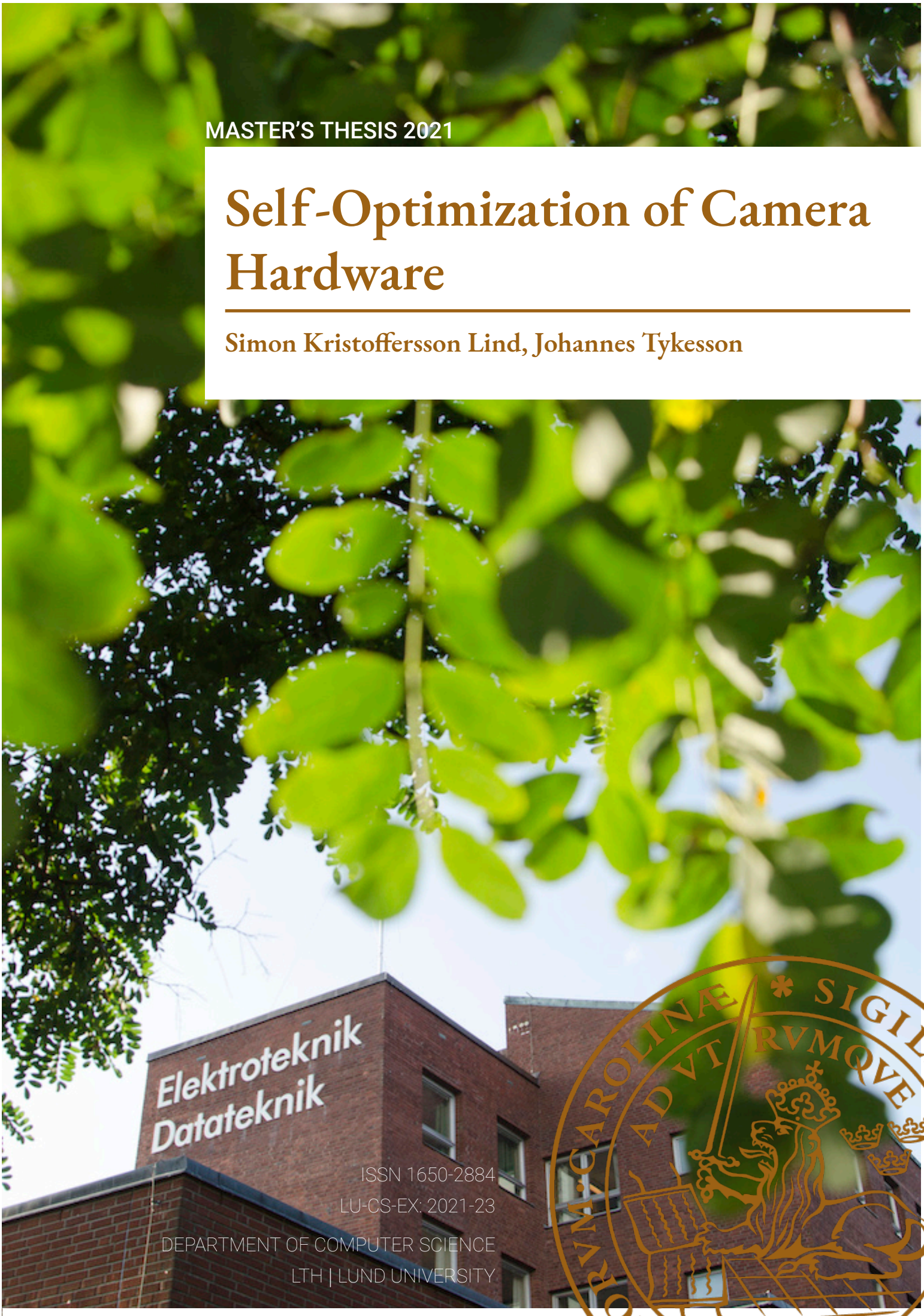


MASTER'S THESIS 2021

Self-Optimization of Camera Hardware

Simon Kristoffersson Lind, Johannes Tykesson

ISSN 1650-2884
LU-CS-EX: 2021-23

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-23

Self-Optimization of Camera Hardware

Automatisk Optimering för Kamerahårdvara

Simon Kristoffersson Lind, Johannes Tykesson

Self-Optimization of Camera Hardware

Simon Kristoffersson Lind
si8270an-s@student.lu.se

Johannes Tykesson
jo6761ty-s@student.lu.se

June 24, 2021

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Luigi Nardi, luigi.nardi@cs.lth.se
Waqar Hameed, waqarh@axis.com

Examiner: Volker Krueger, volker.krueger@cs.lth.se

Abstract

This thesis aims to investigate the automatic tuning of hardware parameters in a camera's image processing pipeline. In order to solve the tuning problem, it is formulated as a black-box optimization problem centered around a physical camera unit. Optimization is performed by comparing the camera's output to a reference image. Several black-box optimization algorithms were tested: Bayesian Optimization, Evolutionary Optimization, Particle Swarm Optimization, Simulated Annealing, DIRECT, and Rowan's Subplex Method. Results indicate that it is feasible to automatically tune camera hardware parameters using black-box optimization algorithms. For 14 parameters, Rowan's Subplex Method performs best with an average error of 6.25. When optimizing a much larger set of 71 parameters, Simulated Annealing, Evolutionary, and Rowan's Subplex Method perform best with an average error of 9.77, 17.92, and 18.05 respectively.

Keywords: Optimization, Black-box optimization, Evolutionary, Bayesian, Simulated Annealing, DIRECT, Particle Swarm, Simplex

Acknowledgements

We would like to offer our thanks to our supervisors, Luigi Nardi and Waqar Hameed, for their guidance throughout this work.

Our gratitude is also extended to Axis' imaging engineers: Jimmie Jönsson, Gunnar Dahlgren, Philip Siederer, Wei Wen, and William Chaze, for providing us with much needed practical knowledge.

A final thanks to our hiring manager at Axis, Sabina Ahlberg, for making this thesis possible.

Contents

1 Introduction **7**
- 1.1 Contributions . . . 8
- 1.2 Related work . . . 8
- 1.3 Outline . . . 9

2 Background **11**
- 2.1 Axis' Cameras . . . 11
  - 2.1.1 Camera Hardware Parameters . . . 11
- 2.2 Theory for Comparison . . . 13
  - 2.2.1 RMSE . . . 13
- 2.3 Theory for Images . . . 13
  - 2.3.1 OpenCV . . . 13
  - 2.3.2 YCbCr Format . . . 13
  - 2.3.3 Image Derivatives and Sobel Filters . . . 14
  - 2.3.4 Gradient Histograms . . . 14
  - 2.3.5 Image Histograms . . . 15
  - 2.3.6 Feature Detection . . . 15
  - 2.3.7 Image Alignment . . . 16
- 2.4 Optimization theory . . . 17
  - 2.4.1 Black-Box Optimization . . . 17
  - 2.4.2 Evolutionary Optimization . . . 18
  - 2.4.3 Bayesian Optimization . . . 18
  - 2.4.4 Particle Swarm Optimization . . . 21
  - 2.4.5 Simulated Annealing . . . 22
  - 2.4.6 Nelder Mead Simplex . . . 24
  - 2.4.7 Rowan's Subplex Method . . . 25
  - 2.4.8 DIRECT . . . 27

3 Methodology **29**
- 3.1 Black Box Architecture . . . 29

3.1.1 Image Injection . . . 29
3.1.2 Setting Parameters in the Camera . . . 30
3.1.3 Requesting an Image from the Camera . . . 30
3.1.4 Image Comparison . . . 31
3.1.5 Highlighting Differences in Images . . . 32
3.1.6 Dataset and Reference Images . . . 32
3.2 Random Sampling . . . 33
3.3 Optimization Algorithms . . . 33
3.3.1 Evolutionary Optimization . . . 34
3.3.2 Bayesian Optimization . . . 34
3.3.3 Particle Swarm Optimization . . . 35
3.3.4 Simulated Annealing . . . 35
3.3.5 Rowan's Subplex Method . . . 35
3.3.6 DIRECT . . . 35
3.3.7 Robustness of Optimizer Implementations . . . 36

4 Experiments **39**
4.1 Optimizing 14 Parameters . . . 39
4.1.1 Methodology . . . 39
4.1.2 Results . . . 40
4.1.3 Discussion . . . 45
4.2 Optimizing 71 Parameters . . . 46
4.2.1 Methodology . . . 46
4.2.2 Results . . . 46
4.2.3 Discussion . . . 52

5 Conclusion **55**
5.1 Choice of Algorithm . . . 55
5.2 Future work . . . 55

References **57**

Appendix A Individual plots with standard deviation **63**

Appendix B Popular Science Article **67**

Chapter 1
Introduction

This thesis is done at Axis Communications, informally known as just Axis. Axis was founded in Lund, Sweden, in 1984, and its main products are network cameras. In 2019, Axis had just over 3600 employees in 50 countries [1].

When a consumer buys a camera, they can generally start using their camera immediately, and get nice images without having to tweak any settings. However, for the engineers working with cameras, it is an entirely different story. Modern cameras at Axis are complex pieces of hardware and include several components that transform the image on its journey from the sensor to a display. Each of these components contains its own knobs and dials that can be tweaked in order to change different qualities in the image.

Naturally, there are no actual knobs or dials in the camera components, only parameters in the form of binary numbers. Imaging engineers are tasked with adjusting all these numbers to produce a good image quality for the consumer. This process has to be repeated every time new camera hardware is created.

Interviews with imaging engineers at Axis reveal that each camera hardware generally has over 100 individual parameters that affect image quality and that tuning all of them can take a team of engineers several weeks.

In order to alleviate the tedious tuning process for the image engineers, this thesis aims to develop a method for automatically tuning the camera hardware parameters.

Different parameters control different parts of the image quality on the camera and significantly impact how the image output from the camera looks. Examples of the difference the parameters can make are shown in figure 1.1. In Fig. 1.1, the rightmost image is taken with parameters that are manually tuned by expert imaging engineers. Two random configurations are shown next to the hand-tuned image. Both random configurations have very poor color values, which suggests that many color-related parameters are different from the hand-tuned. It can also be seen that the leftmost image has much sharper lines and more noise, which suggests that

the sharpness and contrast parameters are wrong. Most notable in the middle image is that it is very dark, which suggests that the brighness parameters are wrong.

Figure 1.1: Pictures of the same scene with different parameters. Rightmost image is taken with parameters that were hand-tuned by imaging engineers.

Since it can be assumed that a company such as Axis has many cameras that have already been manually tuned, this thesis will assume that it is feasible to produce a good image that can be used as a reference.

Given said reference image, a black-box optimization problem will be constructed in an attempt to automatically tune parameters in a camera hardware. Camera parameters will form the input to a black-box function. The output of the black-box function will be constructed by comparing an image from the camera to a reference image. Several methods for optimizing the black-box function will be tested to investigate whether this is a feasible approach or not.

An important thing to note is that this thesis does not aim to *improve* the image quality with respect to the reference image, only replicate the *same* image quality. Human imaging engineers are still needed in order to reach any improvement.

1.1 Contributions

Primary contributions of this thesis are:

- Formulation of camera harware parameter tuning as a black-box optimization problem.
- Development of an optimization framework capable of optimizing camera hardware parameters.

1.2 Related work

A similar approach to the one in this thesis is presented by Mosleh, Sharma, Onzon, Mannan, Robidoux, and Heide in their 2020 paper [19]. Mosleh et al. set up a hardware-in-the-loop framework and optimize camera parameters. The optimization is performed by using CMA-ES and search-space reduction. While their approach is similar to the one presented in this

thesis, this work extends the optimization problem by testing several other optimizers and by optimizing larger problems with more parameters.

1.3 Outline

Chapter 2 will introduce the problem in more detail and provide the theoretical background needed to understand this thesis. Both background for image comparison as well as theory for black-box problems and optimization algorithms will be presented. In chapter 3, details are given surrounding the software architecture used in experiments. Experiments are described in chapter 4, including methodology, results, and discussion. Finally chapter 5 summarizes conclusions drawn from the experiments.

Chapter 2
Background

2.1 Axis' Cameras

Disclaimer: This section will give a brief insight into Axis' cameras in order to provide context for the rest of this thesis. As such, this section will contain few references and will be intentionally vague to avoid disclosing any confidential information.

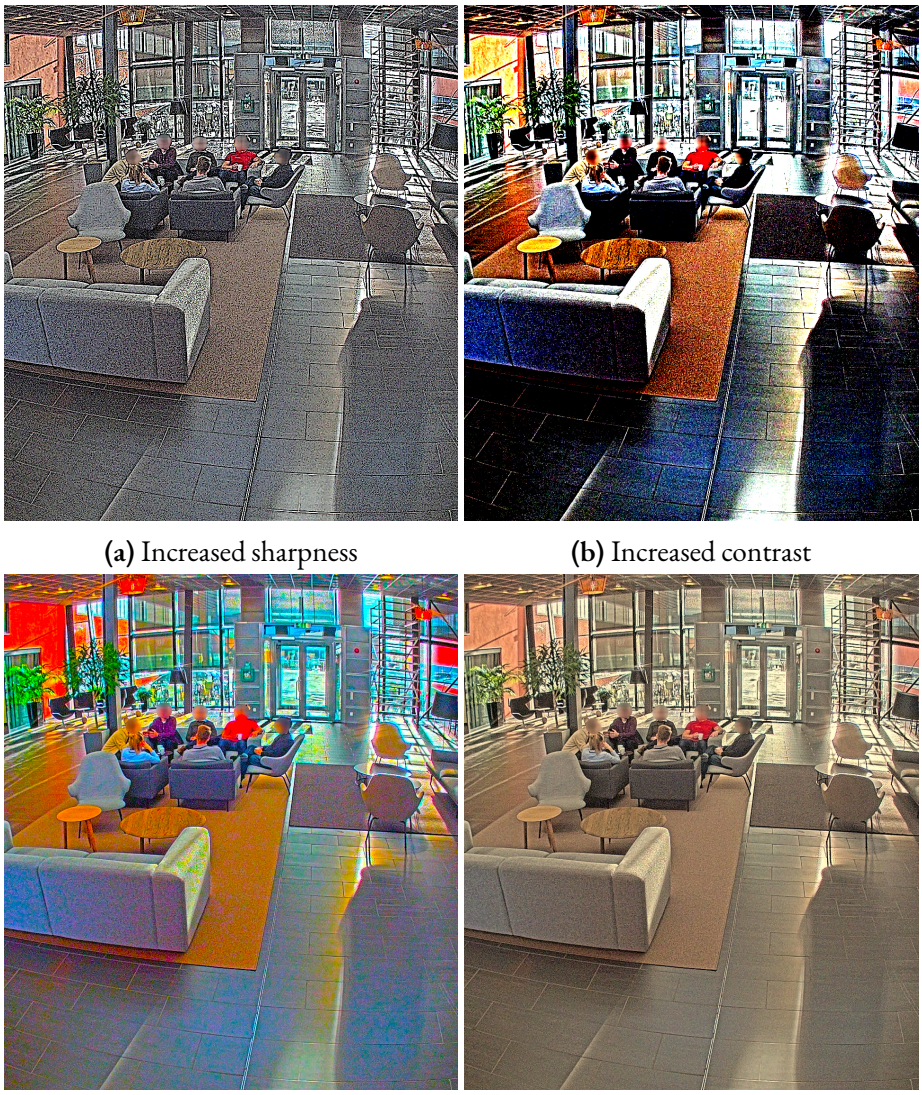
What all Axis' cameras have in common is that they are network cameras, which essentially means that they are controlled through a network interface [2]. Most cameras also share a common hardware platform developed by Axis. Said platform includes the image processing pipeline (IPP), which contains the parameters this thesis intends to tune. Thus, results from this thesis will generally apply to most Axis cameras. A shared hardware platform also gives the benefit that parameters only have to be tuned whenever the hardware platform is changed.

