Privacy-compliant non-intrusive monitoring of occupants' window behaviour using infrared thermography

BSc Project

Project report

Grp. 620, IN6

AALBORG UNIVERSITY

STUDENT REPORT

Aalborg University
Internet Technologies and Computer Systems

Department of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg Øst
http://es.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Privacy-compliant non-intrusive monitoring of occupants' window behaviour using infrared thermography

Theme:
BSc Project
(Information Processing Systems)

Project Period:
Spring Semester 2020
(3. February 2020 - 27. May 2020)

Project Group:
Grp. 620, IN6

Participant(s):
Benjamin Bach Jensen
Hyunho Shin
Mikkel Steen Hansen

Supervisor(s):
Rikke Gade

Copies: 1

Page Numbers: 85

Date of Completion:
February 25, 2021

Abstract:

This paper presents a privacy-compliant non-intrusive method to conduct research regarding occupants' window opening behaviour. Current solutions and problems regarding the state of the art research methods within the field, are analysed and discussed. The analysis conclusively ends with presenting a problem statement specifying a focus on developing a privacy-compliant non-intrusive way of monitoring, not only the binary state of window opening, but an arbitrary scale for openness, using thermal imagery. With this problem statement in mind, a technical analysis is conducted, examining vision pipeline design, image processing methods and classification. As a result of this analysis a window detection pipeline and state classifier is designed and implemented. The functionality of the window state classifier however, suffers from a poor dataset. The paper concludes a serious effort needs to be put into the creation of a thermal image dataset of building facades, in order to fully realise window detection and classification.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Table of Contents

Chapter 1

Introduction

Throughout history, humans have striven to improve their living conditions in a plethora of ways. Plentiful and varied food sources, safety, warmth and comfort are, to mention a few, needs that humans have tried to provide for themselves, but also for society as a whole. This, has to a large extent happened in, at least some parts of, the world. Buildings have been a large part of this progress with new materials and design concepts allowing buildings to help humans grow as a whole. Of course this process was very complicated and involved many other factors, the take away is that some very basic needs have been progressively better taken care off through continued development. Today, this has changed such that in the industrialised world, the concern is changing from providing for our needs today, to being able to provide for tomorrow. Climate change itself seems undeniable at this point, but grasping the full extent of the consequences of climate change? This is a very different problem. Hence, instead of trying to foresee all the problems that climate change will bring or being indifferent about the issue, people might in general be better off trying to avoid it.

This brings us to the current problems of our civilisation. When it became apparent that certain resources would eventually run out and that many technological solution are causing climate change, we found ourselves facing the largest challenge yet. How can we solve all of these problems again, but in sustainable ways? The first idea is to limit the impact of our current way of life. One part of this is energy efficiency. To that end "smartification", that is using computers to optimise the control of systems, of many different types of systems has tried to increase the efficiency of these systems as well as conserving resources at the same time. The area within which this report will focus, is buildings.

Building improvement used to be about making buildings last longer through rougher weather and making them easier to heat. Eventually, the average building was so well built, that only small amounts of air could enter and or escape the building. This was a good thing for the heating budget of a building, but it caused a problem with air quality inside the building. Now it suddenly became a necessity to actively exchange air with

the outside of the building. This goes against the wish for lower heating costs. The easiest solution to this problem was to open windows or doors, to passively ventilate the building. The problem is, that one might not notice the need for ventilation at all, and in that case the indoor climate would get quite bad. To manage this process without the use of windows, ventilation systems were made. This should have been the final solution to the problem, but even with modern ventilation systems problems can arise. One might imagine a room being too cold from too much ventilation or it being too hot from too little ventilation. So in one room the inhabitants are cold, and might try to block the ventilation physically, and in the other room the windows and doors are open. On top of this, the ventilation system might be inactive because of some actions by the inhabitants, like opening a window, that they are unaware of being the cause of the problem.

With all of this in mind it seems reasonable that the actual energy consumption of a building can be influenced a fair bit by the inhabitants of a given building. This was shown to be the case in a paper on occupants‘ window opening behaviour[1]. This also begs the question, if energy consumption of a building is heavily dependent on the inhabitants of the building, how could it then be modelled? A mathematical model for this problem is needed if new buildings are ever to be truly optimised in their energy consumption.

To that end studies have been done to collect data on inhabitants, but the methods employed, involving manually installing sensors in each window, have been time consuming and costly. One study, non-intrusive measurement methods[2], tried to use a camera and computer vision to observe many windows simultaneously. While they were successful in detecting open windows, they noted that there were certain privacy concerns with pointing a camera at a building facade.

Given how much more difficult it is to identify people on thermal images, it was considered whether using a thermal camera for such non-intrusive measurement would be feasible.

Research question:
How can we improve window opening behaviour data gathering using thermal imagery?

Chapter 2

Problem analysis

This chapter will firstly look into what is of importance when dealing with window opening behaviour, and a review of the issue. Then the current state of the art will be introduced with the focus on using cameras, moreover, it will cover each camera's pros and cons, as well. After that a further look into the properties of windows, specifically windows seen from an infrared camera. Lastly, focal points are summed up and a list of requirements and a problem statement is presented.

2.1 The importance of window opening monitoring

In modern buildings, evermore sections are getting automated, areas like: lighting, ventilation, heating etc. This trend is a result of a constant strive to achieve maximum efficiency on a number of different parameters like energy, human comfort and ease of use and management etc. Windows are one of the parameters still controlled mainly by occupants based on behavioural and context of the given occupant. The natural ventilation caused by opened windows, can affect a building's energy usage considerably, as Branco et al.[3] found in their study. The study noted in their experiment that true energy usage was 50% higher than what was estimated. This may suggest that occupants are somewhat unpredictable variables as numerous studies conclude that occupants play a critical role in a building's energy usage, as well as the variance of it[1]. Therefore it is of great interest to determine the factors that makes occupants open windows, and use that data to create even more realistic simulation models.

The study of occupant's window opening behaviour has resulted in a plethora of different reports with varying methods and results. In the paper: "Occupants' window opening behaviour: A literature review of factors influencing occupant behaviour and models"[1], a significant amount of papers on the subject matter were reviewed and concluded upon. According to the paper, existing studies focus on the window's (binary) state, instead of the transition between the two states, as stated in the excerpt below.

> "...Moreover, existing studies on window opening behaviour are aimed at investigating the state of the window itself instead of the transition from one state to another (opening and closing). This might be problematic, since the indoor environment is affected by the state of the window with the consequence that the predictive variables are influenced by the state that they are trying to predict..."
>
> *- An excerpt of the conclusion of the paper "Occupants' window opening behaviour: A literature review of factors influencing occupant behaviour and models"[1].*

The transition of state may be an interesting parameter in the constant strive for new and improved simulation models for the building industry.

This was echoed by Andersen RV. in his master's thesis about occupants behaviour in indoor environments.[4] Andersen also stated the value of measuring the degree or tilt of opening. This variable should be considered when creating realistic simulation of indoor climates, especially considering that not every window is created or operated equally. Drivers (general factors influencing occupants behaviour) and the overall context might change how much a window is opened.

One could even imagine that occupants with primarily psychological drives (e.g. concerned with odour) and primarily physical drives (e.g. environmental concerns) have noticeably different window opening behaviours. In the same fashion that windows are not all similar, situations are not either.

To get a better idea about what could improve current methods of window opening behaviour monitoring, the current solutions for data gathering is explored.

2.2 Current solutions for window monitoring

In this section the current solutions for window state monitoring will be investigated. Firstly, the window mounted sensors will be discussed followed by another section on camera based solutions. At the end of the camera section a comparison of the methods will be given.

2.2.1 Window sensors

(a) *A window state sensor; A magnet and a reed switch. Image courtesy of Pezcame[5].*

(b) *Window position sensor. Image courtesy of R. Andersen (2013)[6].*

Figure 2.1 *Two types of window sensors.*

Window sensors are commonly used when it comes to monitoring natural ventilation of windows for buildings, especially for modern energy-efficient buildings. Furthermore, these sensors are used in the rising home security field with many companies chiming in[7, 8, 9]. The home security field is a part of the smart home trend, that allows private individuals to deploy their own wireless sensor clusters without the need of technicians.

The window sensor uses a magnet and reed switch (mechanical switches that activates using magnetic fields)[10] and are typically installed by placing the main sensor (reed switch) in the window frame, and placing the magnet on the window pane, as seen in fig. 2.1a.[11]
In a window opening behaviour study conducted in 2013,[6] the windows subject to research were in addition to having window state sensors, also equipped with window position sensors, as seen in fig. 2.1b. These sensors are three axis accelerometers that allows for measuring the swivel or tilt of a window. This approach does however require an extra sensor for each window where the tilt/swivel is wanted. These sensors cost about 80$ each.[12]

2.2.2 Camera

Having discussed window mounted sensors, a non intrusive option based on cameras and image recognition is discussed. In contrast to using window mounted sensors, a single camera could be used to photograph a facade, and through these images detect the open windows.

When using window mounted sensors, there are no problems with privacy concerns, because the wanted data is captured directly. This goes for both the binary window behaviour but also the tilt/swivel angles. However, the cost of the sensors and the man-power to install them are significant. Ideally, using a camera based detection would be cheaper and faster to deploy. At least when the number of windows on a facade is significant, such as on apartment buildings.

Detecting window states with a camera requires more complex software than an array of window sensors would. It would be required to capture the image, do the necessary image processing and finally using a classifier based on machine learning.

That was done in a paper on non-intrusive measurement methods [2]. Here the authors applied image recognition, as seen in facial recognition, to the detection of window opening and estimation of the degree of each window's opening. The authors used a regular camera to capture the facade and a transformation of the image to make each window the same size and shape (the windows were distorted by the angle with which the image was taken). It is noted in the paper that the intensities of closed and open windows differ significantly. The pixel intensity is therefore used as the basis for the recognition algorithm.

This process naturally requires much more processing power than simply reading values from some sensors. It should be noted though, that capturing raw images like these could potentially be used for many other things, than just window opening behaviour monitoring. This could be a problem, if the images were to be processed on an off site server, then these images could potentially be compromised and misused. On the other hand, on site processing puts constraints on the amount of processing power cheaply available. If more processing power is needed, the cost of local processing will go up. The cost of central processing would in contrast scale less aggressively.

As the stated in the introduction to the report, it is believed that this camera based approach could be done with thermal cameras instead. This is interesting to investigate, as thermal images are harder to use in identifying specific individuals and would therefore arguably pose less of a privacy concern. Moreover, if it will be needed to use off site processing of the images, as seen in table 2.1, this would be less of a problem than with regular imagery.
To be able to use thermal cameras in window opening detection, it will be necessary for the facade of the building to have different temperatures. Fortunately, there seems to

be a temperature difference between the both the building facade and the window frame as well as between the window frame and the window itself. However, one interesting thing to note, is that glass is reflective to infrared light. This means that any image of a window can be influenced by the temperature of the reflection, which could come from many different places, given the possible angles.
It should also be noted though, that to detect the window a difference in temperature is what is needed, not any particular set of temperatures. If the temperature of a reflection in a window exactly matches the temperature of the window frame, a thermal camera would not be able to tell them apart. But the likelihood that a random reflection in a window matches the temperature of the window frame is low.

To sum up the pros and cons of the discussed methods table 2.1 is presented. Here the four discussed approaches are compared. The window mounted sensors are expensive to buy and install especially if one considers large facades with many windows. It is also noted that, unlike the camera based approaches, this method scales about linearly in cost and installation time with the number of windows to be observed. However, the data gathered from the sensors is exactly the wanted data and therefore it is the most direct way to obtain the wanted data.

The camera based methods will capture more data than what is needed. This creates the issue of privacy, as the captured images could in principle be misused. This concern created the two different camera approaches, the locally processed and the centrally processed. The locally processed would be some smaller computer packaged with the camera to process the images and finally output the data to local storage or the internet. This data would closely resemble the data that the window mounted sensors would output. This would greatly limit the privacy concern of having cameras observing a facade.

But this approach assumes that the recognition model with a sufficient accuracy is able to run, fast enough, on a local device. The higher the required accuracy and the faster it has to run, the more expensive will the computer required be. This could be a problem if the proposed recognition model becomes very calculation intensive. If this is the case, the central processing approach may be necessary, as it would be much cheaper to buy node hours, than buying the necessary hardware and placing it with the camera. That approach of course makes the privacy concerns more important, as the images are much more likely to fall into other hands, when sent over the internet.

Finally, the thermal camera approach is similar to the two regular camera approaches (as the thermal images could also be processed either locally or off site), but in either case, it is argued that the privacy concerns are minimised. It should however, be mentioned that reflections can cause problems. Moreover, high resolution thermal cameras are much more expensive than regular cameras.

Table 2.1 *List of pros and cons to compare the five approaches. Here the window mounted sensors represent the conventional approach, the Camera(local and central) represent approaches similar to what was described in a paper on non-intrusive measurement methods [2] and Thermal(local and central) represent approaches similar to the "Camera", but with a thermal camera instead.*

	Mounted sensors	Camera (local)	Camera (central)	Thermal (local)	Thermal (central)
Units	Multiple	Single	Single	Single	Single
Price pr. unit	Low	Medium	Medium	Expensive[1]	Expensive[1]
Installation cost	Expensive[2]	Low	Low	Low	Low
Processing cost	No[3]	Expensive	Medium	Expensive	Medium
Privacy concerns	Minimal	Limited	Yes	Minimal	Limited
Portability	Fixed	Movable	Movable	Movable	Movable

[1] Expensive camera, if high resolution is needed. [2] Price scales linearly with the amount of windows. [3] The raw data is the wanted data.

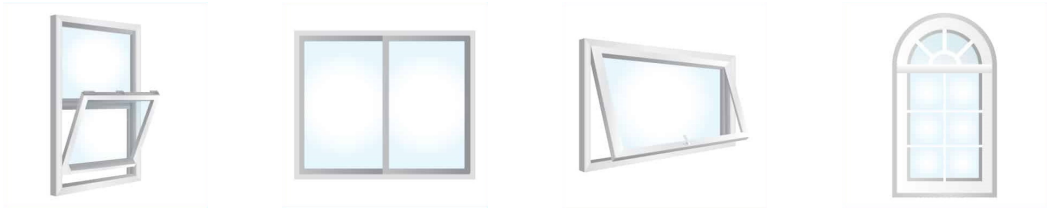
Here the current solutions were discussed and it was reasoned that a camera based one carries significant advantages. The greatest problem with a camera based approach was the potential privacy issues. Since people are more easily identified in the visible spectrum than in the infrared, the potential of thermal cameras is considered to be worth investigating. In the next section, the window study will be presented, from the perspective of using thermal cameras.

2.3 Window study

In this section an investigation of windows will be done. The types of windows of interest will be determined and the important characteristics of the windows will be considered.

2.3.1 The state of a window

To be able to discuss the state of a window it will be necessary to determine what a window looks like and how it opens. Unfortunately, a lot of different windows types can be found, as can be seen in fig. 2.2.



(a) Single Hung. *(b) Two-Panel Slider.* *(c) Awning.* *(d) Arched.*

Figure 2.2 *4 different types of windows. Images courtesy of Home Stratosphere[13].*

fig. 2.2 shows the single hung window in fig. 2.2a which is fairly good for air exchange in warm climates, fig. 2.2b the two panel slider which is good for air exchange in all climates, as the height of the open window allows for a larger pressure difference. The awning window in fig. 2.2c does not provide a lot of air exchange and lastly, the arched window in fig. 2.2d is an example of a window that does not open at all as well as being comprised of a few different shapes.
However, for the purpose of this study not all windows need to be examined. The study will focus on some of the windows that are accessible for testing. Specifically, a type of window that can open in two ways. Firstly, it can swivel inwards (from the left or the right) such that the window can open from a few degrees up to 90° degrees. Secondly, it can tilt inwards at about 20° degrees.

These modes of opening affect the amount of airflow and reflection differently. When the window swivels, the airflow can be much higher that if it tilts. It also causes the reflection to be come from a different direction. Assuming that the camera and the window are at the same height, the reflection would come from either the right or the left.

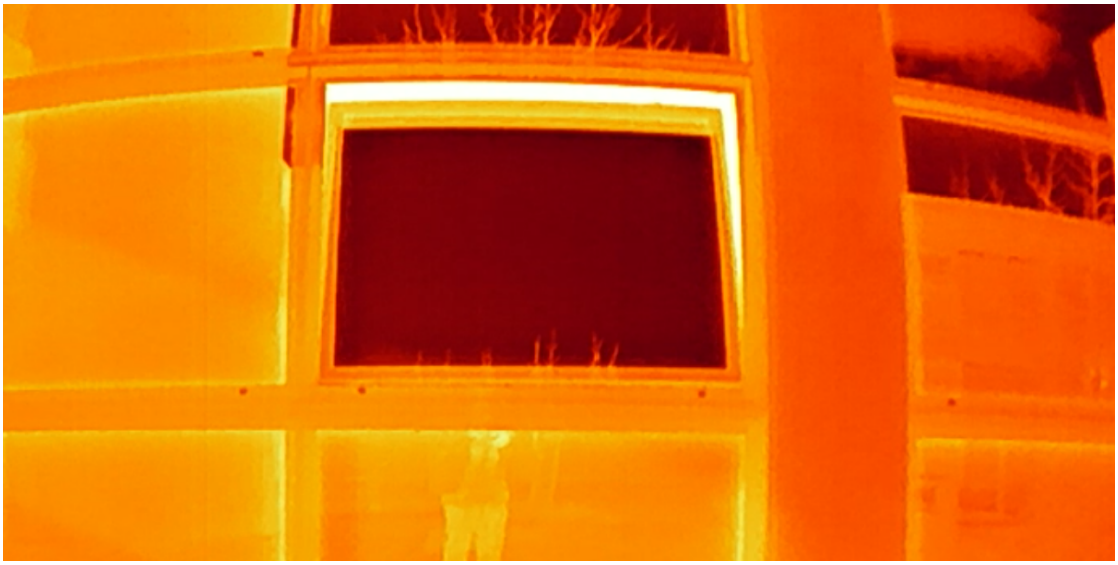


Figure 2.3 *Window reflection of the cold sky.*

In the case of a tilted window, the reflection would come from either below or above the window. Since the windows in question can tilt inwards, the reflections would come from above the window. This would in many cases be a reflection of the sky. Unless the sun shines directly on the window, such that it is reflected into the camera, the sky is rather cold [See fig. 2.3].
However, if the reflection comes from a building facade, e.g. when a tall building is behind the camera or the window swivels, then the amount of infrared radiation can be much higher than that of the sky.

Temperature

Moreover, if the window swivels inwards into the room, then one might also get reflections from inside the building [See fig. 2.4 #4]. Clearly, in conditions where the outside is colder than the inside, these reflections would be more intense that reflections from the outside. But assuming that a very warm reflection equals a very large angle of opening, is also assuming that these conditions will always be true. One could imagine that a room could approach the temperature of the outside, if the window had been open for long enough. Therefore, the inside of a room might not always be much warmer than the outside.
Naturally these conditions depend on the season and the geography of the setup, but in general it might not be possible to assume how the reflections of a window will look.
The point here being, that the amount of IR radiation coming from a window, whether it is open or not, can be very different depending on these conditions.
Therefore, general assumptions should probably not be made about the radiation being reflected by the glass.

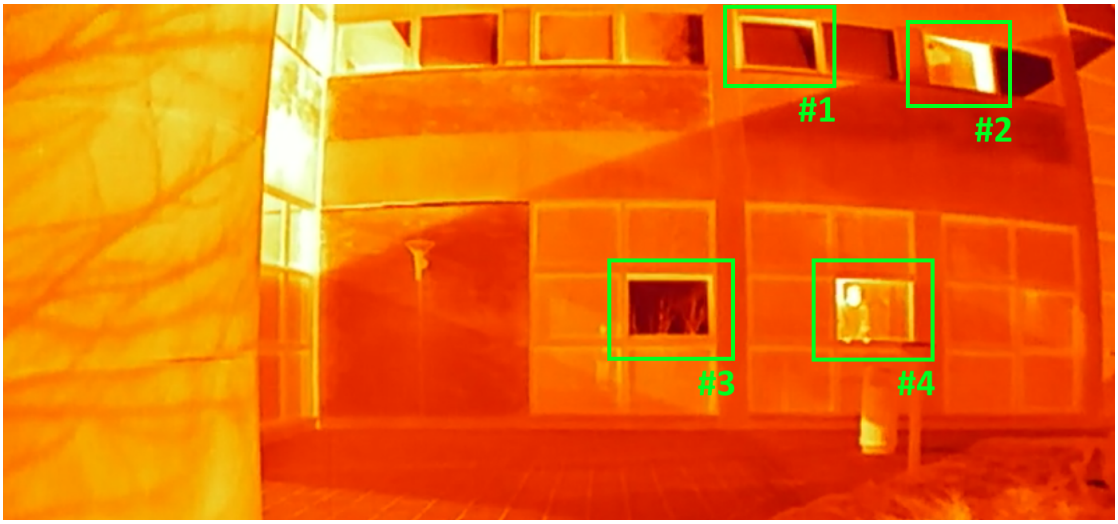


Figure 2.4 *Facade with 4 open windows: 1. Slightly opened inwards from the top. 2. Opened inwards from the right, the inside of the room stands in clear contrast. 3. Slightly opened inwards from the top, here the reflections are especially noticeable. 4. Fully opened window with a person standing in the window frame.*

Time

While a newly opened window, swivelling or tilting, might be dominated by reflections, a window that has been open for an hour might have the outer layer of glass heated up, or cooled down, such that it is no longer dominated by the reflections of the window but instead its own temperature.

What will be present in most cases is a difference in temperature between the window frame, the glass of the window and the inside of a building.

2.3.2 Environment surrounding the window

While the windows themselves have some properties relevant to detection of them, other parts of the building also have such properties. One example of this is the facades of buildings. In the visible spectrum facades can have many different colours and intensities. In the IR spectrum facades can also have different intensities, which here represents temperature. These temperature differences can be due to varying degrees of insulation or even something like exhausts from ventilation systems. In essence, there are potentially many unseen features of building facades that could interfere with window detection.
The weather conditions also play a part in how a facade looks in the IR spectrum. One obvious thing is sunlight on the facade. An example of this can be seen in fig. 2.4, where a line from the upper right corner meets the windows on the left, creating a new line from the window to the ground. While these lines show up as differences in temperature, they are also apparent in the visible spectrum.

In regions with snow fall facades can be affected by the reflections from snow in front of the facade, but also if snow was to be on the facade itself it could also interfere with the temperature that would otherwise be shown. In fact, the reflectivity of freshly fallen snow can approach 90%[14].
Given that heat is important to how the windows, and facades, look it is assumed that the airflow, and the temperature of the air, is important for how an open window would look. In this regard curtains and radiators might affect both affect the airflow, and therefore also the temperature of the air.

