NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

DIOGO FILIPE MIRANDA FERREIRA CASTRO
Master in Electrical and Computer Engineering

IMAGE PROCESSING TOOLBOX

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
⟨November⟩, ⟨2021⟩

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

IMAGE PROCESSING TOOLBOX

DIOGO FILIPE MIRANDA FERREIRA CASTRO

Master in Electrical and Computer Engineering

Adviser: José Manuel Matos Ribeiro da Fonseca
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: Paulo Miguel de Araújo Borges Montezuma de Carvalho
Associate Professor, NOVA University Lisbon

Rapporteur: André Teixeira Bento Damas Mora
Auxiliary Professor, NOVA University Lisbon

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
⟨November⟩, ⟨2021⟩

Image Processing Toolbox

Copyright © Diogo Filipe Miranda Ferreira Castro, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.6.7) [1].

Acknowledgements

First of all i want to thank my adviser Professor José Manuel Fonseca for all the guidance and patience throughout the whole process of developing this work.

Secondly, i also want to thank my family for supporting and giving me strength throughout the years and especially during this project. Without their presence this work would not be possible.

Last and not least, I would also like to thank all my friends and colleagues that shared my troubles and were present at all good and bad times, especially during the pandemic, with a special thanks to Afonso Carvalho, Catarina Pires, João Ramalho, Guilherme Lemos, Pedro Granja and Teresa Alves.

Abstract

Huge amounts of information regarding image processing and computer vision exist in the form of articles, books and courses. Most of them are too dense to understand or out of reach for most people. Nowadays most applications and scientific papers require certain advanced knowledge to understand its contents and make use of them. This leads for the need to recap the principles of image processing and computer vision, creating a path to the more complex and relevant concepts of today. In this thesis a structured document is developed explaining the principles and most common algorithms and methodologies, alongside with practical examples for each of them, attempting to solve this issue. In addition, this thesis' work includes the creation of a new open-source library dedicated to understand and learn how each algorithm and method is implemented and what happens to each image pixel during manipulation. The results present in this work relate to the comparison of the implemented methods through three metrics, execution time, image quality and features extracted. It is discussed the performance of these metrics for each method. The basics of image processing and computer vision can be learned from this work, as well as some algorithms and concepts from the image processing and computer vision communities.

Keywords: Image Processing, Computer Vision, Digital Image, Digital Filter, Image Processing Library, Python

Resumo

Grandes quantidades de informação relacionadas com processamento de imagem e visão computacional existem em formato de artigos, livros e cursos. Grande parte deles são demasiado densos para entender ou estão inacessíveis para a maior parte das pessoas. Hoje em dia a maioria das aplicações e artigos científicos é necessário conhecimento avançado para perceber os seus conteúdos e conseguir fazer uso deles. Isto leva à necessidade de recapitular os princípios de processamento de imagem e visão computacional, criando um caminho para conceitos mais complexos e relevantes atualmente. Neste trabalho é desenvolvido um documento a explicar os princípios, e algoritmos e metodologias mais comuns, juntamente com exemplos práticos, tentando resolver este problema. Adicionalmente também é desenvolvido uma biblioteca *open-source* dedicada ao entendimento e aprendizagem de cada algoritmo e método implementado, e o que ocorre a cada pixel durante a manipulação da imagem. Os resultados apresentados neste trabalho relatam as diferenças dos metodos implementados referentes a três metricas, tempo de execução, qualidade de imagem e extração de características. É discutido o desempenho de cada método para cada metrica. Os principios de processamento de imagem e visão computacional podem ser aprendidos através deste trabalho, bem como alguns algoritmos e conceitos provenientes das comunidades de processamento de imagem e visão computacional.

Palavras-chave: Processamento de Imagem, Visão Computacional, Imagem Digital, Filtro Digital, Biblioteca de Processamento de Imagem, Python

Contents

List of Figures

List of Tables

List of Listings

Glossary

Berkeley Software Distribution — The BSD license is a class of simple and very liberal licenses for computer software. The only restrictions placed on users, of software released under a typical BSD license, are that if they redistribute such software in any form, the users must include in the redistribution the original copyright notice, a list of two simple restrictions and a disclaimer of liability. These restrictions can be summarized as, one should not claim that they wrote the software if they did not write it and one should not sue the developer if the software does not function as expected or as desired.

Blob — Large object or anything bright in a dark background of an image. It can be generalized as a group of pixels that form a colony or a large object that is distinguishable from its background.

Compute Unified Device Architecture — CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by executing calculations using both the CPU and GPU.

Laser Imaging, Detection and Ranging LIDAR is a method to measure distances by illuminating the target with laser light and measure the time that the light takes to return back to the sensor. Differences in laser return times and wavelengths can then be used to make digital 3-D representations of the target.

Acronyms

2D	Two-Dimensional
3D	Three-Dimensional
ATAT	Algorithm Test and Analysis Tool
CDF	Cumulative Distribution Function
CPU	Central Processor Unit
FCT	NOVA School of Science and Technology
FEPC	Feature Extraction and Pattern Classification
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HSV	Hue Saturation Value
IDP	Image Processing Design
IPCV	Image Processing and Computer Vision
MLL	Machine Learning Library
PDF	Probability Density Function
RAM	Random Access Memory
RGB	Red Green Blue
SciCV	Computer Vision

1

Introduction

Chapter 1 presents the context of this thesis, main goals, contributions and document structure.

1.1 Context

Humans perceive the Three-Dimensional (3D) structure of the world with apparent ease, for example, when looking at the 3D shape of a vase of flowers sitting on a table, we can pinpoint all the meaningful characteristics, such as the color and flower species, which lead us to know the contents of the imagery in front of us. Looking at a framed group portrait, we can easily count and name all the people in the picture and even guess their emotions from their facial expressions [0]. By doing these simple actions, the human brain is performing pattern recognition and object segmentation without even being self-aware of that, and we do this since we are infants [0] [0].

As technology and artificial intelligence began to evolve, the next step to increase the accuracy and reliability of computer-based decisions was to implement the human brain acquisition and processing prowess into computers. Researchers have been developing mathematical techniques for recovering the 3D shape and appearance of objects in images [0]. The computer vision scientific field was created with the intent of understanding how computers can gain high-level perception from digital images or videos [0].

The process of capturing an image is susceptible to noise, which can arise from different sources. Some examples are: the way the image is taken, such as motion, brightness, blurriness, among other factors, image compression and transport processes, and camera components [0] [0]. Nowadays the importance of computer vision has been growing with the advancements of technology. That leads to the increasing need for faster and more efficient approaches for image capture and analysis [0] [0]. This problem founded the scientific field of image processing, which has the purpose of enhancing and manipulating images by applying filters and algorithms to increase their quality [0].

The importance of a detailed image is increasing in our society [0] [0]. This requirement is particularly evident in the medical field where images from scans or X-rays can

be polluted with noise, making it difficult to provide good analysis and diagnosis. Image processing has to be applied in order to enhance the quality of the images and help avoid life-critical mistakes [0] [0]. Is common in big industries to have robots helping human workers in routine tasks, such as transport of materials or examination of product quality. For such situations, a good image processing algorithm needs to be implemented in order to obtain specific features and deliver them to a supplemental decision algorithm [0] [0]. In civil construction, for example, sonar images of roads can be captured, and image processing algorithms can be applied to aid in the detection of fissures and cracks [0].

The worldwide knowledge obtained so far of computer vision and image processing is very diverse and scattered in many specific applications. A more centralized knowledge base would greatly benefit the image processing community, which could help the development of new ideas and approaches for our modern problems. By doing so, a step back has to be taken and this knowledge base has to be built from traditional methods up to the state-of-the-art algorithms.

1.2 Goals

The goal of this thesis is to start the foundation of a worldwide knowledge base document, where traditional and optimized methods are comprehended and explained. Also for beginners, it could be a reference point to help their learning experience in the computer vision and image processing fields.

A library containing the traditional and optimized algorithms was made so that the work done can be imported, analyzed and tested cross-platforms. The simple syntax and notes appended to the functions allow an easier reading and comprehension of the code, being within reach for everyone with basic programming knowledge to understand it.

Reference points were established between the studied algorithms by means of comparisons between new and current state-of-the-art methods, regarding time consumption, image quality and features extracted.

A toolbox will be made, which will have access to the featured library executing the algorithms implemented, and an intuitive Graphical User Interface (GUI) created to facilitate the presentation of results, being those the output image and the execution time of each algorithm, and their storage in a chart so that they can be reviewed and processed afterward.

Primarily this thesis aims to be an aiding tool for the courses of Sensory Systems and Advanced Topics in Digital Image Processing at NOVA School of Science and Technology (FCT). Students will have access to the proposed documentation, toolbox and library, where problems with each algorithm in specific situations will be addressed, such as handling the various types of noise and proper coding to save computation time. Additionally, cases where some algorithms are more advantageous to use will be reviewed.

1.3 Contributions

This work looks to contribute for the computer vision and image processing communities with the development of a solid knowledge base, that future beginners and researchers can consult or continue to increase and consolidate.

In addition, the Electrical and Computer Engineering Department in the Digital Area at FCT will greatly benefit from the study on traditional and current state-of-the-art methods for image processing, helping the lectures of Sensorial Systems and Advanced Topics in Digital Image Processing classes, allowing students a better understanding of what exists. The toolbox will be provided to the teaching staff, so a better notion of how algorithms are implemented and optimized can be taught to students.

1.4 Document Structure

This document is divided into six chapters. The first (the current chapter) introduces the context of this thesis and presents the main goals and contributions. The second chapter presents and discusses relevant the state-of-the-art toolboxes and libraries to this thesis. The third chapter presents the theoretical background necessary to understand this work. The fourth chapter presents practical examples for the addressed algorithms and methodologies in this thesis. The fifth chapter presents and discusses the results obtained from the algorithms and methodologies. Finally the sixth chapter presents the conclusion and future work regarding this thesis.

2

State-of-the-art

The areas of computer vision and image processing are composed of different methods for acquiring, processing, analyzing and interpreting digital images. The combination of these methods allows significant features of the 3D world to be extracted in order to produce some significant results, for example decisions [0].

To automate the processes of acquiring, processing and analyzing, various libraries and toolboxes were developed over the years, so that creating new applications for specific problems becomes easier.

In chapter 2 some of the most used toolboxes and libraries are briefly described, as well as some noteworthy mentions.

2.1 MathWorks Toolboxes

MATLAB is a programming language created by MathWorks in 1999. It focuses on numeric calculation, being capable of numeric analysis, matrix calculation, signal processing and plot creation in an easy-to-use environment. Since it is specialized in matrices calculation, it is of great use in the fields of control theory and signal processing [0].

With the development of MATLAB, more functionalities were implemented and several toolboxes were created in order to correspond to the increasing need of the users. Regarding the computer vision and image processing fields, various toolboxes were developed, tested and fully documented [0] [0]. Among them, the most relevant ones for this thesis are the Image Processing Toolbox, Computer Vision Toolbox and Image Acquisition Toolbox. These toolboxes are available as paid add-ons in MATLAB.

2.1.1 Computer Vision Toolbox

The Computer Vision Toolbox is a powerful add-on to MATLAB from MathWorks that provides pre-made algorithms, functions and applications for computer vision, 3D vision, and video and image processing systems. It offers the possibility of various common operations, such as feature and object detection, tracking, color space formatting and conversion, and pre-processing conditioning. For 3D vision, the toolbox supports single,

stereo, and fisheye camera calibration, stereo vision, 3D reconstruction, Laser Imaging, Detection and Ranging and 3D point cloud processing [0].

The toolbox supports C/C++ code implementations, allowing the integration of OpenCV C/C++ code into MATLAB. Deep learning and machine learning algorithms can be used to train custom models for certain tasks, for example, in object detection. Multi-core Central Processor Unit (CPU)s and Graphics Processing Unit (GPU)s can be used to improve system and algorithm performance [0].

2.1.2 Image Processing Toolbox

The Image Processing Toolbox, similarly to the Computer Vision Toolbox, provides a comprehensive set of reference-standard algorithms for image processing, analysis, visualization, as well as application development. With these algorithms, it is possible to perform image segmentation, enhancement, and registration. It can also perform noise reduction, geometric transformations, and 3D image processing [0].

The toolbox also supports C/C++ code implementations, which enables the integration of OpenCV C/C++ code into MATLAB. Multi-core CPUs and GPUs can be used to improve system and algorithm performance [0].

2.1.3 Image Acquisition Toolbox

The Image Acquisition Toolbox, similarly to the previously mentioned toolboxes, is an add-on in MATLAB that provides functions and blocks for connecting industrial or scientific devices and cameras to MATLAB and Simulink, for posterior video or image data analysis and processing. It includes an application that detects and enables the configuration of hardware properties. The toolbox enables acquisition modes such as processing in-the-loop, hardware triggering, background acquisition and synchronizing acquisition across multiple devices [0].

2.2 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source library, presented by Intel Corporation in 1999. Its purpose is to provide a common infrastructure for computer vision and machine learning applications on commercial products [0] [0].

This library contains more than 2500 optimized algorithms, including an extensive set of both classic and state-of-the-art computer vision and machine learning algorithms, therefore allowing facial detection and recognition, as well as identification of objects, classification of human actions in videos, tracking moving objects, extracting 3D models of objects, among other features. The library supports machine learning algorithm development, containing a general-purpose Machine Learning Library (MLL) that highly focuses on statistical pattern recognition and clustering [0] [0] [0].

Since OpenCV is an open-source library, it is continually being improved by the community. Additionally, being a ?? licensed product, it is extensively used in companies, research groups and governmental bodies, such as Google, Yahoo, Microsoft, Intel, Sony, IBM, Honda and Toyota [0].

The library has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android, and Mac OS. OpenCV leans mostly towards real-time vision applications. Being developed by Intel, it takes advantage of ?? libraries, which consist of optimized MMX and ?? instructions, when available, increasing the performance of the algorithms. A full-featured ?? and ?? interfaces are being actively developed. The library is written natively in C and C++ and has a template interface that works seamlessly with ?? containers [0] [0] [0].

2.3 Other toolboxes and libraries

There are many other computer vision and image processing related toolboxes, developed by various research groups and individuals. However, these toolboxes either have been developed for specific purposes, may not serve all the needs for image processing and pattern recognition tasks or do not have an active community. Nonetheless they are worth mentioning since they can provide supplementary support to well-established toolboxes or libraries commonly used.

2.3.1 Python Imaging Library

The Python Imaging Library, commonly know as PIL, adds image processing capabilities to the Python interpreter. This library provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities. It should provide a solid foundation for a general image processing tool [0].

This library is ideal for image archival and batch processing applications. The current version identifies and reads a large number of formats. It contains basic image processing functionality, including point operations, filtering with a set of built-in convolution kernels, and colour space conversions. The library also supports image resizing, rotation and arbitrary affine transforms, and possesses a histogram method allowing some statistics to be obtained from an image. This can be used for automatic contrast enhancement, and for global statistical analysis [0].

2.3.2 CVIPTools Toolbox

CVIPtools is a comprehensive GUI-based software for computer vision and image processing applications development made in MATLAB. It was originally developed at the Computer Vision and Image Processing Laboratory at Southern Illinois University at Edwardsville, for the master thesis of Deependra Mishra. Currently it is being updated under the direction of Dr. Scott E. Umbaugh [0] [0].

This toolbox presents an extensive list of computer vision and image processing techniques, for example image segmentation, edge detection, transform filters, spatial filters and image histogram operations. It contains two development tools for batch processing and algorithm analysis. The CVIP-Algorithm Test and Analysis Tool (ATAT), allows for implementation test and analysis, testing various combinations of values and parameters to speed frontend algorithm development. The CVIP-Feature Extraction and Pattern Classification (FEPC), allows for batch processing of images and experimenting multiple combinations of feature extraction and pattern classification parameters and techniques [0] [0] [0] [0].

The toolbox can be installed in the MATLAB environment and the functions can be used as built-in MATLAB functions. In addition to the MATLAB version, there is also a C/C++ skeleton with the same set of libraries written in C/C++ [0] [0] [0].

2.3.3 Peter Kovesi MATLAB/OCTAVE Library

Kovesi developed an extensive library of functions in MATLAB and Octave for computer vision and image processing that it is updated every month. Every function comes with a brief explanation and application example which can be found on the author's website. Those functions can be used and accessed as a built-in MATLAB function [0].

This library is capable of feature detection, segmentation, edge detection, image denoising, geometric transformations, grayscale transformation, among other functions [0].

2.3.4 Scilab

Scilab is an open-source program for numerical computation offering a user-friendly development environment for engineering and scientific applications, founded by the researchers of the National Institute for Research in Computer Science and Automation and the National School of Bridges and Roads of France in 1990. A large number of functionalities are available, for example, application development with a high-level performance programming language, signal processing, statistics and integration with third-party applications written in other programming languages [0] [0] [0] [0].

From this program there were developed three computer vision and image processing related toolboxes, Image Processing and Computer Vision (IPCV), Computer Vision (SciCV) and Image Processing Design (IDP). These toolboxes have similar purposes, although they have different methods of application development.

IPCV has many available functions, such as object detection and tracking, image linear filtering, analysis and transformation [0]. SciCVwas created to provide access to functions and objects of OpenCV in the Scilab environment [0]. IDP allows for a more detailed design and parameterization of the developed algorithms, having available the most common computer vision and image processing algorithms, for example thresholding, edge detection and segmentation [0].

2.3.5 Balu Toolbox

Balu toolbox is a MATLAB software for computer vision, pattern recognition and image processing developed in 2011 by Domingo Mery, professor in the Department of Computer Science at the University of Chile [0]. It has more than 200 functions capable of image processing, feature extraction, transformation, analysis and, selection, classification, among other functions. Also includes 2 GUIs for a more efficient application development [0].

This toolbox presents some limitations in the matter that, to be fully used some additional MATLAB toolboxes need to be installed, such as Image Processing Toolbox and Neural Network Toolbox, which are not freely available in the MATLAB base version [0].

2.3.6 Piotr´s Computer Vision Toolbox

Piotr´s Computer Vision Toolbox is a Berkeley Software Distribution licensed MATLAB software developed by Piotr Dollar in 2005. It is meant to facilitate the manipulation of images and video in MATLAB by complementing MATLAB's Image Processing Toolbox, with additional functions, being code reuse and code efficiency primary concerns. Because this toolbox is a complement of MATLAB's Image Processing Toolbox, MATLAB's toolbox needs to be installed to fully use Piotr's Computer Vision Toolbox [0].

2.4 Conclusion

Upon reviewing the state-of-the-art related to this thesis we can conclude that although no toolbox or library is complete and optimal, they complement each other. By combining the various toolboxes and libraries we can take advantage of their qualities and undermine their defects.

Although MATLAB toolboxes and OpenCV library are very commonly used, they have some limitations. MATLAB toolboxes add expenses to the user in order to fully utilize what it has to offer. OpenCV is complicated to learn and master for novice programmers. The other described toolboxes and libraries in this chapter might not address all the user's requirements, either being for lack of imaging functions or being dependent on other software's.

All these products have been optimized and require some former experience, either from knowing the methods to be used or as programmers. The existent documentation and open-source code can be very extensive and complex. The elaboration of simplified documentation can be of great educational value. For example, adding theoretical explanations and demonstrations of the algorithms, side-by-side with an easy to read code that shows how the featured algorithms were implemented.

3

Theoretical Background

Chapter 3 presents the theoretical background required to understand the methods explored in this thesis.

3.1 Digital Image

An image is a Two-Dimensional (2D) continuous spatial signal $f(x, y)$, where the value of the function f at the position (x, y) represents the intensity of the image at that point.

A digital image is a 2D array of numbers, with X rows and Y columns, that represents the real, continuous intensity distribution of an image spatial signal. This spatial signal is sampled at regular intervals and the intensity is quantized to a finite number of levels that better represent the signal. Each element of the array is referred to as a pixel. [0]

3.1.1 Sampling

Sampling is the process of taking a subset of data from a digital signal according to some specified rule, that subset can be referred to as samples. Inferences can be made about the spatial population of the digital signal based on the subsets [0].

Figure 3.1, represents in blue, the function $f = sin(2x)$. This signal can be sampled by gathering samples over the x-axis, for example, in periods of $\frac{3}{2}$ time units, represented as yellow, referred to as sample set 1, or $\frac{1}{6}$ time units, represented as green, referred to as sample set 2.

The number of samples will depend on the sampling frequency. According to the Nyquist sampling theorem, formulated as follows,

$$f_s > 2 \times f_m \tag{3.1}$$

the sampling frequency f_s has to be greater than twice the highest frequency of the image f_m, so that the sequence of samples captured can replicate the analog signal. If the chosen value is significantly higher than the Nyquist frequency then the image is called oversampled, while if the value is lower the image is called undersampled. An exactly sampled or oversampled image can be reconstructed from the samples obtained, while

an undersampled image will have spectral overlapping which results in an aliasing effect [0].

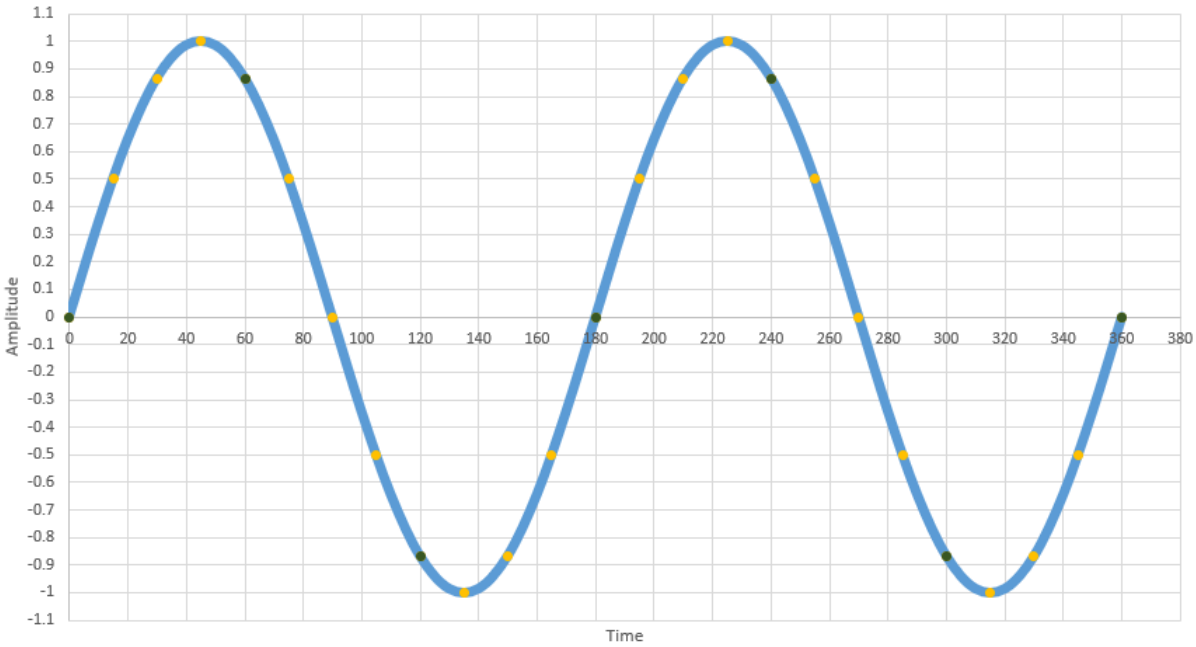


Figure 3.1: Representation of the sine function. In yellow is the sample set 1 (Oversampled subset). In green is the sample set 2 (Undersampled subset).

In signal processing, aliasing is related to distortions or artifacts in the reconstructed digital image, for example elements in the digital image that are not present in the original analog signal. From the points in sample set 2 it will be difficult to reconstruct the original analog signal, as seen in Figure 3.2, resulting in an aliased signal.

After sampling, we will have as many pixels as we have samples, so higher sampling frequencies correspond to a higher number of pixels which leads to a higher image resolution, that is, a more detailed image. Also, the number of samples determine the spatial resolution, that is, the size of the digital image.

3.1.2 Quantization

Quantization is the process of converting an analog sample value to a discrete pixel integer value. Each value is within a set interval and is assigned to a pixel in a way that the reconstructed image is of good quality and resembles as much as possible the original image [0].

Assuming the quantization range is $[-1, 1]$, with an 0.1 interval, Figures 3.3 and 3.4 illustrate the quantization of each sample set. From the sample set 1 there is more detailed information regarding the original signal, in comparison with sample set 2. Since sample set 1 has more samples then sample set 2, sample set 1 has more points to characterize the digital signal. So when reconstructing the original signal the aliasing effect will be lower in sample set 1, as seen in Figure 3.2.

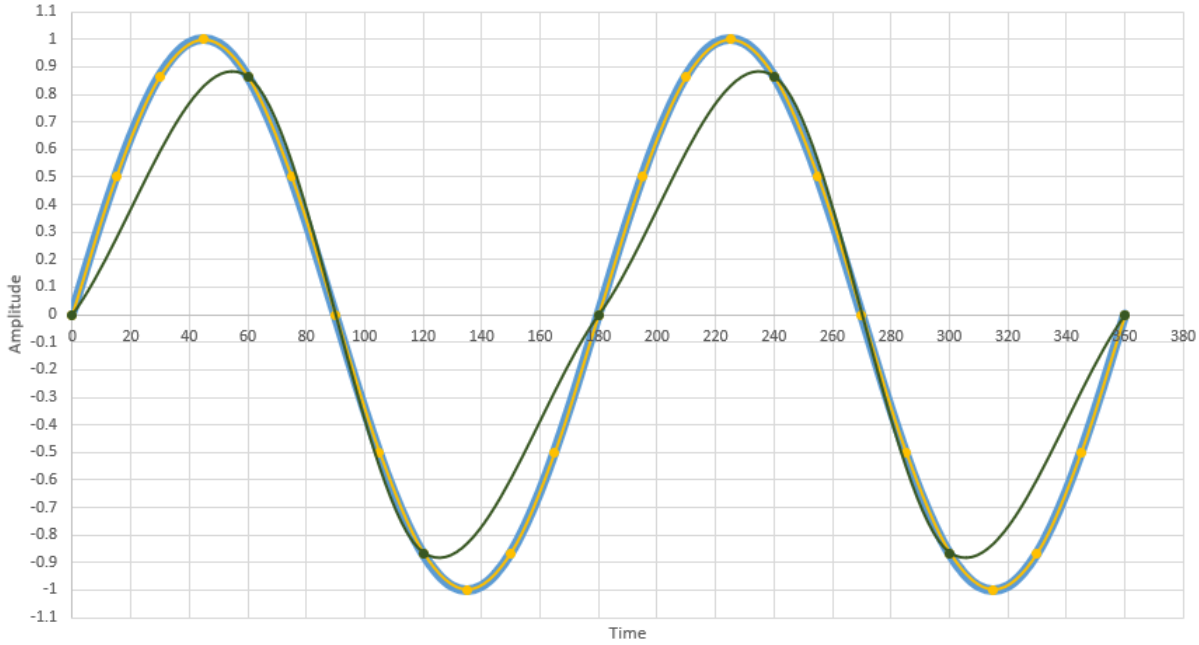


Figure 3.2: Result of reconstructing the original digital signal from sample set 1 and sample set 2.

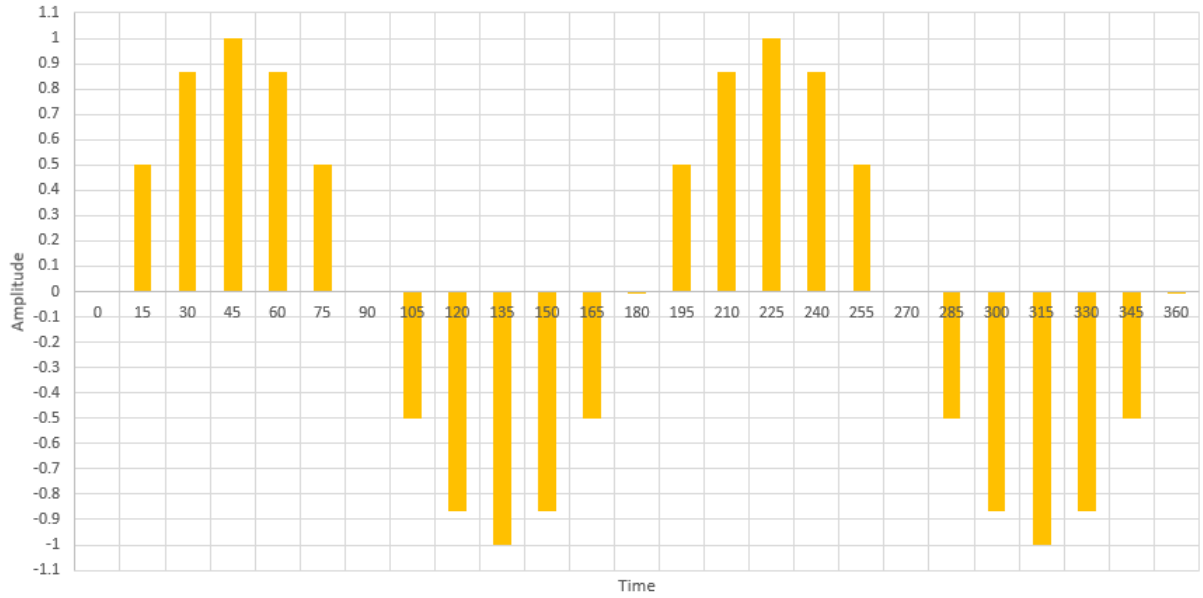


Figure 3.3: Result after quantization of sample set 1.

The interval of quantization is determined by the number of levels each sample can have. Increasing the interval lessens the quantization error, leading to a more detailed signal reconstruction, but it also increases the memory necessary to store those levels. Assuming we have n-bits numbers to assign to each sample, then we will have 2^n levels.