2.1.1 Camera Hardware Parameters

Most commercial cameras include a small number of comprehensive parameters for tweaking the look and quality of an image, for example "Sharpness", "Contrast", "Saturation", "Color Tone" as seen in a Canon camera [3]. Examples of the effects of such parameters are displayed in fig. 2.1.

However, what is not apparent to most consumers is that these parameters actually control several underlying hardware parameters directly in the IPP. Thus, these comprehensive high-level parameters give a good idea of the types of effects the underlying hardware parameters can have on the resulting image.

Hand-tuning parameters are generally done as illustrated in fig. 2.2, where a human imaging engineer has a camera connected to a display and tweaks parameters while seeing them updated on the display in real-time.



(a) Increased sharpness **(b)** Increased contrast

(c) Increased saturation **(d)** Changed color tone

Figure 2.1: Effects of sharpness, contrast, saturation and color tone

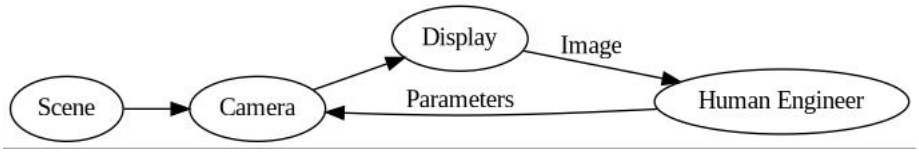


Figure 2.2: Graph of the process of hand-tuning parameters

2.2 Theory for Comparison

For the first part of this thesis, a method will be constructed for comparing two images. Throughout this thesis, MSE and RMSE will be extensively used as comparison metrics, so they are briefly defined here. Definitions below assume a sequence of elements to be compared. Such a sequence can be trivially constructed from an image by simply concatenating all its pixel-values, or by constructing a histogram.

2.2.1 RMSE

A well-established method for comparing the distance between two n-dimensional vectors/sequences x, y is the *Mean Squared Error* (MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 \tag{2.1}$$

Since the MSE can be difficult to interpret (due to the square), the *Root Mean Squared Error* (RMSE) is often used instead:

$$RMSE = \sqrt{MSE} \tag{2.2}$$

Thanks to the square root, RMSE is back to the same unit as the original vectors, which is often easier to interpret. There is also a similarity to the Euclidean distance between the two sequences [5].

2.3 Theory for Images

Determining if an image is of good or bad quality will, in this thesis, be reduced to a comparison with a reference image. In this section, a general theory is presented that will be used to construct methods for reliably comparing two images.

2.3.1 OpenCV

One tool that will be used frequently throughout this thesis is OpenCV. OpenCV is an open-source library for computer vision algorithms available for many programming languages, e.g. C/C++ and Python [22].

2.3.2 YCbCr Format

Many video sources, including the cameras used in this thesis, encode their video data in the YCbCr color space, specifically the NV12 format [12]. Similar to the otherwise common RGB format, the YCbCr format consists of 3 components: Y (luma), Cb (chroma blue), and Cr (chroma red). Luma refers to the brightness of the image. Chroma blue and chroma red refer to the blue and red color components in the image. YCbCr is a so-called YUV color space, which is a broader term for this type of color space which refers to the components as Y, U, V. YUV color spaces originate from the days when televisions were starting to transition to color video.

Thus, the reasoning behind a luma/color split is that gray-scale televisions could still function with only the Y channel, and the more modern color televisions could do the extra work of adding color from the Cb and Cr channels as well [12].

Often, the YUV channels are subsampled, which means that one color space value may contribute to the color of several pixels. In the NV12 format, each pixel corresponds to exactly one Y value, but a 2x2 window of pixels share their Cb and Cr values. As a result of pixels sharing Cb and Cr values, the image is stored in half as many bytes as the ordinary RGB format [12].

2.3.3 Image Derivatives and Sobel Filters

To measure similarity between images, the derivatives of the images can be used since structurally similar images will have similar derivates in the various regions of the image. To determine the derivates of an image, the gradient vector in each pixel will have to be computed. The gradient vector for a pixel is determined by computing the partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, where x is the coordinate of a pixel along the horizontal axis and y is the coordinate along the vertical axis. Togheter these partial derivatives form the gradient vector [10]:

$$\nabla f = \begin{bmatrix} g_x \\ g_y \end{bmatrix} \tag{2.3}$$

The most straightforward way to compute the gradient vector at a given position (x, y) is to compute

$$\nabla f(x, y) = \begin{bmatrix} f(x+1, y) - f(x, y) \\ f(x, y+1) - f(x, y) \end{bmatrix} \tag{2.4}$$

However, this computation does not handle diagonal directions by definition and does not contain much information regarding the direction other than x and y. Therefore a Sobel filter or Sobel operator can be more useful when calculating the partial derivatives of pixels. The Sobel operator uses convolution with the kernels shown below to compute the partial derivative in the x and y-direction, respectively. The kernel uses a 3x3 pixel area, and the resulting gradient vector represents the center pixel.

$$\begin{aligned} \frac{\delta f}{\delta x} &= f * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\ \frac{\delta f}{\delta y} &= f * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \end{aligned} \tag{2.5}$$

The advantage of using this operator is that the Sobel operator creates image smoothing, which is helpful when computing derivatives [10].

2.3.4 Gradient Histograms

To match similar pictures to each other, not based on the color of the images, gradient histograms can be used. Two different properties of the gradient for a pixel can be used, the magnitude and the angle of the gradient. The idea is to compute the gradient of each pixel in the picture by calculating the difference in respect to the x and y-direction, and then use the magnitude

and angle to produce histograms. The histograms will be divided into several bins based on the magnitude and angle of the gradient. Similar pictures should therefore have similar histograms. The gradient is computed by applying a Sobel filter to each pixel in the x and y-direction, respectively [10].

Magnitude

The magnitude of a the gradient for a given pixel is computed by [10]:

$$mag = \sqrt{g_x^2 + g_y^2} \tag{2.6}$$

Direction

The direction of the gradient for a given pixel is computed as an angle θ by [10]:

$$\theta = tan^{-1}\left(\frac{g_y}{g_x}\right) \tag{2.7}$$

2.3.5 Image Histograms

The histogram of a given image is computed by using the Y, Cb, Cr components (See section 2.3.2) of a picture and divide them into bins based on the value of the component at every pixel of the image. For an 8-bit image, 256 bins are used to divide the values of said component [30]. The histogram of an image can then be compared against other images in order to detect color changes. Comparison can be done with, for example, the RMSE distance metric. Intuitively, image histograms are good for detecting differences in color, for example a ligher or darker image. Though, image histograms might not be suitable for detecting changes in contrast or sharpness.

2.3.6 Feature Detection

A common strategy when manipulating images is to extract features [30]. Features are most commonly used in the context of *feature matching*, which is when features in two images are paired to find objects or patterns present in both images. Due to this common task of matching features, the features shoud be invariant to various image transformations. For example, if an image is rotated, it is desirable to find the same features as in the original [30].

Usually, the task of matching features in images is split into two sub-tasks: feature detection and feature description [30].

ORB

Oriented FAST and Rotated BRIEF (ORB) is a combined feature detector and feature descriptor. ORB builds on the FAST [25] feature detector by adding an orientation to the detected features. In order to perform rotationally invariant feature matching, a rotation component is added to the BRIEF [8] feature descriptor [27].

FAST is a feature detector with speed in mind. Specifically, it is a corner detector, which detects corners by comparing 16 pixels in a circle around a target center pixel. If there is a clear split such that a few of the 16 pixels are darker or lighter than the rest, the center point is considered a corner. To find corners faster, FAST employs machine learning to classify the 16 pixels instead of checking all the 16 pixels manually [25].

BRIEF is a feature descriptor based on comparing pixels in a region around a detected feature. Given, for example, a feature point from FAST, BRIEF will compare a number of pixel pairs and encode each comparison as a bit-vector based on which pixel is brighter [8].

2.3.7 Image Alignment

In order to compare two different images of the same scene that may not be pixel-exact, the images can be aligned based on features from a feature detector. Given feature matches between the two images, a system of linear equations can be solved to find a transformation matrix that maps coordinates from one image to the other [30].

Formally, given a set of matched feature points $\{(x, y), (x', y')\}$ an *affine transformation matrix* is a matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & t_1 \\ a_{21} & a_{22} & t_2 \end{bmatrix} \tag{2.8}$$

such that [30]

$$A \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} \tag{2.9}$$

Thus, the task at hand is to solve for A.

Since there is uncertainty when matching features across two images, there may be outliers that are not actually correct matches. In the presence of outliers, RANSAC is a common algorithm for solving the transformation matrix [30].

RANSAC, or RANdom SAmple Consensus, is an algorithm that randomly samples a minimal subset of data required to solve the equation at hand, and then counts inliers and outliers. After sampling many minimal subsets, the solution with the fewest outliers is chosen [30].

In the case of image alignment, since there are six unknown variables in the matrix A, six equations are needed. Each matched feature gives two equations:

$$\begin{cases} a_{11}x + a_{12}y + t_1 = x' \\ a_{21}x + a_{22}y + t_2 = y' \end{cases} \tag{2.10}$$

Thus *three* matched features is the minimum to solve for A.

Given three feature matches $\{(x_1, y_1), (x'_1, y'_1)\}, \{(x_2, y_2), (x'_2, y'_2)\}, \{(x_3, y_3), (x'_3, y'_3)\}$,

a system of equations can be used to solve A entirely:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ t_1 \\ a_{21} \\ a_{22} \\ t_2 \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{bmatrix} \tag{2.11}$$

With the transformation matrix A solved, all the other feature matches are classified as inliers or outliers based on the accuracy of the predicted points. Formally, given a matched feature $\{(x_i, y_i), (x'_i, y'_i)\}$:

$$\begin{aligned} &\mathbf{a} = \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} x'_i \\ y'_i \end{bmatrix} \\ &\mathbf{b}_{pred} = A\mathbf{a} \\ &\begin{cases} \|\mathbf{b}_{pred} - \mathbf{b}\| \leq \epsilon \implies \text{inlier} \\ \|\mathbf{b}_{pred} - \mathbf{b}\| > \epsilon \implies \text{outlier} \end{cases} \end{aligned} \tag{2.12}$$

where ϵ is a pre-defined outlier boundary.

From there, the matrix A with the fewest outliers is chosen.

2.4 Optimization theory

The parameter tuning problem will be tackled as a black-box optimization problem. Therefore, this section gives a formal introduction to black-box optimization, along with several algorithms for solving black-box optimization problems.

2.4.1 Black-Box Optimization

A black-box is a process that receives an input and generates an output while the actual process is hidden. The process of optimizing an objective that is the result of a black-box process is simply referred to as black-box optimization. Due to the unknown nature of a black-box, a black-box optimization procedure has to optimize the objective based on just the input and the output and cannot assume any knowledge about the actual process in the black-box [4].

Formally, black-box optimization aims to solve:

$$\underset{x}{\operatorname{argmin}} f(x) \quad \text{or} \quad \underset{x}{\operatorname{argmax}} f(x) \quad . \tag{2.13}$$

However, only one of these need to be considered since

$$\underset{x}{\operatorname{argmin}} f(x) = \underset{x}{\operatorname{argmax}} \Big[-f(x) \Big] \quad . \tag{2.14}$$

Since $f(x)$ is analytically unknown, no assumptions about continuity, differentiability, or smoothness can be made to ease the process of optimization [4].

One issue that arises when performing optimization of any kind is when to stop. It is common to stop when little to no improvement is made in terms of the function value, referred to as *Generation Stalling*. Another common strategy is an *Evaluation Budget*, where a maximum number of function evaluations is set. When said number of evaluations have been made, the algorithm is terminated and the best value found within the budget is chosen as the solution [4]. Throughout this thesis, an evaluation budget will be used as a default stopping criterion.

Camera Parameters as a Black-Box Problem

Optimizing camera parameters with respect to an image quality output can intuitively be thought of as a black-box optimization problem. Camera parameters act as input to the black-box, and the image quality measure acts as output. Since the problem formulation assumes no knowledge of the camera hardware pipeline, it acts as the black-box in this formulation.

2.4.2 Evolutionary Optimization

Evolutionary algorithms are algorithms based on the idea of evolution and natural selection [15]. The algorithms prioritize strong solutions that produce an output with good results and sorts out weak ones with bad results. Therefore, the parameters used in a strong solution are used to create new better solutions, while the parameters producing weak solutions are discarded. [4].

Given a function to optimize, $f(x)$, an evolutionary algorithm starts by sampling a *population* P_0 of candidate points $\mathbf{x}_i \in P_0$. The term *generation* is used to refer to new populations over time. In other words, for a set of populations P_i, each P_i is called a generation. Each of the points in P_i are given a *fitness* score. In every iteration of the algorithm, points are added to the next generation P_{i+1} either by keeping a point in the current generation or by selecting two *parents* from the current generation and creating an *offspring* from those.

Selection of parents is usually based on the *fitness measure* in order to select parents that are likely to produce good offspring. One such selection method is *tournament selection*. Tournament selection starts by sampling n random parents from the current population and comparing these parents against each other. The parent with the best fitness is returned and therefore used when creating the offspring [4].

Another frequently used selection method is *elitism selection*, which selects the n parents with the best fitness. *Roulette wheel selection* is a third alternative that picks n random parents from the population and uses these to create the new population [4].

Creation of new offspring can be done by using several methods, one of which is *crossover and mutation*. *Crossover* uses common elements from strong parents to create new offspring, which hopefully is better. Offspring created from previous parents is then slightly *mutated* with a given *mutation probability* in order to create a new set of parameters [4].

Basic pseudo-code is found in Algorithm 1 below.

2.4.3 Bayesian Optimization

Bayesian optimization is a machine-learning-based optimization method [9]. Similar to evolutionary optimization, Bayesian optimization keeps a dataset of points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ with

Algorithm 1 Evolutionary Optimization pseudo-code

Use random sampling to create initial population X of size n
Evaluate fitness for initial population $\mathbf{y}_i = f(\mathbf{x}_i) \; \forall \, \mathbf{x}_i \in X$
while stopping criterion is not met **do**
    Select 2 candidates $\mathbf{x}_1$, $\mathbf{x}_2$ from the population to create offspring $\mathbf{o}$
    for each parameter o_i in offspring $\mathbf{o}$ **do** ▷ Crossover
        Select o_i from either $\mathbf{x}_1$ or $\mathbf{x}_2$
    end for
    for each parameter o_i in offspring $\mathbf{o}$ **do**
        if random(0,1) < p **then** ▷ Mutate with probability p
            $o_i \leftarrow$ random(lower_bound, upper_bound)
        end if
    end for
    Add $\mathbf{o}$ to population
    Remove oldest or worst parent in population
end while

known function values $f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_n)$ and uses those to make a decision about which point to evaluate next. Instead of simply modifying a known point, Bayesian optimization works by fitting a statistical model to the dataset. That statistical model is then optimized over an *acquisition function* in order to find a new point $\mathbf{x}_{n+1}$ to evaluate [9].

In pseudo-code, Bayesian optimization looks very simple (algorithm 2).

