitions are now the mean of the least square errors, $\overline{\varepsilon_N}$, given in Equation 4.6:

$$\overline{\varepsilon_N} = y \tag{4.9}$$

The decision to include the next pixel, p, in the curve fitting, is taken if its value is within the threshold marked by

$$(\overline{\varepsilon_N} - \sigma) \leq p \leq (\overline{\varepsilon_N} + \sigma) \tag{4.10}$$

p resets the filter if it is outside the threshold.

The standard deviation, σ, where M is the filter length, is:

$$\sigma = \sqrt{\frac{\overline{\varepsilon_N}}{2M+1}} \tag{4.11}$$

Shown by Jardim and Morgado-Dias [35], the color sharp transition maintains clear image's contours, i. e., it keeps out the blurriness. Figure 4.7, shows an image with a row of pixels sampled, on top, which was added variance noise of 0.001.

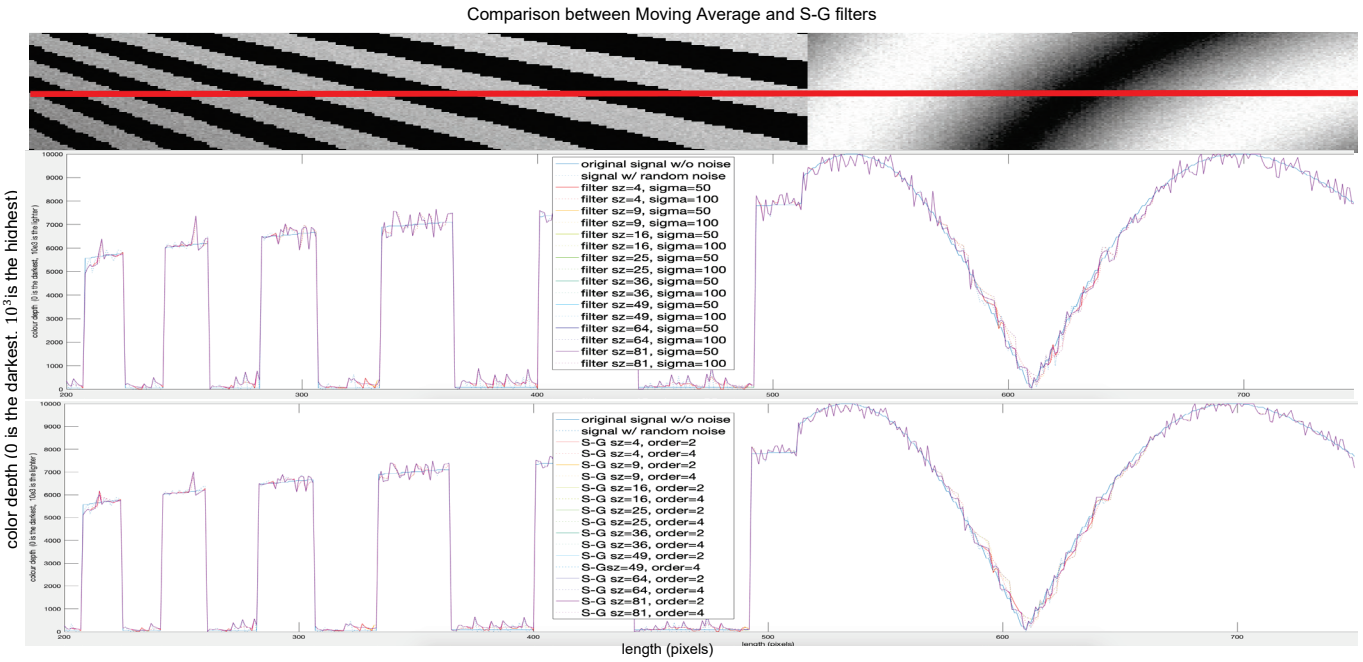

Figure 4.4 - Filtering an image pixels row, profiled by a red line (on top) with a Moving Average filter (middle curve) and with a S-G filter (bottom curve).

In the middle, the row is filtered using various sizes of moving average and in the bottom, filtered with several sizes Savitzky-Golay filters with 2^{nd} and 4^{th} orders. It is immediately visible that the S-G filters make a better filtering job. High values reveal lighter tones while low values are darker tones.

It is our intention to show that, substituting the Moving Average filters, implemented in FPGAs [35], by Savitzky-Golay filters, not only does not increase complexity much more but also, reveal better results.

4.3. Test settings

The methodology used for the tests settings was to obtain as clear as possible, i. e., noise free, high-resolution images, as a reference, and then generate pseudo-random noise white Gaussian noise, AWGN, using a variance

of $\sigma^2 = 1 \times 10^{-4}$.

The algorithms are then tested in a fixed-point FPGA board comparing it with a floating-point MATLAB version with the same bit-depth. Both platforms ran a version of the moving average filter comparing it with the proposed Savitzky-Golay filter. Tests were done using 13-bit depth resolution source images, which were acquired from a high sensitivity sensor on a balanced brightness-contrast environment, targeting the minimization of the photonic noise dependency.

In order to generate the Gaussian noise, the reference source images were fed into MATLAB, were normalized and noise contaminated then the version to be tested in the FPGA board was converted back to fixed point 13-bit depth resolution. Normalization in MATLAB is a required process since mostly functions work by taking a minimum quantization scaling error in the range between 0 to 1.

Due to its versatility, fixed-point tests ran in a ZedBoard FPGA SoC Zynq-7000, model XC7Z020-CLG484-1, that can be seen in Figure 4.5. Xilinx Vivado 2018.3 version, was used as platform environment. The main FPGA programming language used was Very-High Speed Integrated Circuit Hardware Descriptive Language, VHDL, and some APIs in Verilog. The noise contaminated image, fixed-point version, was transferred to the FPGA board using a mini SD card, the same that bootstraps a Linux OS board adapted version.

ZedBoad Zynq-7000 SoC FPGA boad

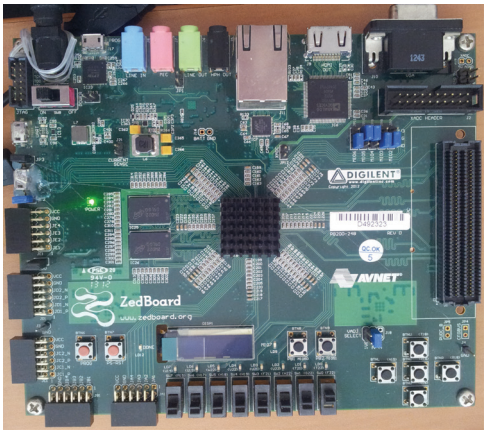

Figure 4.5 - Testing FPGA SoC ZedBoard Zynq-7000 platform.

Some math operations such as multiplication can easily overflow if care is not taken to properly dimension the registers used for operations. Images were acquired using a 13-bit pixel photosensor, but for consistency and platform compatibility (FPGA and MATLAB), images were scaled up to 16-bit. This is because register reusage due to successive refeed operations can easily lead to sky rock roundoff errors. Therefore, MATLAB's natural operation with 64-bit

registers was chosen to be implemented on the FPGA also not only to maintain the errors to a minimum, because it floors integers to the nearest decimal zero, but also accommodates deeper bit resolutions, that is, above 13 bits. It should be said that this sensor has a photon to electrons factor conversion of 1 to 1×10^3.

Jardim and Morgado-Dias [35] verified that the squared errors differences between the fixed- and the floating-point calculations was roughly $2 \times 10^{-3}\%$ and $2 \times 10^{-2}\%$. We expect to maintain the same order of magnitude for the two filters' comparisons.

4.4. Results

Figure 4.6 shows a comparison between a moving average filter a S-G filters. The filtering curves were generated without searching the colors' contours. It is clearly visible that the S-G filters give the best results in the signal preservation.

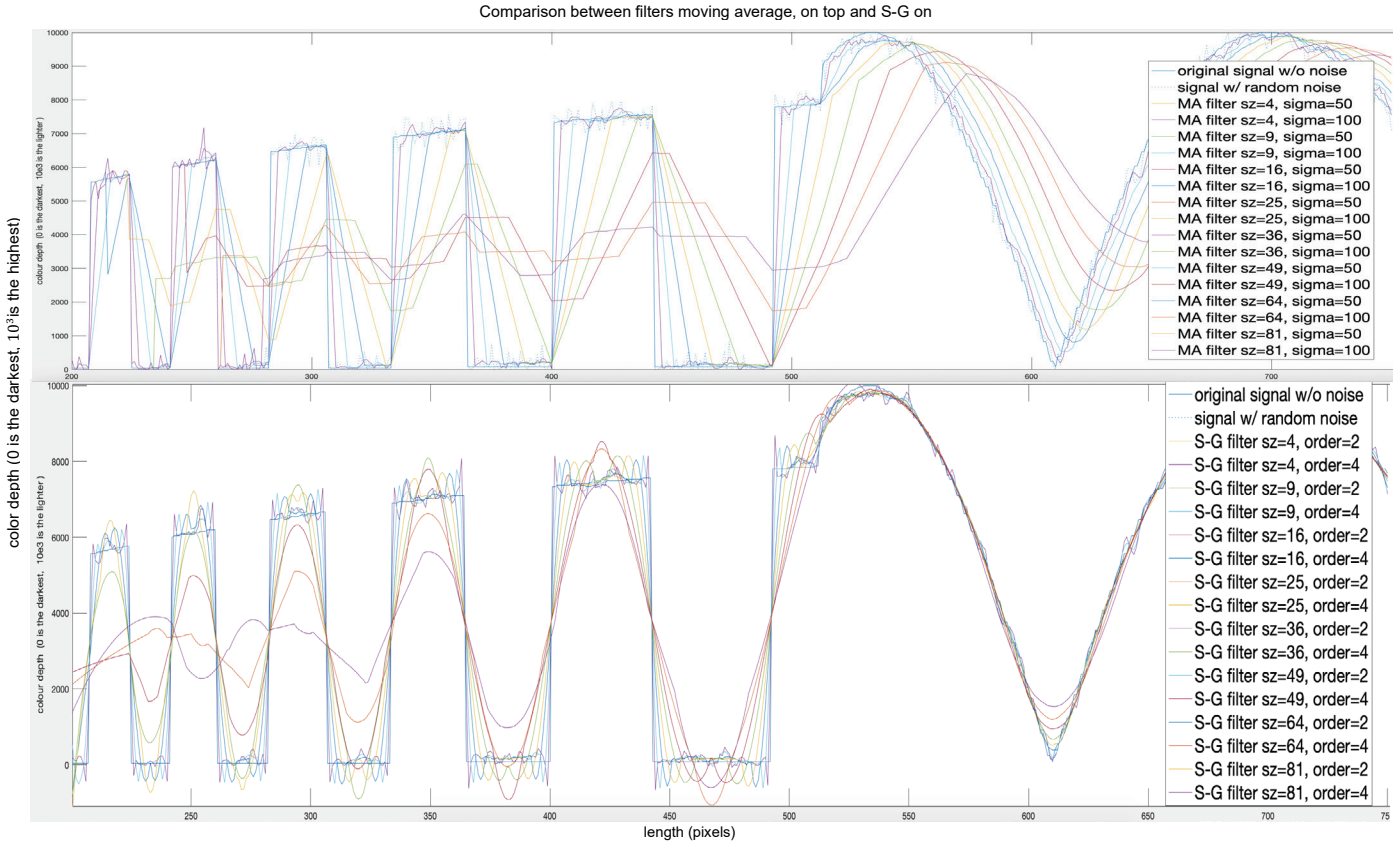

Figure 4.6 - Above, moving average vs. below, Savitzky-Golay.

A pair of images is given in Figure 4.7, the top one is the noiseless and the bottom is the same image but with 0.001 AWGN variance. Figures 4.8 through 4.12 show comparisons between MA filters, on top and S-G filters, on bottom. As verified by Jardim and Morgado-Dias [35], for MA filters above $N = 25$, the filtering removes too much image information, hence, in this paper, S-G filtering will not be extended above this value. Given the results achieved by Jardim and Morgado-Dias [35], the best filters sizes are for $N = 4$, 9, 16 or 25. Since Savitzky-Golay filters work better with odd filter sizes, they will be incremented, if even. Regarding to the MA's Sigma Multiplicative Factor, SMF=50 or SMF=100, it will be used, in the case of Savitzky-Golay, two polynomial orders of $N = 2$ and $N = 4$, respectively.

Appendix 4.A, shows a table with the results for the MA and the S-G filters. The fixed-point ones were done in the Zynq-7000 FPGA and the floating-point ones in MATLAB.

4.4.1 Discussion of the results

Results are shown below, in Table 4.1. Comparison was made between a moving average filter of several sizes and the Savitzky-Golay filter order 2, compared to SMF of 50, and the order 4, compared to SMF of 100. These orders were chosen according to several studies already made, being consistently verified to give the best results. This is majorly due to the optimal coefficients that makes the polynomial best fit the data with the minimum squares error, hence, as can be showed in Figure 4.6, that the S-G filter "constantly" seeks to fit a given signal, even if it encounters high frequency steeply transitions, using lengthy filters, contrary to the MAs which pretty much behave by averaging the whole signal, meaning to lose all the image's information details.

On other hand, MA filters deal better with noisy flat colors, since they bounce with a less degree, especially after a high derivative transition, that is, are less affected by Gibbs effect.

It can be questionable whether comparing MAs' SMFs with S-Gs' polynomial orders would be a fair treat. However, comparing the same filtering length for both filters, a trade between curve adaptability (in S-G) for an increase of variance (in MA) would be feasible, given the attained results. S-G achieves better results in shorter filter sizes, degrading to lengthy filters, in orders of 2 vs. 4, (38 to 40 vs. 27 to 36 dB), comparatively to MA filtering.

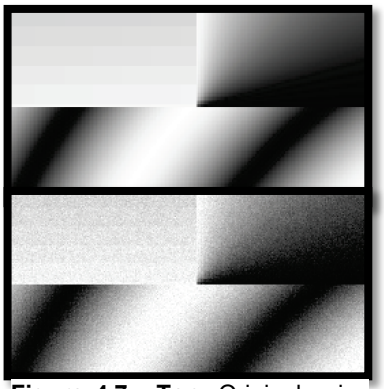

Figure 4.7 - **Top:** Original noise free. **Bottom:** Added noise with 0.001 AWGN variance (σ^2).

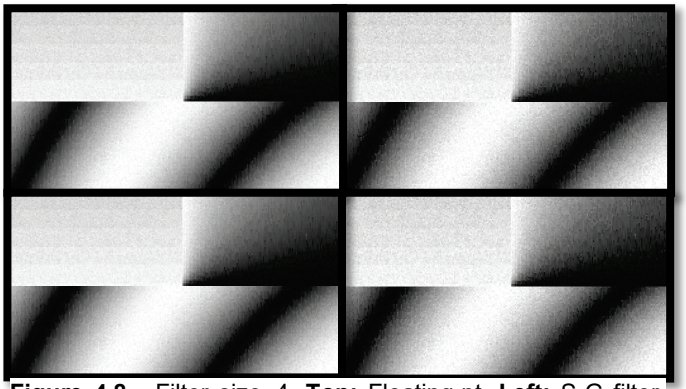

Figure 4.8 - Filter size=4. **Top:** Floating-pt, **Left:** S-G filter, Ord=2, **Right:** MA filter, SMF=50. **Bottom:** Fixed-pt. version of the Top.

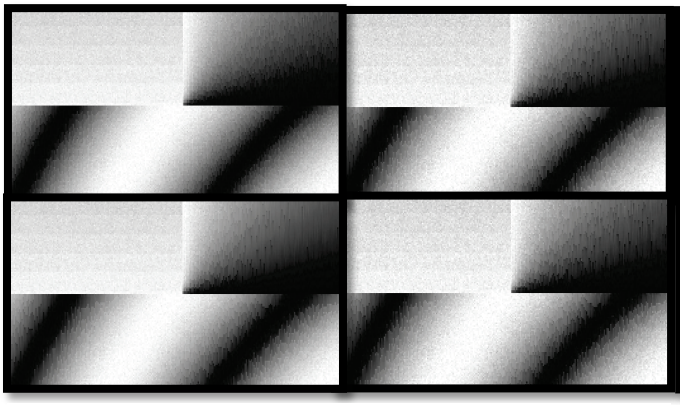

Figure 4.9 - Filter size=16. **Top:** Floating-pt, **Left:** S-G filter, Ord=2, **Right:** MA filter, SMF=50. **Bottom:** Fixed-pt. version of the Top.

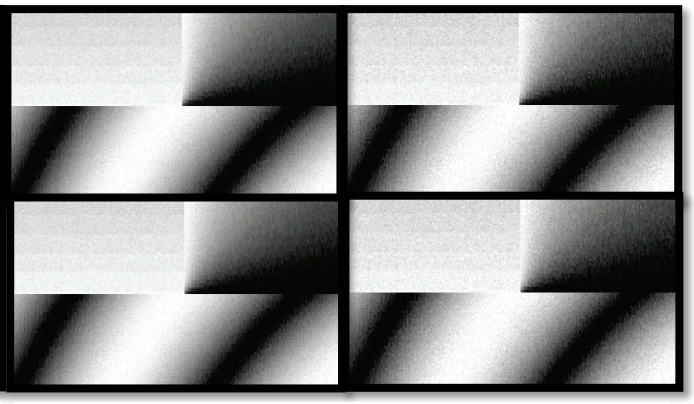

Figure 4.10 - Filter size=9. **Top:** Floating-pt, **Left:** S-G filter, Ord=4, **Right:** MA filter, SMF=100. **Bottom:** Fixed-pt. version of the Top.

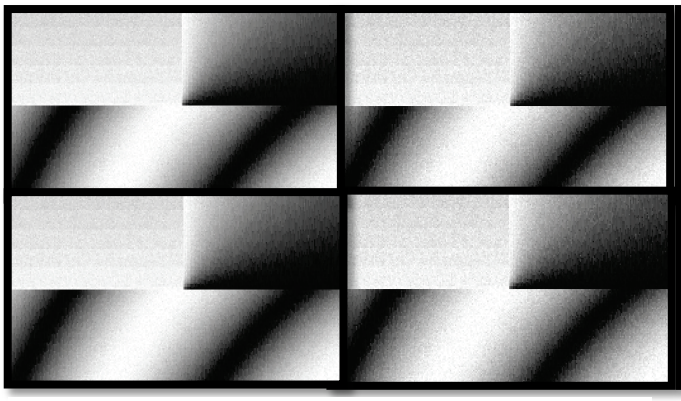

Figure 4.11 - Filter size=25. **Top:** Floating-pt, **Left:** S-G filter, Ord=4, **Right:** MA filter, SMF=100. **Bottom:** Fixed-pt. version of the Top.

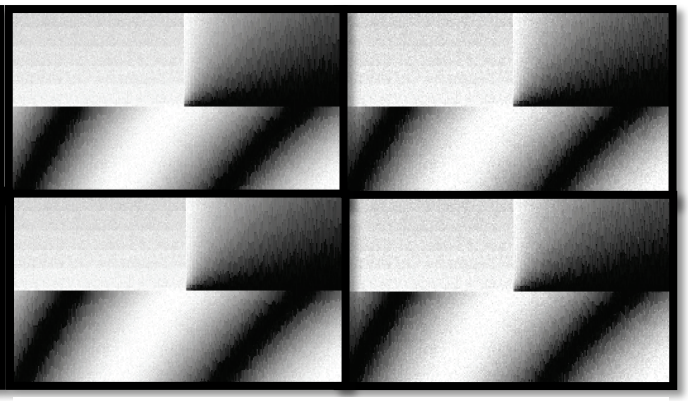

Figure 4.12 - Filter size=16. **Top:** Floating-pt, **Left:** S-G filter, Ord=4, **Right:** MA filter, SMF=100. **Bottom:** Fixed-pt. version of the Top.

The last set of images, in Figure 4.12, show that both the MA and the S-G, filters with length above 25 points the smoothing is done in patches, visibly a shear reduction in the image's quality.

4.5. Conclusions

There is available several degrees of freedom to exploit, and it is important to say that, due to the polynomial nature of the Savitzky-Golay filters, each new derivative can reveal further information not immediately seen in the data being smoothed. It is possible to infer from the figures, that the best results are achieved by S-G filters, those on the left columns.

The final inferences can result in ambiguous decisions and it should be said that these tests were done, using as source, a generated synthetic image that can, most of the time, resemble real-world pictures due to its trade in smooth color changes and sharp color transitions. Comparison between floating- and fixed-point calculations are minimal due to the use of 64-bit registers. This is possible because the registers are much larger than the images bit depth resolutions and this way it is possible to reduce the round-off errors. Consequently, as a rule of thumb, the registers to be chose, in order to make these computations should be at least the quadruple the bit-depth size the highest digital number of the picture, rounded up to the nearest multiple of 8. In this case, the 13-bit depth images used for the tests were round up to 16-bit and all the math done on 64-bit registers.

An inverse scaling polynomial function of low order degree was created, modeled and adjusted as a post-filter to the application of the S-G filters to further smooth the noise in darker colors relatively to the lightest ones, because the noise is more perceptive in dark tones.

4.6. Acknowledgments

The author would like to acknowledge the Portuguese Foundation for Science and Technology for their support through Projeto Estratégico LA 9 - UID/EEA/50009/2019.

4.A. Appendix

Table 4.1 - Numerical results, measured with parameters MSE, SNR, PeakSNR, Global Similarity and comparison between S–G and MA filters. Some of these results correspond to the Figs. 4.8 – 4.12, as indicated by the Test/Figure column.

Test / Figure	Parameter	Floating Point (MATLAB)		Fixed Point (FPGA)		Change (%)			
	Filter Used	Savitzky-Golay	MA	Savitzky-Golay	MA	S-G Fix. to Flt.	MA Fix. to Flt.	Float S-G to MA	Fixd. S-G to MA
1 / 4.8	Filter length	4							
	Polynom. Order	2		2					
	SMA		50		50				
	MSE	0.000337336	0.000473407	0.000337345	0.000473429	2.67E-03	4.65E-03	-28.74	-28.74
	SNR (dB)	32.59	31.12	32.59	31.12	-2.53E-03	-4.61E-03	40.34	40.34
	PeakSNR (dB)	34.72	33.25	34.72	33.25	-2.76E-03	-4.61E-03	40.34	40.34
	GS (%)	83.81	78.93	83.81	78.93	4.18E-03	3.04E-03	4.88	4.88
2 / NA	Filter length	4							
	Polynom. Order	4		4					
	SMA		100		100				
	MSE	0.000315486	0.000430113	0.00031552	0.000430134	1.08E-02	4.88E-03	-26.65	-26.65
	SNR (dB)	32.89	31.54	32.88	31.54	-1.11E-02	-4.61E-03	36.33	36.33
	PeakSNR (dB)	35.01	33.66	35.01	33.66	-1.11E-02	-4.61E-03	36.33	36.33
	GS (%)	85.60	81.24	85.60	81.24	5.84E-03	4.92E-03	4.36	4.36
3 / NA	Filter length	9							
	Polynom. Order	2		2					
	SMA		50		50				
	MSE	0.00035738	0.00049495	0.000357373	0.000494928	-1.96E-03	-4.44E-03	-27.79	-27.79
	SNR (dB)	32.34354	30.92923	32.34362	30.92942	1.84E-03	4.38E-03	38.49	38.49
	PeakSNR (dB)	34.4687	33.05438	34.46878	33.05458	1.84E-03	4.61E-03	38.49	38.49
	GS (%)	82.89	78.06	82.90	78.07	4.34E-03	4.48E-03	4.83	4.83
4 / 4.9	Filter length	9							
	Polynom. Order	4		4					
	SMA		100		100				
	MSE	0.000389991	0.000500329	0.000390046	0.000500359	1.41E-02	6.00E-03	-22.05	-22.05
	SNR (dB)	31.96429	30.88229	31.96368	30.88202	-1.40E-02	-6.22E-03	28.29	28.28
	PeakSNR (dB)	34.08945	33.00745	34.08884	33.00718	-1.40E-02	-6.22E-03	28.29	28.28
	GS (%)	84.54	80.10	84.54	80.10	6.62E-03	4.49E-03	4.44	4.44
5 / 4.10	Filter length	16							
	Polynom. Order	2		2					
	SMA		50		50				
	MSE	0.000358618	0.00049607	0.000358593	0.000496053	-6.97E-03	-3.43E-03	-27.71	-27.71
	SNR (dB)	32.32852	30.91941	32.32883	30.91956	7.14E-03	3.45E-03	38.33	38.33
	PeakSNR (dB)	34.45368	33.04457	34.45399	33.04472	7.14E-03	3.45E-03	38.33	38.33
	GS (%)	82.78	78.00	82.79	78.00	5.19E-03	4.49E-03	4.79	4.79
6 / 4.11	Filter length	16							
	Polynom. Order	4		4					
	SMA		100		100				
	MSE	0.000404021	0.00051525	0.000404014	0.000515254	-1.73E-03	7.76E-04	-21.59	-21.59
	SNR (dB)	31.8108	30.75466	31.81088	30.75463	1.84E-03	-6.91E-04	27.53	27.53
	PeakSNR (dB)	33.93596	32.87982	33.93604	32.87979	1.84E-03	-6.91E-04	27.53	27.53
	GS (%)	83.67	79.15	83.68	79.15	8.72E-03	3.79E-03	4.52	4.53
7 / NA	Filter length	25							
	Polynom. Order	2		2					
	SMA		50		50				
	MSE	0.000358561	0.000496088	0.000358545	0.000496071	-4.46E-03	-3.43E-03	-27.72	-27.72
	SNR (dB)	32.32921	30.91925	32.3294	30.91941	4.38E-03	3.68E-03	38.36	38.36
	PeakSNR (dB)	34.45437	33.04441	34.45456	33.04457	4.38E-03	3.68E-03	38.36	38.36
	GS (%)	82.79	77.99	82.79	78.00	5.07E-03	4.87E-03	4.79	4.79
8 / 4.12	Filter length	25							
	Order	4		4					
	SMA		100		100				
	MSE	0.000406163	0.000516593	0.000406159	0.00051658	-9.85E-04	-2.52E-03	-21.38	-21.38
	SNR (dB)	31.78784	30.74335	31.78789	30.74347	1.15E-03	2.76E-03	27.19	27.19
	PeakSNR (dB)	33.913	32.86851	33.91304	32.86863	9.21E-04	2.76E-03	27.19	27.19
	GS (%)	83.25	78.91	83.26	78.92	7.69E-03	4.94E-03	4.34	4.34

5. Testing

5.1 Platform

The testing platform was already covered in Chapters 3 and 4. However, and summing up, the main processing device used for tests and simulations was the ZedBoard Zynq-7000 All Programmable System on Chip (SoC) XC7Z020-CLG484-1 FPGA board, which is a hybrid FPGA+CPU all-in-one chip.