Now that the window study has been presented, a partial conclusion summing up the discussions will be presented.

2.4 Partial conclusion

Based on the findings, using a camera for window detection seems to be advantageous with respect to cost and time. It does however, also seem to pose some challenges regarding privacy and reflections. Going from a regular camera to a thermal one would lessen the privacy concerns while still facing the reflection challenges.

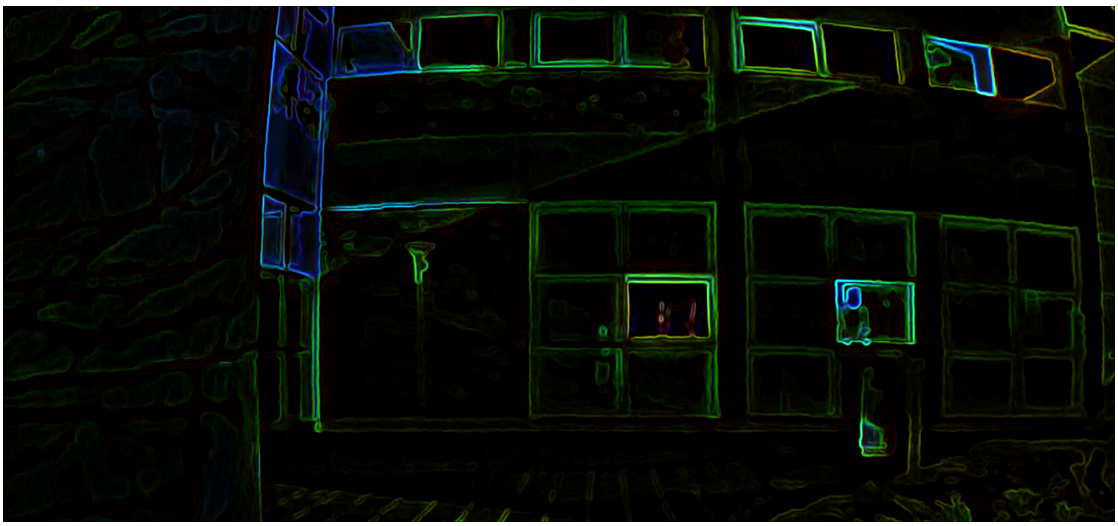


Figure 2.5 *This image was made by subtracting the mean blurred image from the original, followed by the default edge detection algorithm in ImageJ*.*

However, if these reflection challenges can be overcome it seems to be an overall better solution. Moreover, if additional processing power is needed, sending thermal images to central processing would be less of a privacy concern than with normal images. Since detecting small openings is assumed to be fairly difficult with regular cameras, it is noted

*ImageJ is an open source image processing tool useful for prototyping an image processing algorithm, before implementation using OpenCV[15].

that using thermal cameras might create a more easily identifiable feature. This feature is expected to be a sliver of heat coming from the inside of the room, as seen in fig. 2.5.

2.5 Problem statement

How can a thermal camera be used in developing a privacy-compliant non-intrusive window position monitoring system?

2.6 Requirements

Req 1 The system installation time should scale less than linearly (preferably be constant) with the number of windows observed (unlike the installation of window sensors)

Req 2 The system should be able to identify several windows in a single image

Req 3 The system should be designed to reduce privacy concerns

- The system should not output imagery to the internet

Req 4 The system should, for each window:

- Output open/closed state
- Output openness [arbitrary scale (e.g. full, on gap, locked position) or angle]
- Window ID

Req 5 System performance

(a) Performance of window detection
- Minimum level of precision 80%
- Minimum level of recall of 95%

(b) Performance of window state classification
- Minimum level of accuracy of 90%
- Minimum level of precision of 95%
- Minimum level of recall of 80%

(c) Performance of window position estimation
- The form of this is to be determined

2.6.1 Explanation of requirements

In **Req 5**, the system performance is split into three parts: the window detection, the window state classification and the window position estimation. Here, detection and classification are differentiated between by the fact that detection involves locating the object in an image, whereas classification is only about giving the right output based on the object input. The window state classification and the window position estimation differ in that the state classification is a classification problem with two possible classes, open and closed, and the state estimation is a regression problem, in that the output is a value in between a maximum and a minimum. These extremes represent fully opened and fully closed windows.

Firstly, the performance of the window detection. To quantify this performance, two measurements will be used: Precision and Recall. The precision is meant as the rate of TP (**T**rue **P**ositives) divided by the sum of TP and FP (**F**alse **P**ositives).

$$Precision = \frac{TP}{TP \ + \ FP} \tag{2.1}$$

This measurement describes how many of the detection positives are true positives. Therefore, this measurement quantifies the degree to which the classifier includes too many objects into the positives class.

The recall measure is given here:

$$Recall = \frac{TP}{TP \ + \ FN} \tag{2.2}$$

It describes the ratio of classified positives to the total number of true positives, classified and not classified (**F**alse **N**egatives). Therefore, it speaks to the degree with which the classifier is being too restrictive about what is classified as a positive.

Secondly, the performance of the window state classification. To quantify this performance, three measurements will be used: Accuracy, Precision and Recall. Since the last two are defined above, the accuracy measurement will be given here:

$$Accuracy = \frac{TP \ + \ TN}{Number \ of \ samples} \tag{2.3}$$

This represents a measure of how well the classifier identifies objects correctly as a class (TP) or correctly as no class at all (**T**rue **N**egatives). A performance measurement for the window state detection of 92% was given in the non-intrusive measurement methods paper[2], although it was not specified what kind of accuracy it was. Therefore, it is merely assumed that the accuracy rate given is of the form given in eq. (2.3).

Thirdly, is the performance of the non-binary state detection. How this performance is calculated depends on the nature of the output from the classifier. If the outputs

are continuous, then a simple error rate might be useful. If the outputs are discrete in nature, then it might be necessary to define an error measurement specifically for that. The nature of the output however, still remains to be determined.

Regarding the specific performance numbers, they are chosen based on the idea that these percentages would be sufficient to show that the concept is usable, but it should also be achievable during the project. Furthermore, it is expected that these percentages can be greatly improved, if the appropriate time and effort was put into further developing the technique, and improving the training data. The reason why the recall and precision numbers are different for the two parts of the pipeline, is that in the detection part of the pipeline, it is preferred to have false positives rather than false negatives. This is due to the fact that if a window is not detected in the first place, then the system will not be able to obtain any data at all about the window. However, if the second part of the pipeline was given false positives, then it would need to be able to reject these and only give actual output for the true positives.

So with this setup it is envisioned that the window detection will detect as many windows as possible, while the second part will be able to eliminate most of the false positives. In this way it is hoped that the system will have an overall better performance than if the recall and precision targets had been equal for both parts of the pipeline.

Chapter 3

Technical analysis

This chapter will introduce computer vision and the steps of image processing of thermal images, and some algorithms for that. Then the needs of the project will be discussed on a high level. A discussion of how normal and thermal cameras work respectively, while focusing on why this project aims to use thermal cameras, will be covered as well. One of this project's goals is making the connection between the level of how much the window is tilted and the temperature of the window by computer vision. Thus, after applying the image processing, the computer should recognise the image, and then, we can use the machine learning algorithm so that it can make the connection itself. A discussion on some classifiers is given such that a fitting classifier can be chosen for this project.

3.1 Introduction to the computer vision pipeline

It may not come as a surprise, but a computer's "brain" functions differently than a human brain. One area where this is especially apparent, is in the creation of artificial vision. A general framework of an artificial vision system (See fig. 3.1), takes a digital image through a number of steps and ending out with a "guess".

Figure 3.1 *Diagram showing the general framework of a general vision system.*

The individual steps in the artificial vision framework will be walked through in general terms, to better understand the propose each step serves.

3.1.1 Image acquisition

First of all, the actual input data must be acquired. This is done using a digital camera, in this project a thermal camera.
While humans, break down (digital) images by motives and scenery using our perception of colour and objects, a computer interprets in terms of pixels. Pixels are a form of data representation of numeral values or an array of numeral values, that holds information about a single dot colour. An image is simply a matrix of all these pixels. A grey-scaled image consists of an intensity integer value between 0 and 255 (8 bits), while coloured image typically use RGB, that is an array of intensity channels for **R**ed, **G**reen and **B**lue respectively. A 2D matrix is used to represent grey-scale (single channel) images and a 3D matrix for coloured (3 channel) images.
The thermal camera outputs a single channel representing temperature intensity in a given temperature span.

3.1.2 Pre-processing

After an input image is acquired, then the image is subject to pre-processing. Pre-pocessing of images in a vision system is done to reduce or preferably remove all unwanted noise and details from a given image. Noise and unwanted details are removed to help the segmentation step extract data about the object(s) in question.
Image processing is facilitated using tools like: blurring filters, edge detection and histogram processing.

3.1.3 Segmentation

The segmentation can be viewed as an extension of the pre-processing step, in that this process further reduces noise. The reduction of noise in this step is done by binarising the image in a way such that, the subject matter is represented as groups of connected white (object) pixels. This is known as BLOB's (**B**inary **L**arge **Ob**ject) and is the end goal of the segmentation process.
Generating the binarised image is handled by the use of thresholding, while further manipulation of the binary image can be achieved with methods like: Dilation and Erosion, which makes BLOB's bigger and smaller respectively.

3.1.4 Representation

The representation step generates the input, that the classifier uses to classify. Here the BLOB's from the segmentation step come in handy, because it is through analysing these BLOB's, that characteristics about the subject are extracted. These characteristics are called: "features", and a set of them: a "feature vector". Feature vectors are representations of certain BLOB properties, for example size, shape or positions etc.

3.1.5 Classification

The last step of the general computer vision pipeline is the classification step. Here an output is decided based on the one of many different classifiers. There's the decision boundary, k-nearest neighbour classifier, Bayes classifier to name a few. Common for all of them is that their "world view" is based on a set of predefined labels. The classifier will then make a "guess" or estimation about the input data. This "guess" is based on a classification system's estimation towards a certain label of the set of labels.

3.2 The project's pipeline

Having discussed the recognition pipeline as a whole in theory in the previous section, here the overall plan for the next sections will be laid out. The specific structure of the proposed pipeline will also be discussed.
While the purpose of the pipeline is to output whether a window is open or closed and if open, then how open it is, this might require a few steps. One way of doing this is to first off detect the windows themselves. This is done through some BLOB (**B**inary **L**arge **Ob**ject) analysis of the raw image, which contains any number of windows. Each of these potential windows must then be put through the classifier and the BLOBs classified as windows, can be used in the second part. This initial part of the pipeline is illustrated on fig. 3.2.

The representation of these windows would then be put into the second classifier that would give the binary window state output. One might think, that it would also be possible to put the same representation into a third classifier to let this one give the continuous state output.

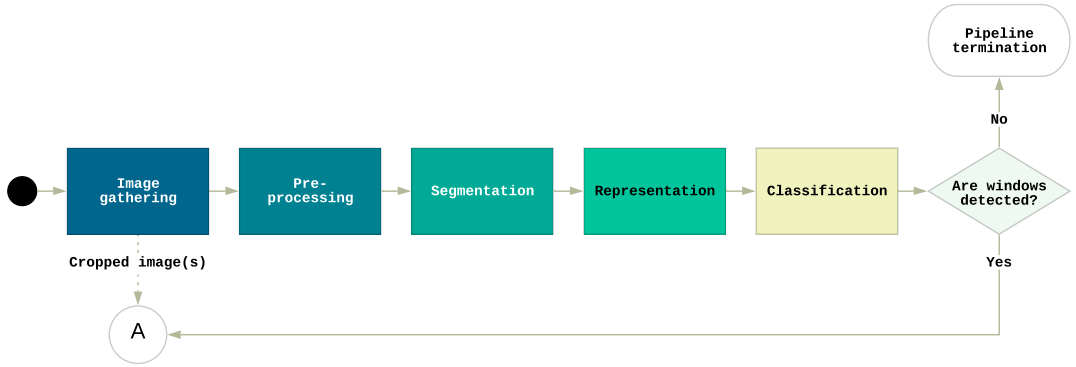


Figure 3.2 *First part of the proposed pipeline. This pipeline follow the general vision structure closely. The classification step outputs $n \times successful$ window detections. The representation or feature vectors of these windows will go through the second phase.*

However, this is all assuming that the representation of the windows used for detecting

the windows is also appropriate for the other two classifiers. In principle each of the three classifiers might benefit from their own individual representations, which then are based on their own pre-processing and segmentation.

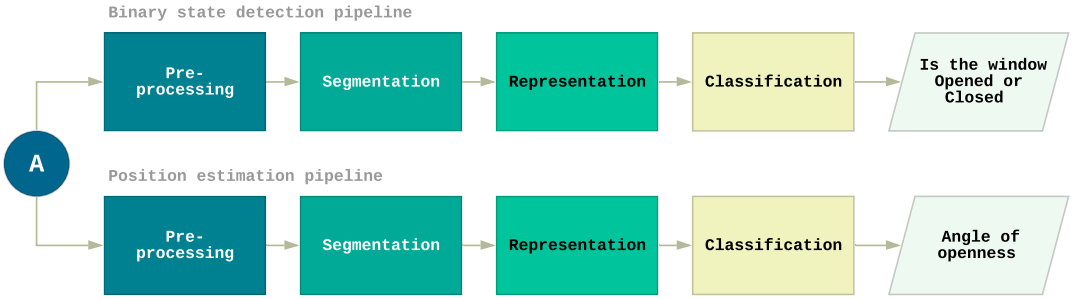


Figure 3.3 *Last part of the proposed pipeline. This pipeline receives a cropped window from the first phase as input. The pipeline then splits into two separate branches; top half is a pipeline for binary state classification, while the bottom half classifies openness.*

If this would be the case, then the input to the binary and continuous classifiers might simply be sets of coordinates that allows them to get the raw representation of the windows and do a different pre-processing and segmentation. Only after that, would the new representations be sent to the two state classifiers. This part of the pipeline is illustrated on fig. 3.3.

This pipeline structure is based on the assumption that a single classifier will not be able to do all of these three things on its own. Whether this is actually the case should be explored further.

3.3 Computer vision toolbox

Having discussed the proposed pipeline(s) for the project's recognition system, an analysis of various tools in vision frameworks, for the purpose of designing the aforementioned pipeline structure is presented here. The tools for each step of the vision pipeline will be explained in detail, with the intention to gather all necessary techniques to go from a raw thermal image to detection and classification of windows.

3.3.1 Thermal image gathering

> This section cites the research of "Thermal Cameras and Applications; A Survey" by Rikke Gade & Thomas B. Moeslund[16].

There are different kinds of light surrounding us all in our daily lives. Some of them are visible, others are invisible, and infrared radiation which thermal cameras use, is an example of invisible light. The thermal sensors available on the market acquire light in

the MWIR (**M**id-**w**avelength **i**nfra**r**ed from $3\mu m$ to $6\mu m$) and LWIR (**L**ong-**w**avelength **i**nfra**r**ed from $6\mu m$ to $15\mu m$) spectrum, because of atmospheric transmittance and the band of peak emissions. This high transmittance (or low absorption) leads to a good range on the camera. Similarly, if a human was to see an object in the distance, then this would be much easier if there was no fog, while heavy fog would make it impossible to see the object. These are examples of high transmittance (in clear weather) and low transmittance (in foggy weather). Another reason for choosing different ranges of wavelengths, is that higher temperatures correspond to shorter wavelengths. Therefore, if the objects in question are very cold, one would need sensors that function in the LWIR spectrum, were in contrast much warmer objects would need sensors in the MWIR spectrum to be visible to the camera. It should be noted, that this trend holds when objects heat up even more where, eventually, they will be glowing red, or even white, in the visible spectrum.

Difference between thermal and normal cameras

In cases of normal cameras, the light that passes through the optical lens and the aperture is converted into a digital signal by a CCD (**C**harge-**C**oupled **D**evice) or a CMOS (**C**omplementary **M**etal–**O**xide–**S**emiconductor) sensor in them. Thermal cameras are equipped with microbolometers that respond to infrared light instead of visible light. Infrared light that passes through the lens is converted into an electronic signal through an infrared sensor.
A regular camera needs visible light to be able to function. This is not the case for a thermal camera, as it only needs thermal radiation within the spectrum that its microbolometers are sensitive to. This is an advantage for a thermal camera as the object to be detected provides the light needed for it to be seen. However, the two different types of cameras also provide different features in the imagery. An example of this, is that people are much easier to identify with a normal camera than a thermal one. Conversely, a feature such as heat leaking from a building, in the form of air for example, would be hard to detect with a normal camera, but easy with a thermal one.

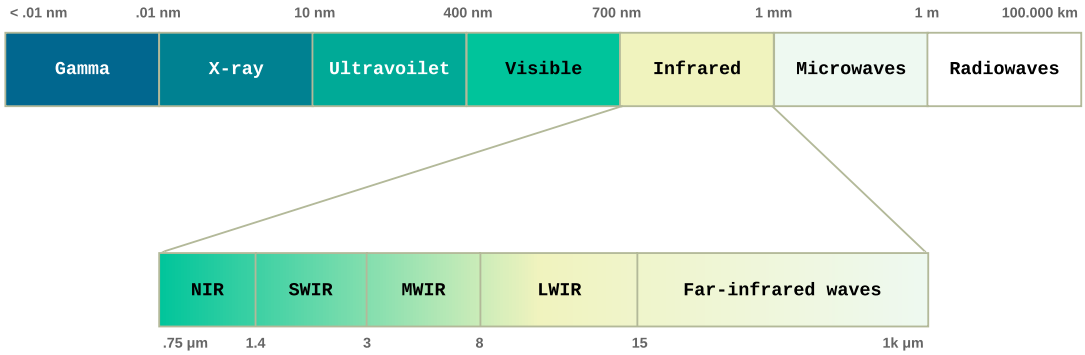


Figure 3.4 *The infrared spectrum.*

The importance of using thermal camera for this project

The fact that thermal cameras operate in a different spectrum that normal cameras mean that the images will contain different information. As an example, an image of a face taken with a normal camera contains information about the colour of the persons eyes, hair, skin and so forth. The same image taken with a thermal camera contains information about the absolute temperature of different parts of the face. This makes the image contain the shape of the face, assuming it is not the same temperature as the background, and the different temperature regions of the face. Since humans are not used to identifying each other based on these features, any face looks somewhat anonymous in the thermal spectrum, and it would therefore be much more difficult to extract personal data from a thermal image feed. One would of course still be able to tell whether someone is home at all, if they were to walk by an open window, but in general the use of such imagery would be more limited. So, for the purpose of window detection itself, thermal cameras are not preferred for the features they provide, but instead for the features that they do not provide.

In this section, some characteristics of light were discussed and a comparison of cameras working in two different spectra, the visible and the infrared, was presented. As the gathering of images is the first part of the pipeline, the next section will discuss the next part of the pipeline: Pre-processing & segmentation.

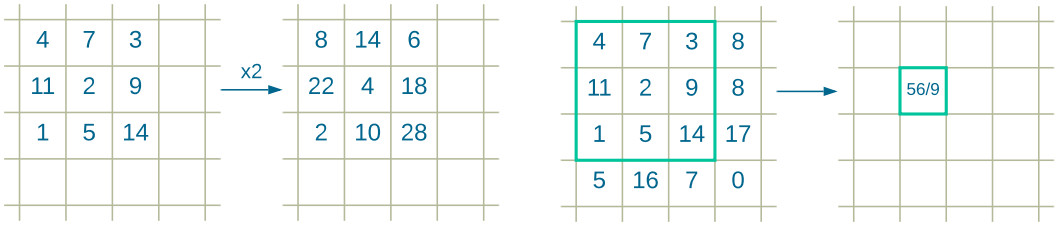
3.3.2 Pre-processing & segmentation techniques

Now that the raw image from the thermal camera is gathered, the next part of the pipeline process is to pre-process said image. In this section, various pre-processing and segmentation techniques will be presented, with the goal of organising a toolbox with relevant tools for the project's computer vision pipeline.

Firstly, a look at a fundamental technique when manipulating images, namely the concept of neighbourhood processing and correlation.

Neighbourhood processing & correlation

Neighbourhood processing is the natural evolution of the simpler point processing, where one input pixel value is transformed through some function into a output; a one to one point transformation [See fig. 3.5a]. The neighbourhood processing on the other hand takes several pixel values as the input and puts that data through some function, that gives a single new pixel value, as seen in fig. 3.5b.



(a) *This is an example of point processing. Each point in the pixel matrix is transformed individually (in this example they are doubled).*

(b) *This is an example of 2D neighbourhood processing. This example features a 3×3 kernel (From eq. (3.1)), that uses 9 pixel to create an output.*

Figure 3.5 *Examples of both point processing and neighbourhood processing.*

For the neighbourhood processing example in fig. 3.5b, a simple 3×3 normalised kernel [See eq. (3.1)] "scans" a given image one row at a time from left to right. This operation is called correlation. The chosen kernel for correlation can vary in size depending on the desired magnitude of the filter effect, but usually a 2 dimensional kernel is an $n \times n$ matrix where $n \geq 3$, and often an uneven number.

$$\frac{1}{Kernel\ height \cdot Kernel\ width} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.1}$$

The maths behind the correlation operation goes as follows:

$$g(x,y) = h(x,y) \circ f(x,y) = \sum_{i=-R}^{R} \sum_{j=-R}^{R} h(i,j) \cdot f(x+i, y+j) \tag{3.2}$$

Where:

- $g(x,y)$: is the output
- $f(x,y)$: is the input
- $h(x,y)$: is the filter
- $\circ$: is the correlation operation
- R: is the radius
- i: is the height of the radius
- j: is the width of the radius

These square correlation kernel can lead to a problem called the "border problem". This is were the correlation of kernels can't generate output around the boundaries of the image [See fig. 3.6]. This problem can be solved by the use of truncated kernels with arbitrary quadrilateral kernel sizes, for example $3 \times 3 \rightarrow 2 \times 3$.

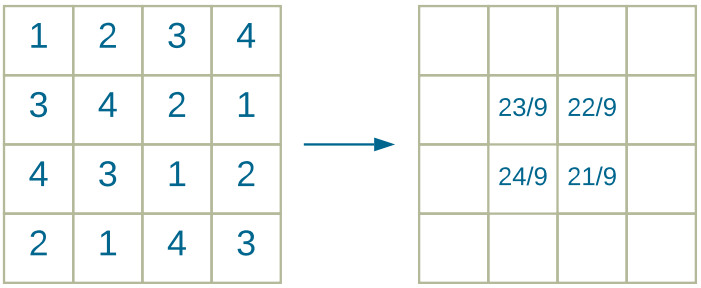


Figure 3.6 *With $n \times n$ kernels, there will often be points around the image's border where the kernel cannot generate an output. This example uses the 3×3 mean blur kernel (From eq. (3.1)).*

Another way to deal with the border problem, is to copy the value of the neighbour pixel (or the average of the two values, if considering a corner pixel). One could also wrap the image around. This means that a border pixel on the left would get the value of the outer most pixel from the opposite side of the image, as if the edges of the image touched its own sides. Which of these methods that yield the best results, is application dependent.