Algorithm 2 Bayesian optimization pseudo-code

Pick t initial points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$
Evaluate $f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_t)$
while stopping criterion not met **do**
    Fit model $\mathbf{M}$ to dataset $\{(\mathbf{x}_1, f(\mathbf{x}_1)), (\mathbf{x}_2, f(\mathbf{x}_2)), \ldots, (\mathbf{x}_t, f(\mathbf{x}_t))\}$
    $\mathbf{x}_{t+1} \leftarrow \operatorname{argmax}_{\mathbf{x}} \mathit{acquisition}(\mathbf{x}; \mathbf{M})$
    add $(\mathbf{x}_{t+1}, f(\mathbf{x}_{t+1}))$ to dataset
    $t \leftarrow t + 1$
end while
return $\operatorname{argmin}_{\mathbf{x} \in \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t\}} f(\mathbf{x})$

Expected improvement (EI), is the most common choice of aquisition function [9]:

$$EI(x) = \mathbb{E}\big(f^* - \hat{f}(x)\big)^+ \tag{2.15}$$

Where f^* is the best known function value so far, and $\hat{f}(x)$ comes from the posterior probability distribution given by the underlying statistical model $\mathtt{M}$. Note the superscript $^+$, which effectively means that negative improvement is considered the same as no improvement.

Gaussian Process

In its base form, the statistical model used in Bayesian optimization is a Gaussian process, which provides a Bayesian posterior probability distribution for previously unknown points [9]:

$$f(\mathbf{x}) \sim \text{Normal}(\mu(\mathbf{x}), \sigma^2(\mathbf{x})) \tag{2.16}$$

where, μ and σ^2 are the mean and variance respectively.

For simplicity, a shorthand notation is used in following formulas:

$$\begin{aligned} f(\mathbf{x}_{1:n}) &= \begin{bmatrix} f(\mathbf{x}_1) & f(\mathbf{x}_2) & \dots & f(\mathbf{x}_n) \end{bmatrix}^T \\ \mu_0(\mathbf{x}_{1:n}) &= \begin{bmatrix} \mu_0(\mathbf{x}_1) & \mu_0(\mathbf{x}_2) & \dots & \mu_0(\mathbf{x}_n) \end{bmatrix}^T \\ \Sigma_0(\mathbf{x}_{1:n}, \mathbf{x}_{1:n}) &= \begin{bmatrix} \Sigma_0(\mathbf{x}_1, \mathbf{x}_1) & \Sigma_0(\mathbf{x}_1, \mathbf{x}_2) & \dots & \Sigma_0(\mathbf{x}_1, \mathbf{x}_n) \\ \Sigma_0(\mathbf{x}_2, \mathbf{x}_1) & \Sigma_0(\mathbf{x}_2, \mathbf{x}_2) & \dots & \Sigma_0(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_0(\mathbf{x}_n, \mathbf{x}_1) & \Sigma_0(\mathbf{x}_n, \mathbf{x}_2) & \dots & \Sigma_0(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix} \end{aligned} \tag{2.17}$$

With a Gaussian process and a dataset $\{(\mathbf{x}_1, f(\mathbf{x}_1)), (\mathbf{x}_2, f(\mathbf{x}_2)), \dots, (\mathbf{x}_n, f(\mathbf{x}_n))\}$, the mean and variance are modeled as such:

$$\begin{aligned} \mu(\mathbf{x}) &= \Sigma_0(\mathbf{x}, \mathbf{x}_{1:n})\Sigma_0(\mathbf{x}_{1:n}, \mathbf{x}_{1:n})^{-1}(f(\mathbf{x}_{1:n}) - \mu_0(\mathbf{x}_{1:n})) + \mu_0(\mathbf{x}) \\ \sigma^2(\mathbf{x}) &= \Sigma_0(\mathbf{x}, \mathbf{x}) - \Sigma_0(\mathbf{x}, \mathbf{x}_{1:n})\Sigma_0(\mathbf{x}_{1:n}, \mathbf{x}_{1:n})^{-1}\Sigma_0(\mathbf{x}_{1:n}, \mathbf{x}) \end{aligned} \tag{2.18}$$

$\mu_0(\mathbf{x})$ is called the mean function, and $\Sigma_0(\mathbf{x}, \mathbf{x}')$ is called the kernel of the Gaussian process [9].

Common choices are a constant mean function, and the *power exponential* kernel [9]:

$$\begin{aligned} \mu_0(\mathbf{x}) &= \mu, \quad \mu \text{ constant} \\ \Sigma_0^2(\mathbf{x}, \mathbf{x}') &= \alpha_0 e^{-\|\mathbf{x}-\mathbf{x}'\|^2}, \quad \alpha_0 \text{ constant} \end{aligned} \tag{2.19}$$

Random Forests

Another choice for an underlying model in Bayesian optimization, for example included in the HyperMapper software [20], is a Random Forest [7]. Random forests are based on decision and regression trees. Specifically, random forests are an ensemble method, which combines many tree classifiers, and introduces randomization into the learning process by selecting a subset of parameters for evaluating a split at each node. Bagging, also known as bootstrap aggregation, [6] is also used for each tree in the forest which means that a randomly selected subset of the data is used when creating the tree. This technique is used to prevent several trees in the same forest from being created by the same data.

Using random forests in Bayesian optimization can be advantageous for several reasons. Perhaps most notably, which is the rationale in HyperMapper [20] is that trees are easily adapted for both classification and regression, which make them suitable for both continuous and discrete data. Another reason, as suggested in [28], is that a full Gaussian process scales poorly due to the quadratic behaviour of the kernel function. Random forests on the other hand, scale readily to large input spaces.

A downside to using random forests is that, while the mean $\mu_n(\mathbf{x})$ is simply the forest's prediction, the variance $\sigma_n(\mathbf{x})$ is not as straight forward to compute. One approach is to compute the variance as a sum of the variance across the predictions from each tree, plus the average variance of each tree [11].

2.4.4 Particle Swarm Optimization

Particle Swarm optimization (PSO) is another black-box, derivative free optimization method that has been proven to be successful [4]. The algorithm was first outlined in the paper *Particle Swarm Optimization* by James Kennedy and Russell Eberhart in 1995, and the method draws inspiration from swarming theory and the nature of birds flocking and fish schooling. Furthermore, the algorithm is also related to Evolutionary programming [16].

PSO is implemented as several particles p_i, each with a position in the input space and a velocity v_i. In order to make the particles swarm and explore the input space, each particle has knowledge of:

- the best position for each particle, $p_{best}^{(i)}$
- the best position for the whole population of particles, $best$

Each iteration, all the particles are updated according to the following formula [16]:

$$\begin{aligned} v_i &\leftarrow v_i + r_1 C_1(best - p_i) + r_2 C_2(p_{best}^{(i)} - p_i) \\ p_i &\leftarrow p_i + v_i \end{aligned} \tag{2.20}$$

Where C_1, C_2 are constants, and r_1, r_2 are uniform random numbers in $[0, 1)$. If the new position would be outside a potential bound for a particle, the position is adjusted to be inside the bounds. [16]

The advantage of the Particle Swarm Optimization algorithm is that it is a straightforward algorithm. Furthermore, since the algorithms use simple calculations and do not store large amounts of information it is very efficient and inexpensive to run. Another advantage is that there are only two constants to set C_p and C_i, making it very easy to try out different values to find the optimal algorithm [16].

A few updates have been made to PSO, some of them described in [24]. Most of the changes proposed to PSO have the purpose of constraining the velocities v_i to maintain stability in the particle system. One such constraining change is called *constriction coefficients*, which can be implemented in many ways. A simple method, and the most commonly used is to rewrite the particle update as follows:

$$\begin{aligned} v_i &\leftarrow \chi\big(v_i + r_1 C_1(best - p_i) + r_2 C_2(p_{best} - p_i)\big) \\ p_i &\leftarrow p_i + v_i \\ \chi &= \frac{2}{C - 2 + \sqrt{C^2 - 4C}}, \quad \text{where } C = C_1 + C_2 > 4 \end{aligned} \tag{2.21}$$

The canonical version of PSO, as per [24], is to use this update formula with $C_1 = C_2 = 2.05$, while also limiting the velocity $v_i = \max(v_i, v_{max})$, where v_{max} is usually the maximum range of each variable in the function's input space.

Algorithm 3 below describes the Particle Swarm Optimization algorithm implemented with the changes described above [24].

Algorithm 3 Particle Swarm Optimization pseudo-code

Initialize position p_i for each particle randomly
Initialize velocity v_i for each particle randomly
$best \leftarrow$ particle in swarm with best fitness
$p_{best}^{(i)} \leftarrow p_i$
while loop until stopping criterion is met: **do**
  for for each particle in swarm **do**
    $r_1,\ r_2 \leftarrow random(0, 1)$
    $v_i \leftarrow \chi\big(v_i + r_1 C_1(best - p_i) + r_2 C_2(p_{best}^{(i)} - p_i)\big)$
    $p_i \leftarrow p_i + v_i$
    if $f(p_i) < f(p_{best}^{(i)})$ **then**
      $p_{best}^{(i)} \leftarrow p_i$
      if $f(p_i) < f(best)$ **then**
        $best \leftarrow p_i$
      end if
    end if
  end for
end while

2.4.5 Simulated Annealing

Simulated annealing is an optimization method inspired by the movements of an atom inside a material under successive cooling [17]. When temperatures are high, the atom moves quickly and travels through many high and low energy states. As the temperature cools, the atom moves less and settles into a low energy state. Thus, simulated annealing uses this cooling idea for optimization by simulating a particle x moving through the input space of the function to be optimized, f. When temperatures are high, the particle x takes big steps when moving, and it is also more likely to go to positions where $f(x_{i+1}) > f(x_i)$. As it cools, the steps become smaller, and larger function values are less likely to be accepted.

Formally, (classical) simulated annealing (CSA), works as described in algorithm 4.

We have the formula $r < e^{-(E_{x_{next}} - E_x)/t}$, that determines whether or not a point with higher energy is accepted. It can easily be verified that as the temperature decreases, this probability decreases as well.

Later, an updated version of the algorithm called fast simulated annealing (FSA) was proposed [31], which shows better performance than CSA in most tests. In FSA, the overall algorithm is the same. Instead of sampling the next point x_{next} by taking a random step from x, the next point is instead sampled from a lorentzian distribution:

$$g(x) = \frac{t}{(x^2 + t^2)^{\frac{D+1}{2}}} \quad , \tag{2.22}$$

Algorithm 4 Simulated Annealing Optimization pseudo-code

Select a random starting point x, and evaluate its energy $E_x \leftarrow f(x)$
Set an initial temperature t
while loop until stopping criterion is met **do**
  Generate next position x_{next} by taking a random step from x
  Evaluate the new energy $E_{x_{next}} \leftarrow f(x_{next})$
  if $E_{x_{next}} < E_x$ **then**
    accept the new point $x \leftarrow x_{next}$
  else
    generate a random number $r \in [0, 1]$
    if $r < e^{-(E_{x_{next}} - E_x)/t}$ **then**
      accept the new point $x \leftarrow x_{next}$
    end if
  end if
  decrease temperature $t \leftarrow \omega t, \omega \in [0, 1]$
end while

where t is the temperature, and D is the dimensionality of x. Since the Lorentzian distribution promotes occasional large steps, a faster temperature cooling schedule can be used, $t_n = \frac{t_0}{1+n}$, where n is the number of iterations that have been made in the algorithm.

Yet another updated version of this algorithm was proposed, called generalized simulated annealing (GSA) because it generalizes both CSA and FSA [32]. In this version, another set of changes are made from FSA. First, the distribution for selecting x_{next} is changed to a Cauchy-Lorentz distribution:

$$g(\Delta x) = \frac{t^{-\frac{D}{3-q_v}}}{\left(1 + (q_v - 1)\frac{\Delta x^2}{t^{\frac{2}{3-q_v}}}\right)^{\frac{1}{q_v - 1} + \frac{D-1}{2}}} \quad , \tag{2.23}$$

where we see q_v introduced as a parameter of the algorithm. Temperature cooling is updated from FSA:

$$t_n = t_0 \frac{2^{q_v - 1} - 1}{(1+n)^{q_v - 1} - 1} \quad . \tag{2.24}$$

Note q_v here as well. In GSA, the formula for accepting or rejecting a point has also been updated:

$$p(accept) = \begin{cases} 1, & E_{x_{next}} < E_x \\ 0, & (1 + (q_a - 1)(E_{x_{next}} - E_x)/t_a) < 0 \\ (1 + (q_a - 1)(E_{x_{next}} - E_x)/t_a)^{\frac{1}{1-q_a}}, & \text{otherwise} \end{cases} \tag{2.25}$$

Here q_a is introduced as another parameter to the algorithm. t_a is introduced as the *acceptance temperature*, which in the original paper is just $t_a = t$, though other choices exist.

Choosing q_v and q_a allows GSA to generalize both CSA and FSA. For the choice of $q_v = 1$, $q_a = 1$, the algorithm behaves like CSA, and for $q_v = 2$, $q_a = 1$ it behaves like FSA. Suggested values in the original paper are $q_v \approx 2.7$ and $q_a \approx -5$.

2.4.6 Nelder Mead Simplex

The Nelder Mead Simplex (NMS) algorithm is a local search heuristic that focuses on finding a local minimum, opposite of the evolutionary algorithms previously discussed [4]. John Nelder and Roger Mead first described the method in their 1965 paper "A simplex method for function minimazation". It is called a simplex method because it uses a simplex to perform optimization.

Formally, a *simplex* in $\mathbf{R}^n$ is a "bounded convex polytope with nonempty interior and exactly $n+1$ vertices" [4]. A more intuitive way of looking at a simplex is as a collection of points $\{v_0, v_1, \ldots, v_n\}$ in $\mathbf{R}^n$ such that the vectors $\{(v_1 - v_0), (v_2 - v_0), \ldots, (v_n - v_0)\}$ form a basis in $\mathbf{R}^n$ [4].

NMS calculates x_o as the centroid of all vertices, except the one with the worst fitness value. Using x_o, along with *reflection*, *expansion*, *contraction*, and *shrinking* operations, NMS iteratively maneuvers its simplex towards a local minimum.

1. Reflection is the first method used to calculate new points. Reflection computes a reflection point based on the centroid x_o and the worst point in the simplex according to the formula [21]:

$$x_r = (1 + \alpha) \cdot x_o - \alpha \cdot x_{worst} \qquad \alpha \text{ Reflection coefficient} \tag{2.26}$$

   if the resulting point is better than the second worse point, but worse than the best point, i.e. $f(x_{best}) \leq f(x_r) < f(x_{worst})$, the resulting point replaces the worst point and the iteration is restared. However, if the reflected point x_r is better than the best point x_{best}, the iteration continues.

2. The second updating method is expansion. The expanded point is computed by using the centroid x_o and the reflected point according to [21]:

$$x_e = \gamma \cdot x_r + (1 - \gamma) \cdot x_o \qquad \gamma \text{ Expansion coefficient} \tag{2.27}$$

   If the expanded point x_e is better than the best point, i.e. $f(x_e) < f(x_{best})$, the worst point x_{worst} is replaced by the expanded point x_e. However, if the expanded point x_e is worse than the best point x_{best} the point is considered as a failed expansion, and the worst point x_{worst} is replaced by the reflected point x_r.

3. The next step is the contraction step, where the contracted point is computed by using the centroid x_o and the worst point, or the reflected point depending on which has the lowest score. The contracted point is calculated according to [21]:

$$x_c = \beta \cdot x_{worst} + (1 - \beta) \cdot x_o \qquad \beta \text{ Contraction coefficient} \tag{2.28}$$

   If the contracted point is better than the worst point, the contracted point replaces the worst point and, the iteration restarts. If the contracted point is not better than the worst point, the algorithm replaces all points in the simplex except for the best one with points based on the centroid x_o and the current point. This step is sometimes referred toas the shrink [4].

