Norwegian University
of Life Sciences

Master's Thesis 2017 30 ECTS

Faculty of Science and Technology
Ivar Maalen-Johansen

Brick wall analysis by photogrammetry and image processing techniques

Espen Johnsen
Geomatics
Faculty of Science and Technology

Preface

I would like to express my sincere gratitude towards my three supervisors: Ingunn Burud, Thomas Kringlebotn Thiis and Ivar Maalen-Johansen.
Ingunn Burud, I am very grateful for your continuous support throughout this thesis. Your insights and guidance has been of most importance to me.
I would also like to thank Thomas Kringlebotn Thiis for his invaluable contributions to this project. Thank you for showing interest in the progression of this thesis, it has been very motivating.
A special thanks to my teacher throughout these five years, Ivar Maalen-Johansen. Thank you for sharing your vast knowledge in photogrammetry, our conversations have been very helpful to me.

To Sigrid, thank you for all your kind-heartedness.

Finally, I would like to express my gratitude to Kristine for your unfailing support and encouragement throughout my years of study.

Espen Johnsen, Ås, 12. May 2018

Abstract

Deterioration of brick wall buildings can be seen as an effect from years of exposure to wind and weather. For example, through repeated freeze-thaw cycles, structural damage of the bricks might occur. In this thesis a method to automatically estimate the condition of a brick building is explored. Images of the building are captured with a Remote Piloted Aircraft System and then further processed to produce orthophotos of the facades of the building. Pix4D mapper is used as the photogrammetry software to produce the orthophotos. A script in Python programming language is developed to assess the condition of the walls. The script makes use of the Scikit-image library to implement several image processing techniques. The script is written in order to segment each brick in the orthophoto and classify them according to their condition. The segmentation is based on a series of image processing techniques, such as thresholding, binary morphological operations and labeling of binary regions. The classification of the bricks are based on entropy calculations.

The results show that a brick by brick segmentation and classification is possible. For one of the orthophotos in this project, an estimated 95% of the bricks seen in the orthophoto were identified and analysed. Weaknesses in the method are related to orthophoto quality and the scripts ability to accurately segment and classify the bricks. Improvements are explored and suggested for both issues. Different segmentation and classification methods are discussed. In addition the possibility of adding terrestrial sourced images to improve orthophotos is explored. Importance of camera and image quality, flight procedure and the need for radiometric consistency in the orthophotos are discussed as well.

Samandrag

Forvitring av mursteinbygningar kan sjåast på som ein effekt av mange års eksponering for vær og vind. Sprekkdanningar i mursteinane kan komme av at vatn som trenger inn i det porøse materialet ekspanderer når det frys til is. I denne oppgåva tar ein sikte på å estimera tilstanden til ein mursteinsbygning ved fotogrammetri og digital biletanalyse. Bilete av bygningen vart tatt med ein drone og ortofoto av fasadane vart produsert i Pix4D, ein programvare for fotogrammetri. Eit skript vart utvikla i programmeringsspråket Python for å estimera tilstanden til bygningen. I Python vart eit bibliotek, kalla Scikit-image, nytta til å implementera fleire bildebehandlingsprosedyrar. Tanken med skriptet var å segmentera ut kvar murstein som er synleg i ortofotoet, og dermed klassifisera mursteinen basert på kva tilstand den er i. Segmenteringa er basert på ein serie av bildeprosesseringsteknikkar som terskling, morfologiske operasjonar og identifisering av binære bilderegionar. Klassifiseringa av tilstanden til mursteinane er basert på entropiverdiar.

Resultatet visar att ein analyse er mogleg å gjennomføra på eit detaljnivå som tilsvarar ein analyse av murstein for murstein. Til dømes, i eit ortofoto i dette prosjektet har ein estimert at omtrent 95% av musteinane synleg i ortofotoet vart segmentert og klassifisert. Svakhetar ved metoden og resultatet er knytt opp til kvaliteten på ortofotoet og skriptets evne til å segmentera og klassifisera mursteinane nøyaktig. Forslag til forbetringar er utforska og føreslått for begge tema. Forskjellige segmenterings og klassifiserings algoritmar er diskutert, samt verdien av kamerakvalitet, bildekvalitet og behovet for radiometrisk kontroll.

Contents

1 Introduction

1.1 Brick buildings and deterioration

Depending on location and local climate, buildings must withstand deterioration from years of exposure to wind and weather. This is evident in brick wall buildings[11]. One of the damaging effects comes from wind-driven rain. Moisture from the rain can penetrate the porous material and expand upon freezing, damaging the brick wall. The damage effect is magnified with repeated freeze-thaw cycles[24].

The KA-building (Chemical Analysis, Figure 1.1) of the Norwegian University of Life Sciences serves as a great example of a brick building with bricks that vary from good to very poor condition. Therefore this building serves as a good training site for developing different methods to analyse the condition of brick walls in general. The size and placement of the building is shown in Figure 1.2.

Figure 1.1: The Chemical Analysis building

1.2 Previous work

In 2013, Kristian S. Førde tried to develop a toolkit to identify and recognize bricks with freeze-thaw damages[7]. Førde's thesis might serve as a good complementary text to read for anyone who wish to further investigate the challenge of classifying the condition of brick buildings.

On Pix4D official website, a project that bear some resemblance to this can be found[32]. In this project a facade inspection of Messina building in Italy was done. By capturing images with a drone (DJI Inspire 1+ FC350) of the building

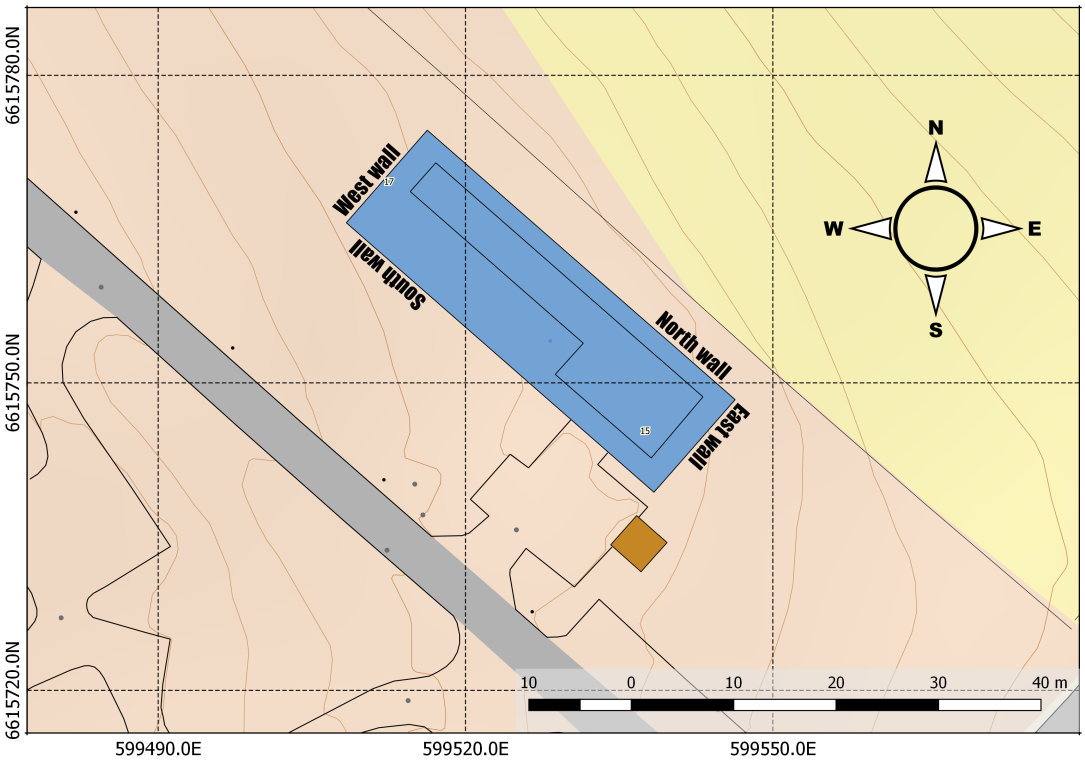

Figure 1.2: Location of the building. The walls are named accordingly and will be used throughout the thesis. Coordinate system of the map is WGS84/UTM 32N

facade, an orthophoto was produced in Pix4D software and a manual analysis was done in CAD software.

1.3 Aim of thesis

How could one automatically estimate the condition of a brick building, such as the Chemical Analysis building. By creating and analysing orthophotos of the brick walls, is it possible through image processing techniques to assess the condition of the building?

2 Software and Hardware

2.1 Pix4D mapper

Pix4D mapper is a proprietary software that make use of photogrammetry and computer vision techniques to produce, among other things, three-dimensional point clouds and Orthophotos. The main inputs of Pix4D are digital images in either JPEG or TIFF format. For TIFF it is possible to use infrared and thermal images as well as RGB images. Pix4D mapper Pro version 4.2.25 was used.

The main outputs of Pix4D are 3D point clouds, Digital Surface and Terrain Models, 3D textured models and Orthophotos. Other "intermediate" products such as calibrated camera parameters, undistorted images and more can also be exported. Pix4D allows for exporting the products in many formats[19]. This facilitates the possibility of further processing the data in other software such as AutoCAD, Qgis and Blender to mention a few.

The software can be installed as a desktop solution, but also offers processing in the cloud. One solution is to directly upload images to the cloud and automatically start processing without defining any processing parameters. A more robust solution is to define the processing options in the desktop version before uploading to the cloud. The results can be downloaded again and further improvements and processing can be done. For example, by adding manual tie points or defining orthoplanes to produce orthomosaics of facades[16]. In the desktop version it is also possible to define a processing area to minimize processing time.

2.1.1 Outline of processing steps

There are three main processing steps in Pix4D mapper:

1. Initial Processing
2. Point Cloud and Mesh
3. DMS, Orthomosaic and Index

For the purpose of creating an orthomosaic of the building facades, only step 1 and 2 are needed. In step 1, the software will try and match points across images (tie-points) and through automatic aerotriangulation and bundle block adjustment create a three dimensional point cloud, called ray-cloud in Pix4D. The second step will densify the point cloud from step 1. Typically, For a 14 megapixel camera and a project with 250 images, this will create points in the range of 20 - 50 million points. The coordinates x, y, z for each point is stored, as well as the color information for each point. Step 2 increases the density of the three-dimensional point cloud created in step 1. This should also increase the accuracy of the Digital Surface Model and the orthomosaic.

2.2 Python

Python is an object-oriented programming language. It was developed by Guido van Rossum and first published in 1991. The software is freely usable and distributable[22]. The script in this thesis was written with Spyder 3[29]. Spyder stands for "Scientific Python Development Environment". It comes with popular packages such as NumPy for linear algebra, SciPy for signal and image processing, and matplotlib for interactive plotting. Python version 3.6.1 was used with Spyder version 3.1.4.

2.2.1 Scikit-image

scikit-image[33] is a package used for image processing in Python. It extends the SciPy module and provides a set of image processing algorithms. Version 0.13.0 was used.

2.3 Adobe Photoshop, Camera Raw

Adobe Photoshop is a raster graphics editor. Camera Raw[1] is used to edit images to make adjustments to, for example, exposure and white balance.

2.4 Phantom 4

The phantom 4, seen in Figure 2.1, is a light-weight Remotely Piloted Aircraft System (RPAS), weighing 1380 grams. It has Global Navigational Satellite System (GNSS) capabilities and captures positional data from both GPS and GLONASS. It also has an Inertial Measurement Unit(IMU) and compass. The camera on the drone is attached with a Gimbal system to reduce effects of vibrations during flight. It also enables the camera to change viewing-directions from -90 to +30 degrees. The camera attached has a CMOS with a size of $6.17mm \times 4.55mm$, with a resolution of 4000×3000 pixels. The camera has a rolling shutter mechanism. Images can be recorded to a micro SD memory card in JPEG and DNG format.

Figure 2.1: Phantom 4. Image from the official website of DJI

3 Theory

3.1 Photogrammetry

Photogrammetry can be described as the science of making measurements from images. Measurements are used to derive placement, shape and size of the photographed object[5]. This is possible when the geometry of the situation during the exposure is re-established. For example the geometry of the internal parts of the camera (interior orientation), the position and attitude of the camera (exterior orientation) and the relative position between images (relative orientation).

Through careful measurements and an ingenious consideration of the geometric relationships between image and object coordinates it is possible to calculate the coordinates of the photographed object in three dimensional space, for example through triangulation. This information is used to produce products such as three-dimensional point clouds, Digital Elevation and Surface models and Orthophotos. The quality and precision of these outputs are often correlated with the success of establishing the parameters of interior, exterior and relative orientation.

3.1.1 Interior, exterior and relative orientation

When you take a photo, you capture the rays of light reflecting from the scene, through the projection center of the camera and onto a two-dimensional plane. Ideally this will fulfil the central projection and collinearity condition. With central projection it is meant that all light reflected from the scene converges in the central projection center of the camera before it is spreads out again on the image sensor plane. The collinearity condition means that through a given point P on the projection plane (the image sensor) a straight line can be drawn through the projection center and onto the object that reflected the light in the first place, see Figure 3.1.

The interior orientation relates to the geometry inside the camera, such as focal length, principal point, radial and tangential lens distortion. Some of these are seen in Figure 3.2

The exterior orientation parameters will place the position and attitude of the cameras projection center in a three-dimensional space. The position can be described as a vector: $[x, y, z]$, while the attitude if the camera regards the angles: $[\omega, \phi, \kappa]$.

The relative orientation concerns the relative position and orientation between, for example, two images. When these sizes are known, it is possible to triangulate the three-dimensional coordinates of object seen in the overlapping images.

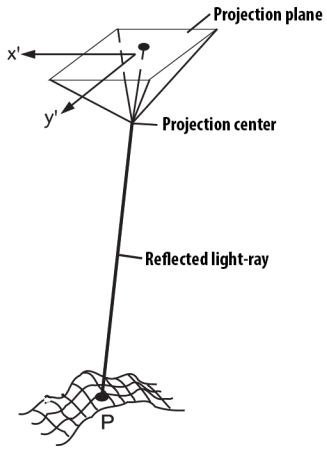

Figure 3.1: A geometric description of the relation between the projection plane, projection center and point P on the ground. Modified figure from [2].