Now that the concept of correlation has been explored, lets now take a look at some applications of correlation. Mainly the kernels concerning blurring and edge detection in imagery.

Blurring

The blurring effect filter is used in removal of identities and other details in a given image. A blurring filter can be achieved by using a kernel such as those shown in table 3.2. While both of them blur an image, the blurs are quite different. The mean blur, table 3.1b, averages a region of pixels, here in a 3×3 grid. This makes it a powerful tool to either find the average illumination of an image, since small details will be disappeared.

```
1  // Import the input image
2  Mat input_image = imread("C:/somefile.jpg");
3
4  Size kernel_size = Size(3, 3);
5
6  // Blur the image with a 3x3 Gaussian kernel
7  Mat gaussian_output_image;
8  GaussianBlur(input_image, output_image, kernel_size, 0);
9
10 // Blur the image with a 3x3 Mean kernel
11 Mat mean_output_image;
12 blur(input_image, mean_output_image, kernel_size);
```

Code 3.1 *OpenCV blurring filter implementation (C++) example.*

The Gaussian blur retains a much greater amount of detail in the image making it better for cases where one wants some detail, but not all. An example of this is trying to remove

noise. If one uses a mean filter to remove noise, then the details in the image are removed alongside the noise. The Gaussian blur can also reduce the amount of noise, but without removing as much detail. This is especially useful, if one wants to do edge detection on an image after noise reduction, as edges can easily disappear with a mean blur.

$$Gaussian\ kernel = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \qquad Mean\ kernel = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

(a) *This is the* 3×3 *kernel used for Gaussian blur.* ***(b)*** *This is the* 3×3 *kernel used for mean blur.*

Table 3.2 *This table shows two different* 3×3 *kernels for blurring.*

It should be noted, that when using kernels on data, the operation in total should be normalised, that is every value calculated with a 3×3 mean kernel, should be multiplied by $\frac{1}{9}$, since the sum of the entries is 9 (the $width \times height$ of the kernel). This rule, of multiplying by one over the sum of the kernel, works as long as the sum of the kernel is not zero. In that case the kernel itself *is* normalised and therefore the normalisation is not needed. Also, multiplying by $\frac{1}{0}$ is undefined anyway.

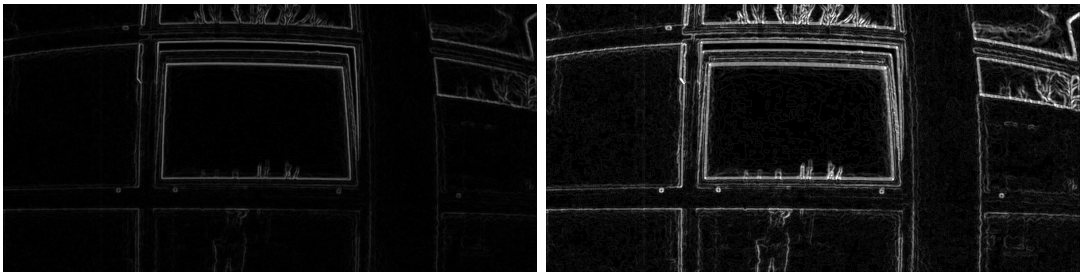
Edge detection

Edge detection [See fig. 3.7a & fig. 3.7b] is a technique that could make sense to use when working with objects with sharp/straight edges, like windows have. Like the blurring effect, edge detection can be achieved by the use of correlation with selected kernels, for instance using the Sobel and Scharr kernels [See table 3.4]. A notable difference between blur filters and edge detection filters are that edge detection uses two kernels for correlation, for detecting edges across both the x and y-plane (Horizontal and vertical "scanning").

$$f_x, f_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad g_x, g_y = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}, \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

(a) *Horizontal and vertical Sobel kernels.* ***(b)*** *Horizontal and vertical Scharr kernels.*

Table 3.4 *Edge detection kernels.*



(a) Edge extraction using Sobel kernel. *(b) Edge extraction using Scharr kernel.*

Figure 3.7 *Edge detection processed images using Sobel and Scharr kernels. Created using OpenCV.*

When the image has been subject to sufficient noise reduction and edge extraction measures, the image leaves the "pre-processing" phase and enters segmentation.

Thresholding

To binarise an image, it is needed to find a suitable threshold above which a pixel is set to 1 and below which it is set to 0. Finding this threshold is based on a histogram of the grey scaled image. After all the previous processing, there should be a good separation between the bright pixels that are of interest, and the dark ones which are not.

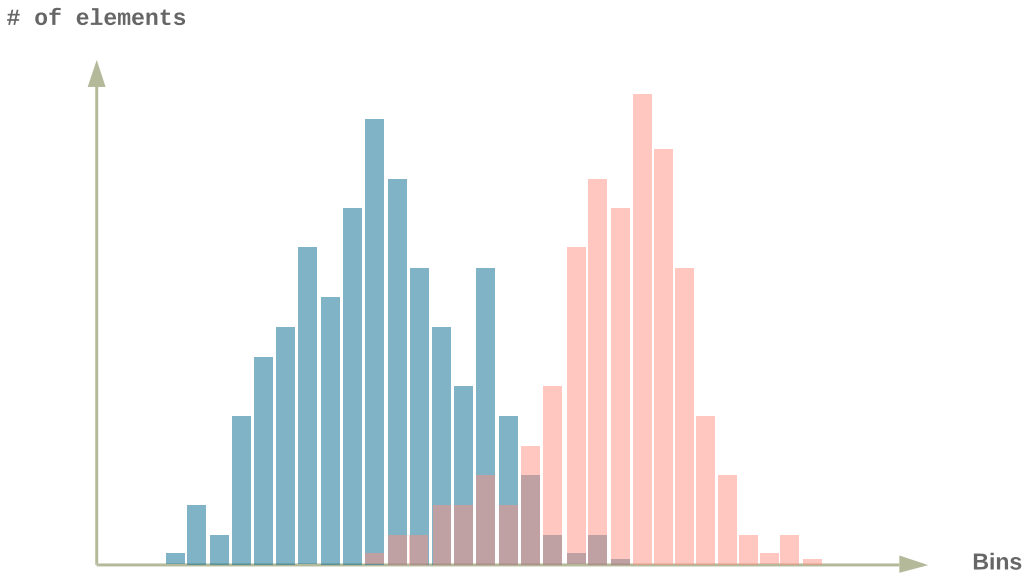


Figure 3.8 *This histogram has two clearly defined peaks. The threshold for this histogram would be in the valley.*

In the example fig. 3.8 it should be evident that this segmentation of the two groups is fairly clear. It should however also be mentioned that there is some data in the valley which is likely to be on the wrong side of the threshold. As long as the majority of the data is on the right side of the threshold, the binarised image can be further improved by morphology.

Morphology

As briefly mentioned in section 3.1.3 morphology comprises the operations that one can use on a binarised image. The two mentioned in section 3.1.3 were dilation and erosion. Dilation is a process where a kernel, called a structuring element in this context, is run across the entire image like with the other kernel based operations discussed earlier in this chapter. If a pixel location, with the kernel on top, "hits" at least one white pixel, then this pixel is set to white. This process therefore increases the size of every binary object in the image. Erosion is the opposite and therefore makes objects smaller. The simple combinations of these two processes are called "Closing" and "Opening". Closing is defined as first dilation followed by erosion. This helps to remove black holes in objects like dilation, but it is less destructive in general. This is because much of the added pixels are removed, if they are not internal to an object. Opening is the opposite process, erosion followed by dilation. This removes noisy white pixels and helps to isolate objects. Again this is like erosion, but the pixels removed from larger objects are put back by the dilation process.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(a) *A* 5×5 *rectangular kernel.* ***(b)*** *A* 5×5 *cross-shaped kernel.* ***(c)*** *A* 5×5 *elliptical kernel.*

Table 3.6 *Rectangular, cross-shaped and elliptical structuring elements of* 5×5 *implemented in OpenCV's* `getStructuringElement()` *function[17].*

When using these operations the kernel shape, and size, can be changed according to specific application needs. As an example one might want to remove lines from the binary image, while preserving round objects, this could be done with a round kernel. This kernel shape preserves round objects but removes other object shapes, including lines. Moreover, tweaking the size of the structuring element is needed to remove the wanted elements, since removal of larger elements require larger kernel sizes.

BLOB analysis

In BLOB analysis a vector of pixels, with their values and positions, is received for each object. The most important point from here is extracting the features from the

BLOB. For this, BLOB analysis is essential. First of all, there are a variety of examples of features of BLOBs. For example, area (number of pixels), number of holes, size of holes, length of the perimeter, and bounding box can be examples of features of BLOBs. These features can then be used for describing different objects, and are all calculated from the vector of pixels that represent the BLOBs directly. Each feature vector literally designates each object's characteristics by listing the values of the individual features of this object. For this to work certain BLOBs have to be discarded. One example of this, is when a BLOB is on the border of the image. These BLOBs could in principle represent anything, as the BLOB is incomplete. This work, after extracting features of each object, allows the computer to analyse each of them so that it will be able to move onto next step for this project.

This section covered some pre-processing and segmentation techniques. These steps are necessary to make the computer able to recognise and analyse the image. Without it, it would be more difficult. The next step is representation, which consists of the way to represent the data from the pre-processing step. It will be introduced precisely in the next section.

3.3.3 Data representation

Here it will be discussed how the input for a classifier can be. Essentially, one wants to be able to give the classifier one object at a time. Moreover, the input to the classifier should be a good representation of the features of the object. The features of an object were found by BLOB analysis. One representation of the features is the form in which they were calculated. This is an obvious choice and can work just fine. However, since the data resides in vectors and matrices, linear algebra allows one to do many different manipulations of such data. Such manipulations can be used to change the representation of the data in a matrix, and can even create more informationally dense representations of data. This kind of process is called dimensionality reduction.

Dimensionality reduction

If the computational time or memory requirements of a machine learning system become too great, then it might be useful to consider dimensionality reduction. Generally speaking, some features are more important than others and through dimensionality reduction, one can reduce the computational and or memory requirements more so, than the performance. One possible way of reducing the dimensions of a feature space is by a technique called Principal Component Analysis.

In a nutshell, one calculates the covariance matrix of the inputted features, uses an eigenvalue decomposition and uses the first n eigenvectors as the representation of the features.
Such a covariance matrix is a measurement for to which degree there is variance between different features. If there exists a high degree of covariance between two features,

that means that these two features are quite dependent on each other. Dependence in this context means that one could be expressed as a multiple of the other. Therefore, having such two features that are merely scaled versions of each other, gives no additional information to the system. Conversely, the features with the lowest covariances show independence to the other features and therefore contain information that is not present in the other features. On this covariance matrix an eigenvalue decomposition is performed. Such a decomposition transforms from the original feature space to another space where the features are represented by the eigenvectors. The eigenvalues of the respective eigenvectors show the importance of the eigenvectors in representing the features. Therefore, if one was to choose the first n eigenvectors, one would have a much more information dense representation of the features. This more dense representation could then be sent on to the classifier.

3.3.4 Classification

Now that the previous parts of a generalised computer vision pipeline have been discussed, here the end of the pipeline will be discussed in detail: The classifier. The reason for having a classifier, is that no matter what has been done previously in the pipeline to the data, eventually this data should be converted to one of the possible classes of the classifier, hence the name. In a computer vision pipeline these classes could be, “dog”, “cat”, “ball”, “car” or “none” as a few examples. The idea is that a given classifier is tuned to classify data into a specific set of classes. This set of classes is then chosen based on the application, such that a classifier is only capable of classifying data relevant to the application.

The definition of these classes is done through training the classifier on some training data. The training data consists of examples of each class, e.g. many pictures of dogs and cats, each tagged with the what class it belongs to. Given these different inputs representing a single class, the classifier then learns what a dog is, based on these examples. The same then goes for the other classes as well. Importantly, the classifier should have examples of data that do not belong to any of the classes. If this is not done, then the classifier, which will always map some input to an output, would be forced to output either cat or dog, even if neither is presented in the image. This is often not the wanted behaviour. This approach is, in general, called “supervised learning” and is the preferred classifying strategy for when you know what you are looking for beforehand, e.g. use of predefined classes/categories for classification outputs. It does however, also require that labelled data is on hand for the training process to be possible.

Given how the classifier is supposed to learn from the data, it should be obvious that the quality of the provided data should be good. If not, then the classifier cannot perform well. However, the data is only one half of the classifier, the other being the algorithm. Here three different approaches will be mentioned: “K-nearest neighbour” “Bayesian” and “Neural network”.

K-nearest neighbour

The K-nearest neighbour is a classification algorithm that utilises the idea that similar objects (belonging to the same class) are similar in their features. Training this algorithm consists of ordering the labelled input data based on their features. When this data is loaded, new unlabelled inputs will then be compared to the k-nearest neighbours, as shown in fig. 3.9. These neighbours are counted, that is how many neighbours belong to each class, and the class which the majority of the neighbours belong to, is chosen as the output for the current input. In this way an input is classified as the class which is the most similar to the examples given in the training data.

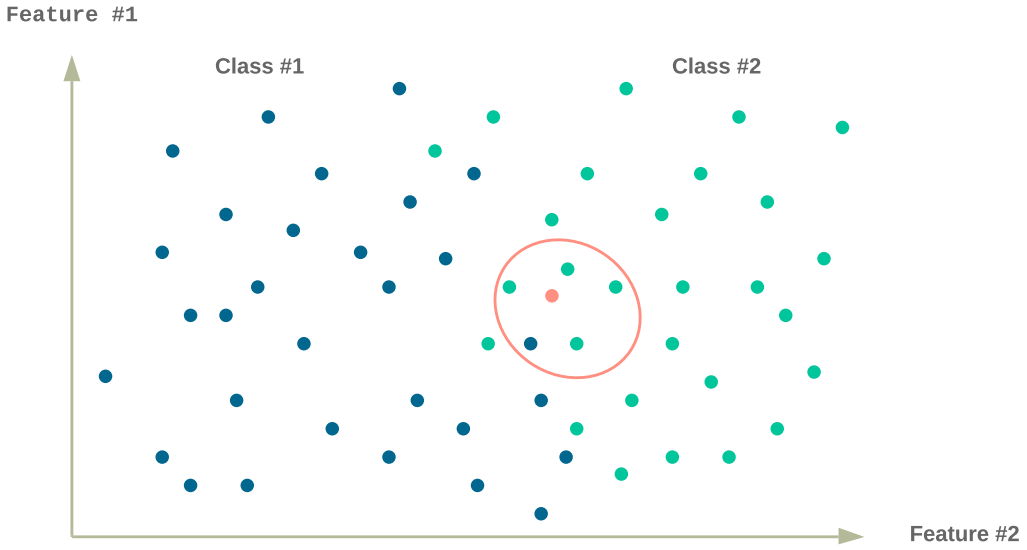


Figure 3.9 *Here is an example of a K-nearest neighbour based classification. It shows two groups, one blue and one cyan, which represent the training data. The salmon coloured dot represents the new data to be classified. The circle around the data point shows the 5-nearest neighbours. Given that a majority of them are cyan, the data point will be classified as cyan.*

This technique involves calculating the distances between a new input and all of the training examples, in the n feature dimensions describing the input. While this technique requires no processing on the training data (meaning no initial time investment), the actual run-time of the classifier will be bogged down the more training data provided. Since more training data, and more features, are simple ways to improve the output it is a problem that it also hinders the computational performance of the classifier as well. This is the primary limitation of the k-nearest neighbour classifier.

Bayesian classifier

Bayesian classifier uses the probabilities among the experimental sets so that they can be used for comparing each other, and the classified result can be solved.
For Bayesian classifier, the notion of the Bayesian formula is needed, and the formula is:

$$P(\omega_j|x) = \frac{p(x|\omega_j) \cdot p(\omega_j)}{p(x)} = \frac{likelihood \times prior}{evidence} \tag{3.3}$$

and for meaning of each member,
The prior probability $P(\omega_j)$: The probability of observing class ω_j without taking any observations into account.
The likelihood function $P(x|\omega_j)$: The probability of observing the feature x assuming class ω_j
The evidence $P(x)$: The probability of observing the feature x. A normalisation that ensures that $\sum_j P(\omega_j|x) = 1$
The posterior $p(\omega_j|x)$: The probability of class ω_j given feature x, i.e. after accounting for the observation.

Thus, as mentioned above, if there are two sets(ω_1, ω_2), we should compare $P(\omega_1|x)$ and $P(\omega_2|x)$ for classifying, and if the former one is higher, then the result is ω_1.

Neural network

Neural networks are based on the idea of neurons. These neurons take an input, execute a mathematical operation on this input and provides its output to a number of other neurons. This suggests a layered structure to a neural network and the simplest of this kind is a network with only three layers. The first layer, the input layer, contains as many neurons as there are features of the input. The next layer, the hidden layer, holds a number of neurons each of which are all connected to the neurons in the first layer and the final layer. The final layer, the output layer, contains as many neurons as there are classes in the classification. Given that each neuron represents a calculation with tunable parameters and each connection to another neuron also represents a calculation with tunable parameters, it is clear that neural networks provide a great deal of flexibility in what they can map between. It should also be clear, that this parameter tuning cannot be done by hand. This is done through training.

The process of training consists of first randomly initialising the parameters in the network. Next, the first piece of training data is inputted and the output is calculated. The values of the output neurons are compared to the label of the input data. Here the neurons will generally give values far from what they should be (1 in the correct class neuron and 0 in the rest). To approach the correct values, the process of backpropagation is used. Backpropagation is about changing the parameters of the neurons in contact with the output layer. The parameters are changed slightly according to the

negative gradient of each neuron's function. This work is done for each layer in the network, except the input which does not have these parameters.

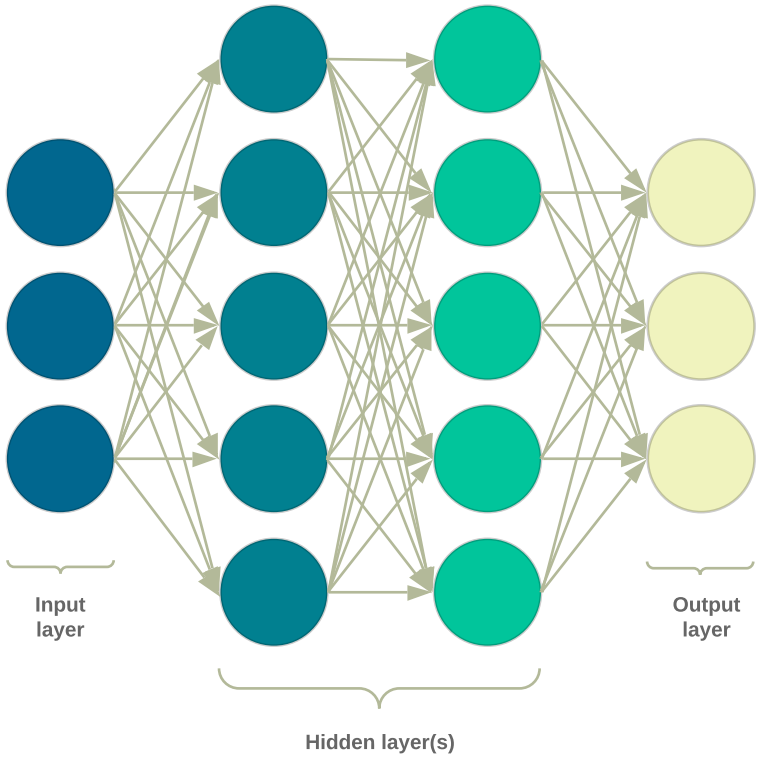


Figure 3.10 *This is an example of a neural network. The network shown has four layers, but as mentioned in the section, the smallest would have only three layers. In that case, the second hidden layer would be removed.*

This process is then done for each piece of training data. The process as a whole can then be repeated for a given number of epochs, as they are called. The rate of error when using the test data, that is data which has not been used for training, should keep going down until a point. Eventually, the network will be over-fitted to the training data, which is not wanted. Over-fitting happens when the classifier begins to learn the noise in the input data. Using this noise in actual classification, with test data, does not yield good results. The training should be stopped such that over-fitting is minimised, which is why one should find the error rate on the test data in between epochs of training. This would give some indication as to when it is time to stop training.

A number of operations to use a neural network based classifier does not depend on the amount of input data, in contrast to the k-nearest neighbour classifier. Therefore, it is possible to achieve better overall performance, but at the cost of much larger data sets.

This is because the amount of training data only affects how long it will take to train the network. It should be noted though, that to use larger and more complex networks, it would be sensible to have more training data, as larger networks can master greater complexity and therefore more nuances in the data. Following this line of thought, more data also makes it possible and preferable to use a larger network for better overall performance. But one is not forced to do this, if the use of the classifier is computationally limited.

Chapter 4

Design

In this chapter the system design as a whole will be presented. The design section is split into three main sections: ideal system, demarcation and actual system design. Firstly, in the ideal system section, the contents will cover the system which is ideal, and without any constraints or bad conditions. That system would be this project's ultimate goal. For demarcation, it will describe some constraints while progressing this project as well as conditions. In the actual system design, image processing for window detection will be summarised as well as the window state classification.

4.1 Ideal system overview

To begin the design of the system, the next section will present a user story to put the reader into the mind of the potential user of the system.

4.1.1 Usecase

A team of researchers has the task of studying occupants window opening behaviour. They find a building facade with inhabitants willing to participate as test subjects, and deploy the thermal camera platform. The deployment consists of finding a location with a good view of the facade as well as bringing power and ethernet connectivity to the setup. The setup is preconfigured with a server which it will connect to. After all cameras are setup, the researchers use a login to the server, they can then monitor the state of the individual camera setups, online, offline, error etc. Certain settings may also be changed, if a given setup should send data with a lower frequency or a higher one. They can of course also view the data already in the database.

This user story highlights some important features of the system:

- Connectivity
- Data availability

- Privacy of participants

The privacy of the participants is important for two reasons in particular. Firstly, it would be much harder to find participants, if some degree privacy could not be promised. Secondly, it would not be ethically responsible if the researchers were not able to provide adequate privacy for the participants.

4.1.2 Ideal system design

This section will briefly discuss how an ideal system, one that actually provides the wanted functionality in the usecase, could be designed conceptually.

To provide the functionality regarding the accessibility of data, the system needs to be connected to the internet. Also, to do so in a manner that is ethically responsible as discussed in section 2.2.2, it is preferable to only output the necessary data. This reduces the potential damage, if the data was to be compromised during operation. The use of onsite processing also puts more pressure on choosing the algorithm with the best performance to cost ratio, as onsite processing could become expensive if more processing was needed. The deployment of costly setups that would have to be unattended for weeks on end is also a security risk in it self, as it would be problematic if it was stolen.