The algorithm can be summarized using the following pseudo-code [4, 21]:

Algorithm 5 Nelder Mead Optimization pseudo-code

simplex $x_0, x_1, \ldots x_n$
$x_o \leftarrow$ centroid of points $x_0, x_1, \ldots x_{n-1}$
reflection:
$x_r \leftarrow (1 + \alpha) \cdot x_o - \alpha \cdot x_{worst}$
if $f(x_{best}) \leq f(x_r) < f(x_{worst})$ **then**
    $x_{worst} \leftarrow x_r$, return to start
else if $f(x_{best}) > f(x_r)$ **then**
    goto expansion
else
    goto contraction
end if
expansion:
$x_e \leftarrow \gamma \cdot x_r + (1 - \gamma) \cdot x_o$
if $f(x_{best}) > f(x_e)$ **then**
    $x_{worst} \leftarrow x_e$ return to start
end if
contraction:
$x_c \leftarrow \beta \cdot x_{worst} + (1 - \beta) \cdot x_o$ then
if $min(f(x_{best}, f(x_r)) < f(x_c)$ **then**
    $x_{worst} \leftarrow x_c$ return to start
else
    $x_i \leftarrow x_o + \delta \cdot (x_i - x_o),$ for $i \in [1, n]$
end if

2.4.7 Rowan's Subplex Method

Nelder and Mead's simplex methods have a few well-known weaknesses [26]. Most notably, NMS sometimes collapses into a subspace when there are bounds constraints on the input space. Furthermore, NMS generally performs poorly as dimensionality increases. In his Ph.D. thesis [26], Rowan describes a new method that he calls the *Subplex* method, which aims to alleviate the weaknesses of NMS.

Pseudo-code for the subplex method, described in algorithm 6, is very simple at first glance. Now, of course, there is a significant amount of complexity hidden in that pseudo code.

Algorithm 6 Rowan's Subplex method pseudo-code

```
while termination test not satisfied do
    set stepsizes
    set subspaces
    for each subspace do
        search subspace with Nelder Mead Simplex method
    end for
    check termination test
end while
```

In the subplex method, a few parameters are used:

n:	the problem dimension
x:	the currently best known point
$step$:	step sizes in each dimension
Δx:	the change in x from the previous subplex iteration
$\alpha, \beta, \gamma, \delta$:	Nelder Mead Simplex parameters
ψ:	simplex reduction coefficient
ω:	step reduction coefficient
$nsmin, nsmax$:	minimum and maximum subspace dimensions
$nsubs$:	the number of subspaces searched

Set stepsizes: Initial step sizes are user-defined in Rowan's work. Usually initial step sizes are determined based on the effective range in the input space for each dimension. In every subsequent iteration of the subplex method, the step sizes are set based on the current values, and Δx, as well as the parameters ψ, ω:

$$step = \begin{cases} \min(\max(\frac{\|\Delta x\|_1}{\|step\|_1}, \omega), \frac{1}{\omega}) \cdot step, & \text{when } nsubs > 1 \\ \psi \cdot step, & \text{when } nsubs = 1 \end{cases} \tag{2.29}$$

After scaling $step$ using 2.29, $step$ is oriented as follows:

$$step_i = \begin{cases} sign(\Delta x_i) \cdot |step_i| & \text{when } \Delta x_i \neq 0 \\ -step_i & \text{when } \Delta x_i = 0 \end{cases} \tag{2.30}$$

Where the subscript i denotes the ith element of the vectors.

Set subspaces: This is the step where the most complexity comes in. Rowan's method uses Δx to determine the most promising subspace to search using NMS. Setting subspaces is done by first sorting the elements in Δx by their absolute value so that the largest values are first. With Δx sorted, the following function is maximized:

$$\begin{cases} \frac{\|(\Delta x_1, \ldots, \Delta x_k)\|_1}{k} - \frac{\|(\Delta x_{k+1}, \ldots, \Delta x_n)\|_1}{n-k} & \text{when } k < n \\ \frac{\|(\Delta x_1, \ldots, \Delta x_n)\|_1}{n} & \text{when } k = n \end{cases} \tag{2.31}$$

Maximizing 2.31 results in finding any distinct drops in the absolute value of Δx_i. Thus, Rowan's method selects the subspace of $k \in [nsmin, nsmax]$ that has the largest absolute

values in Δx. Δx is repeatedly partitioned until there are no subspaces left, at which point each subspace is searched using NMS.

Termination test: Rowan's method terminates when little-to-no progress is made in successive iterations. His termination criterion is:

$$\frac{\min(\|\Delta x\|_\infty, \|step\|_\infty \cdot \psi)}{max(\|x\|_\infty, 1)} \leq tolerance \tag{2.32}$$

Where $tolerance$ is user defined. This termination test checks that both Δx and $step$ are sufficiently small.

2.4.8 DIRECT

First presented by Jones, Perttunen & Stuckman, the DIviding RECTangles method (DIRECT), is based on Shubert's method [14].

Shubert's method is constructed around the assumption that the function being optimized is globally Lipschitzian, which means that there is a known C, which is the maximum rate of change in the function [29]. Formally there exists a constant C, called the Lipschitz constant, such that:

$$|f(x) - f(x')| \leq C|x - x'| \tag{2.33}$$

f being Lipschitzian implies that for every point $x \in [a, b]$ [14]:

$$\begin{aligned} f(x) &\geq f(a) - C(x - a) \\ f(x) &\geq f(b) + C(x - b) \end{aligned} \tag{2.34}$$

or more generally:

$$f(x) \geq f(x') - C|x - x'| \tag{2.35}$$

for any $x, x' \in [a, b]$.

Shubert's method uses this as a heuristic when selecting the next point to evaluate. Given previously evaluated points $x_0 \ldots x_n$, the method select x_{n+1} such that [29]:

$$x_{n+1} = \underset{x \in [a,b]}{\operatorname{argmin}} \max_{k=0,\ldots,n} f(x_k) - C|x - x_k| \tag{2.36}$$

Intuitively, Shubert's method evaluates two ends of an interval $[a, b], x_0 = a, x_1 = b$. From there, two lines are drawn:

$$\begin{aligned} g_1(x) &= f(x_0) - C|x - x_0| \\ g_2(x) &= f(x_1) - C|x - x_1| \end{aligned} \tag{2.37}$$

x_2 is then selected such that $g_1(x_2) = g_2(x_2)$, and the interval is subdivided into two intervals $[x_0, x_2], [x_2, x_1]$, and the same procedure is applied when selecting x_3. Now the problem becomes which of the intervals to subdivide when looking for x_3. Recall, the selection criterion 2.36 is formulated as $\operatorname{argmin}(\max(...))$ of all previously known points, which is equivalent to selecting the subinterval where the intersection $g_1(x) = g_2(x)$ has the smallest value.

DIRECT is largely based on Shubert's method. However, it doesn't assume knowledge of a Lipschitz constant [14]. Instead of sampling points at the ends of an interval, DIRECT samples the center point. Thus, the initial point in DIRECT for a function in the interval $[a, b]$ is:

$$x_0 = \frac{a+b}{2} \tag{2.38}$$

So each subinterval $[a_i, b_i]$ in DIRECT is based around its center point x_i. When selecting a subinterval to subdivide, DIRECT uses the center function value $f(x_i)$ as well as the size of the interval $\frac{b_i - a_i}{2}$. Having selected an interval $([a_i, b_i], x_i)$, it is divided into thirds with centers $x_{i+1} = x_i - \frac{b-a}{3}, x_{i+2} = x_i + \frac{b-a}{3}$.

For selecting an interval to split, the authors define what they call *potentially optimal intervals*. An interval $x_j \in [a_j, b_j]$ is potentially optimal if it satisfies:

$$\begin{aligned} f(x_j) - K\frac{b_j - a_j}{2} &\leq f(x_i) - K\frac{b_i - a_i}{2}, \ \forall\, i = 1..n \\ f(x_j) - K\frac{b_j - a_j}{2} &\leq f_{min} - \epsilon|f_{min}| \end{aligned} \tag{2.39}$$

where f_{min} is the best known function value, $\epsilon > 0$ constant, and $K > 0$ an arbitrary rate-of-change constant. This selection criterion is equivalent to plotting each subinterval in a 2-dimensional canvas with $\frac{b_i - a_i}{2}$ on the x-axis, and $f(x_i)$ on the y-axis, and selecting the lower convex hull of the points [14].

Generalizing DIRECT to multiple dimensions is done by subdividing intervals, now hyper-rectangles, in a single dimension at a time. Selecting a hyper-rectangle is done by the same procedure as above (2.39), only using $\frac{\|b_i - a_i\|_2}{2}$ since a_i, b_i are now vectors. When a hyper-rectangle has been selected, DIRECT singles out the dimensions with the largest range. Intuitively, that is the longest sides of the hyper-rectangle. If only a single dimension has the largest range, it is split. However, if several dimensions are the largest, all said dimensions are subdivided, and their corresponding centers evaluated. Dimensions are then subdivided according to the ordering of their function values, lowest first.

Chapter 3
Methodology

This chapter will discuss the construction and design choices made when creating the software architecture required for the experiments.

3.1 Black Box Architecture

In order to tune parameters on a camera and determine whether or not the parameters produce an acceptable image, a software pipeline has to be put in place. That software pipeline will form the black-box to be optimized in the experiments.

Figure 3.1 presents the architecture used.

Since it is designed as a black-box for optimization, the software pipeline has to take a set of parameters as input and give a number as output. In order to accomplish that, a controller is designed responsible for communicating with a physical camera unit. When given a set of parameters, the controller will first set the corresponding parameters in the physical camera. After the parameters have been updated in the camera, the controller requests an image, which is then compared to the reference image.

With execution speed as a primary goal, C and C++ were the most obvious language choices. For ease of implementation, C++ was eventually chosen and is used for all components.

3.1.1 Image Injection

One useful feature found in the cameras is the ability to *inject* raw sensor data into the camera pipeline, as illustrated in Fig. 3.2

Image injection effectively allows for a pixel-exact scene to be run through the image processing pipeline in a loop, which eliminates many complications from having a dynamic moving scene.

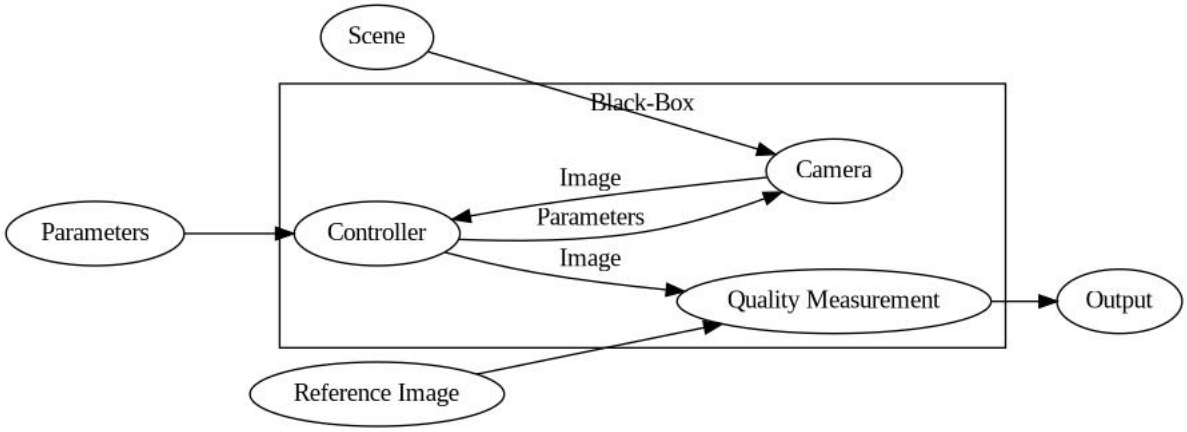


Figure 3.1: A graph of the black-box

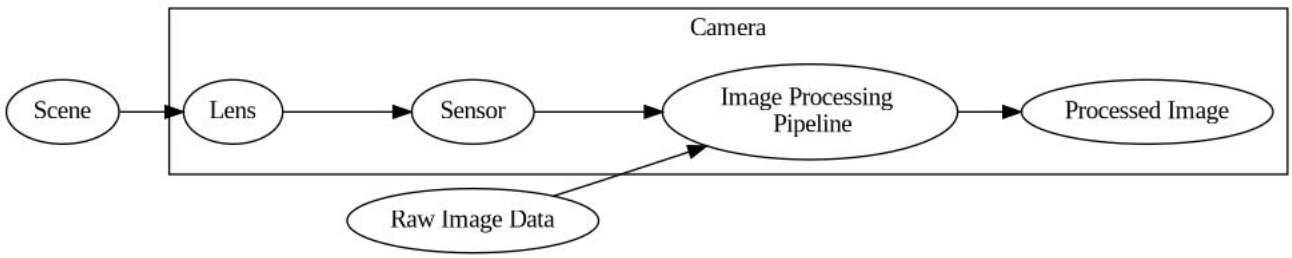


Figure 3.2: Illustration if image injection

3.1.2 Setting Parameters in the Camera

Cameras used in this thesis are controlled over a network connection, as mentioned in section 2.1. The network socket interface provides a simple way to set parameters in the camera. The parameters are all real-valued with different lower and upper bounds. Some parameters are limited to a lower bound of 0 and an upper bound of 1, while others can be limited to -40 and 40. These bounds are provided to the algorithm when initializing the optimization, and no pre-processing of the bounds is done beforehand.

3.1.3 Requesting an Image from the Camera

When hardware parameters are updated, there is a delay from the time that they are sent from the controller until they take effect in the image. By default, there is no way to automatically detect when parameters have taken effect.

Retrieving an image from the camera is split into two separate programs: a client that is a part of the controller and a server running on the camera.

In order to retrieve an image, the client sends a request to the server on the camera. The server then starts capturing images from the image pipeline. Every time a new image is captured, the server computes an MSE between the current and previous image. Only after N images in a row have an MSE of less than 10^{-4} the last image is sent back to the client. This approach thus ensures that nothing is changing in the image before sending it to the client. Such a simple algorithm would not be possible if not for the guarantee of a pixel-exact scene.

Throughout the experiments, $N = 5$ is used, so five images are stable before sending an image to the client.

3.1.4 Image Comparison

In order to determine if an image from the camera is of good quality, it is compared to a reference image. Several methods were constructed with the help of OpenCV, namely:

- Y-channel histogram comparison
- U-channel histogram comparison
- V-channel histogram comparison
- Gradient magnitude histogram comparison
- Gradient direction histogram comparison
- Image-aligned pixel comparison (Y-channel)
- Image-aligned pixel comparison (U-channel)
- Image-aligned pixel comparison (V-channel)

Gradient direction histogram quickly failed to produce a desirable result, so it was almost immediately excluded from further testing.

From the rest of the methods, two compound methods were constructed. First, the histogram methods were merged into a single histogram comparison method consisting of Y, U, V, and gradient magnitude histograms. Secondly, image-aligned pixel comparison was combined into a single method consisting of pixel-by-pixel comparison across Y, U, and V layers. Both of these methods were then tested on manually altered images in order to see which methods corresponded best with a subjective opinion.

A reference image was taken, followed by an image with the camera slightly rotated. The rotated image was then manually modified by changing focus, brightness, contrast, saturation, and sharpness. Unfortunately, these images can not be included here, as they were taken inside Axis' facilities.

Test Results

All images were looked at by humans, who subjectively scored the images based on their similarity to the reference image. Scoring was done before running any tests. Figure 3.3 shows a plot of how the two compound methods performed. Images on the X-axis are ordered by how subjectively different they are compared to the reference image. Thus, an ideal comparison would produce a strictly increasing line. On the Y-axis is RMSE values produced by the comparison algorithms, normalized for readability.