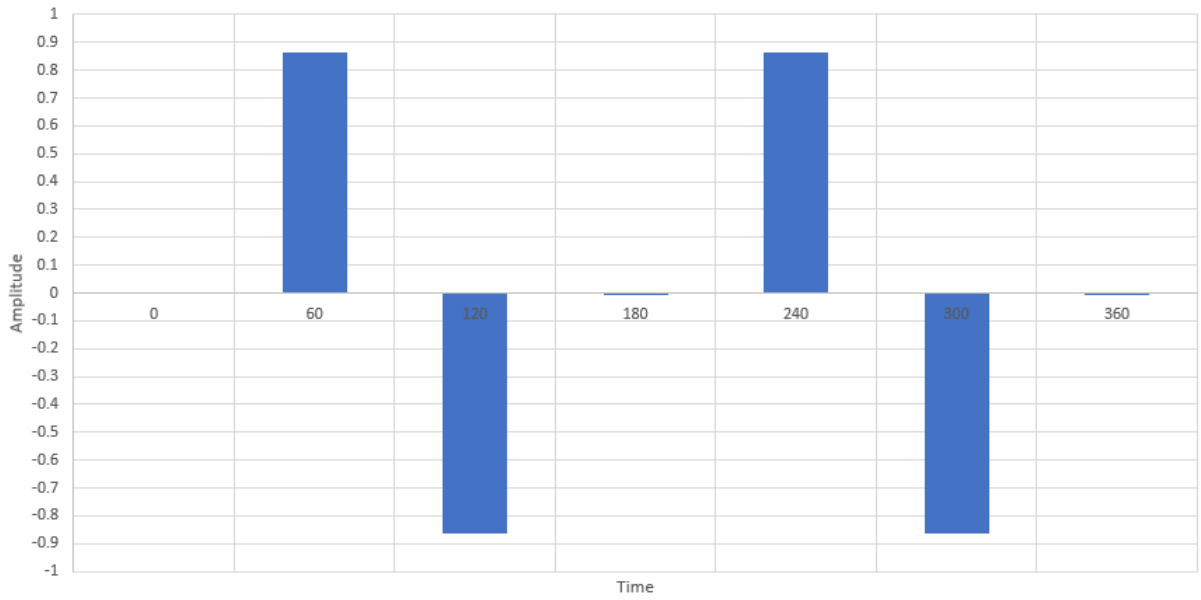


Figure 3.4: Result after quantization of sample set 2.

3.2 Interpolation

Interpolation is a method to find new values given a discrete known set of data points. It is mostly used in geometrical transformations, such as rotations, translations or resizing, to determine the resulting geometric coordinate of each pixel, and in estimating pixel values, based on the surrounding pixels, for the desired coordinates of a digital image.

There are two categories of interpolation algorithms: Adaptive and Non-adaptive. Adaptive methods change their behavior according to the interpolation region, such as sharp edges or smooth textures. This leads to a reduction in blurring effect and image quality improvement. However, these methods come with higher algorithm complexity. On the other hand, non-adaptive methods do not distinguish regions in the image treating all the pixels the same way. Non-adaptive methods have a lower algorithm complexity, but some regions have a decrease in image quality [0] [0]. This thesis work is focused on non-adaptive algorithms, more specifically the nearest-neighbor, bilinear and bicubic interpolations.

3.2.1 Nearest-neighbor interpolation

Nearest-neighbor interpolation is the simplest of the three algorithms. It takes the pixel to be interpolated and assigns it the value of the nearest pixel. The nearest pixel can be determined by rounding up or down the interpolated pixel coordinates. It is a very simple and fast algorithm but it produces a low-quality image [0] [0].

3.2.2 Bilinear interpolation

Bilinear interpolation considers the 2x2 neighborhood of known pixels surrounding the interpolated pixel. It takes the linear average of the 4 neighbor pixels to achieve the final result of the interpolation [0] [0]. As an example, in Figure 3.5 its shown the pixel to be interpolated (red pixel) and the 4 surrounding neighbor pixels (blue pixels). The real value of the red pixel cannot be determined since its coordinates do not coincide with the panel grid. Assuming that from the coordinates of the red pixel (X, Y), x is the decimal part of X and y is the decimal part of Y, and the pixel coordinates (X_1, Y_1), (X_2, Y_2), (X_3, Y_3) and (X_4, Y_4), correspond to values of V_1, V_2, V_3 and V_4, respectively. The generic bilinear interpolation formula is

$$f(a,b) = (1-x) \times a + x \times b \tag{3.2}$$

By interpolating in the x-axis, $A_1 = f(V_1, V_2)$ and $A_2 = f(V_3, V_4)$, where A_1 is the interpolation for the upper segment and A_2 the interpolation for the lower segment, and then in the y-axis, $V = f(A_1, A_2)$, the value for the red pixel V can be determine.

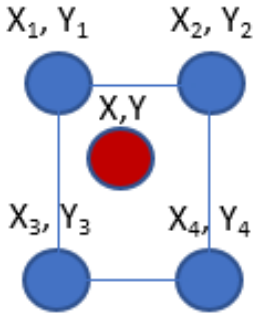


Figure 3.5: Grid for bilinear interpolation.

3.2.3 Bicubic interpolation

Bicubic interpolation considers the 4x4 known pixels neighborhood of the interpolated pixel. It produces noticeably sharper images than the previous two methods, and is perhaps the ideal combination of processing time and output quality [0] [0]. In Figure 3.6 we can see the pixel to be interpolated (red pixel) and the sixteen surrounding neighbor pixels (blue pixels).

Assuming that from the coordinates of the red pixel (X, Y), x is the decimal part of X and y is the decimal part of Y each pixel position corresponds to a value, each pixel coordinate corresponds to a pixel value V_x and each row is comprised of four values. The generic bicubic interpolation formula is defined as

$$g(a,b,c,d) = b + x \times (0.5 \times c - 0.5 \times a) + x^2 \times (2 \times c + a - 2.5 \times b - 0.5 \times c) + x^3 \times (0.5 \times d + 1.5 \times b - 0.5 \times a - 1.5 \times c) \tag{3.3}$$

By interpolating each row in the x-axis, $A_1 = g(V_1, V_2, V_3, V_4)$, $A_2 = g(V_5, V_6, V_7, V_8)$, $A_3 = g(V_9, V_{10}, V_{11}, V_{12})$ and $A_4 = g(V_{13}, V_{14}, V_{15}, V_{16})$, where each A_x is the interpolation of

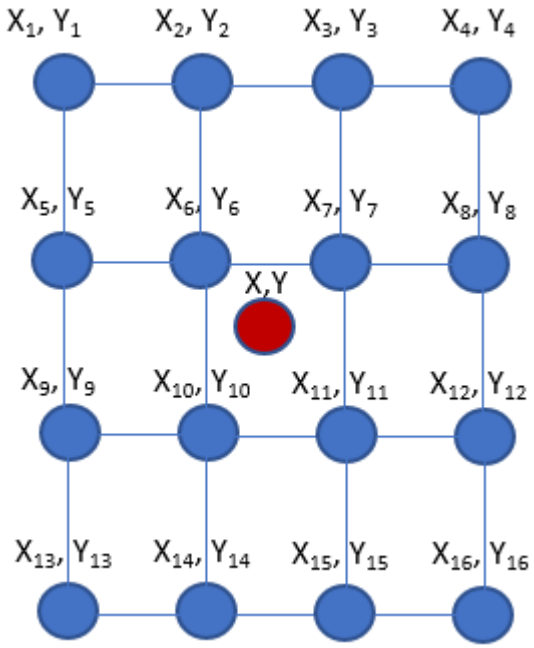


Figure 3.6: Grid for bicubic interpolation.

each segment, and then in the y-axis, $V = g(A_1, A_2, A_3, A_4)$, we can determine V, the value for the red pixel.

3.3 Color Model Transformation

An image can be represented in different color models, and each of them can make images more suitable for digital image and computer vision processing. To select the correct color model to represent an image, we need to know what type of problem is being addressed and what information needs to be extracted from the image [0].

The color models addressed in this thesis are: Red Green Blue (RGB), YCrCb, Hue Saturation Value (HSV), Grayscale and Binary representation.

3.3.1 RGB Color Model

The most common color model is RGB, which represents an image with 3 color layers, the red layer (R), the green layer (G) and the blue layer (B). Each layer is composed of 256-bits. Every pixel in the image is the combination of these layers, and can reproduce a broad array of colors. A common use for this color model is the display of colors in televisions, computer monitor or large screens.

3.3.2 YCrCb Color Model

The YCrCb color model represents an image by its luminance (Y), red chrominance (Cr) and blue chrominance components (Cb). Luminance corresponds to the brightness of the

color, the red chrominance is the relationship between the image red component and the green component and the blue chrominance is the relationship between the image blue component towards the green component. This color model is obtain through conversion from RGB with the following formula,

$$\begin{bmatrix} Y \\ Cr \\ Cb \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ 0.701 & -0.587 & -0.144 \\ -0.299 & -0.587 & 0.886 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3.4}$$

A popular use for YCrCb color model is the compression of images for storage or transportation [0] [0].

3.3.3 HSV Color Model

HSV is the color system that describes colors as perceived by human beings. Hue is the chromatic component of our perception in reference to a color. In this color model exists 6 unique hues, red, yellow, orange, green, blue and purple, as seen in Figure 3.7. The value of Hue is the degree that best describes the chromatic value of an object [0] [0] [0]. Saturation refers to the quantity of white light mixed in a color. A pure color like red is fully saturated, that is, it has no white light mixed into it. On the other hand, the pink color (red and white) is less saturated since it has some light (white) mixed with the red color. In Figure 3.7 it can be seen the relation between a specific color and the saturation variation [0] [0] [0]. Value is a measurement of the amount of light that is emitted or reflected from an object. As seen in Figure 3.7 the lower the the amount of emitted light the darker the color and that translates to a lower Value [0] [0] [0].

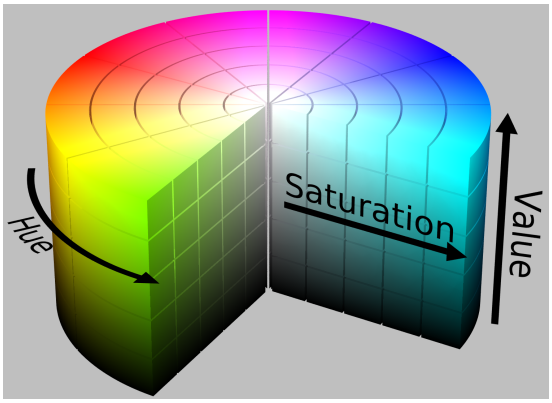


Figure 3.7: HSV color model [0].

Certain applications and methods transform an image from RGB to HSV for digital image and computer vision processing. By doing so, it leads to great advantages when identifying and processing information, related to color in an image. Given a pixel with (R, G, B) values, its normalized values (r, g, b) are given by,

$$r = \frac{R}{R+G+B}, \; g = \frac{G}{R+G+B} \text{ and } b = \frac{B}{R+G+B}.$$

After normalization, the following formulas are applied to determine the (H, S, V),

$$V = \max(r, g, b) \tag{3.5}$$

$$S = \begin{cases} 0 & \text{if } V = 0 \\ V - \frac{\min(r,g,b)}{V} & \text{if } V > 0 \end{cases} \tag{3.6}$$

$$H = \begin{cases} 0 & \text{if } S = 0 \\ \frac{60 \times (g-b)}{S \times V} & \text{if } V = r \\ 60 \times \left[2 + \frac{b+r}{S \times V}\right] & \text{if } V = g \\ 60 \times \left[4 + \frac{r-g}{S \times V}\right] & \text{if } V = b, \end{cases} \tag{3.7}$$

$$H = H + 360 \qquad \text{if } H < 0.$$

These equations have proven to yield good results [0] [0].

3.3.4 GrayScale Color Model

Other applications benefit from transforming a RGB image to Grayscale. This transformation reduces the information to be processed, since the 3 color layers are converted to 1. This transformation can be done using the following formula

$$G = 0.333 \times R + 0.5 \times G + 0.1666 \times B \,,$$

where G is the resultant gray-scale value, R, G and B represent the red, green and blue components of the input image, respectively [0].

3.3.5 Binary Color Model

Binary representation shows a black and white image, enhancing the contrast between the background and objects. Most commonly the background is represented as black and the objects as white. This representation eases the extraction of object characteristics, for example perimeter, area and shape, since all the noise of the background is highly reduced and the object becomes a homogeneous Blob [0] [0]. There are two categories of binarization, global and local.

In the global binarization, a threshold is defined and all pixels are compared to it. According to that comparison the new values 0 and 1 are assigned to the output pixels, representing black and white, respectively. The threshold is a boundary intensity between the background and the objects in the image, it can be obtained with histogram analysis. This method can be formulated as

$$g(u, v) = \begin{cases} 1 & \text{if } T < f(x, y) \\ 0 & \text{else,} \end{cases} \tag{3.8}$$

where $g(u,v)$ is the binary value at the position (u,v) of the output image, $f(x,y)$ is the intensity at the position (x,y) of the input image and T is the threshold. This is a simple approach with low computational complexity, but yields bad results on images where the range of brightness varies greatly [0] [0].

In the local binarization, the pixel to be processed is compared to its neighborhood. A simple approach is, if the pixel is significantly darker than its neighborhood it is converted to black, otherwise it is converted to white. By using the neighborhood information, the binary value of the pixel to be considered is determined based on local conditions, instead of a single valued threshold. Some methods have been developed based on weighted running average, local contrast measure and local edge approach. This provides a better quality output since the conditions of binarization adapt to each region of the image [0] [0].

3.4 Geometric Transformations

A geometric transformation is an image processing operation that redefines the spatial relationship between points in an image. This facilitates the manipulation of an image's spatial layout, that is, its size and shape. This area has received considerable attention due to its practical importance in remote sensing, medical imaging, computer vision, and computer graphics. It is often used for correcting images for lens distortion, correcting effects of camera orientation or image morphing [0] [0].

3.4.1 Spatial transformation

Spatial transformation defines the geometric relationship or map between each point in the input image and the output image. The input image consists entirely of reference points where the coordinate values are known. On the other hand, the output image is composed of observed data. The general mapping function for the spatial transformation can be given in two forms forward mapping, Equation 3.9 and inverse mapping, Equation 3.10, respectively. The input coordinate system is represented by $[u,v]$, the output coordinate system is represented by $[x,y]$, and X, Y, U and V represent arbitrary mapping functions. Since X and Y map the input image onto the output image they are referred to as forward mapping functions. U and V map the output image onto the input image and are referred to as inverse mapping functions [0] [0].

$$[x,y] = [X(u,v), Y(u,v)] \tag{3.9}$$

$$[u,v] = [U(x,y), V(x,y)] \tag{3.10}$$

Forward mapping consists in relating the input image coordinate system to that of the output image, copying each input pixel onto the output image positions determined by the mapping functions X and Y. Inverse mapping consists in relating the output image

coordinate system to that of the input image, copying each input pixel onto the output image positions determined by the mapping functions U and V. Both these mapping functions require a stage of interpolation when they result in non-integral pixel positions [0] [0].

3.4.2 Transformation Matrix

The general transformation matrix can be expressed by a generic 3x3 matrix, as shown bellow. Since we are only interested in 2D image projections, we can ignore the 3D component in the matrix. When multiplied with [x,y,w], a representation of integral coordinates, it results in the corresponded coordinates [u,v,w'] [0] [0].

$$[u, v, w'] = [x, y, w] \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{3.11}$$

The transformation matrix, in Equation 3.11, can be partitioned in 4 sections for better understanding of its behaviour. The 2x2 matrix

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \tag{3.12}$$

corresponds to linear transformation for local resizing, shearing and rotation. The 1x2 matrix $[a_{31}\ a_{32}]$ corresponds to translation. The 2x1 matrix $[a_{13}\ a_{23}]^T$ corresponds to perspective transformation. Finally the element a_{33} corresponds to overall resizing [0] [0].

3.5 Digital Filters

Digital filters are used to enhance important information in an image. They can be divided in two categories: global and local processing. The first correlates the whole image or a large section of it to obtain a smoothed image. The second uses a sliding window and correlates the pixels that fall under the window, that runs through the whole image [0]. Processing the boundary pixels of an image can impose some difficulties, since some positions of the window might not be populated. Some ways to prevent this issue are ignoring the margins of the image, duplicate the margins as many times as needed, mirror the margins or "warp-around"the image and consider it is cylindrical.

The focus in this thesis are local filters that, reduce noise in an image (Image denoising) and identify edges in an image (Edge detection).

3.5.1 Image Denoising

Reduction of noise is an important process in computer vision and image processing, since noise can lead to misreads when analysing an image. Noise is the variation of

brightness or color that leads to the presence of undesirable information in an image. Noise translates to values, by adding, subtracting or multiplying, that are different from the true value of a pixel. It can be introduced by many means, during image acquisition, transportation, conversion from analog-to-digital, among others [0] [4].

The challenge in suppressing noise is maintaining the fundamental structure of the image, that is, image details, like corners or sharp edges. There are two different types of filters, linear and nonlinear. Linear filters have low computational complexity, but do not preserve edges and other details in an efficient manner, causing blurring and distortion effects. Nonlinear filters add more implementation complexity, but perform better in preserving image details [0].

There are different kinds of noise, and each of them is better dealt with using certain filters.

3.5.1.1 Salt and pepper Noise

Salt and pepper noise is an impulse type of noise, where the corrupted pixel has the probability of assuming the minimum and maximum pixel values. For an 8-bit image, the minimum would be 0 and the maximum 255. The Probability Density Function (PDF) for the noise is,

$$G(p) = \begin{cases} P_a & for \quad p = a \\ P_b & for \quad p = b \\ 0 & otherwise \end{cases} \tag{3.13}$$

where a and b represent the minimum and maximum pixel values, respectively, P_a and P_b represent the probability of a pixel being corrupted with the minimum and maximum values, respectively, and p represents a gray level. The Median filter effectively reduces this type of noise [4] [0] [0].

3.5.1.2 Gaussian Noise

Gaussian noise is an additive type of noise, where the corrupted pixel has its true pixel value added a random value from the Gaussian noise distribution function. This type of noise follows a bell shaped PDF given by,

$$G(p) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(p-\mu)^2}{2\sigma^2}} \tag{3.14}$$

where the function p represents a gray level, μ represents the mean of the function and σ^2 represents the variance of noise. The Mean filter significantly reduces this type of noise [4] [0] [0].

3.5.1.3 Speckle Noise

Speckle noise is a multiplicative type of noise, where the corrupted pixel has its true pixel value multiplied by a random value from a gamma distribution function. The gamma

distribution function is defined as,

$$G(p) = \frac{p^{\alpha-1}}{(\alpha-1)!a^{\alpha}} e^{\frac{-p}{a}} \tag{3.15}$$

where the function $G(p)$ represents the probability for the gray level p, α represents the shape parameter and a represents the scale parameter. Wavelet filters best reduce this type of noise. [4] [0] [0].

3.5.1.4 Poisson Noise

Poisson noise occurs when particles that carry energy, like electrons or photons, in an electric circuit originate detectable random fluctuations in measurements. This noise model can be given as,

$$G(x,y) = \alpha \times P_{\alpha}(x,y) + N_{\alpha}(x,y) \tag{3.16}$$

where $P_{\alpha}(x,y)$ represents a Poisson distribution, $N_{\alpha}(x,y)$ represents a Gaussian distribution and $\alpha > 0$. Similar to the Gaussian noise, the Mean filter reduces the Poisson noise [4] [0] [0].

3.5.1.5 White Noise

White noise is essentially identified by the noise power. The range of noise power is infinite in the frequency domain, and although not fully constant, in the visible spectrum it has a constant intensity, therefore, just like white light, all frequency components are equally present. In white noise, correlation is impossible because every pixel value is different from its neighbor, so auto-correlation is zero [0].

3.5.2 Edge Detection

Edge detection is the process of identifying and locating sharp discontinuities in an image. These discontinuities are abrupt changes in gray pixel intensities which relates to boundaries between objects and the background. Usually it is used as a preparation step for feature extraction and image segmentation. This process can significantly reduce the amount of data to be processed since it filters out less relevant information, while maintaining the important structural image properties [0] [0] [4].

Classical methods involve convolving the image with an operator, a 2D filter, built to be sensitive to large gradients in an image while returning zero in uniform regions. An edge is composed of two parameters, gradient (strength of the edge) and direction (angle of the gradient). Since in real images an object is not a perfect uniform region, upon setting a threshold T and compare it to the gradient of the object pixels we can determine which pixels belong to edges. The geometry of the operators determine the direction which they are most sensitive. They can be optimized to detect horizontal, vertical or diagonal edges. Analysing a noisy image poses difficulties, since both edges and noise contain high frequency content. Attempting to reduce noise mostly leads to blurred and

distorted edges, which can result in poorly edge detection. In those cases, a larger scope operator is used so more true pixels can be averaged and the noisy pixels undermined [0] [0].

Most methods can be grouped into two categories, gradient-based edge detection and laplacian-based edge detection. Gradient methods detect edges by looking for the maximum and minimum in the first derivative of the image. Laplacian methods detect edges by looking for zero crossings in the second derivative of the image [0]. We will focus on gradient methods, such as Differentiation, Roberts and Sobel.

The problems related to edge detection are false edges, missing true edges, locating edges, noisy images, high computational time, among others [0]. These factors determine the best operator to use depending on the results to be achieved.

3.6 Image Enhancement

Image enhancement is the process of removing unwanted distortion due to deterioration in contrast, unwanted noise, improper intensity saturation, blurring and distortion effects caused by lens, and highlight the hidden information contained in an image. The final goal of image enhancement is to transform the input image so that a visually better image output is obtained. This proves of great importance for analysing and processing images for most applications, like biometric analysis, pattern recognition and facial recognition [0].

This section focus on contrast enhancement of an image using histogram equalization.

Contrast enhancement expands the range of brightness values in an image. The density values in a scene are scattered over a greater range. The effect is to increase the visual contrast between areas of different uniform densities. This enable areas previously having a small difference in density, to be more easily distinguished [0].

Contrast refers to the difference in luminance or gray levels in an image. Contrast ratio can be defined as the ratio between the highest and lowest pixel intensities. It is one of the main image properties that makes objects distinguishable from one another. The larger the ratio, the easier it is to interpret the image [0].

Histograms of images provides a description of the image appearance globally. It shows the occurrences of each intensity value in an image. For an 8-bit gray scale image, there are 256 possible intensities, and so the histogram will show those 256 values and the distribution of pixels amongst those values [0].

Histogram equalization is a contrast enhancement method where the histogram of the original image is redistributed to produce an uniform density population. It increases the dynamic range of the histogram of the input image. This method acquires the PDF and Cumulative Distribution Function (CDF) from the input image histogram, applies them to replace the pixel intensities for the new pixel intensities, and generates a new histogram for the output image. The overall image contrast is improved and an uniform histogram is obtained. It can be used globally or locally. Global histogram equalization takes in

consideration the histogram of the whole image, while local histogram equalization only uses pixels under a window to obtain the new intensity of the center pixel [0].

PDF is a function whose value at any given point in the sample space can be interpreted as the relative likelihood that the value of a random variable would equal that sample. It can be given as,

$$P(r_i) = \sum_{i=0}^{k} \frac{n_i}{n} \tag{3.17}$$

where n_i represents the number of pixels with intensity r_i, n is the total number of pixels in the image and k represents the range of pixel intensities. For an 8-bit gray scale image, k can vary from 0 to 255 [7] [0]. For example, in a bag with 10 balls, where 7 are yellow and 3 are red, the PDF for a yellow ball is 70%.

CDF represents the probability that a real-value random variable X will take a value less or equal to x. This function is bound to the PDF, in a manner that, a CDF probability corresponds to the sum of probabilities from the PDFin the range of 0 to x. It can be given as,

$$s_k = C(r_k) = \sum_{i=0}^{k} P(r_i)$$

where $P(r_i)$ represents the PDF for a pixel intensity and k represents the range of pixel intensities [7] [0]. For example, the probability of a random variable X being equal or lower then 0.5 is the sum of all probabilities from 0 to 0.5 in the PDF.

3.7 Morphological Transformations

Mathematical morphology is related to a branch of biology that deals with forms and structures of animals and plants. When applied to image processing and computer vision, it is used as a tool to extract useful image data in the representation and description of object shape, while preserving the essential shape characteristics and eliminating non-essential data, like noise or imperfections [0] [0]. This subject has two categories, morphological transformations and skeletonization techniques.

The language of mathematical morphology is set theory. In the context of this topic, sets are objects, that either are present in an image or act has a structuring element. They represent shapes, such as boundaries and skeletons of objects [0].

Morphological transformations involve geometric analysis of shapes and textures in images. Morphological operators work with two components, the active image, and the structuring element. The first is the image to be processed and the second is a kernel with a designated shape, with similar behaviour of a filter to the active image. The more basic operations are dilation and erosion, which can be combined in sequence and create new operations, such as opening and closing [0].

Dilation is the morphological transformation that combines two sets using vector addition of two set elements. It is known for thickening or expanding objects. Erosion

is the dual to dilation, it combines two sets using vector subtraction. It is known for shrinking or thinning objects. Opening is the combination of erosion and dilation, in this order. It is generally used to break narrow sections, smooth object contour and eliminate protrusions. Closing is the combination of dilation and erosion, in this order. It is generally used to fill gaps and holes and fuse narrow breaks [0] [0].

Skeletonization is the process of extracting skeletons from an object in a binary image. It is a morphological operation that deletes foreground pixels iteratively layer by layer, thinning the object, until a one-pixel width skeleton is obtained. By reducing an object to only a skeleton important features like, end-points and connection among the components are preserved, and unimportant information and image noise can be filtered out [0].

3.8 Image Segmentation

Segmentation refers to the process of partitioning an image into segments, such as regions or objects. These segments are sets of pixels that are similar according to some homogeneity criteria, such as color, intensity or texture. The chosen criteria will determine the objects to locate and identify in the image. The main purpose for image segmentation is to isolate areas of interest in an image for a more efficient analysis. In some cases, some denoising may need to be applied before segmentation, so no false contours or irrelevant information are highlighted in the final image [0] [0].

Segmentation algorithms are based in two pixel intensity properties, discontinuity and similarity. The first relates to partitioning the image based on sudden changes in intensity, like edges and lines. The second, according to some predefined criteria, uses similarity in regions to partition the image [0].

This section focus on region-based algorithms, with a brief description of edge-based algorithms.

3.8.1 Edge-based Algorithms

Edges and lines are sets of linked pixels that represent the boundary between two distinctly different regions and the shape of objects. An image can be segmented by detecting these discontinuities in the image. The difference between an edge and a line is that, a line may be embedded inside an object as opposed to an edge, so lines are not reliable information for these type of algorithms [0].

Edge-based methods convolve gradient operators with the image. High gradient values indicate regions of abrupt change in pixel intensity, that is edge pixels. These pixels, if linked, form close boundaries for objects. Common edge detection operators used are Canny, Laplace, Laplacian of Gaussian and, the previous mentioned, Differentiation, Roberts and Sobel. These methods relay better results depending on the efficiency of the edge detection method [0] [0].

3.8.2 Region-based algorithms

Region-based algorithms segment an image into regions that are similar according to a predefined criteria. In comparison with edge-based algorithms, they are simpler and more immune to noise. The main methods are thresholding and region operating [0].

Thresholding or binarization methods convert a multi-level image into a binary image by selecting the pixels that are in the intensity range imposed by a threshold T. Those pixels belong to the objects in the scene. Selection of a threshold T can be done in ways, globally or locally. Global thresholding assumes a constant value T for the whole image. Common, and implemented, methods are Otsu algorithm, K-means and Gray histogram technique. Local thresholding generates a threshold value for each region of the image. Common methods are Wellner algorithm, Bradley and Roth algorithm, statistical thresholding and entropy-based thresholding [0] [0].

Region operating methods find the objects directly from the image. They group pixels in the image into sub regions following a predefined criteria. These kind of methods are of iterative nature, so they require more computational time. Some examples are Region Growing, Connected Component Labeling and Watershed Transform [0].

3.9 Feature Extraction

Feature extraction is the process of reducing objects to their distinctive shape characteristics. The main goal of feature extraction is to obtain the most relevant information from objects and represent that information in a reduced format. The selected information is denoted as features [0] [0].

Features can be classified into: Local features and Global features. Local features are usually geometric properties, such as area, perimeter and compactness. Global features are usually topological properties, such as connectivity and projection profiles, and statistical properties, such as moments. Object features are primarily used for pattern recognition [0].

Pattern recognition focus on taking input data and correctly assign it to one of the possible output classes. This process can be decomposed in two steps: Feature selection and Classification.

Feature selection is a critical step in pattern recognition, since the higher the quality of the features the more efficient is the classification process. There are two stages for feature selection. First, features with potential for classification are extracted. Secondly, a subset of the most relevant features is created from the original feature set. In order for a feature subset to be useful for classification, it should have four important properties:

1. Define a complete set,
2. Be congruent,
3. Have invariant properties,

4. Be a compact set.

The first denotes that, two objects have the same feature set, if and only if, they have the same shape. The second denotes that, similar objects should have similar features. The third denotes that, the chosen features should yield the same results regardless of image variations. For example, when an object is rotated or resized. The fourth denotes that, the feature set should represent the essential and minimal information to identify an object as unique or different from other objects [0] [0].

Feature selection can lead to some problems for classification. These problems are mostly related to the computational time in selecting the optimal combination of features from the original feature set for the classification process, and determining which feature is more meaningful for each object class [0].

Classification is the step where a class is associated with a sample set by using the features that describes it. The main idea is to find the class that differs the least amount from the sample set. One example of how to do this evaluation can be in terms of Euclidean distance, using the difference d between the measurements of the sample feature set, s, and the measurements of a known class feature set, k,

$$d = \sqrt{\sum_{i=1}^{M} (s_i - k_i)^2} \tag{3.18}$$

where M is the number of features. The class with the lower distance d is assigned to the sample set [0].

4 Methods

This chapter presents an overview on how to use the develop library and of the algorithms developed for this thesis. It explains the process behind each algorithm and shows practical examples for a better understanding. Each subsection title of this chapter is followed by the reference to the respective code snippet available in Annex I.

4.1 Library Tutorial

The library is composed of two main files, test.py where the library functions are executed and PIPpackage.py where the functions are implemented.

In test.py the PIPpackage.py file is imported and the functions that are going to be executed, with their respective input parameters, are invoked. In every function an input image is required. This input image is a 2D array where every position represents an image pixel and each value a pixel intensity. The input image can be created using a normal array, as shown in Figures 4.1 and 4.2, imported from an image file using function imread() from python library matplotlib, or using function genImage() from the image processing library. Functions that require more input parameters have proper documentation and reading their description will show the user the requirements to execute the function.

```
img = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

Figure 4.1: Binary image of a rectangle.

```
img = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0],
                [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0],
                [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                [0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
                [0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

Figure 4.2: Binary image of various objects.

Using the legacy function print() from python library and the invoked image processing function as the input parameter, shows in the output console the result of the image processing function invoked.

Some examples of how to run the image processing library functions are present in test.py file.

4.2 Geometric Operations

Geometric operations manipulate the spatial layout of an image, that is, its size and shape. By applying the transformation matrix we can express many spatial transformations. Here we will focus on translation, rotation and resizing. If these formulas are applied directly we will be in a scenario of forward mapping. If they are inverted we will be in a case of inverse mapping.