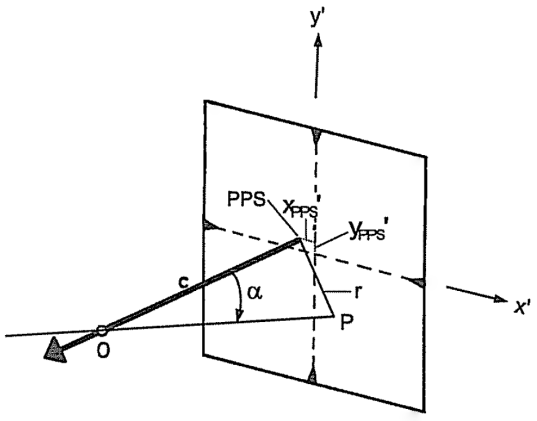

Figure 3.2: The image plane with projection center (O), focal length (c) and principal point (PPS). Modified figure from [2].

3.1.2 Orientation process

There are several ways to establish the orientation parameters described above. The process is highly dependent on the use case and the software involved. A short comparison between traditional fixed-wing aerial photogrammetry and RPAS photogrammetry can highlight some of the differences.

In traditional aerial photogrammetry the camera used will often be a high-end camera, with very large image sensors. The interior orientation parameters can be established before the flight, known as pre-calibration. Through precise GNSS and IMU instruments, good approximate exterior orientation parameters are established. The direction and overlap of the images are relatively ordered and taken in a nadir direction (orthogonal to the ground). With the addition of visible ground control points in the images, it is possible to correct and control for errors in the orientation process.
In RPAS photogrammetry the camera could be a lightweight consumer-grade camera, with a smaller image sensor. The interior orientation parameters are established during the orientation process, known as self calibration. The GNSS and IMU instruments are not as precise, but can be used as approximate values in the software. The orderliness of image overlap are possibly much more randomised, and the camera direction could vary from nadir to oblique.

These differences requires different methods for calculating the orientation parameters involved. For modern photogrammetry software, computer vision techniques are implemented to take advantage of increasingly larger amount of images with very high overlap percentage, as is often the case for RPAS projects. For example through implementing algorithms such as SIFT (Scale-invariant feature transform)[12]. The algorithm is used to automatically match key points across images to produce tie-points. Key points are features in an image with high contrast and interesting texture, wich makes it an ideal point to recognice in other images. Tie-points are essentially key points matched across different images. Tie-points and approximate values from GNSS and orientation instruments from the RPAS are used in a bundle block adjustment[30] to establish the model[10].

3.1.3 Orthophoto

In this thesis the term orthophoto relates to the orthomosaic produced in Pix4D with the orthoplane tool[17].

A major distinction between a photo and an orthophoto is the difference in projection. Typically a photo will have a central projection, which creates distortions as described in the previous section. An orthophoto will have an orthogonal projection, which gives the photo map like qualities. This facilitates the use of measurements in the image to be correct across the entire orthophoto. The projection is visualised in Figure 3.3.

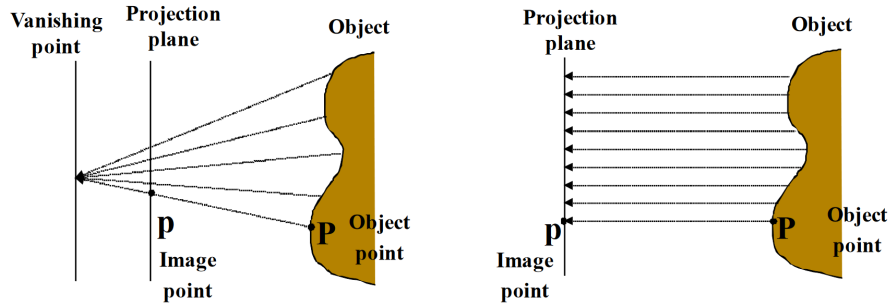

Figure 3.3: To the left we see the typical central projection found in most cameras. To the right is an orthogonal projection, same as the projections one can find in maps. Figure from [15].

3.1.4 Ground Sampling Distance

The ground sampling distance (GSD) is an important parameter regarding the final resolution of photogrammetry products. The ground sampling distance relates to the size of the image "footprint" on the ground/object during image acquisition. The GSD is dependent on the camera sensor size, focal length and the distance to the object photographed. The relation between distance and camera parameters can be seen in Figure 3.4. The higher the ground sampling distance is, the lower the spatial resolution of the image, hence less details are visible in the image.
To calculate the ground sampling distance, one needs to know the image sensor size (Sw), distance to object (H), focal length (Fr) and the image width in pixels (imW). The relationship is described in Equation 1. The ground sampling distance value relates to the length one pixel represents on the object photographed.

$$GSD\left(\frac{cm}{px}\right) = \frac{S_w(mm) \times H(M) \times 100}{F_r(mm) \times imW(px)} \tag{1}$$

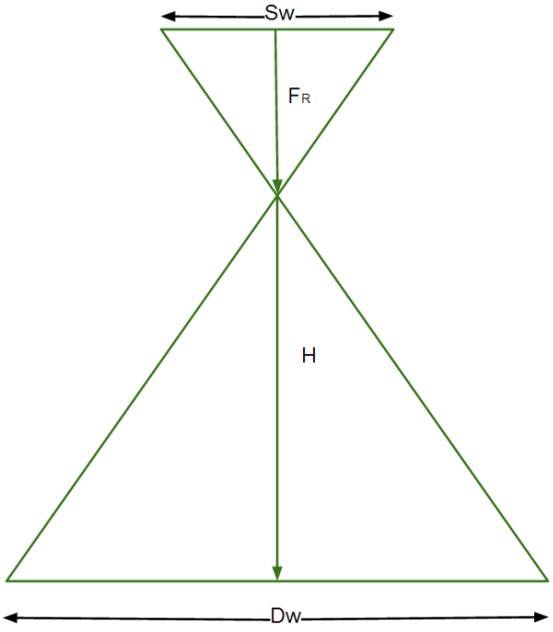

Figure 3.4: The relation between focal length and distance to terrain

3.2 Digital Image processing

In this section, the main characteristics of an digital image is introduced. Some terminology is defined and the main packages from scikit-image are described.

3.2.1 Digital image representation

A digital image can be described as a two-dimensional matrix where each element in the matrix is known as a pixel. The pixel value refers to the measured light intensity in that specific part of the image. For an eight-bit image the intensity values will be within the range of $0 - 255$. For a typical color image there will be a matrix for each primary color: red, green and blue. Stacked on top of each other, this will produce a pixel-depth of 24 bits. Which means that there are approximately 16 million different variations in color available, as described in Equation 2.

$$[0 - 255]^3 \times 3 \approx 16\,million \tag{2}$$

Another typical intensity range is that of a binary image. This image will only have two intensity values: $[0, 1]$. One can think of it as an image with only foreground and background elements. An important concept in binary images relates to the consecutive connection of foreground pixels. This can be described as a region in the binary image. Two regions are separate when there is no connecting foreground pixels between them. Whether or not a pixel is connected to its neighbouring pixels depends on the definition of the neighbourhood of the pixel. Usually divided in 4- and 8-neighbourhood. For all purposes in this thesis, the relation between pixels are in an 8-neighbourhood.

Figure 3.5: To the left a 4-neighbourhood, to the right an 8-neighbourhood

3.2.2 Thresholding

An image can be turned into a binary image through the process of thresholding. In this process, each pixel value (a) in the image will be reassigned into either 0 or 1, based on the threshold value (q). If the pixel value is less then the threshold value, it is reassigned as 0 (a_0). And for a pixel value equal or above the threshold value it is reassigned as 1 (a_1).

$$f_{threshold}(a) = \begin{matrix} a_0 & \text{for } a < \text{q} \\ a_1 & \text{for } a \geq \text{q} \end{matrix} \tag{3}$$

Usually the threshold value is constant while reassigning the pixel values of the entire image. However it can also be adaptive, meaning that the threshold value (q) will vary with the local variation of pixel values in a certain area around the pixel, as shown in Figure 3.6.

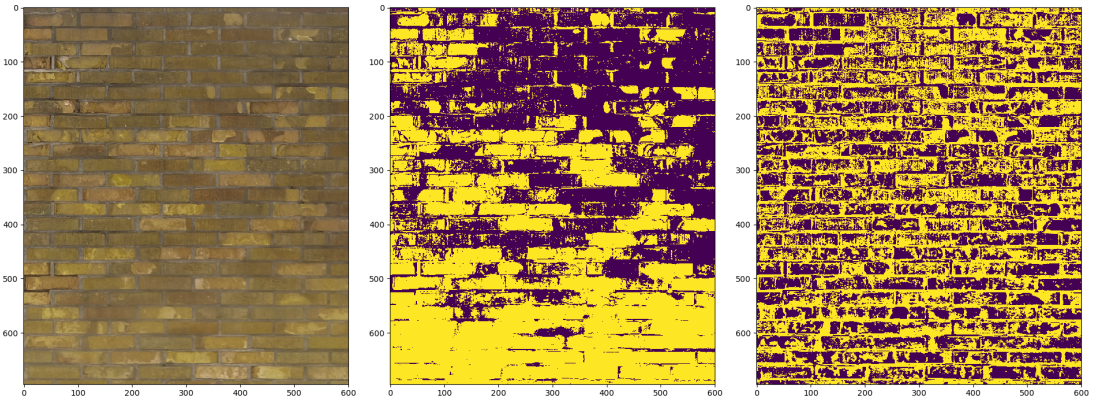

Figure 3.6: To the left is an RGB image. In the middle the threshold value (q) is constant and to the right the threshold value (q) is varying with the location in the image.

3.2.3 Morphology

Morphological operations are usually performed on binary images to alter the structure of the regions in the image. Usually a morphological operation will use a search window, known as a structure element, to decide how the morphological operation will manipulate the pixel based on the pixels within the search window.

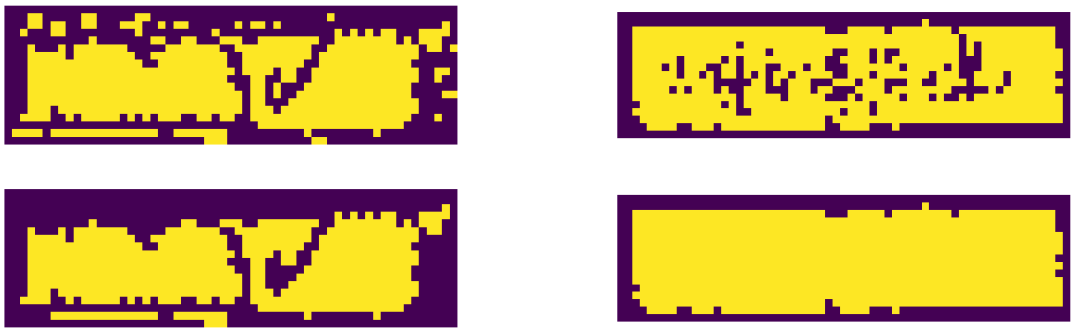

Figure 3.7: The two figures on the left side shows the effect of removing small objects. To the right we see how the fill holes method works. Upper: before morphological operation, lower: after morphological operation.

Two typical morphological operations are erosion and dilation.

With erosion, the regions in the image will shrink. This happens because the lowest intensity value within the structure element will be used to replace the center pixel. For dilation the opposite is true. This means that the center pixel will be replaced by the highest intensity value within the structure element. The erosion and dilation operators can be sensitive to the size and shape of the structure element used, and how the region in the binary image looks like. In the Scikit-image package, two similar operators can be found, they are not dependent on the size and shape of the structure element, but examines the properties of the regions in the image.

Two methods implemented by scikit-image in the morphology module are: *remove small objects* and *remove small holes*. The former will remove regions in the binary image, based on a threshold regarding minimum size of the region to be removed. The other function will remove small holes within a region, based on a threshold value concerning what size the hole has to be before removing it. The effect of both methods are shown in Figure 3.7

3.2.4 Labels and properties

Scikit-image has implemented some methods to work with regions in a binary image. Namely *label*[27] and *regionprops*[25] found within the *measure* module. As mentioned, a binary image will only have two levels, 0 and 1. With the *label* method[6][34], the binary image will identify each binary region and replace the values as an identification for connected regions. The algorithm can be described as a flood filling algorithm[3], where regions are marked by scanning the connection between consecutive pixel in either a 4- or 8-neighbourhood. The highest value in the resulting image now reflects the total number of regions in the original binary image.

The *regionprops* method will measure many properties of these binary labeled regions[25]. The main property of interest, used in the method, is the bounding box(bbox) property. The bounding box property will return four image coordinate values: minx, miny, maxx, maxy. This represents the lower left and upper right corner of the minimal bounding box for each labeled region in the binary image.

3.2.5 Entropy

The *entropy* method, from the scikit-image library[26], returns the minimum number of bits needed to encode the local grey-level distribution. Entropy is an important concept in theory of digital communication[28]. A simple example of entropy for digital images: Low entropy images will have large regions with similar pixel values, which essentially means there is little information present in the image. High entropy images will have a large variation in pixel intensity values, which means that there is more information available in the image.

4 Method

An attempt was made to produce orthophotos of the brick walls of the KA-building. These were made by first sourcing images with a Remotely Piloted Aircraft System, and then images were processed in Pix4D mapper to produce the orthophotos. Then a script was developed in Python to try and automatically extract information about the condition of the bricks visible in the orthophotos.

The method section describes each step taken from flight to image analysis. The procedure for the different steps in Pix4D mapper are extensively documented in the manual[20] and on the Pix4D support site[21]. The procedure and thought process behind the development of the script is explained in more detail.

4.1 Image acquisition

The images were acquired with a Phantom 4 using the on-board camera. During flight the application DJI GO was set to record images every 2 seconds. Camera settings were set to automatic (ISO, White balance, Exposure). The flight was done manually and a high overlap between images was aimed for. The camera was oriented towards the walls of the building, some example images can be seen in Figure 4.1. In total 929 pictures were captured.

Figure 4.1: A selection of three images captured during flight

The images were recorded to the on-board micro SD card. They were stored as JPEG files with a resolution of 3000×4000 pixels. The Phantom 4 also stored information about camera settings, geographical position and gimbal orientation parameters to the metadata section of the JPEG files. An example of how the position and orientation data from one of the JPEG files are shown in table 1.