Based on Fig. 3.3, it is clear that an aligned pixel-by-pixel comparison corresponds best with a human's subjective opinion. As such, pixel-by-pixel comparison is the method of choice for the rest of this thesis.

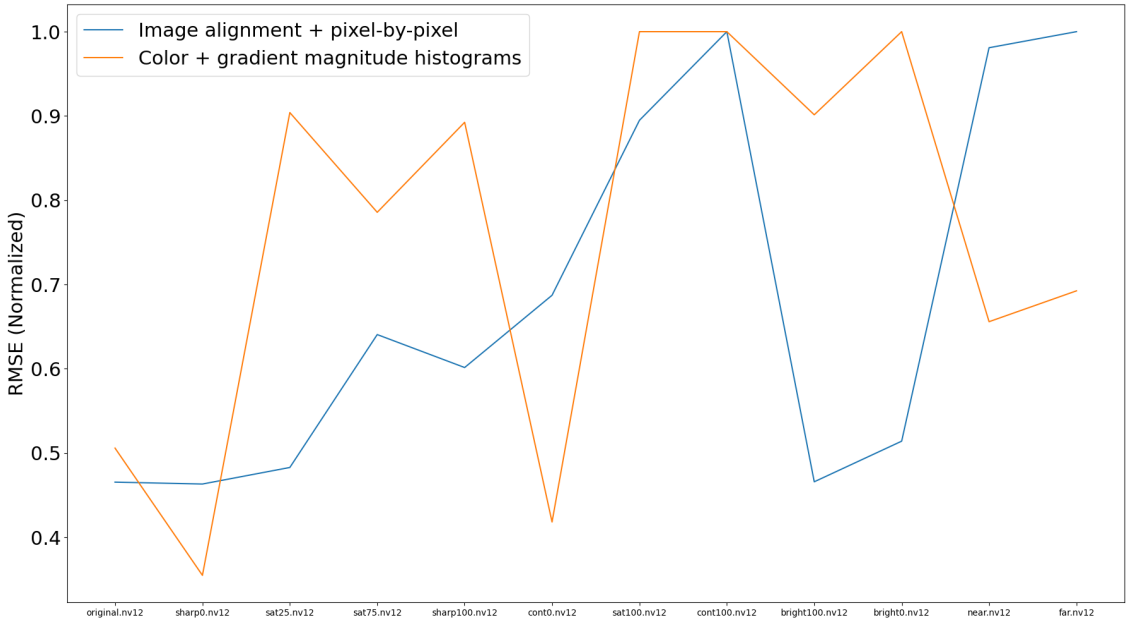


Figure 3.3: Histogram-based comparison vs. Image-aligned pixel-by-pixel comparison. Y-axis is RMSE, normalized for readability. X-axis is different images, ordered by (subjective) similarity to the reference image.

3.1.5 Highlighting Differences in Images

When looking at the resulting images from optimization, it can sometimes be difficult to spot differences. In order to highlight differences between two images, a *Difference Map* will be used. The computation of a difference map is very simple, and the pseudo-code is shown in algorithm 7. In short, the difference is computed pixel-by-pixel, and then normalized to $[0, 1]$, where a value of 1 will be completely white, and 0 will be completely black.

Algorithm 7 Difference map pseudo-code

Input: Images a and b
Initialize black output image o
for each pixel a_i, b_i **do**
    $o_i \leftarrow |a_i - b_i|$
end for
$o \leftarrow \frac{o}{\max(o)}$

3.1.6 Dataset and Reference Images

For this thesis, reference images are created by performing image injection and capturing a processed image with the camera's human-tuned parameters. In a real-world scenario, it is trivial to capture a reference image from another camera that has already been manually tuned.

Three images, or scenes, will be used in the experiments. They are presented in Fig. 3.4. These scenes will be tested separately in the experiments, only to validate that the results are not image-dependent. Results will thus not be combined across scenes. The reason for picking these specific scenes is because they represent very different use cases for the camera. For example, the focus and distance to the objects in the scene are different in each picture. Furthermore, the colors and contrast are also very different between the images.

(a) Lobby **(b)** Desk **(c)** Plant

Figure 3.4: The three scenes used when testing with the main scene on the left.

3.2 Random Sampling

In order to have a baseline result that the optimization algorithms can be compared to, random sampling will be used. Random sampling works by setting each parameter to a uniform random value in its allowed range.

Algorithm 8 Random sampling pseudo-code

while stopping criterion is not met **do**
  for each parameter x_i in $\mathbf{x}$ **do**
    $x_i \leftarrow \text{random}(lower_i,\ upper_i)$
  end for
  evaluate $f(\mathbf{x})$
end while

3.3 Optimization Algorithms

The following subsections describe implementation details and parameter choices for each optimization algorithm. All optimization algorithms are implemented in C++, just as the black-box.

3.3.1 Evolutionary Optimization

For evolutionary optimization, population size was chosen to be 50 based on data presented by De Jong [15]. De Jong suggests that a population size of less than 10 often failed to converge. Furthermore, a population size of over 100 often caused slow convergence. As such, a population size of 50 seems like a rational choice. Initial testing supports 50 as a default population size.

In terms of selection methods, both elitism selection and tournament selection have been implemented. Both methods seem to produce very similar results in initial testing, which leads to elitism being selected for the more extensive experiments due to its simplicity.

A ranked version of crossover is also used such that any parameter in the offspring has a 65% chance of being selected from the parent with better fitness. Each parameter in the offspring is also mutated with a 20% probability.

Finally, the population is regularized by removing the oldest member in the population at each iteration.

3.3.2 Bayesian Optimization

When constructing the implementation of Bayesian optimization for this thesis, it was decided that a variation of random forests was to be used, specifically a random forest of *Extra Randomized Trees* proposed in [23].

Recall that plain random forests introduce randomization by only considering a subset of parameters when selecting the best split (section 2.4.3).

Each individual split is computed by finding the single value in the parameters that minimizes the variance in the resulting subtrees. Extra randomized trees instead pick a random split value for all parameters, and then selects the parameter that happend to get the best split value. Pseudo code for computing splits are found in algorithms 9 and 10, respectively.

Selecting the best split for each parameter is the most computationally expensive part of constructing a random forest. As a result of avoiding the computation of best splits, extra randomized trees are significantly faster to construct [23].

Algorithm 9 Splitting procedure for plain random forest

Input: N parameters $x_1, x_2, \ldots, x_N$, integer $n < N$
Randomly select n parameters $p_1, p_2, \ldots, p_n$ from $x_1, x_2, \ldots x_N$
for i in 1 to n **do**
    Compute best split s_i for parameter p_i
end for
return best split s_{best}

Algorithm 10 Splitting procedure for extra randomized trees

Input: N parameters $x_1, x_2, \ldots, x_N$
for i in 1 to N **do**
    Compute random split s_i for parameter x_i
end for
return best split s_{best}

Optimization over the Random Forest model is done by first predicting 10000 random points, and then refining the ten most promising points using a simple local search procedure. Local search is done by iteratively predicting neighboring positions until no improvement is found. Neighboring positions are found by simply adding random steps to each parameter.

3.3.3 Particle Swarm Optimization

Particle Swarm Optimization is implemented exactly as the canonical version presented in 2.4.4. As such, the velocities are limited, and parameters are $C_1 = C_2 = 2.05$. Prior testing revealed that a population of 25 seems to perform well, so 25 is the chosen default population size.

3.3.4 Simulated Annealing

Generalized simulated annealing is implemented, with parameters choices $q_v = 2.6$ and $q_a = -5$ as suggested in [33]. Appropriate initial temperature varies based on the task, and no default is used.

3.3.5 Rowan's Subplex Method

All parameters associated with Rowan's subplex method are set to the default values presented in Rowan's thesis [26], presented in Table 3.1.

Table 3.1: Parameters used in Rowan's subplex method

Parameters	Values
$\alpha, \beta, \gamma, \delta$:	1, 2, 0.5, 0.5
ψ:	0.25
ω:	0.1
$nsmin, nsmax$:	2, 5

3.3.6 DIRECT

DIRECT does not contain any details left to the individual implementation, and as such it is implemented as described in 2.4.8.

3.3.7 Robustness of Optimizer Implementations

When implementing algorithms in code, it is appropriate to run tests in order to verify the correctness of said implementation.

In order to verify the correctness of implementations used in this thesis, they were all tested on a number of benchmark functions from a paper by Jamil and Yang [13]. They were also compared to an established off-the-shelf optimizer, specifically BayesOpt [18]. BayesOpt was chosen over some more popular frameworks on the basis that it is a C/C++ library.

Benchmark functions included are presented in table 3.2.

Table 3.2: Benchmark functions used

Function	Dimensionality
Ackley	2, 4, 8, and 16
Colville	4
Griewank	2, 4, 8, and 16
Sphere	2, 4, 8, 16, and 32
Rosenbrock	2, 4, 8, 16, and 32

Benchmark Results

Results, presented in tables 3.3, 3.4, 3.5, 3.6, and 3.7 show the function values found after 1000 iterations of each algorithm, with standard deviations. Each value is the average of five separate optimizations. Note that lower values represent a better result. Implementations constructed for this thesis are marked with an asterisk (*). These tables show that implementations used in this thesis perform competitively with established off-the-shelf optimizers.

Table 3.3: Benchmark results for the Ackley function. Presented as mean ± standard deviation. Each number is computed from 5 runs of 1000 iterations each. Best optimizers in **bold**.

Optimizer	2D	4D	8D	16D
BayesOpt	0.19 ± 0.16	0.53 ± 0.12	1.04 ± 0.18	1.34 ± 0.31
Bayesian*	$\mathbf{0.01 \pm 0.01}$	0.78 ± 0.09	2.26 ± 0.03	2.98 ± 0.10
Evolutionary*	0.48 ± 0.18	0.80 ± 0.20	1.26 ± 0.20	2.95 ± 0.22
Particle Swarm*	0.13 ± 0.15	0.82 ± 0.13	1.63 ± 0.17	2.55 ± 0.28
Simulated Annealing*	0.22 ± 0.21	$\mathbf{0.41 \pm 0.20}$	0.86 ± 0.35	1.32 ± 0.33
Rowan's Subplex*	0.50 ± 0.32	1.19 ± 0.50	$\mathbf{0.71 \pm 0.38}$	$\mathbf{0.99 \pm 0.31}$
DIRECT*	0.62 ± 0.00	0.78 ± 0.00	0.78 ± 0.00	3.07 ± 1.10

Table 3.4: Benchmark results for the Colville function. Presented as mean ± standard deviation. Each number is computed from 5 runs of 1000 iterations each. Best optimizers in **bold**.

Optimizer	4D
BayesOpt	$\mathbf{0.12 \pm 0.10}$
Bayesian*	4.97 ± 0.65
Evolutionary*	19.26 ± 17.58
Particle Swarm*	5.42 ± 5.25
Simulated Annealing*	24.23 ± 35.26
Rowan's Subplex*	1.47 ± 1.80
DIRECT*	1.52 ± 1.48

Table 3.5: Benchmark results for the Griewank function. Presented as mean ± standard deviation. Each number is computed from 5 runs of 1000 iterations each. Best optimizers in **bold**.

Optimizer	2D	4D	8D	16D
BayesOpt	$\mathbf{0.00 \pm 0.00}$	0.19 ± 0.12	0.58 ± 0.21	0.80 ± 0.17
Bayesian*	0.12 ± 0.08	0.16 ± 0.05	0.95 ± 0.12	1.45 ± 0.05
Evolutionary*	0.05 ± 0.04	0.18 ± 0.06	0.87 ± 0.16	1.53 ± 0.12
Particle Swarm*	0.02 ± 0.01	0.16 ± 0.08	0.48 ± 0.12	1.11 ± 0.03
Simulated Annealing*	0.13 ± 0.08	0.23 ± 0.20	0.15 ± 0.14	0.63 ± 0.25
Rowan's Subplex*	0.09 ± 0.02	0.17 ± 0.08	0.20 ± 0.34	$\mathbf{0.01 \pm 0.02}$
DIRECT*	0.02 ± 0.01	$\mathbf{0.02 \pm 0.02}$	$\mathbf{0.01 \pm 0.00}$	0.18 ± 0.16

Table 3.6: Benchmark results for the Sphere function. Presented as mean ± standard deviation. Each number is computed from 5 runs of 1000 iterations each. Best optimizers in **bold**.

Optimizer	2D	4D	8D	16D	32D
BayesOpt	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$
Bayesian*	$\mathbf{0.00 \pm 0.00}$	0.05 ± 0.02	2.93 ± 0.92	20.58 ± 1.42	67.57 ± 6.14
Evolutionary*	0.02 ± 0.03	0.03 ± 0.02	0.86 ± 0.40	16.93 ± 3.70	141.82 ± 18.49
Particle Swarm*	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	0.14 ± 0.07	6.91 ± 5.96	24.22 ± 8.60
Simulated Annealing*	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	0.20 ± 0.26	6.86 ± 2.21
Rowan's Subplex*	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	0.58 ± 0.26
DIRECT*	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	0.79 ± 0.00	97.19 ± 1.58

Table 3.7: Benchmark results for the Rosenbrock function. Presented as mean ± standard deviation. Each number is computed from 5 runs of 1000 iterations each. Numbers for Rosenbrock were *very* large, so they have been converted into a logarithmic scale for readability. Best optimizers in **bold**.

Optimizer	2D	4D	8D	16D	32D
BayesOpt	-4.51 ± 2.49	2.28 ± 0.683	6.27 ± 0.682	8.56 ± 0.545	10.4 ± 0.607
Bayesian*	-3.48 ± 1.6	3.32 ± 0.859	9.16 ± 0.407	12.5 ± 0.597	14.8 ± 0.109
Evolutionary*	1.59 ± 2.17	5.76 ± 1.05	8.28 ± 0.511	12.2 ± 0.602	16.6 ± 0.243
Particle Swarm*	-2.03 ± 3.46	2.92 ± 1.68	7.01 ± 0.795	10.3 ± 1.04	13.6 ± 0.615
Simulated Annealing*	1.49 ± 2.02	4.33 ± 2.28	4.43 ± 2.57	7.53 ± 1.06	11.2 ± 0.769
Rowan's Subplex*	-10.6 ± 3.87	$\mathbf{-3.59 \pm 4.35}$	3.28 ± 1.76	$\mathbf{5.05 \pm 0.764}$	$\mathbf{8.97 \pm 0.468}$
DIRECT*	$\mathbf{-13 \pm 5.01}$	-1.86 ± 0.188	$\mathbf{3 \pm 0.513}$	6.09 ± 1.59	14.2 ± 0.47

Chapter 4
Experiments

In this chapter, the results from the experiments made during this thesis will be presented. Two main experiments were performed, one on 14 parameters and a more extensive experiment on 71 parameters. Each experiment is presented on its own in the sections below. All optimizers used in these experiments were implemented for this thesis. No off-the-shelf optimizers are used.

4.1 Optimizing 14 Parameters

Initial experiments that were made in this thesis explored optimization over 14 parameters. Most of the 14 parameters controlled blue fringe settings, while other parameters controlled the sharpness, contrast, brightness, and saturation. It is important to note that there are many more parameters in the camera, which are left unchanged in these initial experiments. Essentially, these 14 parameters form only a subset of the whole tuning problem. The primary reason for attempting a 14 parameter subset first, is to see if optimization works for a small problem before attempting a more complete parameter-space.

4.1.1 Methodology

With all 14 parameters implemented in the black-box, all optimizers were tested by running five separate optimizations of 500 iterations each. Optimization was performed on the three different scenes presented in section 3.1.6. Primary results and analysis are constructed around the main scene “Lobby”. Scenes “Desk” and “Plant” are used to verify results from the main scene.