4.2.1 Translation [3]

To perform a translation all pixels in the input image will be shifted by offsets of Δx and Δy to the x and y coordinates, respectively, creating a new pair of coordinates u and v for each pixel. The transformation matrix will have the following structure,

$$[u, v, 1] = [x, y, 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta x & \Delta y & 1 \end{bmatrix} \tag{4.1}$$

The above formula can be translated to,

$$u = x + \Delta x v = y + \Delta y \tag{4.2}$$

4.2.2 Rotation [4]

To perform a rotation all pixels in the input image are rotated θ degrees clockwise around the origin of the coordinate system, creating a new pair of coordinates u and v for each

pixel. The transformation matrix will have the following structure,

$$[u, v, 1] = [x, y, 1] \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

The above formula can be translated to,

$$u = x \times \cos\theta + y \times \sin\theta$$
$$v = y \times \cos\theta - x \times \sin\theta$$

4.2.3 Resizing [5]

To perform image resizing all pixel coordinates in the input image are scaled by the factors of S_x and S_y to x and y coordinates, respectively, creating a new pair of coordinates u and v for each pixel. If the factors have a negative value the image is reflected, resulting in a mirror image. If the factors are not identical, the resulting image will have its proportions altered, yielding in a different size image. The transformation matrix will have the following structure,

$$[u, v, 1] = [x, y, 1] \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

The above formula can be translated to,

$$u = x \times S_x v = y \times S_y \tag{4.5}$$

4.3 Linear Filters

Linear filters follow two properties, scaling and superposition. The first says that the output signal α is proportional to the amplitude of the input signal $A\alpha$, and the second says that when two signals A and B are added together and fed to the filter, $A + B$, the filter output is the same as if each signal passed through the filter separately and then the outputs were added.

One classical linear filter, the Mean filter, was implemented with three different approaches, A, B and C. The filter has the intend to reduce specific types of noise in an image, at the cost of blurring and distorting some details.

4.3.1 Mean Filter A [6]

The A algorithm is the simplest of the three. At every pixel of the image, the sum of the center pixel and its neighbors is taken and stored. This sum is going to be denominated S. The center pixel intensity is replaced with the average of S. Although this method has a simple implementation, it takes a lot more computational time than the others, B and C. It makes nine operations to calculate the new center pixel intensity.

4.3.2 Mean Filter B [7]

The B algorithm adds a complexity layer to the previous algorithm. The new intensity for the first pixel is computed as in algorithm A. For the rest of the image the next pixel intensity is calculated by taking the last calculated S and subtracting the first column and adding the new column of the sliding window, and finally take its average. For example, if Figure 4.3 represents an image section where margins are to be ignored, the first iteration of the algorithm is represented in Figure 4.4 and the second iteration in Figure 4.5. In the first, S is calculated by summing all pixels in the green window, and then the center pixel 44 is replaced by the average. In the second, it subtracts the red column and adds the yellow column to the previous S, and then the center pixel 204 is replaced with the average. This method takes, in average, six operations to calculate the new center pixel intensity.

125	247	108	99
54	44	204	104
164	187	133	200

Figure 4.3: Example image section.

125	247	108	99
54	44	204	104
164	187	133	200

Figure 4.4: First pixel treatment.

125	247	108	99
54	44	204	104
164	187	133	200

Figure 4.5: General pixel treatment.

4.3.3 Mean Filter C [8]

The C algorithm is an improved version of the previous ones. The first pixel is treated with the A algorithm, and the rest of the first row with the B algorithm. The first column is treated with a similar algorithm from the B algorithm, from the sum S from the above pixel window is subtracted its first line and added the new line. An example is shown in Figure 4.6, where the S from pixel 44 is subtracted the red line and added the yellow line. The rest of the image is treated with the following formula,

$$S_o = S_b - S_a + S_c + a - b - c + d \tag{4.6}$$

where S_o is the sum of the current center pixel and its neighbors, S_a, S_b and S_c are the sums of previous calculated windows, and a, b, c and d isolated pixel intensities.

Figure 4.7 depicts the above formula, being A the window for S_a,represented as red, B the window for S_b, as blue, C the window for S_c, as green, and O the window for S_o, as a dashed line.

125	247	108	99
54	44	204	104
164	187	133	200
48	260	178	100

Figure 4.6: Column pixels treatment.

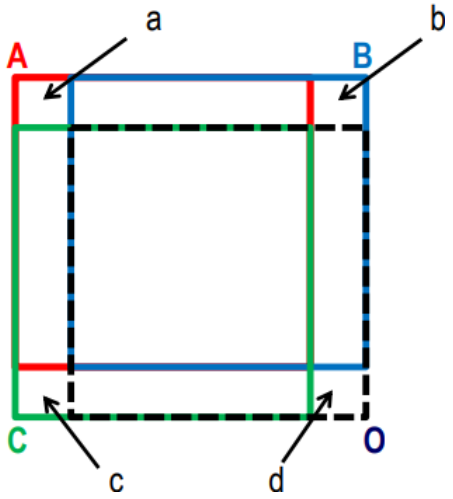


Figure 4.7: Mean Filter C exemplification [2].

This algorithm, unlike algorithms A and B, has constant execution time regardless of the sliding window size.

4.3.4 Integral Image

The integral image or summed area table, is the sum of all the pixels before each pixel in an up left rectangle, as seen in Figure 4.9, and demonstrated in the following equation,

$$I(x,y) = f(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1) \tag{4.7}$$

where $I(x,y)$ represents the integral image and $f(x,y)$ is the original image.

1	4	3	7
0	4	8	2
5	2	0	5
1	0	0	9

Figure 4.8: Example image.

1	5	8	15
1	9	20	29
6	16	27	41
7	17	28	51

Figure 4.9: Integral Image of the example image.

This method is an effective way of calculating the sum pixel values in an image. The Mean C filter algorithm benefits of this method since it can use the integral image to calculate the sum of the current center pixel and its neighbors.

4.4 Nonlinear Filters

In these type of filters, as opposed to linear filters, when the signals a and b pass through it and outputs A and B, respectively, the signal $a + b$ might not output the signal $A + B$. These example can also be applied to images.

Seven classical nonlinear filters were implemented: Nonlinear Mean filter, Nonuniform Mean filter, Median filter, Hybrid-Median filter, K-Nearest Mean filter, Sigma filter and Mode filter. These filters were designed to reduce specific types of noise in an image with improved performance in comparison to the linear filters.

4.4.1 Nonlinear Mean Filter [10]

The Nonlinear Mean filter is a sliding MxM window spatial filter that scans the whole image. The filter takes a threshold value T that determines which pixels are to be replaced with the average sum of their neighboring pixels. This behaviour can be translated to the following formula,

$$g(x,y) = \begin{cases} \frac{1}{M} \times \sum_{(n,m)\in S} f(n,m) & \text{if} \quad |f(x,y) - \frac{1}{M} \times \sum_{(n,m)\in S} f(n,m)| < T \\ f(x,y) & \text{otherwise} \end{cases} \tag{4.8}$$

4.4.2 Nonuniform Mean Filter [9]

The Nonuniform Mean filter follows the same sliding window principle as the previous filter. The filter replaces the center pixel with the average sum of the neighbor pixels, including itself. The difference with the linear Mean filter is the weighted coefficients in the MxM spatial window. The coefficients allow to give more weight to certain aspects of the filter, when calculating the center pixel intensity. An example can be a matrix where it is given more weight to the center element, and lower weight to the corners,

1	2	1
2	4	2
1	2	1

4.4.3 Median Filter [14]

The Median filter follows the same sliding window principle as the previous filters. The filter calculates the median of the pixels that fall under the window, and replace the center pixel with it. This is done by sorting the pixel intensities in order and take the middle value as the new center pixel intensity. There were implemented three different sorting algorithms, complete BubbleSort [11], quick BubbleSort [12] and MinMax [13].

The complete BubbleSort sorts an array of numbers by repeatedly swapping the adjacent elements if they are in the wrong order. This is done until no changes in the array occur. For example, the array [6, 1, 7, 3, 2] would be sorted in four iterations with sixteen comparisons, as seen in the example bellow.

1. [**61**732] → [1**67**32] → [16**73**2] → [163**72**] → [16327]

2. [**16**327] → [1**63**27] → [13**62**7] → [132**67**] → [13267]

3. [**13**267] → [1**32**67] → [12**36**7] → [123**67**] → [12367]

4. [**12**367] → [1**23**67] → [12**36**7] → [123**67**] → [12367]

The quick BubbleSort works in the same way that the complete BubbleSort, but stops when the array is sorted up to the middle element. For example, with the array [6, 1, 7, 3, 2], the algorithm would stop when the array is sorted up to the third element. This would be done in three iterations with ten comparisons.

In general the MinMax algorithm is a more complex and deep function comparing to the others, but in the context of sorting it has a simple approach. In the first iteration, it takes the array and finds the smallest and largest numbers and place them in the appropriate positions. In the second iteration, ignores the already sorted numbers, and finds the second smallest and largest numbers and place them in the appropriate positions. These sequence goes on until the whole array is sorted. For example, the array [6, 1, 7, 3, 2] would be sorted in two iterations with two comparisons.

1. [6**17**32] → [16237]

2. [1**62**37] → [12367]

4.4.4 Hybrid-Median Filter [22]

This filter is an improved version of the Median filter where the edges are better preserved with the same computational time. The filter sorts and calculates the median from the diagonal intensities from the center pixel, the median from the horizontal and vertical intensities from the center pixel and finally the median between the two obtained medians and the center pixel. This last median is the new intensity for the center pixel. For example, in a window

127	128	211
11	213	107
168	170	43

the diagonal array would be [127, 213, 107, 168, 211] and the horizontal and vertical array would be [128, 213, 170, 11, 107]. Their medians are 168 and 128, respectively. The final array is [213, 168, 128], with the median and center pixel intensity 128.

4.4.5 K-Nearest Mean Filter [20]

The K-Nearest Mean filter, as the above filters, uses the sliding MxM dynamic window. The filter calculates the averaging sum of the pixels that fall under the window and follow

a specific rule. It selects the K pixels that have intensities closest to the center pixel intensity, including the center pixel itself to this selection. For example, in a window

152	180	145
98	140	130
143	122	100

with $K = 4$ and the rule being *"if |f(x,y) - P| ≤ T then selected "*, where P is the center pixel intensity and T is the threshold value, in this case $P = 140$ and $T = 20$. The selected values would be [130, 140, 145, 152] and their averaging sum and new value for the center pixel is 141.

Choosing K and T, depends on the image structure and desired output. The lower the values, more details are preserved but less noise is removed.

4.4.6 Sigma Filter [21]

The Sigma filter works in a similar manner as the K-Nearest Mean filter, it uses the sliding MxM dynamic window and selects pixels that fall under the window and follow a specific rule. In this case, the rule is all selected pixels must lie within a two standard deviation range of the central pixel. Upon selecting the pixels, the average sum is the new intensity for the center pixel. The generic formula for the algorithm is,

$$g(x,y) = \frac{1}{K} \times \sum_{n \in S} s_n \quad where\ S = \left\{ n | P - 2\sigma \leq s_n \leq P + 2\sigma \right\} \tag{4.9}$$

where K is the number of selected pixels, P is the intensity of the center pixel and σ is the standard deviation.

4.4.7 Mode Filter [15]

The Mode filter uses the sliding MxM window dynamic and uses all the pixels that fall under the window. This algorithm selects the new center pixel intensity by choosing the pixel intensity with more occurrences. If a stalemate occurs, the tiebreak can be decided by a rule of the users choice.

4.5 Edge Detection Operators

Four classical edge detection methods were implemented: Differentiation, Roberts, Prewitt and Sobel operators. Each of them presents different computational times and image outputs, being the Differentiation operator the simplest but yields worse results, and the Sobel operator more complex with better edge detection.

4.5.1 Differentiation [16]

The Differentiation operator provides an approximation of the gradient magnitude using the following formula,

$$G(x,y) = |f(x,y) - f(x+1,y)| + |f(x,y) - f(x,y+1)| \tag{4.10}$$

where (x,y) represents the pixel position and $f(x,y)$ represents the intensity of the pixel. Using convolution windows we get,

$$G(x,y) = |G_x| + |G_y| \tag{4.11}$$

where G_x and G_y are calculated as follows,

$$G_x = \begin{array}{|c|c|}\hline 1 & -1 \\ \hline 0 & 0 \\ \hline \end{array} \quad G_y = \begin{array}{|c|c|}\hline 1 & 0 \\ \hline -1 & 0 \\ \hline \end{array} \tag{4.12}$$

4.5.2 Roberts [17]

The Roberts operator provides an approximation of the gradient magnitude using the following formula,

$$G(x,y) = |f(x,y) - f(x+1,y+1)| + |f(x+1,y) - f(x,y+1)| \tag{4.13}$$

where (x,y) represents the pixel position and $f(x,y)$ represents the intensity of the pixel. Using convolution windows we get,

$$G(x,y) = |G_x| + |G_y| \tag{4.14}$$

where G_x and G_y are calculated as follows,

$$G_x = \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & -1 \\ \hline \end{array} \quad G_y = \begin{array}{|c|c|}\hline 0 & -1 \\ \hline 1 & 0 \\ \hline \end{array} \tag{4.15}$$

4.5.3 Prewitt [18]

The Prewitt operator provides an approximation of the gradient magnitude using the following formula,

$$G(x,y) = |S_x| + |S_y| \tag{4.16}$$

where (x,y) represents the pixel position, $f(x,y)$ represents the intensity of the pixel, S_x and S_y represent the vertical and horizontal component of the gradient, respectively. Assuming that the following schema illustrates a 3x3 window of an image pixel and its neighbors,

a	b	c
d	e	f
g	h	i

S_x and S_y are formulated as,

$$S_x = (c + f + i) - (a + d + g) S_y = (a + b + c) - (g + h + i) \tag{4.17}$$

Using convolution windows we get,

$$G(x, y) = |G_x| + |G_y| \tag{4.18}$$

where G_x and G_y are calculated as follows,

$G_x =$

-1	0	1
-1	0	1
-1	0	1

$G_y =$

1	1	1
0	0	0
-1	-1	-1

4.5.4 Sobel [19]

The 3x3 Sobel operator, similar to the Prewitt operator, provides an approximation of the gradient magnitude using the following formula,

$$G(x, y) = |S_x| + |S_y| \tag{4.19}$$

where (x, y) represents the pixel position, $f(x, y)$ represents the intensity of the pixel, S_x and S_y represent the vertical and horizontal component of the gradient, respectively. Assuming that the following schema illustrates a 3x3 window of an image pixel and its neighbors,

a	b	c
d	e	f
g	h	i

S_x and S_y are formulated as,

$$S_x = (c + 2 \times f + i) - (a + 2 \times d + g)$$
$$S_y = (a + 2 \times b + c) - (g + 2 \times h + i)$$

Using convolution windows we get,

$$G(x, y) = |G_x| + |G_y|$$

where G_x and G_y are calculated as follows,

$G_x =$

-1	0	1
-2	0	2
-1	0	1

$G_y =$

1	2	1
0	0	0
-1	-2	-1

4.6 Histogram Equalization [23]

The general implementation of the global histogram equalization is done in three steps:

1. Get image histogram information
2. Calculate new pixel intensity
3. Create a new histogram and output image

In step 1, the image is scanned and the number of occurrences of each pixel intensity is stored.

In step 2, the PDF for each pixel intensity is calculated, alongside with each pixel CDF. A range of 0 to 100, now referred to by S, is partitioned in L+1 blocks, being L the maximum intensity a pixel can have. To each block is assigned a pixel intensity, from 0 to L. Each pixel s_k probability will fall under a specific block, and so that pixel will take the intensity assigned for that block.

In step 3, a new histogram is generated and an output image is created with the new pixel intensities.

For example, if we have a 64x64 image, 4096 pixels, and 8 levels of pixel intensities, from 0 to 7, with the histogram in Table 4.1, the PDF and CDF for each intensity is displayed in Tables 4.2 and 4.3, respectively.

r_i	n_i
$r_0 = 0$	790
$r_1 = 1$	1023
$r_2 = 2$	850
$r_3 = 3$	656
$r_4 = 4$	329
$r_5 = 5$	245
$r_6 = 6$	122
$r_7 = 7$	81

Table 4.1: Image Histogram.

r_i	$P(r_i)$
$r_0 = 0$	19%
$r_1 = 1$	25%
$r_2 = 2$	21%
$r_3 = 3$	16%
$r_4 = 4$	8%
$r_5 = 5$	6%
$r_6 = 6$	3%
$r_7 = 7$	2%

Table 4.2: PDF of the input image.

Partitioning S into 8 blocks and by correlating the information from Table 4.3, each s_k probability falls under a specific block, as seen in Figure 4.10. The new values for each pixel intensity can be seen in Table 4.4 and the histogram and PDF for the output image in Table 4.5.

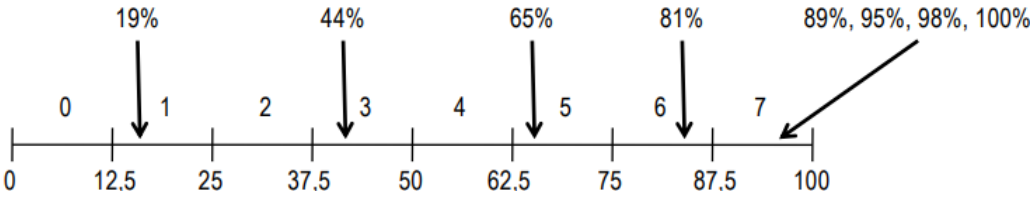


Figure 4.10: Deduction of the new pixel intensities [2].

s_k	$C(r_k)$
s_0	19%
s_1	44%
s_2	65%
s_3	81%
s_4	89%
s_5	95%
s_6	98%
s_7	100%

Table 4.3: CDF of the input image.

s_k	
s_0	1
s_1	3
s_2	5
s_3	6
s_4	7
s_5	7
s_6	7
s_7	7

Table 4.4: New values for each pixel intensity.

r_i	n_i	$P(r_i)$
$r_0 = 0$	0	0%
$r_1 = 1$	790	19%
$r_2 = 2$	0	0%
$r_3 = 3$	1023	25%
$r_4 = 4$	0	0%
$r_5 = 5$	850	21%
$r_6 = 6$	656	16%
$r_7 = 7$	777	19%

Table 4.5: Histogram and PDF of the output image.

4.7 Morphological Operations

Morphological transformations are simple operations based on image shape. It is normally performed on binary images. It needs two inputs, one is the original image and the second is the structuring element. Two basic morphological operators are erosion and dilation, that in sequence may perform opening or closing.

For the following definitions, let A and B denote 2 sets in E^N with elements a and b, respectively, where $A = (a_1, a_2, a_3, ..., a_N)$ and $B = (b_1, b_2, b_3, ..., b_N)$ being N-tuples of element coordinates.

4.7.1 Dilation [29]

The binary dilation of A by B is the set of all possible vectors where the intersection of A with B, in every point, is a non null set. The dilation of A by B, denoted by $A \oplus_b B$, is defined as

$$A \oplus_b B = \{c \in E^N | c = a + b \text{ for some a} \in A \text{ and } b \in B\} \tag{4.20}$$

For example if $A = \{(0,3),(1,1),(2,0),(2,1),(3,3)\}$ and $B = \{(0,0),(0,1)\}$, then $A \oplus_b B = \{(0,3),(1,1),(1,2),(2,0),(2,1),(2,2),(3,3)\}$, as seen in figures 4.11, 4.12 and 4.13 respectively.

0	0	0	1
0	1	0	0
1	1	0	0
0	0	0	1

Figure 4.11: A set for dilation.

1	1

Figure 4.12: B set for dilation.

0	0	0	1
0	1	1	0
1	1	1	0
0	0	0	1

Figure 4.13: Result of dilation.

4.7.2 Erosion [30]

The binary erosion of A by B is the set of all possible vectors where the intersection of A with B, in every point, is B. The erosion of A by B, denoted by $A \ominus_b B$, is defined as

$$A \ominus_b B = \{x \in E^N | x + b \in A \text{ for every } b \in B\} \tag{4.21}$$

For example if $A = \{(0,1),(0,3),(2,0),(2,1),(3,1),(3,3)\}$ and $B = \{(0,0)\}$, then $A \oplus_b B = \{(0,1),(2,1),(3,1)\}$, as seen in figures 4.14, 4.15 and 4.16 respectively.

0	1	0	1
0	0	0	0
1	1	0	0
0	1	0	1

Figure 4.14: A set for erosion.

1	0

Figure 4.15: B set for erosion.

0	1	0	0
0	0	0	0
0	1	0	0
0	1	0	0

Figure 4.16: Result of erosion.

4.7.3 Opening

The opening of A by B is the sequence of the binary operation erosion and dilation. This operation, denoted $A \circ B$, is defined as,

$$A \circ B = (A \ominus B) \oplus B \tag{4.22}$$

4.7.4 Closing

The closing of A by B is the sequence of the binary operation dilation and erosion. This operation, denoted $A \bullet B$, is defined as,

$$A \bullet B = (A \oplus B) \ominus B \tag{4.23}$$

4.7.5 Hit-and-Miss [31]

The hit-and-miss operation of X, being X a set in E^N with element x, is the intersection of the erosion of X by A and the erosion of the complement of X by B. This operation, denoted $X(A, B)$, is defined as,

$$= (X \ominus A) \cap (\overline{X} \ominus B) \tag{4.24}$$

where $A \cap B = \varnothing$.

4.7.6 Skeletonization

Thinning is a process of reducing the image components approximating to the center skeleton, in order to obtain the most basic features of the objects while maintaining the original form which is used to infer shape and topology. The skeleton of an object is a line connecting points, junctions and endpoints. The main objective of skeletonization is to reduce image data and facilitate structural analysis and pattern recognition. The most common application of skeletonization is in binary images.

There were implemented two skeletonization algorithms: Medial Axis Transform and Zhang-Suen.

4.7.6.1 Medial Axis Transform [32]

The Medial Axis Transform is one of the first algorithms with the intend to describe the shape of an object. The algorithm can be defined as follows: given an object, for example a simple polygon A, the medial axis $M(A)$ is the set of points m interior of A such that there are at least two boundary points of A that are equidistant from m and are the closest to m. This definition can be better explained with the grass-fire example. Imagine that the polygon A is ignited from its boundary points. If the flames burn inwards at uniform rate, the interior points where the flames meet and extinguish represent the medial axis of the object.

The algorithm starts by obtaining the set of border pixels (B) and the set of interior pixels (I). For each set it is stored the coordinates of each pixel position. Then for each interior pixel is obtained the distance to each border pixel. The interior pixel is selected as a medial axis pixel if the interior pixel has two boundary pixels equidistant with the minimum distance.

4.7.6.2 Zhang-Suen Algorithm [33]

The Zhang-Suen Algorithm is a fast and simple algorithm to implement. The algorithm takes the local information of the selected pixels and evaluates them assigning them to be foreground or background pixels. The neighborhood of the center pixel is arranged as seen in figure 4.17. The algorithm has 2 main steps.

P9	P2	P3
P8	P1	P4
P7	P6	P5

Figure 4.17: Center pixel neighborhood

Lets define $B(P1)$ as the sum of all foreground pixel in the neighborhood of $P1$ and $A(P1)$ as the number of transactions from black to white in the sequence

$$[P2, P3, P4, P5, P6, P7, P8, P9, P2].$$

In the first step, all foreground pixels (white pixels) are tested and if the following conditions are met they are set to background pixels (black pixels).

1. The pixel is white and has 8 neighbors
2. $2 \leqslant B(P1) \leqslant 6$
3. $A(P1) = 1$
4. At least one of $P2$, $P4$ or $P6$ is white
5. At least one of $P4$, $P6$ or $P8$ is white

In the second step, all foreground pixels are again tested and if he following conditions are met they are set to background pixels.

1. The pixel is white and has 8 neighbors
2. $2 \leqslant B(P1) \leqslant 6$
3. $A(P1) = 1$
4. At least one of $P2$, $P4$ or $P8$ is white
5. At least one of $P2$, $P6$ or $P8$ is white

During the pixel assessments, either in step 1 or step 2, if any pixel is modified then all steps are repeated. The algorithm ends when no changes occur.

4.8 Image Segmentation Methods

Image segmentation is the process of partitioning an image into regions of homogeneous properties, like color or texture. The goal is to extract various features for a more meaningful analyses, interpretation and understanding of images.

4.8.1 Gray histogram technique

Gray histogram technique defines a threshold T for segmentation by analysing the image histogram, finding a valley between two peaks, bi-modal histogram, and selecting that valley intensity for the T value. This can be difficult, since the image can have a multi-modal histogram or be contaminated with noise. In this case, some criteria as to be followed to determine the optimal threshold to use.

4.8.2 Otsu [24]

The Otsu algorithm perceives two classes of pixels in an image, the foreground pixels and background pixels. The optimal threshold T is calculated by iteratively distinguishing the two classes in a way that the intra-class variance is minimum. The intra-class variance is defined as the weighted sum of the variances of the two classes,

$$\sigma_\omega^2(T) = q_1(T) \times \sigma_1^2(T) + q_2(T) \times \sigma_2^2(T). \tag{4.25}$$

The algorithm starts by calculating the image histogram and the PDF for each intensity level. Assuming that the range of intensities is $[0, L]$, the initial value for the threshold is $T = 1$. The CDF for the pixel intensities $[0, T]$, and $[T + 1, L]$, are calculated as follows,

$$q_1(T) = \sum_{i=0}^{T} P(r_i) \qquad q_2(T) = \sum_{i=T+1}^{L} P(r_i). \tag{4.26}$$

The next step is to calculate the class mean for each class, which is done by applying the following formula,

$$\mu_1(T) = \frac{\sum_{i=0}^{T} i \times P(r_i)}{q_1(T)} \qquad \mu_2(T) = \frac{\sum_{i=T+1}^{L} i \times P(r_i)}{q_2(T)}. \tag{4.27}$$

Finally the intra-class variance for each class is given by,

$$\sigma_1^2(T) = \frac{\sum_{i=0}^{T} (\mu_1(T) - i)^2 \times P(i)}{q_1(T)} \qquad \sigma_2^2(T) = \frac{\sum_{i=T+1}^{L} (\mu_2(T) - i)^2 \times P(i)}{q_2(T)}. \tag{4.28}$$

4.8.3 K-Means [25]

The classical K-means algorithm is a supervised algorithm to rearrange information in a point cloud. The algorithm considers that the point cloud is divided into k clusters. Firstly the clusters are composed of random points and after each iteration of the algorithm they are rearranged to better fit the defined clusters.

Let us consider a cloud of points which is clustered to form k clusters. The algorithm has two main steps. In the first step, the k clusters are initialized and the centroids C_k of each cluster is calculated. The second step iterates through every point and calculates the euclidean distance d between the current point and all the centroids. Based on distance d, the points are rearranged to the clusters with the nearest centroid. The centroids for the new clusters are calculated and a revaluation of the Euclidean distance d of all points to the new centroids is done. The algorithm ends when there are no changes to the clusters.

The K-means algorithm applied to computer vision and image processing has a similar approach. Let us consider an image $f(x,y)$ and a random value T_k between all possible pixel intensities $[0,L]$, where L is the maximum intensity. In the first step, the image histogram is divided at the intensity T_k and two groups of pixel intensities are initialized, and for each group is calculated the centroid C_k. The centroid of a group is equal to the average sum of all pixel intensities of that group. The second step is, calculating a new centroid between the previous calculated centroids, this new value is the current T_k. The algorithm ends when the previous and current T_k are the same.

4.8.4 Wellner [26] [27]

The Wellner algorithm is a simple approach to local adaptive thresholding. The algorithm takes the information from s number of input pixels and determines which output pixels in the binary image should be foreground and background.

The algorithm scans the image row by row while calculating the average sum of the last s pixels seen. When a value of a pixel is t percentage lower than the average sum, then it is set to black, otherwise it is set to white.

Let us consider an image $f(x,y)$, where p_n is a pixel at point n and $A_s(n)$, the average sum of the last s pixel values, is defined as,

$$A_s(n) = \frac{\sum_{i=0}^{s-1} p_{n-i}}{s}. \tag{4.29}$$

The pixel at position n in the output image, is either black if p_n is t percentage lower than $A_s(n)$, otherwise is set to white.

This algorithm provides best results when $s = \frac{1}{8}$ of image width and $t = 15\%$

4.8.5 Bradley and Roth [28]

The Bradley and Roth algorithm is an improvement to Wellner's algorithm, as it is more robust to illumination changes within the image.

The algorithm has the same concept, but instead of taking the average sum of the last s pixels seen, it takes the average sum of the pixels under a window, centered on the pixel to be processed. If the center pixel value is t percentage lower than the average sum, it is set to black, otherwise it is set to white.

The algorithm yields better results when the window has size $15x15$ and $t = 10\%$

4.8.6 Connected Labeling Components

The connected labeling components algorithm identifies multiple objects in a binary image by taking advantage on the local information of each binary pixel. It is assigned a sequential numerical label to all foreground pixels. The objective of the algorithm is to assign a distinct label to each object.

A connected component is an object that is separate and independent. An object independency comes with connectivity, and it can be 4-connected or 8-connected. In 4-connected, any element, belonging to the connected component, with coordinates (x, y) has at least one neighbor element, that belongs to the connected component, with coordinates,

$$\{(x, y+1), (x, y-1), (x+1, y), (x-1, y)\}.$$

In 8-connected, the element has at least one neighbor with coordinates,

$$\{(x-1, y-1), (x-1, y), (x-1, y+1), (x, y-1), (x, y+1), (x+1, y-1), (x+1, y), (x+1, y+1)\}.$$

There are two approaches to the algorithm: the Classical and the Iterative. The Classical approach only needs to scan the image twice, it has low computational time, but requires additional logic and memory space, which brings high implementation complexity. The Iterative approach, is simpler but requires multiple scans through the image.

4.8.6.1 Connected Components: Classical Approach [35]

The algorithm begins with scanning the image and assigning a numerical label to each foreground pixel in the binary image, starting from the top left corner to the bottom right corner. The lowest neighbor label is assigned to each pixel. If that is not possible, a new label is generated. When searching for the lowest neighbor label, a conflict between two labels, for example X and Y can occur. In that case, an entry in an equivalence table is made between X and Y. In the second scan, starting from the top left corner to the bottom right corner, all equivalence labels are merged to create an unique label, if $X < Y$ then all pixels with label Y are replaced with the label X.

Let us consider a binary image $f(x, y)$, represented in Figure 4.18. After the first scan, all foreground pixels have been assigned with a label and the equivalence table is populated with the conflicts encountered, as seen in Figure 4.19 and Table 4.20, respectively. In the second scan, all equivalence table entries converge to the lowest label and the respective replacements are done, as seen in Figure 4.21.

Another complexity layer can be added when a single connected component has multiple conflicts. In that case, the equivalence table has to be sorted in a manner that all conflicts merge to the lowest label.

4.8.6.2 Connected Components: Iterative Approach [36]

The algorithm begins by assigning a numerical label to each foreground pixel in the binary image, starting from the top left corner to the bottom right corner. To each pixel

1	1	1	0	0	1	1
0	1	1	1	0	0	1
1	1	0	0	1	1	1
1	1	1	0	0	0	0
0	0	0	1	1	1	1
0	1	1	1	0	1	1
0	0	0	1	1	1	1

Figure 4.18: Binary image.

1	1	1	0	0	2	2
0	1	1	1	0	0	2
3	1	0	0	4	4	2
3	1	1	0	0	0	0
0	0	0	5	5	5	5
0	6	6	5	0	5	5
0	0	0	5	5	5	5

Figure 4.19: First image scan.