The programmable logic is based on the Artix-7 series equivalent FPGA. The CPU is based on a dual ARM Cortex A9 running at 667 MHz. The shared memory between FPGA and the CPU is a 32-bit wide bus with a size of 512 MB type DDR3. This shared memory is a key feature in order to transfer and ease the data flow between the 2 processing units.

On the CPU side, the data is prepared, resized in proper chunks and fed to the FPGA through the memory common address space. The Advanced eXtensible Interface (AXI4)-Lite protocol is used to setup, synchronize, read, write and flag the traffic in both ways. Figure 5.1 shows the layout of the Zynq-7000 SoC chipset.

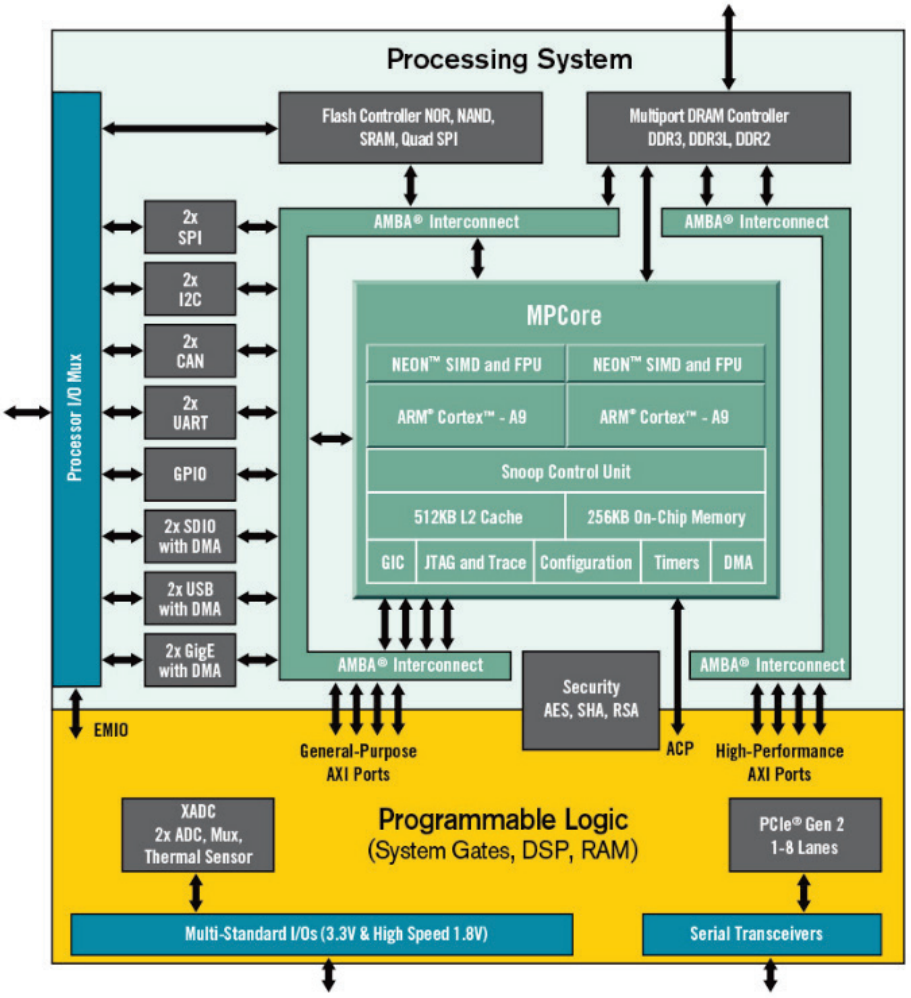

Figure 5.1 - The Zynq-7000 SoC series [38].

The ZedBoard brings an SD card preloaded with a lite version of the Linux OS which is built with a RAM disk file system. This approach, named Xillybus, is an Intellectual Property (IP) developed to let users stay focused straight on the problem/solution paradigm instead of having to divert their attention into technical details of setting up and establish communication between the CPU and the FPGA, which, in fact, increases the chances to introduce additional bugs to the possible ones already existing in their project(s).

Xillybus, comprises a wrapper, i. e., a module which includes data transportation protocols such as Direct Memory Access (DMA) and Peripheral Component Interconnect express (PCIe) in order to handle the data flow not only between the CPU and the FPGA but as well to other peripherals such 100 and Giga bit Ethernet, USB, Video Graphics Array (VGA), and several others, including the dedicated JTAG debug serial channel, through a USB port. Figure 5.2 shows the ZedBoard Zynq-7000 SoC along with all the possible peripheral accesses.

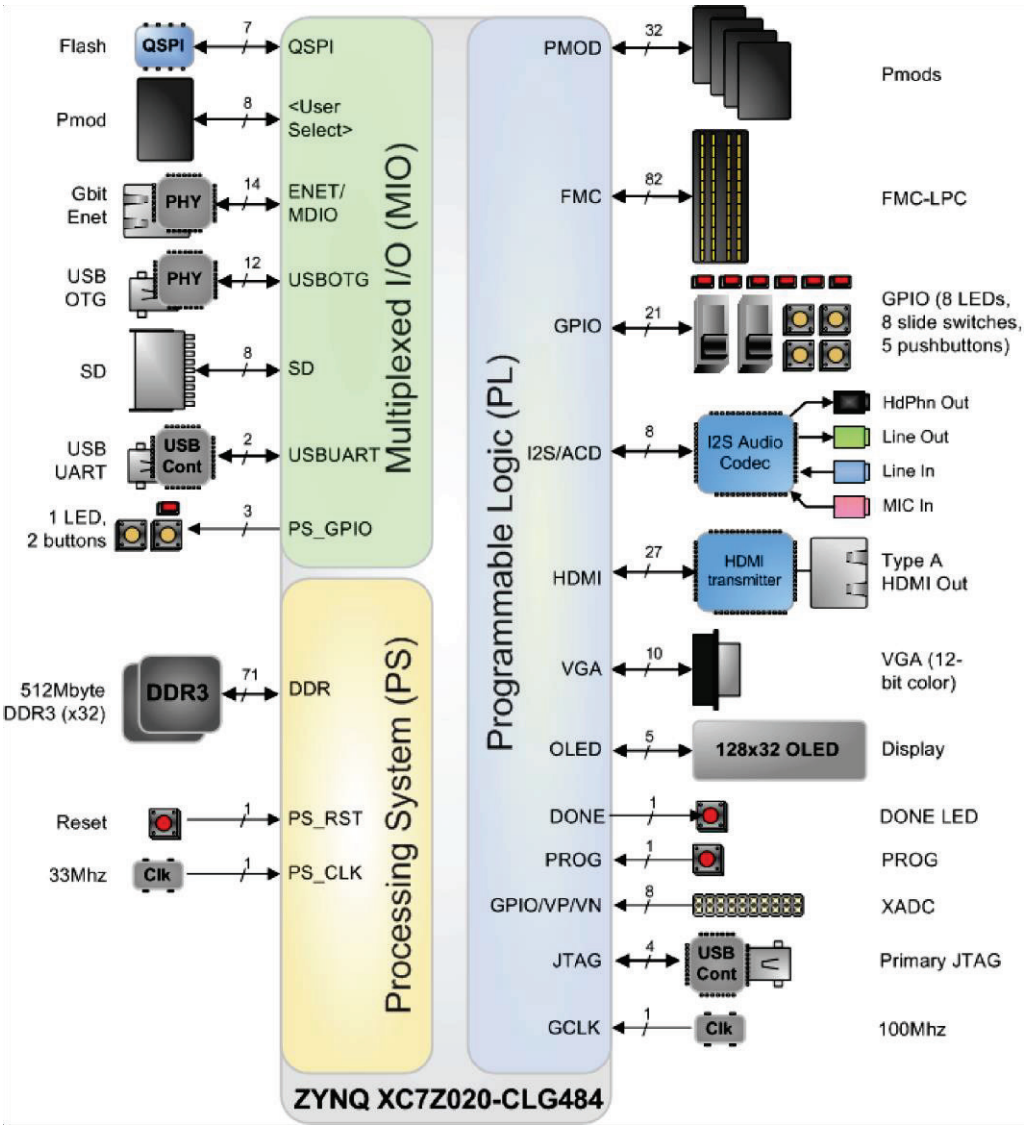

Figure 5.2 - ZedBoard Zynq-7000 SoC with peripheral accesses [39].

5.2 Data flow between FPGA and CPU

As stated, the FPGA and the CPU exchange information through a range of memory addresses.

The FPGA and the CPU tasks are explained more detailed later in the next subchapter. However, in brief, the FPGA's role is to receive sorted columns of pixels from an image which is prepared and sent by the CPU, apply the developed algorithm and then return those processed pixels back to CPU.

The Xillybus IP core manages one or more simultaneous bidirectional data streamings, for example, audio, VGA and a user IP, between the programable logic (FPGA) and the processing unit (CPU), as illustrated in Figure 5.3.

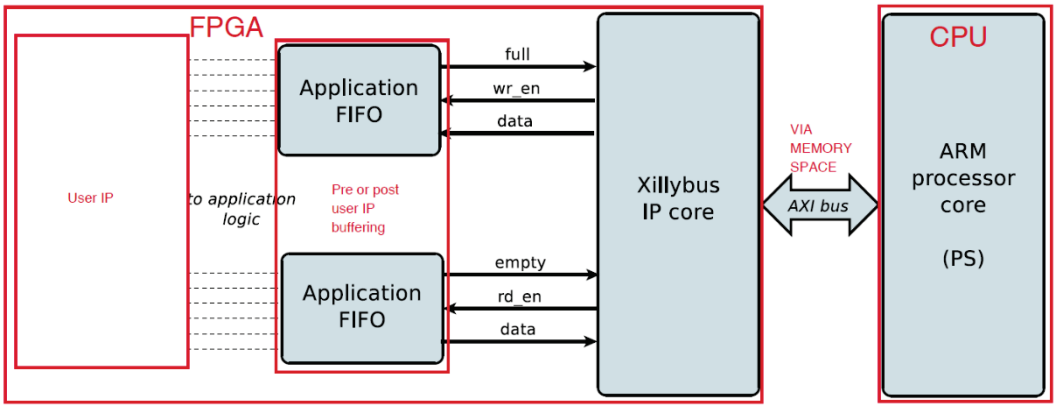

Figure 5.3 - Xillybus IP core functionality. User IP is the author's own work.

The User IP is the author's developed algorithm.

As shown in Figure 5.3, a developed user application running on the CPU side communicates with User IP module, on the FPGA side, through the Xillybus IP core. The application supplies the raw data to be processed to the FPGA User IP module and afterwards collects it after being processed, which can then be saved to the SD card or relayed to the one of the peripherals such as VGA, Ethernet, USB or any other transport channel.

The User IP can be composed by one or more modules, of any size and complexity, limited only to the FPGA resources available. All the modules are wrapped within a top-level module, which given the context of this Master dissertation, is called "Moving Average".

Depending on the type of project being designed, a First In First Out (FIFO) memory buffer can be placed either before, after or both (in some rare design requirements) of the User IP module, relieving the user to have to worry

about traffic details. This is achieved through Xillybus memory fullness or emptiness cycling probing, taking an appropriate action accordingly by triggering or stalling data transfers, through the “full”, “empty”, “read enable” and “write enable” flags. in either streaming directions. The Xillybus streaming bidirectional flow management, for matters of simplicity, is made without any assumptions about any specific data rates. There is, however, a minor efficiency price to pay with this procedure, in that, for small data chunks, instead of buffering a considerable amount of data and thereafter burst it, the FIFO keeps receiving and sending any data chunk sizes. For any other specific project, which every bit of efficiency is accounted, or even the need to control DMA memory access buffers is a critical factor, this solution offers little to none benefits. Its purpose is merely to ease one or more of three possible scenarios, prototyping, what-if testing or DMA future maintenance concerns.

5.2.1 Driver communication process

The communication process behaves pretty much like a named pipe [37]. A pipe is a shared memory section that serves for an intercommunication process between the FPGA IP module and the CPU host application. A named pipe is a one-way or a duplex pipe intended to establish communication between the pipe server and the pipe client. These later roles are exchangeable between the FPGA module and the host CPU, and for them, the pipes are seen as regular files which can be opened, read a written. However, its behavior is more like Internet TCP/IP streams or piped intercommunication processes, which in reality, is a FIFO on an FPGA fabric logic instead of yet another CPU process [37], [40].

For this communication process to be effective, data streams and its parameters are detected by the driver upon its load to the memory and, on the CPU host side, by the running application request, it is opened a memory communication process which is established through the AXI protocol from and to where the application can open, read and write files. Additionally, DMA buffers allocated on the host application and on the FPGA sides are informed about its addresses. The size and the number of the DMA buffers are stream independent. The exchanged data is transparent for both the host application and the IP core module and interpreted as a continuous data stream. In fact, the DMA buffers are being filled, handed over and acknowledged from side to side [37], [40].

5.3 FPGA IP core module

5.3.1 Flowcharts

The IP module is composed by three processes, $proc\_calc\_sum$, $proc\_calc\_mean$ and $proc\_check\_if\_numerat\_is\_0$, which are shown in the Figure 5.4, Figure 5.6, and Figure 5.9, respectively.

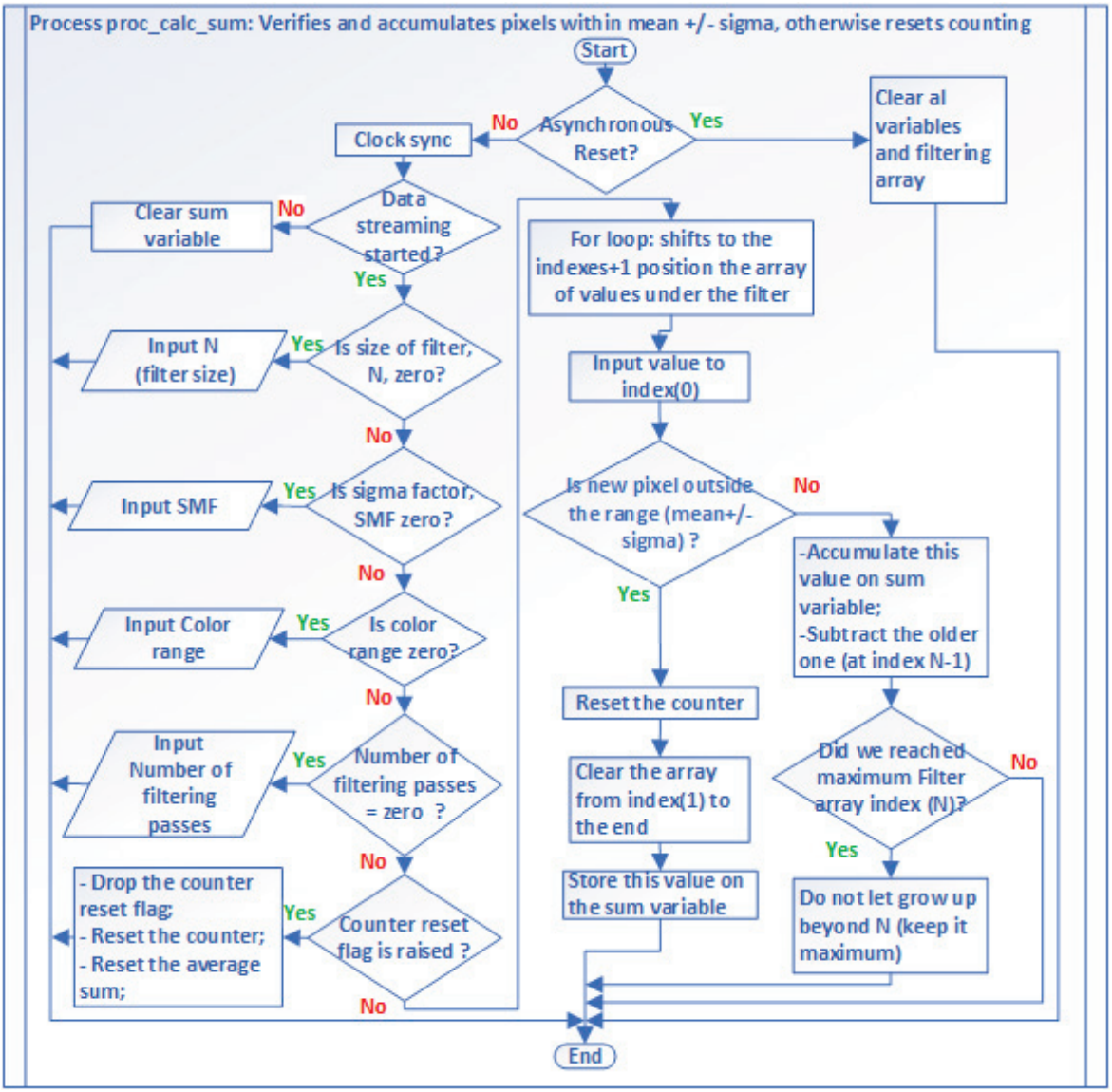

Figure 5.4 - Process used to calculate the sum of pixels used for averaging.

Concerning to the main process $proc\_calc\_sum$, whose flowchart is shown above, in Figure 5.4, as soon as the IP module starts receiving pixel values, send from the host application in the CPU side through the Xillybus management and upon clock synchronization, it immediately verifies if these pixels belong to a new streaming, which has a layout shown, below, in Figure 5.5.

Figure 5.5 – Layout of the bidirectional data streaming between FPGA IP module and CPU host

The bidirectional data streaming is organized with a first section, the header, in red, which is composed by the squared filter size, N^2, it is pre-squared since it eases the calculations on the FPGA, the Sigma Multiplying Factor (SMF), explained in Chapter 3.4, the Factor Wheel Conversion (FWC), explained in Chapter 3.3, the maximum pixel bit depth and the number of passes that the filter should auto convolve, which is further explained in Chapter 3.2.5. The next section of the steaming layout are the pixel values to be processed by the FPGA.

On the left half side of the flowchart shown in Figure 5.4, a sequence of verifications is made to check the reset state of the header variables. All these variables are checked, in sequence, on each rising edge of the clock. If they are null then the received value is attributed to that variable. rst signal is used to clear all the module's signals and variables.

After the header has been received, shown on the right half side of Figure 5.4, there is an index incremental shift for the eventual pixels already under the averaging filter from their actual position to make room for the new input pixel. If the pixel is inside the range between $[\mu - \sigma \dots \mu + \sigma]$, except for the first one, which must start with an initial mean value, then it is accumulated on the variable sum. Otherwise, it resets the counter, cleans the variables and starts a new counting, which means that a sharp color contour was detected on that area of the image where the pixel was token.

Additionally, upon a reset, since the filter N with a certain size, depending on the size chose value, grows starting from zero until $N - 1$ it is constantly being assessed in order to verify if that size does not exceed its maximum value. Should ever this situation happen, the filter maintains the maximum size by extracting the older value and inserting the newer received one.

This sequence of pixels is sent from the CPU host side application, column by column which can be seen as the temporal sequence fashion streamed by a line sensor to the FPGA in order to be processed, which is different from the pixels' sequence of the standard spatial processing extracted from the area sensors or from an image already built.

In parallel with the above described process $proc\_calc\_sum$, it runs another process called $proc\_calc\_mean$, whose flowchart is shown below in

Figure 5.6. Upon the sum variable update triggered by the input of each pixel, that variable triggers the $proc\_calc\_mean$ process. The $mean$ variable is then computed by $\frac{sum}{N}$ if the module pixels' $counter$ is active or cleared otherwise.

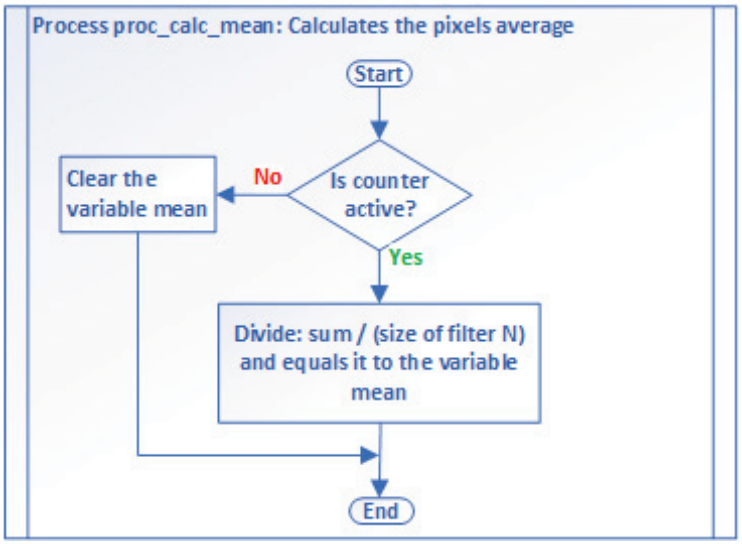

Figure 5.6 – Process which calculates the mean.

After acquiring the mean, μ, in the main module, it is calculated the variance $\sigma^2 = \frac{\mu}{N}$, then standard deviation $\sigma = \sqrt{\frac{\mu}{N}}$ using a thirty-part algorithm [41].

5.3.2 Increasing denoising smoothness on darker colors with a polynomial function

As explained on Chapter 3.5, it was developed and introduced 2^{nd}-order polynomial function to σ range of possible pixels' candidates contributing to the μ average. It was stated that the white noise, because of its equalized Power Spectral Density, it turns to be more perceptible in darker colors. The developed adjustment function increases the filter size in the time domain hence, narrowing the pass-thru filter in the frequencies' domain.

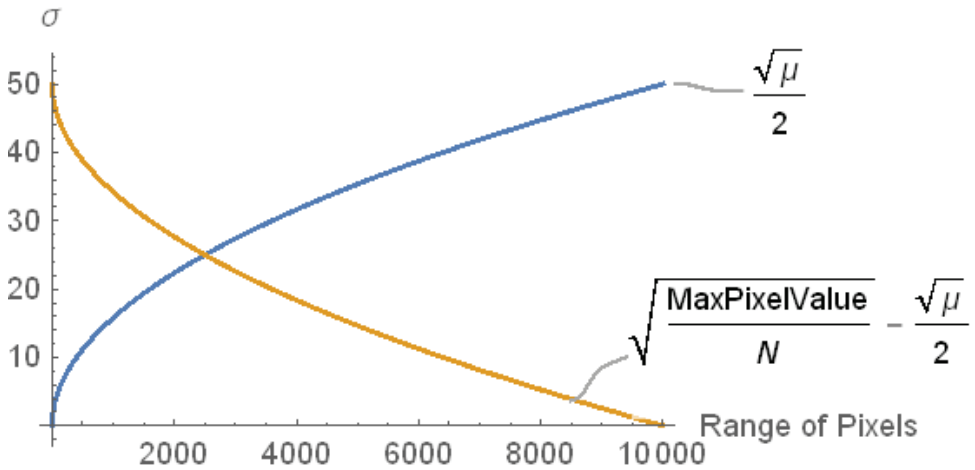

Figure 5.7 – Sigma vs. pixels values curves: theoretical and desirable.