To make the data available at will for the users all the camera setups would have to output the data to a central database.

Regarding the pipeline itself, it would also be ideal to let the proposed window regions be verified by the next part of the pipeline. This would allow for the system to be more certain in that the data sent relates to actual windows. This kind of verification does however require more data than a pipeline without this verification. In the ideal scenario, there would be data containing many different classes of objects that can be expected to be in front of, or on, a generic building facade. These could be objects like, people, cars, AC units or external lights just as a few examples. This data would give the final classifier a chance to classify a potential window as other things as the accuracy of the detector cannot be guaranteed. Moreover, the ideal system would also be able to output data describing not just whether a window is open or not, but *how* open it is.

The accuracy of the system as a whole would of course have to be "good enough". To achieve this, it might be necessary to use more data with more complicated processing and classifiers. These parameters will push the limits of what a small, cheap embedded system will be able to accomplish. Therefore, the optimisation of the pipeline would be very important, both in using the right methods, but also in code optimisation.

Requirements and constraints in the ideal case

This conceptual design leads to some constraints in this case:

Ideal requirements:

- Data availability
- High rate of accuracy
- Performant

Ideal constraints:

- The use of portable hardware and its implications.
- Assumption, that the system isn't time critical.

To achieve these requirements within these constraints, would require a lot of work.
For the purpose of this report, the system to be designed and implemented will be a subset of the ideal scenario outlined here.

4.2 Demarcation

It has been decided that the focus of this report will be the pipeline itself, under much looser constraints.
This report does not aim to design a distributed system, but instead a pipeline which can show that it might be possible to develop such a system, based on the use of thermal cameras. The pipeline will also be more limited in that it will only be able to classify windows as either opened or closed. To allow for a non-binary classification it would be necessary to gather a lot more new data.

State of the art data sets are primarily produced by large-scale teams backed by technology giants, which have the resources and experience to deliver great results. These data sets can be open-source, and a large number of them are[18], but for the usecase of detection, and classification, of windows through the lens of a thermal camera, the dataset needed just isn't there. Therefore, creating such a large dataset has also been deemed out of scope for this project.

With the final thermal camera solution being a mobile one, it's necessary to also execute the vision pipeline on mobile hardware. Luckily, because this solution doesn't have a time critical real-time requirement, processing power shouldn't be that much of an issue. Therefore, the solution can (in good faith) be deployed on low-end portable hardware, like the Raspberry Pi*. Still, deploying the solution to a SBC is beyond the scope of the project. After this demarcation, the actual system design will be discussed.

*The Raspberry Pi is a series of small low-cost SBCs (**S**ingle **B**oard **C**omputer)

4.3 Actual system

In this section the actual system for implementation will be discussed. After this demarcation what remains should be the most important parts of the system. These elements should together be able to show whether it is at all possible do window monitoring with a thermal camera.

4.3.1 Methodology

In this section, the process for getting the contours for getting windows' spots. There were some difficulties while doing this stuff because of some conditions. However, the successful image which has some contours that surrounding windows was derived and the failure one will be covered for comparing two ways for manipulating the input image.

Image processing process

The thermal image that experiments are conducted on:

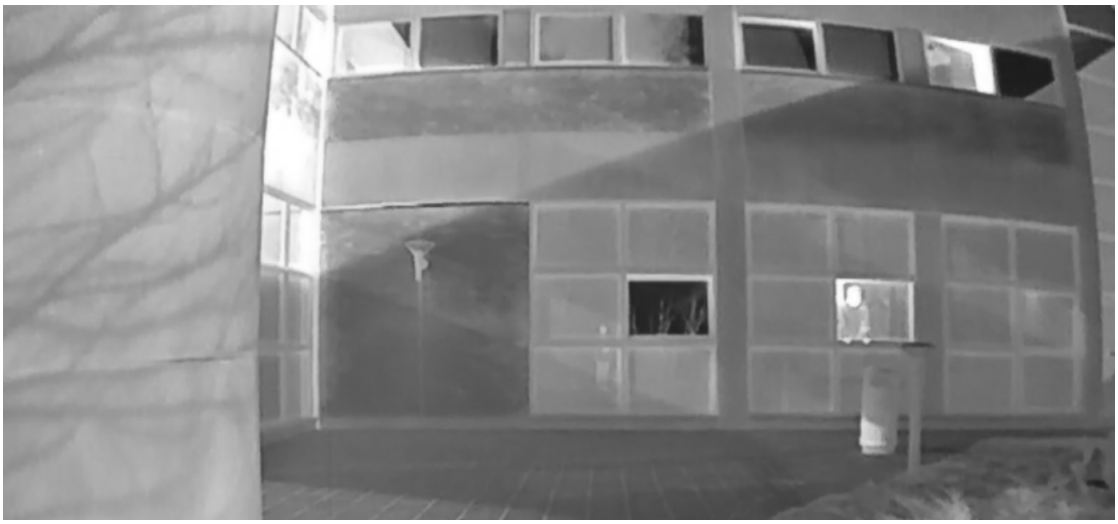


Figure 4.1 *A building facade at the AAU campus, captured with a thermal camera*

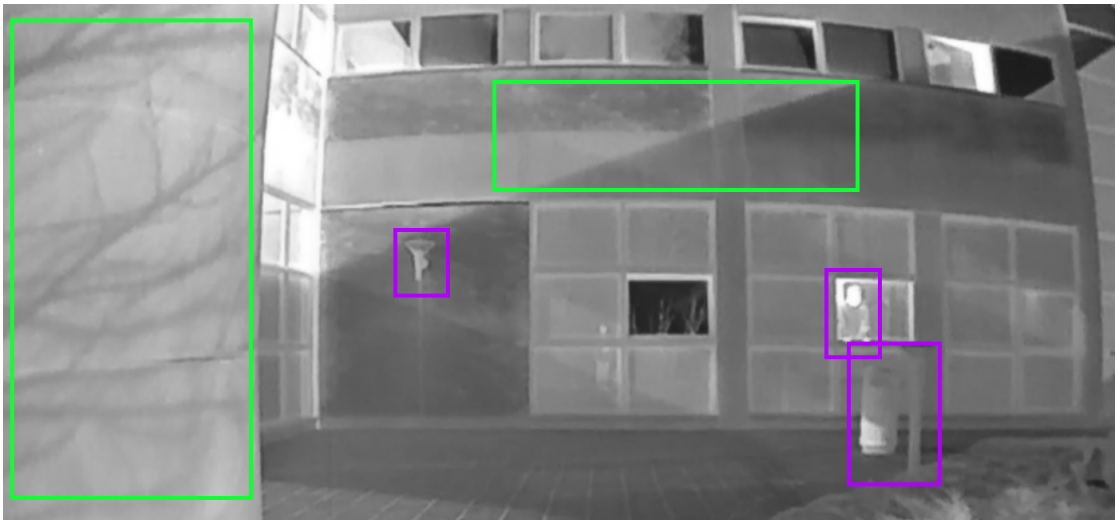


Figure 4.2 *Obstacles in the image for detecting windows*

Some of the issues with the image:

- Green - Unwanted contrasts / shadows
- Purple - Unwanted objects obstructing the facade

First of all, there were some obstacles while getting the contours for the windows. The picture of the previous section, there are some green and purple boxes. In green boxes, there are some shadows of trees around the spot and the other adjacent building so that contours were also appeared for some parts of the shadow, as well.
For the case of purple boxes, one of our member is detected inside the building, and two lamps are detected, too. As a result, there were some contours inside those boxes. To add, when observing two upper right window, one looks brighter than the other, because of the heat inside and reflecting the sky.
From next paragraph, it will introduce each screen and manipulating track bars for manipulating the given image.

When compiling the code in "trackbarmode", there are some track bars for manipulating the threshold image in each screen. This mode allows one to change the values of the individual operations using trackbars. This allowed for rapid testing of different parameters, using the functions implemented in the project.

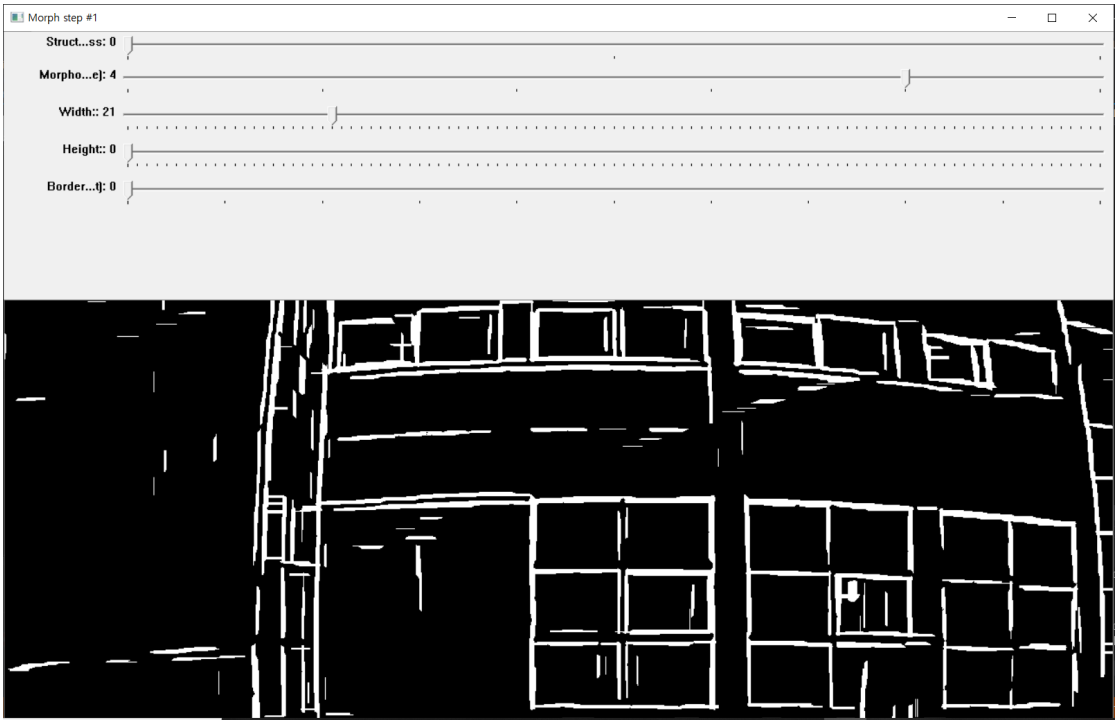


Figure 4.3 *The image which went through the first morphology step. This is the first screen*

The main point is second one, which means 'Morphology type'. By going through some experiments, the best one was choosing open(destroy) which is number 4. It could make the white part of the image sharp before manipulating the width and the height, and this point makes the final image's contours clear than any other images which had other options. The other essential options are width and height. For width, 21 was appropriate for the result and the smallest height value was also the important component. With those two values and the open(destroy) option, the other track bars didn't affect remarkably, as seen in fig. 4.3.

The second screen, as seen in fig. 4.4, here all the options were fixed as 0. For the second morphology step, open was chosen. As a result, when making all the values as 0, the result looked most splendid.

The result of this process can be seen in fig. 4.5. Threshold value was 24, the minimum line size and maximum one were 8 and 19 respectively, and density value was 1. When the maximum value was so high which was about 80, the image was almost full of the red lines. However, after setting it as between 10 and 20, the contours of image looked clear, and the optimal one was 19. After fixing the value for maximum, the next step was fixing threshold and minimum. Those values which were mentioned was also optimal ones. In the aspect of density, 0 and 1 didn't make a difference, but, when setting it over 1, the image didn't look clear than setting smaller values.

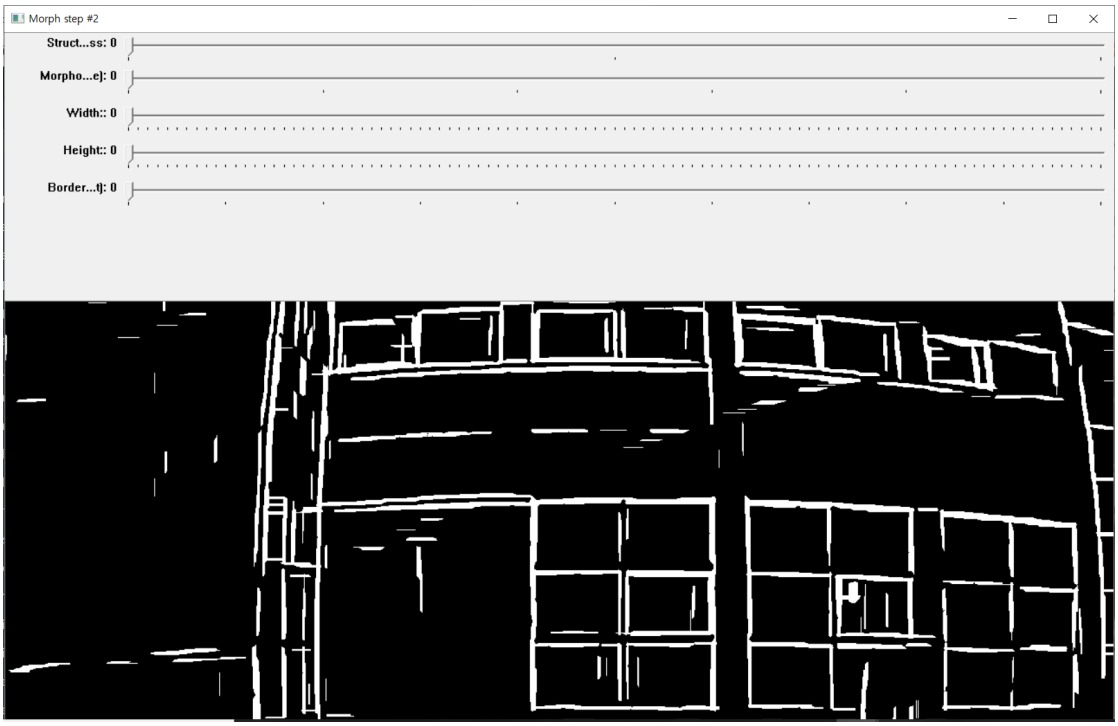


Figure 4.4 *The image which went through the second morphology step. This is the second screen.*

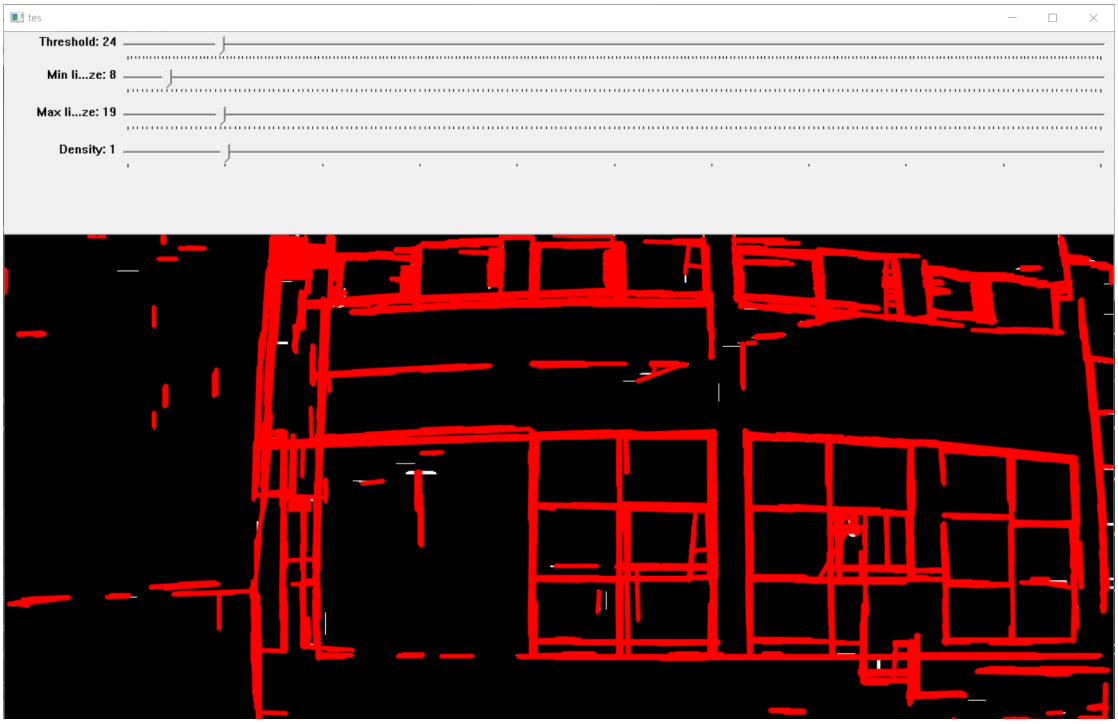
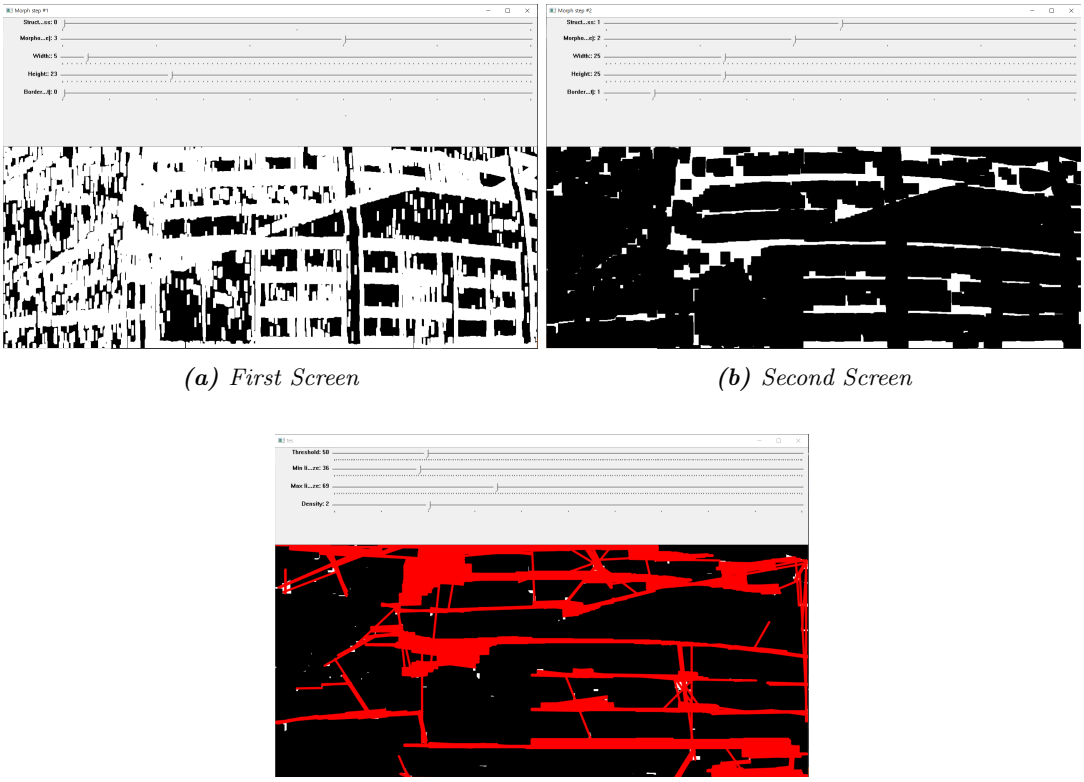


Figure 4.5 *The final image of this process. This is the third screen.*

This is the failure case:



(a) First Screen *(b) Second Screen*

(c) Third Screen

Figure 4.6 *Failure case*

In the first screen, fig. 4.6a, the two critical points for successful one are opposite. Morphology type is different and width value is smaller than the height one. In the next screen, fig. 4.6b, every value is not 0, different from the successful one, and the result screen, fig. 4.6c, has opposite values.

This show just one set of settings that did not result in a useful output. When comparing two cases suggested, in the successful one, window contours are emphasised, and shadow contours are minimised. On the other hand, in the case of the failure one, window contours are invisible, but the shadow from the other building is emphasised. Hence, it should be clear how this tool made a great difference in how quickly it was possible to test different image processing settings.

The contents of the window detection process will be shown in the next section. Moreover, it will summarise a range of pre-processing steps.

4.3.2 Window detection

In order to make a computer detect windows, it needs to know how a window is characterised. Luckily intuition and the nature of how windows look in the thermal spectrum, suggests that edge detection in particular might be an effective strategy in obtaining window detection. The hypothesis goes something like: "a proper edge detection goes a long way in detection objects with clear defined edge like windows".