3	1
4	2
6	5

Figure 4.20: Equivalence table.

1	1	1	0	0	2	2
0	1	1	1	0	0	2
1	1	0	0	2	2	2
1	1	1	0	0	0	0
0	0	0	5	5	5	5
0	5	5	5	0	5	5
0	0	0	5	5	5	5

Figure 4.21: Second image scan.

is assigned the lowest neighbor label, if that is not possible a new label is generated and the assignment continues until the first scan ends. The second scan, starting from the bottom right corner to the top left corner, continues to propagate the lowest label of each connected component. The algorithm continues until there are no changes to the labels.

4.8.7 Region Growing [38]

The region growing algorithm generates groups of pixels into regions based on some predefined similarity criteria, for example, gray level intensity or pixel color.

The algorithm takes a group of seed pixels and, based on the selected similarity criteria, neighbor pixels are appended to the corresponding seed pixel. When no more pixels met the criteria the algorithm ends.

The algorithm often results in undergrowing or overgrowing as an outcome of non-optimal parameters setting.

4.9 Feature Extraction Operations

Features have an important role in distinguishing and grouping objects. Knowing which feature is more relevant for classification depends on the type of object to be analysed.

In this work is possible to obtain the most basic geometric properties of objects, such as area and perimeter, and projection profiles (topological properties).

4.9.1 Geometrical Properties [39]

An object can be defined by two components: region (interior pixels) and boundary (perimeter).

The object boundary is related to shape. A pixel is considered to be a boundary pixel when it is part of the region and has at least one neighbor pixel not belonging to the region. The perimeter of an object, denoted as P, can be calculated by the number of boundary pixels and their neighboring relationships. These relationships are determined by the chosen pixel connectivity. The boundary pixels of an object can be found with the Chain code algorithm. The Chain code algorithm translates the connected set of pixels in the border to a set of connections between the pixels. Starting from one known border pixel the algorithm finds the direction in which the next border pixel can be found. The algorithm continues this process and forms a chain code by concatenating the number that designates the direction of the next border pixel. Depending on the connectivity rule the number of possible directions is different, as seen in Figures 4.22 and 4.23. The algorithm ends when the starting pixel is reached. The perimeter is calculated by summing 1 for all vertical and horizontal directions in the chain code and $\sqrt{2}$ for all diagonal directions in the chain code.

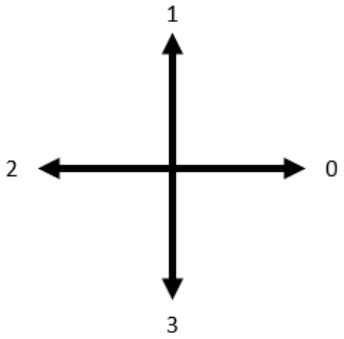


Figure 4.22: 4-connectivity direction system.

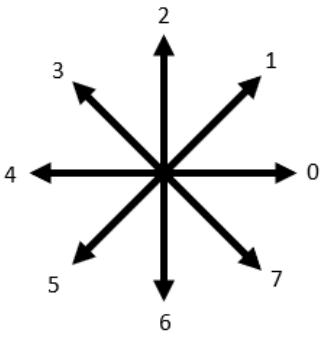


Figure 4.23: 8-connectivity direction system.

The object region is related to scalar measurements based on the objects geometric properties. The simplest geometric feature is area, denoted as A, which represents the number of pixels that compose the object and can be defined as,

$$A = \sum_{(x,y)\in R} 1 \tag{4.30}$$

where $(x, y) \in R$ represents all pixels that belong to the object R. The centroid of an object, denoted as C, represents the gravitational center for that object. The coordinates (C_x, C_y) of the centroid are calculated by the ratio of the sum of all x object pixel coordinates and y object pixel coordinates with the object area,

$$C_x = \frac{\sum_i x_i}{A} \tag{4.31}$$

$$C_y = \frac{\sum_i y_i}{A} \tag{4.32}$$

where x_i and y_i represent the all the coordinates values for the object pixels. Compactness of an object, denoted as Cp, quantifies how close the object is to a circle. This measure is calculated by the ratio of area to perimeter, and can be defined as,

$$Cp = \frac{4\pi \times A}{P^2}. \tag{4.33}$$

A normalized version of compactness, ranging from 0 to 1, can be achieved with the following formula,

$$Cp^{'} = 1 - \frac{4\pi}{Cp}. \tag{4.34}$$

Circularity, denoted as Cc, measures how regular is the object boundary, with a maximum of 1 for a convex shape. This feature can be defined as,

$$Cc = 4\pi \times \frac{A}{Pc^2} \tag{4.35}$$

where Pc represents the object convex perimeter. Aspect coefficient, denoted as Ac, measures the overall proportion of the object. This measure is calculated by the ratio between the maximum and minimum diameters of the object, and can be defined as,

$$Ac = \frac{D_{max}}{D_{min}} \tag{4.36}$$

where D_{max} and D_{min} represent the maximum and minimum diameters of the object, respectively. Convexity, denoted as Cv, represents the overall convexity ratio of the object. This measure is calculated by the ratio of the object perimeter to object convex perimeter, and can be defined as,

$$Cv = \frac{Pc}{P}. \tag{4.37}$$

The main object axis orientation represents the angle, denoted θ, in which the object is related to the image coordinate axis. The angle can be calculated as follows,

$$S_x = \sum_i x_i \quad S_y = \sum_i y_i \quad S_{xx} = \sum_i x_i^2 \quad S_{yy} = \sum_i y_i^2 \quad S_{xy} = \sum_i x_i \times y_i \tag{4.38}$$

$$M_{xx} = S_{xx} - \frac{S_x^2}{A} \qquad M_{yy} = S_{yy} - \frac{S_y^2}{A} \qquad M_{xy} = S_{xy} - \frac{S_x \times S_y}{A} \tag{4.39}$$

$$\theta = \arctan\left\{ \frac{M_{xx} - M_{yy} + \sqrt{(M_{xx} - M_{yy})^2 + 4 \times M_{xy}^2}}{2 \times M_{xy}} \right\}. \tag{4.40}$$

4.9.2 Topological Properties [37]

The structural information of an object contains important features, such as shape, dimension and connectedness. Projection profiles represent the sum of foreground pixels along the x-axis and y-axis. This representation allows for the analysis and comparison of an object composition. Figures 4.25 and 4.26 shows the projection profile of the object shown in Figure 4.24.

0	0	1	1	1	0	0
1	1	1	1	1	1	1
1	1	1	1	1	1	1
0	1	0	0	1	1	0
0	1	0	0	1	1	0
1	1	1	1	1	1	1
1	1	1	1	0	0	0

Figure 4.24: Binary image.

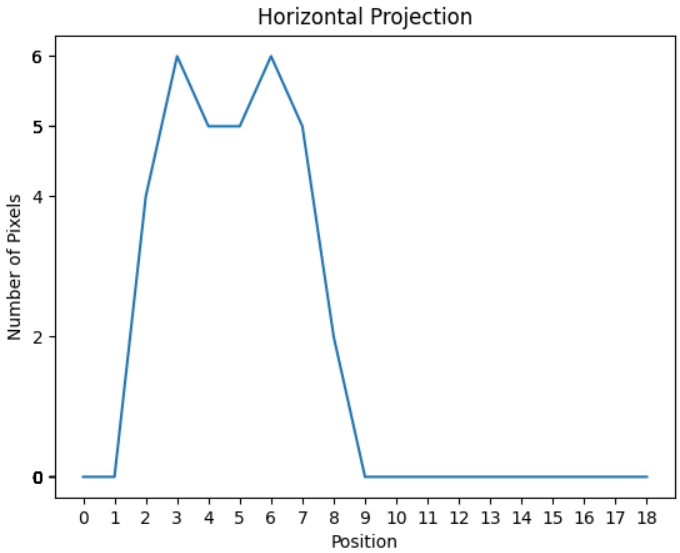


Figure 4.25: Horizontal projection (y-axis).

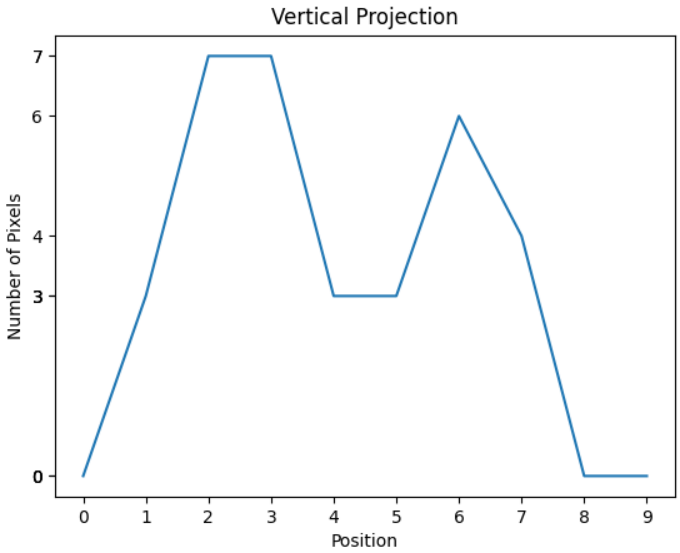


Figure 4.26: Vertical projection (x-axis).

5

Results and Discussion

This chapter presents the results and discussion of the implemented algorithms. The results of each algorithm are shown in tables with execution times, expressed in seconds, and their output images. When appropriate comparisons with approaches and algorithms of other authors is made. For the execution time tables, every algorithm was repeated 10 times for each image and the results shown are the average values over the repetitions.

5.1 Preparation

The computer used to run the methods and algorithms have the following specifications:

1. Processor: AMD Ryzen 5 3600

2. Operating System: Windows 10 64-bit

3. Graphics Card: NVIDIA GeForce GTX 1650 4 GB GDDR6

4. Random Access Memory (RAM): 16 GB

Python is the programming language used for the implementation of this work, since it is an open-source language, and has many tools and support documentation regarding image processing and computer vision easing the implementation process.

Figures 5.1 (size $512x512$) and 5.3 (size $790x790$) are some of the standard images in the image processing and computer vision fields. They were used to obtain the output images for the implemented algorithms so that comparisons can be made. The execution time tables were made using Figure 5.1 as the input image. Salt and pepper noise was added to both figures as shown in Figures 5.2 and 5.4, since it is the simplest to implement as a function and clearly demonstrates the effects each filter has on the images. The image used for the image segmentation algorithms is shown in Figure 5.5. It is a binary image with four different white objects in a black background.

Figure 5.1: Lena image.

Figure 5.2: Noisy Lena image.

Figure 5.3: Camera man image.

Figure 5.4: Noisy Camera man image.

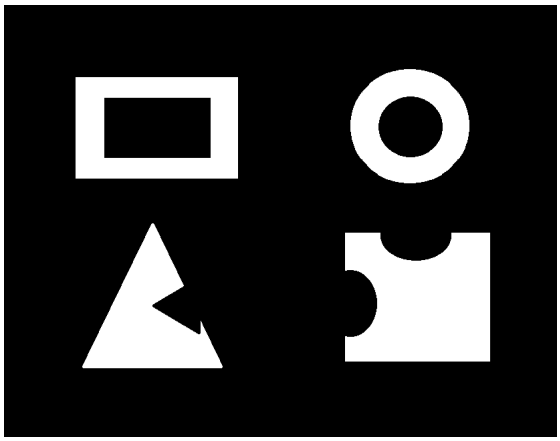


Figure 5.5: Binary image.

5.2 Geometric Transformation Algorithms

This section shows the results from the three geometric transformation algorithms implemented: Translation, Rotation and Resizing. Table 5.1 shows the execution times for each algorithm, and Figures 5.7a, 5.7c, 5.8 and 5.9 the output images. For the Translation operation, no interpolation method was used since all output pixels possess valid coordinates.

On the other hand, in the Rotation and Resizing operations nearest-neighbor, bi-linear and bi-cubic interpolation methods were used. From Table 5.1 it is seen the difference of execution times when using different interpolation methods. The nearest-neighbor interpolation has lower execution times compared with the bi-linear and bi-cubic interpolation, but, a close up in Figures 5.7 and 5.8 show that, the bi-linear and bi-cubic interpolations preserve better image details. The images from the bi-linear and bi-cubic interpolations, although seem similar at the naked human eye, when processed by a computer changes can be found, and depending on the goal of the algorithms, these differences can make an impact on the final result. The images resized with a factor of 0.5 are identical since the original spatial resolution is greater than the final spatial resolution.

Transformation	NN	BL	BC	
Resize ($s = 2.5$)	0.252	2.871	25.827	-
Resize ($s = 0.5$)	0.133	0.771	6.385	-
Rotation ($d = 45$)	0.354	1.860	22.221	-
Translation ($\Delta x = 100$, $\Delta y = 100$)	-	-	-	0.129

Table 5.1: Execution times for the geometric algorithms (s represents the scaling factor ; d represents the rotation degrees; Δx and Δy represent the offset in the x-axis and y-axis, respectively; NN stands for Nearest-Neighbor interpolation;BL stands for Bi-linear interpolation; BC stands for Bi-cubic interpolation).

These results can be improved by using different interpolation variations or methods. The authors in [3] demonstrate an adaptive bi-linear and bi-cubic interpolation algorithms, where local features of the image are considered and affect the interpolation result diminishing the blurring effect.

5.3 Linear Filter Algorithms

This section shows the results from the classical Linear Mean filter implemented with three different variations: Algorithm A, Algorithm B and Algorithm C. Table 5.2 shows the execution times of each variation. Algorithm A and C have approximated the same execution times when applied a $3x3$ sliding window, being Algorithm B the slowest. On the other hand, when applied a $9x9$ sliding window the difference between the variations is more significant. Algorithm A is the slowest of them all and Algorithm C, even with the increase of the sliding window, has an approximately constant execution time. Theoretically the execution time for Algorithm C should be the same for all window sizes, but the computer processing the image, at the time of executing the algorithm, might run other system routines that can cause some fluctuations on the execution time samples. Figure 5.10 show the output images the classical Linear Mean filter. These figures show that this filter does not handle well impulsive noise, since the images are of poor quality.

(a) Result of nearest-neighbor interpolation.

(b) Result of bi-linear interpolation.

(c) Result of bi-cubic interpolation.

Figure 5.6: Resized images with scaling factor of 0.5.

Algorithm	3x3	9x9
A	1.722	8.398
B	2.467	3.525
C	1.572	1.557

Table 5.2: Execution times for the Linear filter algorithms.

5.4 Nonlinear Filter Algorithms

This section shows the results from the implemented classical Nonlinear filters: Nonlinear Mean filter, Nonuniform Mean filter, Median filter, Hybrid-Median filter, K-Nearest Mean filter, Sigma filter and Mode filter. Table 5.3 shows the execution times for each algorithm and Figures 5.11 and 5.12 the result images.

The parameters used for the $3x3$ algorithms are as follows: the Nonuniform Mean Filter coefficients are [1,2,1,2,4,2,1,2,1], the K-Nearest Mean filter number of neighbor pixels is 6, the Sigma filter standard variation is 2 and the Nonlinear Mean filter mean threshold value is 10.

(a) Result of nearest-neighbor interpolation.

(b) Result of bi-linear interpolation.

(c) Result of bi-cubic interpolation.

Figure 5.7: Resized images with scaling factor of 2.5.

The Median filter was executed using three different sorting algorithms: slow BubbleSort (SB), fast BubbleSort (FB) and MinMax algorithm (MM). For a small window size the MinMax algorithm is the fastest, but as the window size increases the slow BubbleSort and the fast BubbleSort algorithm show a better performance than the MinMax algorithm which did not correspond to the expectations. This issue requires further investigation.

The result images show that the Mean filters, the Sigma filter and the Mode filter have poor performance regarding impulsive noise. On the other hand, the Median filters quite effectively reduce the introduced impulsive noise while still preserving image details. Among the Median filters the Hybrid-Median filter has the best image quality.

(a) Result of nearest-neighbor interpolation.

(b) Result of bi-linear interpolation.

(c) Result of bi-cubic interpolation.

Figure 5.8: Rotated images with a 30 degree angle.

Filter	3x3	5x5	9x9
Nonlinear Mean Filter	2.438	-	9.060
Nonuniform Mean Filter	8.260	-	-
Median Filter (SB)	2.798	-	43.220
Median Filter (FB)	2.315	-	31.447
Median Filter (MM)	1.85	-	46.272
Hybrid-Median (SB) Filter	-	7.114	-
Hybrid-Median (FB) Filter	-	6.313	-
Hybrid-Median (MM) Filter	-	4.992	-
K-Nearest Mean Filter	6.712	-	-
Sigma Filter	21.715	-	-
Mode Filter	10.745	-	-

Table 5.3: Execution times for the Nonlinear filter algorithms (SB stands for slow BubbleSort; FB stands for fast BubbleSort; MM stands for MinMax algorithm).

Figure 5.9: Translated image.

(a) 3x3 sliding window.

(b) 9x9 sliding window.

(c) 3x3 sliding window.

(d) 9x9 sliding window.

Figure 5.10: Result images for the Linear Mean filter.

5.5 Edge Detection Algorithms

This section shows the results from the classical edge detection algorithms: Differentiation, Roberts, Prewitt and Sobel. Table 5.4 shows the execution times of each algorithm and Figures 5.13 and 5.14 the output images. As expected higher algorithm complexity leads to a better image quality, but also higher execution times. Amongst the 4 operators

(a) Median filter with 3x3 sliding window. (b) Median filter with 9x9 sliding window. (c) Hybrid-Median filter with 5x5 sliding window.

(d) Mean Nonlinear filter with 3x3 sliding window. (e) Nonlinear Mean filter with 9x9 sliding window. (f) K-Nearest Mean filter with 3x3 sliding window.

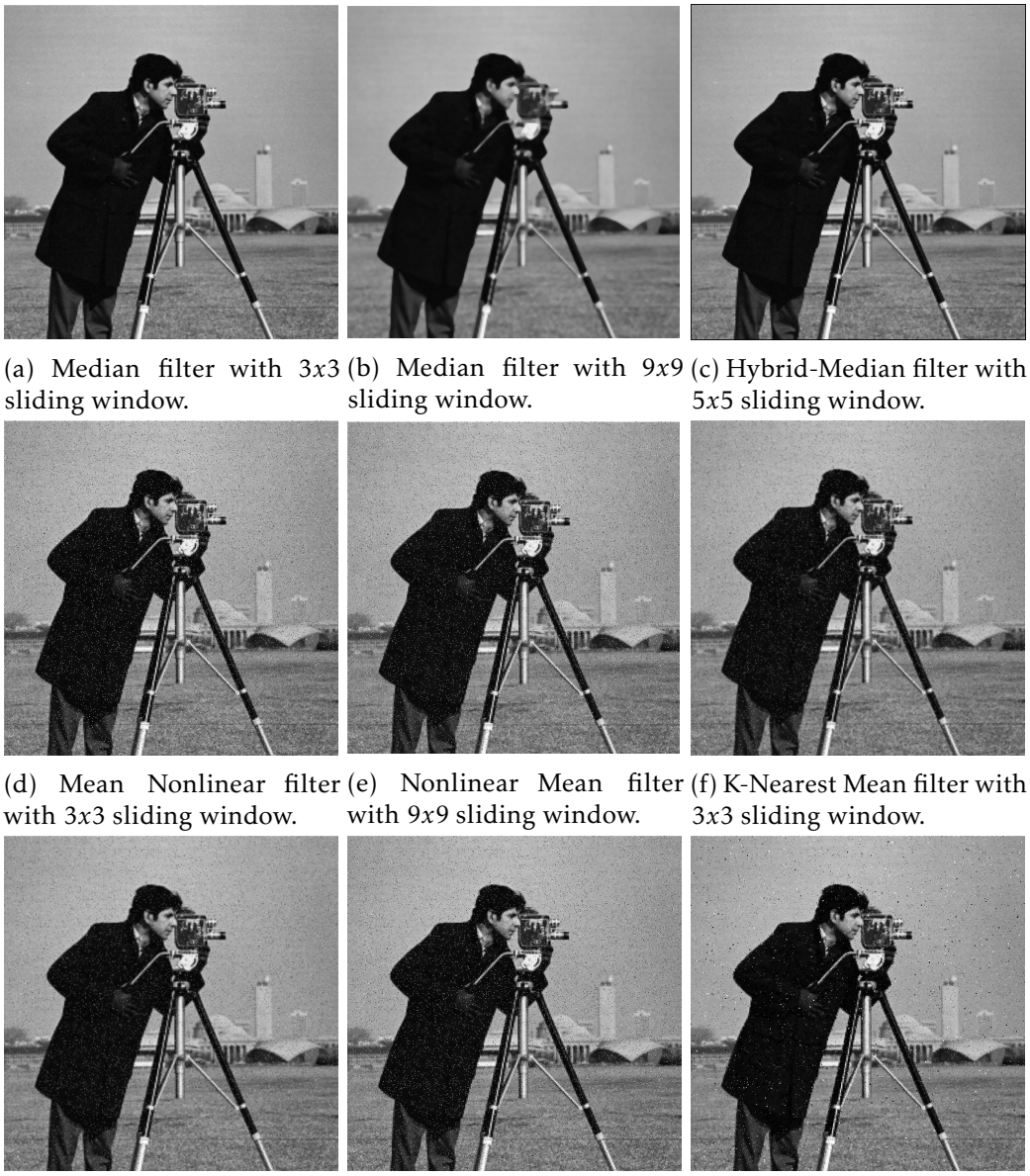
(g) Nonuniform Mean filter with 3x3 sliding window. (h) Sigma filter with 3x3 sliding window. (i) Mode filter with 3x3 sliding window.

Figure 5.11: Result images for the Nonlinear filters with Lena image.

the ratio of image quality and execution time is more noteworthy from the Sobel operator.

Operator	
Differentiation	0.633
Roberts	0.722
Prewitt	2.166
Sobel	5.616

Table 5.4: Execution times for the edge detection algorithms.



(a) Median filter with 3x3 sliding window.
(b) Median filter with 9x9 sliding window.
(c) Hybrid-Median filter with 5x5 sliding window.
(d) Mean Nonlinear filter with 3x3 sliding window.
(e) Nonlinear Mean filter with 9x9 sliding window.
(f) K-Nearest Mean filter with 3x3 sliding window.
(g) Nonuniform Mean filter with 3x3 sliding window.
(h) Sigma filter with 3x3 sliding window.
(i) Mode filter with 3x3 sliding window.

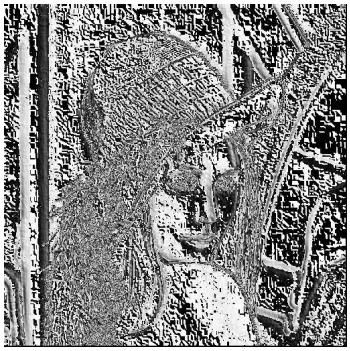
Figure 5.12: Result images for the Nonlinear filters with Camera man image.

5.6 Histogram Equalization Algorithm

This section shows the results from the classical Histogram Equalization method. The execution time is not shown since there are no other implemented algorithms to compare with. Figure 5.15 shows the result images. The image shows that the pixel intensities got more concentrated in certain regions giving more emphases to those areas. This comes from the fact that the pixels got a more even distribution across the image.

(a) Differentiation operator.

(b) Roberts operator.



(c) Prewitt operator.

(d) Sobel operator.

Figure 5.13: Result images for the edge detection operators with the Lena image.

5.7 Morphological Transformation Algorithms

This section shows the results for the implemented morphological and skeletonization algorithms: Dilation, Erosion, Hit-and-Miss (morphological algorithms), Medial Axis Transform and Zhang-Suen (skeletonization algorithms).

5.7.1 Morphological Operators

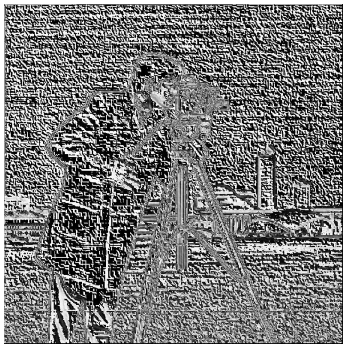
Figures 5.16, 5.17 and 5.18 the output images for the Dilation, Erosion and Hit-and-Miss operators, respectively. The results are as expected, multiple passages of the Dilation operator fills up the gaps in the objects and the Erosion operator degrades the objects. The Hit-and-Miss operator manages to identify the desired patterns in the image, being in this case the patter in Figure 5.18a.

5.7.2 Skeletonization

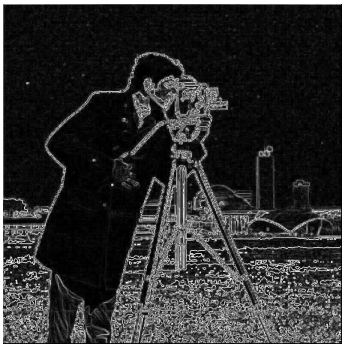
The implemented skeletonization algorithms managed to effectively obtain the core skeletons of the objects present in Figure 5.5, as shown in Figure 5.19. The Medial Axis Transform Algorithm, with some parameters adjustments, is able to obtain a good representation of the objects skeleton but the Zhang-Suen Algorithm, at much lower computation time, obtains much cleaner results.

(a) Differentiation operator.

(b) Roberts operator.



(c) Prewitt operator.



(d) Sobel operator.

Figure 5.14: Result images for the edge detection operators with the Camera man image.

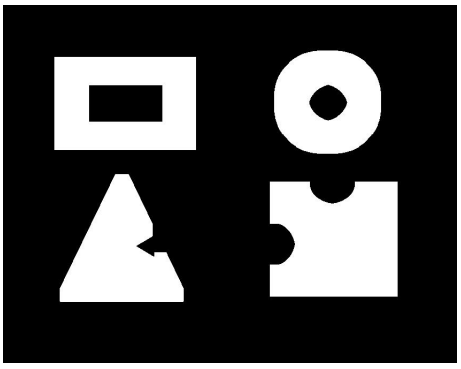
(a) Equalized Lena image.

(b) Equalized Camera man image.

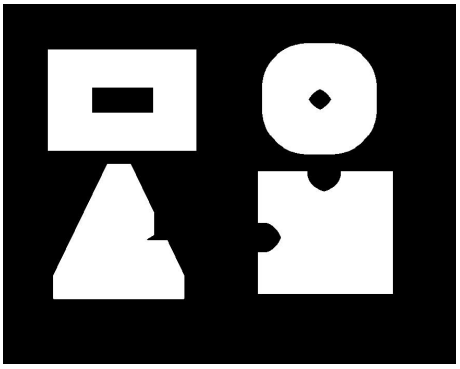
Figure 5.15: Images after Histogram Equalization.

Operator	
Blum	366.348
Zhang-Suen	3.907

Table 5.5: Execution times for the skeletonization algorithms.

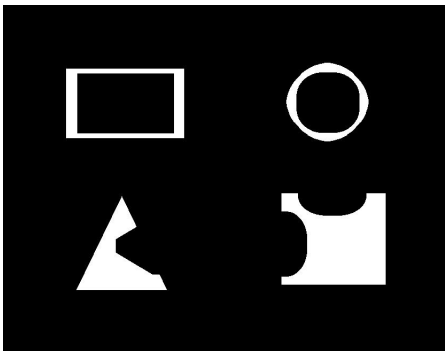


(a) Image after 10 passages.



(b) Image after 20 passages.

Figure 5.16: Result image for the Dilation operator.



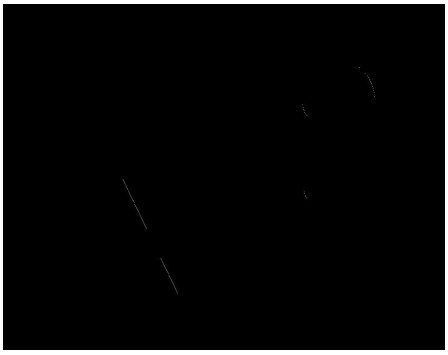
(a) Image after 10 iterations.

(b) Image after 20 iterations.

Figure 5.17: Result image for the Erosion operator.

1	0	0
0	1	0
0	1	0

(a) Kernel for the operator.



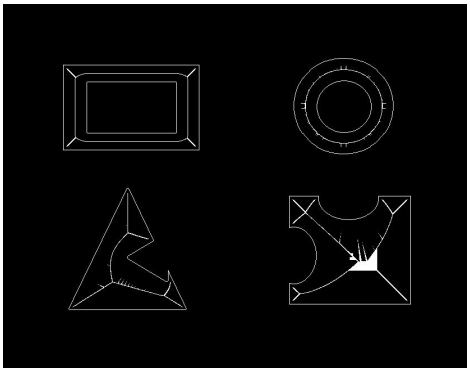
(b) Result image for the Hit-and-Miss operator.

Figure 5.18: Hit-and-Miss operator and kernel.

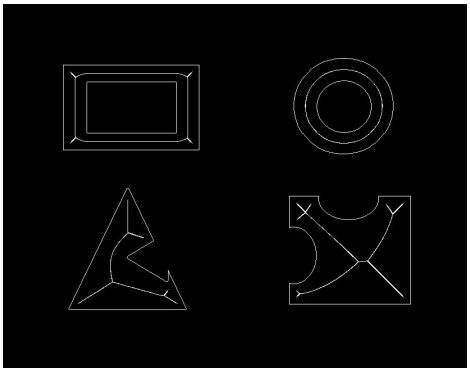
5.8 Image Segmentation Algorithms

This section show the results for the implemented region-based image segmentation methods: Classical algorithm, Iterative algorithm, Otsu, Wellner, C-Means, and Bradley and Roth. Table 5.6 shows the execution times of each algorithm and Figures 5.20 and 5.21 the output images.

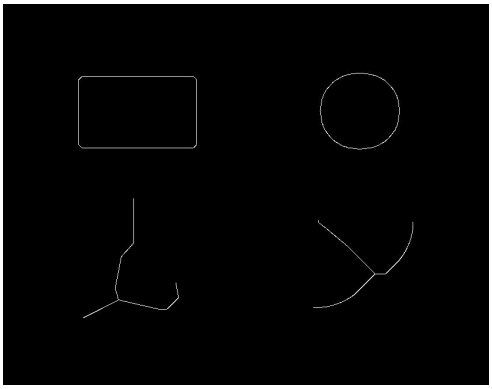
The output images are as expected, the Classical algorithm and Iterative algorithm



(a) Medial Axis Transform with Euclidean distance 10.



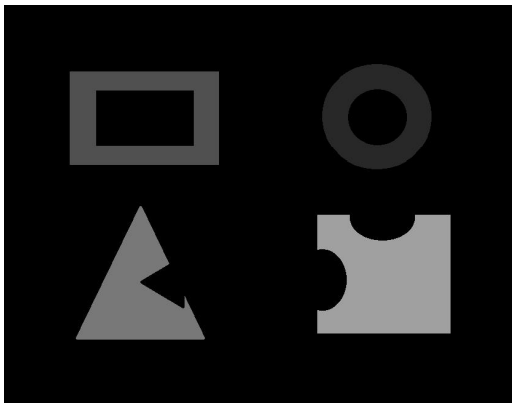
(b) Medial Axis Transform with Euclidean distance 20.