Table 1: A selection of JPEG metadata

GNSS	Latitude	59.667735
	Longitude	10.766825
Gimbal	Yaw	80.1
	Pitch	-35.1
	Roll	0.0

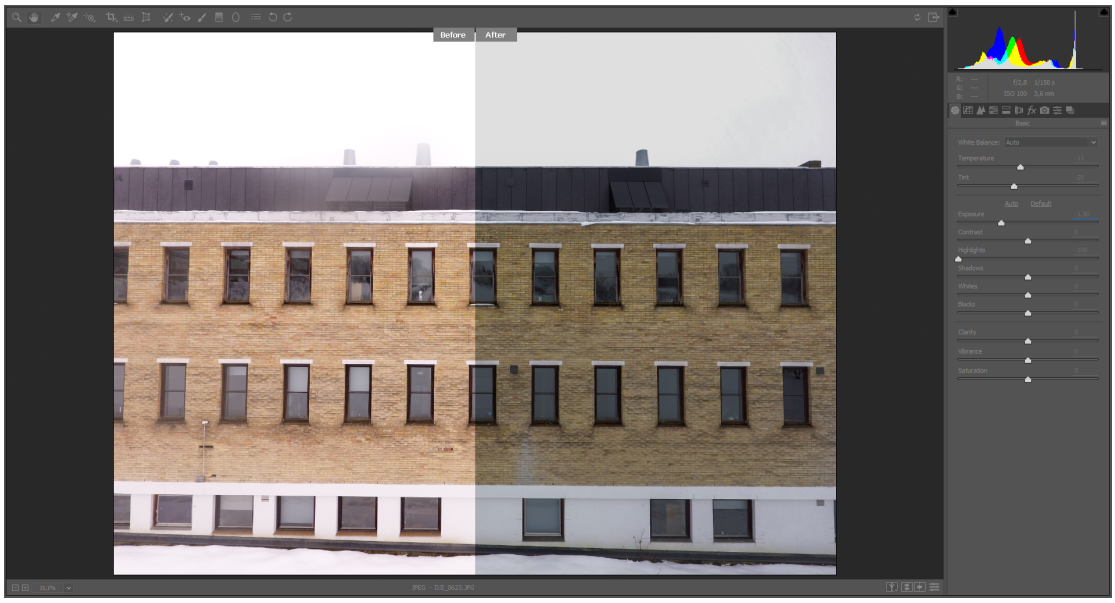

Figure 4.2: Screenshot from Adobe Camera raw plug-in. Exposure and white balance adjusted

Image selection and pre-processing

After the flight, images were transferred to a computer from the SD card. Then a selection of 322 images was done. The selection was done to exclude images captured during take-off and landing. Because the building was not in the frame during take-off and landing, these images would not contribute to the modelling of the building. Also images that had extremely high overlap and little to no variation in position x, y, z was removed. This was done because images which are captured with no variation in position, with some variation in yaw, pitch and roll is in general not the best image acquisition procedure in photogrammetry projects. These selections of images including their metadata were the inputs for processing in Pix4D mapper. Before loading the images into Pix4D mapper, they were edited in Adobe camera raw software to make exposure and white balance consistent across images, as seen in Figure 4.2. After the adjustments were made, the files were saved in JPEG-format and all metadata was included. This ended the image acquisition, selection and editing phase. This formed the inputs of a new project in Pix4D.

4.2 Pix4D mapper

A new project was made in the Pix4D mapper, from the desktop version. The 322 images were added to the project and Pix4D established the geolocation and orientation of all the images from reading the metadata. The coordinate system was recognized as WGS84 / UTM Zone 32N. The geolocation of the images was established before any processing steps was performed, as can be seen in Figure 4.3. The red area illustrates the processing area which was done by manually drawing it inside the Pix4D workspace[18].

Figure 4.3: Pix4D mapper workspace. The red square represents the processing area defined. The red circles are the approximate positions of the images captured.

Before uploading the project for processing in the cloud, processing options were adjusted. Settings used can be seen in the appendix, Figure A.1 and A.2. After selecting for processing step 1 and 2 (step 3 not chosen), the project was uploaded to the cloud for processing. After processing, the project is exported and downloaded to the desktop again for further inspection. The result of step 1 and 2 is seen in Figure 4.4.

Manual tie points (MTPs) were added in order to improve the point cloud and matching between images. MTPs were added to each top and bottom corner of the building facade. Improvements from adding MTPs was taken into account by re-optimizing the project. The re-optimization removes the result from step 2, therefore the project needed to be processed for step 2 again. The adding of MTPs and re-optimization aligned the walls of the building correctly, as can be seen in Figure 4.5. A view of the ray-cloud with camera positions can be seen in Figure 4.6.

After step 1 and 2 were processed, orthophotos of the facades were made. This was done by first defining surfaces in the point cloud corresponding to the four walls of the building. Orthoplanes were inserted and aligned with the corresponding surfaces, as can be seen in 4.7. The orthoplane was adjusted to fit each wall of interest. The resolution of each orthophoto was set to $0.40\frac{cm}{pix}$. After specifying a name for each orthoplane, the orthophotos were generated. Results can be seen in Figures 4.8, 4.9 and 4.10.

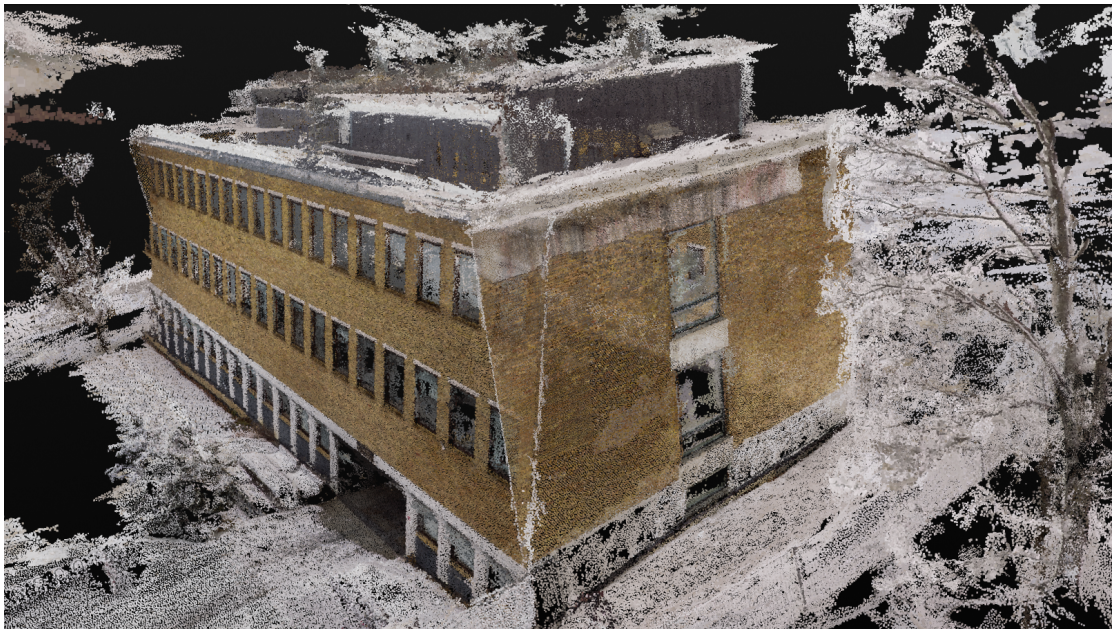

Figure 4.4: Results after processing step 1 and 2. Manual Tie Points would be added to correct for the misalignment of the walls.

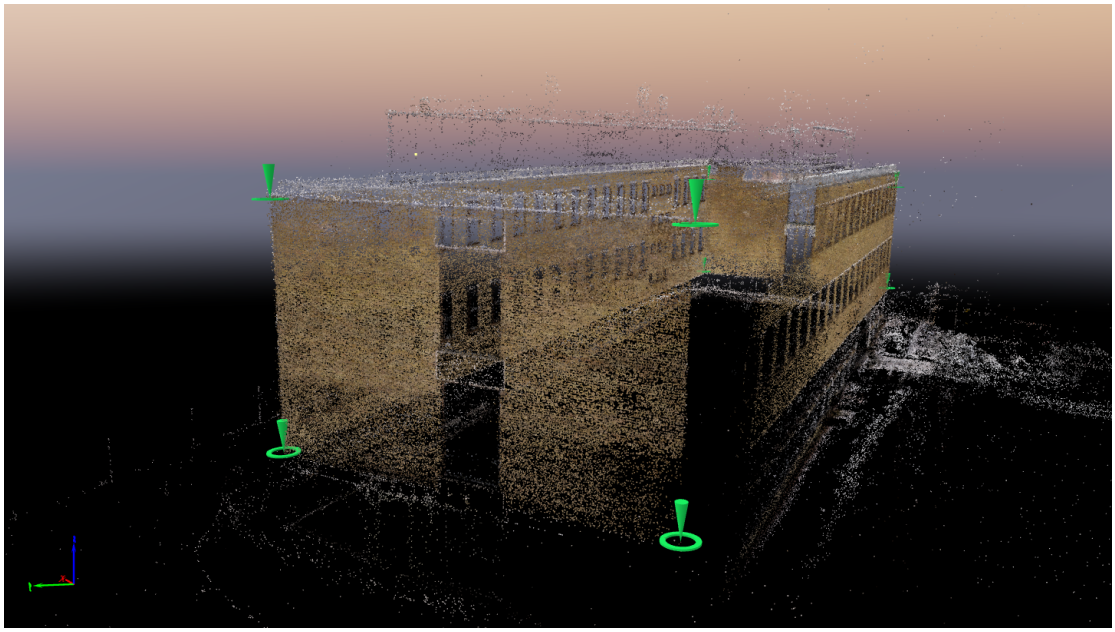

Figure 4.5: MTPs, in green, were added to each corner of the building to improve the point cloud. The alignment of the walls are corrected after re-optimization and step 2 is reprocessed.

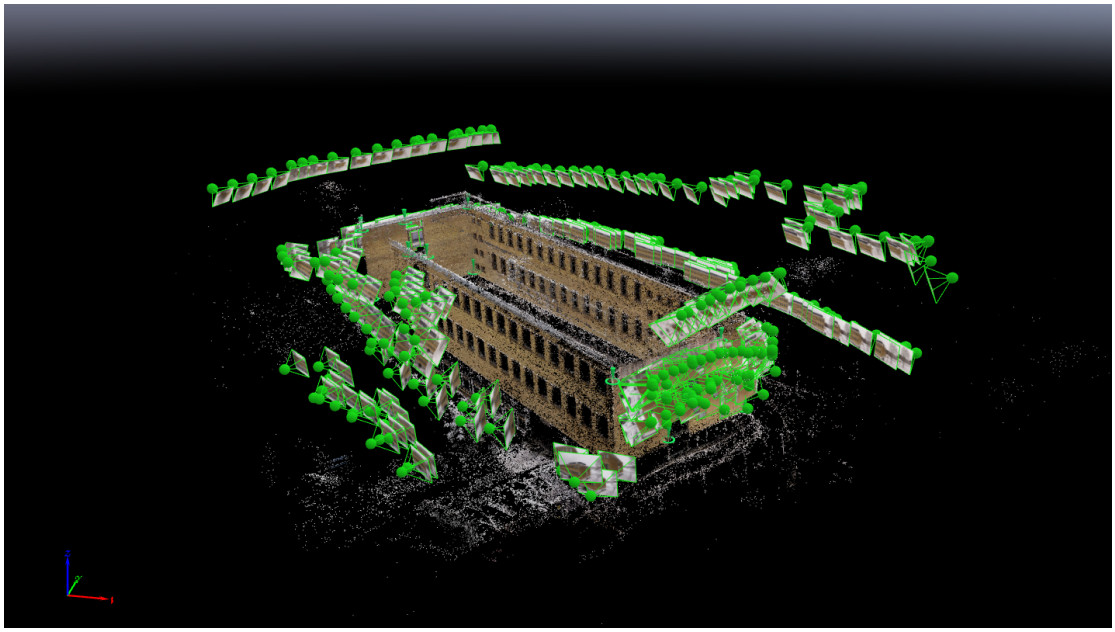

Figure 4.6: Ray-cloud with camera positions.

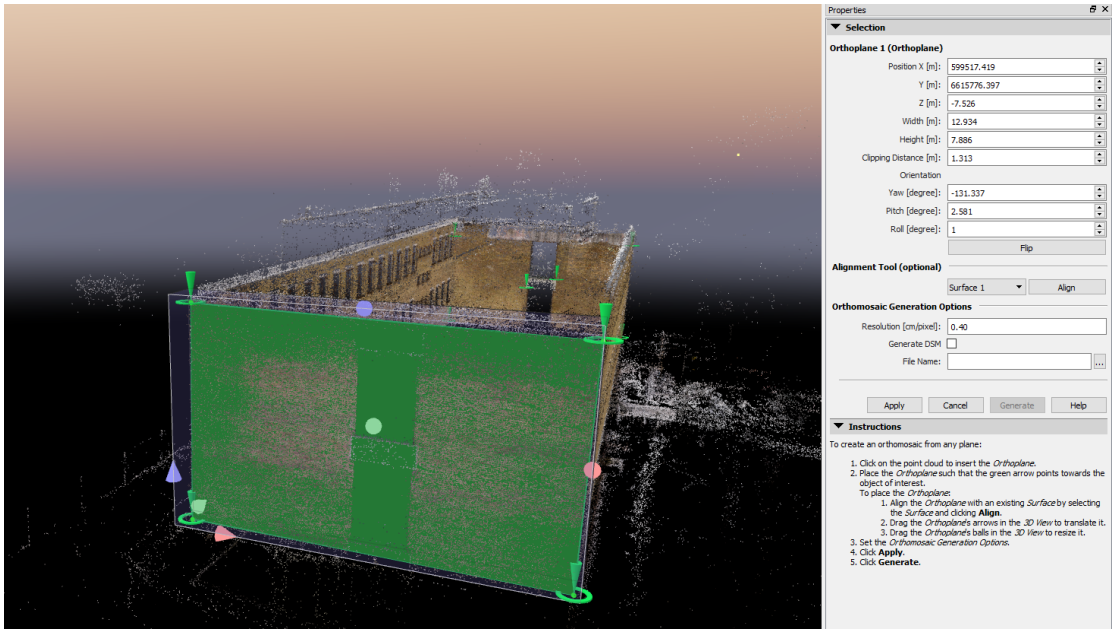

Figure 4.7: The green surface is defined and used in order to align the orthoplane with the wall. Size of the orthoplane is adjusted to cover the whole wall.

Figure 4.8: Orthophoto of KA-building, North wall.

Figure 4.9: Orthophoto of KA-building, South wall.