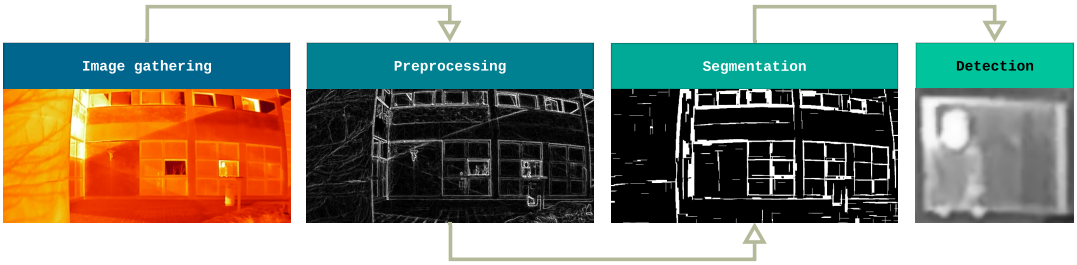


Figure 4.7 *Window detection pipeline flow.*

The steps of the detection algorithm:

1. Apply histogram stretching
2. Blurring the image
    - In order to minimise noise in the image
3. Edge detection
4. Thresholding
5. Morphology operations
    - Make window rectangles stand out
6. Detection

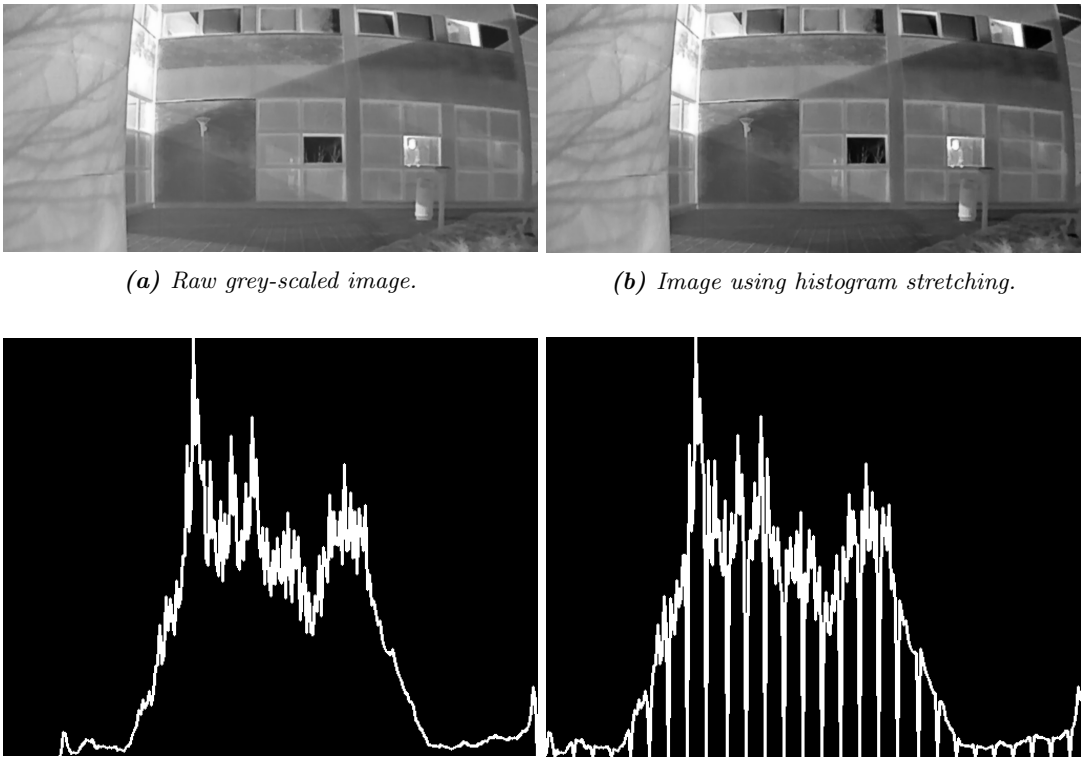
Input acquisition and initial enhancement

The first part of the processing algorithm is to acquire the video frames to process. In OpenCV, importing a video or a video stream is fairly trivial, and can be achieved with a few lines of code (See snippet 4.1). As these frames come from a given thermal camera they are received as single channelled.

```
#include "opencv2/opencv.hpp"

using namespace cv;

// Import video file or URL to live video feed
VideoCapture videoFeed("video_name.mp4");

while(true) {
    Mat singleFrame;

    // Working with a single frame from the imported video feed
   videoFeed >> singleFrame;
}
```

Code 4.1 *Import thermal video feed OpenCV implementation example.*

When a frame has been successfully grabbed, it is firstly subjected to a contrast enhancement. This enhancement is to ensure that the thermal spectrum of the given frame stays consistent, in the sense that every frame spans from all the way from the lowest value of 0 to the maximum of 255.



(a) *Raw grey-scaled image.*

(b) *Image using histogram stretching.*

(c) *The histogram, before histogram stretching.*

(d) *The histogram, after histogram stretching.*

Figure 4.8 *Before and after applying histogram stretching to the image. Full size versions are available in appendix section A.1.1.*

The contrast which is the difference between maximum and minimum pixel intensity is enhanced with technique of histogram stretching. Histogram stretching is achieved by applying the formula 4.1

$$g(x,y) = \frac{f(x,y) - f_{min}}{f_{max} - f_{min}} * 2^{bpp} \tag{4.1}$$

The formula requires finding the minimum and maximum pixel intensity multiply by levels of grey, and $f_{max} - f_{min}$ would stand for the contrast. First of all, bpp means **B**it **P**er **P**ixel. So, if the image is 8 bpp, levels of grey would be 256. Then, 2^{bpp} would be 255. $f(x,y)$ in the formula stands for the value of each pixel intensity. This formula would be used for each $f(x,y)$ in the image. As a result, the contrast of the image will be increased. However, there is a case which is failed to success the histogram stretching. As being mentioned, $f_{max} - f_{min}$ means the contrast. If the contrast and 2^{bpp} were same, they would be offset. Moreover, if the minimum pixel intensity is zero, then, $g(x,y)$ would be same as $f(x,y)$.

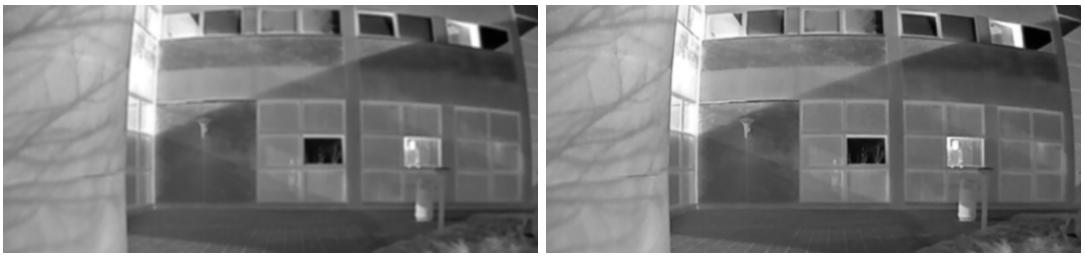
That means the output image is equal to the processed image which means there is

no effect of histogram stretching has been done at the original image. Except this case, histogram stretching will make the contrast bigger than previous image's. [19]

Blurring

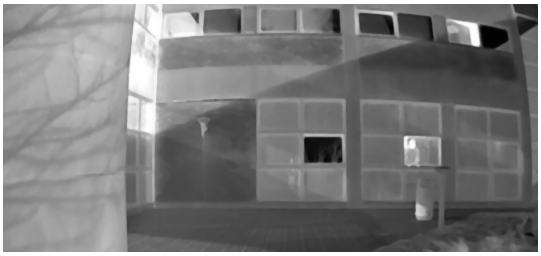
With these thermal images having a relatively low resolution of 800×600 pixels, the details are already scarce as is.
With that said, there are still noise reduction to gain by applying a blurring filter to these images.

- Mean blur
- Median blur
- Gaussian blur
- Bilateral blur

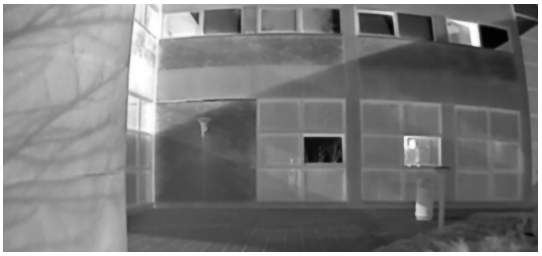


(a) Mean blurring with 9×9.

(b) Gaussian blurring with 9×9.



(c) Median blurring with 9×9.



(d) Bilateral blurring with 60×60.

Figure 4.9 *4 different blurring filters applied to the histogram stretched base image fig. 4.8b. Full size versions are available in appendix section A.1.2.*

Mean blur uses a box linear filter. This is a spatial domain linear filter in which each pixel in the resulting image has a value equal to the average value of its neighbouring pixels in the input image.[20, 21] For this project's code, `blur()` function was used.

Gaussian blur is a linear operation, but it doesn't conserve edges in the input image. The value of sigma decides the degree of smoothing and also the way edges are preserved.[20] In the code, this project used `GaussianBlur()` function.

Median blur uses median filter, and it is a non-linear filter. The difference of this and linear filter is that median filter replace the pixel values with the median value available in the local neighbourhood. Its filter is also noise reducing filter.[20, 22] Moreover, its filter preserves the edge, and this project used `medianBlur()` function in the code.

Bilateral blur uses bilateral filter which is a non-linear filter, as well. It prevents averaging across image edges, while averaging within smooth regions of the image so that makes edge be preserved. In addition, bilateral filter is non-iterative, and non-noise reducing smoothing filter.[20, 23] This project's code used `bilateralFilter()` function for bilateral blur.

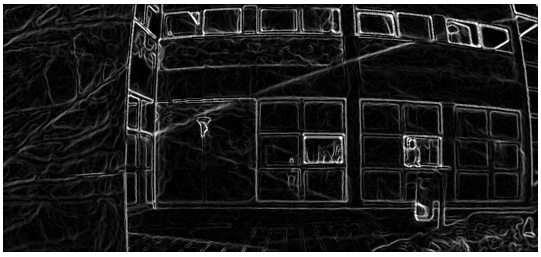
> The median blur seems like the best option. First of all, for preserving edges of the input image, non-linear filter is way more effective than linear filter.
> Then, two types of blur are left, median and bilateral. When focusing on reducing noise, then, median blur is the best option among four types of blur.[24]

Edge detection

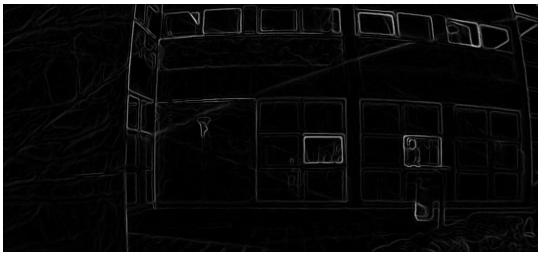
As the primary objective of the pre-processing and segmentation step is to highlight the windows. Based on that notion, clear characteristics of said window need to stand out from the rest of the facade. The contrast in temperature between the window frame and windowpane is a prominent characteristic to emphasise. This can be achieved by the use of edge detection as described in the technical analysis [section 3.3.2].

$$Laplacian = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \qquad Laplacian\ w/\ diagonals = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge detection was performed with a handful of different kernels. In addition to the aforementioned Sobel and Scharr kernel, correlation with Laplacian kernels were also conducted.



(a) Edge detection with Scharr.



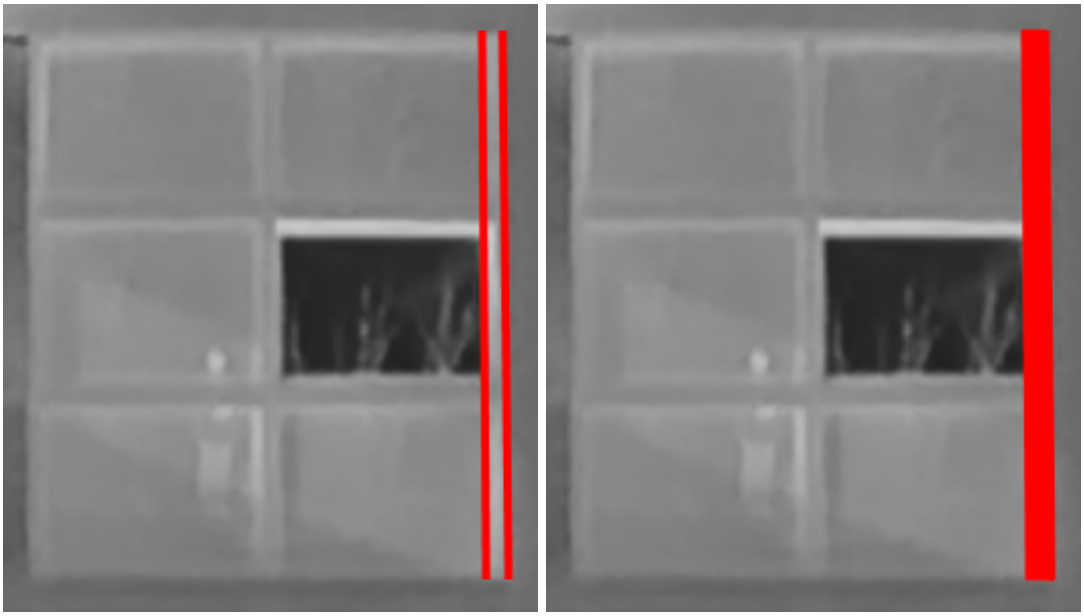
(b) Edge detection with Sobel.

(c) Edge detection with Laplacian w/ diagonals.

Figure 4.10 *3 examples of edge detection. Full size versions are available in appendix section A.1.3.*

While brainstorming about how the pipeline could emphasise window frames in particular, a couple of ideas for "experimental" edge detection kernels were proposed, seen on fig. 4.11.

- Double edge detection
- Thick edge detection



(a) *The idea behind the "double" edge detection kernel.* ***(b)*** *The idea behind the "thick" edge detection kernel.*

Figure 4.11 *Experimental edge detection techniques ideas.*

The double edge detection kernel is simply taking an existing edge detection kernel, like the Sobel kernel, and using it twice. The kernel starts out with the base kernel, then an arbitrary gap of zero columns are inserted based on the distance from edge to edge (3 pixel gap in equation 4.2). Lastly a flipped version of the base kernel is appended to the kernel.

$$Double\ Sobel\ kernel = \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & -1 \\ -2 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & -2 \\ -1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix} \tag{4.2}$$

$$Thick\ Sobel\ kernel = \begin{bmatrix} -1 & 0 & 0 & 0 & 1 \\ -2 & 0 & 0 & 0 & 2 \\ -1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

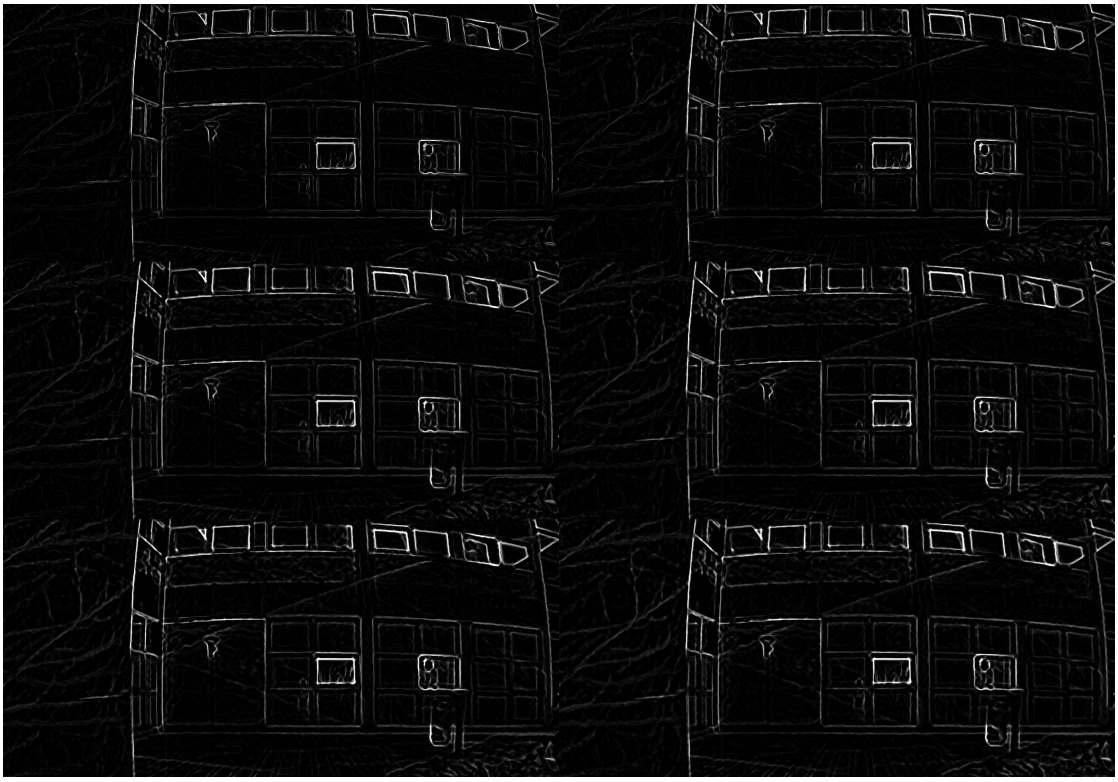


Figure 4.12 *Double edge detection using the Sobel kernel, with an increasing gap value that goes from 1 through 6. Starting from left to right, top to bottom. Full size version are available in appendix section A.1.5.*

These experimental kernels gave some interesting results.
The "double" edge detection kernel highlight and spread parallel edges, as seen on fig. 4.12. This kernel have the best results, and would consistently improve the visibility of the window frames in the facade.

The "thick" edge detection kernel widens edges, with some mixed results, as seen on fig. 4.13. It helps thicken some of the window frame, but it also amplifies noise in the image, notably the shadow cast across the building and the shadows from a nearby tree. The "thick" edge detection kernel only works with kernels that have a zero-column mid-column like the Sobel and Schrr kernels have, and like the Laplacian kernels don't.

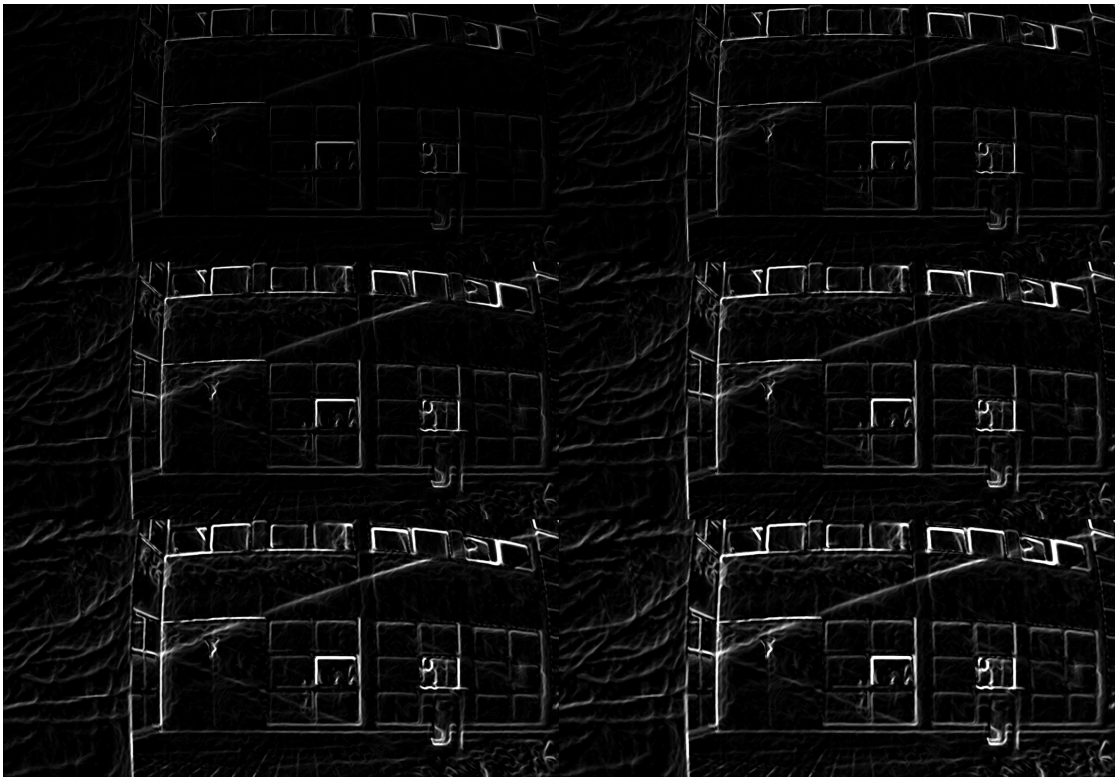
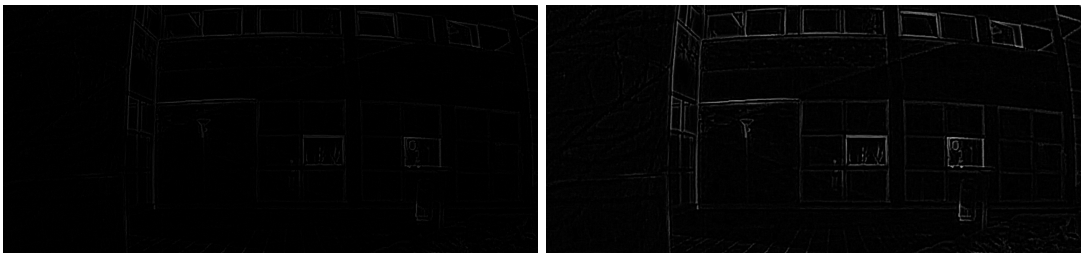


Figure 4.13 *Thick edge detection using the Sobel kernel, with an increasing gap value that goes from 1 through 6. Starting from left to right, top to bottom. Full size version are available in appendix section A.1.5.*

In the end, double edge detection with the Laplacian with diagonal edge detection kernel was chosen, seen on fig. 4.14. The Laplacian edge detection kernel seems to create grainy noise, but contrary to other kernels, it emphasise noise in the same magnitude.



(a) *Standard Laplacian with diagonals kernel edge detection.*

(b) *Double Laplacian with diagonals kernel edge detection. With the gap value set to 0.*

Figure 4.14 *Comparing the Laplacian with diagonals edge detection kernel with both the standard and double configuration. Full size versions are available in appendix section A.1.4.*

Threshold

The next part of the detection pipeline is binarisation, also known as thresholding. The strategy, as with the former steps of the pipeline, is to emphasise the subject as mush as possible while reducing noise as much as possible.
First and foremost a handful of adaptive threshold methods, like Otsu's method, were applied. These threshold methods automate the process of find a threshold value. Unfortunately were Otsu's and other adaptive threshold methods removing too much of the detail of the window frames, especially the windows on the ground floor that are affected by a shadow cast by a nearby building, which can be seen on fig. 4.15.

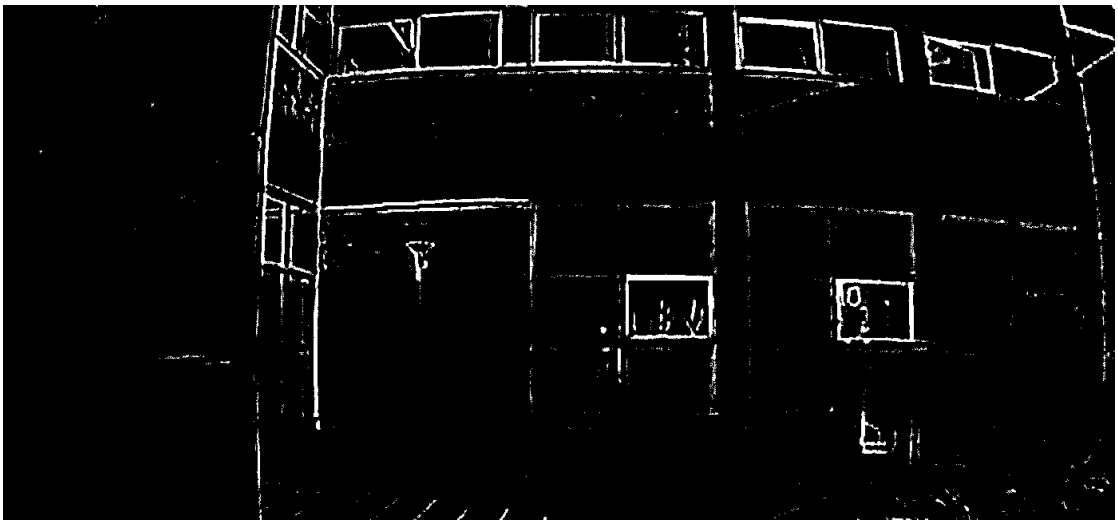


Figure 4.15 *Otsu threshold applied to fig. 4.14b.*

The manual approach of doing binary threshold (manual in the sense of manually adjusting a threshold value), allows for fine-tuning a specific threshold scenario for a specific case. Binary threshold can lead to bias, where a very specific threshold value works for a couple of specific scenario, but poorly for others. Lucky it seems like the best setting isn't some arbitrary value, but the low value of 1. Even though this applied binary threshold (seen on fig. 4.16) is very grainy and noisy, the window frames are somewhat connected, and stand out despite that.