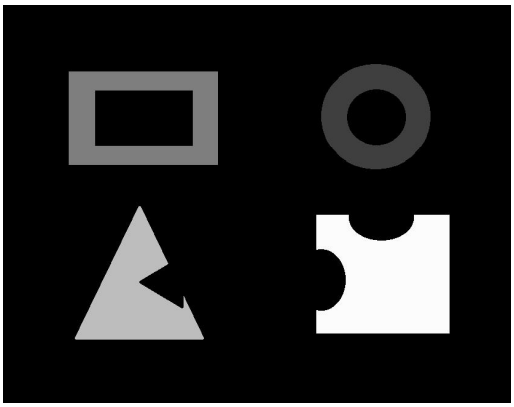
(c) Zhang-Suen algorithm.

Figure 5.19: Result images for the skeletonization algorithms.

managed to identify one object from the original image. The Classical algorithm as a much lower execution time than the Iterative algorithm. The different binarization algorithms managed to emphasise particular details in the image. Otsu, C-Means and Wellner contrast solid like objects, while Bradley and Roth obtain more finer and subtle details.



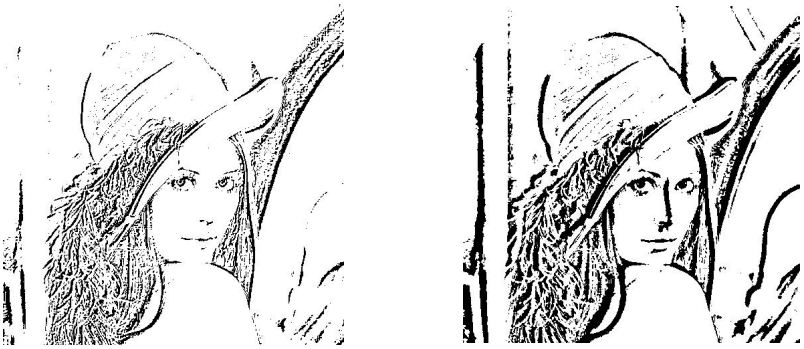
(a) Classic algorithm.



(b) Iterative algorithm.

Figure 5.20: Result images for the connected components algorithms.

(a) Otsu algorithm. (b) Wellner algorithm. (c) C-Means algorithm.



(d) Bradley and Roth algorithm with a $3x3$ sliding window. (e) Bradley and Roth algorithm with a $9x9$ sliding window.

Figure 5.21: Result images for the binarization algorithms.

Algorithm	
Connected Component (Classical)	2.414
Connected Component (Iterative)	3.812
Otsu	1.610
Wellner A	2.933
Wellner B	34.526
C-Means	1.612
Bradley and Roth (3x3)	2.961
Bradley and Roth (9x9)	9.679

Table 5.6: Execution times for the image segmentation algorithms.

5.9 Object Feature Extraction

This section shows the results obtained when extracting geometrical and topological features from Figures 5.22, 5.23 and 5.24. The results from the geometrical extraction can be seen in Tables 5.7, 5.8 and 5.9. The results for the topological extraction can be seen in Figures 5.25, (...), 5.36.

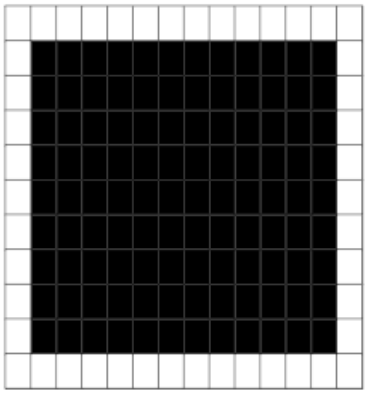


Figure 5.22: Rectangle image.

Area	108
Perimeter	38
Centroid	[5, 10]
Normalized Compactness	0.06
Convex Area	108
Convex Perimeter	38
Convexity	1.00
Circularity	0.94

Table 5.7: Characteristics for the rectangle.

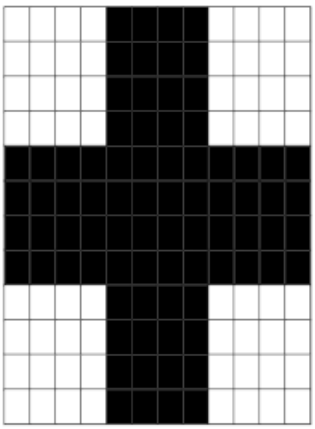


Figure 5.23: Cross image.

Area	68
Perimeter	39.66
Centroid	[6, 10]
Normalized Compactness	0.46
Convex Area	82.5
Convex Perimeter	32.63
Convexity	0.82
Circularity	0.80

Table 5.8: Characteristics for the cross.

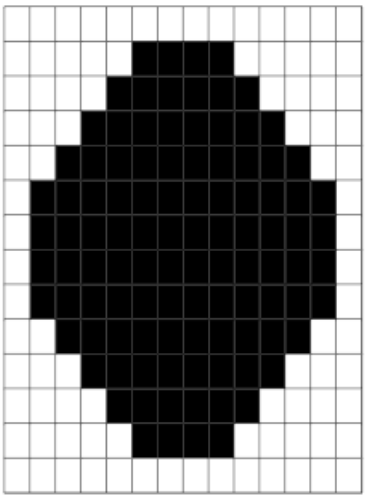


Figure 5.24: Circle image.

Area	86
Perimeter	30.97
Centroid	[6, 9]
Normalized Compactness	0.87
Convex Area	86
Convex Perimeter	30.97
Convexity	0.99
Circularity	1.26

Table 5.9: Characteristics for the circle.

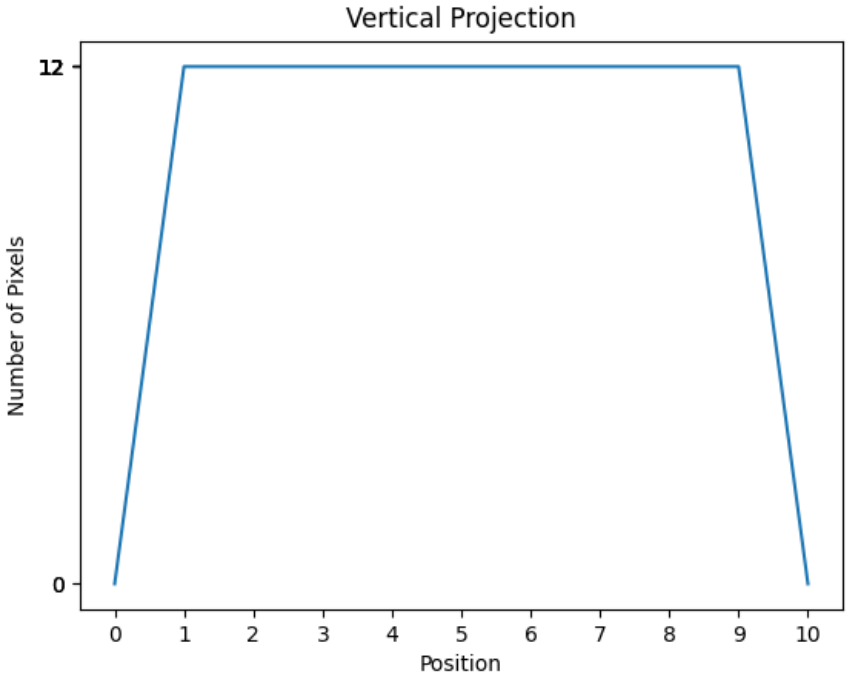


Figure 5.25: Vertical Projection of the Rectangle.

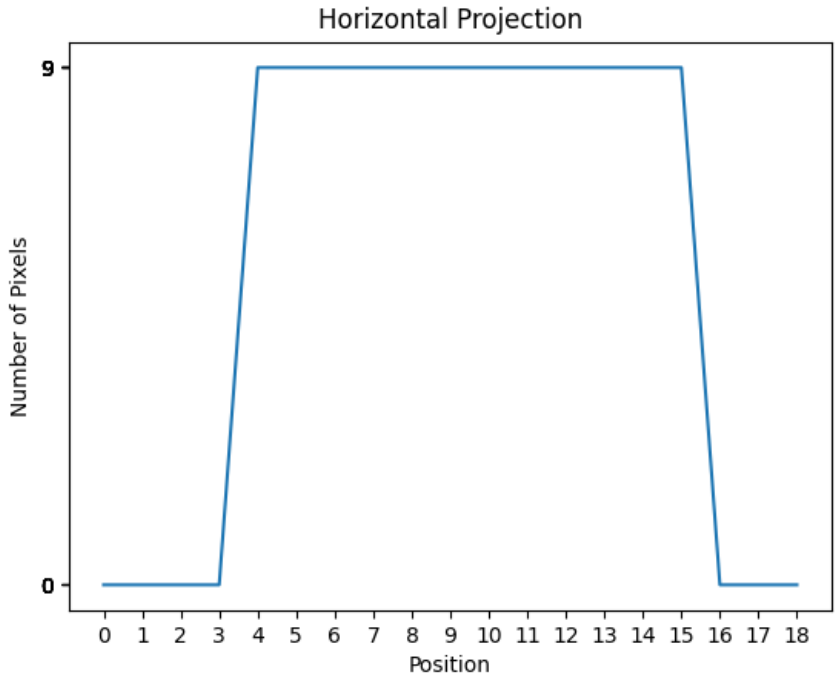


Figure 5.26: Horizontal Projection of the Rectangle.

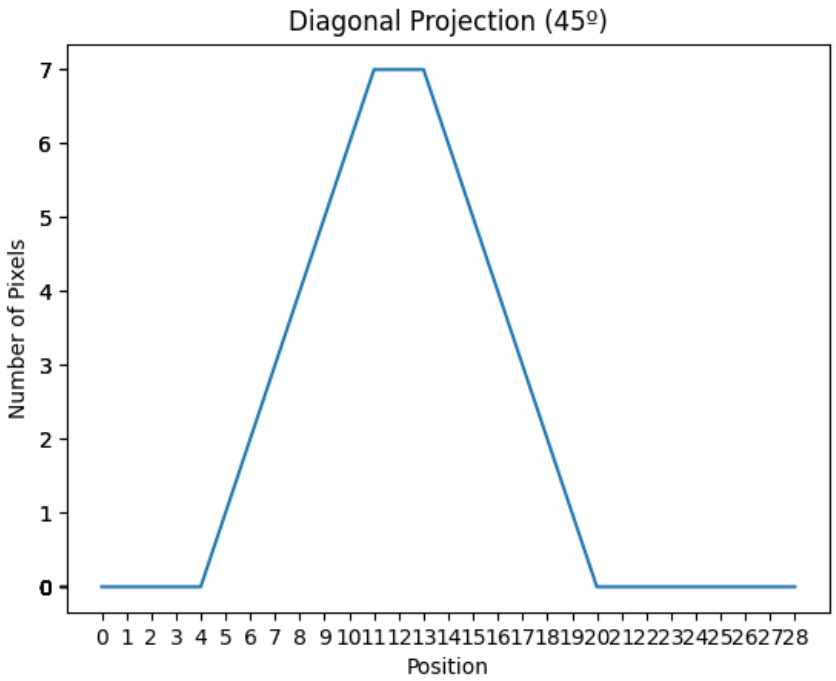


Figure 5.27: Diagonal (45º) Projection of the Rectangle.

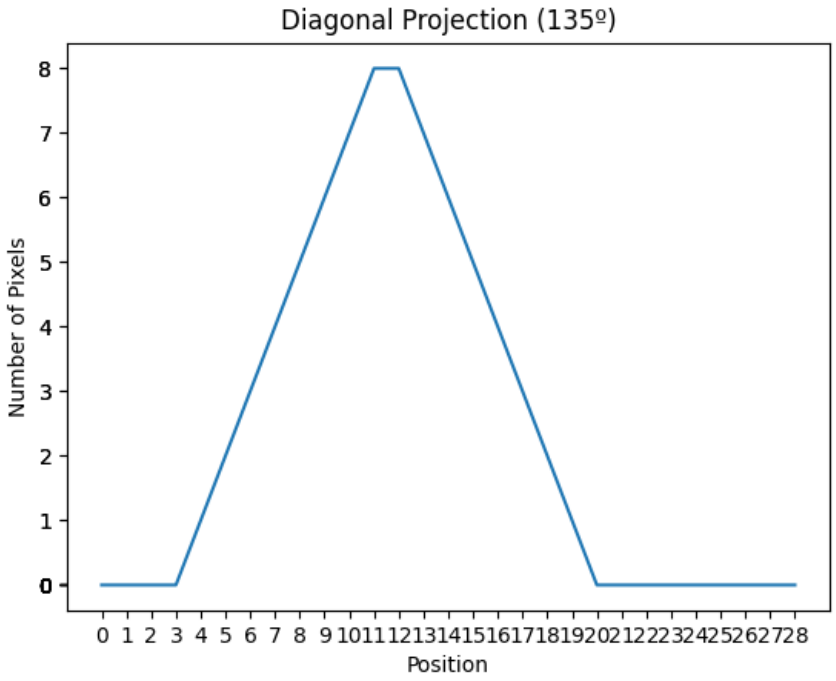


Figure 5.28: Diagonal (135º) Projection of the Rectangle.

5.10 Alternative Methods

In the literature, the author of article [4] shows an Adaptive Mean filter, denoted LMS. The main idea of the filter is to adapt the sliding window weight for each pixel region.

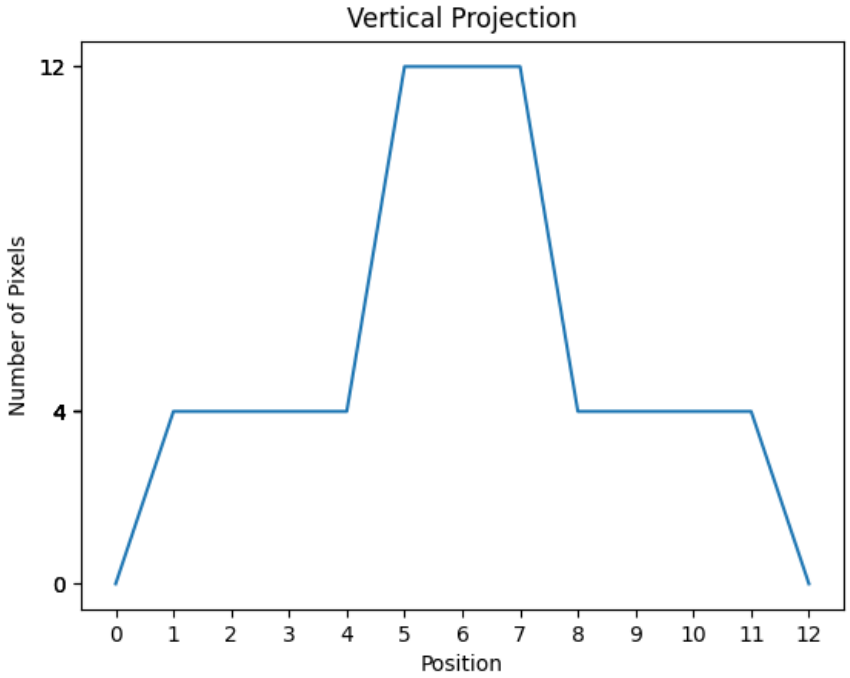


Figure 5.29: Vertical Projection of the Cross.

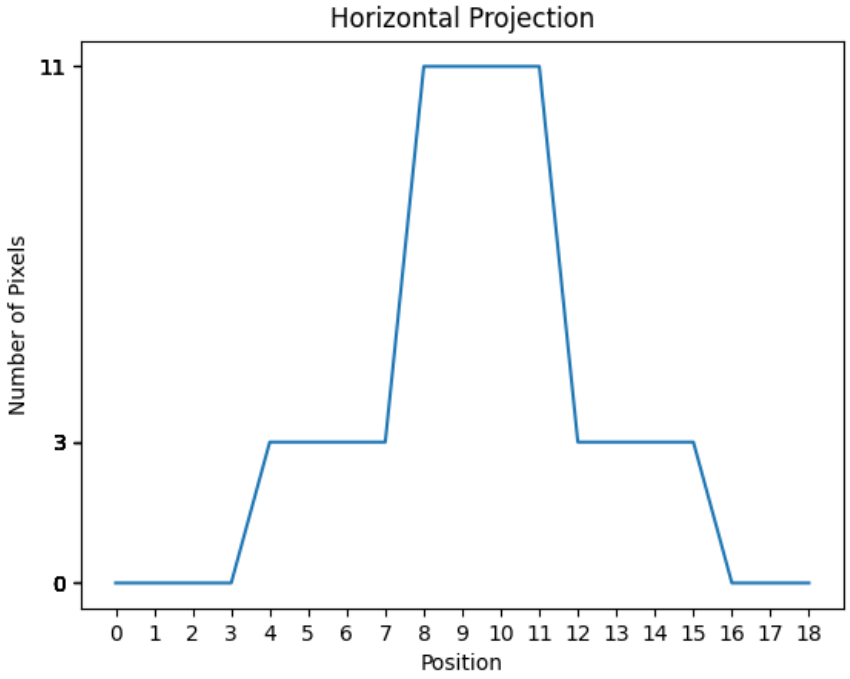


Figure 5.30: Horizontal Projection of the Cross.

This is done by first defining a weight matrix, for example

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

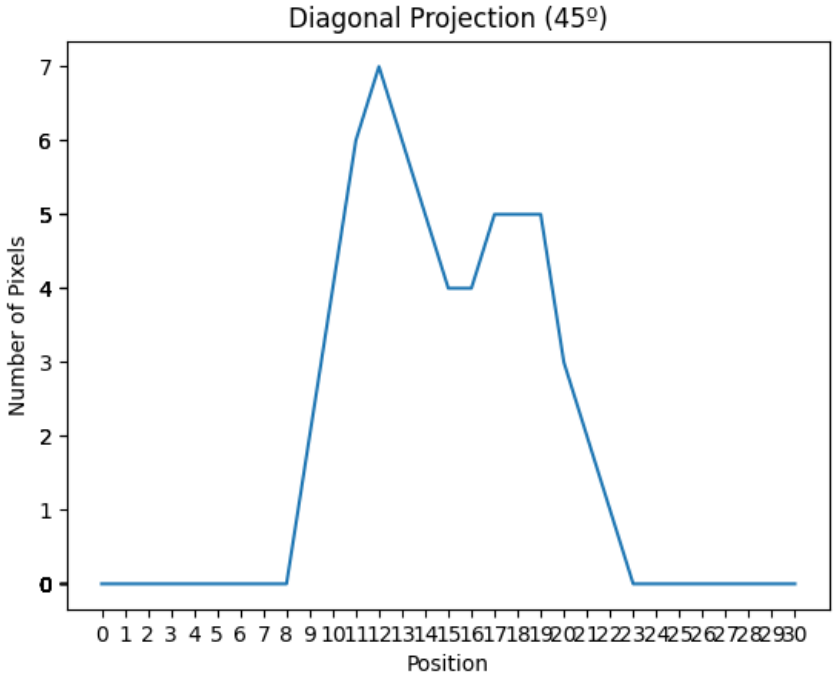


Figure 5.31: Diagonal (45º) Projection of the Cross.

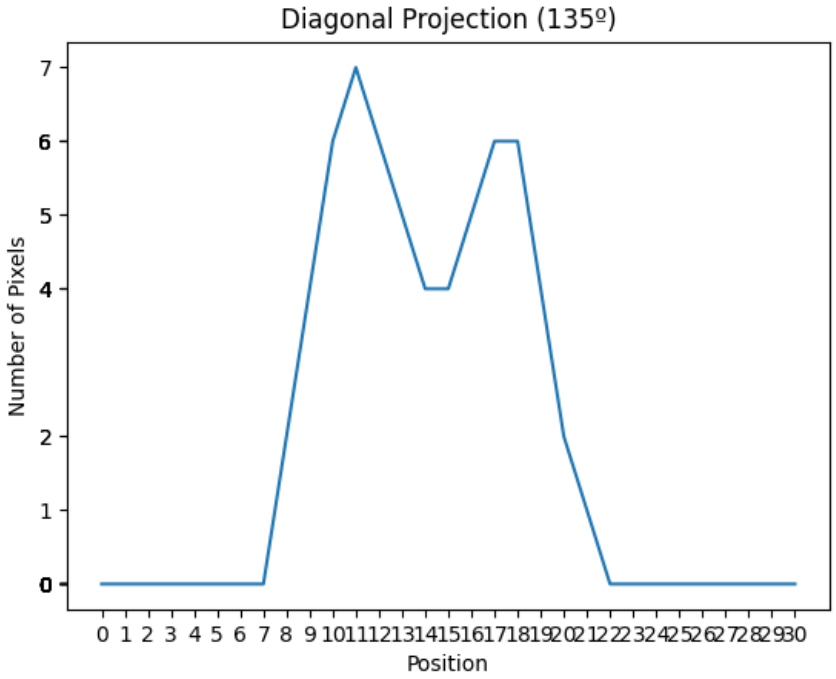


Figure 5.32: Diagonal (135º) Projection of the Cross.

denoted has h_0, then with a sliding window MxN the average sum μ of the pixels is taken. The mean μ is subtracted to the elements of the window creating the residual matrix W^r.

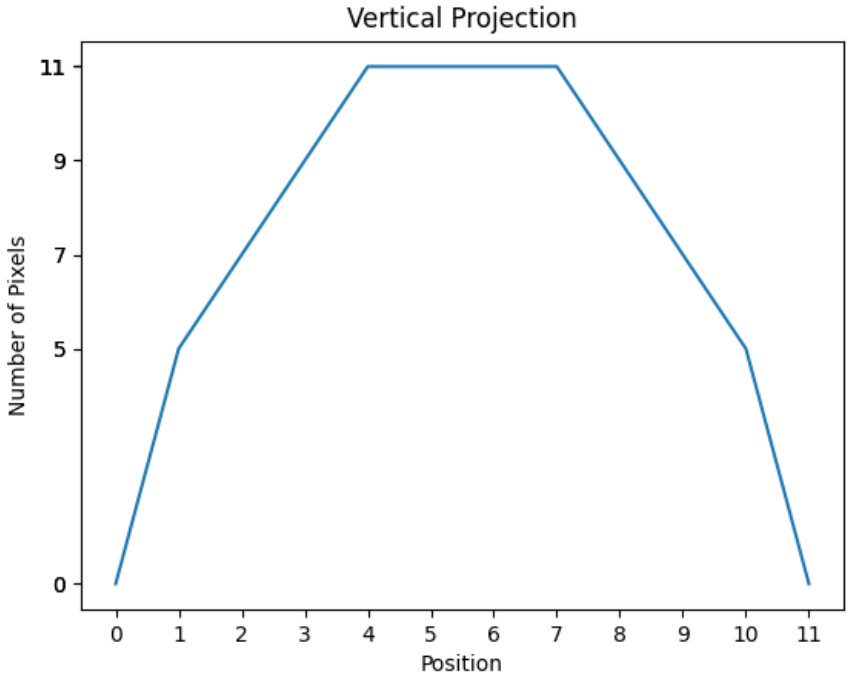


Figure 5.33: Vertical Projection of the Circle.

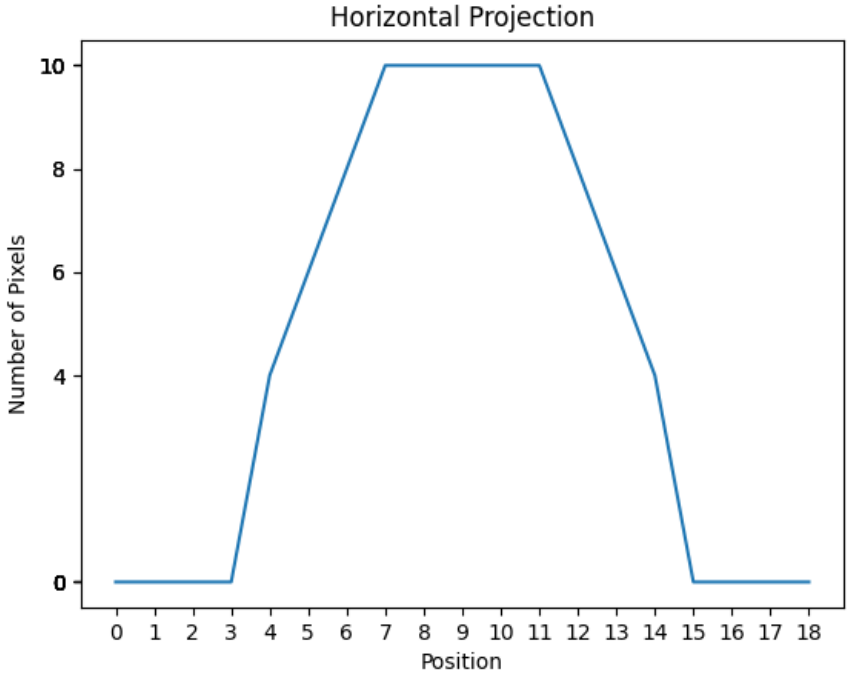


Figure 5.34: Horizontal Projection of the Circle.

A weighted sum $z^{'}$ is defined as,

$$z^{'} = \sum_{(i,j)} h(i,j) \times W^{r} \tag{5.1}$$

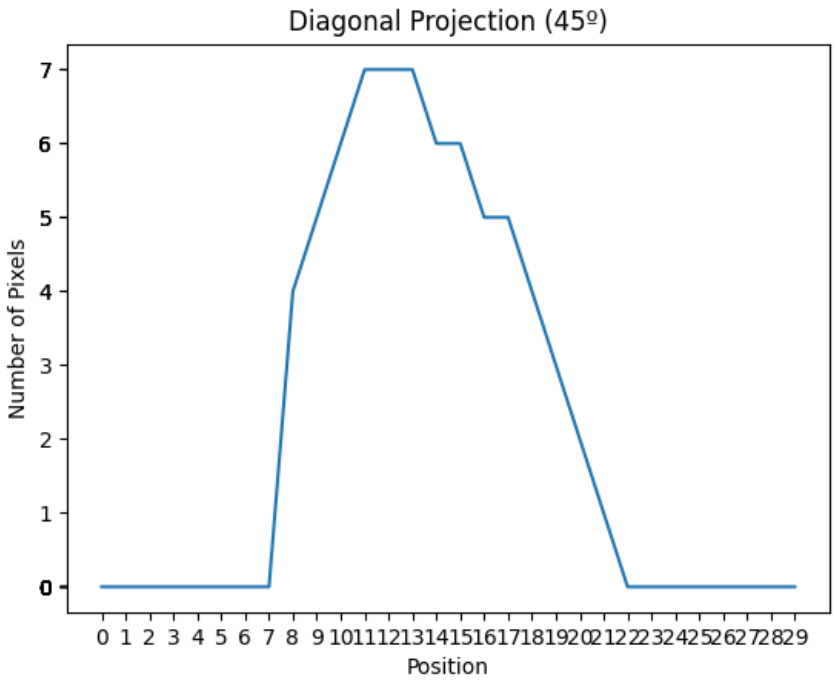


Figure 5.35: Diagonal (45º) Projection of the Circle.

where $h(i,j)$ represents elements of the weight matrix. The sum of $z^{'}$ and μ replaces the value P of the center pixel,

$$P = z^{'} + \mu \, . \tag{5.2}$$

To calculate the following weight matrices the above formula is used,

$$h_{k+1} = h_k + \eta \times e \times W^r \tag{5.3}$$

where h_{k+1} and h_k are the current and previous weight matrices, respectively, e is the deviation computed by,

$$e = z^{'} - W^r \tag{5.4}$$

and η is a value in the range of 0 to $\frac{1}{\lambda}$, where λ is the largest eigenvalue of the sliding window. This filter proves to have a much better performance than the classical Mean Filter, since the sliding window weights adapt accordingly to the image region while the classical version uses the same weights throughout the entire image which can not be optimal.

The authors from the article [5] show an improved variation of the Median filter, where the algorithm makes an educated decision on which pixel is corrupted by noise or noise-free, depending on that judgment the pixel is replaced by the median value of its neighborhood. The decision notation goes as follows,

$$G(u,v) = \begin{cases} f(x,y), & |f(x,y) - V_{med}| < T \\ V_{med}, & |f(x,y) - V_{med}| \geqslant T \end{cases} \tag{5.5}$$

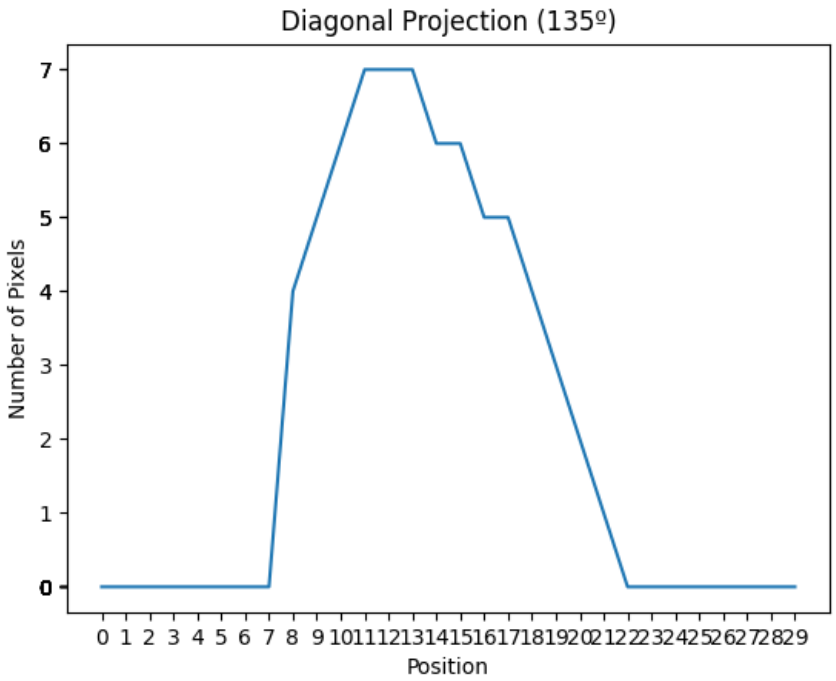


Figure 5.36: Diagonal (135º) Projection of the Circle.

where $G(u,v)$ is the output image, $f(x,y)$ is the input image pixel being processed, V_{med} is the median value of the neighborhood of $f(x,y)$ and T is the threshold value for the decision. The higher the value for T the more susceptible the filter is to noise. A potential advantage for this Median filter variation is that when the image is not corrupted by noise, the filter will not greatly affect the image.

The Canny's edge detection algorithm is a well known and preferred edge detector in most cases. The authors of article [6] review the principles of the algorithm. The algorithm contains a number of adjustable parameters that allows optimization of both efficiency and effectiveness of the algorithm. The steps for the algorithm are as follow,

1. Smooth the image with a Gaussian filter,
2. Compute the gradient and direction of the image pixels,
3. Apply non-maximum suppression,
4. Use double thresholding to determine potential edges,
5. Suppress all edges that are not connected to a strong edge.