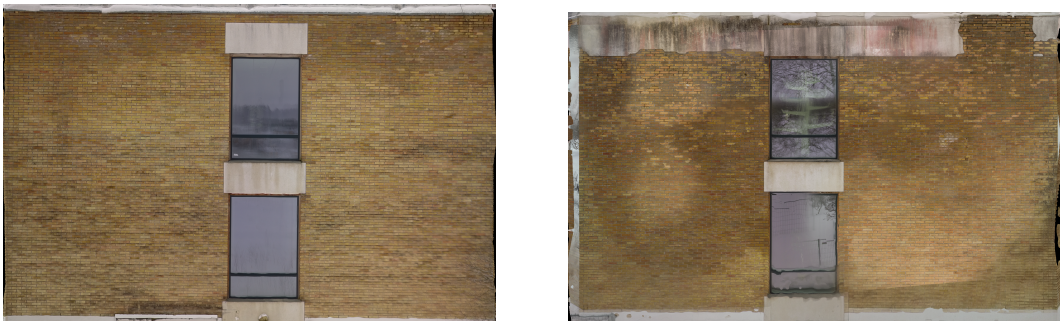

Figure 4.10: To the left: Orthophoto, West wall. To the right: Orthophoto, East wall

4.3 Analysing orthophotos in Python

The orthophotos were the basis of all further analysis. The remaining processing steps were primarily done in Python with the use of scikit-image library. Through several steps, an attempt to segment each individual brick from each other was made. After segmentation an analysis of the segments was done to infer what condition the bricks were in. Before loading the orthophotos into the script, they were cropped in Adobe Camera Raw. The cropping was made to exclude areas that are not part of the wall. In addition, windows on the facade were removed. To illustrate the different steps, a smaller area of the east wall will be used for demonstrating the procedure in Python.

To enable a brick by brick analysis of the bricks seen in the orthophotos, a method was developed in order to segment the bricks from each other. The method aimed to produce binary regions that maps onto each individual brick in the orthophoto. In this way it would be possible to label each region, essentially giving each region a unique identification number. And for each ID there would be information about where it is located in the image, represented by a bounding box with image coordinates. Ideally the semantics of the pixel values in the binary image would be:

$$Binary\,image \begin{cases} 0 & = & \text{mortar} \\ 1 & = & \text{brick} \end{cases} \tag{4}$$

Another important aspect of this binary image is that each region should only map on top of one brick in the orthophoto. The concept of regions was described in

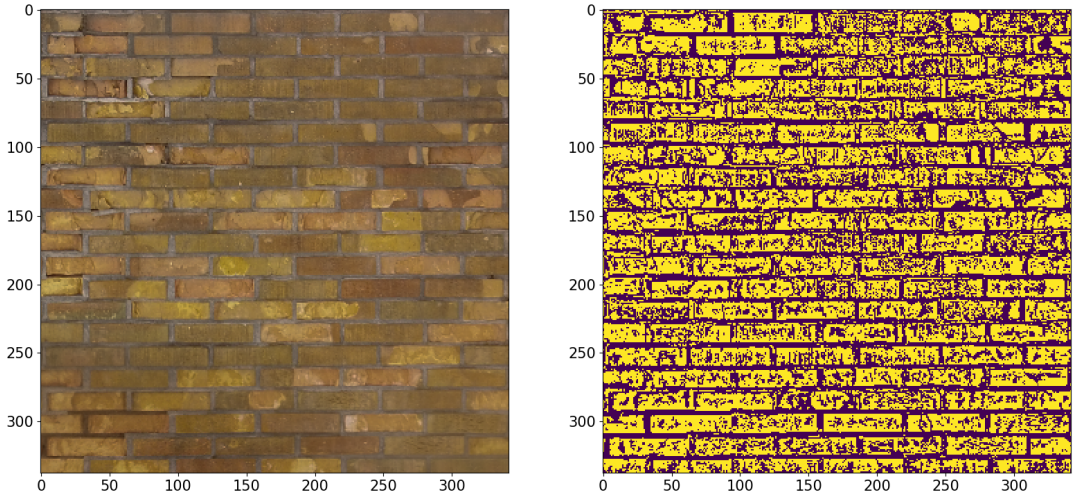

Figure 4.11: Initial binary mask from thresholding with the adaptive threshold method from scikit-image.

the theory section 3.2.1. The term region will be used throughout the sections in the thesis. Ideally the position and extent of one individual region is the position and extent of a brick in the orthophoto.

The first step in creating the binary regions was to threshold the orthophotos with the *threshold adaptive* method from scikit-image. This produced a binary image which formed the foundation of the binary regions intended to map on top of all the bricks in the orthophoto. The results of the first step can be seen in Figure 4.11.

The next steps concerned the manipulation of the binary regions, in order to improve the mapping. The methods *remove small holes* and *remove small objects* from scikit-image was used. The former method would fill holes in the binary regions, the latter would remove small regions. The results of this step is seen in Figure 4.12.

A definitive characteristic in the structure of some brick wall buildings is the horizontal lines of mortar between the layers of bricks. This should ideally also be evident in the binary image. A method to "detect" these horizontal lines was defined in the script, named "horizontal lines detection split" and can be located on line 37 in the script, (see appendix). The input of the function is the binary image processed so far. It considers the entire span across the x-axis for each row on the y-axis. The thought is that if the mean value of this line is lower then a given threshold, the entire line should be 0, essentially being classified as a continuous horizontal line of mortar. If the mean value is higher then the given threshold value, it will not manipulate any values across that row. The result of this step can be seen in Figure 4.13.

Figure 4.12: To the left: initial binary image. To the right: filled holes and small objects removed.

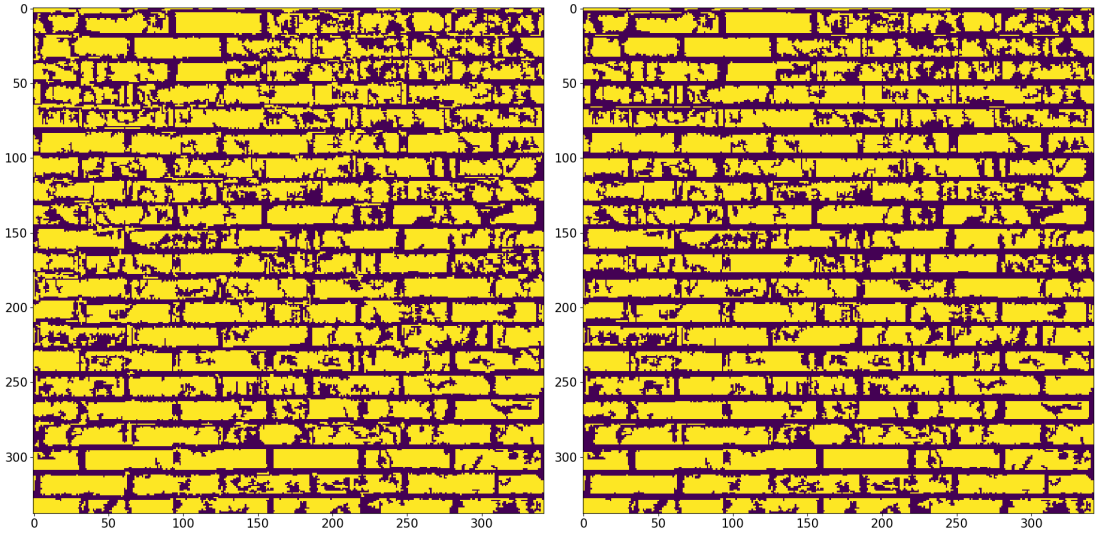

Figure 4.13: To the left: before horizontal lines are detected. To the right: After horizontal lines detection split has been used.

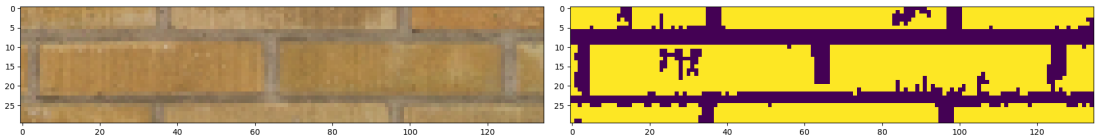

Figure 4.14: Two connected regions can be seen in the right image. Ideally these regions should not be connected.

The effect of the horizontal lines detection split method would ideally split the regions from each other in the vertical direction. The next issue is the connection of regions across different bricks in the horizontal direction. Two connected regions can be seen in Figure 4.14. To gather more information about each region in the binary image, *label regions* method in scikit-image was used. This method gave each region in the binary image an unique identification by changing the value of all connected pixels in that region, as can be seen in Figure 4.15. Next, the *regionprops* method is used to gather information about the length of the regions in x and y directions.
This information was used in combination with knowledge about the actual length of the bricks and the resolution of the orthophoto. The resolution of the orthophoto was set to $0.40\frac{px}{cm}$. And the length of a brick was measured to be $228mm$ long and $54mm$ high. This means that, on average, one region in the binary image should be 57 pixels long and 13.5 pixels high.

$$\begin{aligned} Width &= 22.8\,cm \times \frac{1\,px}{0.4\,cm} = 57\,px \\ Height &= 5.4\,cm \times \frac{1\,px}{0.4\,cm} = 13.5\,px \end{aligned} \tag{5}$$

This knowledge was used in the method named "grow and split" which can be seen in the script on line 64 (see appendix). This method is used in an iterative way. It measures y and x lengths and based on this will either expand the region or acknowledge it as a region that conforms to the size of a normal brick. The process can be seen in Figure 4.16. If the region has a length that equals for example 2, 3, 4 or 5 consecutive bricks, it will split the region accordingly and save the result. Regions that fall outside this lengths are expanded and merged if they overlap, these regions might be acknowledged in subsequent iterations.

4.3.1 Entropy analysis

The binary regions established in the previous steps are the basis for gathering the coordinates and minimum bounding box. The original orthophoto was now analysed based on the region coordinates of the minimum bounding boxes of the regions. The loop that analyses each region starts on line 288 in the script (see appendix). The regions were classified into four types: green, yellow, orange and red. Which is meant to be a descriptive range from best (green) to worst (red).

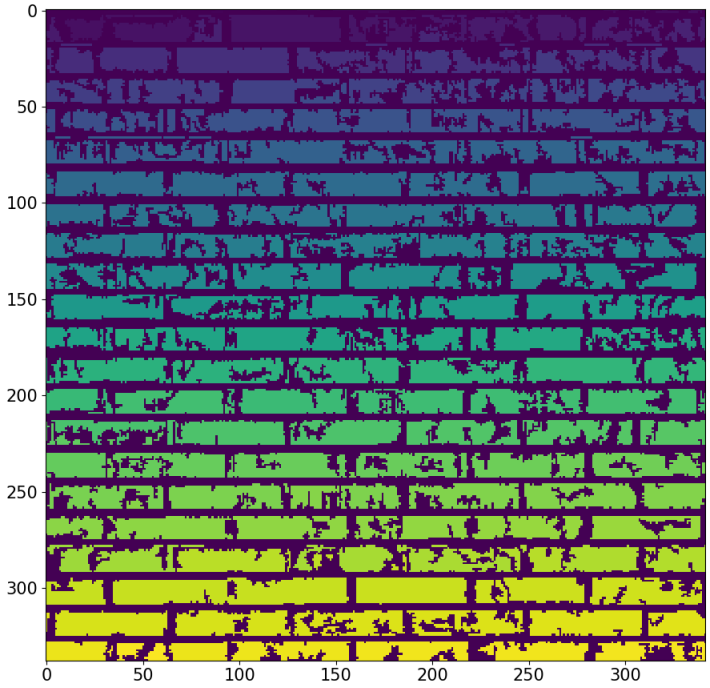

Figure 4.15: The regions in the binary image with a unique label for each region. The difference in color represents different pixel values. This means that the binary image is now turned into a label image. Where each individual region is defined by sharing the same pixel value.

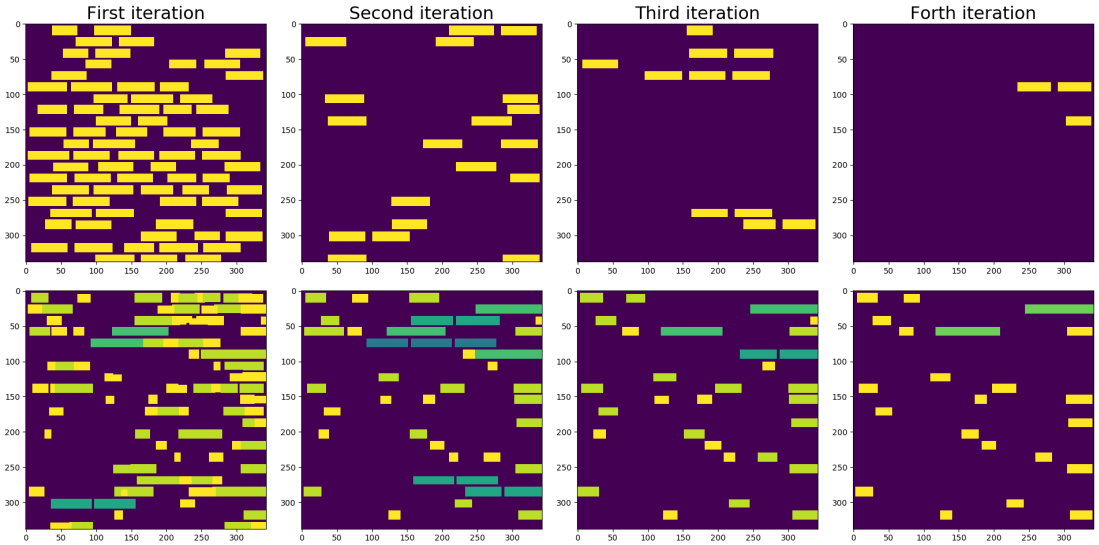

Figure 4.16: This figure illustrates the iterative function of the "grow and split" method defined in the script. On the first iteration, all regions that coheres to the expected size is saved. The other regions are grown and merged. In the end most regions are acknowledged having the "correct" size.