4.1.2 Results

Results for 14 parameters are presented in tables 4.1, 4.2, 4.3, fig. 4.1, figures 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, and 4.8.
Tables 4.1, 4.2, and 4.3 present the average Mean Square Error (MSE) between the reference image and the tuned image from each optimizer. Means and standard deviation are calculated based on five separate optimizations. Minimum and maximum present the best and worst MSE produced by each optimizer. 500 iterations took 3 minutes and 46 seconds on average.

Table 4.1: Main scene: Lobby. Mean ± standard deviation, minimum and maximum values of the MSE found by each optimizer in 500 iterations. Best optimizers in **bold**.

Algorithm	Mean ± standard deviation	Minimum	Maximum
Random Sampling	47.86 ± 11.19	31.49	63.17
Bayesian	9.25 ± 0.97	7.84	**10.79**
DIRECT	16.37 ± 12.86	8.77	42.01
Evolutionary	10.68 ± 6.57	5.86	22.68
Rowan's Subplex	**6.25 ± 4.44**	**3.12**	15.02
Simulated Annealing	7.93 ± 5.65	3.41	18.75
Particle Swarm Optimization	14.18 ± 8.15	6.13	29.08

Table 4.2: Scene: Desk. Mean ± standard deviation, minimum and maximum values of the MSE found by each optimizer in 500 iterations. Best optimizers in **bold**.

Algorithm	Mean ± standard deviation	Minimum	Maximum
Random Sampling	64.39 ± 13.82	42.96	84.10
Bayesian	18.57 ± 9.82	12.14	38.15
DIRECT	53.81 ± 31.16	15.76	87.58
Evolutionary	19.92 ± 12.16	5.43	38.45
Rowan's Subplex	**6.35 ± 2.79**	**2.64**	**9.83**
Simulated Annealing	14.89 ± 7.897	9.23	30.56
Particle Swarm Optimization	13.57 ± 6.43	8.07	25.86

Table 4.3: Scene: Plant. Mean ± standard deviation, minimum and maximum values of the MSE found by each optimizer in 500 iterations for. Best optimizers in **bold**.

Algorithm	Mean ± standard deviation	Minimum	Maximum
Random Sampling	79.04 ± 24.13	32.80	102.96
Bayesian	19.88 ± 9.78	12.32	39.21
DIRECT	29.69 ± 22.96	12.30	72.91
Evolutionary	16.25 ± 6.17	9.92	27.17
Rowan's Subplex	$\mathbf{4.61 \pm 4.05}$	**1.03**	**12.09**
Simulated Annealing	14.19 ± 7.17	3.59	25.04
Particle Swarm Optimization	20.12 ± 8.61	8.41	30.36

Figure 4.1 presents a logarithmic graph based on the values from table 4.1 The figure shows the mean of the best MSE value found by all optimizers, at each iteration up to 500 iterations. A natural logarithm scale is used only for legibility. Standard deviation is excluded from this graph, also for legibility. Individual plots with mean and standard deviation are presented in Appendix A.

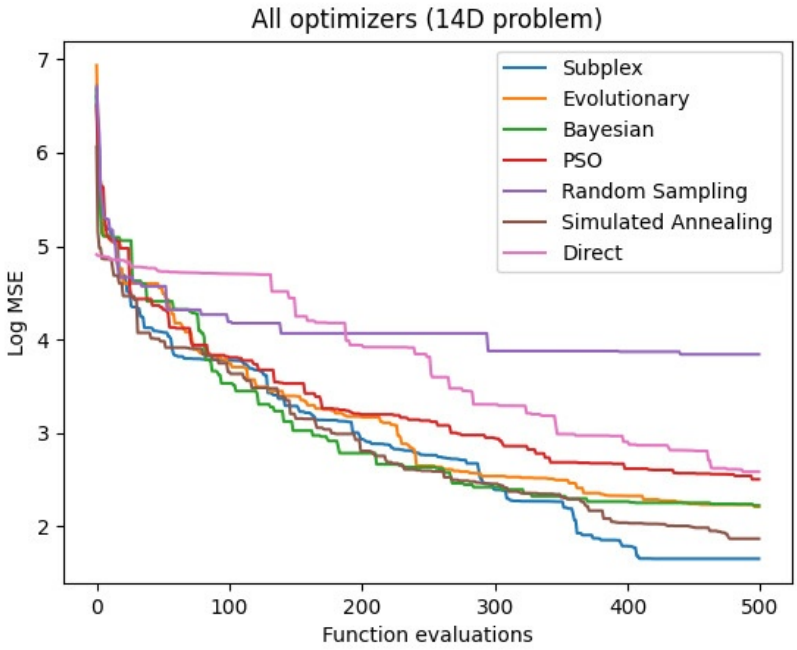
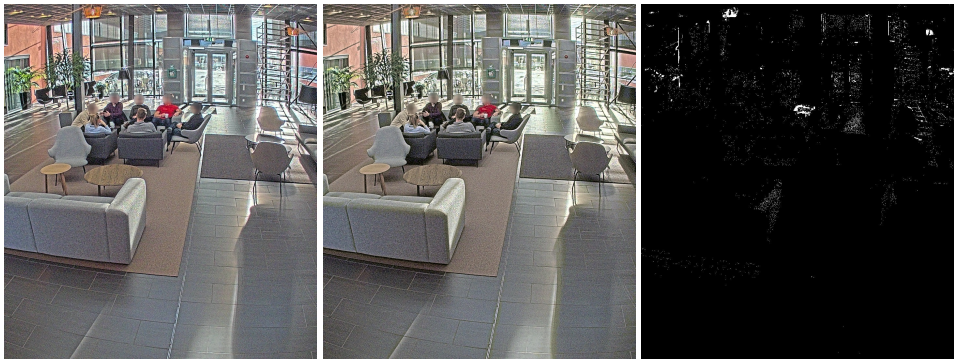


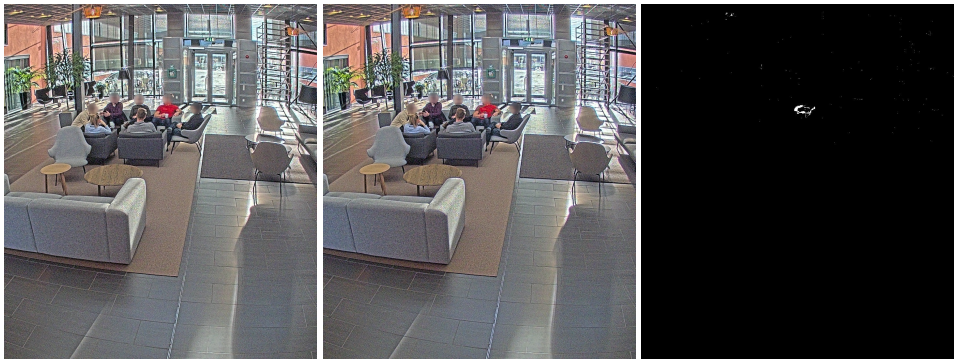
Figure 4.1: Main Scene: Lobby. Mean value of the MSE for each optimizer at each iteration. Note that Y-axis is logarithmic.

Finally, figures 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, and 4.8 show the reference image, along with the best image found by each optimizer after 500 iterations, including random sampling. Note that these images are not neccessarily from the best of the five optimization repetitions. Included is also a difference map.



(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.2: Random Sampling, MSE=42.0 with a maximum pixel difference of 65.7.



(a) Reference image **(b)** Output image **(c)** Difference map

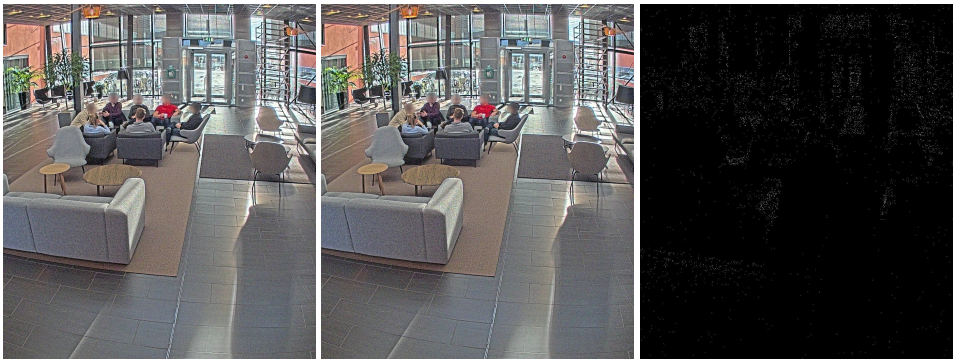
Figure 4.3: Bayesian Optimization, MSE=9.4 with a maximum pixel difference of 58.3.

(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.4: DIRECT, MSE=11.0 with a maximum pixel difference of 67.0.

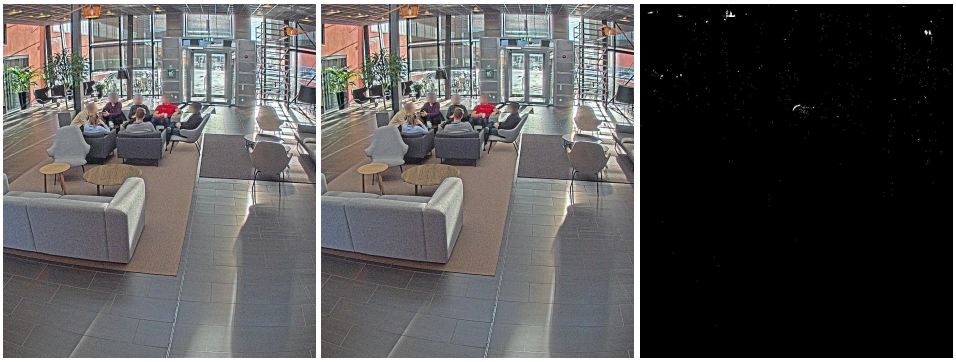
(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.5: Evolutionary Optimization, MSE=6.2 with a maximum pixel difference of 69.7.



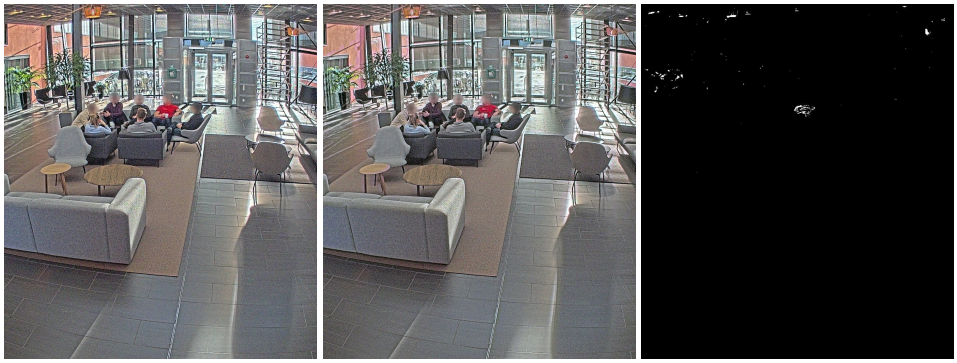
(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.6: Rowan's Subplex, MSE=4.4 with a maximum pixel difference of 34.3.



(a) Reference image **(b)** Output image **(c)** Difference map

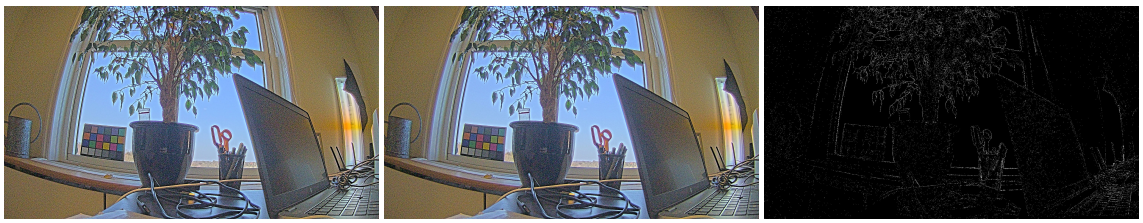
Figure 4.7: Simulated Annealing, MSE=4.1 with a maximum pixel difference of 43.0.



(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.8: Particle Swarm Optimization, MSE=11.8 with a maximum pixel difference of 34.33.

In order to visualize how well the best performing algorithm for 500 iterations did on the two other scenes, desk and plant, the best resulting image is presented in the figures 4.9 and 4.10 below.



(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.9: Rowan's Subplex, MSE=7.35 with a maximum pixel difference of 60.

(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.10: Rowan's Subplex, MSE=5.56 with a maximum pixel difference of 56.33.

4.1.3 Discussion

As can be seen from the optimized images, all of the algorithms perform very well compared to the random sampling strategy. Rowan's Subplex method performs particularly well on this 14-dimensional problem with a mean MSE of 6.25 on the main scene. Results from the main scene are verified on the second and third scene, which produce slightly different but overall similar results.

Note also that these results are only snapshots at 500 iterations. Figure 4.1 suggests that Bayesian optimization is the best performing algorithm between 100 and 300 iterations, which means that it could be the better choice depending on how many iterations the optimization is allowed to run.

When looking at the images produced from the optimizations, it can be seen that random sampling produces an image with a few notable artifacts and an MSE of 42.0. Nevertheless, the image is quite good compared to the reference image for the human eye. However, looking at images produced by the optimizers, all optimizers produce images that are virtually indistinguishable from the reference image.

Further, looking at the difference maps, differences seem to show up around edges and in the pixel-noise for most optimizers. This correlates most with the effects of the sharpness parameters, which suggests that the sharpness parameters may be the most difficult to tune.

Considering the fact that such a good result was produced in under 4 minutes on average, this 14-dimensional problem is considered solved.

4.2 Optimizing 71 Parameters

After the successful optimization experiments made on the 14 initial parameters, it was decided that the optimization should also be tested on a significantly larger portion of the IPP parameters, resulting in 71 parameters. Most of the parameters from the 14-dimensional problem were included in the 71-dimensional problem, apart from two parameters that were removed due to conflicts. Parameters included control virtually all parts of the camera's IPP. More parameters exist in the IPP, but some were not feasible to tune programmatically due to various reasons. For example, some parameters require a dynamic scene in order to take effect. Parameters not included are left to their hand-tuned values in these experiments.

4.2.1 Methodology

Since the new 71-dimensional problem is expected to be more difficult compared to the 14-dimensional problem, it was immediately decided to run the optimization over 1000 iterations instead of 500 to produce a satisfying result. The same three scenes from the 14-dimensional testing were used for these experiments. Five separate optimizations were run on the three scenes for each optimizer.

After looking at results from 1000 iterations, potential for further improvement was observed, which prompted tests with 5000 iterations. Tests were only done with 5000 iterations to explore if the algorithms could indeed converge to even better MSE values. Therefore, experiments with 5000 iterations were only performed on the main scene "Lobby".

4.2.2 Results

Results for 71 parameters are presented in tables 4.4, 4.5, 4.6, 4.7, figures 4.11, 4.12, and figures 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, and 4.19.

Tables 4.4, 4.5, and 4.6 present the average MSE between the reference image and the tuned image from each optimizer, at 1000 iterations for each scene. Average MSE for 5000 iterations is presented in table 4.7. Means are calculated based on five separate optimizations. Minimum and maximum present the best and worst MSE produced by each optimizer.

1000 iterations took 11 minutes and 5 seconds on average. 5000 iterations took approximately 1 hour.

Table 4.4: Main Scene: Lobby. Mean ± standard deviation, minimum and maximum values of the MSE found by each optimizer in 1000 iterations. Best optimizers in **bold**.