Figure 5.7 shows the σ vs. range of pixel values evaluation. In blue, the theoretical $\sigma = \sqrt{\frac{\mu}{N}}$, which contradicts a correct usage of a larger filter for lower pixel values. It feels the need to invert this function to behave properly in agreement with the desirable result, that is, the sigma must broaden the range of pixels with lower values, restricting it as it progresses to higher values, which is shown by the orange curve $\sigma = \sqrt{\frac{MaxPixValue}{N}} - \sqrt{\frac{\mu}{N}}$.

However, due to the high resolution of the testing images ($2^{13}\ bits = 8196$ colors, compared with standard image resolutions with 256 colors), the developed polynomial had to be adjusted for satisfactory visual results, and has a shape shown in Figure 5.8.

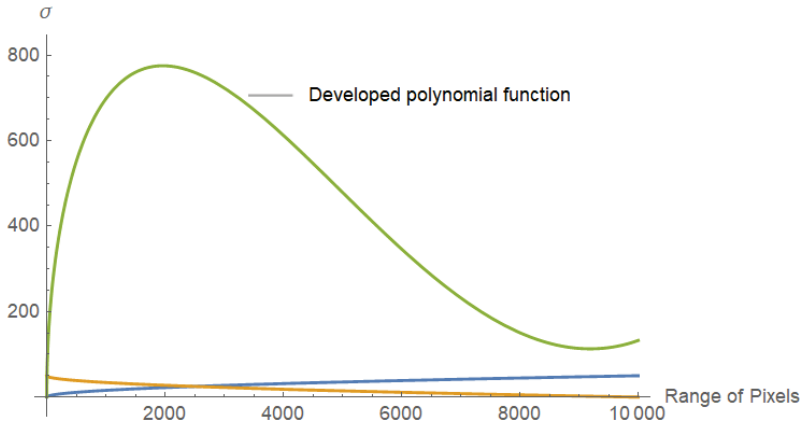

Figure 5.8 – Developed polynomial compared with theoretical and desired curves

The polynomial which was found to be equivalent to the desirable function is illustrated in Equation 5.1:

$$\mathrm{SMF}\left(0.50410^{-8}\left(\mathrm{FWC}\sqrt{\frac{\mu}{N}}\right)^2 - 0.000229\left(\mathrm{FWC}\sqrt{\frac{\mu}{N}}\right) + 1.036\right) \tag{5.1}$$

FWC and SMF have both degrees of freedom which are dependent of the image sensor, in case of the first parameter and the level of noise it inputs, in the latter case. However, as stated above, tests revealed that in the specific case of having high resolution images, a much larger filtering coefficient should be imposed, hence the green curve of the Figure 5.8, is what was implemented after being adjust for the aforementioned best visual results. Additionally, since this function is to be computed on FPGA hardware, it must be converted to integer numbers. The model applied is shown on Equation 5.2, and the members appears in the same order in which the mathematical operations were made, towards efficiency and resources optimization.

$$\frac{\mathrm{SMF}\sqrt{\mu}\,137}{\sqrt{N}\,125} - \frac{\mathrm{SMF}\sqrt{\mu}\,2257\,\mu}{\sqrt{N}\,10^7} + \frac{\mathrm{SMF}\sqrt{\mu}\,\mu^2}{\sqrt{N}\,82372323} \tag{5.2}$$

A verification process is also added to the module in order to check if the numerator is zero. This is trivial and should not be necessary since there is only problems on the division if the denominator is zero. However, problems were experienced in the referred situation and that was the reason this process was added. Since there are no situations where the denominator can be zero, because, as can be verified in Equation 5.2, N can never be zero (its values range from 4 to 100), this process only verifies possible cases where $\mu = 0$. As so, there are 3 members in which divisions applies, each one with a numerator and a respective denominator. Those 3 members are the ones that compose the 2^{nd}-order polynomial function. The checking process is made by verifying if the numerator is zero. Should that be the case and it returns zero, otherwise computes the division. The flowchart of the process $proc\_check\_if\_numerat\_is\_0$ is shown in Figure 5.9.

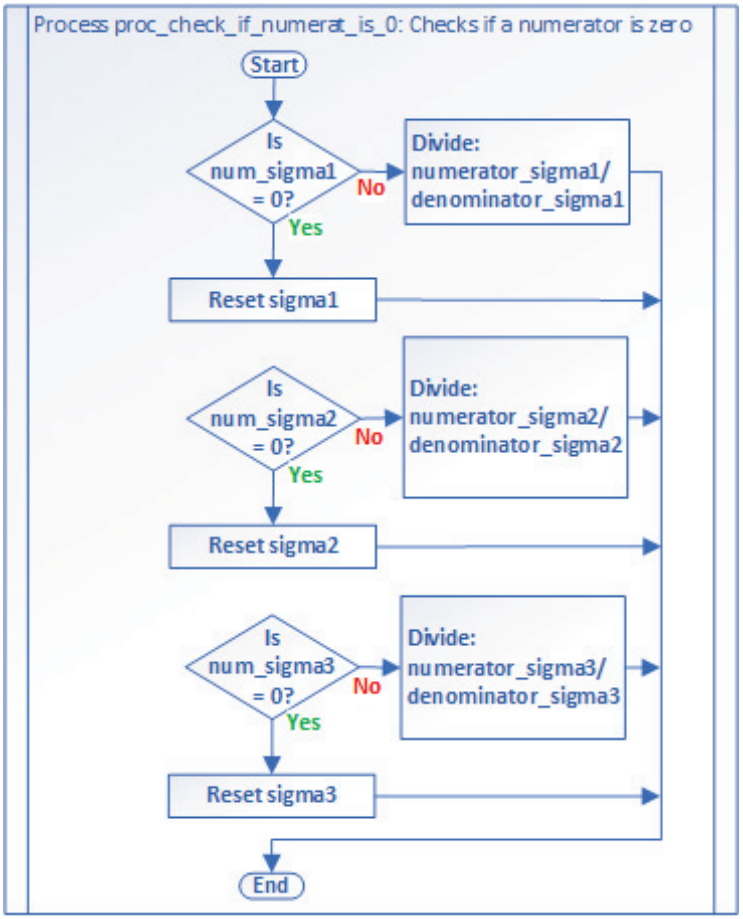

Figure 5.9 - $proc\_check\_if\_numerat\_is\_0$ which verifies if numerators are null, due to hardware issues.

5.3.3 Schematics and Synthesis

The code was developed using the Very High-Speed Integrated Circuit Hardware Description Language (VHDL) on the Xilinx Integrated Development Environment, Vivado, which is a well-known, well-supported hardware

descriptive language. As a note, hardware descriptive languages do not normally compile like software for CPUs/DSPs, which create an object file and the linker then translates it to opcodes or instructions for the CPU. A hardware descriptive language infers the VHDL user instructions into Configurable Logic Blocks (CLBs) or cells of the FPGA, with the best translation it can. These blocks are composed by registers or individual flip-flops, which work commanded by a clock, and by lookup tables (LUTs) where are implemented the combinatorial logic. Blocks are interconnected by programmable routes, while the FPGAs Input/Output (I/O) signals are being taken care by special blocks. Figure 5.10 shows the layout of an FPGA.

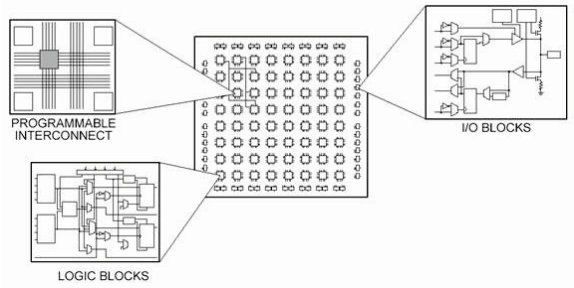

Figure 5.10 – Layout of an FPGA.

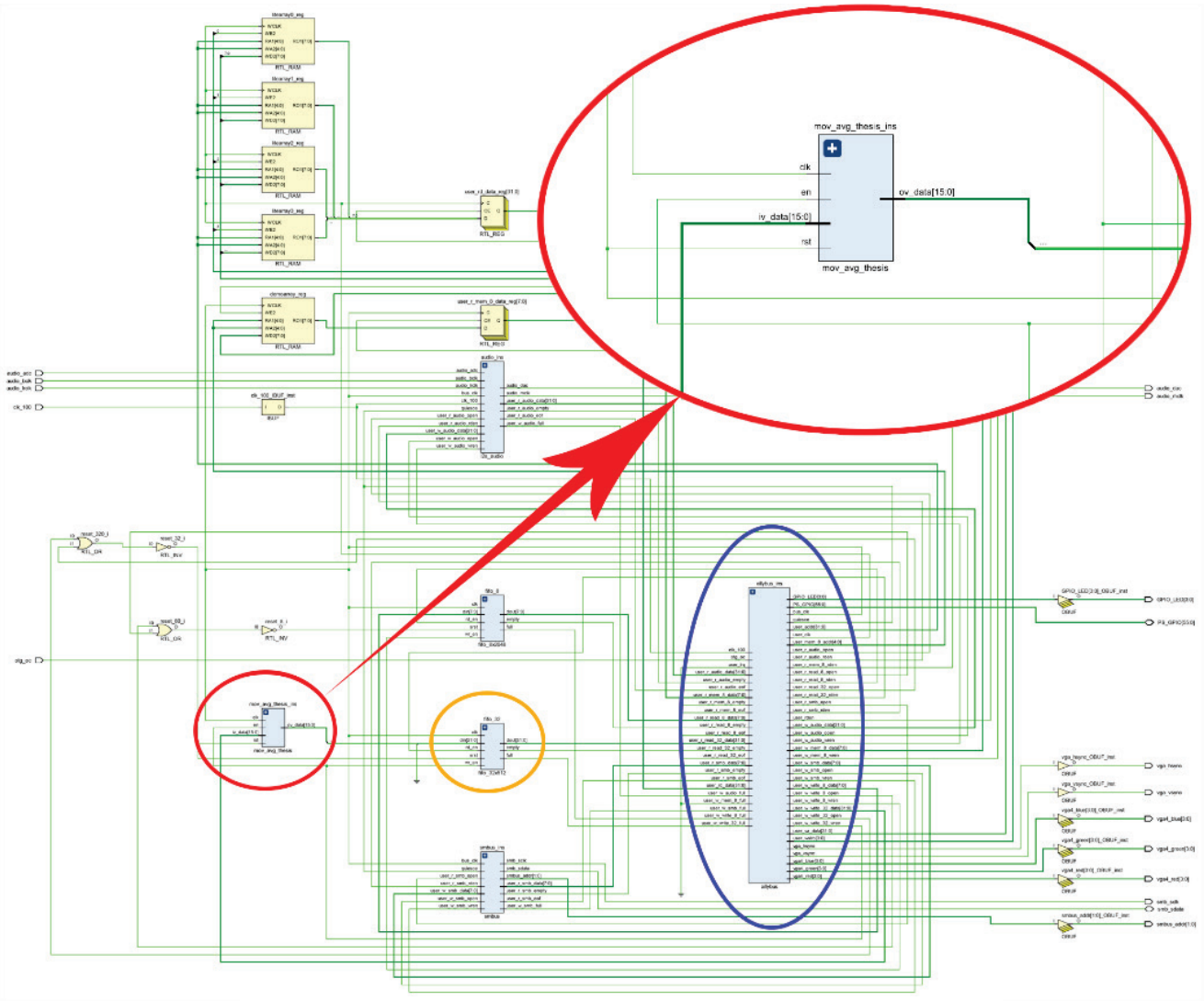

Figure 5.11- Moving Average module connected to the Xillybus HDL platform.

The developed module was connected to a 32-bit FIFO module, as shown in Figure 5.11, on the Xillybus HDL platform, covered in Chapter 5.1.

According to Xillybus recommendations, any developed user's modules should be placed before or after an 8-bit or a 32-bit FIFO. This is because the CPU Processing System (PS) and the Programmable Logic (PL) sides have different clock domains, hence need some a buffering solution in order not to loose data.

The Xillybus platform has available, besides the FIFOs, a 32-bit audio module, an 8-bit System Management bus (SMbus) for lightweight communications which is a subset of the serial I^2C protocol and, two banks, one of 8-bit and one of 32-bit RTL RAM memories. All of these modules are connected to the Xillybus core which serves as a bridge between the PS and the PL sides using Random Access Memory (RAM) as data exchange payload, and in turn, this can be a synchronous or an asynchronous procedure.

Inside the *moving_average_inst* VHDL module (the *_inst* suffix means "instance") as it is called is just a conglomerate of FF's and combinatorial logic and it is a descriptive hardware translation of the VHDL code. Figure 5.12 shows a glimpse of the referred module.

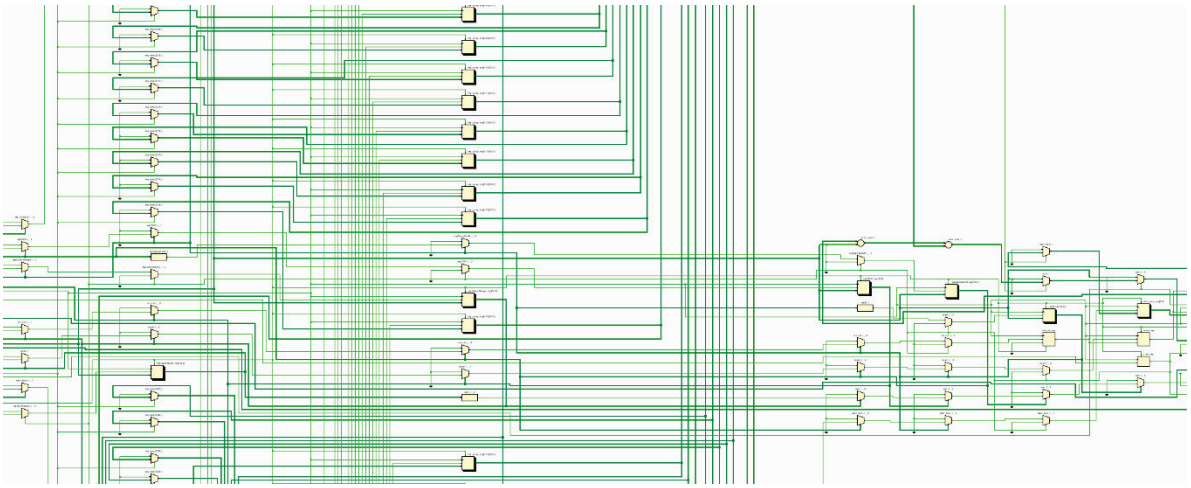

Figure 5.12 – Arbitrary section of the generated *moving_average_inst* module.

5.3.4 Timing diagram

On Vivado, the Hardware Description Language (HDL) development environment of the Xilinx enterprise, the manufacturer of the SoC IC used in our concept-to-chip path Register-Transfer Level (RTL) analysis was made, in order to test its functionality. This is where some racing conditions may occur due to timing issues, triggered, for example, by the asynchronous functionality of the set and reset which is an important consideration for the signaling stability. In

this RTL phase, attributes are generated such Maximum Fanout, Route Behavior and Debug Marks insertion points, along with Clocking, Multipliers, DSPs, Netlists RAM inferences as well as Cross Domains clocking, which in our case reveals of critical importance since the SoC has different clocks, either in the CPU and in the FPGA domains. Figure 5.11 shows an implementation fragment of one of the detailed testing scenarios shown by the Figure 3.16, of the Chapter 3.A.

After testing the circuit's functionality by series of simulations, some timing adjustments, especially the signaling of the counter resets (shown in red circles) revealed to be a critical problem. They are outplaced relative to where the true count reset should have been occurred (orange "eye" shapes and arrows), as can be seen in Figure 5.13.

In other words, the circuit behaves as expected because the filter is reset on the correct places and in fact, the result outputs averaging values when the neighbor pixel falls inside $\mu \pm \sigma$, otherwise it resets the counter and hence restarts a new averaging. However, the counter reset flag is misplaced.

It is suspected this is due to hardware constraints, since these flags, which have a period of 3 ns have less than a clock period which is 10 ns. We wanted to have results coming out of the module immediately after the first pixel, a condition that is verified, however, not all hardware platforms can deal with this tide timings and as a consequence they should be tweaked in order to meet each brand and model constraints. Pulses behavior due to their sub-clock period was not predicted and they should be treated with relaxed timings, that is, they should propagate without timing constraints until output.

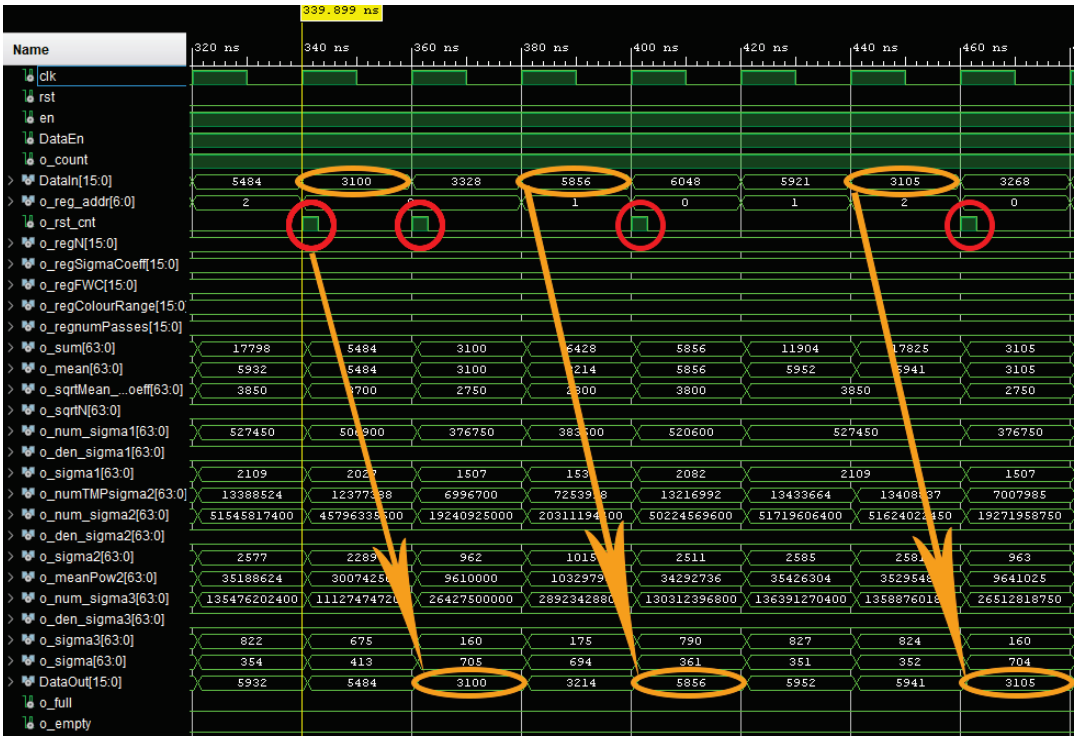

Figure 5.13 – Timing diagram fragment of the testing scenario.

5.3.5 Implementation

After Synthesis, it follows the Implementation stage, where the FPGA's blocks are configured according to the VHDL source code translation made on the Synthesis stage. There, it is calculated the utilization resources of the PL side, shown in the top half of the Figure 5.14, including the power dissipation on bottom half of the same figure. It should be mentioned that FPGAs have a high energy efficiency despite their specific timing and signal structuring constraints, compared to, for example, to 9^{th} generation $i7-9700$ Intel CPU, which has a power dissipation of 65 Watt.

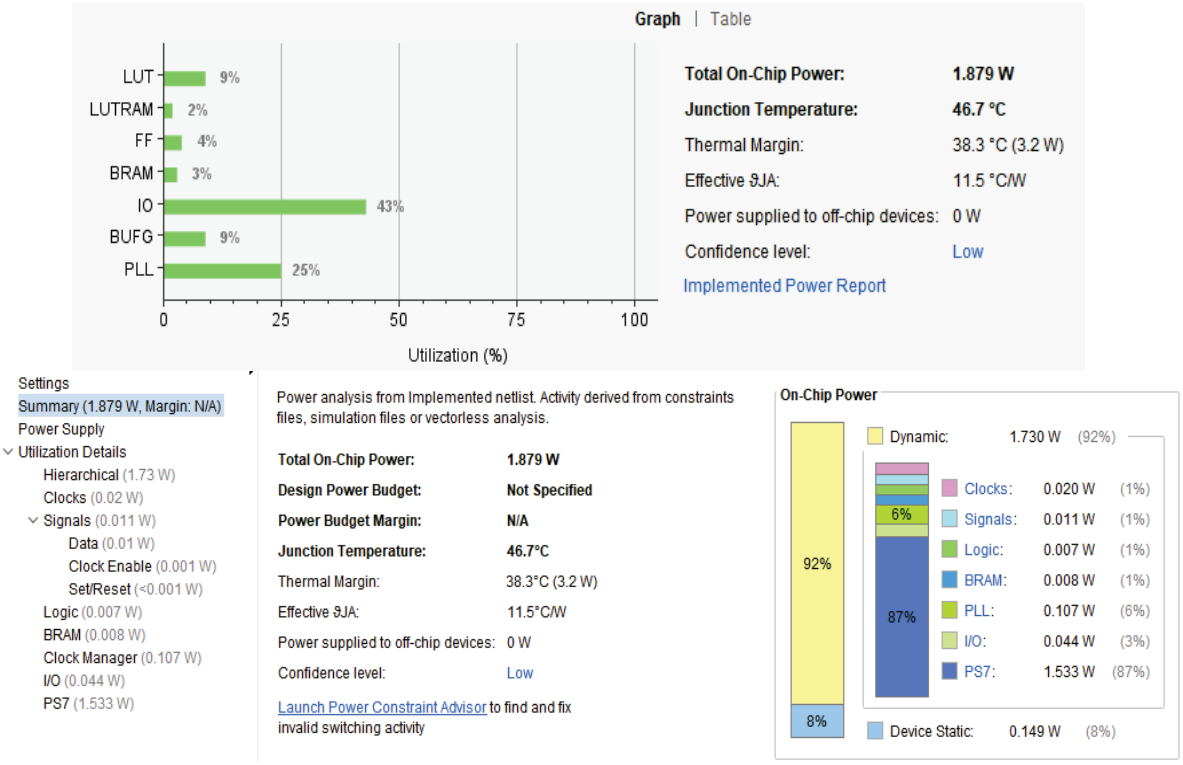

Figure 5.14 – On top half, the resources utilization of the FPGA. On bottom half, the power dissipation of each one of those resources

5.3.6 Bitstream

A bitstream generation is the FPGA resulting hardware configurable file that is to be uploaded upon the operating system boot. This bitstream file, is generated after the completion of the simulation, the synthesis and the implementation phases and it sets the functional logic and sequential gates, the internal routing and maps the external I/O connectivity. Figure 5.15 shows the resulting bitstream mapping for the Zynq-7000 SoC.

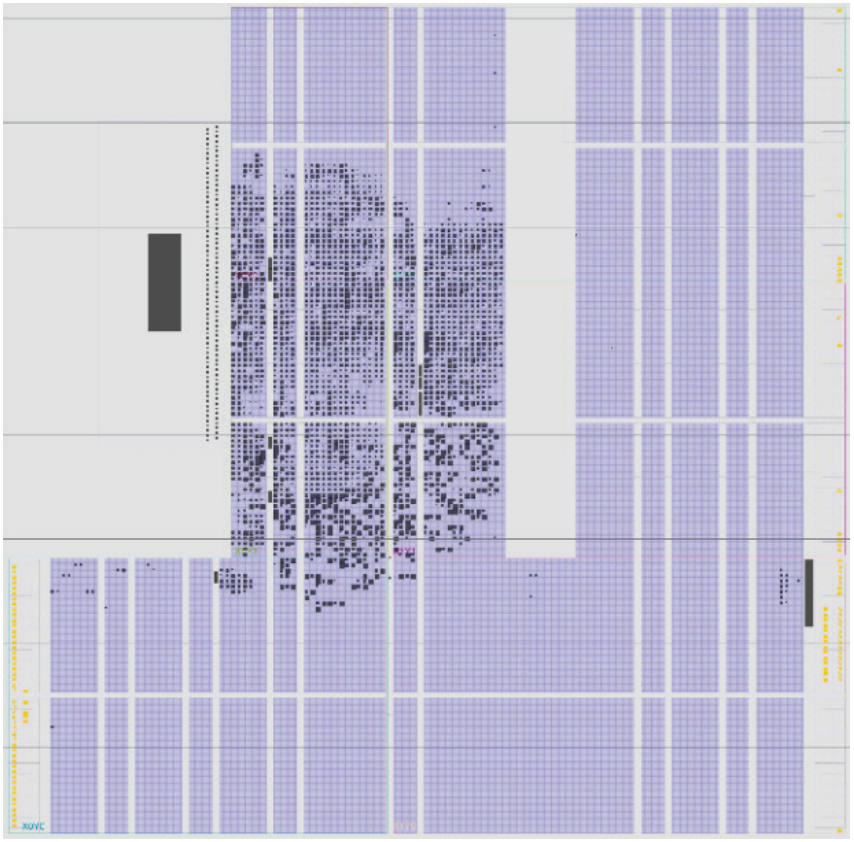

Figure 5.15 – FPGA gate and routing activation of the Moving Average algorithm.