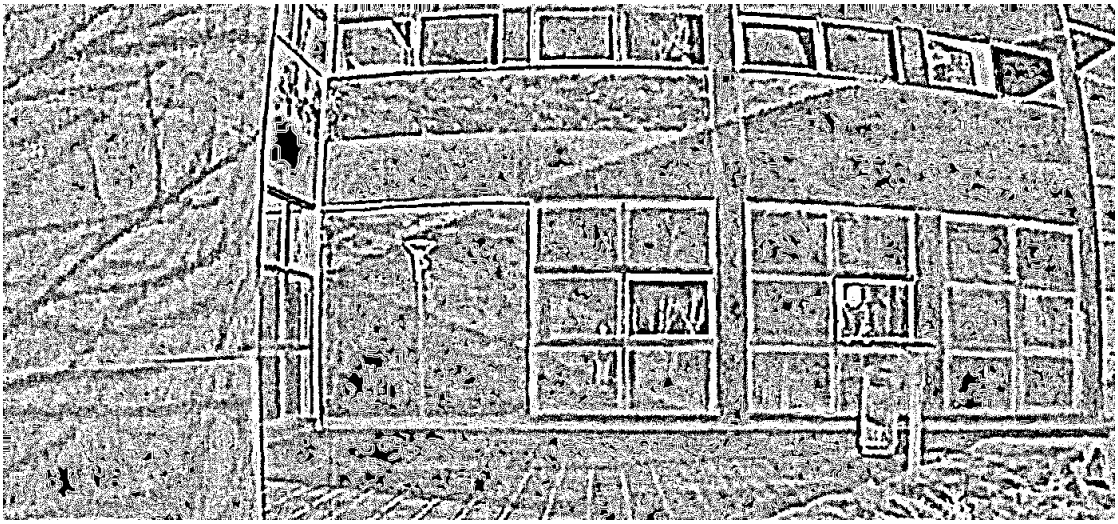


Figure 4.16 *Binary threshold with the threshold* $= 1$ *applied to fig. 4.14b.*

These window frame are very sensitive to changes, not only when adjusting the binary threshold, but also when trying to remove some salt and pepper noise with the median filter.

Morphology

The connected white pixels, that primarily represents the window frames, can be utilised in with the use of morphology operations. With the use of morphology, horizontal and vertical lines in the image can be extracted separately.
This can be achieved by combining the erode and dilate operations, known as "opening", previously mentioned in section 3.3.2. The selected structuring element picked for the opening of horizontal and vertical lines should also represent a line structure, in other words *line thickness* $\times$ *line length* where line thickness would be much smaller than line length. The opening operation is applied twice, using the rectangle representing both horizontal and vertical line, which is just a transpose of original line. The implementation of this can be seen on code snippet 4.2.

```
1  Mat structureElement, flipStructureElement, morphX, morphY;
2
3  structureElement = getStructuringElement(MORPH_RECT, Size(35, 1));
4  flipStructureElement = getStructuringElement(MORPH_RECT, Size(1, 35));
5
6  erode(input, morphX, structureElement);
7  dilate(morphX, morphX, structureElement);
8
9  erode(input, morphY, flipStructureElement);
10 dilate(morphY, morphY, flipStructureElement);
11
```

```
12 addWeighted(morphX, 1, morphY, 1, 0, output);
```

Code 4.2 *OpenCV implementation of morphological open operation with both a horizontal and vertical line structuring element.*

The result will give a sketch-looking image where the window frames are represented with a number of lines. In fig. 4.17 the structuring element is a fairly thin line with measures $1px \times 35px$. This configuration captures the majority of the windows in the facade.

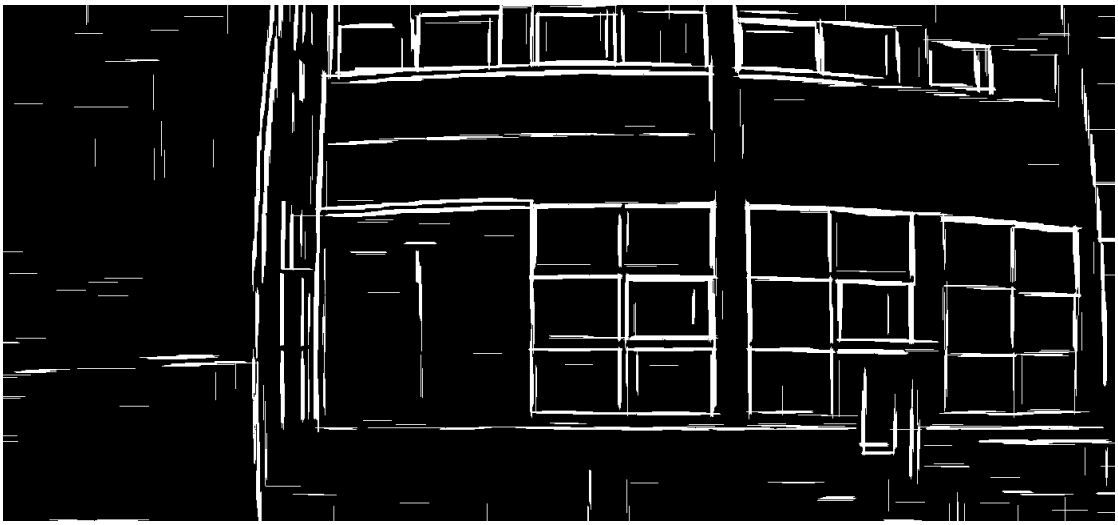


Figure 4.17 *Double morphological open operation using horizontal and vertical line structuring element.*

The morphological open operation removed all salt and pepper noise from the image, but has also left some of the window frame "incomplete". In the next step, the closure of the window frame is conducted. This is achieved with the morphological close operation, again with horizontal and vertical line structuring elements. It is implemented similar to the open operation, with the exception of the order of the `erode` and `dilate` functions. This process closes a number of "incomplete" window frames, especially on the ground floor of the facade, seen on fig. 4.18.

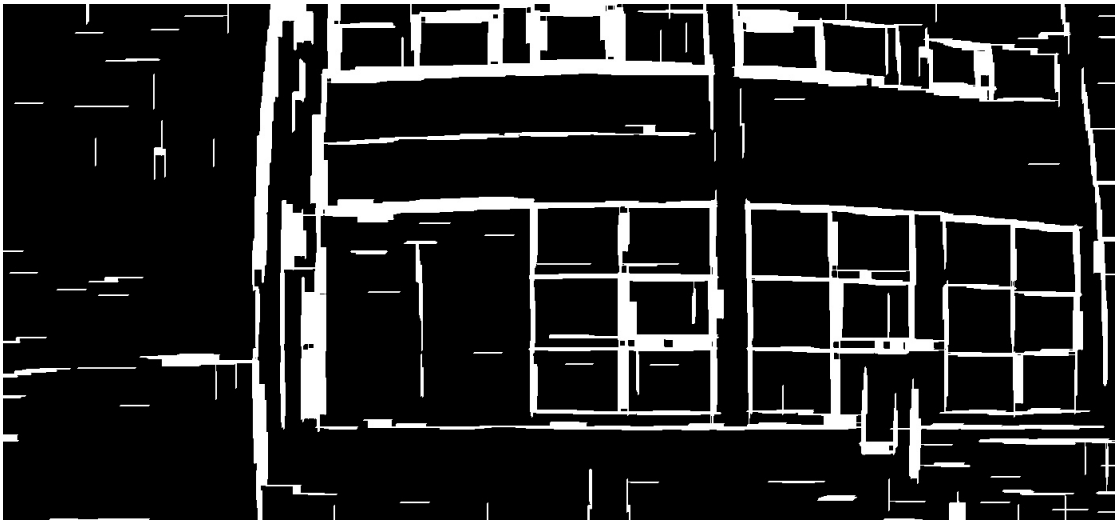


Figure 4.18 *Double morphological close operation using horizontal and vertical line structuring element.*

Next section describes the final result of the input image. It will show how the image looks when windows are detected, and explain some causes why few of them are not detected.

Detection with rectangle contours

Finally, this part will cover the result of the input image which has some rectangle contours for windows. Even though not all windows are detected, most of them are detected (See fig. 4.19).

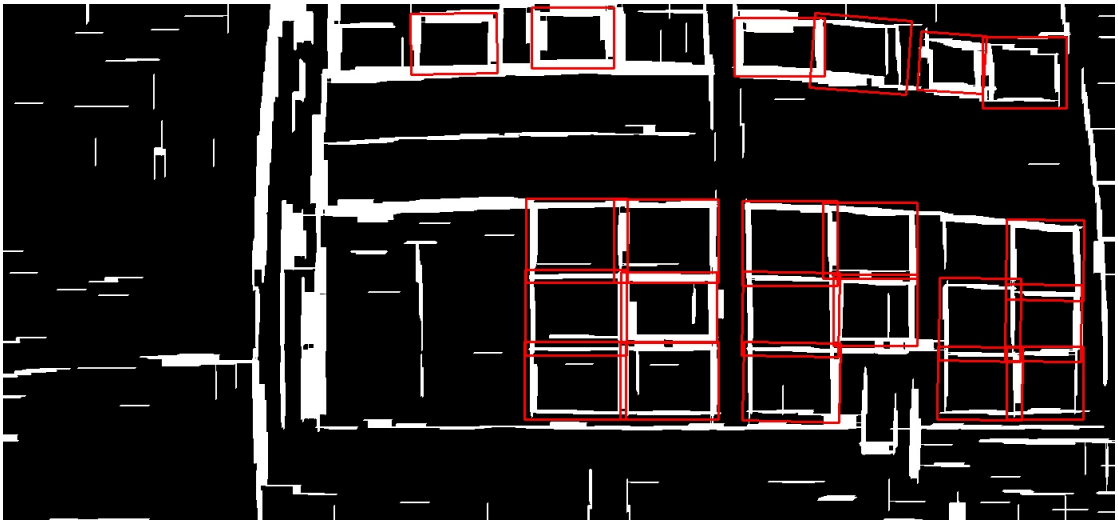


Figure 4.19 *This screenshot shows the final result of the image processing pipeline. The red rectangles around the windows are found by the use of hierarchical contours in the image.*

Red boxes are the detected ones, and green ones are missing ones (See fig. 4.20). By using double edge detection kernel, the thickness of the frame can be a cause of a missing window. Green boxes' frame have some thin frame so that they couldn't be detected.

Figure 4.20 *fig. 4.19 with all missing windows manually drawn with in green rectangles.*

Moreover, some of them are disconnected so, they are not completely rectangle. In addition, because of the lamp in front of the window, one of the lowest windows is not detected. Lamp and the window are overlapped in the image, so it was hard to be detected. On the other hand, from the morphology step, one of our members had been erased so that there was no problem for detecting the window which our member was

in.

Consequently, conditions for detected windows are two things. First, fulfil the complete rectangle. Second, have a enough thickness for window frame.

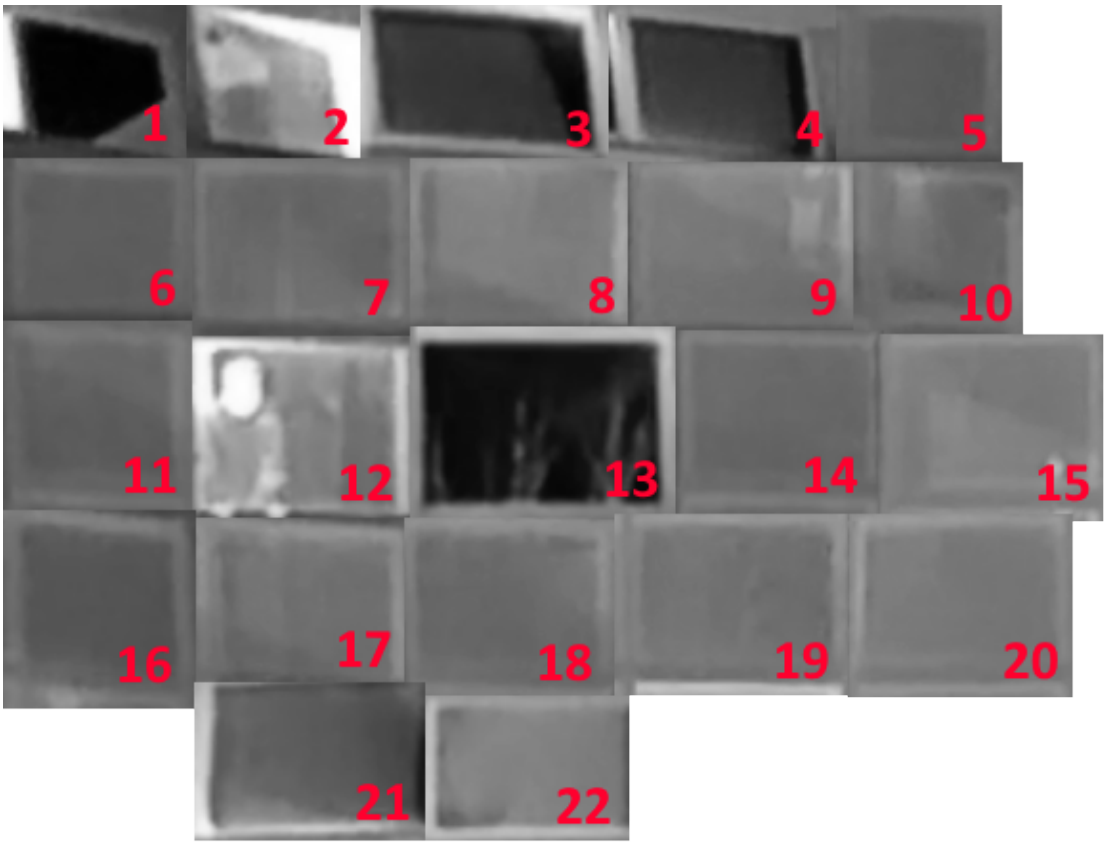


Figure 4.21 *All detected windows from fig. 4.19 cropped, from right before the pre-processing step.*

The actual detection of these closed window frame rectangles happens with the help of rectangle contours and the OpenCV function `findContours()`, seen used in code snippet 4.3.

```
1  Mat output;
2  int padding = 20;
3  std::vector<std::vector<Point>> contours;
4  std::vector<Vec4i> hierarchy;
5  std::vector<RotatedRect> rects;
6
7  findContours(input, contours, hierarchy, RETR_CCOMP, CHAIN_APPROX_NONE);
8
9  for (int i = 0; i < contours.size(); i++) {
10
```

```
11      if (hierarchy[i][2] > 0) continue;
12
13      RotatedRect rr = minAreaRect(contours[i]);
14      if (maxAreaSize == -1) {
15          if (rr.size.area() < minAreaSize)
16              continue;
17      } else if (rr.size.area() < minAreaSize || rr.size.area() > maxArea
            Size)
18          continue;
19      else if ((rr.size.height * 2) <= rr.size.width || (rr.size.width * 2)
             <= rr.size.height)
20          continue;
21      else if (abs(rr.angle) > 10 && abs(rr.angle) < 80)
22          continue;
23
24      rr.size.width += padding;
25      rr.size.height += padding;
26      rects.push_back(rr);
27  }
28
29  drawRectBoundingBoxes(contourOutput, output, rects);
```

Code 4.3 *OpenCV implementation of detection of closed rectangular contours.*

That concludes the window detection, and outlines a viable configuration for the final algorithm.

Final algorithm steps:

- Histogram stretching of the input image.
- Blurring with Median 3×3 blur.
- "Double edge detection" Laplacian With Diagonals.
- Binary threshold ($threshold = 1$).
- Morphology x and y close operations.
- Morphology x and y open operations.
- Find closed rectangular contours.

The next section will discuss about the window state classification, that will take place after each successful window detection (See fig. 4.21).

4.3.3 Window state classification

In this section the classification of window states will be discussed.
The classification is the last part of the pipeline, as this is where an object is finally given a class, based on the features of the object in comparison to the training data. Therefore, it happens after the detection of an object. To make this distinction clear, a

clarification of detection vs. classification will be given.

Detection is when the input is an image with zero or more objects in it, and the wanted output being a box surrounding each of the objects. The outputted box is just a bound for the classifier to work within.
Classification is, in contrast, when a small part of an image is given and this image should be classified as one of several classes. One of these classes could be "None" to signify anything which is not one of the wanted classes.

Some classifiers can also be used in regression problems, with the KNN classifier being one of these. In a regression problem, the output is not a number representing a particular class of objects, but instead a number on an arbitrary scale with some particular meaning. In the context of window states, it could be a floating point between 0 (for fully closed) and 1 (for fully open). One, could of course put these results into buckets containing: closed, slightly open, open and wide open. This would apply the output from the regression problem to create a classification in the end.

Given the constraints, only the binary classification will be used. In the second part of the pipeline, where the windows are classified it is assumed that the input is in fact a window. This assumption is necessary as it is otherwise not possible to do the classification with the data that is available at the moment. The point here is that, if the only labelled data available consists of windows that are open or closed, then the classifier will not be able to classify potential windows as anything else than windows. This effectively puts a lot more importance on the quality of the detector algorithm used.

If time, and access to the university, were not restricted then the dataset would have included examples of other things, that could realistically be found on or in front of a facade. This would allow for the classifier to classify objects with features similar to windows as windows, while classifying objects that are very unlike windows as other objects. Such data would allow the classifier to be more reliable, as it would then be able to output true negatives, which is otherwise not possible.

Chapter 5

Integration

In this chapter, the creation of the dataset will be described as well the final pipeline algorithm from the design section, is also implemented here.

The full source code for both the window detector and classifier can be found here:
`https://bitbucket.org/aauin6/cv-pipeline/src/master/`

5.1 Dataset

The current dataset was made in the beginning of the project where it was not entirely clear what the dataset needed to contain. Moreover, only a single facade was filmed at the university (the one seen on fig. 4.1). These things combined made for a dataset that lacks quantity as well as quality. One thing that was not considered when filming the facade was how steady the camera should be. It turns out that to properly utilise the automation algorithms available, a fairly stable camera setup is needed. The dataset should also have included objects found on or near facades. The wider the range of different objects and the more variation within each category, the better the chance of some test data being similar to some of the training data.

However, to make use of such a large dataset with several different kinds of objects, the dataset would need to be manually labelled. In the project the "Videolabeller" app in Matlab was used. While it did make it possible to create labels, and partially automate the process, it was a surprisingly time consuming process. This also considers the amount of time spent on labelling sessions that were corrupted and user error due to lacking safety. If large scale labelling efforts are needed, this Matlab app is not recommended. The only usable data from the video which is fully labelled, is a 31 second clip. This works out to 776 frames of video with up to about 20 windows. Even if this number of frames should be enough to be usable, it should be noted that most of these frames are fairly similar, whereas about 700 images in a generic dataset would probably not be very similar. Hence, the dataset does not have a great deal of variation in it.

5.2 Classifier

In getting the classifier ready to classify images, a few steps are needed. Firstly, the labels for the training data has to be loaded. This is handled by the `readAllFiles()` function in the `csvReader.cpp` file.

Secondly, the labels have to be correlated with the respective contents within the label in each labelled area of each video frame. The `correlateFrames()` function handles this need. As part of this correlation, each loaded frame from the video must be processed as any other input to the classifier would be. Otherwise, the training data would not look like the input data, and therefore not have the same features. This is ensured by the call to the `singleVisionPipeline()` function.

Thirdly, the features of all the blobs extracted must have their features calculated. This is done by the function `featureCalculatorForWindowBlobs()`.

Fourthly, the KNN classifier object can be constructed. As part of that, the calculated features of all the training data are normalised.
Lastly, a window can be classified using the `classifyWindow()` function.

5.2.1 Correlating frames

Here the `correlateFrames()` function, as seen in code snippet 5.2, will be explained.

Firstly, a `VideoCapture` object is created using the video's file path. It is then controlled, whether or not the video is correctly opened. If not, the program exits with an error message. A matrix to hold the video frames and vector to hold the output structs are initialised. All the labels are then iterated over, and for each iteration the next video frame is grabbed. If the current frame is not grey scaled (or equivalently has more than one colour channel), then it is grey scaled. The frame is then sent to `singleVisionPipeline()` which applies the image processing routine (described in section 4.3.2) to the image.

The processed frame is then sent to the `representBlobsWithinLabelsAsVector()` to extract the BLOBs within the labels. It iterates through every coordinate inside the bounds of the label and saves the coordinates and value of each pixel belonging to a given label (which represents a window). The return type is a struct called `windowBlobs` which contains all labelled windows in a frame. The open and closed windows are separated into two otherwise identical structs inside it. At the end of the loop the `windowBlobs` struct is pushed to the output vector.

```
1 std::vector<windowBlobs> correlateFrames(std::string fn, std::vector<
    frameData> labels) {
2     VideoCapture video = VideoCapture(fn);
3     if(!video.isOpened()) {
```

```
4         std::cerr << "The video file could not be opened, please check
              the path";
5         exit(1);
6     }
7     Mat frame;
8     std::vector<windowBlobs> videoBlobs;
9     for (auto& currentLabel : labels) {
10        video >> frame;
11        if (frame.channels() != 1)
12            cvtColor(frame, frame, COLOR_RGB2GRAY);
13        Mat processedFrame = singleVisionPipeline(frame);
14        windowBlobs blobs = representBlobsWithinLabelsAsVector(
              processedFrame, currentLabel);
15
16        videoBlobs.push_back(blobs);
17    }
18
19    return videoBlobs;
20 }
```

Code 5.1 *correlateFrames() function: This function takes a video file path and the labels for this video. It returns a videoBlobs struct which contains all the content of the labels and their labels.*

```
windowBlobs representBlobsWithinLabelsAsVector(Mat input, frameData
    labels) {
    windowBlobs output;
    std::vector<std::vector<objectPixel>> closedWindows;
    std::vector<std::vector<objectPixel>> openWindows;
    std::vector<window> closedWindowLabels = labels.closedWindows;
    std::vector<window> openWindowLabels = labels.openWindows;

   for (auto& window : closedWindowLabels) {
        int xSize, ySize;
        xSize = window.lowerRightCornerX - window.upperLeftCornerX;
        ySize = window.lowerRightCornerY - window.upperLeftCornerY;
        std::vector<objectPixel> currentWindow;
      for (int i = window.upperLeftCornerX; i < window.lowerRightCornerX;
          i++) {
         for (int j = window.upperLeftCornerY; j < window.
             lowerRightCornerY; j++) {
            objectPixel pixel = { i, j, (int)input.at<uchar>(j, i) };
            currentWindow.push_back(pixel);
         }
      }
   closedWindows.push_back(currentWindow);
   }

    // Repeat for open windows //

    output.closedWindowBlobs = closedWindows;
    output.openWindowBlobs = openWindows;
    return output;
}
```

Code 5.2 *correlateFrames() function: This function takes a video file path and the labels for this video. It returns a videoBlobs struct which contains all the content of the labels and their labels.*

5.2.2 Feature calculator

As the name suggests this function takes `windowBlobs` and returns their features. Specifically, it takes a vector of `windowBlobs` (which could represent an entire video clip) and returns a struct called `videoFeatures`, as shown at the beginning of code snippet 5.3.