Algorithm	Mean ± standard deviation	Minimum	Maximum
Random Sampling	106.37 ± 20.31	73.34	129.47
Bayesian	49.54 ± 4.50	42.58	55.22
DIRECT	83.57 ± 8.21	76.76	96.07
Evolutionary	30.39 ± 8.53	21.82	44.99
Rowan's Subplex	40.93 ± 25.67	**13.71**	86.59
Simulated Annealing	**24.01 ± 7.21**	17.89	**35.95**
Particle Swarm Optimization	48.71 ± 24.57	22.47	92.90

Table 4.5: Scene: Desk. Mean ± standard deviation, minimum and maximum values of the MSE found by each optimizer in 1000 iterations. Best optimizers in **bold**.

Algorithm	Mean ± standard deviation	Minimum	Maximum
Random Sampling	157.18 ± 22.20	113.61	174.45
Bayesian	63.87 ± 2.75	59.72	67.11
DIRECT	84.11 ± 9.48	73.24	101.58
Evolutionary	31.90 ± 3.44	27.38	**37.78**
Rowan's Subplex	37.01 ± 16.62	**10.65**	61.22
Simulated Annealing	**28.86 ± 6.82**	21.01	38.76
Particle Swarm Optimization	48.59 ± 27.25	30.20	102.34

Table 4.6: Scene: Plant. Mean ± standard deviation, minimum and maximum values of the MSE found by each optimizer in 1000 iterations. Best optimizers in **bold**.

Algorithm	Mean ± standard deviation	Minimum	Maximum
Random Sampling	138.69 ± 31.85	96.91	184.99
Bayesian	70.35 ± 15.69	48.19	89.82
DIRECT	125.47 ± 8.57	114.13	135.98
Evolutionary	37.05 ± 10.28	28.52	56.70
Rowan's Subplex	33.99 ± 10.11	**15.84**	46.89
Simulated Annealing	**24.14 ± 6.14**	16.63	**33.63**
Particle Swarm Optimization	42.82 ± 7.27	35.20	55.18

Table 4.7: Main Scene: Lobby. Mean ± standard deviation, minimum and maximum values of the MSE found by each optimizer in 5000 iterations. Best optimizers in **bold**.

Algorithm	Mean ± standard deviation	Minimum	Maximum
Random Sampling	72.37 ± 16.96	41.48	89.41
Bayesian	34.55 ± 3.83	31.12	41.49
DIRECT	44.84 ± 12.98	32.65	68.25
Evolutionary	17.92 ± 1.22	15.79	19.06
Rowan's Subplex	18.05 ± 5.27	12.34	27.71
Simulated Annealing	**9.77 ± 1.71**	**7.62**	**11.89**
Particle Swarm Optimization	25.59 ± 4.70	18.76	32.37

Figure 4.11 presents a logarithmic graph which shows the best MSE value found by all optimizers, at each iteration up to 1000 iterations. A natural logarithm scale is used only for legibility. A similar graph for 5000 iterations is shown in figure 4.12.
Individual plots with mean and standard deviation for 5000 iterations are presented in Appendix A.

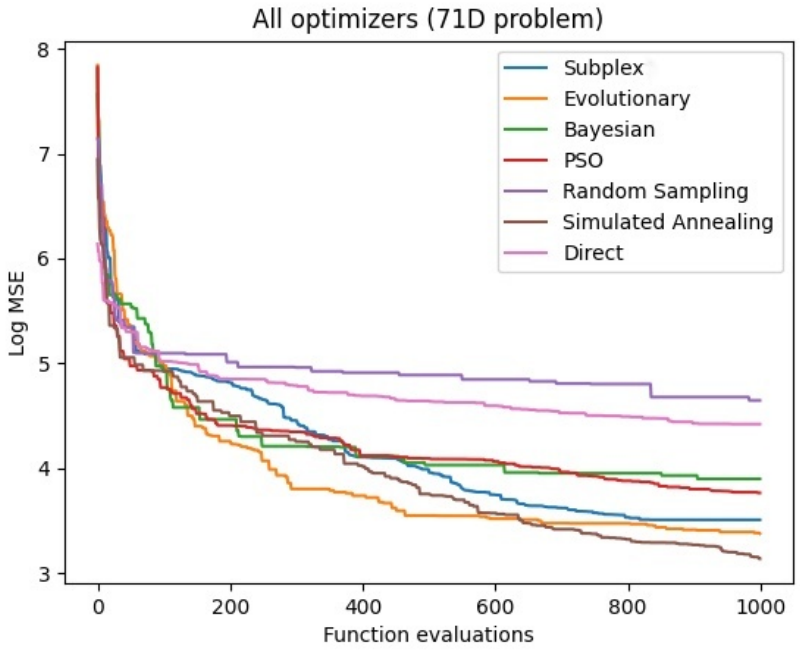


Figure 4.11: Main Scene: Lobby. Mean value of the MSE for each optimizer at each iteration up to 1000. This corresponds to data in table 4.4. Note that Y-axis is logarithmic.

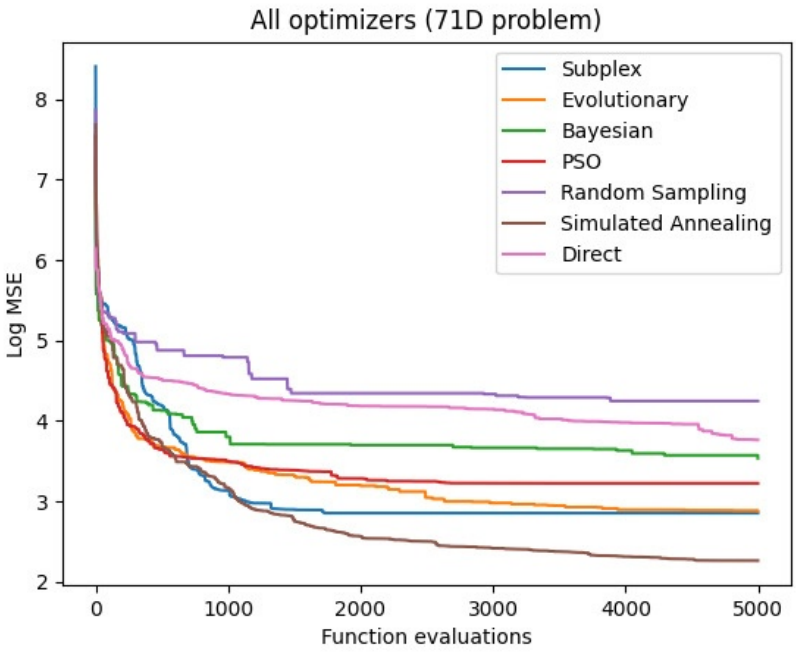
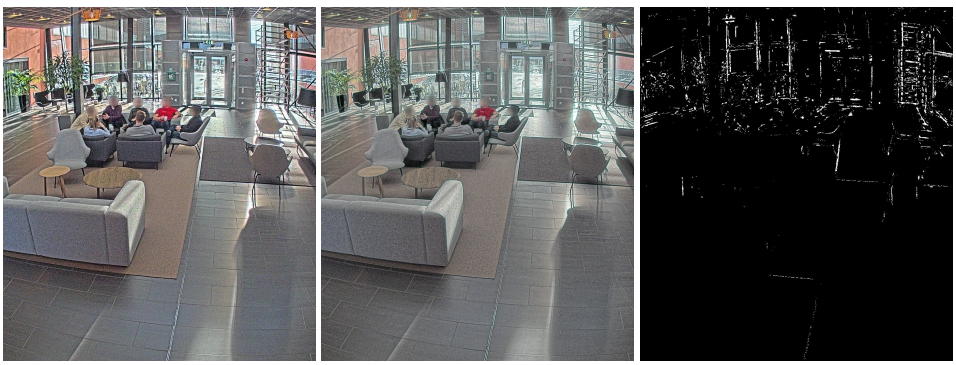


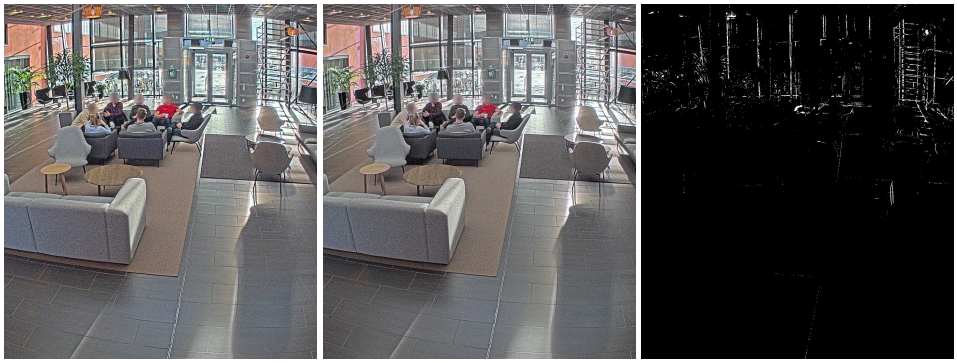
Figure 4.12: Main Scene: Lobby. Mean value of the MSE for each optimizer at each iteration up to 5000. This corresponds to data in table 4.7. Note that Y-axis is logarithmic.

Lastly, figures 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, and 4.19 show the reference image, along with the best image found by each optimizer after 1000 iterations, including random sampling. A difference map is also included to highlight differences.



(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.13: Random Sampling, MSE=107.6 with a maximum pixel difference of 121.0.



(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.14: Bayesian Optimization, MSE=48.7 with a maximum pixel difference of 93.7.

(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.15: DIRECT, MSE=87.6 with a maximum pixel difference of 90.7.

(a) Reference image **(b)** Output image **(c)** Difference map

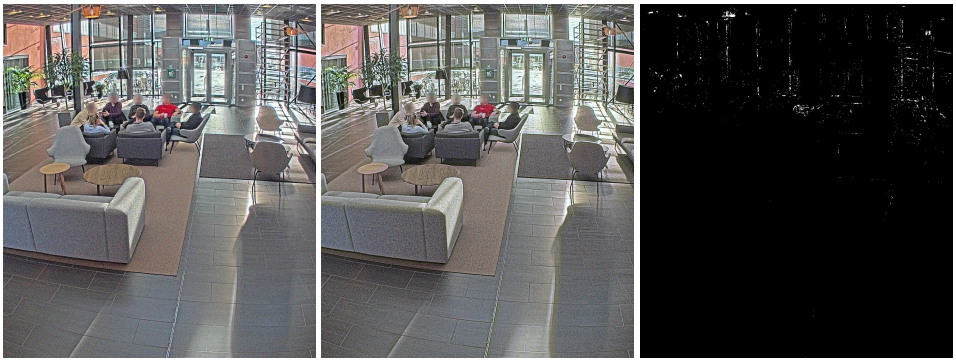
Figure 4.16: Evolutionary Optimization, MSE=28.7 with a maximum pixel difference of 72.7.

(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.17: Rowan's Subplex, MSE=35.0 with a maximum pixel difference of 114.7.

(a) Reference image **(b)** Output image **(c)** Difference map

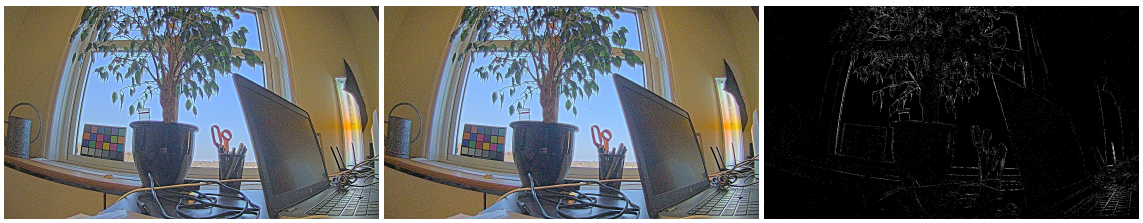
Figure 4.18: Simulated Annealing, MSE=25.8 with a maximum pixel difference of 50.0.



(a) Reference image **(b)** Output image **(c)** Difference map

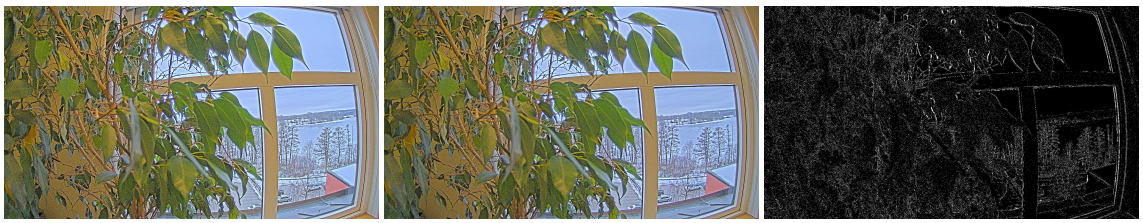
Figure 4.19: Particle Swarm Optimization, MSE=34.7 with a maximum pixel difference of 79.7.

In order to visualize how well the best performing algorithm for 1000 iterations did on the two other scenes, desk and plant, the best resulting image is presented in the figures 4.20 and 4.21 below.



(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.20: Simulated Annealing, MSE=22.56 with a maximum pixel difference of 53.33.



(a) Reference image **(b)** Output image **(c)** Difference map

Figure 4.21: Simulated Annealing, MSE=25.04 with a maximum pixel difference of 75.67.

4.2.3 Discussion

Random sampling, which can be considered the baseline to compare the algorithms against, produces the worst result as expected with a mean MSE of 106.37 and 72.37 after 1000 and 5000 iterations, respectively. One notable change from the 14-dimensional problem is that DIRECT performs significantly worse compared to the other optimizers. Perhaps surprisingly, Simulated Annealing now performs best on average, both at 1000 and 5000 iterations. Moreover, Simulated Annealing also performs best overall on all of the scenes. Close seconds are Rowan's Subplex method and evolutionary optimization. Worth noting is that Rowan's method has a considerable variance after 1000 iterations, resulting in a best MSE of 13.71, and a worst MSE of 86.59 on the main scene. Results from the experiments on the main scene are verified on the other two scenes "Desk" and "Plant", which gives confidence that the result is generalizable across images.

From figure 4.12 it seems like simulated annealing is a clear winner for any number of iterations over 1000. However figure 4.11 suggests that evolutionary optimization might be the better choice for anything less than 600 iterations.

Despite MSE values generally being significantly larger for this problem than the 14-dimensional one, the optimized images still look remarkably similar to the original. Though, as expected, the

best image from random sampling shows a more apparent difference. While there is now an observable difference for some optimizers, most notably DIRECT and particle swarm, the overall result is deemed to be of very good quality. No images were included for 5000 iterations since those experiments were performed only to see if the algorithms could converge further.

Since there are now many more parameters than in the 14-dimensional case, it is much more difficult to reason about which parameters are most difficult to tune. However, it is still the case that the largest differences in the difference maps seem to be around edges and in the pixel noise. Thus it is not unreasonable to believe that sharpness still proves to be a difficult obstacle for the optimizers.

On a final note, since 5000 iterations only takes approximately one hour, random sampling might actually be a perfectly viable option in practice. Even 100000 iterations might be feasible, which should take approximately 20 hours, and would give random sampling a chance at producing a very good result. Given the general timeframe of several weeks for manually tuning a camera hardware, random sampling might be desirable thanks to its simplicity.

Chapter 5
Conclusion

A black-box optimization framework has been put in place that can automatically tune a large number of camera hardware parameters to fit a reference image. Implementing and testing said framework has been the primary purpose of this thesis. In this section, results from the experiments and the implementation will be discussed and concluded. Limitations made during this thesis and possible future work will also be discussed.

5.1 Choice of Algorithm

Several different optimization algorithms have been tested.