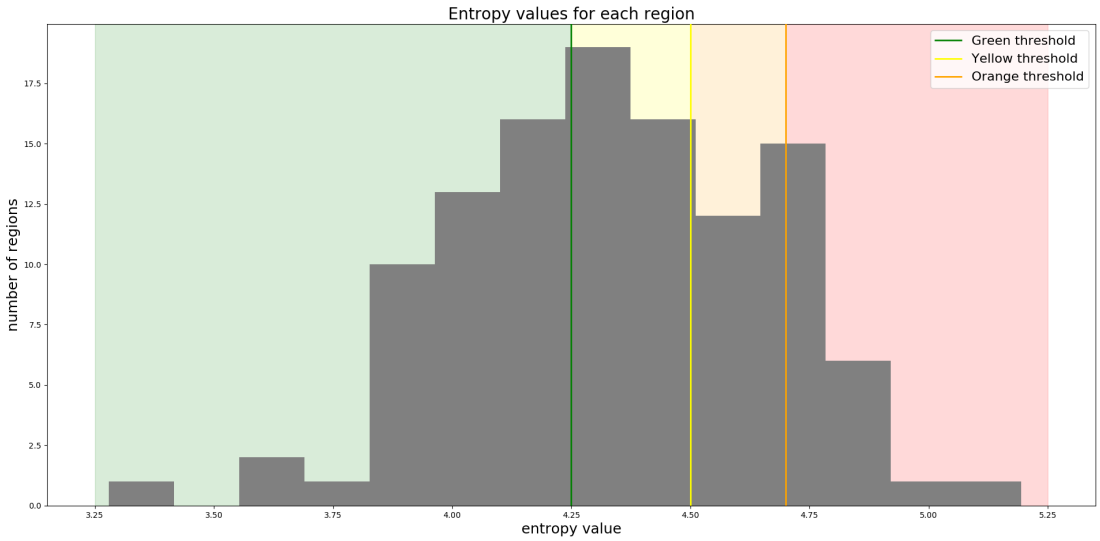

Figure 4.17: A plot that show the distribution of entropy values of all regions identified. The thresholds for each category is also shown. These limits were adjusted in the final implementation and can be seen in the results section.

The variable introduced to classify the regions is the *entropy* method from Scikit-image, this method is mentioned in section 3.2.5.
The limits for each class and the distribution of entropy values can be seen in Figure 4.17. This figure will show how the bricks identified are classified. The analysis should now be on a brick by brick level. If a brick gets an entropy value above the green threshold value 4.25 and below the yellow threshold value 4.50, it will be categorized as yellow. The result for all the bricks identified in this example is visualised in Figure 4.18.

This concludes the methodology section.

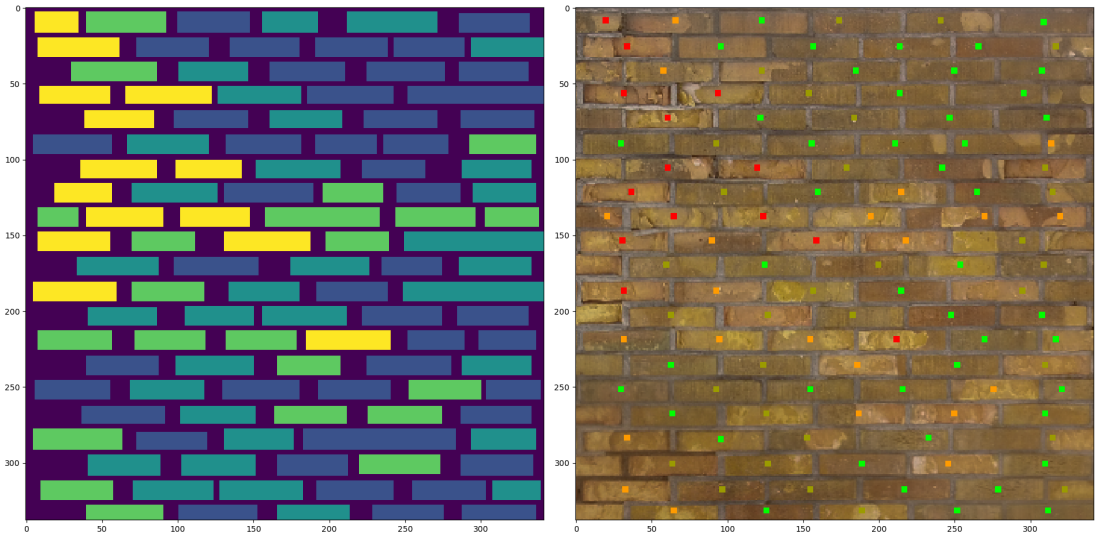

Figure 4.18: To the left you can see the regions corresponding to the mask of the orthophoto. To the right the orthophoto is shown together with the centroid coloured according to the entropy analysis.

5 Results

Four orthophotos were produced and used as input for the script, analysing each wall. Here the results are presented. The bricks were categorized into 4 different categories: green, yellow, orange and red. This should be understood such that green regions are in a better condition than the yellow regions and so forth. The number of bricks in each category is presented in Tables 2, 3, 4 and 5.

Table 2: Condition of East wall

Condition	Percentage	Bricks
Green	77.4	3168
Yellow	19.8	810
Orange	2.4	98
Red	0.5	19
Total	100	4095

Table 3: Condition of West wall

Condition	Percentage	Bricks
Green	74.8	3294
Yellow	24.0	1058
Orange	1.1	48
Red	0.1	2
Total	100	4402

Table 4: Condition of South wall

Condition	Percentage	Bricks
Green	82.2	5671
Yellow	15.5	1068
Orange	2.0	136
Red	0.4	26
Total	100	6901

Table 5: Condition of North wall

Condition	Percentage	Bricks
Green	60.4	6027
Yellow	34.3	3424
Orange	4.8	481
Red	0.5	53
Total	100	9985

5.1 Distribution of entropy values

The distribution of entropy values for each wall is displayed in Figures 5.1, 5.2, 5.3 and 5.4. In these plots we can also see where the limits for each category is set. The higher entropy values are consecutively categorized as green, yellow, orange and finally red for the highest entropy values. These limits are set in lines 302, 309, 316 and 323 of the script (see appendix). The limits were set as is to classify the bricks into one of the four categories: green, yellow, orange or red. Changing these limits changes the percentages of regions in each category.

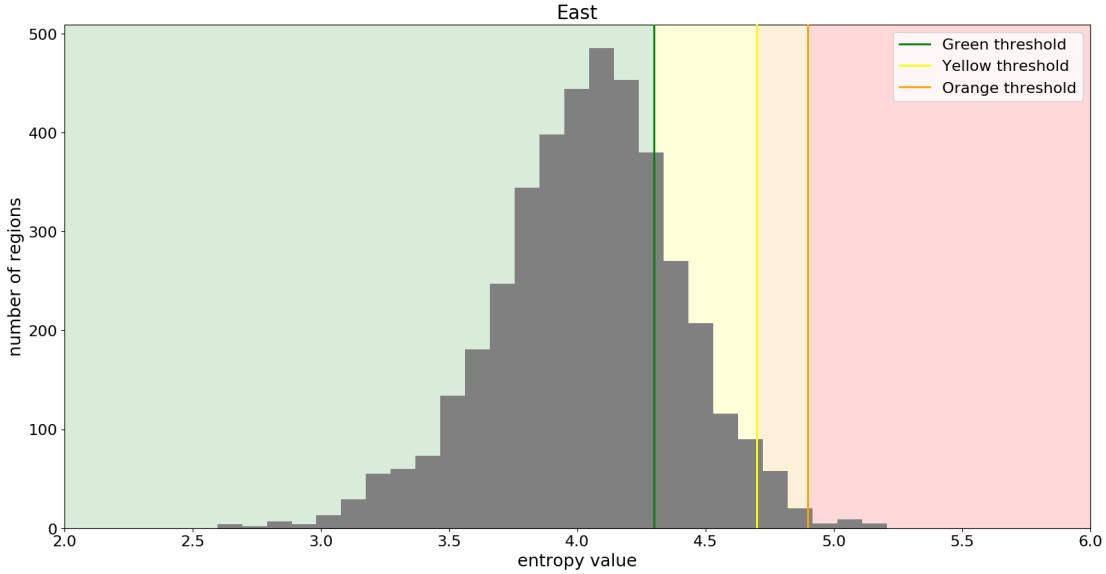

Figure 5.1: The distribution of entropy values for the East wall.

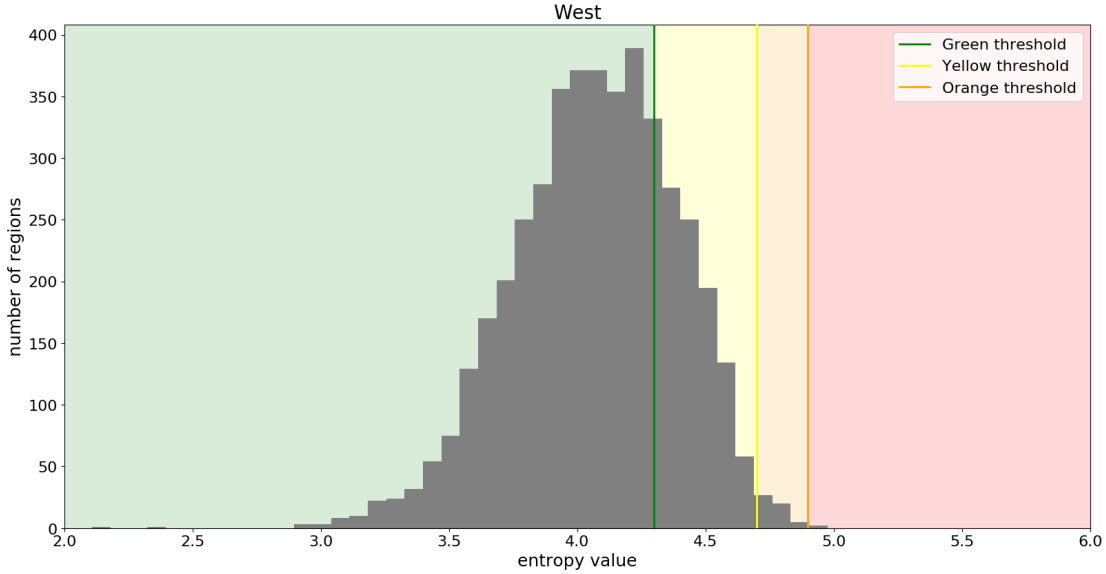

Figure 5.2: Distribution of entropy values, West wall.

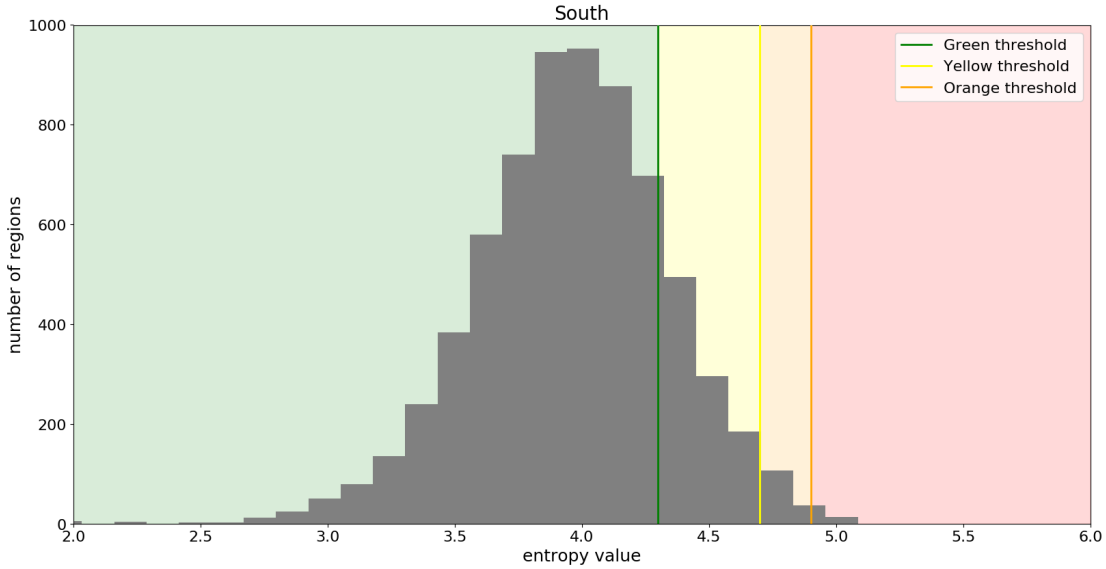

Figure 5.3: Distribution of entropy values, South wall.

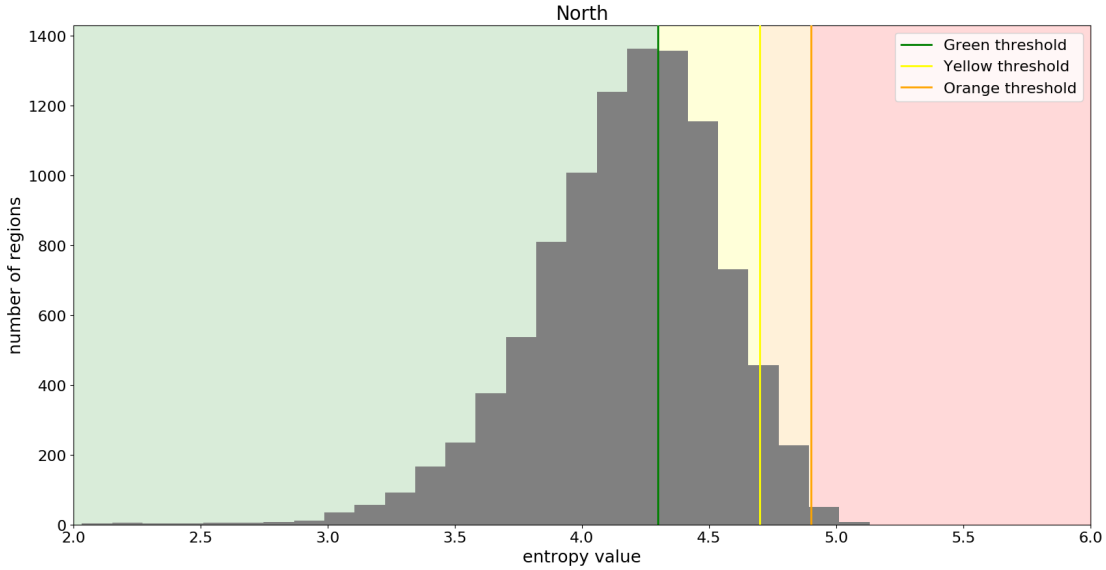

Figure 5.4: Distribution of entropy values, North wall.