This edge detector yields better image results and it is more robust to image noise but also increases the algorithm complexity and computational time.

The authors in article [7] show various variations of the classic Histogram Equalization. The Dynamic Histogram Equalization is one of the popular and simplest variations.

This algorithm tends to eliminate the influence of higher histogram components on lower histogram components and regulates the distribution of gray levels for objective enhancement of the image by using local minima separation of histogram. The partitioning of the histogram allows the assignment of of specific gray levels according to the local data, which provides better brightness preservation and contrast enhancement.

6

Conclusion and Future Work

This chapter concludes the thesis by presenting an overview of the developed work and its achievements, and future work suggestions to the improvement and continuance of the developed document.

6.1 Conclusions

This work over viewed the necessary fundamentals to understand basic image processing and computer vision algorithms and methodologies. The more basic algorithms were first introduced in order to establish a foundation ground and later more complex algorithms are explained creating a chain of learning and understanding.

The working logic behind each algorithm was shown and explained with some practical examples. The featured library can be accessed on GitHub [1] and serves as a complement to this document, showing how the algorithms were implemented, providing some guidelines on how these algorithms can be used. Although in Chapter 5 in Section 5.9 the convex area calculation returned an incorrect value with no apparent solution.

The addressed algorithms were theoretically explained and demonstrated with practical results from the implemented library. The output images and execution times for each algorithms were obtained and compared. These results were congruent with the theoretical explanations, proving the statements that were made.

Relevant and improved variations engendered from the image processing and computer vision communities were presented to show different approaches of the implemented algorithms, regarding computational complexity and execution times. An algorithm can be implemented in different ways resulting in the same output but with different execution times. On the other hand, different algorithms, were the main differences are either their computational complexity, execution time or image details quality, can provide similar outputs.

The main objective for this work, the elaboration of an introductory knowledge base that future beginners and researchers can consult, was accomplished with success.

[1] https://github.com/josemmrf/PIPpackage

6.2 Future Work

The next steps should be the addition and implementation of other variations of the addressed algorithms and methods, and more complex and advanced concepts such as facial detection, watershed transforms, Compute Unified Device Architecture programming and machine learning methods, among other abroad subjects. Also the development of a GUI for an easier user experience.

In the library some improvements can be done regarding good programming conducts, such as optimizing memory usage, code re-utilization and adding more comments, and investigating the issue with the performance of the MinMax Algorithm with large window sizes.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf.

[0] R. Szeliski. *Computer Vision: Algorithms and Applications*. 2010. URL: http://szeliski.org/Book/..

[0] G. Goos et al. *LNCS 5507 - Advances in Neuro-Information Processing*.

[0] I. Kovács. *Human development of perceptual organization*. 2000. URL: www.elsevier.com.

[0] *Computer vision*. URL: https://en.wikipedia.org/wiki/Computer_vision.

[0] G. R. Bradski and A. Kaehler. *Learning OpenCV : computer vision with the OpenCV library*. O'Reilly, 2008, p. 555. ISBN: 9780596516130.

[0] P. Agrawal and J. S. Verma. "A Survey of Linear and Non-Linear Filters for Noise Reduction". In: *International Journal of Advance Research in Computer Science and Management Studies* 1 (3 2013). ISSN: 2321-7782. URL: www.ijarcsms.com.

[0] *Digital image processing*. URL: https://en.wikipedia.org/wiki/Digital_image_processing.

[0] M. J. Mcauliffe' et al. *Medical Image Processing, Analysis and Visualization In Clinical Research*.

[0] I. N. I. N. Bankman. *Handbook of medical image processing and analysis*. Elsevier/Academic Press, 2009, p. 984. ISBN: 9780123739049.

[0] E. Davies. *Image Processing for the Food Industry*. May 2000. ISBN: 981-02-4022-8. DOI: 10.1142/4182.

[0] S.-H. Huang and Y.-C. Pan. "Automated visual inspection in the semiconductor industry: A survey". In: *Computers in Industry* 66 (2015), pp. 1–10. ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2014.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0166361514001845.

[0] I. of Electrical and E. Engineers. "ICIP 2014 : 2014 IEEE International Conference on Image Processing (ICIP) took place 27-30 October 2014 in Paris, France." In: (), p. 6105.

[0] *A Brief History of MATLAB*. URL: https://www.mathworks.com/company/newsletters/articles/a-brief-history-of-matlab.html.

[0] D. Higham and N. Higham. "Matlab guide". In: 2000.

[0] B. RHunt, R. LLipsman, and J. MRosenberg. *A Guide to MATLAB*.

[0] *Computer Vision Toolbox*. URL: https://www.mathworks.com/products/computer-vision.html.

[0] *Image Processing Toolbox*. URL: https://www.mathworks.com/products/image.html.

[0] *Image Acquisition Toolbox*. URL: https://www.mathworks.com/products/image-acquisition.html.

[0] *OpenCV Org*. URL: https://opencv.org/about/.

[0] I. Culjak et al. *A brief introduction to OpenCV*. 2012.

[0] A. Clark. *Pillow (PIL Fork) Documentation*. 2022.

[0] *CVIPTools Org*. URL: https://cviptools.ece.siue.edu/.

[0] D. Mishra et al. *Computer Vision and Image Processing: Development and Optimization of MATLAB CVIP Toolbox*. 2018.

[0] D. K. Mishra et al. "Image processing and pattern recognition with CVIPtools MATLAB toolbox: automatic creation of masks for veterinary thermographic images". In: vol. 9971. SPIE, Sept. 2016, 99713G. ISBN: 9781510603332. DOI: 10.1117/12.2238183.

[0] S. E. Umbaugh. *Digital image processing and analysis: Applications with MATLAB® and CVIPtools: Third edition*. CRC Press, Jan. 2017, pp. 1–874. ISBN: 9781498766067. DOI: 10.1201/9781351228374.

[0] P. D. Kovesi. *MATLAB and Octave Functions for Computer Vision and Image Processing*. Available from: <https://www.peterkovesi.com/matlabfns/>.

[0] *Scilab Org*. URL: https://www.scilab.org/about.

[0] *Scilab Algos*. URL: https://www.scilab.org/software/scilab/algos-development.

[0] *Scilab App*. URL: https://www.scilab.org/software/scilab/app-development.

[0] *Scilab Wiki*. URL: https://pt.wikipedia.org/wiki/Scilab.

[0] *Scilab IPCV*. URL: https://atoms.scilab.org/toolboxes/IPCV.

[0] *Scilab CV*. URL: https://atoms.scilab.org/toolboxes/scicv.

[0] *Scilab IPD*. URL: https://atoms.scilab.org/toolboxes/IPD.

[0] *Balu Org*. URL: https://domingomery.ing.puc.cl/about-me/.

[0] *Balu Wiki*. URL: https://github.com/domingomery/Balu/wiki.

[0] *Balu Install*. URL: https://github.com/domingomery/Balu/wiki/Installation.

[0] *PD Org*. URL: https://pdollar.github.io/toolbox/.

[0] R. Haining. "Spatial Autocorrelation". In: *International Encyclopedia of the Social and Behavioral Sciences*. Ed. by N. J. Smelser and P. B. Baltes. Oxford: Pergamon, 2001, pp. 14763–14768. ISBN: 978-0-08-043076-8. DOI: https://doi.org/10.1016/B0-08-043076-7/02511-0. URL: https://www.sciencedirect.com/science/article/pii/B0080430767025110.

[0] A. K. R. Tinku Acharya. *Image Processing: Principles and Applications*. 2005.

[0] *Interpolation Definition*. URL: https://www.cambridgeincolour.com/tutorials/image-interpolation.htm.

[0] H. B. Kekre and S. D. Thepade. "Improving color to gray and back using Kekre's LUV color space". In: 2009, pp. 1218–1223. ISBN: 9781424429288. DOI: 10.1109/IADCC.2009.4809189.

[0] S. Feruza and T.-H. Kim. *Review on YCrCb color space optimization, TV images compression, Algorithm of Signals Sources Isolation and Optical Fiber*. 2009.

[0] *Digital Signal Processing Konstantinos N. Plataniotis. Anastasios N. Venetsanopoulos Color Image Processing and Applications*.

[0] N. M. M. F. R. H. C. Li et al. *The CIECAM02 color appearance model Recommended Citation The CIECAM02 Color Appearance Model*. 2002. URL: https://scholarworks.rit.edu/other.

[0] *HSV Color Model*. URL: https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder_saturation_gray.png. Accessed 10 September 2021.

[0] *A Theory Based on Conversion of RGB image to Gray image*.

[0] P. W. Palumbo, P. Swaminathan, and S. N. Srihari. *Document image binarization: Evaluation of algorithms*. URL: http://proceedings.spiedigitallibrary.org/.

[0] P. D. Wellner. *Adaptive Thresholding for the DigitalDesk*. 1993.

[0] G. Walberg. *GEOMETRIC TRANSFORMATION TECHNIQUES FOR DIGITAL I IAGES: A SURVEY*. 1988.

[0] H. F. R. Chester. *Lecture 2: Geometric Image Transformations*. 2005.

[0] N. Efford. "Digital image processing: a practical introduction using JAVA". In: (Jan. 2000).

[0] K. K. Nagwanshi and C. Khare. "Image Restoration Technique with Non Linear Filter Image denoising View project Parallel Implementation of k means Clustering View project Image Restoration Technique with Non Linear Filters". In: *International Journal of Engineering Trends and Technology* (). ISSN: 2231-5381. URL: https://www.researchgate.net/publication/303996052.

[0] P. Agrawal and J. S. Verma. "A Survey of Linear and Non-Linear Filters for Noise Reduction". In: *International Journal of Advance Research in Computer Science and Management Studies* 1 (3 2013). ISSN: 2321-7782. URL: www.ijarcsms.com.

[4] M. B. Gupta, M. Shailendra, and S. Negi. *Image Denoising with Linear and Non-Linear Filters: A REVIEW*. 2013. URL: www.IJCSI.org.

[0] P. Patidar, M. Gupta, and A. K. Nagawat. *Image De-noising by Various Filters for Different Noise Sumit Srivastava*. 2010, pp. 975–8887.

[0] A. K. Boyat and B. K. Joshi. "A Review Paper : Noise Models in Digital Image Processing". In: *Signal and Image Processing : An International Journal* 6 (2 Apr. 2015), pp. 63–75. ISSN: 22293922. DOI: 10.5121/sipij.2015.6206.

[0] G. T. Shrivakshan and C Chandrasekar. *A Comparison of various Edge Detection Techniques used in Image Processing*. 2012. URL: www.IJCSI.org.

[0] R. P. Singh and M. Dixit. "Histogram Equalization: A Strong Technique for Image Enhancement". In: *International Journal of Signal Processing, Image Processing and Pattern Recognition* 8 (8 Aug. 2015), pp. 345–352. ISSN: 20054254. DOI: 10.14257/ijsip.2015.8.8.35.

[0] S. S. Bagade, V. K. Shandilya, and A. Proffessor. *USE OF HISTOGRAM EQUALIZATION IN IMAGE PROCESSING FOR IMAGE ENHANCEMENT*.

[7] M Abdullah-Al-Wadud et al. *A Dynamic Histogram Equalization for Image Contrast Enhancement*. 2007.

[0] R. D. Yates and D. J. Goodman. *Probability and stochastic processes : a friendly introduction for electrical and computer engineers*. John Wiley, 1999, p. 454. ISBN: 0471178373.

[0] R. M. Haralick, S. R. Sternberg, and X. Zhuang. *Image Analysis Using Mathematical Morphology*. 1987.

[0] M. Goyal. "Morphological image processing". In: *IJCST* 2.4 (2011).

[0] "CRC Image Processing and Mathematical Morphology Fundamentals and Applications 1420089439". In: ().

[0] S. Soni, R. Chadha, and S. Kaur. "A Review Paper on Thinning of Image Using Zhang and Suen Algorithm and Neural Network". In: 18 (2), pp. 48–51. DOI: 10.9790/0661-1802054851. URL: www.iosrjournals.org.

[0] R Yogamangalam and B Karthikeyan. *Segmentation Techniques Comparison in Image Processing*.

[0] W. X. Kang, Q. Q. Yang, and R. P. Liang. "The comparative research on image segmentation algorithms". In: vol. 2. 2009, pp. 703–707. ISBN: 9780769535579. DOI: 10.1109/ETCS.2009.417.

[0] N. M. Zaitoun and M. J. Aqel. "Survey on Image Segmentation Techniques". In: vol. 65. Elsevier, 2015, pp. 797–806. DOI: 10.1016/j.procs.2015.09.027.

[0] S. S. Al-amri and N. Kalyankar. *Image Segmentation by Using Thershod Techniques*. 2010.

[0] J. Sklansky and S. Member. *Image Segmentation and Feature Extraction*. 1978.

[0] G. Kumar and P. K. Bhatia. "A detailed review of feature extraction in image processing systems". In: Institute of Electrical and Electronics Engineers Inc., 2014, pp. 5–12. ISBN: 9781479949106. DOI: 10.1109/ACCT.2014.74.

[0] M. S. Nixon and A. S. Aguado. *Feature extraction and image processing for computer vision*. Elsevier, Jan. 2019, pp. 1–626. ISBN: 9780128149768. DOI: 10.1016/C2017-0-02153-5.

[2] J. M. Fonseca. "Acondicionamento de Imagem". In: ().

[3] J. W. Hwang and H. S. Lee. "Adaptive Image Interpolation Based on Local Gradient Features". In: *IEEE Signal Processing Letters* 11 (3 Mar. 2004), pp. 359–362. ISSN: 10709908. DOI: 10.1109/LSP.2003.821718.

[5] T. Sun. *Pattern Recognition Letters Detail-preserving median based filters in image processing*. 1994.

[6] S. Gupta, C. Gupta, and S. K. Chakarvarti. *Image Edge Detection: A Review*. 2013. URL: www.ijarcet.org.

Annex 1 Library Code

```
def bicubic(xorig, yorig, img, verb=False):
    '''
    Bi-cubic interpolation
    :param xorig: X coordinate
    :param yorig: Y coordinate
    :param img: Image to interpolate
    :param verb: True if messages are expected
    :return: rounded value of resulting bi-cubic interpolation
    '''
    x1 = floor(xorig)
    y1 = floor(yorig)

    listoff = [[-1, -1], [-1, 0], [-1, 1], [-1, 2], [0, -1],
               [0, 0], [0, 1], [0, 2], [1, -1], [1, 0], [1, 1], [1, 2],
              [2, -1], [2, 0], [2, 1], [2, 2]]

    values = [img[y1 + off[0]][x1 + off[1]] for off in listoff]

    vvs = [bicubic_aux(values[0:4], xorig - x1, verb),
           bicubic_aux(values[4:8], xorig - x1, verb),
           bicubic_aux(values[8:12], xorig - x1, verb),
           bicubic_aux(values[12:16], xorig - x1, verb)]
    res = bicubic_aux(vvs[0:4], yorig - y1, verb)

    if res < 0:
        res = 0
    elif res > 255:
        res = 255

    return round(res)
```

Listing 1: Bicubic Interpolation Algorithm.

```
def bilinear(xorig, yorig, img, verb=False):
    '''
    Bilinear interpolation
    :param xorig: X coordinate
    :param yorig: Y coordinate
    :param img: Image to interpolate
    :param verb: True if messages are expected
    :return: rounded value of resulting bilinear interpolation
    '''
    x1 = floor(xorig)
    x2 = ceil(xorig)
    y1 = floor(yorig)
    y2 = ceil(yorig)
    v1 = img[y1][x1]
    v2 = img[y1][x2]
    v3 = img[y2][x1]
    v4 = img[y2][x2]

    vv1 = interp(v1, v2, xorig - x1, verb)
    vv2 = interp(v3, v4, xorig - x1, verb)
    res = interp(vv1, vv2, yorig - y1, verb)

    if res < 0:
        res = 0
    elif res > 255:
        res = 255

    return round(res)
```

Listing 2: Bilinear Interpolation Algorithm.

```
def TranslateImg(img, x0, y0, verb=False):
    '''
    Translation Operation
    :param img: Image to apply the operation translation
    :param x0: x offset
    :param y0: y offset
    :param verb: True if messages are expected
    :return: Translated image
    '''

    height = img.shape[0]
    width = img.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)

    old_time = time.time()

    for y in range(0, height):
        for x in range(0, width):
```

```
            (a, b) = calcTranslation(x, y, x0, y0)

            if verb:
                print("New coordinates: ", a, b)

            if 0 < a < width - 1 and 0 < b < height - 1:
                res[b][a] = img[b][a]

    new_time = time.time()

    (...)

    return res
```

Listing 3: Translation Algorithm.

```
def RotateImg(img, angle, sel='BL', verb=False):
    '''
    Rotation Operation
    :param img: Image to apply the operation rotation
    :param angle: Rotation angle in  degrees
    :param sel: Selects bilinear (BL) or bicubic (BC) interpolation
    :param verb: True if messages are expected
    :return: Rotated image
    '''

    height = img.shape[0]
    width = img.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)

    old_time = time.time()

    for y in range(0, height):
        for x in range(0, width):
            (a, b) = calcRot(x, y, angle)
            if 0 < a < width - 2 and 0 < b < height - 2:
                if ceil(a) - a != 0 or ceil(b) - b != 0:
                    if sel == 'BL':
                        res[y][x] = bilinear(a, b, img, verb)
                    if sel == 'BC':
                        res[y][x] = bicubic(a, b, img, verb)
                    if sel == 'NN':
                        res[y][x] = img[round(b)][round(a)]
                else:
                    res[y][x] = img[b][a]

    new_time = time.time()

    (...)
```

```
34
35      return res
```

Listing 4: Rotation Algorithm.

```
1   def ResizeImg(img, s, sel='BL', verb=False):
2       '''
3       Resizing Operation
4       :param img: Image to apply the operation rotation
5       :param s: Resizing factor
6       :param sel: Selects bilinear (BL) or bicubic (BC) interpolation
7       :param verb: True if messages are expected
8       :return: Resized image
9       '''
10
11      height = img.shape[0]
12      width = img.shape[1]
13      res = np.zeros((height, width), dtype=np.uint8)
14
15      old_time = time.time()
16
17      for y in range(0, height):
18          for x in range(0, width):
19              (a, b) = calcZoom(x, y, s)
20              if s <= 1:
21                  if 0 < a < width - 2 and 0 < b < height - 2:
22                      if sel == 'BL':
23                          res[y][x] = bilinear(a, b, img, verb)
24                      if sel == 'BC':
25                          res[y][x] = bicubic(a, b, img, verb)
26                      if sel == 'NN':
27                          res[y][x] = img[round(b)][round(a)]
28              else:
29                  if sel == 'BL':
30                      res[y][x] = bilinear(a, b, img, verb)
31                  if sel == 'BC':
32                      res[y][x] = bicubic(a, b, img, verb)
33                  if sel == 'NN':
34                      res[y][x] = img[round(b)][round(a)]
35
36      new_time = time.time()
37
38      (...)
39
40      return res
```

Listing 5: Resizing Algorithm.

```python
def meanFilterA(image, dimX, dimY, verb=False):
    '''
    Mean filter
    :param image: image to be filtered
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    count_sum = 0
    count_div = 0
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.ubyte)

    old_time = time.time()

    for y in range(offY, height - offY):
        for x in range(offX, width - offX):
            values = [image[y + i][x + j] for i in range(-offY, offY + 1)
                      for j in range(-offX, offX + 1)]
            mean = 0
            if verb:
                print('Values to consider:', values)
            for value in values:
                mean += value
            count_sum += dimX * dimY
            res[y - offY][x - offX] = mean / (dimX * dimY)
            count_div += 1
            if verb:
                print('Result: ', res[y - offY][x - offX])

    new_time = time.time()

    (...)

    return res
```

Listing 6: Linear Mean Filter with Algorithm A.

```python
def meanFilterB(image, dimX, dimY, verb=False):
    '''
    Mean filter
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
    :param image: image to be filtered
    :param verb: True if messages are expected
```

```
    :return: returns the result image
    '''
    count_div = 0
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint)
    res2 = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.ubyte)

    old_time = time.time()

    values = [image[offY + i][offX + j] for i in range(-offY, offY + 1)
    for j in range(-offX, offX + 1)]
    mean = 0
    if verb:
        print('Values to consider:', values)
    for value in values:
        mean += value
    res[0][0] = mean
    count_sum = (dimX * dimY)
    if verb:
        print('Result: ', int(res[0][0] / (dimX * dimY)))

    for x in range(offX + 1, width - offX):
        values_sub = [image[offY + i][x - (offX + 1)] for i in range(-offY, offY + 1)]
        values_add = [image[offY + j][x + offX] for j in range(-offY, offY + 1)]
        sub = 0
        add = 0
        if verb:
            print('Values to subtract:', values_sub)
            print('Values to add:', values_add)
        for value in values_sub:
            sub += value
        for value in values_add:
            add += value
        mean = mean + add - sub
        count_sum += (dimX + dimY)
        res[0][x - offX] = mean
        if verb:
            print('Result: ', int(res[0][x - offX] / (dimX * dimY)))

    for y in range(offY + 1, height - offY):
        values_sub = [image[y - (offY + 1)][offX + i] for i in range(-offX, offX + 1)]
        values_add = [image[y + offY][offX + j] for j in range(-offX, offX + 1)]
        sub = 0
        add = 0
        if verb:
            print('Values to subtract:', values_sub)
            print('Values to add:', values_add)
```

```
58          for value in values_sub:
59              sub += value
60          for value in values_add:
61              add += value
62          mean = res[y - offY - 1][0]
63          mean = mean + add - sub
64          count_sum += (dimX + dimY)
65          res[y - offY][0] = mean
66          if verb:
67              print('Result: ', int(res[y - offY][0] / (dimX * dimY)))
68
69          for x in range(offX + 1, width - offX):
70              values_sub = [image[y + i][x - (offX + 1)] for i in range(-offY, offY + 1)]
71              values_add = [image[y + j][x + offX] for j in range(-offY, offY + 1)]
72              sub = 0
73              add = 0
74              if verb:
75                  print('Values to subtract:', values_sub)
76                  print('Values to add:', values_add)
77              for value in values_sub:
78                  sub += value
79              for value in values_add:
80                  add += value
81              mean = mean + add - sub
82              count_sum += (dimX + dimY)
83              res[y - offY][x - offX] = mean
84              if verb:
85                  print('Result: ', int(res[y - offY][x - offX] / (dimX * dimY)))
86
87      for y in range(0, (height - (offY * 2))):
88          for x in range(0, (width - (offX * 2))):
89              # res[y][x] = res[y][x] / (dimX * dimY)
90              res2[y][x] = res[y][x] / (dimX * dimY)
91              count_div += 1
92
93      new_time = time.time()
94
95      (...)
96
97      return res2
```

Listing 7: Linear Mean Filter with Algorithm B.

```
1   def meanFilterC(image, dimX, dimY, verb=False):
2       '''
3       Mean filter
4       :param image: image to be filtered
5       :param dimX: X dimension of the kernel
6       :param dimY: Y dimension of the kernel
```

```
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    count_div = 0
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    conneroff = [[-(offY + 1), -(offX + 1)], [-(offY + 1), offX], [offY, -(offX + 1)],
    [offY, offX]]
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint)
    res2 = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.ubyte)

    old_time = time.time()

    values = [image[offY + i][offX + j] for i in range(-offY, offY + 1)
    for j in range(-offX, offX + 1)]
    mean = 0
    if verb:
        print('Values to consider:', values)
    for value in values:
        mean += value
    res[0][0] = mean
    count_sum = (dimX * dimY)
    if verb:
        print('Result: ', int(res[0][0] / (dimX * dimY)))

    for x in range(offX + 1, width - offX):
        values_sub = [image[offY + i][x - (offX + 1)] for i in range(-offY, offY + 1)]
        values_add = [image[offY + j][x + offX] for j in range(-offY, offY + 1)]
        sub = 0
        add = 0
        if verb:
            print('Values to subtract:', values_sub)
            print('Values to add:', values_add)
        for value in values_sub:
            sub += value
        for value in values_add:
            add += value
        mean = mean + add - sub
        count_sum += (dimX + dimY)
        res[0][x - offX] = mean
        if verb:
            print('Result: ', int(res[0][x - offX] / (dimX * dimY)))

    for y in range(offY + 1, height - offY):
        values_sub = [image[y - (offY + 1)][offX + i] for i in range(-offX, offX + 1)]
        values_add = [image[y + offY][offX + j] for j in range(-offX, offX + 1)]
        sub = 0
        add = 0
```

```
        if verb:
            print('Values to subtract:', values_sub)
            print('Values to add:', values_add)
        for value in values_sub:
            sub += value
        for value in values_add:
            add += value
        mean = res[y - offY - 1][0]
        mean = mean + add - sub
        count_sum += (dimX + dimY)
        res[y - offY][0] = mean
        if verb:
            print('Result: ', int(res[y - offY][0] / (dimX * dimY)))

        for x in range(offX + 1, width - offX):
            m1 = res[y - (offY + 1)][x - offX]
            m2 = res[y - (offY + 1)][x - (offX + 1)]
            m3 = res[y - offY][x - (offX + 1)]
            conner_values = [image[y + off[0]][x + off[1]] for off in conneroff]
            if verb:
                print('Values to subtract:', m2, conner_values[0:2])
                print('Values to add:', m1, m3, conner_values[3])
            mean = m1 - m2 + m3 + conner_values[0] - conner_values[1] -
            conner_values[2] + conner_values[3]
            count_sum += 6
            res[y - offY][x - offX] = mean
            if verb:
                print('Result: ', int(res[y - offY][x - offX] / (dimX * dimY)))

    for y in range(0, height - (offX * 2)):
        for x in range(0, width - (offX * 2)):
            res2[y][x] = res[y][x] / (dimX * dimY)
            count_div += 1

    new_time = time.time()

    (...)

    return res2
```

Listing 8: Linear Mean Filter with Algorithm C.

```
def meanNonUniFilter(coefficients, image, dimX, dimY, verb=False):
    '''
    Mean filter with coefficients associated
    :param coefficients: coefficient list
    :param image: image to be filtered
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
```

```
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint8)
    c = 0

    old_time = time.time()

    for coff in coefficients:
        c = c + coff

    for y in range(offY, height - offY):
        for x in range(offX, width - offX):
            values = [image[y + i][x + j] for i in range(-offY, offY + 1) for
                      j in range(-offX, offX + 1)]
            mean = 0
            if verb:
                print('Values do consider: ', values)
                print('Coefficients: ', coefficients)
            newvalues = [value * coefficient for value,
                         coefficient in zip(values, coefficients)]

            for value in newvalues:
                mean += value

            res[y - offY][x - offX] = mean / c
            if verb:
                print('New values after coefficients to consider: ', newvalues)
                print('Result: ', res[y - offY][x - offX])

    new_time = time.time()

    (...)

    return res
```

Listing 9: Nonuniform Mean Filter.

```
def meanNonLinearFilter(image, dimX, dimY, t, verb=False):
    '''
    Nonlinear mean filter
    :param image: image to be filtered
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
    :param t: mean value threshold
```

```
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint8)

    old_time = time.time()

    for y in range(offY, height - offY):
        for x in range(offX, width - offX):
            values = [image[y + i][x + j] for i in range(-offY, offY + 1)
                      for j in range(-offX, offX + 1)]
            mean = 0
            if verb:
                print('Values to consider:', values)
            for value in values:
                mean += value
            if abs(image[y - offY][x - offX] - mean / (dimX * dimY)) < t:
                res[y - offY][x - offX] = mean / (dimX * dimY)
            else:
                res[y - offY][x - offX] = image[y - offY][x - offX]
            if verb:
                print('Value to compare to threshold: ',
                int(abs(image[y - offY][x - offX] - mean / (dimX * dimY))))
                print('Result: ', res[y - offY][x - offX])

    new_time = time.time()

    (...)

    return res
```

Listing 10: Nonlinear Mean Filter.

```
def bubblesortA(unsorted, mode):
    '''
    Ascending or Descending sorter
    :param mode: ascending (A) or descending (D) mode
    :param unsorted: list to be sorted
    :return: returns sorted list
    '''
    i = 0
    res = np.zeros(len(unsorted), dtype=np.uint8)

    if mode == 'A':
        while len(unsorted) > 0:
```

```
            value = unsorted[0]
            for x in range(0, len(unsorted)):
                if value >= unsorted[x]:
                    value = unsorted[x]
                    index = x
                else:
                    continue
            res[i] = unsorted.pop(index)
            i += 1

    if mode == 'D':
        while len(unsorted) > 0:
            value = unsorted[0]
            for x in range(0, len(unsorted)):
                if unsorted[x] >= value > 0:
                    value = unsorted[x]
                    index = x
                else:
                    continue
            res[i] = unsorted.pop(index)
            i += 1

    return res
```

Listing 11: Slow BubbleSort Algorithm.

```
def bubblesortB(unsorted, dimX, dimY, mode):
    '''
    Ascending or Descending sorter
    :param unsorted: list to be sorted
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
    :param mode: ascending (A) or descending (D) mode
    :return: returns sorted list
    '''
    i = 0
    res = np.zeros(int((dimX * dimY) / 2 + 1), dtype=np.uint8)

    if mode == 'A':
        while len(unsorted) > int((dimX * dimY) / 2):
            value = unsorted[0]
            for x in range(0, len(unsorted)):
                if value >= unsorted[x]:
                    value = unsorted[x]
                    index = x
                else:
                    continue
            res[i] = unsorted.pop(index)
            i += 1
```

```
24
25      if mode == 'D':
26          while len(unsorted) > int((dimX * dimY) / 2):
27              value = unsorted[0]
28              for x in range(0, len(unsorted)):
29                  if unsorted[x] >= value > 0:
30                      value = unsorted[x]
31                      index = x
32                  else:
33                      continue
34              res[i] = unsorted.pop(index)
35              i += 1
36
37      return res
```

Listing 12: Fast BubbleSort Algorithm.

```
1   def minmax(unsorted):
2       '''
3       MinMax Sorter
4       :param unsorted: list to be sorted
5       :return: returns sorted list
6       '''
7       k = len(unsorted) - 1
8
9       for i in range(0, int(len(unsorted) / 2)):
10          for j in range(i, len(unsorted) - i):
11              if unsorted[j] > unsorted[k]:
12                  tmp = unsorted[j]
13                  unsorted[j] = unsorted[k]
14                  unsorted[k] = tmp
15              if unsorted[i] > unsorted[j]:
16                  tmp = unsorted[i]
17                  unsorted[i] = unsorted[j]
18                  unsorted[j] = tmp
19          k -= 1
20
21      return unsorted
```