Every time any change or new functionality needs to be implemented; it must pass through the aforementioned phases until the resulting bitstream file with the *bin* extension which must be saved to the SD card to be booted along with the operating system. FPGA's programmable logic is now ready to receive/ send data from/to CPU as explained in Section 5.2 through DMA access synchronous or asynchronously.

5.4 CPU host applications

On the CPU side, 2 separate C language developed programs, one named *streamwrite_32* and the other named *streamread_32.*

The foremost opens the image file stored on the SD card, reads the first two words, the image's height and width in order to keep tracking of the number of pixels' lines and columns. It then feeds the FPGA's 32-bit FIFO with the remaining of the header, shown in Figure 5.16, along with the image's pixels information. In order to be processed in FPGA, these images are first transposed in MATLAB, that is, lines with columns. This is because the chosen moving average filtering is a temporal filter and the lines were acquired from a line sensor. In this sense, the sensor acquires one line at each moment, passes it to the FPGA and immediately get the next line. The latter program,

streamread_32, reads from the command line parameters passed to it, starting by the image's width and height and then stores processed pixels coming from the FPGA's FIFO into a file, again column by column.

The processed image is transposed again in MATLAB and then formatted into a more popular picture file format.

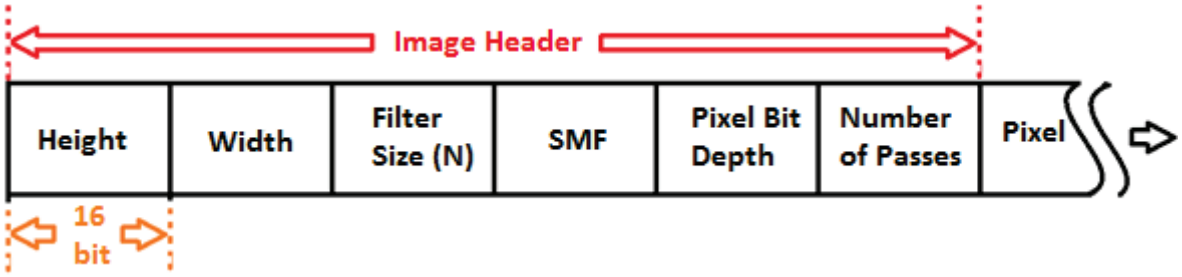

Figure 5.16 - Image header structure

Relatively to the s*treamwrite_32* program, following the already referred dimension words, width and height, each of the remaining words are stuffed with one null word, so becoming 32-bit and then streamed to the FPGA. This stuffing is the most sensible in order to avoid adding unnecessary logic of unpacking 2×16-bit out of a 32-bit, processing the first 16 LSB's while holding the others 16 MSB's for the next clock.

This would make the FPGA's result exiting at half of the input's clock rate. Since there are only two FIFOs on the supplied Xillybus platform, one of 8-bit and other of 32-bit, the latter one was picked, even though with half of the bandwidth used. Speed is not a critical factor here.

From the CPU side, 32-bit are sent to the 32-bit sized FIFO (MSB's are null). From these, the same 32-bit are debited but only 16 LSB bits are routed to the *moving_average_inst* module.

On the other side, s*treamread_32* receives, at a time, the same 32-bit, column by column, but again, only 16 bits have meaningful data. The header, which is passed from the FPGA is saved to the image file untouched, i. e, without suffer any processing. All the other data is saved to a memory buffer and at the end this data suffers a transpose similar to a matrix transpose operation, which accordingly to what was already stated, due to the data being received column sequenced.

The CPU side programs *streamwrite_32* and *streamread_32* only need to keep track the images' sizes and, without any data change, the aforementioned program sends pixels' column sets while the latter program receives them processed from the FPGA.

The flowchart represented in Figure 5.17, shows the sequence of explained actions taken by the *streamwrite_32* C program.

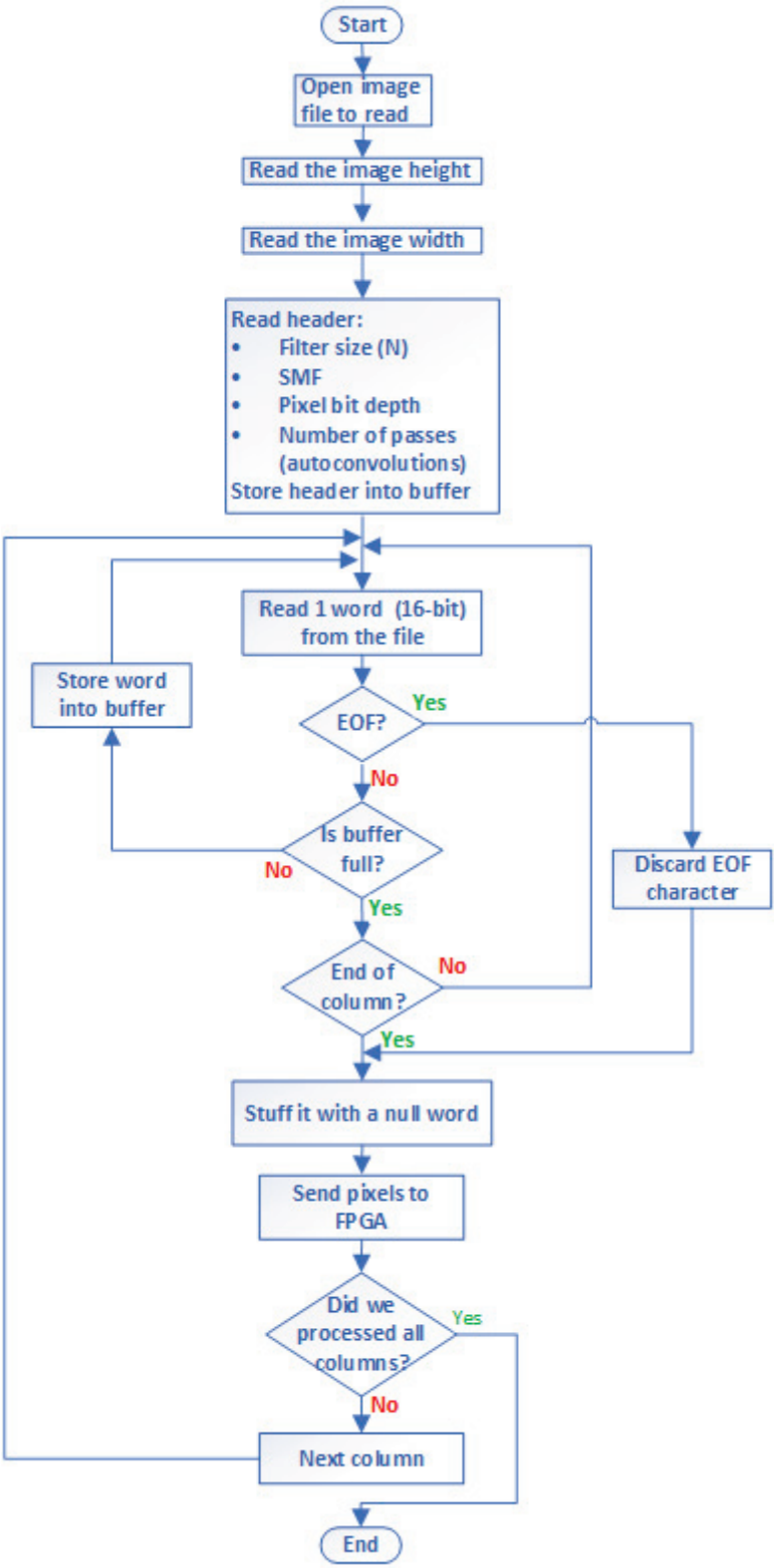

Figure 5.17 *– Flowchart of the streamwrite_32 C program.*

It should not be necessary an additional flowchart for the FPGA

processed data, since the sequence of actions are basically the in reverse order of those stated in Figure 5.17.

5.5 Conclusions

For versatility, online sheer amount of support and the convenience to have a CPU and an FPGA in one chip (SoC) a ZedBoard Zynq-7000 was chosen in order to develop, test and simulate the algorithms developed in this dissertation. However, with some additional work, the same results could have been obtained using solely one FPGA chip.

Another advantage of this board is the utilization of a Linux version, making this a fully testing platform. For one side, the CPU is available on the same platform which can be programmed with any convenient data feeder and collector code in order to support the FPGA's main algorithms. On the other side, from the CPU perspective, the FPGA is an accelerator that can be seen analogically as a sound or graphics accelerator, which can be accessed through a range of RAM memory addresses.

This paradigm greatly simplified the development and the testing processes, particularly on data transportation between the CPU and the FPGA. To sum up and the way the whole process behaves, to the CPU, the FPGA is a file which can be written to and read from. To FPGA, data is a stream that enters the IP modules, is processed and exits to memory.

In order to avoid additional programming complexity, MATLAB was used, especially on the image transpositions before sent to and after received from the testing platform in order to format the image according to a temporal sensor reading idiosyncrasy, and also to insert in each "column" (which is read as a line) the needed header for the FPGA algorithm. This way, the testing images' format were tailored as a convenient raw processing format.

As seen in the Chapter 5.3.4 -Timing diagram, it was possible to obtain a processing rate of pixel per clock also with a latency of 1 clock. However, since these algorithms are relatively low in computational expenditure it is expected to obtain such latencies. However, these should increase proportionally for more complex processing demands.

In all, the whole results came up near to what was expected, notwithstanding some temporal latency in data processing. However, since timing is not an issue, mitigation time to this were not delved.

6. General Conclusions

The objective of this dissertation was initially, to develop a denoising algorithm, using an FPGA as processing unit. Afterwards, on a sequence of an invitation for an event, it surged a second algorithm which was used as means of comparison to the first one. Both solutions would have to be computationally simple and speedy, targeting real-time images acquisitions and processing. This would require managing all the computations to be strictly in the time domain. Therefore, transforms to and from frequency domain would have to be excluded.

The two chosen solutions, whose work resulted in a published paper each, were, a *Moving Average filter*, transcript in the Chapter 3 and a *Savitzky-Golay filter*, transcript in Chapter 4. The research done to the state-of-art up until to the development decision of the first solution, leaded to conclude that quickness would arguably be the one with the required characteristics, that is, one of the most, if not the computational simplest and quickest solution. To compare its results, the Savitzky-Golay filter was opted.

Basically, the working principles of each's one filters are, first, the moving average filter builds up an average of adjacent pixel values. The condition for an adjacent pixel, to enter on that average is stated in Equation 3.8 and 3.9. The number of pixels has a limit which equals to the size of the filter N. The filter resets or restarts a new averaging buildup when the adjacent pixel is out of the range of the referred equations.

The Savitzky-Golay filter, in turn, instead of being a filter with a constant average value, as is the moving average, it is built by adjacent pixels which follows a Minimal Mean Square Errors shape adaptative regression. This approach reveals better behavior particularly in high frequencies preservation.

Even though being a simple time-domain filter, the moving average, when properly scaled to the resolution of the image that will be applied on, it can give comparable results to some other algorithms which require transitions between time-space and frequency domains. Additionally, it does an outstanding job in preserving the image's sharp color transitions.

Before implementing the algorithms in VHDL code, a MATLAB pseudorandom generator function was used to add synthetically noise variances of $\sigma^2 = 1 \times 10^{-4}$ and $\sigma^2 = 1 \times 10^{-3}$ to high-resolution images acquired with a high-quality line sensor. Thus, before this acquisition procedure, the image may be assumed to be virtually noise free. On Chapter 3, Table 3.1 show results with these noise variances.

Considering that the variance $\sigma^2 = 1 \times 10^{-3}$ generates perceptibly more noise to an image than the variance $\sigma^2 = 1 \times 10^{-4}$ does, the latter was chosen

as a reference comparatively to the former for two reasons. First, it is unlikely that an high quality sensor such as the one used to capture the testing images may generate levels of noise variance that equals $\sigma^2 = 1 \times 10^{-4}$, even if those images where to be acquired by a decade year old sensors found on portable cameras and smartphones. Second, it reveals to be a pondered tradeoff between the noise-free image and their processed counterparts. This noise imputed process was done for both filters' solutions.

The column *Change*, in Table 3.1, shows the percentual differences comparison made between MATLAB's floating-point version, taken as reference, and the FPGA fixed-point calculations, including the SNR and PeakSNR logarithmic parameters which were meanwhile converted to decimal.

Floating-point calculations were all done in the value's range between $[0..1]$, since it holds the most precision scale. That is, the maximum machine epsilon floating-point rounding error is 2^{-52} in a scale of binary values between 2^n and 2^{n+1}, when $n = 0$. Because this naturally adds the least error to the final calculations, comparatively to fixed-point, this was the reason they were chosen as reference. Meanwhile, because fixed-point calculations cannot be done in this range, since it would result in zero, those remained in the range $[0..10 \times 10^3]$, as already stated.

It can be verified that $\sigma^2 = 1 \times 10^{-4}$ tests reveal better results than $\sigma^2 = 1 \times 10^{-3}$, with the highest values being at $N = 9$ and $SMF = 50$, while the lowest being at $N = 16$ and $SMF = 100$. SNR and PeakSNR depend inversely on MSE as given by Equations 3.1, 3.2 and 3.3, and Global Similarity follows, as well, this trend. The lower the MSE, the better the SNR and better the match between the Similarity of the original and the processed images.

In general, a noise variance of $\sigma^2 = 1 \times 10^{-4}$ have better results when done with $SMF = 50$ signaled filters. Likewise, the range of N between 4 and 25, have progressive degrading results among the same value of SMF. The exception to this is for $N = 25$, which accomplishes better outcomes for fixed-point in both SMF's fixed- and floating-point results. Image's results reveal that progressive higher quality is achieved with $N = 16$ and $SMF = 100$.

For $N = 25$, images show some adjacent color blurring, resembling pixeled blocks and thus start to lose quality. With $SMFs = 50$, SNR's magnitude orders of $31\ dB$ and PeakSNR's of $37\ dB$ seem acceptable for this level of noise variance of $\sigma^2 = 1 \times 10^{-4}$, whilst with $SMFs = 100$, lower SNR's of 28 or $29\ dB$ and PeakSNR's of $35\ dB$ seem fair, because within the same N frame size filtering, the higher the SMF the more the high-range frequencies which are wiped out along with the noise. This is even more perceptible in these frequency ranges due to loss of the image detail. However, visually, it looks like there is an

additional noise reduction, in the latter SMFs. Not withstand, image degrades because information details are also lost. On the limit, a very high SMF would lead to an image built with only contour patches.

We have made an additional comparison test, which consisted in a simple moving average image noise reduction, without sharp color detection and it was obtained SNR values between 26 and $19\,dB$, hence significantly lower than those of our interest. It is also noticeable that floating-point results are consistent with fixed-point ones.

The differences between both are minor but suffice to conclude that the floating-point, due to a higher precision, generate better results. In relation to $\sigma^2 = 1 \times 10^{-3}$, the best values are obtained when $N = 4$ and $SMF = 100$. There is, however, a significant difference: while better SNR results are obtained with $SMF = 50$ rather than with $SMF = 100$ for a variance of $\sigma^2 = 1 \times 10^{-4}$, which accounts with differences between 2 and $3\,dB$, relatively to $\sigma^2 = 1 \times 10^{-3}$, they are practically the same, around $SNR = 24\,dB$. Considering that $\sigma^2 = 1 \times 10^{-3}$ generates perceptible noisier images, this is an acceptable result.

The most meaningful conclusion here is that it is possible to make calculations in a simple fixed-point notation without any visible difference compared to a more computationally expensive floating-point counterpart, once a relative high pixel bit depth is chosen, which makes the chosen platform very suitable for this type of processing.

Our tests were made with images containing pixels with 10×10^3 colour tones, i.e., a little over $2^{13} = 8192$ shades. This further reduces roundoff errors to a level of precision of $\frac{1}{10 \times 10^3} = 1 \times 10^{-4}$, due to the fact that calculations made in fixed-point always ditch the rational part by flooring down the result to the immediate lower integer available.

Additionally, attention must be given to the whole mathematical operation, tracking specifically where divisions and square roots occur. It is needed to exert some effort to reduce to the minimum the number of square roots and divisions operations. The developed algorithm has a maximum of one square root and a maximum of one division operation. It is a necessary task since round offs only occur on these math operations.

There is yet another feature that needs to be treated properly. A noise contaminated image is more noticeable at darker colors or shades than lighter ones and this deserves special care. A non-linear smoothing filtering function must be adjusted and multiplied by σ, letting darker color shades have more filtering tolerance than the lighter ones.

A quick note on the results: the sparse difference between the

measurement parameters does not quite really translates the visual perception of the processed denoised results. Furthermore, the different filter sizes and the multiplicative sigma factor can make a huge contribution when the image has a large solid patch shades to be processed.

Comparatively with a moving average filter, the Savitzky-Golay filter generally gives better results as can be seen in Table 4.1, in 4.A-Appendix. several degrees of freedom to be exploit, and it is important to say that, due to the polynomial nature of the Savitzky-Golay filters, each new derivative can reveal further information not immediately seen in the data being smoothed. This is a characteristic not directly comparable with a constant average value filter such as the moving average filter. It is possible to infer from the resultant figures, that the best results are achieved by S-G filters, those on the left columns.

The final inferences can be the result of some biased comparison decisions. It should be said that some of these tests were done using as source, computerized generated synthetic images, which can, most of the time, resemble real-world pictures due to its tradeoff in smooth color changes and sharp color transitions.

Comparison between floating- and fixed-point calculations present a minimal gap due to the use of 64-bit registers. This is possible because the registers are much larger than the images bit depth resolutions and this way it is possible to reduce the round-off errors. Consequently, as a rule of thumb, the chosen registers' sizes which make possible these computations should be, at least, the quadruple of the highest bit-depth size of the picture rounded up to the nearest multiple of 8. In this case, the 13-bit depth images used for the tests were round up to 16-bit and all the math done on 64-bit registers.

An inverse scaling polynomial function of low order degree was created, modeled and adjusted as a post-filter to the application of the S-G filters to further smooth the noise in darker colors relatively to the lightest ones, because the noise is more perceptive in dark tones.

This work shows promising results on low power, low complexity, high portability, FPGAs utilization for real-time signal processing.

For future works, we aim to make use of a more challenging approach in frequency domain, including Discrete Cosines (DCT), Fourier and Wavelet transforms. This was one also of the reasons to include it as a reference on Section 3.2.

7. References

[1] Sensitivity and Image Quality of Digital Cameras, Dirks, Friedrich 10.1.1.85.7059.

[2] Image “Garden” in Bayer pattern https://www.cameradebate.com/wp-content/uploads/2015/05/bayer-image-garden.jpg

[3] Alparone L, Selva M, Aiazzi B, Baronti S, Butera F, Chiarantini L, Signal-dependent Noise Modelling and Estimation of New Generation Imaging Spectrometers. 978-1-4244-4687-2/09/$25.00 ©2009 IEEE

[4] Argenti F., Tizano B., Alparone L., Multiresolution MAP Despeckling of SAR Images Based on Locally Adaptative Generalized Gaussian pdf Modeling, 1057-7149/$20.00 © 2006 IEEE.

[5] Argenti F., Torricelli G., Alparone L., MMSE filtering of generalized signal-dependent noise in spatial and shift-invariant wavelet domains, 1057-7149/$20.00 © 2006 IEEE.

[6] Hara K., Horiguchi T., Kinoshita T., Highly efficient photon-to-electron conversion with mercurochrome-sensitized nanoporous oxidesemiconductor solar cells, Elsevier, Solar Energy Materials & Solar Cells 64 (2000) 115-134.

[7] DeValois R. L., DeValois K., Spatial Vision, Oxford University Press, 1990, pp. 212-225.

[8] Varadharajan S, Spatial Vision and Pattern, Chapter 2.1.5, DOI 10.1007978-3-540-79567-4_2.1.5.

[9] Oppenheim A.V., Schafer R.W. (1999) Discrete-Time Signal Processing. Prentice-Hall, Englewood Cliffs, NJ. pp. 65-70.

[10] Donoho D. L., De-noising by soft-thresholding, IEEE Trans. Inform. Theory, vol. 41, pp. 613–627, May 1995.

[11] Klöckner A., Linden, F., Zimmer, D., Noise Generation for Continuous System Simulation, German Aerospace Center (DLR), Institute of System Dynamics and Control, DOI: 10.3384/ECP14096837.

[12] Nyquist, H., Certain topics in telegraph transmission theory, *AIEE Trans*., 47: 621–637, Jan. 1928.

[13] Kalman R.E. (1960) A New Approach to Linear Filtering and Prediction Problems. Trans. of the ASME, Series D, *Journal of Basic Engineering*, 82.

[14] Wilsky A.S. (1979) Digital Signal Processing, Control and Estimation Theory: Points of Tangency, Areas of Intersection and Parallel Directions. MIT Press, Cambridge, MA.

[15] Davenport W.B. and Root W.L. (1958) *An Introduction to the Theory of Random Signals and Noise*. McGraw-Hill, New York.

[16] Therrien C.W. (1992) Discrete Random Signals and Statistical Signal

Processing. Prentice-Hall, Englewood Cliffs, NJ.

[17] Bennett W.R. (1960) Electrical Noise. McGraw-Hill. New York (1960).

[18] Awad A, Denoising images corrupted with impulse, Gaussian, or a mixture of impulse and Gaussian noise, Elsevier, 2019, https://doi.org/10.1016/j.jestch.2019.01.012.

[19] Karp S., Clark, J. R., "Photon Counting: A Problem in Classical Noise Theory" Electronics Research Center, Cambridge Mas., NASA, https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19700014758.pdf.

[20] Foi A., Katkovnik V., Egiazarian K., Pointwise shape adaptive DCT for high-quality denoising and deblocking of grayscale and color images, IEEE Transactions on Image Processing 16 (5) (2007) 1395.

[21] Jean-Luc Starck J-L, Fadili J, Murtagh F, The Undecimated Wavelet Decomposition and its Reconstruction, 1057-7149/$25.00 © 2007 IEEE.

[22] Malek M., Helbert D., Carré P., "The Color Graph-based Wavelet Transform with Perceptual Information" (Oct 20, 2015), Cornell University Library, https://arxiv.org/pdf/1510.05436.pdf.

[23] Deledalle C.-A., Denis L., Iterative Weighted Maximum Likelihood Denoising With Probabilistic Patch-Based Weight, 1057-7149/$26.00 © 2009 IEEE.

[24] Schafer, Ronald W, On the Frequency-Domain Properties of Savitzky-Golay Filters, HP Laboratories-2010-109.

[25] Kailath T. (1980) *Linear Systems*. Prentice Hall, Englewood Cliffs, NJ.

[26] Yuen M., Wu H., A survey of hybrid MC/DPCM/DCT video coding distortions, Signal Processing 70 (3) (1998) 247–278.

[27] Smith, S. W., "The scientist & Engineer's Guide to Digital Processing", Chapter 15, (1997-12-24).

[28] Dong G., Zhichang G., Wu B., A Convex Adaptive Total Variation Model Based on the Gray Level Indicator for Multiplicative Noise Removal, DOI: 10.1155/2013/912373.

[29] Claerbout Jon, Image Estimation by Example: Geophysical Soundings Image Construction, Multidimensional autoregression, Stanford University, http://sepwww.stanford.edu/sep/prof/gee/mda/paper_html/node1.html.

[30] Sikora T., Low complexity shape-adaptive DCT for coding of arbitrarily shaped image segments, Signal processing: Image communication 7 (4–6) (1995) 381–395.

[31] Gilge M., Engelhardt T., Mehlan R., Coding of arbitrarily shaped image segments based on a generalized orthogonal transform, Signal Processing: Image Communication, Vol. 1, No. 2, October 198.

[32] Chen Q., Wu D., "Image denoise by bounded block matching and 3D filtering" Elsevier, vol. A247, pp. 529–551, October 2009.

[33] Dabov K., Foi A., Katkovnik V., Egiazarian K, et al., Image denoising by

sparse 3-D transform-domain collaborative filtering, IEEE Trans- actions on Image Processing 16 (8) (2007) 2080–2095.

[34] Buades A., Coll B., Morel J., A review of image denoising algorithms, with a new one, Multiscale Modeling and Simulation 4 (2) (2006) 490–530.

[35] Jardim R., Morgado-Dias F., Image Denoise FPGA Implementation using a Moving Average Filter with Contour Detection, July 2018, DOI: 10.1109/ICBEA.2018.8471740, Conference: 2018 International Conference on Biomedical Engineering and Applications (ICBEA).

[36] Nascimento I., Jardim R., Morgado-Dias F., A new solution to the hyperbolic tangent implementation in hardware: polynomial modeling of the fractional expon. Part, Neural Comput. & Applic. (2013) 23: 363. https://doi.org/10.1007/s00521-012-0919-0.