5.2 Location of damaged bricks

Figures 5.5, 5.6 and 5.7 shows where the location of the damaged bricks are according to the model. To visualise this, the identified regions are shown in the colors: green, yellow, orange and red. Red regions are the regions with the highest entropy value.

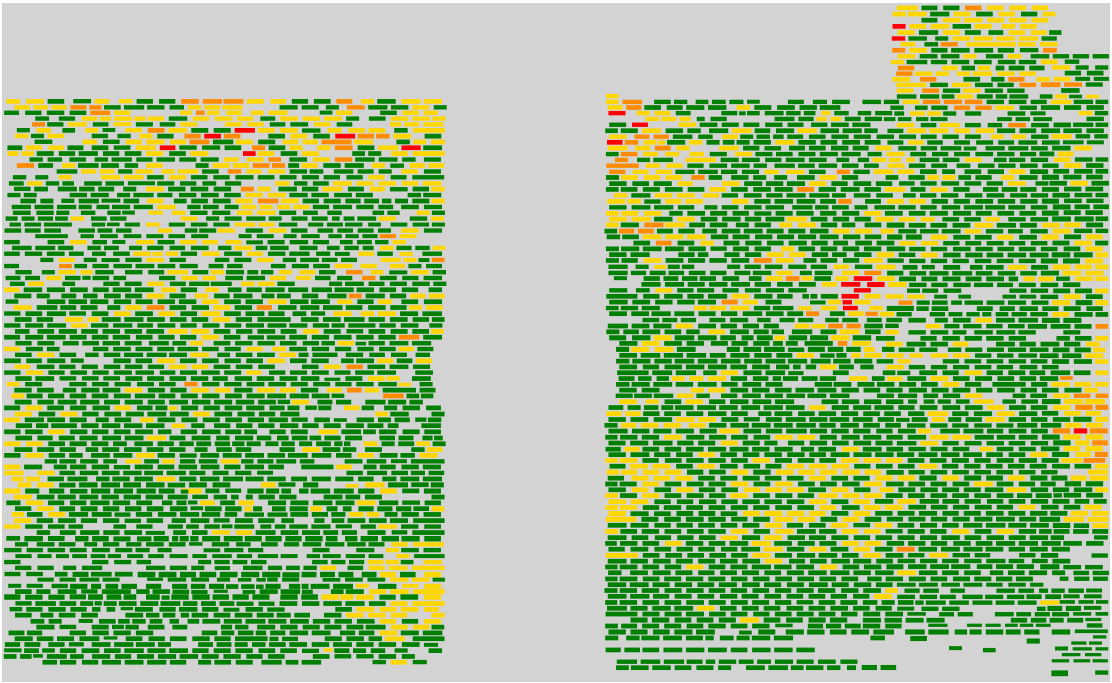

Figure 5.5: East wall. Bricks are coloured according to their entropy value.

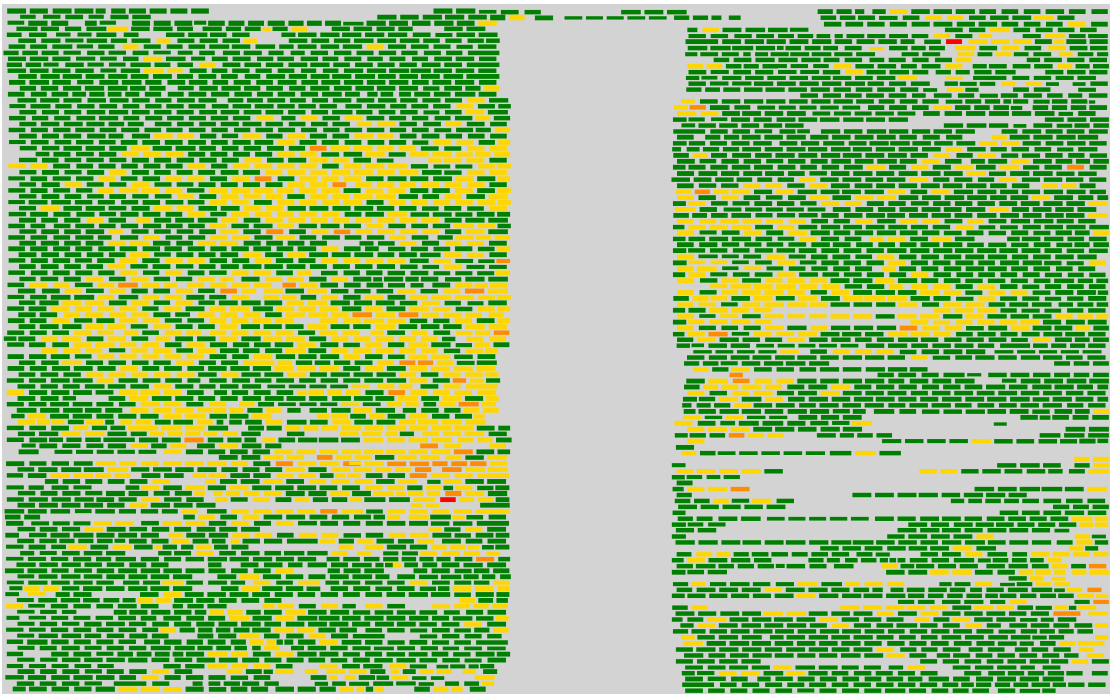

Figure 5.6: West wall. Bricks are coloured according to their entropy value.

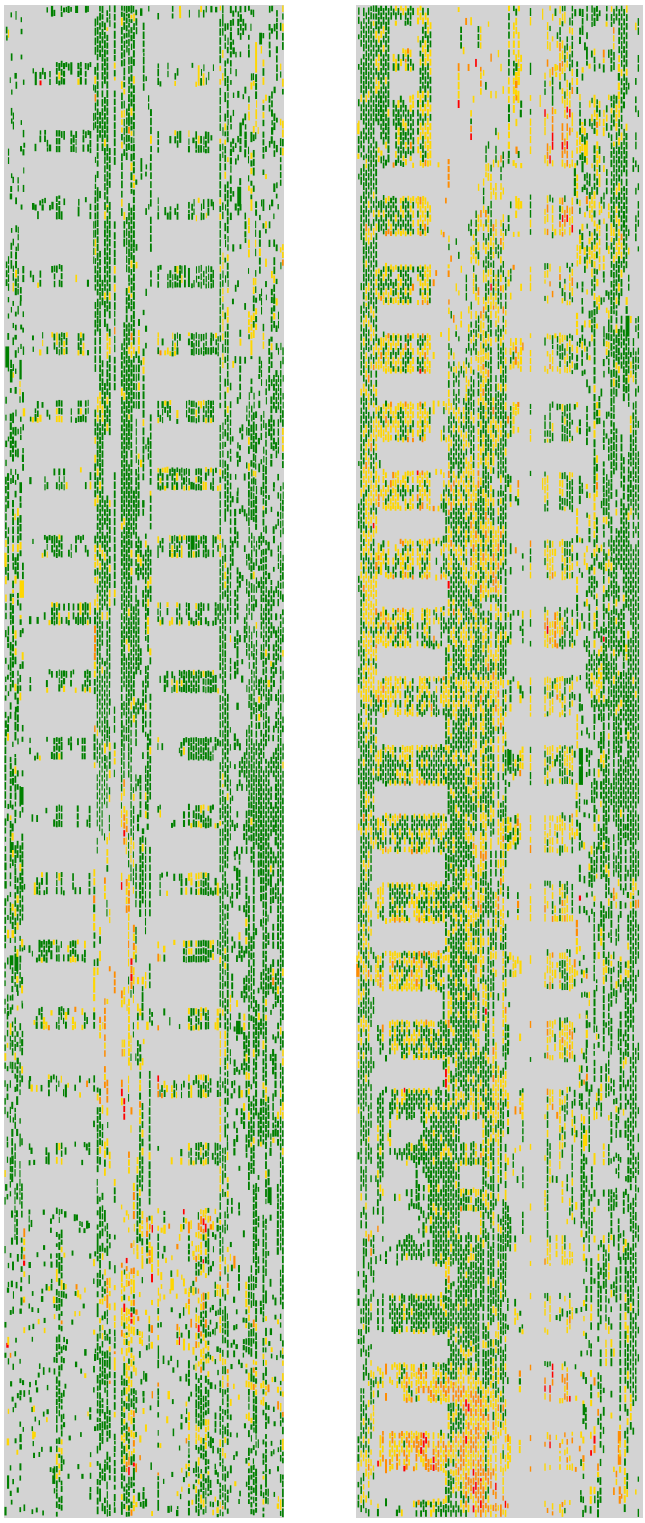

Figure 5.7: On the left side we see the South wall. On the right side we see the North wall.

6 Discussion

6.1 Model analysis

6.1.1 Coverage

In order to compare how many bricks were analysed by the model and how many bricks are visible in the orthophoto, an attempt to estimate the total number of bricks visible was made. The bricks has a size of approximately $228\,mm \times 54\,mm$. Including the mortar this produces about 63 bricks per square meter[23]. The total size of the wall is estimated from the orthophotos, where windows were subtracted from the total area. The total area of the two shorter wall (East and West), are approximately $\approx 90m^2$. And for the two longer walls (North and South) $\approx 300m^2$. The remaining area covered by bricks is shown in Table 6. This table compares the estimation of bricks to the regions discovered in the model.

Table 6: An estimation of how well the coverage of the model was

Wall	Area	Bricks estimate	Regions	Coverage
East	68.2 m^2	4297	4095	95.3 %
West	76.0 m^2	4788	4402	91.9 %
North	236.5 m^2	14897	9985	67.0 %
South	232.6 m^2	14656	6901	47.1 %
Total	613.3 m^2	25383	38638	65.7 %

For the South wall the coverage is estimated to be 47.1%. One major contributor to this low number is the tree that covers the left side of the wall. As often is the case for any remote sensing methods, obstacles in front of the object of interest is detrimental to the results.

In general the two longer walls has a lower coverage rate. There are no obstacles in front of the North wall, which raises the coverage percentage significantly compared to the South wall. Although 67% is still rather low. One element which contributes to this low rate is the lack of sharpness in the orthophoto. For example, the sharpness of the orthophoto is relatively better in the upper part of the orthophoto compared to the lower part. The effect of this variation in sharpness on the success of the segmentation can be seen, in that the upper part is better segmented then the lower part.

Another challenge with the longer walls is the interplay between the length of the horizontal mortar lines and how the script is trying to calculate a mean value along this line in order to detect it. The wall is 40 meter long and in effect the straightness of the horizontal mortar lines are not necessarily consistent across the entire wall, neither in the orthophoto. The effect of this can be seen in Figure 6.1. The figure is a selection of the north wall, lower right area. One can see that

a slight rotation in the horizontal line with regards to the mortar cuts into the region detected in the first place.

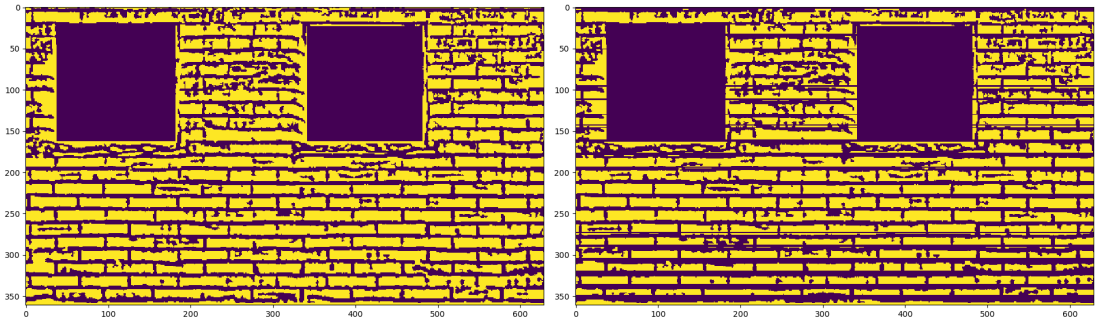

Figure 6.1: A selection of the north, lower right side of the wall. The horizontal lines detection cuts into the regions identified. This comes from the fact that the method consider the whole width of the wall in one go. In effect cutting into regions if they do not fall in line with the entire length of the wall. To the left is the binary image before horizontal lines detection is implemented. The right image shows the effect.

The considerably higher coverage rate on the East and West wall is probably connected to the sharper orthophoto produced and in addition the horizontal lines detection algorithm (see script, line 37) works even better, because the wall is now only around 12 meters long. For the two shorter walls, the definition of the splitting has changed from y to x direction. This means the detection of mortar is done for two sections of the wall independently. In effect this reduces the "demand" of the horizontal correctness of the mortar to be valid across a shorter span. The horizontal lines detection could be improved by adding "splitting" options in x and y directions at the same time. This would most likely improve the success of the segmentation.

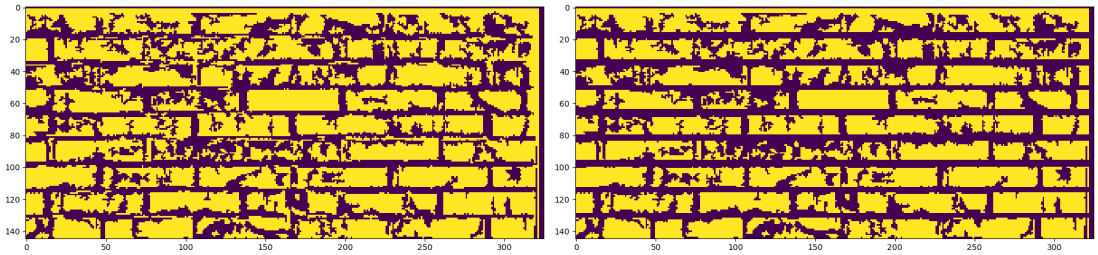

Figure 6.2: Effect of the horizontal lines detection split is more precise when the lines of mortar are shorter. This is selection of the West wall, upper left area.