As a `windowBlob()` represents all objects of interest in a frame, iterating over all of them, is effectively going through all the frames of the video. Inside this loop we iterate over all the closed windows. This should of course also be done for the open windows. The code for this however, has been cut out for the sake of brevity, as it is identical to the code for the closed windows. For each window a `windowFeature` struct is created and the `windowState` is set to closed (in the cut code it would be set to open). The features currently calculated are compactness and the ratio of width to height. To calculate the compactness it is necessary to know the number of white pixels and the

number black pixels, hence these are counted in the innermost loop. To calculate the W/H ratio, the highest and lowest row/col for each window needed. The values are initialised by the values of the first pixel of the object because this ensures that the values are within the bounds of the object. Then all the pixels are iterated over to count the pixels and identify the edges of the object. If a pixel value which is not 0 or 255 is found, it exits with an error message. After the inner loop, the compactness and W/H ratio are calculated. Note the float typecasting, this is necessary to ensure proper floating point division. Otherwise integer division is used, which reduces accuracy. Finally, the calculated values are put into the appropriate structs and returned.

```
1  videoFeatures featureCalculatorForWindowBlobs(std::vector<windowBlobs>
       frames) {
2      std::vector<std::vector<windowFeatures>> videoClosedFeatures;
3      std::vector<std::vector<windowFeatures>> videoOpenFeatures;
4
5     for (auto& frame : frames) {
6          std::vector<std::vector<objectPixel>> closedWindows = frame.
               closedWindowBlobs;
7          std::vector<std::vector<objectPixel>> openWindows = frame.
               openWindowBlobs;
8          int whitePixels;
9          int blackPixels;
10         std::vector<windowFeatures> closedWindowsFeatures;
11         std::vector<windowFeatures> openWindowsFeatures;
12         // Iterate over all closed windows
13         for (auto& window : closedWindows) {
14             windowFeatures currentWindowFeatures;
15             currentWindowFeatures.state = windowState::closed;
16             whitePixels = 0;
17             blackPixels = 0;
18             int lowestRow = window.at(0).row;
19             int lowestCol = window.at(0).col;
20             int highestRow = window.at(0).row;
21             int highestCol = window.at(0).col;
22             for (auto& pixel : window) {
23                 if(pixel.row < lowestRow)
24                     lowestRow = pixel.row;
25                 if(pixel.col < lowestCol)
26                     lowestCol = pixel.col;
27                 if(pixel.row > highestRow)
28                     highestRow = pixel.row;
29                 if(pixel.col > highestCol)
30                     highestCol = pixel.col;
31                 if(pixel.value == 255)
32                     whitePixels++;
33                 else if(pixel.value == 0)
34                     blackPixels++;
35                 else {
36                     std::cerr << "Non binary image detected!" << std::
                           endl;
37                     exit(3);
38                 }
```

```
39              }
40              float compactness = (float)whitePixels / (whitePixels +
                    blackPixels);
41
42              int width = highestRow - lowestRow;
43              int height = highestCol - lowestCol;
44              float widthToHeightRatio = (float) width / height;
45
46              currentWindowFeatures.features.push_back(compactness);
47              currentWindowFeatures.features.push_back(widthToHeightRatio);
48              closedWindowsFeatures.push_back(currentWindowFeatures);
49          }
50          // Iterate over all open windows
51          // -- repeat of previous code -- //
52          videoClosedFeatures.push_back(closedWindowsFeatures);
53          videoOpenFeatures.push_back(openWindowsFeatures);
54      }
55       videoFeatures res;
56       res.videoClosedFeatures = videoClosedFeatures;
57       res.videoOpenFeatures = videoOpenFeatures;
58       return res;
59  }
```

Code 5.3 *featureCalculatorForWindowBlobs() function: This function takes a vector of windowBlobs and calculates all the features of the windows. It returns all these features in a single videoFeatures struct.*

5.2.3 Data normalisation

In initialising the `knnClassifier` object, only one function is run: `trainingNormalizer()` with the `videoData` argument passed to the constructor, as seen in code snippet 5.4. The first thing done, is to set the minimum and maximum values of the calculated features. Here the maximum and minimum values are all initially set to the feature values of the first closed window in the first frame. This is again to ensure that the values are not too great or too small to be changed later. Then every frame is iterated over, and every closed window in the frame is then iterated over. This is repeated for the open windows (not shown in the code snippet). In the innermost loop all features of the individual window is iterated over and it is compared to the current maximum and minimum values of the feature, and set if higher/lower.
In the last loops the actual normalisation takes place. Using the formula:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{5.1}$$

The features are normalised such that they are all between 0 and 1. Then the normalised data is set as the training data in the classifier object.

```
1 knnClassifier::knnClassifier(videoFeatures videoData) {
2     trainingNormalizer(videoData);
3 }
```

```
4
5  void knnClassifier::trainingNormalizer(videoFeatures videoData) {
6      // Normalisation formula
7      // x' = (x - x_min) / (x_max - x_min)
8      int numberOfFeatures = videoData.videoClosedFeatures.at(0).at(0).
           features.size();
9
10     for (int i = 0; i < numberOfFeatures; i++) {
11         maximumValues.push_back(videoData.videoClosedFeatures.at(0).at(0)
               .features.at(i));
12         minimumValues.push_back(videoData.videoClosedFeatures.at(0).at(0)
               .features.at(i));
13     }
14     // Iterate over every frame
15     for (int i = 0; i < videoData.videoClosedFeatures.size(); i++) {
16         // Iterate over every closed window
17         for (int j = 0; j < videoData.videoClosedFeatures.at(i).size(); j
               ++) {
18             std::vector<windowFeatures> windows = videoData.
                   videoClosedFeatures.at(i);
19             for (int win = 0; win < windows.size(); win++) {
20                 std::vector<float> windowFeatures = windows.at(win).features
                       ;
21                 // Iterate over every feature
22                 for (int k = 0; k < numberOfFeatures; k++) {
23                     if (windowFeatures.at(k) < minimumValues.at(k)) {
24                         minimumValues.at(k) = windowFeatures.at(k);
25                     }
26                     else if (windowFeatures.at(k) > maximumValues.at(k))
                           {
27                         maximumValues.at(k) = windowFeatures.at(k);
28                     }
29                 }
30             }
31         }
32         // Iterate over every frame
33         // -- repeat of previous code -- //
34     }
35     float xmin, xmax;
36
37     // Iterate over all features of all closed windows in all frames
38     std::vector<std::vector<windowFeatures>>& allFramesClosed = videoData
           .videoClosedFeatures;
39     for (auto& frameClosed : allFramesClosed) {
40         for (auto& window : frameClosed) {
41             int i = 0;
42             for (auto& feature : window.features) {
43                 xmin = minimumValues.at(i);
44                 xmax = maximumValues.at(i);
45                 feature = (feature - xmin) / (xmax - xmin);
46                 i++;
47             }
48         }
```

```
49      }
50      // Iterate over all features of all open windows in all frames
51      // -- repeat of previous code -- //
52      trainingData = videoData;
53  }
```

Code 5.4 *trainingNormalizer() function: This function takes a* `videoFeatures` *struct and normalises the features in it. To do this, it first finds the maximum and minimum values of every feature, and then calculates the normalised features. These normalised values are set as the training data in the classifier object itself.*

5.2.4 Classify window

The inputs to `classifyWindow()` are an image and a number `kNearest` that defines how many neighbours to count in the voting. The image is first processed in `testDataProcessor()` which returns a `windowFeatures` struct. This struct is then passed to the `inputNormalizer()` which returns the normalised `windowFeatures` struct. These two functions are analogous to previously explained feature calculator and the data normaliser. Of course here, only a single input needs to be treated.
A vector of `distanceToTestData` structs is needed as the `distanceToTestData` represents the euclidean distance from a training data point of a given type, to the test data. Iterating over all frames and all windows in each frame, in the innermost loop, the difference between each feature in the test data and the training data is calculated and squared. After this loop, the total distance is set to the square root of the distance and the distance is put in the vector. After repeating the process for the open windows, the distances vector is sorted.
The k closest `distanceToTestData` structs are picked out and their types define which class their vote goes to. Finally, the class with the most votes is returned, represented by an integer.

```
1  int knnClassifier::classifyWindow(Mat window, int kNearest) {
2      windowFeatures testDataFeatures = testDataProcessor(window);
3      testDataFeatures = inputNormalizer(testDataFeatures);
4
5      std::vector<distanceToTestData> distances;
6      // Calculate distance to objects
7      // Iterate over all closed windows
8      for(auto& frame : trainingData.videoClosedFeatures) {
9          for (auto& closedWindow : frame) {
10             distanceToTestData dist;
11             dist.objectClass = 0;
12             float distance;
13             int i = 0;
14             for (auto& feature : closedWindow.features) {
15                 distance += std::pow(feature - testDataFeatures.features.
                       at(i), 2);
16                 i++;
17             }
18             distance = sqrt(distance);
```

```
19              dist.dist = distance;
20              distances.push_back(dist);
21          }
22      }
23      // Iterate over all open windows
24      // -- repeat of previous code -- //
25
26      // Sort distances to objects
27      std::sort(distances.begin(), distances.end(), compareDistances);
28
29      // Access the K closest objects and their classes
30      int class0 = { 0 }, class1 = { 0 };
31      for (int i = 0; i < kNearest; i++) {
32          if (distances.at(i).objectClass == 0)
33              class0 += 1;
34          else if (distances.at(i).objectClass == 1)
35              class1 += 1;
36      }
37      // return the test class based on a majority vote
38      std::cout << "The number of class 0 counts is: " << class0 << "\n" <<
            "The number of class 1 counts is: " << class1 << std::endl;
39      if (class0 > class1)
40          return 0;
41      else if (class1 > class0)
42          return 1;
43      else {
44          std::cerr << "Both classes have equally many votes, result
                unclear";
45          std::cerr << "Please use an uneven K value, to avoid this";
46          exit(1);
47      }
48  }
```

Code 5.5 *classifyWindow() function: This function takes in a matrix representing an image and an integer defining how many neighbours to count. It calls other functions to extract features and normalise them. It finally calculates the distance between the test and training data, counts the k nearest neighbours, and returns the an integer representing the class with the most votes.*

Chapter 6

Test

In this chapter the individual components of the pipeline are appreciated individually, as well as a whole.
As a disclaimer, it should be noted that the actual pipeline performance is not easy to verify, given the quality of the dataset at hand. Therefore, this section will deal with qualitative tests rather than quantitative tests.

6.1 Image processing

It has allowed for image processing interactively, which has greatly helped figuring out the most useful settings for detecting the windows.

6.2 Classifier

The classifier and the supporting code has been tested for sane outputs during the development process. While automated tests were never made, the individual components seem to work as intended.

6.3 Pipeline as a whole

The test of the entire pipeline that has been done, is to pass 22 images (from the training data) to the pipeline to be processed and classified. The result is that every single window has 5 votes for them being closed and zero for any of them being open. While this is not the result that was hoped for (which would be number image number 2, 3, 12, and 13 (from fig. 4.21) being classified as open).
Despite the classifiers inability to correctly classify, the combination of the image processing and the classifier also seems to work as intended. However, given the lack of good data, it is hard to say whether everything actually works properly. The data could be causing the issue, but on the other hand, so could the code.

Chapter 7

Discussion

7.1 Build tools

Our choice of tools, that of C++ and CMake, has caused us some serious delays throughout the project.
While the idea of using tools that allow for high performance and cross platform compatibility are inline with the ideal scope of the project, it could be argued that it took some focus away from the image processing and classification theory, and put it on the tools instead. This experience might be of value later for us individually, but it might have somewhat limited what was accomplished in the project.

7.2 Classification

As can be seen in the test section (chapter 6) the pipeline did not perform at all when it was given images to classify. This could be due to coding issues, dataset issues or an image processing problem.

7.2.1 Image processing

The image processing done at this stage has been fine tuned to allow the detector to detect as many of the windows as possible. This has resulted in images where the windows have been quite clearly seen. While that is good for detection, it also seems to have made the open and closed windows so similar, that there is essentially no difference between them. Given this outcome the pipeline might need to be changed somewhat. Currently, an image that has been processed to allow for window detection merely has the areas of interest cropped out and the matrix is sent to the classifier. This means that the details in the image that were removed to allow for detection are still not present when sent off to the classifier. It might be prudent to instead crop out the relevant area of the original image, and send that to a separate image processing routine meant to enhance the differences between open and closed windows.

7.2.2 Features

To classify windows, it has been chosen to use features to represent them. This was deemed the most useful approach, as template matching was not giving useful results. The features calculated have been compactness and width to height ratio. While these two might be useful in describing a window generically, they might not be the most useful in telling open and closed windows apart. Therefore, one would do well to investigate the characteristics of open and closed windows and contrast them. This was already done here, but further investigation is encouraged.

7.2.3 Data set

When considering the overall performance of the pipeline, the dataset plays a significant role and should therefore be discussed. First of all the only data used is a single video containing one facade. This means that the test data pulled from the video be very similar to the training data. This should have given the classifier an unfair advantage in classifying data that it has already seen, but that does not even seem to be the case here. Even if it did perform well on this data, it would not be representative of how well the model generalises. To test how well it generalises one could test using a fraction of the available videos, one fourth as an example, and the rest for training. This would make the test a better indicator of the models ability to generalise. To accomplish this however, much more data would have had to be collected, not to mention labelled. If that had been the case however, the project might eventually have run into some problems regarding the runtime performance, and the memory requirements. This would be because of the chosen classifier being a KNN.

7.2.4 Classifier

Was the KNN classifier a good choice for this project? Given the amount of data that was put through it, the limitations of a KNN classifier did not become a problem. However, had a large dataset of this kind already existed, then a KNN would probably not have been the right choice. In the case of a large dataset being available a neural network would have been able to make use of much more data without slowing down the execution speed, given how the execution time does not scale with the amount of data fed to it. If the KNN was switched for a neural network, this network would still need to be designed for adequate performance (execution speed and classification accuracy).

Chapter 8

Conclusion

The purpose of this report is to explore the viability of detecting window states using a thermal camera. In the report it is detailed how it was possible to detect the majority of windows on a facade, but classifying them was not possible.
This is expected to be due to three things:

- Lack of a good dataset.
- Lack of sufficient features.
- Lack of appropriate image processing routine.

The lack of a high quality dataset is a large hurdle in developing a well functioning system, as creating a dataset is no small task. Moreover, no relevant dataset seems to exist at all.

Describing the open and closed windows with a sufficient amount of features is also something that remains to be done. Specifically, identifying features that are distinctive of open and closed windows respectively, is of great importance. So far, the project has only used two features, compactness and width to height ratio, which is assumed to not be enough.

Creating two separate image processing routines also seems to be needed, as the one developed in this project is useful for detecting windows, but seems to remove the differences between open and closed windows altogether. This is of course not beneficial.