[37] Xillybus host application programming guide for Linux, documentation site (Ch 4-5) http://www.xillybus.com/downloads/doc/xillybus_host_programming_guide_linux.pdf.

[38] Xilinx Zynq-7000 System-On-Chip (SoC) series manufacturer webpage, https://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png.

[39] ZedBoard™ Getting Started Guide, Version 7.0, page 10, http://zedboard.org/sites/default/files/documentations/GS-AES-Z7EV-7Z020-G-V7-1.pdf.

[40] Xillybus FPGA designer's guide, documentation site (Ch. 2-3-4) http://www.xillybus.com/downloads/doc/xillybus_fpga_api.pdf.

[41] Li Y, Chu W, A New Non-Restoring Square Root Algorithm and Its VLSI Implementations, International Conference on Computer Design (ICCD'96), October 7–9, 1996, Austin, Texas, USA.

[42] Gauss K.G. (1963) *Theory of Motion of Heavenly Bodies*. Dover, New York.

[43] Cox I. J., Miller M.L., Bloom J.A., Fridrich J. and Kalker T. (2008) *Digital Watermarking and Steganography*, 2nd edit, Morgan Kaufmann.

[44] Kung S.Y. (1993) *Digital Neural Networks*. Prentice-Hall, Englewood Cliffs, NJ.

[45] Vaidyanathan P.P. (1993) *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

[46] Askari, G., Motamedi N., Karimian M., Sadeghi H, Design and Implementation Of an X-Band White Gaussian Noise Generator, Canadian Conference on Electrical and Computer Engineering, June 2008, IEEE, DOI: 10.1109/CCECE.2008.4564708.

[47] Goossens B., Pižurica A., Philips W., Removal of Correlated Noise by Modeling the Signal of Interest in the Wavelet Domain.

A Appendices

NOTE: Some code was commented because does not relate with the code that is active, either in functionality or testing conditions or both.

A.1 Moving average FPGA source code

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 use ieee.numeric_std.all;
5
6 library work;
7 use work.pkg_math.all;
8
9
10 -- Uncomment the following library declaration if instantiating any Xilinx leaf cells in this code.
11 --library UNISIM;
12 --use UNISIM.VComponents.all;
13 entity mov_avg_thesis is
14 generic (
15 COUNTER_DEPTH:integer := 7; --counter will need to count until 100 for N maximum number for averaging,
hence 2^7=128 which is enough
16 G_DATA_W :integer := 16
17 );
18
19 port (
20 clk :in std_logic;
21 rst :in std_logic;
22 en :in std_logic;
23 o_count :buffer std_logic;
24 iv_data :in std_logic_vector(G_DATA_W-1 downto 0);
25 o_reg_addr :buffer std_logic_vector(COUNTER_DEPTH-1 downto 0);
26 o_rst_cnt :buffer std_logic;
27 o_regN :buffer std_logic_vector(G_DATA_W-1 downto 0);
28 o_regsigmaCoeff :buffer std_logic_vector(G_DATA_W-1 downto 0);
29 o_regFWC :buffer std_logic_vector(G_DATA_W-1 downto 0);
30 o_regcolourRange :buffer std_logic_vector(G_DATA_W-1 downto 0);
31 o_regnumPasses :buffer std_logic_vector(G_DATA_W-1 downto 0);
32 o_reg0 :buffer std_logic_vector(G_DATA_W-1 downto 0);
33 o_reg1 :buffer std_logic_vector(G_DATA_W-1 downto 0);
34 o_reg2 :buffer std_logic_vector(G_DATA_W-1 downto 0);
35 o_reg3 :buffer std_logic_vector(G_DATA_W-1 downto 0);
36 o_reg4 :buffer std_logic_vector(G_DATA_W-1 downto 0);
37 o_sum :buffer std_logic_vector(G_DATA_W*4-1 downto 0); -- NOTE: for safe we reserve
inner math operations
38 o_mean :buffer std_logic_vector(G_DATA_W*4-1 downto 0); -- to 4*data_in size,
although 2*datata_in would be
39 o_sqrtMean_x_sigCoeff:buffer std_logic_vector(G_DATA_W*4-1 downto 0); -- suffice
40 o_sqrtN :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
41 o_num_sigma1 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
42 o_den_sigma1 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
43 o_sigma1 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
44 o_numTMPsigma2 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
45 o_num_sigma2 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
46 o_den_sigma2 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
47 o_sigma2 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
48 o_meanPow2 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
49 o_num_sigma3 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
50 o_den_sigma3 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
51 o_sigma3 :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
52 o_sigma :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
53 ov_data :out std_logic_vector(G_DATA_W-1 downto 0)
54 );
55 end entity;
56
57
58
59 architecture mov_avg_thesis_rtl of mov_avg_thesis is
60
61 constant REG_SIZE :integer := 2**COUNTER_DEPTH; -- this is the filter size constant. it is set
for 128 but we will only need 100, which is the maximum filter size
62 signal start_from :std_logic_vector(COUNTER_DEPTH-1 downto 0); -- connected to the input port
of the counter. used to start the counter with any arbitrary number
63 signal reg_addr :std_logic_vector(COUNTER_DEPTH-1 downto 0); -- connected to the output Q
port of the counter
64 signal idx_reg_addr :integer range 0 to REG_SIZE-1; -- integer index translator of the
std_logig_vector of the previous reg_addr
65 type type_reg_array is array(0 TO REG_SIZE-1) of unsigned(G_DATA_W*4-1 downto 0); -- this is the filter
size. it is set for 128 registers of 64-bits, but we will only need 100, which is the maximum filter size
66 signal reg_array :type_reg_array; -- array of registers for filter previously declared
67
68 signal s_en :std_logic;
```

```
69 signal count :std_logic;
70 signal rst_cnt :std_logic;
71
72 signal regN :unsigned(G_DATA_W-1 downto 0); -- reg for storing the filter length N to
calculate sigma = sqrt(mean) / sqrt(N). because N comes already squared, so possible values are 2, 3, 4,
5, 6, 7, 8 , 9 and 10, then sigma = sqrt(pixel) / N,
73 signal regSigmaCoeff :unsigned(G_DATA_W-1 downto 0); -- reg for storing multiplicative sigma
coefficient. Can be 50 or 100 for testing
74 signal regFWC :unsigned(G_DATA_W-1 downto 0);
75 signal regColourRange :unsigned(G_DATA_W-1 downto 0);
76 signal regNumPasses :unsigned(G_DATA_W-1 downto 0);
77
78 signal sum :unsigned(G_DATA_W*4-1 downto 0); -- the accumulator to generate the average
of the mean
79 signal mean :unsigned(G_DATA_W*4-1 downto 0); -- by dividing the above sum by N we
calculate the mean
80 signal sqrtMean_x_sigCoeff :unsigned(G_DATA_W*4-1 downto 0); -- pre-multiplies mean x sigmaCoeff (for
efficiency on math operations it will only be needed to make 3 divisions at the end)
81 signal sqrtN :unsigned(G_DATA_W*4-1 downto 0); -- calculates sqrt of N (for efficiency on
math operations this is calculated only once)
82 signal num_sigma1 :unsigned(G_DATA_W*4-1 downto 0); -- numerator for operation 1
83 signal den_sigma1 :unsigned(G_DATA_W*4-1 downto 0); -- denominator for operation 1
84 signal sigma1 :unsigned(G_DATA_W*4-1 downto 0); -- result of the operation 1
85 signal numTMPsigma2 :unsigned(G_DATA_W*4-1 downto 0); -- temporary result for operation 2
86 signal num_sigma2 :unsigned(G_DATA_W*4-1 downto 0); -- numerator for operation 2
87 signal den_sigma2 :unsigned(G_DATA_W*4-1 downto 0); -- denominator for operation 2
88 signal sigma2 :unsigned(G_DATA_W*4-1 downto 0); -- result of the operation 2
89 signal meanPow2 :unsigned(G_DATA_W*4-1 downto 0); -- mean^2
90 signal num_sigma3 :unsigned(G_DATA_W*4-1 downto 0); -- numerator for operation 3
91 signal den_sigma3 :unsigned(G_DATA_W*4-1 downto 0); -- denominator for operation 3
92 signal sigma3 :unsigned(G_DATA_W*4-1 downto 0); -- result of the operation 3
93 signal sigma :unsigned(G_DATA_W*4-1 downto 0); -- final result
94
95 component counter -- declares the counter to count data_in
96 --generic(n: natural := 5);
97 port(
98 clock: in std_logic;
99 clear: in std_logic;
100 count: in std_logic;
101 start_from: in std_logic_vector(COUNTER_DEPTH-1 downto 0); -- tells from which number to start
counting from, instead of the usual zero
102 Q: out std_logic_vector(COUNTER_DEPTH-1 downto 0) -- the out value of the counter
103 );
104 end component;
105
106
107 begin
108 o_count <= count; -- commands the counter to start counting
109 o_reg_addr <= reg_addr; -- for debug: watches the value Q of the counter
110 o_rst_cnt <= rst_cnt; -- reset the clock when outside the moving average mean-sigma or
mean+sigma (when a sharp image coutour is detected)
111 o_regN <= std_logic_vector(regN); -- this stores the size of the filter, which is supplied
in the header of the data_in streamming
112 o_regsigmaCoeff <= std_logic_vector(regSigmaCoeff); -- this stores the sigma multiplicative
factor, which is supplied in the header of the data_in streamming
113 o_regFWC <= std_logic_vector(regFWC); -- this stores the factor wave conversion (num of egenerated
by each photon) which is rough the same as pixel bith depth, which is supplied in the header of
the data_in streamming
114 o_regcolourRange <= std_logic_vector(regColourRange); -- this stores the aka bith depth, which is
supplied in the header of the data_in streamming
115 o_regNumPasses <= std_logic_vector(regNumPasses); -- this stores the number of filter passes, which
is supplied in the header of the data_in streamming
116 o_reg0 <= std_logic_vector(reg_array(0)(G_DATA_W-1 downto 0)); -- the next 5 registers are
only for debug watch
117 o_reg1 <= std_logic_vector(reg_array(1)(G_DATA_W-1 downto 0));
118 o_reg2 <= std_logic_vector(reg_array(2)(G_DATA_W-1 downto 0));
119 o_reg3 <= std_logic_vector(reg_array(3)(G_DATA_W-1 downto 0));
120 o_reg4 <= std_logic_vector(reg_array(4)(G_DATA_W-1 downto 0));
121 o_sum <= std_logic_vector(sum); -- all these variables are for inner math calculations
122 o_mean <= std_logic_vector(mean);
123 o_sqrtMean_x_sigCoeff <= std_logic_vector(sqrtMean_x_sigCoeff);
124 o_sqrtN <= std_logic_vector(sqrtN);
125 o_num_sigma1 <= std_logic_vector(num_sigma1);
126 o_den_sigma1 <= std_logic_vector(den_sigma1);
127 o_sigma1 <= std_logic_vector(sigma1);
128 o_numTMPsigma2 <= std_logic_vector(numTMPsigma2);
129 o_num_sigma2 <= std_logic_vector(num_sigma2);
130 o_den_sigma2 <= std_logic_vector(den_sigma2);
131 o_sigma2 <= std_logic_vector(sigma2);
132 o_meanPow2 <= std_logic_vector(meanPow2);
133 o_num_sigma3 <= std_logic_vector(num_sigma3);
134 o_den_sigma3 <= std_logic_vector(den_sigma3);
135 o_sigma3 <= std_logic_vector(sigma3);
136 o_sigma <= std_logic_vector(sigma);
137
138 counter_inst : entity work.counter generic map(count_depth => COUNTER_DEPTH) -- instantiate the counter
139 port map(clock => clk,
140 clear => rst_cnt,
141 count => count,--gives order to start counter of the leading edge (the
```

```
end) of the filter
142 start_from => start_from, -- tells from which number to start counting
from, instead of the usual zero
143 Q => reg_addr);
144
145 idx_reg_addr <= conv_integer(reg_addr); -- just convert the Q exit from the counter, which is a
std_logic_vector to interger, because indexing is made with intger numbers
146
147
148 proc_calc_mean: process(sum) -- this procedure is to be done in parallell by procedure "proc_calc_sum:
process(clk, rst)" below
149 begin -- for now, this is only done when the size of the filter, N, is 4. TODO:
check for other cases, but at 1st analysis, seems
150 if count = '1' then -- mean <= sum / (idx_reg_addr + frst_rst_made)
151 mean <= sum / (idx_reg_addr + 1);
152 else
153 mean <= (others => '0');
154 end if;
155 end process;
156
157 ov_data <= std_logic_vector(mean(G_DATA_W-1 downto 0));
158 -- below are the variables that run in parallell with the procedure "proc_calc_sum", below the
"proc_check_if_numerat_is_0", as well as with the above procedure "proc_calc_mean"
159 -- it calculates sqrt(mean / (i - M0 + 1)) * (sigmaFactor/FWC*100) * (1.214e-8*FWC^2*mean^2 -
0.0002290*FWC*mean + 1.096), in which:
160 -- (i - M0 + 1) is the maximum length of the filter heading at the index i and tailing at M0
161 -- (sigmaFactor/FWC*100) is the normalized sigma factor (percentage to be scaled in pixel range
between 0..1)
162 -- and (1.214e-8*FWC^2*mean^2 - 0.0002290*FWC*mean + 1.096) is the smoothing polynomial equation
that smooths more in darker colours than in white ones (also normalized)
163 -- all is rearranged in nominators and denominators to equate all in 1 division and 1 sqrt to increase
precision
164 sqrtMean_x_sigCoeff <= x"00000000" & sqrt_Li_Chu(mean(G_DATA_W*2-1 downto 0)) * regSigmaCoeff; -- does
floor(sqrt_mean) * sigma_coeff (floor is implicit on fixed point FPGA maths)
165 sqrtN <= x"000000000000" & sqrt_Li_Chu( x"0000" & regN); -- calculates sqrt(N)
166 num_sigma1 <= x"0089" * sqrtMean_x_sigCoeff(G_DATA_W*3-1 downto 0); -- x"0089"=137; calculates
the sigma1 numerator member: 137 * sqrtMeanXsigmaCoeff
167 den_sigma1 <= sqrtN(G_DATA_W*3-1 downto 0) * x"007D"; --x"007D"=125; calculates the sigma1
denominator member: sqrtN * 125
168
169 numTMPsigma2 <= x"08D1" * mean(G_DATA_W*3-1 downto 0); -- x"08D1"=2257; calculates the sigma1
numerator member: 2257 * mean
170 num_sigma2 <= numTMPsigma2(G_DATA_W*2-1 downto 0) * sqrtMean_x_sigCoeff(G_DATA_W*2-1 downto 0);
-- calculates 2257 * mean * sqrtMeanXsigmaCoeff
171 den_sigma2 <= sqrtN(G_DATA_W*2-1 downto 0) * x"00989680"; --x"989680"=10^7; calculates: sqrtN *
10^7
172
173 meanPow2 <= mean(G_DATA_W*2-1 downto 0) * mean(G_DATA_W*2-1 downto 0); -- calculates mean^2
174 num_sigma3 <= meanPow2(G_DATA_W*2-1 downto 0) * sqrtMean_x_sigCoeff(G_DATA_W*2-1 downto 0); --
x"0089"=137; calculates: mean^2 * sqrtMeanXsigmaCoeff
175 den_sigma3 <= sqrtN(G_DATA_W*2-1 downto 0) * x"04E8E6E3"; --x"04E8E6E3"=82372323; calclates sqrtN
* 82372323
176
177 -- below sre the variables that run in parallell by procedure "proc_calc_sum: process(clk, rst)" below as
do the above procedure
178 proc_check_if_numerat_is_0: process(num_sigma1, num_sigma2, num_sigma3)
179 begin
180 if num_sigma1 = to_unsigned(0, num_sigma1'length) then
181 sigma1 <= (others => '0');
182 else
183 sigma1 <= num_sigma1 / den_sigma1;
184 end if;
185 if num_sigma2 = to_unsigned(0, num_sigma2'length) then
186 sigma2 <= (others => '0');
187 else
188 sigma2 <= num_sigma2 / den_sigma2;
189 end if;
190 if num_sigma3 = to_unsigned(0, num_sigma3'length) then
191 sigma3 <= (others => '0');
192 else
193 sigma3 <= num_sigma3 / den_sigma3;
194 end if;
195 end process;
196 sigma <= sigma1 - sigma2 + sigma3;
197
198
199 proc_calc_sum: process(clk, rst)
200 begin
201
202 if rst = '1' then -- if set, FPGA should expect a stream of data that starts below on
rising_edge(clk), so reset all
203 s_en <= '0';
204 count <= '0';
205 rst_cnt <= '1';
206 start_from <= (others => '0');
207 regN <= (others => '0');
208 regSigmaCoeff <= (others => '0');
209 regFWC <= (others => '0');
210 regColourRange <= (others => '0');
```

```
211 regNumPasses <= (others => '0');
212 for i in 0 to REG_SIZE-1 loop
213 reg_array(i)<= (others => '0');
214 end loop;
215 sum <= (others => '0');
216 elsif rising_edge(clk) then
217 s_en <= en;
218
219 if s_en = '1' then -- starts the streaming of word bytes
220 if regN = to_unsigned(0, regN'length) then -- this is the 1st parameter passed to FPGA, which
is the size of the filter to convolute with the array of pixels
221 regN <= unsigned(iv_data);
222 elsif regSigmaCoeff = to_unsigned(0, regSigmaCoeff'length) then -- the multiplicative sigma
factor (50 or 100, or other)
223 regSigmaCoeff <= unsigned(iv_data);
224 elsif regFWC = to_unsigned(0, regFWC'length) then -- how many electrons are generated by each
photon (normally the same as the maximum image pixel depth
225 regFWC <= unsigned(iv_data);
226 elsif regColourRange = to_unsigned(0, regColourRange'length) then -- maximum color tones
(normally the same as FWC
227 regColourRange <= unsigned(iv_data);
228 elsif regNumPasses = to_unsigned(0, regNumPasses'length) then -- number of passes (convolution
of several passes generates fiferent filters, from rectangular, triangular and different shapes
of gaussian
229 regNumPasses <= unsigned(iv_data);
230 elsif rst_cnt = '1' then -- since rst_cnt has been '1' from the begining of this streaming, set
it now to zero, still this clock, to start a new processiong
231 rst_cnt <= '0'; -- reset clock still on this clock. this is used also when next pixel value
is (< mean - sigma) or (> mean + sigma)
232 count <= '1'; -- start the counter 1st word
233 sum <= x"000000000000" & unsigned(iv_data); -- start already with the 1st value it has
234 else
235 for i in 0 to REG_SIZE-2 loop -- this "for loop" is used to shift register the position of
the input data position to the position + 1 place...
236 reg_array(i+1) <= reg_array(i);
237 end loop;
238 reg_array(0) <= x"000000000000" & unsigned(iv_data); -- ...for then receive the next word
data in
239
240 if (unsigned(iv_data) < mean - sigma or
241 unsigned(iv_data) > mean + sigma) then -- if this word is outside the moving average
mean-sigma or mean+sigma (when a sharp image coutour is detected) ...
242 rst_cnt <= '1'; -- rise the flag
243 rst_cnt <= '0' after 3 ns; -- drop the flag (this rising and dropping cannot exceed a
whole clock period
244 start_from <= (others => '0'); -- put back reset_from = 0, that was forced to stay in the
value regN - 1, in order to maintain the average
245
246 for i in 1 to REG_SIZE-1 loop -- sets all memory ...
247 reg_array(i) <= (others => '0'); -- ...to zero due to a sharp detection
248 end loop;
249 sum <= x"000000000000" & unsigned(iv_data) after 1 ns; -- restart only with this new value
250 else
251 sum <= sum + (x"000000000000" & unsigned(iv_data)) - reg_array(conv_integer(o_regN)-1)
after 1 ns; -- makes moving average until the size of of the filter included
252 if (regN - 1) = idx_reg_addr then -- if the counter reaches teh filter size, N (actualy
counter must be equal to N-1 because it starts on 0)
253 start_from <= reg_addr; -- if reached then keep it there (dont let go higher than N
254 end if;
255 end if;
256
257 end if;
258 else
259 sum <= (others => '0');
260 end if;
261 end if;
262
263 end process;
264
265
266 end architecture;
```

A.2 Simulation file of moving average FPGA source code

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.ALL;
4 use ieee.numeric_std.ALL;
5
6 entity tb_mov_avg_thesis is
7 generic (
8 G_DATA_W :integer := 16;
9 COUNTER_DEPTH :integer := 7 --counter will need to count until 100 for N maximum number for
averaging, hence 2^7=128 which is enough
10 );
11 end entity;
12
13 architecture tb_mov_avg_thesis_rtl of tb_mov_avg_thesis is
14 signal clk :std_logic := '0';
15 signal rst :std_logic := '0';
16 signal en :std_logic := '0';
17 signal DataEn :std_logic := '0';
18 signal DataEn_reg1 :std_logic := '0';
19 signal DataEn_reg2 :std_logic := '0';
20 signal DataEn_reg3 :std_logic := '0';
21 signal DataEn_reg4 :std_logic := '0';
22 signal DataEn_reg5 :std_logic := '0';
23 signal o_count :std_logic;
24 signal DataIn :std_logic_vector(G_DATA_W-1 downto 0);
25 signal o_reg_addr :std_logic_vector(COUNTER_DEPTH-1 downto 0);
26 signal o_rst_cnt :std_logic;
27 --signal o_can_reset :std_logic;
28 --signal o_frst_rst_made :std_logic;
29 signal o_regN :std_logic_vector(G_DATA_W-1 downto 0);
30 signal o_regSigmaCoeff :std_logic_vector(G_DATA_W-1 downto 0);
31 signal o_regFWC :std_logic_vector(G_DATA_W-1 downto 0);
32 signal o_regColourRange :std_logic_vector(G_DATA_W-1 downto 0);
33 signal o_regnumPasses :std_logic_vector(G_DATA_W-1 downto 0);
34 signal o_reg0 :std_logic_vector(G_DATA_W-1 downto 0);
35 signal o_reg1 :std_logic_vector(G_DATA_W-1 downto 0);
36 signal o_reg2 :std_logic_vector(G_DATA_W-1 downto 0);
37 signal o_reg3 :std_logic_vector(G_DATA_W-1 downto 0);
38 signal o_reg4 :std_logic_vector(G_DATA_W-1 downto 0);
39 signal o_sum :std_logic_vector(G_DATA_W*4-1 downto 0);
40 signal o_mean :std_logic_vector(G_DATA_W*4-1 downto 0);
41 signal o_sqrtMean_x_sigCoeff:std_logic_vector(G_DATA_W*4-1 downto 0);
42 signal o_sqrtN :std_logic_vector(G_DATA_W*4-1 downto 0);
43 signal o_num_sigma1 :std_logic_vector(G_DATA_W*4-1 downto 0);
44 signal o_den_sigma1 :std_logic_vector(G_DATA_W*4-1 downto 0);
45 signal o_sigma1 :std_logic_vector(G_DATA_W*4-1 downto 0);
46 signal o_numTMPsigma2 :std_logic_vector(G_DATA_W*4-1 downto 0);
47 signal o_num_sigma2 :std_logic_vector(G_DATA_W*4-1 downto 0);
48 signal o_den_sigma2 :std_logic_vector(G_DATA_W*4-1 downto 0);
49 signal o_sigma2 :std_logic_vector(G_DATA_W*4-1 downto 0);
50 signal o_meanPow2 :std_logic_vector(G_DATA_W*4-1 downto 0);
51 signal o_num_sigma3 :std_logic_vector(G_DATA_W*4-1 downto 0);
52 signal o_den_sigma3 :std_logic_vector(G_DATA_W*4-1 downto 0);
53 signal o_sigma3 :std_logic_vector(G_DATA_W*4-1 downto 0);
54 signal o_sigma :std_logic_vector(G_DATA_W*4-1 downto 0);
55 signal DataOut :std_logic_vector(G_DATA_W-1 downto 0);
56 -- for ReadDataFromFile and WriteDataToFile
57 signal eof :std_logic := '0';
58 signal eof_reg1 :std_logic := '0';
59 signal eof_reg2 :std_logic := '0';
60
61 signal o_fifo_DataOut :std_logic_vector(G_DATA_W*2-1 downto 0); --user_r_read_32_data,
62 signal o_full :std_logic := '0'; --user_w_write_32_full,
63 signal o_empty :std_logic := '1'; --user_r_read_32_empty
64
65 constant ClkGenConst :time := 10 ns;
66
67 begin
68 ------------------------------------------------------------------------------------------------------------------------
69 read_data_inst: entity work.read_data_from_file
70 generic map (
71 G_FILE_NAME =>
"C:\Users\rj\Xilinx\RJa\Projects_HDL\project_2\project_2.srcs\sources_1\bd\design_1\hdl\simulation\mov_a
vg_thesis/xn3508_in.dat",
72
--"D:\RJa\Projects_HDL\project_2\project_2.srcs\sources_1\bd\design_1\hdl\simulation\mov_
avg_thesis/xn3508_in.dat", -- :string := "DataIn.dat";
73 G_DATA_W => G_DATA_W -- :integer := 16
74 )
75 port map (
76 clk => clk, -- :in std_logic;
77 en => en, -- :in std_logic;
78 o_data_en => DataEn, -- :out std_logic;
79 ov_data => DataIn, -- :out std_logic_vector(G_DATA_W-1 downto 0);
80 o_eof => eof -- :out std_logic
81 );
```

```
82
-------------------------------------------------------------------------------------------------------------
83 mov_avg_inst: entity work.mov_avg_thesis
84 generic map (
85 G_DATA_W => G_DATA_W
86 )
87 port map (
88 clk => clk, -- :in std_logic;
89 rst => rst, -- :in std_logic;
90 en => en, --DataEn, -- :in std_logic;
91 o_count => o_count, -- :buffer std_logic;
92 iv_data => DataIn, -- :in std_logic_vector(G_DATA_W-1 downto 0);
93 o_reg_addr => o_reg_addr, -- :buffer std_logic_vector(COUNTER_DEPTH-1 downto 0);
94 o_rst_cnt => o_rst_cnt, -- :buffer std_logic;
95 o_regN => o_regN, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
96 o_regSigmaCoeff => o_regSigmaCoeff, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
97 o_regFWC => o_regFWC, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
98 o_regColourRange => o_regColourRange,-- :buffer std_logic_vector(G_DATA_W-1 downto 0);
99 o_regNumPasses => o_regNumPasses, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
100 o_reg0 => o_reg0, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
101 o_reg1 => o_reg1, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
102 o_reg2 => o_reg2, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
103 o_reg3 => o_reg3, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
104 o_reg4 => o_reg4, -- :buffer std_logic_vector(G_DATA_W-1 downto 0);
105 o_sum => o_sum, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0); -- NOTE: for
safe we reserve inner math operations to
106 o_mean => o_mean, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0); --
4*data_in size, although 2*datata_in would be
107 o_sqrtMean_x_sigCoeff => o_sqrtMean_x_sigCoeff,--:buffer std_logic_vector(G_DATA_W*4-1 downto 0); --
suffice
108 o_sqrtN => o_sqrtN, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
109 o_num_sigma1 => o_num_sigma1, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
110 o_den_sigma1 => o_den_sigma1, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
111 o_sigma1 => o_sigma1, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
112 o_numTMPsigma2 => o_numTMPsigma2, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
113 o_num_sigma2 => o_num_sigma2, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
114 o_den_sigma2 => o_den_sigma2, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
115 o_sigma2 => o_sigma2, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
116 o_meanPow2 => o_meanPow2, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
117 o_num_sigma3 => o_num_sigma3, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
118 o_den_sigma3 => o_den_sigma3, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
119 o_sigma3 => o_sigma3, -- :buffer std_logic_vector(G_DATA_W*4-1 downto 0);
120 o_sigma => o_sigma, -- :buffer std_logic _vector(G_DATA_W*4-1 downto 0);
121 ov_data => DataOut -- :out std_logic_vector(G_DATA_W-1 downto 0)
122 );
123
124 -- 32-bit loopback
125 fifo_32_inst : entity work.fifo_32x512
126 port map(
127 clk => clk, --bus_clk, -- :in std_logic;
128 srst => rst, --reset_32, -- :in std_logic;
129 din => x"0000" & DataOut, --user_w_write_32_data, -- :in std_logic_vector(31 downto 0);
130 wr_en => DataEn_reg4, --user_w_write_32_wren, -- :in std_logic;
131 rd_en => o_count, --o_count, --user_r_read_32_rden, -- :in std_logic;
132 dout => o_fifo_DataOut, --user_r_read_32_data, -- :out std_logic_vector(31 downto 0);
133 full => o_full, --user_w_write_32_full, -- :out std_logic;
134 empty => o_empty --user_r_read_32_empty -- :out std_logic;
135 );
136 --reset_32 <= not (user_w_write_32_open or user_r_read_32_open);
137 --user_r_read_32_eof <= '0';
138
139 -------------------------------------------------------------------------------------------------------------
140 write_data_inst: entity work.write_data_to_file
141 generic map (
142 G_FILE_NAME =>
"C:\Users\rj\Xilinx\RJa\Projects_HDL\project_2\project_2.srcs\sources_1\bd\design_1\hdl\simulation\mov_a
vg_thesis/xn3508_out.dat",
143
--"D:\RJa\Projects_HDL\project_2\project_2.srcs\sources_1\bd\design_1\hdl\simulation\mov_
avg_thesis/xn3508_out.dat", -- :string := "DataOut.dat";
144 G_DATA_W => G_DATA_W -- :integer := 16
145 )
146 port map (
147 clk => clk, -- :in std_logic;
148 en => o_count, --DataEn_reg2, -- :in std_logic;
149 iv_data => DataOut, -- :out std_logic;
150 i_eod => eof_reg1, -- :out std_logic_vector(G_DATA_W-1 downto 0);
151 o_eof => open -- :out std_logic
152 );
153
-------------------------------------------------------------------------------------------------------------
154 clock_generator: process
155 begin
156 clk <= '0' after ClkGenConst, '1' after 2*ClkGenConst;
157 wait for 2*ClkGenConst;
158 end process;
159 -------------------------------------------------------------------------------------------------------------
160 rst <= '1', '0' after 100 ns;
161
```

```
-------------------------------------------------------------------------------------------------------------
en_proc: process
begin
en <= '0';
wait until rst = '0';
wait until rising_edge(clk);
wait for 35 ns;
en <= '1';
wait until eof = '1';
en <= '0';
--wait until o_empty = '1';
wait;
end process;

-------------------------------------------------------------------------------------------------------------
delay_en_for_write: process(clk)
begin
if rising_edge(clk) then
DataEn_reg1 <= DataEn;
DataEn_reg2 <= DataEn_reg1;
DataEn_reg3 <= DataEn_reg2;
DataEn_reg4 <= DataEn_reg3;
DataEn_reg5 <= DataEn_reg4;
eof_reg1 <= eof;
eof_reg2 <= eof_reg1;
end if;
end process;
```