6.1.2 Classification of bricks

The idea behind classifying the bricks is that bricks in good condition should be homogeneous and "smooth" across its surface. A structural damaged brick will

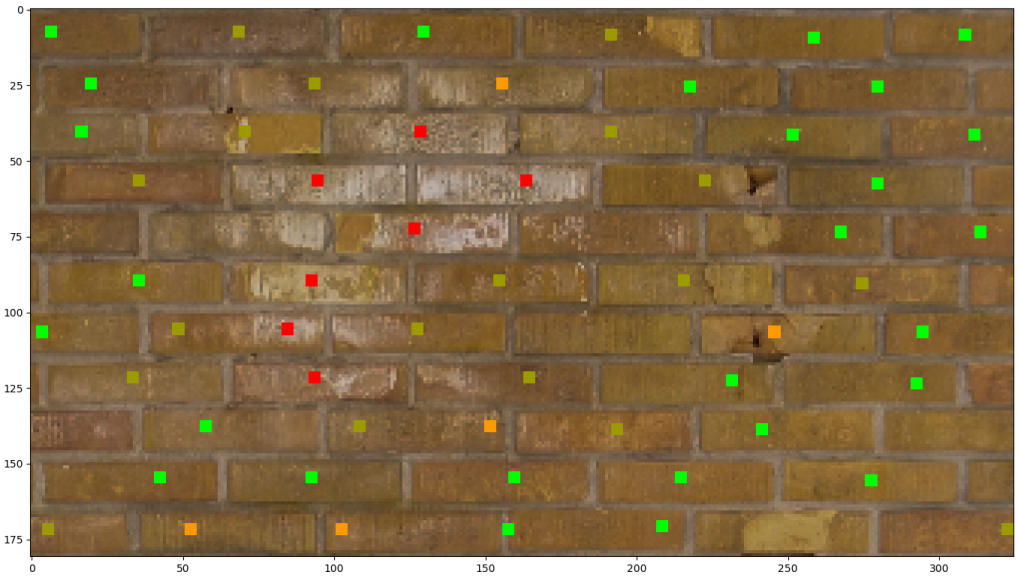

Figure 6.3: Here we see the prominence of white salt on the bricks. This texture causes the entropy value to be high and it is therefore categorized as red. Even though the structure of the bricks are good in this area. This area is located on the right side of the East wall.

have a rough surface and create shadow regions in the orthophoto. In an attempt to distinguish the bricks from each other the entropy value for each region was used.

In general the model will recognise and distinguish a good brick from a damaged brick, given a precise segmentation. But sometimes a brick in relatively good condition will be classified as category red. There are two characteristics that makes this happen, when salt or algae are visible on the bricks surfaces. This can be seen in figure 6.3 and figure 6.4 respectively.

Another weakness in the classification comes from the quality of the orthophoto. The entropy value regards how much information there is in an image. In general a blurry image will have less information then a sharp image. Therefore, if a brick is identified in a blurry part of the orthophoto it is more likely categorized as green. Vice versa if a brick is identified in a section where the orthophoto is sharp, the brick will contain more information and is more likely to be categorized as red. A blurry part of the orthophoto and its effect on the classification is illustrated in figure 6.5.

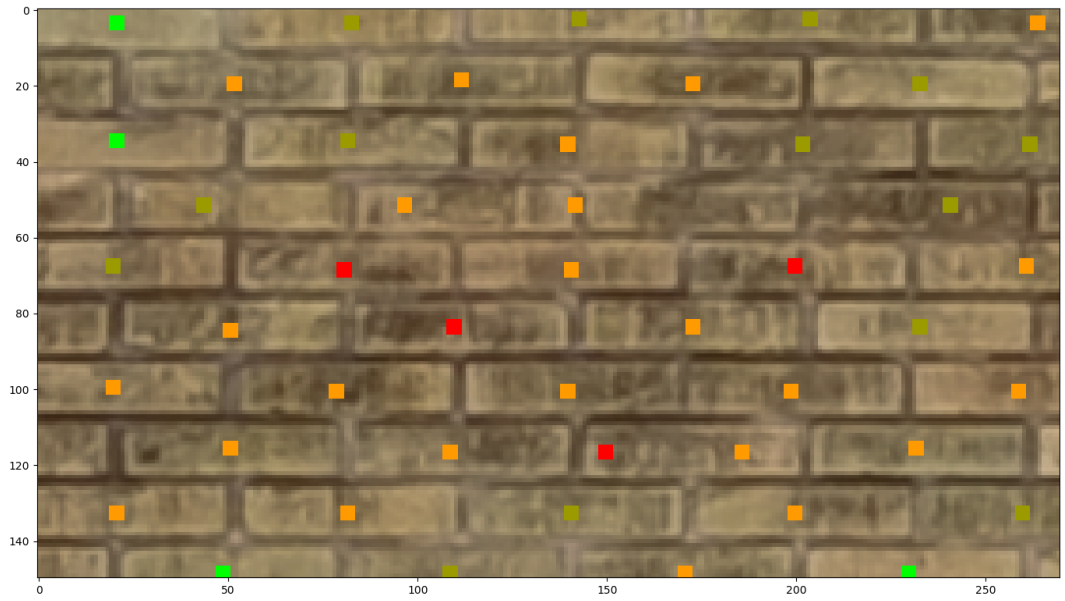

Figure 6.4: The black stains on these bricks is most likely algae. This texture will give a high entropy value and are therefore categorized as yellow, orange or red. This image is from the leftmost part of the North wall.

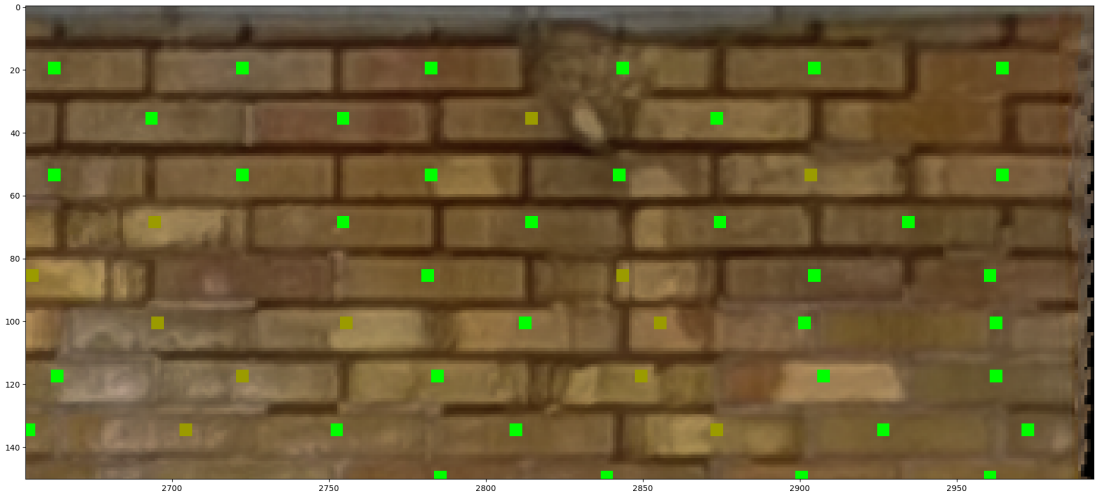

Figure 6.5: This is the West wall, upper right side. Most of the bricks visible in the top region are structurally damaged and should be categorized as red. But because the orthophoto is blurry, hence less information is available, the entropy value is reduced. Ideally, some of these bricks should be in the red category.

6.2 Further improvements

In this section other attempts and thoughts about improvements are mentioned. The two first sections (6.2.1 and 6.2.2) concerns improvements and other attempts made at segmenting and classifying the bricks in the orthophotos. The remaining sections after that concerns possible improvements for better orthophotos.

6.2.1 Segmentation

The adaptive threshold method is most successful when the contrast between brick and mortar is good. There are probably several ways one could take advantage of the contrast between brick and mortar to enable a segmentation of the bricks.

An attempt was made with the k-means clustering method[9][4]. This is an unsupervised classification method that assigns the pixel values into n clusters, where the number of clusters is defined by the user. The implementation of k-means from scikit-image was tested. For a small image with only one brick and some mortar visible, the results were very good as seen in Figure 6.6. The result when considering a larger part of the wall can be seen in 6.7. The results might be improved by adjusting the settings in the algorithm or experimenting by adding another index channel before segmentation, such an index channel is explained in the next paragraph.

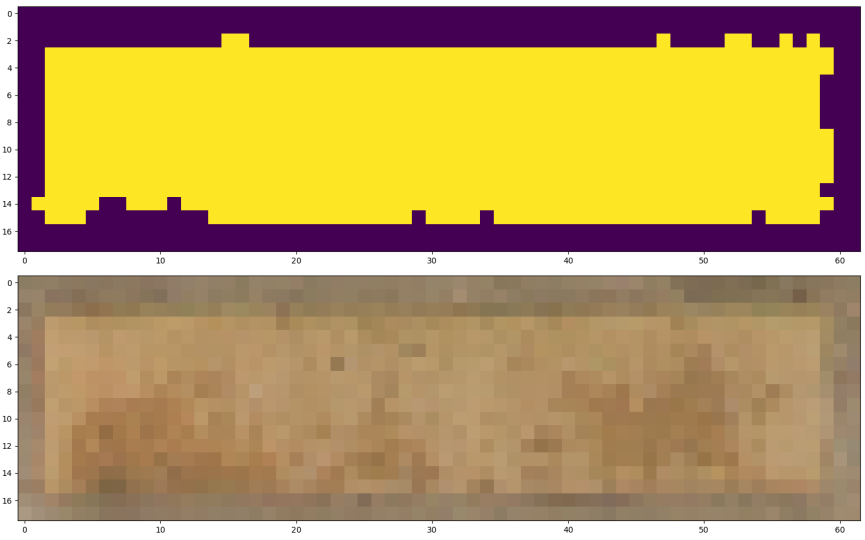

Figure 6.6: Here we see that the k-means algorithm works very well when considering only one brick.

Another attempt at splitting bricks from mortar in the orthophoto, was to make an index based on the color information available in the orthophoto. Inspired by a well known method in remote sensing and vegetation measurements, is the Normalized Difference Vegetation Index[31]. The idea is that different material will reflect different wavelengths of light, and an index of this relation might accentuate

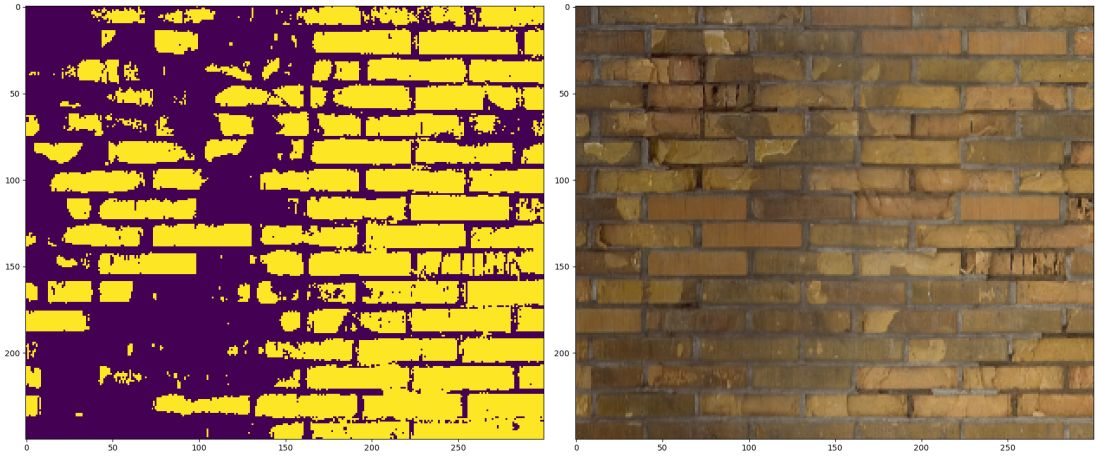

Figure 6.7: The success of the segmentation by the k-means algorithm varies across the orthophoto.

the differences. The camera attached to the Phantom 4 has three channels: red, green and blue. The index calculation that provided the best results are shown in Equation 6, Listing 1 and Figure 6.8.

$$Index = \frac{Green + Blue}{Green - Blue} \tag{6}$$

```
img = imread('01_East_040.tif')
img = img[250:600,0:800,:]
shape = np.shape(img)
channel_index = img.copy()
channel_index = channel_index.mean(axis=2)

for i in range(shape[0]):
    for j in range(shape[1]):
        over = int(img[i,j,1])+int(img[i,j,2])
        under = int(img[i,j,1])-int(img[i,j,2])
        pixval = abs(over/(under+0.1))
        if pixval >=12:
            channel_index[i,j] = 255
        else:
            channel_index[i,j] = pixval

plt.imshow(channel_index, vmax=10)
```

Listing 1: Attempt at channel index using the green and blue channels of the orthophoto. The result of this code can be seen in figure 6.8.

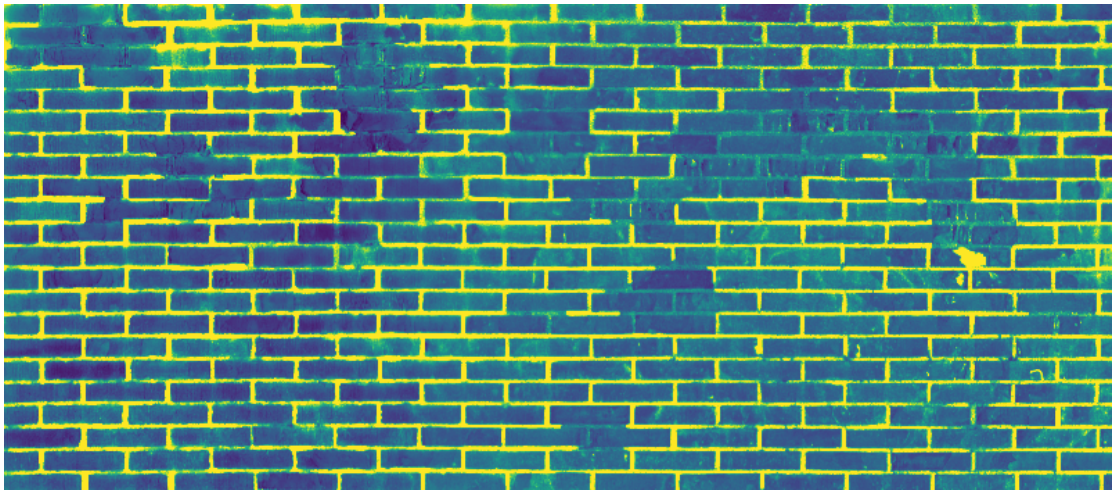

Figure 6.8: This shows the results of indexing the image like in Equation 6. The code for this output image can be seen in Listing 1. The mortar seems to "pop-out" from the bricks. This method might prove successful given a greater control over the radiometric conditions in the project. This is discussed under Radiometric control and consistency, section 6.2.4.