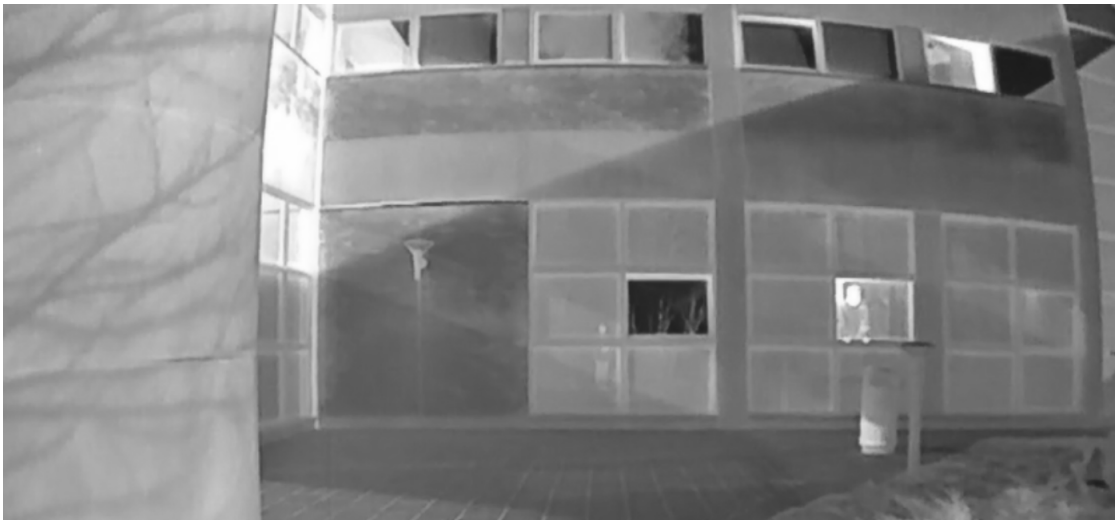
Appendix A

Appendix

A.1 Image processing - Full size screenshots

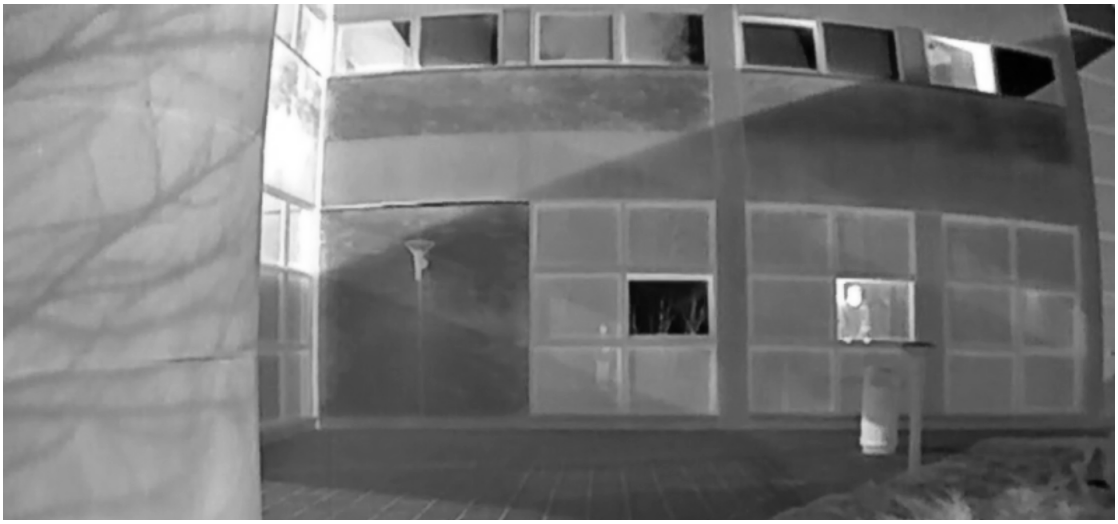
A.1.1 Histogram stretching screenshots

Raw grey-scaled image



Link: `https://i.imgur.com/mny1FrL.jpg`

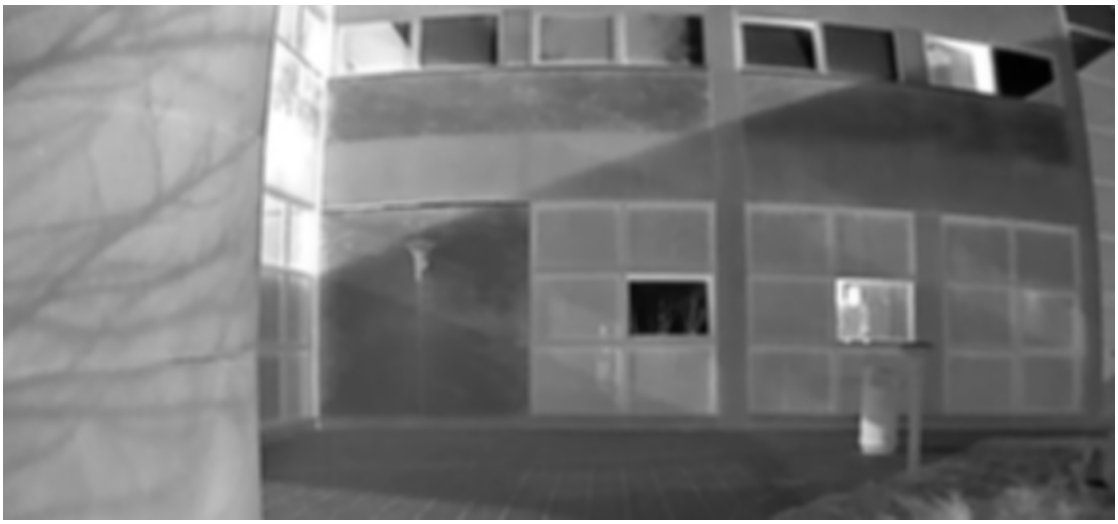
Image using histogram stretching



Link: `https://i.imgur.com/55zqWMU.jpg`

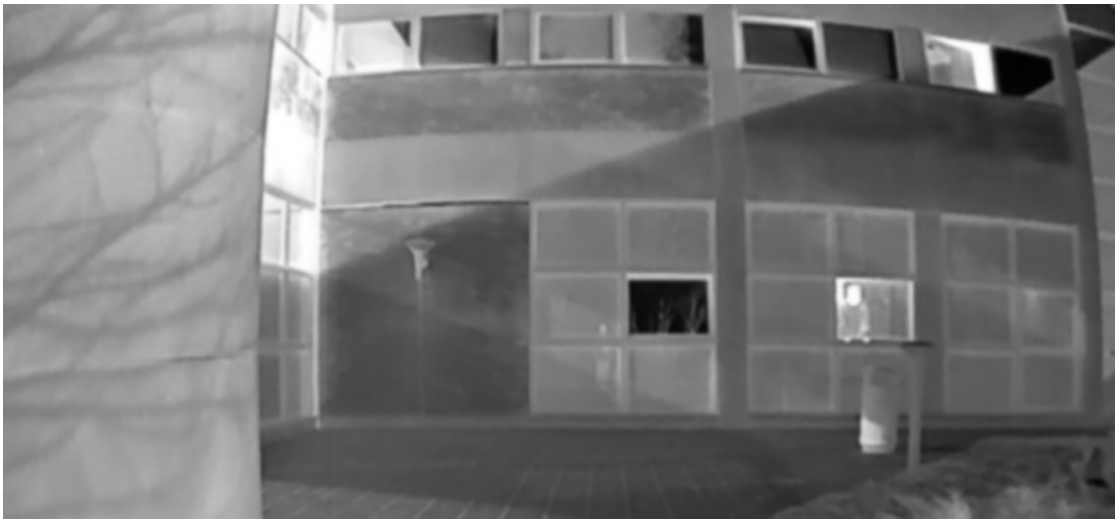
A.1.2 Blurring screenshots

Mean blur with 9 by 9 kernel



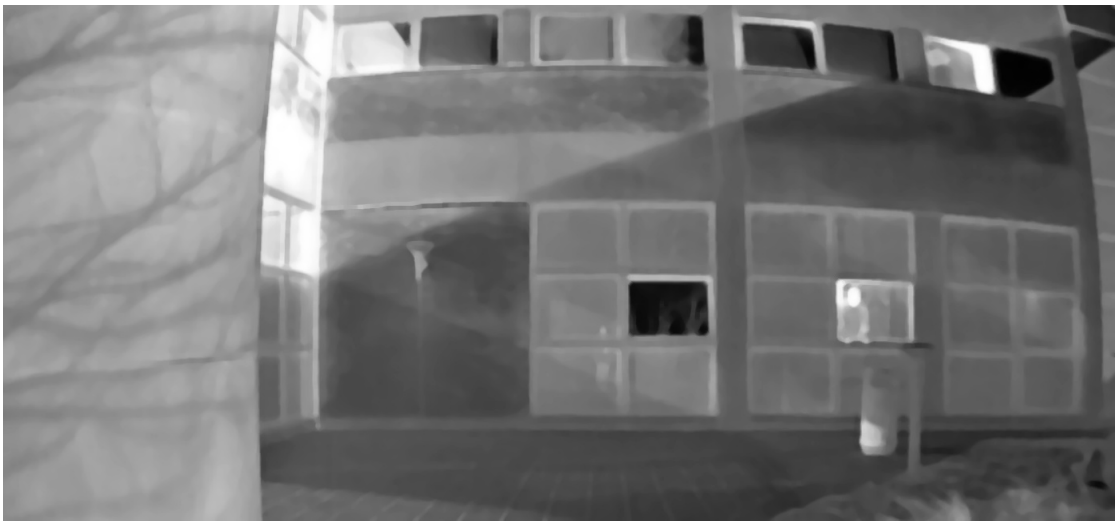
Link: `https://i.imgur.com/eASHqAW.jpg`

Gaussian blur with 9 by 9 kernel



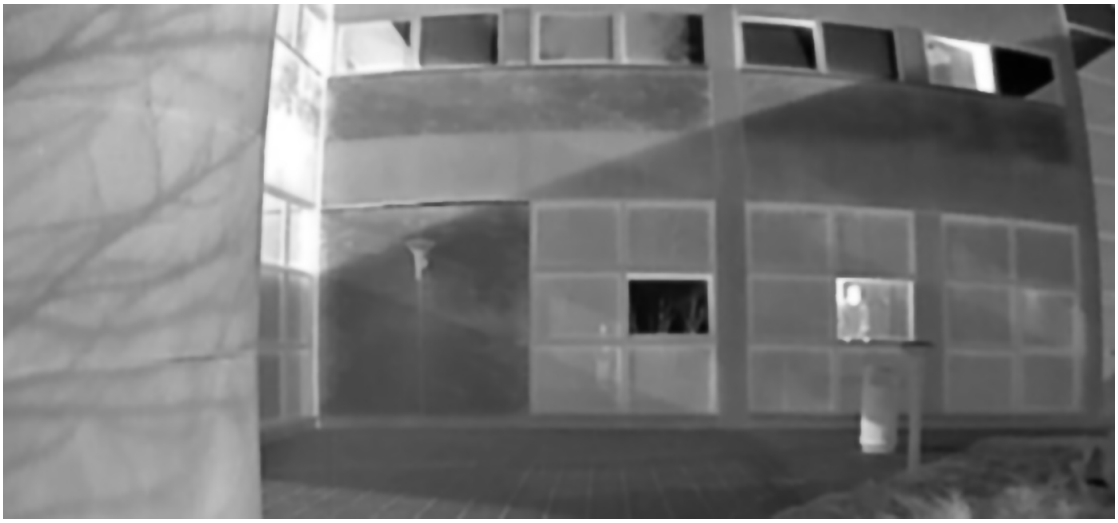
Link: `https://i.imgur.com/8f9pGRG.jpg`

Median blur with 9 by 9 kernel



Link: `https://i.imgur.com/QeqkkJE.jpg`

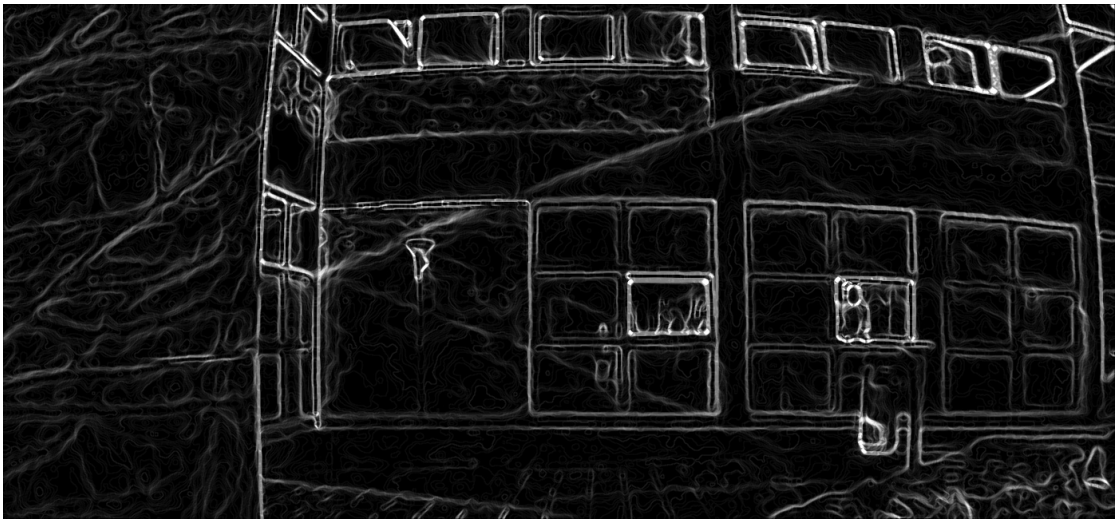
Bilateral blur with 9 by 9 kernel



Link: `https://i.imgur.com/Ix62FTI.jpg`

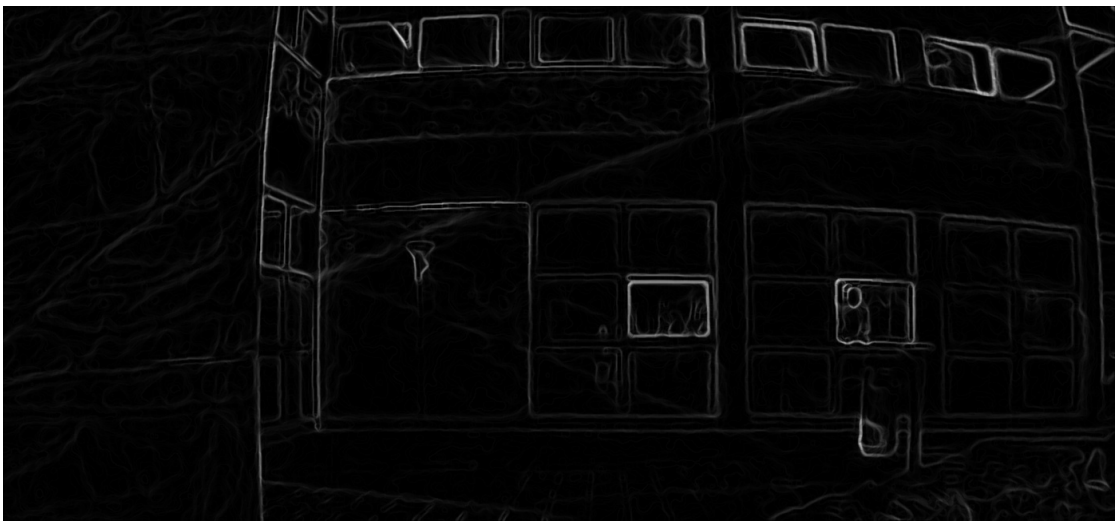
A.1.3 Edge detection screenshots

Edge detection with Scharr



Link: `https://i.imgur.com/Ik8Sp4K.jpg`

Edge detection with Sobel



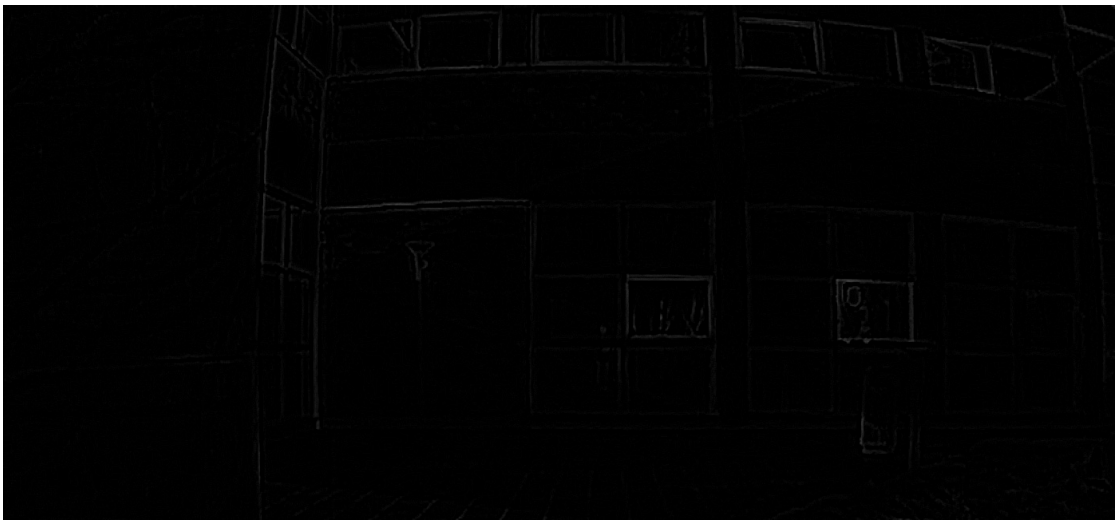
Link: `https://i.imgur.com/5ALfGgU.jpg`

Edge detection with Laplacian w/ diagonals

Link: `https://i.imgur.com/qMIUpqn.jpg`

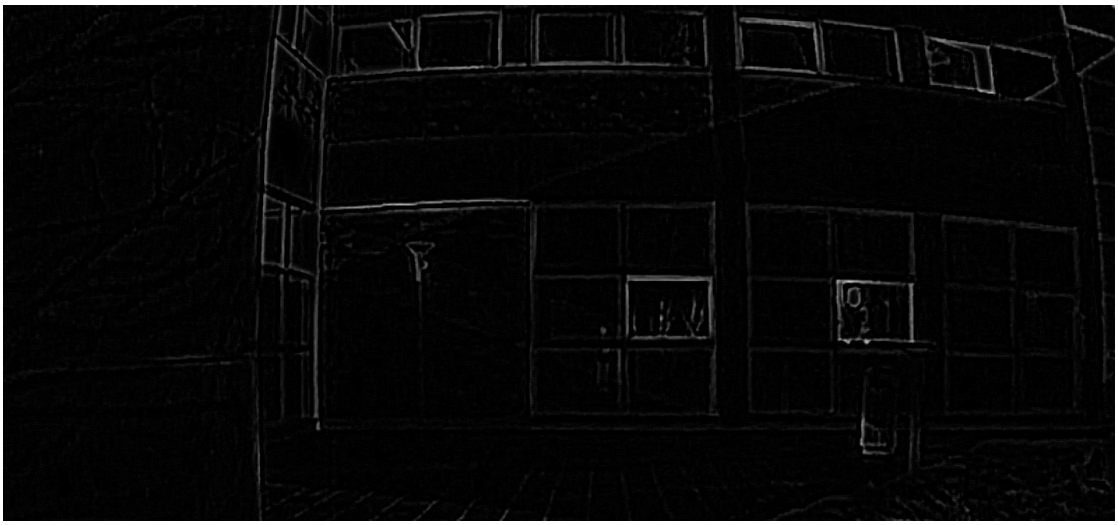
A.1.4 Edge detection (Laplacian) screenshots

Standard Laplacian with diagonals kernel edge detection



Link: `https://i.imgur.com/VJnNlbM.jpg`

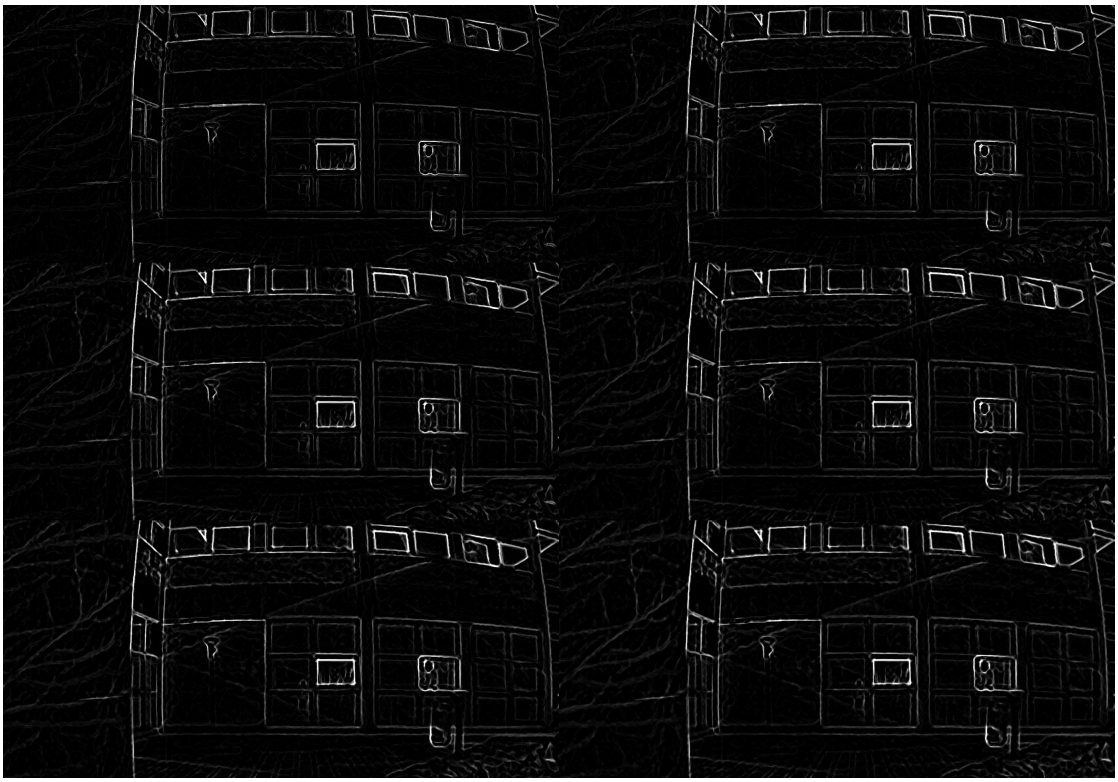
Double Laplacian with diagonals kernel edge detection



Link: `https://i.imgur.com/ewCzx0y.jpg`

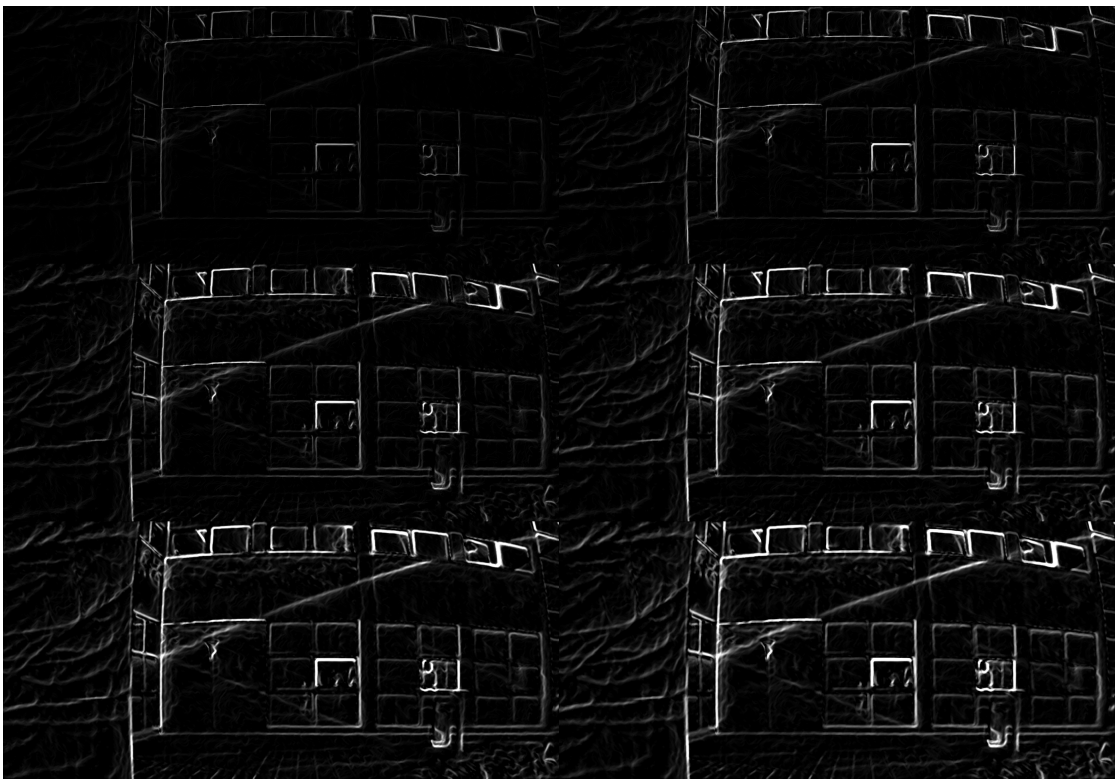
A.1.5 Experimental edge detection

Double edge detection test



Link: `https://i.imgur.com/BipFoiH.jpg`

Thick edge detection test



Link: `https://i.imgur.com/Qc5Cpew.jpg`

Chapter 9

Bibliography

[1] Stefano Corgnati Valentina Fabi, Rune Vinther Andersen and Bjarne W. Olesen. Occupants' window opening behaviour: A literature review of factors influencing occupant behaviour and models. *Building and Environment*, 58:188–198, 2012. doi: https://doi.org/10.1016/j.buildenv.2012.07.009.

[2] Fei; Cai Hao; Zhang Kai Zheng, Hengjie; Li. Non-intrusive measurement method for the window opening behavior. *Energy and Buildings*, 197:171–176, 2019. doi: https://doi.org/10.1016/j.enbuild.2019.05.052.

[3] G Branco; B Lachal; P Gallinelli; W Weber. Predicted versus observed heat consumption of a low energy multifamily complex in Switzerland based on long-term experimental data. *Energy and Buildings*, 36:543–555, 2004. doi: https://doi.org/10.1016/j.enbuild.2004.01.028.

[4] Andersen RV. Occupant behaviour with regard to control of the indoor environment. Master's thesis, Technical University of Denmark, Denmark, 2009.

[5] Pezcame. `https://tinyurl.com/td7ls7l`, N/A. [Image; accessed 16. March 2020].

[6] Jorn Toftum Stefano P. Corgnati Bjarne W. Olesen Rune Andersen, Valentina Fabi. Window opening behaviour modelled from measurements in Danish dwellings. *Building and Environment*, 69:101–113, 2013. doi: https://doi.org/10.1016/j.buildenv.2013.07.005.

[7] Home from the Future. Door / window sensor. `http://homefromthefuture.ae/door-window-sensor/`, N/A. [Online; accessed 24. February 2020].

[8] Pressac Communications. Door and window sensors. `https://www.pressac.com/door-and-window-sensors/`, N/A. [Online; accessed 24. February 2020].

[9] Safety.com. The best door and window sensors. `https://www.safety.com/door-window-sensors/`, N/A. [Online; accessed 24. February 2020].

[10] Arrow Electronics. What is a reed switch and how does it work? https://www.arrow.com/en/research-and-events/articles/the-reed-switch-ingeniously-simple-sensing/, 14 Aug 2018. [Online; accessed 24. February 2020].

[11] SafeWise Team. How does a window sensor work? https://www.safewise.com/home-security-faq/how-window-sensors-work/, 2020. [Online; accessed 21. February 2020].

[12] Onset Computer Corporation. Hobo pendant g data logger. https://www.onsetcomp.com/products/data-loggers/ua-004-64, N/A. [Online; accessed 11. March 2020].

[13] Home Stratosphere. https://www.homestratosphere.com/wp-content/uploads/2018/04/23_Types-of-Windows.jpg, N/A. [Image; accessed 12. March 2020].

[14] Jeff Haby. Snow cover and radiation budget. http://www.theweatherprediction.com/habyhints2/534/, N/A. [Online; accessed 3. March 2020].

[15] ImageJ. Imagej - an open platform for scientific image analysis. https://imagej.net/Welcome, 2018. [Online; accessed 16. March 2020].

[16] T. B. Gade, R. & Moeslund. Thermal Cameras and Applications. *Machine Vision & Applications*, 25(1):245–262, 2014. doi: https://doi.org/10.1007/s00138-013-0570-5.

[17] OpenCV. Morphological transformations. https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html, N/A. [Online; accessed 14. April 2020].

[18] caesar0301 (Github user). Awesome public datasets. https://github.com/awesomedata/awesome-public-datasets, N/A. [Online; accessed 15. April 2020].

[19] tutorialspoint. Histogram stretching. https://www.tutorialspoint.com/dip/histogram_stretching.htm, N/A. [Online; accessed 18. May 2020].

[20] Anonymous user on Quora. What are the advantages of gaussian blur, median blur, and the bilateral filter? https://www.quora.com/What-are-the-advantages-of-Gaussian-blur-median-blur-and-the-bilateral-filter, 2018. [Online; accessed 27. May 2020].

[21] Opencv. Image filtering. https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html?highlight=blur#blur, N/A. [Online; accessed 27. May 2020].

[22] Wikipedia. Median filter. https://en.wikipedia.org/wiki/Median_filter, 2019. [Online; accessed 27. May 2020].

[23] Wikipedia. Bilateral filter. https://en.wikipedia.org/wiki/Bilateral_filter, 2019. [Online; accessed 27. May 2020].

[24] Ery Arias-Castro, San Diego David L. Donoho, University of California, and Stanford University. Does median filtering truly preserve edges better than linear filtering? `https://arxiv.org/pdf/math/0612422.pdf`, 2009. [Online; accessed 27. May 2020].