A.3 CPU side C source code

A.3.3 Streamread.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* streamread.c -- Demonstrate read from a Xillybus FIFO.

This simple command-line application is given one argument: The device
file to read from. The read data is sent to standard output.

This program has no advantage over the classic UNIX 'cat' command. It was
written merely to demonstrate the coding technique.

We don't use allread() here (see memread.c), because if less than the
desired number of bytes arrives, they should be handled immediately.

See http://www.xillybus.com/doc/ for usage examples an information.

*/

void allwrite(int fd, unsigned char *buf, int len);

int main(int argc, char *argv[]) {

int fd, rc;
unsigned char buf[128];


if (argc!=2) {
fprintf(stderr, "Usage: %s devfile\n", argv[0]);
exit(1);
}

fd = open(argv[1], O_RDONLY);

if (fd < 0) {
if (errno == ENODEV)
fprintf(stderr, "(Maybe %s a write-only file?)\n", argv[1]);

perror("Failed to open devfile");
exit(1);
}

while (1) {
rc = read(fd, buf, sizeof(buf));

if ((rc < 0) && (errno == EINTR))
continue;

if (rc < 0) {
```

```
perror("allread() failed to read");
exit(1);
}

if (rc == 0) {
fprintf(stderr, "Reached read EOF.\n");
exit(0);
}

// Write all data to standard output = file descriptor 1
// rc contains the number of bytes that were read.

allwrite(1, buf, rc);
}
}

/*
Plain write() may not write all bytes requested in the buffer, so
allwrite() loops until all data was indeed written, or exits in
case of failure, except for EINTR. The way the EINTR condition is
handled is the standard way of making sure the process can be suspended
with CTRL-Z and then continue running properly.

The function has no return value, because it always succeeds (or exits
instead of returning).

The function doesn't expect to reach EOF either.
*/

void allwrite(int fd, unsigned char *buf, int len) {
int sent = 0;
int rc;

while (sent < len) {
rc = write(fd, buf + sent, len - sent);

if ((rc < 0) && (errno == EINTR))
continue;

if (rc < 0) {
perror("allwrite() failed to write");
exit(1);
}

if (rc == 0) {
fprintf(stderr, "Reached write EOF (?!)\n");
exit(1);
}

sent += rc;
}
}
```

A.3.4 Streamwrite.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termio.h>
#include <signal.h>

/* streamwrite.c -- Demonstrate write to a Xillybus FIFO

This simple command-line application is given one argument: The device
file to write to. The data is read from standard input.

This program can't be substituted by UNIX' 'cat', because the latter works
at line-by-line basis.

See http://www.xillybus.com/doc/ for usage examples an information.

*/

void allwrite(int fd, unsigned char *buf, int len);
void config_console();

int main(int argc, char *argv[]) {

int fd, rc;
unsigned char buf[128];

if (argc!=2) {
fprintf(stderr, "Usage: %s devfile\n", argv[0]);
```

```
exit(1);
}

fd = open(argv[1], O_WRONLY);

if (fd < 0) {
if (errno == ENODEV)
fprintf(stderr, "(Maybe %s a read-only file?)\n", argv[1]);

perror("Failed to open devfile");
exit(1);
}

config_console(); // Configure standard input not to wait for CR

while (1) { // this loop only ends if Ctrl+C pressed
rc = read(0, buf, sizeof(buf)); // Read from standard input = file descriptor 0

if ((rc < 0) && (errno == EINTR))
continue;

if (rc < 0) {
perror("allread() failed to read");
exit(1);
}

if (rc == 0) {
fprintf(stderr, "Reached read EOF.\n");
exit(0);
}

allwrite(fd, buf, rc);
}
}

/*
Plain write() may not write all bytes requested in the buffer, so
allwrite() loops until all data was indeed written, or exits in
case of failure, except for EINTR. The way the EINTR condition is
handled is the standard way of making sure the process can be suspended
with CTRL-Z and then continue running properly.

The function has no return value, because it always succeeds (or exits
instead of returning).

The function doesn't expect to reach EOF either.
*/

void allwrite(int fd, unsigned char *buf, int len) {
int sent = 0;
int rc;

while (sent < len) {
rc = write(fd, buf + sent, len - sent);

if ((rc < 0) && (errno == EINTR))
continue;

if (rc < 0) {
perror("allwrite() failed to write");
exit(1);
}

if (rc == 0) {
fprintf(stderr, "Reached write EOF (?!)\n");
exit(1);
}

sent += rc;
}
}

/* config_console() does some good-old-UNIX vodoo standard input, so that
read() won't wait for a carriage-return to get data. It also catches
CTRL-C and other nasty stuff so it can return the terminal interface to
what is was before. In short, a lot of mumbo-jumbo, with nothing relevant
to Xillybus.
*/

void config_console() {
struct termio console_attributes;

if (ioctl(0, TCGETA, &console_attributes) != -1) {
// If we got here, we're reading from console

console_attributes.c_lflag &= ~ICANON; // Turn off canonical mode
console_attributes.c_cc[VMIN] = 1; // One character at least
console_attributes.c_cc[VTIME] = 0; // No timeouts

```

```
122 if (ioctl(0, TCSETAF, &console_attributes) == -1)
123 fprintf(stderr, "Warning: Failed to set console to char-by-char\n");
124 }
125 }
126
```

A.3.5 MATLAB scrips and source C code

A.3.5.1 NoiseAndHistograms.m

```
1   delete(findall(0, 'Type', 'figure'));

2   FWC = 10e3; % Coefficient (or Factor) of the Wavelength Conversion in electrons (accepted value is 10000 e-)

3   img_original = imread('img_MartinPattern.png');
4   img_original = double(img_original(:, :, 1));
5   img_original = img_original/max(max(img_original)); % multiply by the maximum value of electrons

6   var = 0.001;

7   img_noisy=imnoise(img_original, 'gaussian', 0, var);

8   f1=figure('Name', 'img original', 'NumberTitle', 'off'); imshow(img_original)
9   f1.OuterPosition = [1 1805 1375 573];
10  f2=figure('Name', sprintf('noisy variance=%g%%', var*100), 'NumberTitle', 'off'); imshow(img_noisy)
11  f2.OuterPosition = [265 1805 1375 573];

12  col=1900;%col=4014;

13  f=figure('Name', 'histogram original img', 'NumberTitle', 'off');
14  f.OuterPosition = [1922 1735 341 493];
15  h_img_original=histogram(img_original(:,col) * FWC, 'Normalization', 'count');
16  f=figure('Name', sprintf('histogram noisy variance=%g%%, col=%d', var*100, col), 'NumberTitle', 'off');
17  f.OuterPosition = [1922 1201 341 493];
18  h_img_noisy=histogram(img_noisy(:,col) * FWC, 'Normalization', 'count');

19  figure('Name', sprintf('img 0.1%% noisy: col=%d', col), 'NumberTitle', 'off');
20  img_noisy=histogram(img_noisy(:,col), 'Normalization', 'count')
21  SNR_MovingAverage(img_noisy, img_original,  [1,2,3,4,5,6,7,8,9].^2, [50, 100], FWC, [0 FWC], 1, col)

22  return
```