The choice of an algorithm to use when optimizing a black-box problem often depends on the nature of the problem. This was demonstrated during the testing of the robustness of our algorithms in section 3.3.7. In those tests, the algorithms performed very differently depending on the problem. What was clear was that the algorithms that performed best overall were Simulated Annealing, Evolutionary and Rowan's Subplex. When optimizing hardware parameters in a physical camera unit, all three algorithms performed well regardless of dimensionality, number of iterations, or the nature of the scene optimized upon. Therefore the choice of algorithm to use on these particular problems should reasonably be one of Simulated Annealing, Evolutionary, or Rowan's Subplex algorithm. Across all tests, Simulated Annealing seems to be the most reliable.

5.2 Future work

For the majority of this thesis, the testing was made on the 14-dimensional problem. Therefore the implementation and study of algorithms were made on that premise. Testing of the 71-dimensional problem was only made during the latter stages of the thesis process. Therefore, there was no time to consider studying or testing other algorithms or methods possibly

better suited for a problem of higher dimensionality. One suggestion of future work would be to investigate different algorithms and methods to test if anything could yield a better result, particularly for the 71-dimensional problem.

Even though the 71-dimensional problem is of very high dimensionality, there exist even more parameters in the cameras used. Therefore another suggestion for further investigation would be to test more parameters to see if the algorithms could replace the human part of tuning. Another related suggestion would be to see if there are any gains to be made by tuning the blocks one by one and limiting the number of parameters used instead of tuning all of the parameters available at once.

This thesis did not explore how results generalize to different scenes. A final suggestion for future reasearch is to explore for example portfolio optimization, and to investigate how well a solution from one scene generalizes to others.

References

[1] About axis. https://www.axis.com/sv-se/about-axis, Retrieved 2021-04-28.

[2] Axis network cameras. https://www.axis.com/sv-se/products/network-cameras, Retrieved 2021-04-28.

[3] Picture style settings and customization. https://support.usa.canon.com/kb/index?page=content&id=ART170199, Retrieved 2021-04-28.

[4] Charles Audet and Warren Hare. *Derivative-Free and Blackbox Optimization*. Springer, 2017.

[5] Gunnar Blom, Jan Enger, Gunnar Englund, Jan Grandell, and Lars Holst. *Sannolikhetsteori och statistikteori med tillämpningar*. Studentlitteratur, 7th edition, 2017.

[6] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123 – 140, 1996.

[7] Leo Breiman. Random forests. *Machine Learning*, 45(1):5 – 32, 2001.

[8] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. *BRIEF: Binary robust independent elementary features.*, volume 6314 LNCS of *Lecture Notes in Computer Science*. Springer Verlag, EPFL, 2010.

[9] Peter I. Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018. Available at https://arxiv.org/abs/1807.02811.

[10] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 4th edition, 2018.

[11] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79 – 111, 2014.

[12] Keith Jack. *Video Demystified: A Handbook for the Digital Engineer*. Newnes, 2007.

[13] Momin Jamil and Xin-She Yang. A literature survey of benchmark functions for global optimization problems. 2013.

[14] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157 – 181, 1993.

[15] Kenneth A. De Jong. *Evolutionary Computation A Unified Approach*. MIT Press, 2006.

[16] James Kennedy and Russel Eberhart. Particle swarm optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*, pages 1942 – 1948, 1995.

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671 – 680, 1983.

[18] Ruben Martinez-Cantin. Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits. *Journal of Machine Learning Research*, 15:3735 – 3739, 2014.

[19] Ali Mosleh, Avinash Sharma, Emmanuel Onzon, Fahim Mannan, Nicolas Robidoux, and Felix Heide. Hardware-in-the-loop end-to-end optimization of camera image processing pipelines. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Computer Vision and Pattern Recognition (CVPR), 2020 IEEE/CVF Conference on, CVPR*, pages 7526 – 7535, 2020.

[20] Luigi Nardi, David Koeplinger, and Kunle Olukotun. Practical design space exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 347–358. IEEE, 2019.

[21] John A. Nelder and Roger Mead. A simplex method for function minimization. *The computer journal 7.4*, pages 308 – 313, 1965.

[22] OpenCV. Introduction. Available at https://docs.opencv.org/4.5.1/d1/dfb/intro.html.

[23] Geurts Pierre, Ernst Damien, and Wehenkel Louis. Extremely randomized trees. *Machine Learning*, 63(1):3 – 42, 2006.

[24] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[25] E. Rosten, R. Porter, and T. Drummond. Faster and better: A machine learning approach to corner detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence, Pattern Analysis and Machine Intelligence, IEEE Transactions on, IEEE Trans. Pattern Anal. Mach. Intell*, 32(1):105 – 119, 2010.

[26] Thomas Harvey Rowan. Functional stability analysis of numerical algorithms. Technical report, 1990.

[27] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. *2011 International Conference on Computer Vision, Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564 – 2571, 2011.

[28] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[29] Bruno O. Shubert. A sequential method seeking the global maximum of a function. *SIAM Journal on Numerical Analysis*, 9(3):379 – 388, 1972.

[30] Richard Szeliski. Computer vision: Algorithms and applications. 2nd ed. working draft, 2021.

[31] H. Szu and R. Hartley. Fast simulated annealing. *Physics Letters A*, 122(3):157 – 162, 1987.

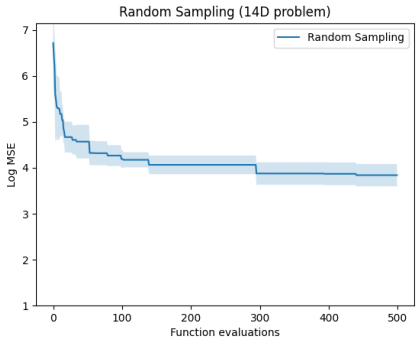
[32] C. Tsallis and D.A. Stariolo. Generalized simulated annealing. *Physica A*, 233(1):395 – 406, 1996.

[33] Xiang Yang, Sylvain Gubian, Brian Suomela, and Julia Hoeng. Generalized simulated annealing for global optimization: The gensa package. *R Journal*, 5(1):13 – 28, 2013.

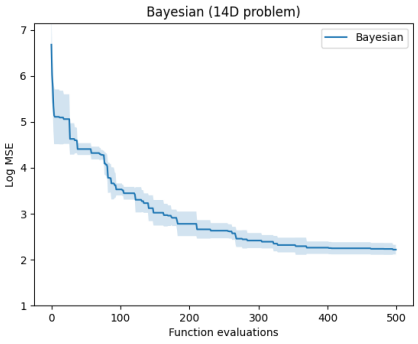
Appendices

Appendix A
Individual plots with standard deviation

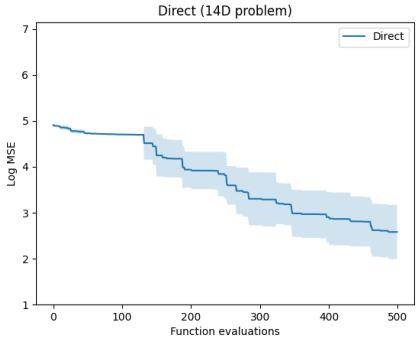
Individual plots for each optimizer for the 14 and 71 parameters are presented on the following pages. The first page shows the plots for 14 parameters, and the second page shows the plots for 71 parameters. All of these plots are generated for the Lobby scene. Each plot shows the mean plus and minus standard deviation for one optimizer.



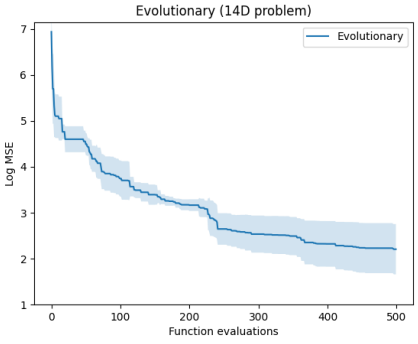
(a) Random Sampling



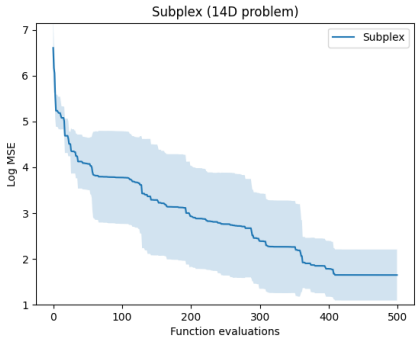
(b) Bayesian



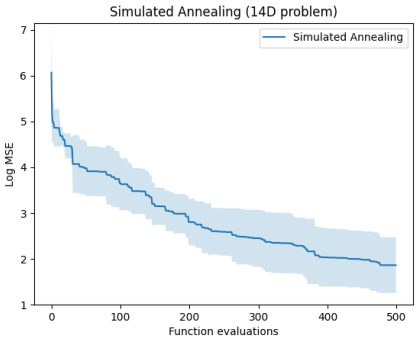
(c) Direct



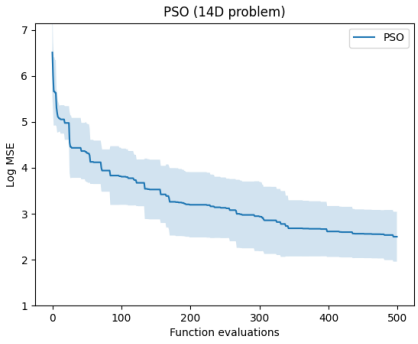
(d) Evolutionary



(e) Subplex

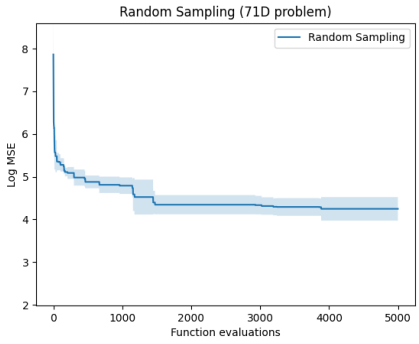


(f) Simulated Annealing

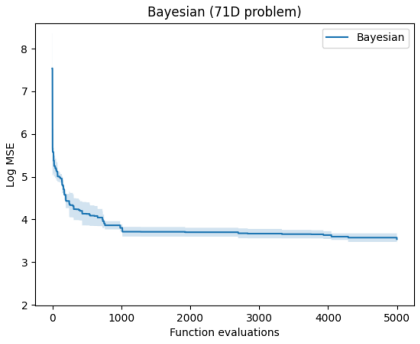


(g) PSO

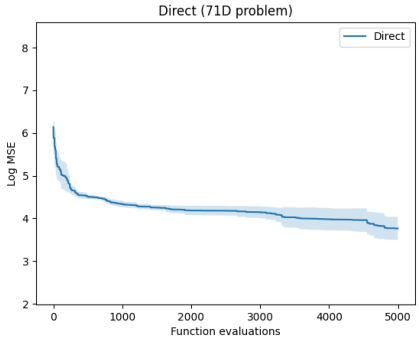
Figure A.1: Main Scene: Lobby with 14 parameters. Mean value and mean ± standard deviation of the MSE for each optimizer at each iteration up to 500. Note that Y-axis is logarithmic.



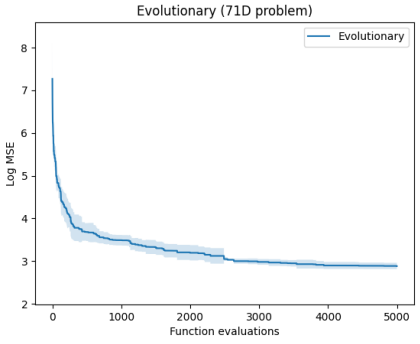
(a) Random Sampling



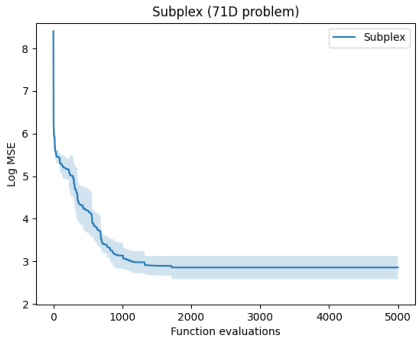
(b) Bayesian



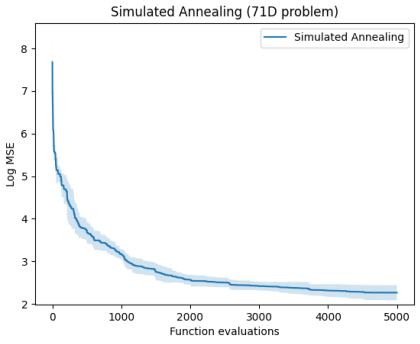
(c) Direct



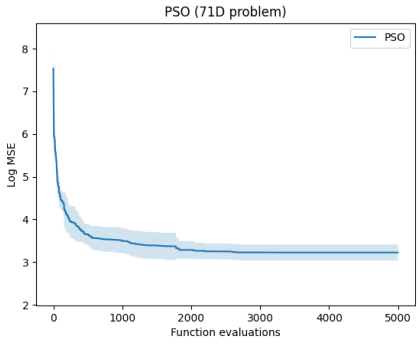
(d) Evolutionary



(e) Subplex



(f) Simulated Annealing



(g) PSO

Figure A.2: Main Scene: Lobby with 71 parameters. Mean value and mean $\pm$ standard deviation of the MSE for each optimizer at each iteration up to 5000. Note that Y-axis is logarithmic.

Appendix B
Popular Science Article

The next page shows the popular science article written in the process of this master thesis.

DEPARTMENT OF COMPUTER SCIENCE | LTH - LUND UNIVERSITY | PRESENTED 2021-06-21

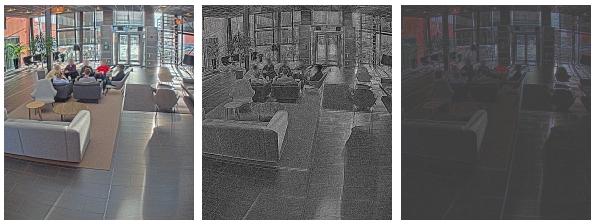
MASTER THESIS Self-Optimization of Camera Hardware
STUDENTS Simon Kristofferson Lind, Johannes Tykesson
SUPERVISORS Luigi Nardi (LTH), Waqar Hameed (Axis Communications)
EXAMINER Volker Krueger (LTH)

Automagically tuning camera quality

POPULAR SCIENCE SUMMARY **Simon Kristofferson Lind, Johannes Tykesson**

When new cameras are developed, their image quality has to be tuned by expert engineers, which normally takes several weeks. Our thesis shows that tuning can be done automatically in a matter of minutes.

Most cameras today come with a large number of settings that can change the image quality in many ways. Examples of such parameters are *contrast* and *saturation*. Our thesis was done at Axis Communications, and our work focused on parameters in the camera's ISP chip. These ISP chips usually contain hundreds of individual parameters that all have different effects on the image quality. Some effects are shown below:



Traditionally, all these parameters are manually tuned by expert imaging engineers who often spend several weeks to tune a single camera. Therefore, our thesis aims to automate this tuning process.

First, we needed to be able to tell if an image from the camera is good or bad. In order to do that, we compared it to a *reference* image. In practice, this reference image comes from a camera that has already been tuned.

Next, we implemented several optimization algorithms. These algorithms were then hooked directly to a camera, which allowed parameters to be tuned automatically.

Since hundreds of parameters is a massive task for most optimization algorithms, we started out small by experimenting with 14 parameters. Seeing that 14 parameters worked very well, we expanded to include the most important components in the ISP chip. In total, we tuned a staggering 71 parameters.

After running our tests, we were pleased to find that our tuned images were nearly identical to the reference image:

Can you tell which one is the reference image? Neither can we.

In conclusion, our results show that it is possible to automatically tune 71 parameters in under 15 minutes.

Tuning all these parameters automatically should save a lot of time for the human engineers.