Listing 13: MinMax Algorithm.

```
1   def medianFilter(image, dimX, dimY, sel='SB', verb=False):
2       '''
3       Median filter
4       :param image: image to be filtered
5       :param dimX: X dimension of the kernel
6       :param dimY: Y dimension of the kernel
7       :param sel: Selects the sorting algorithm (SB = Slow BubleSort;
```

```
    FB = Fast BubleSort; MM = MinMax)
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint8)

    old_time = time.time()

    for y in range(offY, height - offY):
        for x in range(offX, width - offX):
            values = [image[y + i][x + j] for i in range(-offY, offY + 1) for
            j in range(-offX, offX + 1)]

            if verb:
                print('Values to consider:', values)
            if sel == 'SB':
                sorted_values = bubblesortA(values, 'A')
            if sel == 'FB':
                sorted_values = bubblesortB(values, dimX, dimY, 'A')
            if sel == 'MM':
                sorted_values = minmax(values)
            if verb:
                print('Sorted values:', sorted_values)
                print('Median value: ', sorted_values[int((dimX * dimY) / 2)])

            res[y - offY][x - offX] = sorted_values[int((dimX * dimY) / 2)]

    new_time = time.time()

    (...)

    return res
```

Listing 14: Median Filter Algorithm.

```
def modeFilter(image, dimX, dimY, verb=False):
    '''
    Mode filter
    :param image: image to be filtered
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    def exists(p, tab):
```

```
        for j in range(0, len(tab)):
            if p in tab[j]:
                return 0
        return -1

    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint8)

    old_time = time.time()

    for y in range(offY, height - offY):
        for x in range(offX, width - offX):
            values = [image[y + i][x + j] for i in range(-offY, offY + 1) for
            j in range(-offX, offX + 1)]
            table = []
            if verb:
                print('Values to consider: ', values)
            for level in values:
                count = 0
                if exists(level, table) == -1:
                    for pixel in values:
                        if pixel == level:
                            count += 1
                    table.append([level, count])
            if verb:
                print('Number of pixel occurrences in kernel: ', table)
            for entry in table:
                if count <= entry[1]:
                    count = entry[1]
                    value = entry[0]
            if verb:
                print('Pixel with most occurrences: ', value)

            res[y - offY][x - offX] = value

    new_time = time.time()

    (...)

    return res
```

Listing 15: Mode Filter Algorithm.

```
def difFilter(image, verb=False):
    '''
    Differentiation filter
```

```
    :param image: image to be filtered
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    listoff = [[0, 0], [0, 1], [1, 0]]
    height = image.shape[0]
    width = image.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)

    old_time = time.time()

    for y in range(0, height - 1):
        for x in range(0, width - 1):
            values = [image[y + off[0]][x + off[1]] for off in listoff]
            if verb:
                print('Values to consider: ', values)
            res[y][x] = abs(values[0] - values[1]) + abs(values[0] - values[2])
            if verb:
                print('Result: ', res[y][x])

    new_time = time.time()

    (...)

    return res
```

Listing 16: Differentiation Operator.

```
def robertsFilter(image, verb=False):
    '''
    Roberts filter
    :param image: image to be filtered
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    listoff = [[0, 0], [0, 1], [1, 0], [1, 1]]
    height = image.shape[0]
    width = image.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)

    old_time = time.time()

    for y in range(0, height - 1):
        for x in range(0, width - 1):
            values = [image[y + off[0]][x + off[1]] for off in listoff]
            if verb:
                print('Values to consider: ', values)
            res[y][x] = abs(values[0] - values[3]) + abs(values[1] - values[2])
            if verb:
```

```
                print('Result: ', res[y][x])

    new_time = time.time()

    (...)

    return res
```

Listing 17: Roberts Operator.

```
def prewittFilter(image, verb=False):
    '''
    Prewitt filter
    :param image: image to be filtered
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    listoffx = [[-1, -1], [1, -1], [-1, 1], [1, 1], [0, -1], [0, 1]]
    listoffy = [[-1, -1], [-1, 1], [-1, 0], [1, -1], [1, 1], [1, 0]]
    height = image.shape[0]
    width = image.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)

    old_time = time.time()

    for y in range(1, height - 1):
        for x in range(1, width - 1):
            valuesx = [image[y + off[0]][x + off[1]] for off in listoffx]
            valuesy = [image[y + off[0]][x + off[1]] for off in listoffy]
            if verb:
                print('X values to consider: ', valuesx)
                print('Y values to consider: ', valuesy)
            sx = abs((valuesx[0] + valuesx[1] + valuesx[4]) -
            (valuesx[2] + valuesx[3] + valuesx[5]))

            sy = abs((valuesy[3] + valuesy[4] + valuesy[5]) -
            (valuesy[0] + valuesy[1] + valuesy[2]))
            res[y - 1][x - 1] = sx + sy
            if verb:
                print('X Operator: ', sx)
                print('Y Operator: ', sy)
                print('Result: ', res[y - 1][x - 1])

    new_time = time.time()

    (...)

    return res
```

Listing 18: Prewitt Operator.

```
def sobelFilter(image, verb=False):
    '''
    Sobel filter
    :param image: image to be filtered
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    listoffx = [[-1, -1], [1, -1], [-1, 1], [1, 1], [0, -1], [0, 1]]
    listoffy = [[-1, -1], [-1, 1], [-1, 0], [1, -1], [1, 1], [1, 0]]
    height = image.shape[0]
    width = image.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)

    old_time = time.time()

    for y in range(1, height - 1):
        for x in range(1, width - 1):
            valuesx = [image[y + off[0]][x + off[1]] for off in listoffx]
            valuesy = [image[y + off[0]][x + off[1]] for off in listoffy]
            if verb:
                print('X values to consider: ', valuesx)
                print('Y values to consider: ', valuesy)
            sx = abs((valuesx[0] + valuesx[1] + valuesx[4] * 2) -
            (valuesx[2] + valuesx[3] + valuesx[5] * 2))

            sy = abs((valuesy[3] + valuesy[4] + valuesy[5] * 2) -
            (valuesy[0] + valuesy[1] + valuesy[2] * 2))
            res[y - 1][x - 1] = sx + sy
            if verb:
                print('X Operator: ', sx)
                print('Y Operator: ', sy)
                print('Result: ', res[y - 1][x - 1])

    new_time = time.time()

    (...)

    return res
```

Listing 19: Sobel Operator.

```
def kNearestFilter(image, dimX, dimY, k, verb=False):
    '''
    K-Nearest Neighbour Filter
    :param image: image to be filtered
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
```

```
    :param k: number of neighbours
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint8)

    old_time = time.time()

    for y in range(offY, height - offY):
        for x in range(offX, width - offX):
            values = [image[y + i][x + j] for i in range(-offY, offY + 1)
                      for j in range(-offX, offX + 1)]
            table = []
            diffs = []
            pixels = []
            mean = 0
            if verb:
                print('Center pixel: ', image[y][x])
                print('Values to consider: ', values)

            for i in range(0, len(values)):
                table.append([values[i], abs(image[y][x] - values[i])])

            for i in range(0, k):
                count = 0
                diff = 256
                for entry in table:
                    if entry[1] < diff:
                        diff = entry[1]
                        pixel = entry[0]
                        index = count
                    count += 1
                diffs.append(diff)
                pixels.append(pixel)
                table.pop(index)

            if verb:
                print('Closest values to center pixel: ', pixels)

            for value in pixels:
                mean += value

            res[y - offY][x - offX] = mean / k

    new_time = time.time()

```

```
    (...)

    return res
```

Listing 20: K-Nearest Mean Filter Algorithm.

```
def sigmaFilter(image, dimX, dimY, s, verb=False):
    '''
    Sigma Filter
    :param image: image to be filtered
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
    :param s: standard variation
    :param verb: True if messages are expected
    :return: returns the result image
    '''
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint8)

    old_time = time.time()

    for y in range(offY, height - offY):
        for x in range(offX, width - offX):
            values = [image[y + i][x + j] for i in range(-offY, offY + 1)
                      for j in range(-offX, offX + 1)]
            if verb:
                print('Values to consider: ', values)
            tab = []
            mean = 0

            for i in range(0, len(values)):
                if image[y][x] - 2 * s <= values[i] <= image[y][x] + 2 * s:
                    tab.append(values[i])
            if verb:
                print('Values that respect the interval: ', tab)

            for entry in tab:
                mean += entry

            res[y - offY][x - offX] = mean / len(tab)

    new_time = time.time()

    (...)

    return res
```

Listing 21: Sigma Filter Algorithm.

```
def medianHybrid5x5(img, sel='SB', verb=False):
    '''
    Hybrid median filter
    :param img: Image to be filtered
    :param sel: Selects the sorting algorithm
    (SB = Slow BubleSort; FB = Fast BubleSort; MM = MinMax)
    :param verb: True if messages are expected
    :return:
    '''
    diag = [[-2, -2], [-1, -1], [0, 0], [1, 1], [2, 2], [2, -2], [1, -1], [-1, 1], [-2, 2]]
    cross = [[0, -2], [0, -1], [0, 0], [0, 1], [0, 2], [-2, 0], [-1, 0], [1, 0], [2, 0]]
    height = img.shape[0]
    width = img.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)

    def getValues(a, b, off, image):
        v = np.zeros([len(off)], dtype=np.uint8)
        for i in range(len(off)):
            v[i] = image[b + off[i][1]][a + off[i][0]]
        return v

    old_time = time.time()

    for y in range(2, height-2):
        for x in range(2, width-2):
            l1 = list(getValues(x, y, diag, img))
            l2 = list(getValues(x, y, cross, img))

            if sel == 'SB':
                v1 = bubblesortA(l1, 'A')
                v2 = bubblesortA(l2, 'A')
                v3 = bubblesortA(list((v1[4], v2[4], img[y][x])), 'A')
                res[y][x] = v3[1]
            if sel == 'FB':
                v1 = bubblesortB(l1, 3, 3, 'A')
                v2 = bubblesortB(l2, 3, 3, 'A')
                v3 = bubblesortB(list((v1[4], v2[4], img[y][x])), 3, 1, 'A')
                res[y][x] = v3[1]
            if sel == 'MM':
                v1 = minmax(l1)
                v2 = minmax(l2)
                v3 = minmax(list((v1[4], v2[4], img[y][x])))
                res[y][x] = v3[1]

    new_time = time.time()

    (...)
```

```
48
49      return res
```

Listing 22: Hybrid Median Filter Algorithm.

```
1   def eqHist(img, maxLevel, verb=False):
2       '''
3       Histogram equalization
4       :param img: image to equalize
5       :param maxLevel: maximum level of gray
6       :param verb: True if messages are expected
7       :return: equalized image
8       '''
9       # Initialize histogram with zero for all the levels
10      hist = {v: 0 for v in range(maxLevel + 1)}
11      tot = 0
12
13      old_time = time.time()
14
15      for y in range(img.shape[0]):
16          for x in range(img.shape[1]):
17              hist[round(img[y][x], 0)] = hist[round(img[y][x], 0)] + 1
18              tot += 1
19
20      if verb:
21          print('Histogram', hist)
22
23      accProb = hist.copy()
24      prev = 0
25      for i in range(len(accProb)):
26          prob = accProb[i] / tot
27          accProb[i] = prob + prev
28          prev += prob
29
30      if verb:
31          print('Accumulated probabilities: ', end='')
32          for i in range(len(accProb)):
33              print("{:.3f}".format(accProb[i]), end='  ')
34          print()
35
36      step = 1 / (maxLevel + 1)
37      newLevels = accProb.copy()
38
39      for i in range(len(newLevels)):
40          newLevels[i] = ceil(newLevels[i] / step) - 1
41
42      if verb:
43          print('Transformation table', newLevels)
44
```

```
    imgRes = deepcopy(img)

    for j in range(imgRes.shape[0]):
        for i in range(imgRes.shape[1]):
            imgRes[j][i] = newLevels[imgRes[j][i]]

    if verb:
        print('Resulting image')
        printImg(imgRes)

    new_time = time.time()

    return imgRes
```

Listing 23: Histogram Equalization Algorithm.

```
def otsuMethod(img, maxLevel=255, verb=False):
    '''
    Otsu Algorithm
    :param img: image to analyse
    :param maxLevel: maximum pixel value in image
    :param verb: True if messages are expected
    :return: Binarized image
    '''
    # Initialize histogram dictionary with zero for all the levels
    hist = {v: 0 for v in range(maxLevel + 1)}
    tot = 0
    height = img.shape[0]
    width = img.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)
    old_time = time.time()

    for y in range(img.shape[0]):
        for x in range(img.shape[1]):
            hist[round(img[y][x], 0)] = hist[round(img[y][x], 0)] + 1
            tot += 1
    if verb:
        print('Histogram', hist)

    probs = hist.copy()

    for i in range(len(probs)):
        prob = probs[i] / tot
        probs[i] = prob

    if verb:
        print('Probabilities: ', end='')
        for i in range(len(probs)):
            print("{:.3f}".format(probs[i]), end='  ')
```

```
        print()

    q1 = probs[0]
    q2 = 1 - q1
    u1 = list(probs.keys())[0]
    u2 = 0
    for i in range(1, len(probs)):
        u2 = (u2 + i * probs[i])

    t = 0

    if u1 > 0:
        maxV = q1 * q2 * pow(u1 / q1 - u2 / q2, 2)
    else:
        maxV = 0

    if verb:
        print('Current threshold= ', t)
        print('Q1= ', q1)
        print('Q2= ', q2)
        print('U1= ', u1 / q1)
        print('U2= ', u2 / q2)
        print('Inter-class variation= ', maxV)

    for i in range(1, len(probs)):
        q1 = q1 + probs[i]
        if q1 == 0:
            continue

        q2 = q2 - probs[i]
        if q2 == 0:
            break

        u1 = u1 + i * probs[i]
        u2 = u2 - i * probs[i]

        var = q1 * q2 * pow(u1 / q1 - u2 / q2, 2)

        if verb:
            print('Current threshold= ', i)
            print('Q1= ', q1)
            print('Q2= ', q2)
            print('U1= ', u1 / q1)
            print('U2= ', u2 / q2)
            print('Inter-class variation= ', var)

        if var > maxV:
            t = i
            maxV = var

```

```
    if verb:
        print('Optimal threshold value= ', t)

    for y in range(img.shape[0]):
        for x in range(img.shape[1]):
            if img[y][x] < t:
                res[y][x] = 0
            else:
                res[y][x] = 255

    new_time = time.time()

    (...)

    return res
```

Listing 24: Otsu Filter Algorithm.

```
def cmeansMethod(img, maxLevel=255, verb=False):
    '''
    C-Means Algorithm
    :param img: image to analyse
    :param maxLevel: maximum pixel value in image
    :param verb: True if messages are expected
    :return: Binarized image
    '''
    # Initialize histogram dictionary with zero for all the levels
    hist = {v: 0 for v in range(maxLevel + 1)}
    tot = 0
    height = img.shape[0]
    width = img.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)

    old_time = time.time()
    for y in range(img.shape[0]):
        for x in range(img.shape[1]):
            hist[round(img[y][x], 0)] = hist[round(img[y][x], 0)] + 1
            tot += 1
    if verb:
        print('Histogram', hist)

    old_t = round(len(hist) / 2)

    for k in range(len(hist)):
        value1 = 0
        value2 = 0
        value3 = 0
        value4 = 0

```

```
32          for i in range(old_t):
33              value1 = value1 + i * hist[i]
34          for i in range(old_t):
35              value2 = value2 + hist[i]
36          for j in range(old_t + 1, maxLevel):
37              value3 = value3 + j * hist[j]
38          for j in range(old_t + 1, maxLevel):
39              value4 = value4 + hist[j]
40
41          if verb:
42              print('Trying current threshold: ', old_t)
43              print('Values for the operation: ', value1, value2, value3, value4)
44
45          new_t = round(1 / 2 * (value1 / value2 + value3 / value4))
46
47          if new_t == old_t:
48              break
49          else:
50              old_t = new_t
51
52      for y in range(img.shape[0]):
53          for x in range(img.shape[1]):
54              if img[y][x] < old_t:
55                  res[y][x] = 0
56              else:
57                  res[y][x] = 255
58
59      new_time = time.time()
60
61      (...)
62
63      return res
```

Listing 25: C-Means Filter Algorithm.

```
1   def wellnerMethodA(image, r=1 / 8, t=15, verb=False):
2       '''
3       Wellner Algorithm
4       :param image: image to binarize
5       :param r: ratio of pixel to process according to image width
6       :param t: percentage do classify the pixel as black or white
7       :param verb: True if messages are expected
8       :return: returns the binary image
9       '''
10      height = image.shape[0]
11      width = image.shape[1]
12      res = np.zeros((height, width), dtype=np.uint8)
13      values = []
14      s = round(width * r)
```

```
15
16      if verb:
17          print('Number of pixels to consider: ', s)
18
19      old_time = time.time()
20
21      for y in range(0, height):
22          for x in range(0, width):
23              values.append(image[y, x])
24              mean = 0
25              for i in values:
26                  mean = mean + i
27
28              mean = mean / len(values)
29
30              if image[y, x] < mean * (1 - t / 100):
31                  res[y, x] = 0
32              else:
33                  res[y, x] = 255
34
35              if verb:
36                  print('Current pixel value: ', image[y, x])
37                  print('Array of the latest values: ', values)
38                  print('Mean of the latest values: ', mean)
39
40              if len(values) == s:
41                  values.pop(0)
42              else:
43                  continue
44
45      new_time = time.time()
46
47      (...)
48
49      return res
```

Listing 26: Wellner Filter with Algorithm A.

```
1   def wellnerMethodB(image, r=1 / 8, t=15, verb=False):
2       '''
3       Wellner Algorithm
4       :param image: image to binarize
5       :param r: ratio of pixel to process according to image width
6       :param t: percentage do classify the pixel as black or white
7       :param verb: True if messages are expected
8       :return: returns the binary image
9       '''
10      height = image.shape[0]
11      width = image.shape[1]
```

```
    res = np.zeros((height, width), dtype=np.uint8)
    values = []
    s = round(width * r)

    if verb:
        print('Number of pixels to consider: ', s)

    old_time = time.time()

    for y in range(0, height):
        for x in range(0, width):
            values.append(image[y, x])
            mean = 0
            index = 0
            for i in values:
                mean = mean + math.pow((1 - 1 / s), index) * i
                index = + 1

            mean = mean / len(values)

            if image[y, x] < mean * (1 - t / 100):
                res[y, x] = 0
            else:
                res[y, x] = 255

            if verb:
                print('Current pixel value: ', image[y, x])
                print('Array of the latest values: ', values)
                print('Mean of the latest values: ', mean)

            if len(values) == s:
                values.pop(0)
            else:
                continue

    new_time = time.time()

    (...)

    return res
```

Listing 27: Wellner Filter with Algorithm B.

```
def brarotMethod(image, dimX, dimY, t=15, verb=False):
    '''
    Bradley & Roth Algorithm
    :param image: image to binarize
    :param r: ratio of pixel to process according to image width
    :param t: percentage do classify the pixel as black or white
```

```
    :param dimX: X dimension of the kernel
    :param dimY: Y dimension of the kernel
    :param verb: True if messages are expected
    :return: returns the binary image
    '''
    height = image.shape[0]
    width = image.shape[1]
    offX = int(dimX / 2)
    offY = int(dimY / 2)
    res = np.zeros((height - (offY * 2), width - (offX * 2)), dtype=np.uint8)

    old_time = time.time()

    for y in range(offY, height - offY):
        for x in range(offX, width - offX):
            values = [image[y + i][x + j] for i in range(-offY, offY + 1)
                      for j in range(-offX, offX + 1)]
            mean = 0
            for i in values:
                mean = mean + i

            mean = mean / (dimX * dimY)

            if image[y - offY][x - offX] < mean * (1 - t / 100):
                res[y - offY][x - offX] = 0
            else:
                res[y - offY][x - offX] = 255

            if verb:
                print('Current pixel value: ', image[y - offY][x - offX])
                print('Array of the latest values: ', values)
                print('Mean of the latest values: ', mean)

    new_time = time.time()

    (...)

    return res
```

Listing 28: Bradley and Roth Algorithm.

```
def dilation(imIn, ker, verb):
    '''
    Dilation of a binary image
    :param imIn: binary image to dilate
    :param ker: dilation kernel
    :param verb: True if messages are expected
    :return: dilated image
    '''
```

```
    (dkY, dkX) = ker.shape
    offX = floor(dkX / 2)
    offY = floor(dkY / 2)
    img = imagePad(imIn, [offY, offY, offX, offX])
    dimY = img.shape[0]
    dimX = img.shape[1]
    imRes = np.zeros((dimY, dimX), dtype=np.uint8)

    old_time = time.time()

    for y in range(offY, dimY - offY):
        for x in range(offX, dimX - offX):
            imRes[y - offY][x - offX] = 0
            for i in range(-offY, offY + 1):
                for j in range(-offX, offX + 1):
                    if img[y + i][x + j] == 1 and ker[i + 1][j + 1] == 1:
                        imRes[y - offY][x - offX] = 1

    new_time = time.time()

    (...)

    return imRes
```

Listing 29: Dilation Operator.

```
def erosion(imIn, ker, verb):
    '''
    Erodes a binary image with the binary kernel
    :param imIn: binary image to erode
    :param ker: erosion kernel
    :param verb: True if messages are expected
    :return:
    '''
    (dkY, dkX) = ker.shape
    offX = floor(dkX / 2)
    offY = floor(dkY / 2)
    img = imagePad(imIn, [offY, offY, offX, offX])
    dimY = img.shape[0]
    dimX = img.shape[1]
    imRes = np.empty_like(imIn)

    old_time = time.time()

    for y in range(offY, dimY - offY):
        for x in range(offX, dimX - offX):
            imRes[y - offY][x - offX] = 1
            for i in range(-offY, offY + 1):
                for j in range(-offX, offX + 1):
```

```
                    if img[y + i][x + j] == 0 and ker[i + 1][j + 1] == 1:
                        imRes[y - offY][x - offX] = 0

    new_time = time.time()

    (...)

    return imRes
```

Listing 30: Erosion Operator.

```
def hitAndMiss(imIn, ker, verb):
    '''
    Hit and Miss operation
    :param imIn: input image
    :param ker: kernel
    :param verb: True if messages are expected
    :return: processed image
    '''
    dkX  = len(ker[1])
    dkY = len(ker)
    offX = floor(dkX / 2)
    offY = floor(dkY / 2)
    img = imagePad(imIn, [offY, offY, offX, offX])
    dimY = img.shape[0]
    dimX = img.shape[1]
    imRes = np.empty_like(imIn)

    old_time = time.time()

    for y in range(offY, dimY - offY):
        for x in range(offX, dimX - offX):
            imRes[y - offY][x - offX] = 1
            for i in range(-offY, offY + 1):
                for j in range(-offX, offX + 1):
                    if ker[i + 1][j + 1] != -1 and
                        ker[i + 1][j + 1] != img[y + i][x + j]:
                            imRes[y - offY][x - offX] = 0

    new_time = time.time()

    (...)

    return imRes
```

Listing 31: Hit and Miss Operator.

```
def blumAlg(img, verb=False):
    '''
    Skeletization of a binary image
    :param img: input image
    :param verb: True if messages are expected
    :return: returns the skeletization of the input image
    '''
    dimY = img.shape[0]
    dimX = img.shape[1]
    border = np.zeros((dimY, dimX), dtype=np.uint8)
    inside = np.zeros((dimY, dimX), dtype=np.uint8)
    res = np.zeros((dimY, dimX), dtype=np.uint8)
    borderP = []
    insideP = []

    def skeleton(e, f, p):
        minDist = math.sqrt(math.pow((e - p[0][1]), 2) + math.pow((f - p[0][0]), 2))
        indMin = 0
        a = 0
        for pos in p:
            dist = math.sqrt(math.pow((e - pos[1]), 2) + math.pow((f - pos[0]), 2))
            if dist < minDist:
                minDist = dist
                indMin = a
            a += 1

        Np = 1
        a = 0
        if verb:
            print('minDist: ', minDist)
            print('indMin: ', indMin)

        for pos in p:
            if a != indMin:
                dist = math.sqrt(math.pow((e - pos[1]), 2) + math.pow((f - pos[0]), 2))
                if abs(dist - minDist) <= 1 and
                    math.sqrt( math.pow((p[a][1] - p[indMin][1]), 2) +
                        math.pow((p[a][0] - p[indMin][0]), 2)) > 20:
                            Np += 1
            a += 1
        return Np

    def findPos(n, im):
        (c, d) = im.shape
        for z in range(0, c):
            for w in range(0, d):
                if im[z][w] != 0:
                    n.append([z, w])

    for y in range(0, img.shape[0]):
```

```
51          for x in range(0, img.shape[1]):
52              if img[y][x] == 255:
53                  img[y][x] = 1
54
55      old_time = time.time()
56
57      # border pixels of the object
58      for i in range(1, dimY - 1):
59          for j in range(1, dimX - 1):
60              if img[i][j] == 1:
61                  if img[i - 1][j] == 0:
62                      border[i][j] = 1
63                  elif img[i + 1][j] == 0:
64                      border[i][j] = 1
65                  elif img[i][j - 1] == 0:
66                      border[i][j] = 1
67                  elif img[i][j + 1] == 0:
68                      border[i][j] = 1
69
70      # interior pixels of the object
71      for i in range(0, dimY):
72          for j in range(0, dimX):
73              inside[i][j] = img[i][j] - border[i][j]
74
75      # get list of border pixel position
76      findPos(borderP, border)
77
78      # get list of inside pixel position
79      findPos(insideP, inside)
80
81      if verb:
82          print('Border:')
83          print(border)
84          print('Border pixel positions', borderP)
85          print('Inside:')
86          print(inside)
87          print('Inside pixel positions', insideP)
88
89      # compare the distance of each object pixel with the border pixels
90      for posObject in insideP:
91          if skeleton(posObject[1], posObject[0], borderP) > 1:
92              res[posObject[0]][posObject[1]] = 1
93
94      res = res + border
95
96      new_time = time.time()
97
98      (...)
99
100     return res
```

Listing 32: Medial Axis Transform Algorithm.

```
def zhSuAlg(img, verb=False, showB=False):
    '''
    Skeletization of a binary image
    :param img: input image
    :param verb: True if messages are expected
    :param showB: Show object border
    :return: returns the skeletization of the input image
    '''

    dimY = img.shape[0]
    dimX = img.shape[1]
    border = np.zeros((dimY, dimX), dtype=np.uint8)
    change = True

    for y in range(0, img.shape[0]):
        for x in range(0, img.shape[1]):
            if img[y][x] == 255:
                img[y][x] = 1

    if showB:
        for i in range(1, dimY - 1):
            for j in range(1, dimX - 1):
                if img[i][j] == 1:
                    if img[i - 1][j] == 0:
                        border[i][j] = 1
                    elif img[i + 1][j] == 0:
                        border[i][j] = 1
                    elif img[i][j - 1] == 0:
                        border[i][j] = 1
                    elif img[i][j + 1] == 0:
                        border[i][j] = 1

    def countBlack(n):
        s = 0
        for value in n:
            s = s + value

        s = 9 - s

        if verb:
            print('Number of black pixels: ', s)

        return s

    def countTransitions(n):
        c = 0
```

```
        for i in range(1, 8):
            if n[i] < n[i + 1]:
                c += 1
        if n[8] < n[1]:
            c += 1

        if verb:
            print('Number of transitions: ', c)

        return c

    old_time = time.time()

    while change:
        change = False
        list1 = []
        list2 = []
        for y in range(1, dimY - 1):
            for x in range(1, dimX - 1):
                values = [img[y][x], img[y - 1][x], img[y - 1][x + 1],
                          img[y][x + 1], img[y + 1][x + 1], img[y + 1][x],
                          img[y + 1][x - 1], img[y][x - 1], img[y - 1][x - 1]]
                if verb:
                    print('Values for step 1: ', values)

                # first step
                if values[0] == 1:
                    if values[1] == 0 or values[3] == 0 or values[5] == 0:
                        if values[3] == 0 or values[5] == 0 or values[7] == 0:
                            if 2 <= countBlack(values) <= 6:
                                if countTransitions(values) == 1:
                                    list1.append([y, x])
                                    change = True

        for pos in list1:
            img[pos[0]][pos[1]] = 0

        for y in range(1, dimY - 1):
            for x in range(1, dimX - 1):
                values = [img[y][x], img[y - 1][x], img[y - 1][x + 1],
                          img[y][x + 1], img[y + 1][x + 1], img[y + 1][x],
                          img[y + 1][x - 1], img[y][x - 1], img[y - 1][x - 1]]

                if verb:
                    print('Values for step 2: ', values)

                # second step
                if values[0] == 1:
                    if values[1] == 0 or values[3] == 0 or values[7] == 0:
                        if values[1] == 0 or values[5] == 0 or values[7] == 0:
```

```
                            if 2 <= countBlack(values) <= 6:
                                if countTransitions(values) == 1:
                                    list2.append([y, x])
                                    change = True

        for pos in list2:
            img[pos[0]][pos[1]] = 0

        if change and verb:
            print('A change occurred')
        if not change and verb:
            print('Image done')

    new_time = time.time()

    (...)

    return img + border
```

Listing 33: Zhang-Suen Algorithm.

```
def multiSeg(img, verb=False):
    '''
    Segmentation of multiclass images - inspired on the classic algorithm
    :param img: input image with any number of labels
    :param verb: True if messages are expected
    :return: labelled image with a different label for each object and the
             number of labels on the resulting image. The result will have
             labels from 0 sequentially to the number of labels-1
    '''

    def getEqs(cTab, lab):
        for i in range(len(cTab)):
            if lab in cTab[i]:
                return (i)
        return (-1)

    def insertConf(cTab, l1, l2):
        id1 = getEqs(cTab, l1)
        id2 = getEqs(cTab, l2)
        if id1 == -1 and id2 != -1:
            cTab[id2].append(l1)
        elif id1 != -1 and id2 == -1:
            cTab[id1].append(l2)
        elif id1 == -1 and id2 == -1:
            cTab.append([l1, l2])
        else:
            if id1 != id2:
                cTab[id1] += cTab[id2]
```

```
                del (cTab[id2])
        return (cTab)

    def labEq(lab, confTab):
        for i in range(len(confTab)):
            if lab in confTab[i]:
                return (min(confTab[i]))
        return (lab)

    height = img.shape[0]
    width = img.shape[1]
    labels = np.empty_like(img, int)

    nextLabel = 0
    labels[0][0] = nextLabel  # First pixel
    nextLabel += 1
    for x in range(1, width):  # First row
        if img[0][x] == img[0][x - 1]:
            labels[0][x] = labels[0][x - 1]
        else:
            labels[0][x] = nextLabel
            nextLabel += 1

    confTab = []

    for y in range(1, height):  # Remaining rows
        if img[y][0] == img[y - 1][0]:  # First column
            labels[y][0] = labels[y - 1][0]
        else:
            labels[y][0] = nextLabel
            nextLabel += 1
        for x in range(1, width):  # Remaining columns
            if img[y][x] == img[y - 1][x]:
                labels[y][x] = labels[y - 1][x]
                if img[y][x] == img[y][x - 1] and labels[y][x] != labels[y][x - 1]:
                    confTab = insertConf(confTab, labels[y][x - 1], labels[y][x])
            elif img[y][x] == img[y][x - 1]:
                labels[y][x] = labels[y][x - 1]
            else:
                labels[y][x] = nextLabel
                nextLabel += 1
    labDict = {}
    off = 0
    for l in range(nextLabel):
        le = labEq(l, confTab)
        if l != le:
            labDict[l] = labDict[le]
            off += 1
        else:
            labDict[l] = l - off
```

```
79
80      for y in range(0, height):
81          for x in range(0, width):
82              labels[y][x] = labDict[labels[y][x]]
83
84      (...)
85
86      return labels, nextLabel - off
```