A.3.5.2 SNR_MovingAverage.m

```
1   %=========================================================================================================================
2   % USAGE: [out, var] = SNR_Martin(noisy_image; original_image (clean image for comparison); sigmaCoeff=1,2,3;
3   % FWD=10000e-; colourRange=[0 2^16]; numPasses=1,2,3,4,...);
4   %=========================================================================================================================
5   % read first the noisy and the original images to the Workspace:
6   % im = imread('img.pgm'); the clean image for comparison
7   % im_noisy001 = imread('img_noisy001.pgm'); then noisy image for processing
8   % the rest of parameters are:
9   % filterLength = [1,2,3,4,5,6,7,8,9].^2; array of moving average filters of several lengths. the more lengthy,
    the more effective but also more blurry the picture
10  % sigmaCoeff = [1, 3]; (standard deviation, sigma, that restricts the noise variance for about 68% if 1xsigma or
    is more noise permissive about 99% if 3xsigma
11  % FWC = Coefficient (or Factor) of the Wavelength Conversion in electrons (accepted value is 10000 e-)
12  % colourRange = [0 1]; pictures range of colors
13  % numPasses = 1; number of passes that filter executes over image
14  % col = number of the column to plot a profile
15  % EXAMPLE: SNR_Martin_MovingAverage(im_noisy001, im,  [1,2,3,4,5,6,7,8,9].^2, [1, 3], [0 1], 10000, 1, col)
16  %        SNR_Martin_MovingAverage(img0001_noisy, img_original,  [1,2,3,4,5,6,7,8,9].^2, [50, 100],  FWC, [0 FWC],
    1,         num columns to plot for profile)
17  function SNR_MovingAverage (imNoisy,      imOriginal,    filterLength,           sigmaCoeff, FWC, colourRange,
    numPasses, col)
18
19  %     delete(findall(0, 'Type', 'figure')); % delete all figure. we want to start clean
20      mp = get(0,'monitorpositions'); %get the starting positions and the length of all monitors attached (this is
    for a multimonitor scenario)
21
22      attr = whos ('imNoisy'); % used to get image size (attr.size(1) is the height and attr.size(2) is the width)
23      imgsz = attr.size(1) * attr.size(2); %get the size of the image to be processed
24
25  % =========================================================================================================================
26  % this switch both images arguments to normalize double precision
27  % =========================================================================================================================
28      if attr.class ~= 'double'
```

```
29          imOriginal = double(imOriginal);% swithch to double...
30          imOriginal = imOriginal / max(max(imOriginal)); % ...and normalize it to to increase the precision
31          imNoisy = double(imNoisy);% swithch to double...
32          imNoisy = imNoisy / max( max(imNoisy));     % switch to double precision and normalize it to get accurate
    results
33      end
34
35      noise = sqrt(  abs( mean(mean(imNoisy.^2)) - mean(mean(imNoisy))^2 )  * imgsz/(imgsz-1)); %supplied by
    Ricardo Sousa, but Martin noticed the mean doesnt make sense, due to the various colours of the image
36
37
38  %=============================================================================================================
39  % this first nested for, for..end, end is for the variance with respect to the ROWS (comparison between ROWS in
    the same COLUMN)
40  %=============================================================================================================
41  %     lnMean  = imNoisy(1,:);                   %  line of means
42  %     imVar  = variance_factor * sqrt( imNoisy(:, :)); % matrix of variances.
43  %     imN = ones(1,attr.size(2));   % vector of number of pixels processed for this average. will reset (N = 1,
    due to MATLAB indexing) if a new average counting
44
45      initialmagnification = 250; %zoom the picture to show
46
47  %=============================================================================================================
48  % this creates a noisy piecewise function composed by 2 heavisides and a sin
49  %=============================================================================================================
50      t = -5:0.01:10; % independent variable
51      rng default %initiate random generator
52      a1 = find(t>=-5 & t<-2); % independent var for one Heaviside
53      a2 = find(t>=-2 & t<4); % independent var for another Heaviside
54      a3 = find(t>=4); % independent var for sin function
55      x(a1) = (heaviside(t(a1))-heaviside(t(a1)-3)); %it is composed by 2 step heavisides (kind of stairs) and a
    sin wave
56      x(a2) = heaviside(t(a2))-heaviside(t(a2)-3) + 1; % another heaviside
57      x(a3) = sin(t(a3)) + 1.2; % raise sin() on YY' axis to not get negative numbers (we want sin above 0
58      xn = x + 0.25*rand(size(t)); %create a squared step with noise
59
60  % =============================================================================================================
61  % this is a figure to compare lines with filters of several lengths. it takes a column from both images and
    compares the efficiency of the moving average algorithm using 3 methods:
62  % y1 = 'Simple moving average without regarding to sharp edge detection';
63  % y2 = 'Enterprises''s given algorithm: moving average with sharp edge detection';
64  % =============================================================================================================
65      % use a image's column of data to apply the filters and watch how they behave
66      t = 200 : 750; %size(imNoisy); % size() returns a brackets array of Lines By Columns, but if the brakets
    aren't used then the parameter Lines is choosed
67      x = imOriginal(t, 1137)';%2713)'; %3508)'; % we select any column of the original image
68      xn = imNoisy(t, 1137)';%2713)'; %3508)'; % and the same column and the noisy image column ot compre wih
69
70      y1 = 'Simple moving average'; % we will use the simple moving average wihtout reagard to the sharp colour
    detection
71      y2 = 'Moving average with sharp edge floating-point'; % and for comparison with the sharp egde detection
72      y3 = 'Moving average with sharp edge fixed-point';
73
74      szFilterLength  = length(filterLength); % take the length of the array of the filter's number of points
    (might be 10 or more filters of increasing number of point to choos ethe best efficiency)
75      szSigmaCoeff = length(sigmaCoeff); % take the length of the standard deviation, 1xsigma (~68%), or 3xsigma
    (~99%)
76      for i = 1 : szFilterLength % run the array of the several length filters for all the column
77          filter = (1/filterLength(i))*ones(1,filterLength(i)); % press F1 on help for 'filter' for the meaning of
    B...
78  %         A = 1;% ...and A
79          for j = 1 : szSigmaCoeff % does this for all sigmas (for ex. if sigmas are [1 3], does for 1xsigma and
    for 3xsigma)
80              eval(['[' sprintf('y1_%gx%g', filterLength(i), sigmaCoeff(j)) ', ' ...
81                        sprintf('y2_%gx%g', filterLength(i), sigmaCoeff(j)) ', ' ...
82                        sprintf('y3_%gx%g', filterLength(i), sigmaCoeff(j)) ']' '= myfilter_matlab(filter, xn,
    sigmaCoeff(j), FWC, numPasses);']);
83
84              % uncomment below, if ever needed to scale up from [0.0..1.0] to [0..FWC]
85  %             eval([ sprintf('y1_%gx%g', filterLength(i), sigmaCoeff(j)) '= FWC *' sprintf('y1_%gx%g',
    filterLength(i), sigmaCoeff(j)) ';']);
86  %             eval([ sprintf('y2_%gx%g', filterLength(i), sigmaCoeff(j)) '= FWC *' sprintf('y2_%gx%g',
    filterLength(i), sigmaCoeff(j)) ';']);
87          end
88      end
89
90
91      cc=hsv(9); %get the colours to plot the curves. the  Hue-saturation-value color map to plot each curve
92
93  % =============================================================================================================
94  % this is a figure to compare lines with filters of several lengths for y1 = 'Simple moving average without
    regarding to sharp edge detection';
95  % we will use the simple moving average wihtout reagard to the sharp colour detection
```

```
% ================================================================================================================
    figure('Name', y1,  'NumberTitle', 'off'); % create a figure frame to plot all curves for y2 (w/o sharp detection)
    set(gcf,'position',[mp(1,1) mp(1,2)+mp(1,4)*0.6 mp(1,3) mp(1,4)*0.45]); % set the position of the frame
    set(gca,'Position',[.03 .06 .95 .93], 'box','on'); % set the position of the axes
    hold on
    ax = gca;
    plot(t, x* FWC, 'Color', ax.ColorOrder(1,:), 'LineStyle', '-'); % plots 'original signal w/o noise', '-' means normal solid line
    plot(t, xn* FWC, 'Color', ax.ColorOrder(1,:), 'LineStyle', ':'); % plots 'signal w/ random noise', ':' means dotted line
    for i = 2 : szFilterLength %for each moving average window size
        for j = 1 : szSigmaCoeff %for each sigma
            if j==1,  plotTrace = '-';    else,    plotTrace =':';   end % sets the line type to be ploted, if solid or if dotted
            yy = eval(sprintf('y1_%dx%d', filterLength(i), sigmaCoeff(j))), 'LineStyle', plotTrace, 'Color', cc(i-1,:);
            plot(t, yy * FWC);
        end
    end
    hold off
    legend( 'original signal w/o noise', 'signal w/ random noise', ...
            sprintf('MA filter sz=%d, sigma=%d', filterLength(2), sigmaCoeff(1)), sprintf('MA filter sz=%d, sigma=%d', filterLength(2), sigmaCoeff(2)), ...
            sprintf('MA filter sz=%d, sigma=%d', filterLength(3), sigmaCoeff(1)), sprintf('MA filter sz=%d, sigma=%d', filterLength(3), sigmaCoeff(2)), ...
            sprintf('MA filter sz=%d, sigma=%d', filterLength(4), sigmaCoeff(1)), sprintf('MA filter sz=%d, sigma=%d', filterLength(4), sigmaCoeff(2)), ...
            sprintf('MA filter sz=%d, sigma=%d', filterLength(5), sigmaCoeff(1)), sprintf('MA filter sz=%d, sigma=%d', filterLength(5), sigmaCoeff(2)), ...
            sprintf('MA filter sz=%d, sigma=%d', filterLength(6), sigmaCoeff(1)), sprintf('MA filter sz=%d, sigma=%d', filterLength(6), sigmaCoeff(2)), ...
            sprintf('MA filter sz=%d, sigma=%d', filterLength(7), sigmaCoeff(1)), sprintf('MA filter sz=%d, sigma=%d', filterLength(7), sigmaCoeff(2)), ...
            sprintf('MA filter sz=%d, sigma=%d', filterLength(8), sigmaCoeff(1)), sprintf('MA filter sz=%d, sigma=%d', filterLength(8), sigmaCoeff(2)), ...
            sprintf('MA filter sz=%d, sigma=%d', filterLength(9), sigmaCoeff(1)), sprintf('MA filter sz=%d, sigma=%d', filterLength(9), sigmaCoeff(2)));
    xlabel('length (pixels)');
    ylabel('colour depth  (0 is the darkest,  10e3 is the lighter )');

% ================================================================================================================
% this is a figure to compare lines with filters of several lengths for y2 = 'Enterprises's given algorithm: moving average with sharp edge detection'; % and for comparison with the sharp egde detection
% ================================================================================================================
    figure('Name', sprintf('%s, sigma factor: [%d, %d] FWC: %g, num passes: %d', y2, sigmaCoeff, FWC, numPasses), 'NumberTitle', 'off'); % creates a frame window to plot all curves for y3 (with sharp detection)
    set(gcf,'position',[mp(1,1) mp(1,2) mp(1,3) mp(1,4)*0.4]); % set the position of the frame
    set(gca,'Position',[.03 .07 .95 .92], 'box','on'); % set the position of the window with all curves
    hold on
    ax = gca;
    plot(t, x* FWC, 'Color', ax.ColorOrder(1,:), 'LineStyle', '-'); % plots 'original signal w/o noise', '-' means normal solid line (TAKE OUT "* FWC" of "x * FWC" 2nd parameter, THE MULTIPLICATIVE SCALE PLOTED LINE IF LINE BETWEEN [0..1])
    plot(t, x* FWC, 'Color', ax.ColorOrder(1,:), 'LineStyle', ':'); % plots 'signal w/ random noise', ':' means dotted line (TAKE OUT "* FWC" of "xn * FWC" 2nd parameter, THE MULTIPLICATIVE SCALE PLOTED LINE IF LINE BETWEEN [0..1])
    for i = 2 : szFilterLength % for each moving average window size
        for j = 1 : szSigmaCoeff % for each sigma
            if j==1,  plotTrace = '-'; else,  plotTrace =':'; end % chooses between solid (for sigma=1) or dotted line (for sigma=3)
            yy = eval(sprintf('y3_%dx%d', filterLength(i), sigmaCoeff(j))), 'LineStyle', plotTrace, 'Color', cc(i-1,:);
             plot(t, yy * FWC);
        end
    end
    hold off
    legend( 'original signal w/o noise', 'signal w/ random noise', ...
            sprintf('S-G filter sz=%d, order=2', filterLength(2)), sprintf('S-G filter sz=%d, order=4', filterLength(2)), ...
            sprintf('S-G filter sz=%d, order=2', filterLength(3)), sprintf('S-G filter sz=%d, order=4', filterLength(3)), ...
            sprintf('S-G filter sz=%d, order=2', filterLength(4)), sprintf('S-G filter sz=%d, order=4', filterLength(4)), ...
            sprintf('S-G filter sz=%d, order=2', filterLength(5)), sprintf('S-G filter sz=%d, order=4', filterLength(5)), ...
            sprintf('S-G filter sz=%d, order=2', filterLength(6)), sprintf('S-G filter sz=%d, order=4', filterLength(6)), ...
            sprintf('S-G filter sz=%d, order=2', filterLength(7)), sprintf('S-G filter sz=%d, order=4', filterLength(7)), ...
            sprintf('S-G filter sz=%d, order=2', filterLength(8)), sprintf('S-G filter sz=%d, order=4', filterLength(8)), ...
            sprintf('S-G filter sz=%d, order=2', filterLength(9)), sprintf('S-G filter sz=%d, order=4', filterLength(9)));
    xlabel('length (pixels)');
    ylabel('colour depth  (0 is the darkest,  10e3 is the lighter )');
return;
%     figure('Name', 'Original Image',  'NumberTitle', 'off');  % figure frame for the original image
```

```
155 %     imshow(imOriginal,  'Border', 'tight', 'DisplayRange', colourRange, 'InitialMagnification',
    initialmagnification); % show the original image
156 %     figure('Name', 'Noisy Image',  'NumberTitle', 'off');  % figure frame for the noisy image
157 %     imshow(imNoisy,  'Border', 'tight', 'DisplayRange', colourRange, 'InitialMagnification',
    initialmagnification);% show the noisy image
158
159
160     filter = (1/filterLength(2))*ones(1,filterLength(2));
161     [imDenoised_04x1_SMA, noise_04x1_SMA, mse_04x1_SMA, peaksnr_04x1_SMA, snr_04x1_SMA, global_sim_04x1_SMA, ~,
    ...
162      imDenoised_04x1_FlP, noise_04x1_FlP, mse_04x1_FlP, peaksnr_04x1_FlP, snr_04x1_FlP, global_sim_04x1_FlP, ~,
    ...
163      imDenoised_04x1_FxP, noise_04x1_FxP, mse_04x1_FxP, peaksnr_04x1_FxP, snr_04x1_FxP, global_sim_04x1_FxP, ~
    ...
164     ] = calculateNoiseAndShowImage(filter, attr, sigmaCoeff(1), FWC, imNoisy, noise, imOriginal, colourRange,
    initialmagnification, numPasses, [1 1805-151 1375 573]);
165 %    f=figure('Name', sprintf('imDenoised: filter=4pts sigma=%d col=%d', sigmaCoeff(1), col), 'NumberTitle',
    'off');
166     f.OuterPosition = [2264 1735 341 493];
167 %    h_imDenoised_04x1=histogram(imDenoised_04x1_FxP(:,col) * FWC, 'Normalization', 'count');
168
169     filter = (1/filterLength(2))*ones(1,filterLength(2));
170     [imDenoised_04x3_SMA, noise_04x3_SMA, mse_04x3_SMA, peaksnr_04x3_SMA, snr_04x3_SMA, global_sim_04x3_SMA, ~,
    ...
171      imDenoised_04x3_FlP, noise_04x3_FlP, mse_04x3_FlP, peaksnr_04x3_FlP, snr_04x3_FlP, global_sim_04x3_FlP, ~,
    ...
172      imDenoised_04x3_FxP, noise_04x3_FxP, mse_04x3_FxP, peaksnr_04x3_FxP, snr_04x3_FxP, global_sim_04x3_FxP, ~
    ...
173     ] = calculateNoiseAndShowImage(filter, attr, sigmaCoeff(2), FWC, imNoisy, noise, imOriginal, colourRange,
    initialmagnification, numPasses, [546 1805-151 1375 573]);
174 %    f=figure('Name', sprintf('imDenoised: filter=4pts sigma=%d, col=%d', sigmaCoeff(2), col), 'NumberTitle',
    'off');
175     f.OuterPosition = [2264 1201 341 493];
176 %    h_imDenoised_04x3=histogram(imDenoised_04x3_FxP(:,col) * FWC, 'Normalization', 'count');
177
178     filter = (1/filterLength(3))*ones(1,filterLength(3));
179     [imDenoised_09x1_SMA, noise_09x1_SMA, mse_09x1_SMA, peaksnr_09x1_SMA, snr_09x1_SMA, global_sim_09x1_SMA, ~,
    ...
180      imDenoised_09x1_FlP, noise_09x1_FlP, mse_09x1_FlP, peaksnr_09x1_FlP, snr_09x1_FlP, global_sim_09x1_FlP, ~,
    ...
181      imDenoised_09x1_FxP, noise_09x1_FxP, mse_09x1_FxP, peaksnr_09x1_FxP, snr_09x1_FxP, global_sim_09x1_FxP, ~
    ...
182     ] = calculateNoiseAndShowImage(filter, attr, sigmaCoeff(1), FWC, imNoisy, noise, imOriginal, colourRange,
    initialmagnification, numPasses, [1 1805-302 1375 573]);
183 %    f=figure('Name', sprintf('imDenoised: filter=9pts, sigma=%d, col=%d', sigmaCoeff(1), col), 'NumberTitle',
    'off');
184     f.OuterPosition = [2607 1735 341 493];
185  %   h_imDenoised_09x1=histogram(imDenoised_09x1_FxP(:,col) * FWC, 'Normalization', 'count');
186
187     filter = (1/filterLength(3))*ones(1,filterLength(3));
188     [imDenoised_09x3_SMA, noise_09x3_SMA, mse_09x3_SMA, peaksnr_09x3_SMA, snr_09x3_SMA, global_sim_09x3_SMA, ~,
    ...
189      imDenoised_09x3_FlP, noise_09x3_FlP, mse_09x3_FlP, peaksnr_09x3_FlP, snr_09x3_FlP, global_sim_09x3_FlP, ~,
    ...
190      imDenoised_09x3_FxP, noise_09x3_FxP, mse_09x3_FxP, peaksnr_09x3_FxP, snr_09x3_FxP, global_sim_09x3_FxP, ~
    ...
191     ] = calculateNoiseAndShowImage(filter, attr, sigmaCoeff(2), FWC, imNoisy, noise, imOriginal, colourRange,
    initialmagnification, numPasses, [546 1805-302 1375 573]);
192 %    f=figure('Name', sprintf('imDenoised: filter=9pts, sigma=%d, col=%d', sigmaCoeff(2), col), 'NumberTitle',
    'off');
193     f.OuterPosition = [2607 1201 341 493];
194 %    h_imDenoised_09x3=histogram(imDenoised_09x3_FxP(:,col) * FWC, 'Normalization', 'count');
195
196     filter = (1/filterLength(4))*ones(1,filterLength(4));
197     [imDenoised_16x1_SMA, noise_16x1_SMA, mse_16x1_SMA, peaksnr_16x1_SMA, snr_16x1_SMA, global_sim_16x1_SMA, ~,
    ...
198      imDenoised_16x1_FlP, noise_16x1_FlP, mse_16x1_FlP, peaksnr_16x1_FlP, snr_16x1_FlP, global_sim_16x1_FlP, ~,
    ...
199      imDenoised_16x1_FxP, noise_16x1_FxP, mse_16x1_FxP, peaksnr_16x1_FxP, snr_16x1_FxP, global_sim_16x1_FxP, ~
    ...
200     ] = calculateNoiseAndShowImage(filter, attr, sigmaCoeff(1), FWC, imNoisy, noise, imOriginal, colourRange,
    initialmagnification, numPasses, [1 1805-453 1375 573]);
201 %    f=figure('Name', sprintf('imDenoised: filter=16pts, sigma=%d, col=%d', sigmaCoeff(1), col), 'NumberTitle',
    'off');
202     f.OuterPosition = [2949 1735 341 493];
203 %    h_imDenoised_16x1=histogram(imDenoised_16x1_FxP(:,col) * FWC, 'Normalization', 'count');
204
205     filter = (1/filterLength(4))*ones(1,filterLength(4));
206     [imDenoised_16x3_SMA, noise_16x3_SMA, mse_16x3_SMA, peaksnr_16x3_SMA, snr_16x3_SMA, global_sim_16x3_SMA, ~,
    ...
207      imDenoised_16x3_FlP, noise_16x3_FlP, mse_16x3_FlP, peaksnr_16x3_FlP, snr_16x3_FlP, global_sim_16x3_FlP, ~,
    ...
208      imDenoised_16x3_FxP, noise_16x3_FxP, mse_16x3_FxP, peaksnr_16x3_FxP, snr_16x3_FxP, global_sim_16x3_FxP, ~
    ...
209     ] = calculateNoiseAndShowImage(filter, attr, sigmaCoeff(2), FWC, imNoisy, noise, imOriginal, colourRange,
    initialmagnification, numPasses, [546 1805-453 1375 573]);
```

```
%    f=figure('Name', sprintf('imDenoised: filter=16pts, sigma=%d, col=%d', sigmaCoeff(2), col), 'NumberTitle', 'off');
    f.OuterPosition = [2949 1201 341 493];
%    h_imDenoised_16x3=histogram(imDenoised_16x3_FxP(:,col) * FWC, 'Normalization', 'count');

    filter = (1/filterLength(5))*ones(1,filterLength(5));
    [imDenoised_25x1_SMA, noise_25x1_SMA, mse_25x1_SMA, peaksnr_25x1_SMA, snr_25x1_SMA, global_sim_25x1_SMA, ~, ...
     imDenoised_25x1_FlP, noise_25x1_FlP, mse_25x1_FlP, peaksnr_25x1_FlP, snr_25x1_FlP, global_sim_25x1_FlP, ~, ...
     imDenoised_25x1_FxP, noise_25x1_FxP, mse_25x1_FxP, peaksnr_25x1_FxP, snr_25x1_FxP, global_sim_25x1_FxP, ~ ...
    ] = calculateNoiseAndShowImage(filter, attr, sigmaCoeff(1), FWC, imNoisy, noise, imOriginal, colourRange, initialmagnification, numPasses, [1 1805-604 1375 573]);
%    f=figure('Name', sprintf('imDenoised: filter=25pts, sigma=%d, col=%d', sigmaCoeff(1), col), 'NumberTitle', 'off');
    f.OuterPosition = [3260 1735 341 493];
%    h_imDenoised_25x1=histogram(imDenoised_25x1_FxP(:,col) * FWC, 'Normalization', 'count');

    filter = (1/filterLength(5))*ones(1,filterLength(5));
    [imDenoised_25x3_SMA, noise_25x3_SMA, mse_25x3_SMA, peaksnr_25x3_SMA, snr_25x3_SMA, global_sim_25x3_SMA, ~, ...
     imDenoised_25x3_FlP, noise_25x3_FlP, mse_25x3_FlP, peaksnr_25x3_FlP, snr_25x3_FlP, global_sim_25x3_FlP, ~, ...
     imDenoised_25x3_FxP, noise_25x3_FxP, mse_25x3_FxP, peaksnr_25x3_FxP, snr_25x3_FxP, global_sim_25x3_FxP, ~ ...
    ] = calculateNoiseAndShowImage(filter, attr, sigmaCoeff(2), FWC, imNoisy, noise, imOriginal, colourRange, initialmagnification, numPasses, [546 1805-604 1375 573]);
%    f=figure('Name', sprintf('imDenoised: filter=25pts, sigma=%d, col=%d', sigmaCoeff(2), col), 'NumberTitle', 'off');
    f.OuterPosition = [3260 1201 341 493];
%    h_imDenoised_25x3=histogram(imDenoised_25x3_FxP(:,col) * FWC, 'Normalization', 'count');


% ======================  COMPARISON BETWEEN NOISY AND ORIGINAL ============================================


%     mse = immse(imNoisy, imOriginalforCompar);                                    % immse(A,ref) calculates the mean-squared error (MSE = 1/N * Sum(Yhat-Y)^2)
%     [peaksnr, snr] = psnr(imNoisy, imOriginalforCompar);                          % SNR = 10 * log10(Y^2 / MSE);    PSNR = 10 * log10( MaxPixelRes^2 / MSE)
%      str = sprintf('noisy image, noise=%g', noise);     % puts these measurements in the a string ...
%      subplot('Position', [0, 0.5, 0.5, 0.5]);
%     figure('Name', str,  'NumberTitle', 'off', 'Position', [1 ssz(4)*2/3 ssz(4)/2 ssz(4)/3]);
%  .. to put as title in the denoised image to show
%      imshow(imNoisy,  'Border', 'tight', 'DisplayRange', range);                    % show the denoise image
%      title(str);
%
%     [global_sim, local_sim] = ssim(imNoisy, imOriginalforCompar);                 % ssim(A,ref) computes the Structural Similarity Index (SSIM). The returning values,
%     str = sprintf('noisy image, Structural similarity, global=%g', global_sim);   %   global_sim, a unique value, is the average and the local_sim, an array,
%     subplot('Position', [0.5, 1/3, 0.5, 0.5]);
%     figure('Name', str,  'NumberTitle', 'off', 'Position', [ssz(4)/2 ssz(4)*2/3 ssz(4)/2 ssz(4)/3]);
%   is the individual values. the more white (near to 1), the more similar
%     imshow(local_sim,  'Border', 'tight', 'DisplayRange', range);                 % show the image
%     title('Original');


%      S = mean(mean(double(imOriginalforCompar) .^ 2));                            % signal power
%      MSE = mean(mean( (double(imOriginalforCompar) - double(imNoisy)) .^ 2 ));
%      SNR = 10 * log10(  mean(mean(double(imNoisy) .^ 2))  / MSE  );
%      PeakSNR = 10 * log10( range(2)^2 / MSE );

    return
end

%%
function [imDenoised_SMA, noise_SMA, mse_SMA, peaksnr_SMA, snr_SMA, global_sim_SMA,  local_sim_SMA, ...  % SMA = Simple Moving Average
          imDenoised_FlP, noise_FlP, mse_FlP, peaksnr_FlP, snr_FlP, global_sim_FlP,  local_sim_FlP, ...  % FlP = Floating-Point
          imDenoised_FxP, noise_FxP, mse_FxP, peaksnr_FxP, snr_FxP, global_sim_FxP,  local_sim_FxP] = ...% FxP = Fixed-Point
          calculateNoiseAndShowImage(filter, attr, sigmaCoeff, FWC, imNoisy, noiseOriginal,  imOriginal, range, initialmagnification, numPasses, figPos)
%FWC=100;
    imgsz = attr.size(1) * attr.size(2); %get the image size
    [~, filterSize] = size(filter);%take size of the filter window to convolute with the image pixels
    imDenoised_SMA = zeros(attr.size(1), attr.size(2));
    imDenoised_FlP = zeros(attr.size(1), attr.size(2));
    imDenoised_FxP = zeros(attr.size(1), attr.size(2));

    parfor i=1 : attr.size(2)  % loop for each column. on each loop, a whole column is being treated
```

```
273         imNoisyTemp = imNoisy(:, i)'; % we need the column in the form of a line, hence the transpose
274         [Temp1, Temp2, Temp3] = myfilter(filter, imNoisyTemp, sigmaCoeff,    FWC, numPasses); % this is where the moving average is working
275         imDenoised_SMA(:, i) = Temp1'; % get the line back to a column and insert it on the image
276         imDenoised_FlP(:, i) = Temp2'; % same as above but for floating-point
277         imDenoised_FxP(:, i) = Temp3'; % same as above but for fixed-point
278     end
279
280     noise_SMA = sqrt(  abs(mean(mean(imDenoised_SMA.^2)) - mean(mean(imDenoised_SMA))^2 ) * imgsz/(imgsz-1));
281     noise_FlP = sqrt(  abs(mean(mean(imDenoised_FlP.^2)) - mean(mean(imDenoised_FlP))^2 ) * imgsz/(imgsz-1));
282     noise_FxP = sqrt(  abs(mean(mean(imDenoised_FxP.^2)) - mean(mean(imDenoised_FxP))^2 ) * imgsz/(imgsz-1));
283
284     mse_SMA = immse(imDenoised_SMA,         imOriginal); % immse(A,ref) calculates the mean-squared error (MSE)
285     mse_FlP = immse(imDenoised_FlP,         imOriginal);
286     mse_FxP = immse(imDenoised_FxP / FWC,   imOriginal); % Fixed point must be restore back (/ FWC) to the range [0..1] as the original image
287
288     [peaksnr_SMA, snr_SMA] = psnr(imDenoised_SMA,         imOriginal); % psnr(A,ref) calculates peak signal-to-noise ratio
289     [peaksnr_FlP, snr_FlP] = psnr(imDenoised_FlP,         imOriginal);
290     [peaksnr_FxP, snr_FxP] = psnr(imDenoised_FxP / FWC,   imOriginal); % Fixed point must be restore back (/ FWC) to the range [0..1] as the original image
291
292     [global_sim_SMA, local_sim_SMA] = ssim(imDenoised_SMA      ,   imOriginal); % ssim(A,ref) computes the Structural Similarity Index (SSIM). Global is a unique
293     [global_sim_FlP, local_sim_FlP] = ssim(imDenoised_FlP      ,   imOriginal); %  ... variable and Local, a pixel per pixel comparison (differential image).
294     [global_sim_FxP, local_sim_FxP] = ssim(imDenoised_FxP / FWC,   imOriginal); % Fixed point must be restore back (/ FWC) to the range [0..1] as the original image
295
296
297     str1 = sprintf('SimpMovAvg:
P=%d,W=%g,S=%g,mse=%.6g,SNR=%.5f,peakSNR=%.5f,globalSim=%.6g,noise=%.6g,originalNoise=%.5f', ...
298                    numPasses, filterSize, sigmaCoeff, mse_SMA, snr_SMA, peaksnr_SMA, global_sim_SMA*100, noise_SMA, noiseOriginal);  %   global_sim, a unique value, is the average and the local_sim, an array,
299     str2 = sprintf('Float Point:
P=%d,W=%g,S=%g,mse=%.6g,SNR=%.5f,peakSNR=%.5f,globalSim=%.6g,noise=%.6g,originalNoise=%.5f', ...
300                    numPasses, filterSize, sigmaCoeff, mse_FlP, snr_FlP, peaksnr_FlP, global_sim_FlP*100, noise_FlP, noiseOriginal)  %   global_sim, a unique value, is the average and the local_sim, an array,
301     str3 = sprintf('Fixed Point:
P=%d,W=%g,S=%g,mse=%.6g,SNR=%.5f,peakSNR=%.5f,globalSim=%.6g,noise=%.6g,originalNoise=%.5f', ...
302                    numPasses, filterSize, sigmaCoeff, mse_FxP, snr_FxP, peaksnr_FxP, global_sim_FxP*100, noise_FxP, noiseOriginal)  %   global_sim, a unique value, is the average and the local_sim, an array,
303     newline
304
305 %     f=figure('Name', str1,  'NumberTitle', 'off');  % show the denoise image
306 %     imshow(imDenoised_SMA,  'Border', 'tight', 'DisplayRange', [0 1], 'InitialMagnification', initialmagnification); % show the denoise image
307 %     f.OuterPosition = [figPos(1) figPos(2) figPos(3) figPos(4)];
308     f=figure('Name', str2,  'NumberTitle', 'off');  % show the denoise image
309     imshow(imDenoised_FlP,  'Border', 'tight', 'DisplayRange', [0 1], 'InitialMagnification', initialmagnification); % show the denoise image
310     f.OuterPosition = [figPos(1) figPos(2)-25 figPos(3) figPos(4)];
311     f=figure('Name', str3,  'NumberTitle', 'off');  % show the denoise image
312     imshow(imDenoised_FxP,  'Border', 'tight', 'DisplayRange', [0 FWC], 'InitialMagnification', initialmagnification); % show the denoise image
313     f.OuterPosition = [figPos(1) figPos(2)-50 figPos(3) figPos(4)];
314 end
315
316 % This function is implemented in a mex file "myfilter.mexmaci64" for spped up
317 function [vecY1, vecY2, vecY3] = myfilter_matlab (filter, vecX, sigma_coeff, FWC, numPasses)%, sharp_detect )
318     %, vecY4, vecY5, vecY6] = myfilter_matlab (filter, vecX, sigma_coeff, FWC, numPasses)%, sharp_detect )
319
320 %     mwSize i, j, n, M0;
321 %     double N, colorScale1, colorScale2, colorScale3;
322     sf2 = sigma_coeff^2;
323     f = 1.0; % this is a multiplicative scaling factor to stretch the maximum bit range of Digital Numbers used in integer calculations nature of FPGA's. Due to integer floor division and sqrt rounding results from math FPGA'a operations, this factor can be used as a "smoothing agent" since this can improve accuracy by a factor of 1/f^3 (see below the function used to calculate vecY3[])
324
325     if (numPasses >= 2)
326         backup = vecX;
327     end
328
329     %%
330     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
331     % simple moving average without regarding to sharp edge detection
332     % works like this: the pixel(i) gets the value of the Mean of all pixels under filter over the source vector
333     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
334     [~, sizeFilter] = size(filter); % get the filter width, this will be the size of the filter to convolute with the image X
335     [~, lenX] = size(vecX);
336     vecY1 = vecX; % initialize output
```

```
    vecY2 =  vecX; % initialize output
    vecY3 =  vecX; % initialize output
    for j=sizeFilter : lenX % loop for the whole vector
        vecY1(j) = mean(vecX(j-sizeFilter+1:j)); % simply make the average of the window filter size
    end
    if sizeFilter >2
        if (sizeFilter==4 || sizeFilter==16 || sizeFilter==36 || sizeFilter==64 || sizeFilter==100)
            vecY3 = sgolayfilt(vecX,6,2+sizeFilter+1);
        else
            vecY3 = sgolayfilt(vecX,6,2+sizeFilter);
        end
    end

    %%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Enterprise's given algorithm: moving average with sharp edge detection
    % works the same way as the above simple moving average, but if a sharp change in color is detected, outside [pixel(n-1)-sigma..[pixel(n-1)+sigma] restart pixel counting for a new growing filter until the specified size
    % Algorithm is: if [pixel(n-1)-sigma <= pixel(n) <= [pixel(n-1)+sigma] then make average, else start a new counting (reset average), being sigma = sqrt(pixel) / sqrt(N);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    [~, lenX] = size(vecX); % get the picture height
    [~, sizeFilter] = size(filter); % get the filter width, this will be the size of the filter to convolve with the image X

    for n = 1 : numPasses % number of passes means the number of filter auto-convolutions. so, if numPasses=1, filter is a unit step. if 2, filter is triangular, if 3, a Gaussian shape and so on
        vecY2 = zeros(1, lenX);% initialize output (the length of the image rows, temporal noise)
        M0 = 1;% M0 is the beginning index of the filter,  so start now by initializing it
        for i = 1 : lenX %do it for the whole number of the image's rows (temporal noise)

            if i == M0 % the beginning of the filter (M0) has the same index as the one being processed (i)
                vecY2(i) = vecX(i); % get the same value of the source vector
            else % or the filter is growing in size while maintaining the average, so, the beginning position maintains the same
                if i - M0 >= sizeFilter % dont let the filter size grow up more than size of it was requested
                    M0 = M0 + 1; % if it does, just move the initial index position forward, maintaining its size as moving occurs
                end

                % since dark colors are noise noticeable, we need more smoothing on these colors, so let's use functions for it
                %colourScale = exp(-3e-4 * vecY2[i-1]);  // concave smooth function - inverse exponential smooth function (smooths the noise on all range of colours inverse exponential, meaning lighter colours get less smoothing, and reverse otherwise) -> to make tests between 3e-4 (more decay) until 1e-4 (less decay)
                colourScale = 1.214e-8*FWC^2*vecY2(i-1)^2 - 0.0002290*FWC*vecY2(i-1) + 1.096; % polynomial 2nd degree
                %colourScale = 2/(exp(0.0004*vecY2[i-1]+0.05)+exp(-(0.0004*vecY2[i-1]+0.05))); // sec hyperbolic smooth function (factor multiplying vecY2[i-1] scales the function, the smaller the spreaded the function. the plus factor displaces the function)
                %colourScale = 2/(exp(0.0005*vecY2[i-1]+0.88)-exp(-(0.0005*vecY2[i-1]+0.88))); // cosec hyperbolic smooth function (factor multiplying vecY2[i-1] scales the function, the smaller the spreaded the function. the plus factor displaces the function)
                %colourScale = (-1e-4 * vecY2[i-1] + 1); // linear smooth function, negative inclination- (smooths the noise on all range of colours linearly negatively)
                %colourScale = (-1e-8 * pow(vecY2[i-1], 2) + 1); // convex smooth function, inverted - inverted parabola smooth
                %colourScale = 1-1/(exp((-vecY2[i-1]+FWC/2)/(0.1*FWC)) + 1); // logistic sigmoid smooth function, mirrored around 0.5*FWD- mirrored sigmoid smooth

                % now, calculate sigma, given that sigma = sqrt(mean) / sqrt(N), and check if [pixel(n-1)-sigma <= pixel(n) <= [pixel(n-1)+sigma]
                vecY2(i) = sqrt(vecY2(i-1) / (i - M0 + 1)) * (sigma_coeff/FWC*100) * colourScale; % this is sigma = sqrt(mean) / sqrt(N)

                if vecX(i) < vecY2(i-1) - vecY2(i) || ... % if this pixel is outside interval [mean-sigma..mean+sigma]
                   vecX(i) > vecY2(i-1) + vecY2(i)
                    vecY2(i) = vecX(i);%restart a new moving average (picture edge detection)
                    M0 = i;
                else % else, the pixel is inside the interval, include this pixel to contribute to the mean
                    vecY2(i) = mean(vecX(M0 : i)); % make average accounting with all pixels under the filter
%                     lf = length(vecX(M0 : i));
%                     if lf >= 4
%                         vecY5(i) = sgolayfilt(vecX,2,lf);
%                     else
%                         vecY5(i) = vecY2(i);
%                     end
                end


            end
        end

        if numPasses >= 2 % if more than 1 pass...
            vecX = vecY2; % ... then use this processed vector as a source for the next pass
```

```
        end
    end

return;

    if (numPasses >= 2)
        vecX = backup;
    end
    for i = 1 : lenX  % fixed-point numbers cannot be calculated within [0..1]. must be calculated in...
        vecX(i) = round(vecX(i) * FWC); %  ... a much greater range, say [0..FWC], being FWC=10000. round here is used as if Matlab has passed this vector
    end
    for n = 1 : numPasses % number of passes means the number of filter auto-convolutions. so, if numPasses=1, filter is a unit step. if 2, filter is triangular, if 3, a Gaussian shape and so on
        vecY3 = zeros(1, lenX);% initialize output (the length of the image rows, temporal noise)
        M0 = 1;% M0 is the beginning index of the filter,  so start now by initializing it
        for i = 1 : lenX %do it for the whole number of the image's rows (temporal noise)

            if i == M0 % the beginning of the filter (M0) has the same index as the one being processed (i)
                vecY3(i) = floor(vecX(i)); % get the same value of the source vector
            else % or the filter is growing in size while maintaining the average, so, the beginning position maintains the same
                if i - M0 >= sizeFilter % dont let the filter size grow up more than size of it was requested
                    M0 = M0 + 1; % if it does, just move the initial index position forward, maintaining its size as moving occurs
                end

                N = i - M0 + 1;

                % since dark colors are noise noticeable, we need more smoothing on these colors, so let's use functions for it
                %colourScale = exp(-3e-4 * vecY2[i-1]);  // concave smooth function - inverse exponential smooth function (smooths the noise on all range of colours inverse exponential, meaning lighter colours get less smoothing, and reverse otherwise) -> to make tests between 3e-4 (more decay) until 1e-4 (less decay)
%colourScale = 1.214e-8*vecY2(i-1)^2 - 0.0002257*vecY2(i-1) + 1.096; % polynomial 2nd degree
                %colourScale = 2/(exp(0.0004*vecY2[i-1]+0.05)+exp(-(0.0004*vecY2[i-1]+0.05))); // sec hyperbolic smooth function (factor multiplying vecY2[i-1] scales the function, the smaller the spreaded the function. the plus factor displaces the function)
                %colourScale = 2/(exp(0.0005*vecY2[i-1]+0.88)-exp(-(0.0005*vecY2[i-1]+0.88))); // cosec hyperbolic smooth function (factor multiplying vecY2[i-1] scales the function, the smaller the spreaded the function. the plus factor displaces the function)
                %colourScale = (-1e-4 * vecY2[i-1] + 1); // linear smooth function, negative inclination- (smooths the noise on all range of colours linearly negatively)
                %colourScale = (-1e-8 * pow(vecY2[i-1], 2) + 1); // convex smooth function, inverted - inverted parabola smooth
                %colourScale = 1-1/(exp((-vecY2[i-1]+FWC/2)/(0.1*FWC)) + 1); // logistic sigmoid smooth function, mirrored around 0.5*FWD- mirrored sigmoid smooth

                % now, calculate sigma, given that sigma = sqrt(mean) / sqrt(N), and check if [pixel(n-1)-sigma <= pixel(n) <= [pixel(n-1)+sigma]
%vecY2(i) = sqrt(vecY2(i-1) / (i - M0 + 1)) * sigma_coeff * colourScale; % this is sigma = sqrt(mean) / sqrt(N)
                 vecY3(i) = floor(sqrt( floor( ...
                                                sf2 * vecY3(i-1) * (828114186436020984673730560*(f^2)-17302750793234380800000.0*f*vecY3(i-1)+917272465632599875.0*(vecY3(i-1)^2))^2 ...
/ ...
(57089907708238395242331438777979805455309864960000000.0*(f^3)*N) ...
                                                ) ...
                                            ) ...
                                    );

                if vecX(i) < vecY3(i-1) - vecY3(i) || ... % if this pixel is outside interval [mean-sigma..mean+sigma]
                   vecX(i) > vecY3(i-1) + vecY3(i)
                    vecY3(i) = floor(vecX(i));%restart a new moving average (picture edge detection)
                    M0 = i;
                else % else, the pixel is inside the interval, include this pixel to contribute to the mean
                    vecY3(i) = mean(vecX(M0 : i)); % make average accounting with all pixels under the filter

                end

            end
        end

        if numPasses >= 2 % if more than 1 pass...
            vecX = vecY3; % ... then use this processed vector as a source for the next pass
        end
    end

end
%%
% ======================RESET THE IMAGE'S RESOLUTION BACK TO THE ORIGINAL=================================
function imOut = resetImageResolution(imIn, attr)
```

```
    imOut = imIn;
    if strcmp(attr.class,  'uint8')
        imOut    = uint8(round(imIn));
    elseif strcmp(attr.class,  'uint16')
        imOut    = uint16(round(imIn));
    elseif strcmp(attr.class,  'uint32')
        imOut    = uint32(round(imIn));
    elseif strcmp(attr.class,  'uint64')
        imOut    = uint64(round(imIn));
    end
End
```

A.3.5.3 myfilter.c (used to speed up MATLAB scripts)

```
/*=========================================================
 * myfiltert.c - a MAC executable that is to be called by SNR_Martin_MovingAverage.m
 *
 * Given an array of values as an arguments, it returns 2 arrays:
 * one, is a simple moving average filtering of the input argument
 * and the second is also a moving average filtering but sharp colour
 * transitions detection. It speeds up greatly the calculations in
 * relation to the same implementation done inside the script
 * SNR_Martin_MovingAverage.m
 *
 * The calling syntax is:
 *
 *      [array1, array2] = arrayProduct(filter, inputArray, sigmaFactor, FWC, numPasses)
 * array1 = simple moving average of 'inputArray' convoluted with the 'filter'
 * array2 = moving average with colour sharp detection of 'inputArray' convoluted with the 'filter'
 * filter = to convolute with 'inputArray'
 * sigmaCoeff = tolerance of the moving average operation
 * numPasses = number of convolution passes of 'inputArray' with 'filter'
 *
 * This is a MEX-file for MATLAB.
 * Copyright kykaku for Master Thesis.
 *
 *=========================================================*/
#include <string.h> // for memcpy()
// #include "engine.h"
#include <math.h>
#include "mex.h"
/*=========================================================*/
void mean(double *dest, double *ptrIniVec, double *ptrEndVec)
{
    double *ptr = ptrIniVec;
    *dest = 0;

    for (ptr = ptrIniVec; ptr <= ptrEndVec; ptr++) {
        *dest += *ptr;
    }
    *dest /= (ptrEndVec-ptrIniVec+1);
}
void mean_integer(double *dest, double *ptrIniVec, double *ptrEndVec) //same as above but results return number integer part only by floor(). this is to simulate FPGA's calculations
{
    double *ptr = ptrIniVec;
    *dest = 0;

    for (ptr = ptrIniVec; ptr <= ptrEndVec; ptr++) {
        *dest += *ptr;
    }
    *dest = floor(*dest / (ptrEndVec-ptrIniVec+1));
}

/*=========================================================================================================================================*/
// The computational routine
void myfilter(  double *filter, size_t lenfilter, // filter size and filter length
                double *vecX, size_t lenvecX, double sigmaFactor, double FWC, size_t numPasses, // src vector, length, and parameters: sigmaFactior
                double *vecY1, double *vecY2, double *vecY3)//, // destination vectors: vecY1 (simple mov average) vecY2 (mov avg w/ edge detection) vecY3 (sama as vecY2 but on fixed point to simulate FPGA calculations)
                //double *vecY4, double *vecY5, double *vecY6) // destination vectors: vecY4 (savitzky-golay filter) vecY5 (s-g w/ edge detection) vecY6 (sama as vecY5 but on fixed point to simulate FPGA calculations)
{
    mwSize i, j, n, M0;
    double N, colorScale;
    double sf2 = pow(sigmaFactor, 2);
    double f = 1.0; // this is a multiplicative scaling factor to stretch the maximum bit range of Digital Numbers used in integer calculations nature of FPGA's. Due to integer floor division and sqrt rounding results from math FPGA'a operations, this factor can be used as a "smoothing agent" since this can improve accuracy by a factor of 1/f^3 (see below the function used to calculate vecY3[])
    double* backup = (double*) malloc(sizeof(double)*lenvecX); // allocate space for original backup
    memcpy((void *)backup, (void *)vecX, sizeof(double)*lenvecX); // 1st save a copy of the original, because it might be overwriten if numpasses > 1

```

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// %     % simple moving average without regarding to sharp edge detection (this is calculated in the range [0..1]
// %     % works like this: the pixel(i) gets the value of the Mean of all pixels under filter over the source vector
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    memcpy((void *)vecY1, (void *)vecX, sizeof(double)*lenvecX); // initialize output
    for (j = lenfilter-1; j < lenvecX; j++) {// loop for the whole vector
        mean( &vecY1[j], vecX + (j - lenfilter + 1), vecX + j ); // simply make the average of the window filter size
    }

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// %     % Enterprise given algorithm: moving average with sharp edge detection (this is the floating-point version calculated within range [0..1])
// %     % works the same way as the above simple moving average, but if a sharp change in colour is detected, outside [pixel(n-1)-sigma..[pixel(n-1)+sigma] restart pixel counting for a new growing filter until the specified size
// %     % Algorithm is: if [pixel(n-1)-sigma <= pixel(n) <= [pixel(n-1)+sigma] then make average, else start a new counting (reset average), being sigma = sqrt(pixel) / sqrt(N);
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    for (n = 0; n < numPasses; n++) { // number of passes means the number of filter auto-convolutions. so, if numPasses=1, filter is a unit step. if 2, filter is triangular, if 3, a Gaussian shape and so on
        memset(vecY2, 0, sizeof(double)*lenvecX); // initialize output (the length of the image rows, temporal noise)
        M0 = 0; // M0 is the beginning index of the filter,  so start now by initializing it

        for (i = 0; i < lenvecX; i++) { // run for this whole row (temporal noise)
            if (i == M0) { // if the beginning (M0) and the end (i) of the filter match
                vecY2[i] = vecX[i]; // get the same value of the source vector
            } else { // else, if the filter is growing (maintaining the average), the beginning (M0) doesn't move while the end of the filtar (i) keeps moving
                if (i - M0 >= lenfilter) { // dont let the filter size grow up more than size of it was requested
                    M0 += 1; // if it does, just move the initial index position forward, maintaining its size as moving occurs
                }

                // since dark colors are noise noticeable, we need more smoothing on these colors, so let's use functions for it.
                // FWC must be joined to each members which figures vecY2[i-1] in the same degree (linear, squared, ...) to scale this function between [0..1]
                // polynomial 2nd degree is already scaled. check all the others
                //colorScale = exp(-3e-4*FWC*vecY2[i-1]);  // concave smooth function - inverse exponential smooth function (smooths the noise on all range of colours inverse exponential, meaning lighter colours get less smoothing, and reverse otherwise) -> to make tests between 3e-4 (more decay) until 1e-4 (less decay)
                colorScale = 1.214e-8*pow(FWC,2)*pow(vecY2[i-1],2) - 0.0002290*FWC*vecY2[i-1] + 1.096; // polynomial 2nd degree.
                //colorScale = 2/(exp(0.0004*FWC*vecY2[i-1]+0.05) + exp(-(0.0004*FWC*vecY2[i-1]+0.05))); // sec hyperbolic smooth function (factor multiplying vecY2[i-1] scales the function, the smaller the spreaded the function. the plus factor displaces the function)
                //colorScale = 2/(exp(0.0005*FWC*vecY2[i-1]+0.88) - exp(-(0.0005*FWC*vecY2[i-1]+0.88))); // cosec hyperbolic smooth function (factor multiplying vecY2[i-1] scales the function, the smaller the spreaded the function. the plus factor displaces the function)
                //colorScale = -1e-4*FWC*vecY2[i-1] + 1; // negative inclination linear smoothing function, - (smooths the noise out linearly negative)
                //colorScale = (-1e-8*pow(FWC,2)*pow(vecY2[i-1],2) + 1); // convex smooth function, inverted - inverted parabola smooth
                //colorScale = 1-1/(exp((-FWC*vecY2[i-1]+FWC/2)/(0.1*FWC)) + 1); // logistic sigmoid smooth function, mirrored around 0.5*FWD- mirrored sigmoid smooth

                // now, calculate sigma*sigmaCoeff, given that sigma = sqrt(pixel) / sqrt(N), and check if [pixel(n-1)-sigma <= pixel(n) <= [pixel(n-1)+sigma]
                vecY2[i] = sqrt(vecY2[i-1] / (i - M0 + 1)) * (sigmaFactor/FWC*100) * colorScale; // this is sigma = sqrt(pixel / N) times the desired sigma factor (if we want more or less smoothing) times the color scale function
                                                                                                   // (sigmaFactor/FWC*100) needs to be adjusted in percentage (divide by FWC, then multiplied for 100)
                if (vecX[i] < vecY2[i-1] - vecY2[i] ||  // if this pixel is outside interval [pixel-sigma..pixel+sigma]
                    vecX[i] > vecY2[i-1] + vecY2[i]) {
                    vecY2[i] = vecX[i]; // restart a new moving average (picture edge detection)
                    M0 = i;
                } else { // else, the pixel is inside the interval, include this pixel to contribute to the mean
                    mean( &vecY2[i], vecX + M0,  vecX + i ); // make average accounting with all pixels under the filter
                }

```

```
            }
        }

        if (numPasses > 1)
            memcpy(vecX,  vecY2, sizeof(double)*lenvecX); // use this processed vector as a source for the next pass
    }



//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// %     % Enterprise given algorithm: moving average with sharp edge detection (fixed point version of the above to simulate FPGA's calculations where it will be implemented)
// %     %  fixed-point cannot be calculated within [0..1]. must be calculated in a much bigger range, say [0..FWC], being FWC=10000 in this case
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    if (numPasses > 1)
        memcpy((void *)vecX, (void *)backup, sizeof(double)*lenvecX);

    for (i = 0; i < lenvecX; i++) { // fixed-point numbers cannot be calculated within [0..1]. must be calculated in...
        vecX[i] = round(vecX[i] * FWC); //  ... a much greater range, say [0..FWC], being FWC=10000. round here is used as if Matlab has passed this vector
    }
    for (n = 0; n < numPasses; n++) { // number of passes means the number of filter auto-convolutions. so, if numPasses=1, filter is a unit step. if 2, filter is triangular, if 3, a Gaussian shape and so on
        memset(vecY3, 0, sizeof(double)*lenvecX); // initialize output (the length of the image rows, temporal noise)
        M0 = 0; // M0 is the beginning index of the filter,  so start now by initializing it

        for (i = 0; i < lenvecX; i++) { // run for this whole row (temporal noise)
            if (i == M0) { // if the beginning (M0) and the end (i) of the filter match
                vecY3[i] = floor(vecX[i]); // get the same value of the source vector
            } else { // else, if the filter is growing (maintaining the average), the beginning (M0) doesn't move while the end of the filtar (i) keeps moving
                if (i - M0 >= lenfilter) { // dont let the filter size grow up more than size of it was requested
                    M0 += 1; // if it does, just move the initial index position forward, maintaining its size as moving occurs
                }

                N = i - (double)M0 + 1;

                // this is the same function as the above  vecY2[i] = sqrt(vecY2[i-1] / (i - M0 + 1)) * (sigmaFactor/FWC*100) * colorScale, but properly formated for fixed-point calculations
                //  generation of this function was made with algebric manipulation to reduce all to 1 sqrt and 1 division for better precision on fixed-point, and having the above function as starting point
                //  the coefficients were get with the help of MATLAB
                vecY3[i] = floor(sqrt( floor(
                                           sf2 * vecY3[i-1] * pow(82811418643602098467373056.0*pow(f,2)-17302750793234380800000.0*f*vecY3[i-1]+917272465632599875.0*pow(vecY3[i-1],2) , 2)
/
(570899077082383952423314387779798054553098649600000.0*pow(f,3)*N)
                                           )
                                      )
                                 );


                if (vecX[i] < vecY3[i-1] - vecY3[i] ||  // if this pixel is outside interval [pixel-sigma..pixel+sigma]
                    vecX[i] > vecY3[i-1] + vecY3[i]) {
                    vecY3[i] = floor(vecX[i]); // restart a new moving average (picture edge detection)
                    M0 = i;
                } else { // else, the pixel is inside the interval, include this pixel to contribute to the mean
                    mean_integer( &vecY3[i], vecX + M0,  vecX + i ); // make average accounting with all pixels under the filter
                }
            }
        }

        if (numPasses > 1)
            memcpy(vecX,  vecY3, sizeof(double)*lenvecX); // then use this processed vector as a source for the next pass
    }

//  if (numPasses > 1)
        memcpy((void *)vecX, (void *)backup, sizeof(double)*lenvecX); // restore back the original vector
}
```

```
176
177 // The gateway function
178 void mexFunction( int nlhs, mxArray *plhs[],
179                   int nrhs, const mxArray *prhs[])
180 {
181     double *filter;      // input: filter vector
182     size_t lenfilter;    // input: size of the filter
183     double *vecX;        // input: source vector
184     size_t lenvecX;      // input: size of source vector
185     double sigmaFactor;  // input: sigma smoother factor
186     double FWC;          // input: FWC (highest digital number of the image) or (Coefficient (or Factor) of the
    Wavelength Conversion in electrons, accepted value is 10000 e-)
187     size_t numPasses;    // input: number of passes to convolve filters
188     double *vecY1;       // output: vector (Moving Average filtering without sharp detection)
189     double *vecY2;       // output: vector (Moving Average filtering with sharp edge detection, floating-point)
190     double *vecY3;       // output: vector (Moving Average filtering with sharp edge detection, fixed-point to
    simulate FPGA)
191     double *vecY4;       // output: vector (Savitzky-Golay filtering)
192     double *vecY5;       // output: vector (Savitzky-Golay filtering with sharp edge detection)
193     double *vecY6;       // output: vector (Savitzky-Golay filtering with sharp edge detection, fixed-point to
    simulate FPGA)
194
195     if(nrhs!=5) { // check for proper number of input arguments
196         mexErrMsgIdAndTxt("MyToolbox:myfilter:nrhs","Msg from myfilter.c: 5 inputs required: *filter, *source,
    sigma_coeff, FWC and numPasses");
197     }
198     if(nlhs!=3) { // check for output parameters (3 Moving Average: simple, floating-point and fixed-point and 3
    Savitzky-Golay: simple, floating-point and fixed-point)
199         mexErrMsgIdAndTxt("MyToolbox:myfilter:nlhs","Msg from myfilter.c: 3 outputs required.");
200     }
201
202     if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0])) { // make sure the 1st input argument is type double
203         mexErrMsgIdAndTxt("MyToolbox:myfilter:notDouble","Msg from myfilter.c: Input filter vector must be type
    double.");
204     }
205     if(mxGetM(prhs[0])!=1) { // check that number of rows in 1st input argument is 1
206         mexErrMsgIdAndTxt("MyToolbox:myfilter:notRowVector","Msg from myfilter.c: 1st input argument, filter,
    must be a row vector.");
207     }
208
209     if( !mxIsDouble(prhs[1]) || mxIsComplex(prhs[1])) { // make sure the 2nd input argument is type double
210         mexErrMsgIdAndTxt("MyToolbox:myfilter:notDouble","Msg from myfilter.c: Input source vector must be type
    double.");
211     }
212     if(mxGetM(prhs[1])!=1) { // check that number of rows in 2nd input argument is 1
213         mexErrMsgIdAndTxt("MyToolbox:myfilter:notRowVector","Msg from myfilter.c: 2nd input argument, source
    vector, must be a row vector.");
214     }
215     if( !mxIsDouble(prhs[2]) || mxIsComplex(prhs[2]) ||  mxGetNumberOfElements(prhs[2])!=1 ) { // make sure the
    3rd input argument is scalar
216         mexErrMsgIdAndTxt("MyToolbox:myfilter:notScalar","Msg from myfilter.c: Input source length must be a
    scalar.");
217     }
218     if( !mxIsDouble(prhs[3]) || mxIsComplex(prhs[3]) ||  mxGetNumberOfElements(prhs[3])!=1 ) { // make sure the
    4th input argument is scalar
219         mexErrMsgIdAndTxt("MyToolbox:myfilter:notScalar","Msg from myfilter.c: Input sigmaFactor length must be a
    scalar.");
220     }
221     if( !mxIsDouble(prhs[4]) || mxIsComplex(prhs[4]) ||  mxGetNumberOfElements(prhs[4])!=1 ) { // make sure the
    5th input argument is scalar
222         mexErrMsgIdAndTxt("MyToolbox:myfilter:notScalar","Msg from myfilter.c: Input source length must be a
    scalar.");
223     }
224
225
226     filter = mxGetPr(prhs[0]);  // create a pointer to the real data in the input filter
227     lenfilter = mxGetN(prhs[0]);// get the dimensions of the input filter matrix filter
228     vecX = mxGetPr(prhs[1]);    // create a pointer to the real data in the input source
229     lenvecX = mxGetN(prhs[1]);  // get the value of the scalar nlenvecX input
230     sigmaFactor = mxGetScalar(prhs[2]); // get the value of the scalar sigma_coeff input
231     FWC = mxGetScalar(prhs[3]); // get the value of the scalar FWC input
232     numPasses = mxGetScalar(prhs[4]); // get the value of the scalar numPasses input
233
234     plhs[0] = mxCreateDoubleMatrix(1,(mwSize)lenvecX,mxREAL); // create the output vector for Simple moving
    average (numRows, numCols, realVals)
235     vecY1 = mxGetPr(plhs[0]);   // get a pointer to the real data in the output vecY1
236     plhs[1] = mxCreateDoubleMatrix(1,(mwSize)lenvecX,mxREAL); // create the output vector (Moving average
    floating-point with sharp edge detection)
237     vecY2 = mxGetPr(plhs[1]);   // get a pointer to the real data in the output vecY2
238     plhs[2] = mxCreateDoubleMatrix(1,(mwSize)lenvecX,mxREAL); // create the output vector (Moving average fixed-
    point with sharp edge detection)
239     vecY3 = mxGetPr(plhs[2]);     // get a pointer to the real data in the output vecY2
240 /*
241     plhs[3] = mxCreateDoubleMatrix(1,(mwSize)lenvecX,mxREAL); // create the output vector for Savitzky-Golay
    filter(numRows, numCols, realVals)
242     vecY4 = mxGetPr(plhs[3]);   // get a pointer to the real data in the output vecY1
```

```
243       plhs[4] = mxCreateDoubleMatrix(1,(mwSize)lenvecX,mxREAL); // create the output vector (Savitzky-Golay
    floating-point with sharp edge detection)
244       vecY5 = mxGetPr(plhs[4]);   // get a pointer to the real data in the output vecY2
245       plhs[5] = mxCreateDoubleMatrix(1,(mwSize)lenvecX,mxREAL); // create the output vector (Savitzky-Golay fixed-
    point with sharp edge detection)
246       vecY6 = mxGetPr(plhs[5]);     // get a pointer to the real data in the output vecY2
247    */
248       myfilter(filter, (mwSize) lenfilter, vecX, (mwSize) lenvecX, sigmaFactor, FWC, numPasses, vecY1, vecY2,
    vecY3);//, vecY4, vecY5, vecY6); // call the computational routine
249    }
250
251
```