Listing 34: Multi class Image Segmentation Algorithm.

```
1   def binSeg(img, verb=False):
2       '''
3       Binary segmentation - classic algorithm
4       :param img: binary image to segment
5       :param verb: True if messages are expected
6       :return: segmented image and number of labels (between 1 and N)
7       '''
8
9       def getEqs(cTab, lab):
10          for i in range(len(cTab)):
11              if lab in cTab[i]:
12                  return (i)
13          return (-1)
14
15      def insertConf(cTab, l1, l2):
16          id1 = getEqs(cTab, l1)
17          id2 = getEqs(cTab, l2)
18          if id1 == -1 and id2 != -1:
19              cTab[id2].append(l1)
20          elif id1 != -1 and id2 == -1:
21              cTab[id1].append(l2)
22          elif id1 == -1 and id2 == -1:
23              cTab.append([l1, l2])
24          else:
25              if id1 != id2:
26                  cTab[id1] += cTab[id2]
27                  del (cTab[id2])
28          return (cTab)
29
30      def labEq(lab, confTab):
31          for i in range(len(confTab)):
32              if lab in confTab[i]:
33                  return (min(confTab[i]))
34          return (lab)
35
36      if verb:
37          print('Input image')
38          print(img)
```

```
39
40      height = img.shape[0]
41      width = img.shape[1]
42
43      old_time = time.time()
44
45      # Labels between 1 and N
46      nextLabel = 1
47      if img[0][0] == 1:
48          img[0][0] = nextLabel  # First pixel
49          nextLabel += 1
50      for x in range(1, width):  # First row
51          if img[0][x] == 1:
52              if img[0][x - 1] != 0:
53                  img[0][x] = img[0][x - 1]
54              else:
55                  img[0][x] = nextLabel
56                  nextLabel += 1
57
58      confTab = []
59
60      for y in range(1, height):  # Remaining rows
61          if img[y][0] == 1:
62              if img[y - 1][0] != 0:  # First column
63                  img[y][0] = img[y - 1][0]
64              else:
65                  img[y][0] = nextLabel
66                  nextLabel += 1
67          for x in range(1, width):  # Remaining columns
68              if img[y][x] == 1:
69                  if img[y - 1][x] != 0:
70                      img[y][x] = img[y - 1][x]
71                      if img[y][x - 1] != 0 and img[y][x - 1] != img[y][x]:
72                          if verb:
73                              print('Conflict:', img[y][x - 1], img[y][x])
74                          confTab = insertConf(confTab, img[y][x - 1], img[y][x])
75                  elif img[y][x - 1] != 0:
76                      img[y][x] = img[y][x - 1]
77                  else:
78                      img[y][x] = nextLabel
79                      nextLabel += 1
80      if verb:
81          print('Labels before equivalencies')
82          print(img)
83
84      labDict = {0: 0}
85      off = 0
86      for l in range(1, nextLabel):
87          le = labEq(l, confTab)
88          if l != le:
```

```
89              labDict[l] = labDict[le]
90              off += 1
91          else:
92              labDict[l] = l - off
93      if verb:
94          print('Conflict table')
95          print(confTab)
96          print('Number of labels before equivalencies:', nextLabel)
97          print('Dictionary')
98          print(labDict)
99          print('Number of labels after equivalencies:', nextLabel - off - 1)
100
101     for y in range(0, height):
102         for x in range(0, width):
103             img[y][x] = labDict[img[y][x]]
104
105     new_time = time.time()
106
107     (...)
108
109     return img, nextLabel - off - 1
```

Listing 35: Binary Image Segmentation Classical Algorithm.

```
1   def binItSeg(img, verb=False):
2       '''
3       Binary segmentation - iterative algorithm
4       :param img: binary image to segment
5       :param verb: True if messages are expected
6       :return: segmented image and number of labels (between 1 and N)
7       '''
8
9       height = img.shape[0]
10      width = img.shape[1]
11      change = True
12      cnt = 0
13      res = np.zeros((height, width), dtype=np.long)
14      labels = []
15      color = 0
16
17      old_time = time.time()
18
19      # Labels between 1 and N
20      nextLabel = 1
21      if img[0][0] == 1:
22          res[0][0] = nextLabel  # First pixel
23          nextLabel += 1
24
25      for x in range(1, width):
```

```
        if img[0][x] == 1:
            res[0][x] = nextLabel
            nextLabel += 1

    for y in range(1, height):  # Remaining rows
        if img[y][0] == 1:
            res[y][0] = nextLabel
            nextLabel += 1
        for x in range(1, width):  # Remaining columns
            if img[y][x] == 1:
                res[y][x] = nextLabel
                nextLabel += 1

    if verb:
        print('First labels')
        print(res)

    # Zig-Zag through the image
    while change:
        change = False
        cnt = cnt + 1
        for x in range(1, width):  # First row
            if res[0][x] != 0:
                if res[0][x - 1] < res[0][x] and res[0][x - 1] != 0:
                    res[0][x] = res[0][x - 1]
                    change = True
                if res[1][x] < res[0][x] and res[1][x] != 0:
                    res[0][x] = res[1][x]
                    change = True

        for y in range(1, height):  # Remaining rows
            if res[y][0] != 0:
                if res[y - 1][0] < res[y][0] and res[y - 1][0] != 0:
                    res[y][0] = res[y - 1][0]
                    change = True
                elif res[y + 1][0] < res[y][0] and res[y + 1][0] != 0:
                    res[y][0] = res[y + 1][0]
                    change = True
                elif res[y][1] < res[y][0] and res[y][1] != 0:
                    res[y][0] = res[y][1]
                    change = True
            for x in range(1, width):  # Remaining columns
                if res[y][x] != 0:
                    if res[y - 1][x] < res[y][x] and res[y - 1][x] != 0:
                        res[y][x] = res[y - 1][x]
                        change = True
                    elif res[y][x - 1] < res[y][x] and res[y][x - 1] != 0:
                        res[y][x] = res[y][x - 1]
                        change = True
                    if x != width - 1:
```

```
                        if res[y][x + 1] < res[y][x] and res[y][x + 1] != 0:
                            res[y][x] = res[y][x + 1]
                            change = True
                    if y != height - 1:
                        if res[y + 1][x] < res[y][x] and res[y + 1][x] != 0:
                            res[y][x] = res[y + 1][x]
                            change = True
        if verb:
            print('Current labels')
            print(res)

        if change:
            change = False
            cnt = cnt + 1
            for x in range(width - 1, 0, -1):  # Last row
                if res[height - 1][x] != 0:
                    if res[height - 1][x - 1] < res[height - 1][x] and
                        res[height - 1][x - 1] != 0:
                            res[height - 1][x] = res[height - 1][x - 1]
                                change = True

            for y in range(height - 1, 0, -1):  # Remaining rows
                if res[y][width - 1] != 0:
                    if y != 0:
                        if res[y - 1][width - 1] < res[y][width - 1] and
                            res[y - 1][width - 1] != 0:
                                res[y][width - 1] = res[y - 1][width - 1]
                                    change = True
                        elif res[y - 1][width - 2] < res[y][width - 1] and
                                res[y - 1][width - 2] != 0:
                                    res[y][width - 1] = res[y - 1][width - 2]
                                        change = True
                    if y != height - 1:
                        if res[y + 1][width - 1] < res[y][width - 1] and
                            res[y + 1][width - 1] != 0:
                                res[y][width - 1] = res[y + 1][width - 1]
                                    change = True
                for x in range(width - 1, 0, -1):  # Remaining columns
                    if res[y][x] != 0:
                        if res[y - 1][x] < res[y][x] and res[y - 1][x] != 0:
                            res[y][x] = res[y - 1][x]
                            change = True
                        elif res[y][x - 1] < res[y][x] and res[y][x - 1] != 0:
                            res[y][x] = res[y][x - 1]
                            change = True
                        if x != width - 1:
                            if res[y][x + 1] < res[y][x] and res[y][x + 1] != 0:
                                res[y][x] = res[y][x + 1]
                                change = True
                        if y != height - 1:
```

```
                            if res[y + 1][x] < res[y][x] and res[y + 1][x] != 0:
                                res[y][x] = res[y + 1][x]
                                change = True

            if verb:
                print('Current labels')
                print(res)

    for i in range(0, height):
        for j in range(0, width):
            if not (labels.__contains__(res[i][j])) and res[i][j] != 0:
                labels.append(res[i][j])

    if verb:
        print("Object labels: ", labels)

    color = int(255 / len(labels))

    for i in range(0, height):
        for j in range(0, width):
            if labels.__contains__(res[i][j]):
                res[i][j] = color * (labels.index(res[i][j]) + 1)

    img = res

    new_time = time.time()

    (...)

    return img
```

Listing 36: Binary Image Segmentation Iterative Algorithm.

```
def imgProjXYD(img, X=False, Y=False, D1=False, D2=False):
    '''
    Image Projections
    :param img: binary image
    :param X: True if Horizontal Projection
    :param Y: True if Vertical Projection
    :param D1: True if Diagonal Projection (45º)
    :param D2: True if Diagonal Projection (135º)
    :return:
    '''

    height = img.shape[0]
    width = img.shape[1]
    sum_y = []
    sum_x = []
    sum_d1 = []
```

```
    sum_d2 = []

    if Y:
        for i in range(0, height):
            cnt = 0
            for j in range(0, width):
                if img[i, j] == 1:
                    cnt = cnt + 1

            sum_x.append([i, cnt])

        y = []
        x = []
        for entry in sum_x:
            y.append(entry[1])
            x.append(entry[0])

        plt.plot(x, y)
        plt.yticks(y)
        plt.xticks(x)
        plt.xlabel("Position")
        plt.ylabel("Number of Pixels")
        plt.title("Vertical Projection")
        plt.show()

    if X:
        for i in range(0, width):
            cnt = 0
            for j in range(0, height):
                if img[j, i] == 1:
                    cnt = cnt + 1

            sum_y.append([i, cnt])

        y = []
        x = []
        for entry in sum_y:
            y.append(entry[1])
            x.append(entry[0])

        plt.plot(x, y)
        plt.yticks(y)
        plt.xticks(x)
        plt.xlabel("Position")
        plt.ylabel("Number of Pixels")
        plt.title("Horizontal Projection")
        plt.show()

    if D1:
        if img[0, 0] == 1:
```

```
            sum_d1.append([0, 1])
        else:
            sum_d1.append([0, 0])

        dif = 0
        for i in range(1, height + width - 1):
            y = i
            x = 0
            cnt = 0
            if i > height - 1:
                y = height - 1
                dif = dif + 1
                x = x + dif
            for j in range(0, i - dif * 2 + 1):
                if img[y, x] == 1:
                    cnt = cnt + 1
                y = y - 1
                x = x + 1

            sum_d1.append([i, cnt])

        y = []
        x = []
        for entry in sum_d1:
            y.append(entry[1])
            x.append(entry[0])

        plt.plot(x, y)
        plt.yticks(y)
        plt.xticks(x)
        plt.xlabel("Position")
        plt.ylabel("Number of Pixels")
        plt.title("Diagonal Projection (45°)")
        plt.show()

    if D2:
        if img[0, width - 1] == 1:
            sum_d2.append([0, 1])
        else:
            sum_d2.append([0, 0])

        dif = 0
        for i in range(1, height + width - 1):
            y = i
            x = width - 1
            cnt = 0
            if i > height - 1:
                y = height - 1
                dif = dif + 1
                x = x - dif
```

```
            for j in range(0, i - 2 * dif + 1):
                if img[y, x] == 1:
                    cnt = cnt + 1
                y = y - 1
                x = x - 1

            sum_d2.append([i, cnt])

        y = []
        x = []
        for entry in sum_d2:
            y.append(entry[1])
            x.append(entry[0])

        plt.plot(x, y)
        plt.yticks(y)
        plt.xticks(x)
        plt.xlabel("Position")
        plt.ylabel("Number of Pixels")
        plt.title("Diagonal Projection (135º)")
        plt.show()
```

Listing 37: Image Projections Method.

```
def regGro(img, seeds, t, sel=False, verb=False):
    '''
    Region Growing
    :param img: grayscale image
    :param seeds: seed points ([y,x])
    :param t: threshold for pixel acceptance
    :param sel: select 4 (False) or 8 (True) connected neighbourhood
    :param verb: True if messages expected
    :return: returns segmented image
    '''

    height = img.shape[0]
    width = img.shape[1]
    res = np.zeros((height, width), dtype=np.uint8)
    mark = 1

    def selectNeighbourhood(s):
        if s:
            n = [[-1, -1], [-1, 0], [-1, 1], [0, -1], [0, 1], [1, -1], [1, 0], [1, 1]]
        else:
            n = [[0, -1], [1, 0], [0, 1], [-1, 0]]

        return n

    neigh = selectNeighbourhood(sel)
```

```
26
27      if verb:
28          print(len(neigh), "-connected neighbourhood")
29
30      old_time = time.time()
31
32      while len(seeds) > 0:
33          currentSeed = seeds.pop(0)
34          if verb:
35              print("Current seed: ", currentSeed)
36
37          res[currentSeed[0], currentSeed[1]] = mark
38
39          for pos in neigh:
40              x = currentSeed[1] + pos[1]
41              y = currentSeed[0] + pos[0]
42
43              if x < 0 or y < 0 or x >= width or y >= height:
44                  continue
45
46              diff = abs(img[currentSeed[0], currentSeed[1]] - img[y, x])
47              if verb:
48                  print("Difference with neighbourhood [", y, x, "]: ", diff)
49
50              if diff < t and res[y, x] == 0:
51                  res[y, x] = mark
52                  seeds.append([y, x])
53          if verb:
54              print("")
55
56      new_time = time.time()
57
58      (...)
59
60      return res
```

Listing 38: Region Growing Algorithm.

```
1   def objCharact(img, sel=False, verb=False):
2       '''
3       Gets/Calculates object properties
4       :param img: binary image with a single object
5       :param sel: select 4 (False) or 8 (True) connected neighbourhood
6       :param verb: True if messages are expected
7       :return: list of object properties
8       '''
9
10      height = img.shape[0]
11      width = img.shape[1]
```

```
    cntP = 0
    cntA = 0
    stop = False
    last_x = -1
    last_y = -1
    polarCoord = []
    imgcpy = np.zeros((height, width), dtype=np.uint8)

    # Object Area
    for i in range(0, height):
        for j in range(0, width):
            if img[i, j] == 1:
                cntA = cntA + 1

    print("Area: ", cntA)

    ################

    # Object Centroid
    cntI = 0
    cntJ = 0
    cntII = 0
    cntJJ = 0
    cntIJ = 0

    for i in range(0, height):
        for j in range(0, width):
            if img[i, j] == 1:
                cntI = cntI + i
                cntJ = cntJ + j
                cntII = cntII + math.pow(i, 2)
                cntJJ = cntJJ + math.pow(j, 2)
                cntIJ = cntIJ + i * j

    cY = cntI / cntA
    cX = cntJ / cntA

    if verb:
        print("Object Centroid (True Coordinates): [", cY, ",", cX, "]")

    cY = round(cY)
    cX = round(cX)

    print("Object Centroid: [", cY, ",", cX, "]")

    ################

    # Object Perimeter
    def selectNeighbourhood(s):
        if s:
```

```
            n = [[0, 1], [1, 1], [1, 0], [1, -1], [0, -1], [-1, -1], [-1, 0], [-1, 1]]
        else:
            n = [[0, 1], [1, 0], [0, -1], [-1, 0]]

        return n

    def nextDir8(argument):
        switcher = {
            0: 5,
            1: 6,
            2: 7,
            3: 0,
            4: 1,
            5: 2,
            6: 3,
            7: 4,
        }
        return switcher.get(argument, "-1")

    def nextDir4(argument):
        switcher = {
            0: 3,
            1: 0,
            2: 1,
            3: 2,
        }
        return switcher.get(argument, "-1")

    neigh = selectNeighbourhood(sel)

    for i in range(0, height):
        if stop:
            break
        for j in range(0, width):
            if img[i, j] == 1:
                initial_y = i
                initial_x = j
                stop = True
                break

    curr_x = initial_x
    curr_y = initial_y

    if verb and sel:
        print("Selected neighbourhood 8")

    if verb and not sel:
        print("Selected neighbourhood 4")

    if verb:
```

```
112         print("\nCoordinates for border points\n----")
113 
114     for p in range(0, len(neigh)):
115         if img[curr_y + neigh[p][0], curr_x + neigh[p][1]] == 1:
116             if curr_y + neigh[p][0] == last_y and curr_x + neigh[p][1] == last_x:
117                 continue
118             else:
119                 initial_dir = p
120                 last_x = curr_x
121                 last_y = curr_y
122                 curr_y = curr_y + neigh[p][0]
123                 curr_x = curr_x + neigh[p][1]
124 
125                 y = curr_y - cY
126                 x = curr_x - cX
127                 polarCoord.append([math.sqrt(math.pow(y, 2) + math.pow(x, 2)),
128                                    [curr_y, curr_x]])
129 
130                 if verb:
131                     print(last_y, last_x, p)
132 
133                 break
134 
135     while stop:
136         if sel:
137             p = nextDir8(p)
138         else:
139             p = nextDir4(p)
140 
141         while True:
142             if img[curr_y + neigh[p][0], curr_x + neigh[p][1]] == 1:
143                 curr_dir = p
144                 last_x = curr_x
145                 last_y = curr_y
146                 curr_y = curr_y + neigh[p][0]
147                 curr_x = curr_x + neigh[p][1]
148 
149                 y = curr_y - cY
150                 x = curr_x - cX
151                 polarCoord.append([math.sqrt(math.pow(y, 2) + math.pow(x, 2)),
152                                    [curr_y, curr_x]])
153 
154                 if verb:
155                     print(last_y, last_x, p)
156 
157                 if sel:
158                     if p == 0 or p == 2 or p == 4 or p == 6:
159                         cntP = cntP + 1
160                     else:
161                         cntP = cntP + math.sqrt(2)
```

```
                else:
                    cntP = cntP + 1

                break

            p += 1
            if p > 7 and sel:
                p = 0
            if p > 3 and not sel:
                p = 0

        if last_x == initial_x and last_y == initial_y and curr_dir == initial_dir:
            stop = False

    if verb:
        print("----")

    print("Perimeter: ", cntP)

    ################

    # Object Form Factor
    ff = (4 * math.pi * cntA) / math.pow(cntP, 2)

    print("Form factor: ", ff)

    ################

    for i in range(0, height):
        if stop:
            break
        for j in range(0, width):
            if img[i, j] == 1:
                upperX = j
                upperY = i
                stop = True
                break

    if verb:
        print("Upper pixel coordinates: [", upperY, ",", upperX, "]")

    stop = False

    for i in range(height - 1, 0, -1):
        if stop:
            break
        for j in range(width - 1, 0, -1):
            if img[i, j] == 1:
                lowerX = j
                lowerY = i
```

```
                stop = True
                break

    if verb:
        print("Lower pixel coordinates: [", lowerY, ",", lowerX, "]")

    stop = False

    for i in range(width - 1, 0, -1):
        if stop:
            break
        for j in range(height - 1, 0, -1):
            if img[j, i] == 1:
                rightX = i
                rightY = j
                stop = True
                break

    if verb:
        print("Right pixel coordinates: [", rightY, ",", rightX, "]")

    stop = False

    for i in range(0, width):
        if stop:
            break
        for j in range(0, height):
            if img[j, i] == 1:
                leftX = i
                leftY = j
                stop = True
                break

    if verb:
        print("Left pixel coordinates: [", leftY, ",", leftX, "]")

    # Get Max Diameter
    if rightX - leftX < lowerX - upperX:
        w = abs(lowerX - upperX) + 1
        h = abs(lowerY - upperY) + 1
        mxD = math.sqrt(math.pow(h, 2) + math.pow(w, 2))
        if verb:
            print("Height: ", h, "\nWidth: ", w)
    else:
        w = abs(rightX - leftX) + 1
        h = abs(rightY - leftY) + 1
        mxD = math.sqrt(math.pow(h, 2) + math.pow(w, 2))
        if verb:
            print("Height: ", h, "\nWidth: ", w)

```

```
    print("Max Diameter: ", mxD)

    ################

    # Get Compactness
    cmp = (4 * math.pi * cntA)/math.pow(cntP, 2)
    cmp = math.pow(cntP, 2)/cntA

    print("Compactness: ", cmp)

    cmpN = 1 - 4 * math.pi / cmp
    if cmpN < 0:
        cmpN = 1 + cmpN

    print("Normalized compactness: ", cmpN)

    ################

    # Object Orientation
    mII = cntII - math.pow(cntI, 2) / cntA
    mJJ = cntJJ - math.pow(cntJ, 2) / cntA
    mIJ = cntIJ - (cntI * cntJ) / cntA
    if mIJ != 0:
        ori = math.atan((mII - mJJ + math.sqrt(math.pow((mII - mJJ), 2) +
                4 * math.pow(mIJ, 2))) / (2 * mIJ))
        ori = -ori * 180 / math.pi
    else:
        ori = '?'
        print("Cannot obtain object orientation.")

    if verb:
        print("Sx: ", cntI, "\nSy: ", cntJ, "\nSxx: ", cntII,
            "\nSyy: ", cntJJ, "\nSxy: ", cntIJ, "\nMxx: ", mII,
              "\nMyy: ", mJJ, "\nMxy: ", mIJ)
        print("Object Orientation: ", ori, "°")

    ################

    # Convex Perimeter

    minD = []
    maxD = []
    convexP = 0

    if polarCoord[0][0] > polarCoord[1][0] < polarCoord[2][0]:
        look = 'max'
        minD.append(polarCoord[1][1])
    elif polarCoord[0][0] < polarCoord[1][0] < polarCoord[2][0]:
        look = 'max'

```

```
312     if polarCoord[0][0] < polarCoord[1][0] > polarCoord[2][0]:
313         look = 'min'
314         maxD.append(polarCoord[1][1])
315     elif polarCoord[0][0] > polarCoord[1][0] > polarCoord[2][0]:
316         look = 'min'
317 
318     for i in range(3, len(polarCoord)):
319         if look == 'min':
320             if polarCoord[i-1][0] < polarCoord[i][0]:
321                 look = 'max'
322                 minD.append(polarCoord[i-1][1])
323         else:
324             if polarCoord[i - 1][0] > polarCoord[i][0]:
325                 look = 'min'
326                 maxD.append(polarCoord[i-1][1])
327 
328     for i in range(0, len(maxD)-1):
329         convexP = convexP + math.sqrt(math.pow(maxD[i][0] - maxD[i+1][0], 2) +
330                   math.pow(maxD[i][1] - maxD[i+1][1], 2))
331 
332     convexP = convexP + math.sqrt(math.pow(maxD[len(maxD)-1][0] - maxD[0][0], 2) +
333               math.pow(maxD[len(maxD)-1][1] - maxD[0][1], 2))
334 
335     if verb:
336         print('Distance to centroid: ', polarCoord)
337         print('Min pixels: ', minD)
338         print('Max pixels: ', maxD)
339 
340     print('Convex Perimeter: ', convexP)
341 
342     ################
343 
344     # Convex Area
345 
346     convexA = 0
347     end = True
348     convexBorder = []
349 
350     if maxD[0][1] - maxD[len(maxD) - 1][1] != 0:
351         if maxD[0][0] - maxD[len(maxD) - 1][0] != 0:
352             m = (maxD[len(maxD) - 1][0] - maxD[0][0]) /
353                 (maxD[len(maxD) - 1][1] - maxD[0][1])
354             numberX = abs(maxD[0][1] - maxD[len(maxD) - 1][1])
355             b = maxD[i]
356 
357             if m > 0 and maxD[len(maxD) - 1][0] > maxD[0][0]:
358                 varY = 1
359                 varX = 1
360             elif m < 0 and maxD[len(maxD) - 1][0] > maxD[0][0]:
361                 varY = 1
```

```
                varX = -1
                m = m * -1
            elif m < 0 and maxD[len(maxD) - 1][0] < maxD[0][0]:
                varY = -1
                varX = 1
                m = m * -1
            elif m > 0 and maxD[len(maxD) - 1][0] < maxD[0][0]:
                varY = -1
                varX = -1

            for j in range(1, numberX + 1):
                valueY0 = round(j * m * varY + b[0])
                valueY1 = round(j * m * varX + b[1])
                if verb:
                    print('Convex shape new pixels [y,x]: ', '[', valueY0, ']',
                        '[', valueY1, ']')
                imgcpy[valueY0][valueY1] = 1
                convexBorder.append([valueY0, valueY1])
        else:
            if maxD[0][1] > maxD[len(maxD) - 1][1]:
                a = maxD[len(maxD) - 1][1]
                while maxD[0][1] != a - 1:
                    if verb:
                        print('Convex shape new pixels [y,x]: ',
                            '[', maxD[0][0], ']', '[', a, ']')
                    imgcpy[maxD[0][0]][a] = 1
                    convexBorder.append([maxD[0][0], a])
                    a += 1
            elif maxD[0][1] < maxD[len(maxD) - 1][1]:
                a = maxD[0][1]
                while maxD[len(maxD) - 1][1] != a:
                    if verb:
                        print('Convex shape new pixels [y,x]: ',
                            '[', maxD[0][0], ']', '[', a, ']')
                    imgcpy[maxD[0][0]][a] = 1
                    convexBorder.append([maxD[0][0], a])
                    a += 1
    else:
        if maxD[0][0] > maxD[len(maxD) - 1][0]:
            a = maxD[0][0]
            while maxD[len(maxD) - 1][0] != a - 1:
                a -= 1
                imgcpy[a][maxD[0][1]] = 1
                convexBorder.append([a, maxD[0][1]])
                if verb:
                    print('Convex shape new pixels [y,x]: ', '[', a, ']',
                        '[', maxD[0][1], ']')
        elif maxD[0][0] < maxD[len(maxD) - 1][0]:
            a = maxD[0][0]
            while maxD[len(maxD) - 1][0] != a + 1:
```

```
412                 a += 1
413                 imgcpy[a][maxD[0][1]] = 1
414                 convexBorder.append([a, maxD[0][1]])
415                 if verb:
416                     print('Convex shape new pixels [y,x]: ', '[', a, ']',
417                         '[', maxD[0][1], ']')
418 
419     for i in range(0, len(maxD)-1):
420         if maxD[i][1] - maxD[i + 1][1] != 0:
421             if maxD[i][0] - maxD[i+1][0] != 0:
422                 m = (maxD[i+1][0] - maxD[i][0])/(maxD[i+1][1] - maxD[i][1])
423                 numberX = abs(maxD[i][1] - maxD[i + 1][1])
424                 b = maxD[i]
425 
426                 if m > 0 and maxD[i + 1][0] > maxD[i][0]:
427                     varY = 1
428                     varX = 1
429                 elif m < 0 and maxD[i + 1][0] > maxD[i][0]:
430                     varY = 1
431                     varX = -1
432                     m = m * -1
433                 elif m < 0 and maxD[i + 1][0] < maxD[i][0]:
434                     varY = -1
435                     varX = 1
436                     m = m * -1
437                 elif m > 0 and maxD[i + 1][0] < maxD[i][0]:
438                     varY = -1
439                     varX = -1
440 
441                 for j in range(1, numberX+1):
442                     valueY0 = round(j * m * varY + b[0])
443                     valueY1 = round(j * m * varX + b[1])
444                     if verb:
445                         print('Convex shape new pixels [y,x]: ',
446                             '[', valueY0, ']', '[', valueY1, ']')
447                     imgcpy[valueY0][valueY1] = 1
448                     convexBorder.append([valueY0, valueY1])
449             else:
450                 if maxD[i][1] > maxD[i + 1][1]:
451                     a = maxD[i][1]
452                     while maxD[i + 1][1] != a + 1:
453                         if verb:
454                             print('Convex shape new pixels [y,x]: ',
455                                 '[', maxD[i][0], ']', '[', a, ']')
456                         imgcpy[maxD[i][0]][a] = 1
457                         convexBorder.append([maxD[i][0], a])
458                         a -= 1
459                 elif maxD[i][1] < maxD[i + 1][1]:
460                     a = maxD[i][1]
461                     while maxD[i + 1][1] != a:
```

```
                        a += 1
                        if verb:
                            print('Convex shape new pixels [y,x]: ',
                                '[', maxD[i][0], ']', '[', a, ']')
                        imgcpy[maxD[i][0]][a] = 1
                        convexBorder.append([maxD[i][0], a])

            else:
                if maxD[i][0] > maxD[i + 1][0]:
                    a = maxD[i][0]
                    while maxD[i + 1][0] != a:
                        a -= 1
                        imgcpy[a][maxD[i][1]] = 1
                        convexBorder.append([a, maxD[i][1]])
                        if verb:
                            print('Convex shape new pixels [y,x]: ', '[', a, ']',
                                '[', maxD[i][1], ']')
                elif maxD[i][0] < maxD[i + 1][0]:
                    a = maxD[i][0]
                    while maxD[i+1][0] != a:
                        a += 1
                        imgcpy[a][maxD[i][1]] = 1
                        convexBorder.append([a, maxD[i][1]])
                        if verb:
                            print('Convex shape new pixels [y,x]: ', '[', a, ']',
                                '[', maxD[i][1], ']')

        if verb:
            print('Convex border:')
            print(imgcpy)
            print('Convex border positions: ', convexBorder)

        for i in range(0, len(convexBorder)-2):
            convexA += convexBorder[i][1] * convexBorder[i + 1][0]
            convexA -= convexBorder[i + 1][1] * convexBorder[i][0]

        convexA += convexBorder[len(convexBorder)-1][1] * convexBorder[0][0]
        convexA -= convexBorder[0][1] * convexBorder[len(convexBorder)-1][0]

        convexA /= 2

        print('Convex Area: ', convexA)

        ################

        # Get Convexity
        cov = convexP/cntP

        print("Convexity: ", cov)

```

```
512     ################
513
514     # Get Solidity
515     if convexA != 0:
516         sd = cntA/convexA
517         print("Solidity: ", sd)
518
519     ################
520
521     # Get Circularity
522     cir = (4 * math.pi * cntA) / math.pow(convexP, 2)
523
524     print("Circularity: ", cir)
525
526     ################
```

Listing 39: Object Feature Extraction Methods.