An interesting strategy would be to take advantage of the build pattern in the brick wall, for example the running bond build seen in the KA-building. An attempt to segment the facade of buildings through orthophotos can be found in this article: "Tiling of Ortho-Rectified Facade Images"[14]. A hybrid solution might take into account traditional remote sensing classification techniques and modern pattern recognition techniques used in computer vision.

6.2.2 Classification

The model only considers the entropy value in the classification process. The success of the classification might be improved by considering several factors and not only the entropy value. A popular method used in texture analysis is the grey level co-occurrence matrix (GLCM)[8]. This method compares intensity values in the image and counts how often they occur between each other at given distances and angles. There are several statistics that can be analysed in a grey level co-occurance matrix, such as contrast, dissimilarity and homogeneity. An attempt was made with the homogeneity attribute of the GLCM analysis, but the classification result were much the same as the entropy method.

If the bricks are perfectly segmented, it is possible to store one single image for each brick seen in the orthophotos and continue the brick by brick analysis by examining this dataset. Some of the bricks in this dataset could be extracted as a training set, where damaged and healthy bricks are identified(supervised classification). Then the rest of the bricks could be assigned to each class in a data-driven, machine learning manner. A visualisation of how such a data set would look like for this project is shown in Figure 6.9. The figure shows all the twelve bricks categorized as red for the East wall.

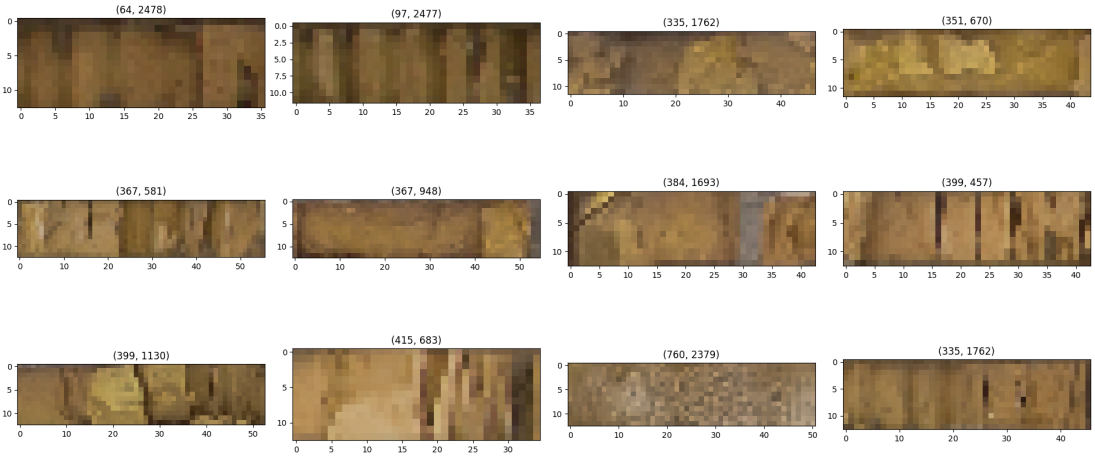

Figure 6.9: A visualisation of 12 bricks, categorized as red, from the orthophoto of the East wall. Notice the image coordinates for each brick is shown in the title. This is just some of the information one could consider in a data driven classification of bricks.

6.2.3 Camera

The camera attached to Phantom 4 has a CMOS with dimensions: $6.17mm \times 4.55mm$. This seems to produce satisfactory results in some areas of the orthophotos in this project. A bigger sensor would contribute to lower the GSD value and depending on the flight, increase image overlap. This makes it easier for the processing in Pix4D to identify keypoints and tie-points, which should contribute to better orthophotos. A bigger camera sensor can also make it easier to fly, as the distance to the buildings facade might be extended while piloting without reducing the level of detail available in the image.

Another characteristic of the Phantom 4 camera is the rolling shutter mechanism[13]. This shutter mechanism is typical for low-grade CMOS cameras. The effects of a rolling shutter mechanism becomes visible in the image if either the camera or object is moving during exposure. It can produce geometrical distortions in the image. A camera with a global shutter mechanism might be preferred in photogrammetry projects.

Compared to a small image sensor, the larger sensor needs less light to make a correct exposure, hence a faster shutter speed can be achieved. This is useful since the drone might be moving during exposure. For images that are intended for photogrammetry, there needs to be a balance between having enough time to make the correct exposure and keeping the open shutter time short enough to freeze the motion of the drone, so that blurry images are not captured. If blurry images are taken, they should be removed during image selection.

6.2.4 Radiometric control and consistency

One should strive for consistent and optimal exposure during the whole flight. Under or overexposed images will lose important information used in the photogrammetry software, for example in recognising tie-points. Changes in exposure from image to image can be detrimental to the quality of the orthophoto. Manual settings can be used to circumvent automatic adjustments of exposure and white balance settings during flight.

The best conditions to acquire images seems to be on cloudy days, at the brightest part of day. This could make consistent exposure and white balance easier to achieve. The light reflected from the building is more likely to be consistent for all the walls, as dense clouds will contribute to spread the light evenly across the scene. Clouds can also reduce the chance of shadows being cast on the building from nearby trees or buildings.
If the ground is covered in snow, number of tie-points identified in the project might suffer due to lack of variations in texture on the snows surface. In addition, correct exposure settings as well as white balance settings might be more challenging if there is much snow in the scene.

Connected to the discussion in section 6.2.1 concerning different segmentation strategies, the success of these might improve the higher the radiometric consistency is in the orthophotos.

6.2.5 Addition of terrestrial sourced images

A reoccurring problem in the orthophotos produced in this project is lack of sharpness and correct geometry of the bricks at the lowest part of the walls. This makes sense because of the lower overlap of images in this part of the walls. Either collecting more images by flying lower or adding terrestrial sourced images could circumvent this problem. An attempt was made at adding terrestrial sourced images, this is described in the next paragraph.

In an attempt to raise the quality of the East wall orthophoto, images taken from the ground were acquired as well, albeit not on the same day as the flight was done. The images were acquired with a Nikon D600 camera, with a 50mm fixed lens. This is a full-format camera with a CMOS of $35.9\,mm \times 24.0\,mm$ and has a resolution of 6016×4016. Two projects were merged, one for the drone flight and one for the terrestrial. They were successfully merged and the orthophoto result can be seen in figure 6.10. The sharpness and geometric quality increased to some degree, but still there was a problem in the lower part of the orthophoto, and the white balance was not consistent across the entire orthophoto. This might be in part of the different light conditions between the two projects, and because of poor image management regarding the pre-processing step where white balance and exposure settings were adjusted. Although this attempt was not as successful, it is recommended to combine aerial and terrestrial images in a project such as this. The results might improve if images are captured on the same day under similar weather conditions. It might be beneficial to take images with the same camera as well, although Pix4D can handle multiple cameras in one project.

6.2.6 Flight plan

After considering the camera itself, the way in which images are gathered are most important for the final results. The way one would choose to fly depends on the object which is to be reconstructed. One should take into consideration the required GSD one wants for the project and how much overlap is needed. If the overlap is above 85% in flight direction and 70% in side overlap, good results should be within reach in most cases.

Another important aspect regards to the movement between images. One should avoid capturing images at the same position, only to vary the orientation of the camera (ϕ, γ, κ). If a series of images are captured at the same position during flight, for example, if the drone is hovering and still recording images, these should then be removed during the image selection phase.

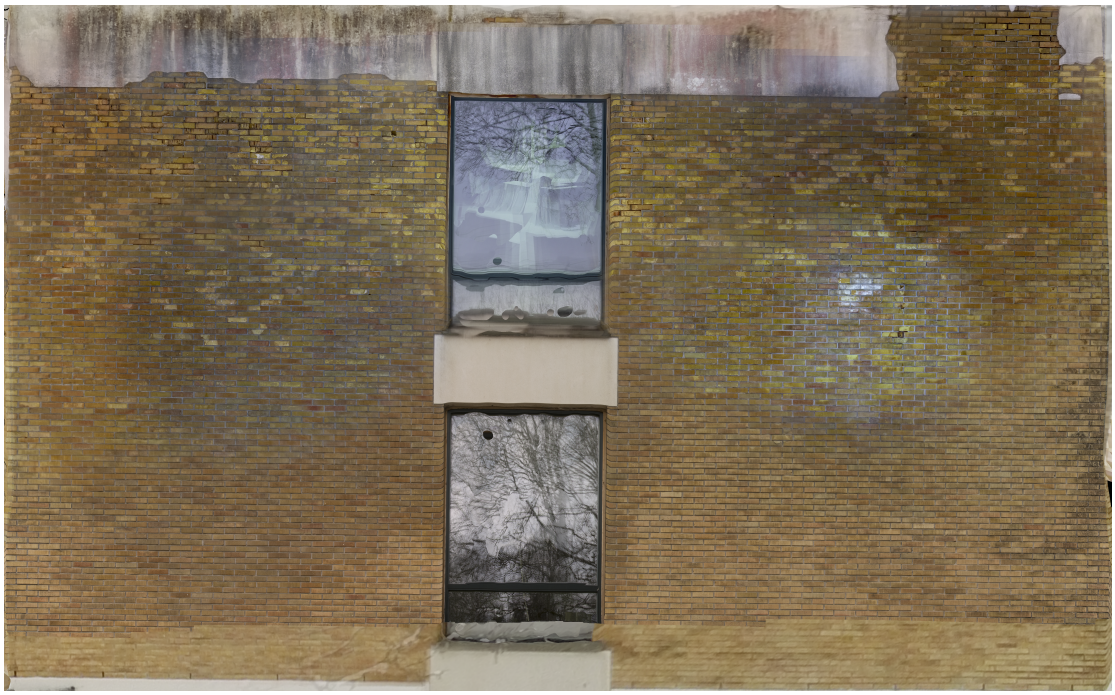

Figure 6.10: Orthophoto from a merged project with images from the camera attached to the Phantom 4 and also a Nikon D600 camera taken from the ground. The variation in white balance and light conditions are evident. One can see the difference in white balance of the top and bottom window. This effect comes from different light conditions during exposure, but also an unfortunate slip of poor image management, as photos with wrong white balance settings were added. Still, geometric distortions can be seen in the lower part of the orthophoto. Excluding that lowest part, the sharpness and detail was better after adding the terrestrial sourced images.

7 Conclusion

This thesis has explored how one could assess the condition of a brick building by producing orthophotos of the walls and analyse them with image processing techniques. The method developed shows that it is possible to make a segmentation on a brick by brick level and classify them according to their condition. Although the completeness in segmentation of bricks and precision in classification varies. Implementation of suggested improvements could possibly refine the completeness of segmentation and precision in classification of the bricks. Exploring the same method on other brick buildings with different bricks and facade structures might also reveal other challenges that should be considered.

This project might also be seen as a small example of how the use of light-weight RPAS and merging of multiple technologies, from photogrammetry to image analysis techniques, can be used together in order to develop new insights. Light-weight RPAS photogrammetry might help to close the gap between traditional maps and building information modelling.

References

[1] Adobe. *Camera raw*. 2018. URL: https://helpx.adobe.com/camera-raw/using/supported-cameras.html.

[2] Øystein Andersen. *Orientering i Stereoinstrument*. 2003.

[3] Wilhelm Burger and Mark J Burge. *Digital image processing: an algorithmic introduction using Java*. Springer, 2016.

[4] Nameirakpam Dhanachandra, Khumanthem Manglem, and Yambem Jina Chanu. "Image segmentation using K-means clustering algorithm and subtractive clustering algorithm". In: *Procedia Computer Science* 54.2015 (2015), pp. 764–771.

[5] Øystein Bjarne Dick. *Geomatikk Kartfaglig Bildebruk*. 2003.

[6] Christophe Fiorio and Jens Gustedt. "Two linear time union-find strategies for image processing". In: *Theoretical Computer Science* 154.2 (1996), pp. 165–181.

[7] Sandvik Kristian Førde. "A toolkit for classification of state of buildings". MA thesis. NMBU, 2013.

[8] Robert M Haralick, Karthikeyan Shanmugam, et al. "Textural features for image classification". In: *IEEE Transactions on systems, man, and cybernetics* 6 (1973), pp. 610–621.

[9] John A Hartigan and Manchek A Wong. "Algorithm AS 136: A k-means clustering algorithm". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.

[10] O Küng et al. "The accuracy of automatic photogrammetric techniques on ultra-light UAV imagery". In: ().

[11] Kim Robert Lisø et al. "A frost decay exposure index for porous, mineral building materials". In: *Building and Environment* 42.10 (2007), pp. 3547–3555.

[12] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[13] Marci Meingast, Christopher Geyer, and Shankar Sastry. "Geometric models of rolling-shutter cameras". In: *arXiv preprint cs/0503076* (2005).

[14] Przemyslaw Musialski et al. "Tiling of ortho-rectified facade images". In: *Proceedings of the 26th Spring Conference on Computer Graphics*. ACM. 2010, pp. 117–126.

[15] Jae-Hyeung Park et al. "View image generation in perspective and orthographic projection geometry based on integral imaging". In: *Optics Express* 16.12 (2008), pp. 8800–8813.

[16] Pix4D. *How to create the Orthomosaic of a Facade*. 2018. URL: https://support.pix4d.com/hc/en-us/articles/202559889-How-to-create-the-Orthomosaic-of-a-Facade.

[17] Pix4D. *How to create the Orthomosaic of a Facade*. 2018. URL: https://support.pix4d.com/hc/en-us/articles/204664359-How-to-draw-a-new-orthoplane.

[18] Pix4D. *How to select / draw the processing area*. 2018. URL: https://support.pix4d.com/hc/en-us/articles/202560179-How-to-select-draw-the-Processing-Area.

[19] Pix4D. *Using Pix4Dmapper Output Files with Other Software By Output*. 2018. URL: https://support.pix4d.com/hc/en-us/articles/202558499-Using-Pix4Dmapper-Output-Files-with-Other-Software-By-Output.

[20] pix4d.com. *Pix4D manual*. 2018. URL: https://support.pix4d.com/hc/en-us/articles/202557969-Pix4Dmapper-Software-Manual-Table-View.

[21] pix4d.com. *Pix4D support site*. 2018. URL: https://support.pix4d.com/hc/en-us.

[22] Python. *History and License*. 2018. URL: https://docs.python.org/3/license.html.

[23] randerstegl. *randerstegl*. 2018. URL: http://www.randerstegl.no/no/murstein/fullmurt-bygg/beregning-av-mursteinsforbruk.

[24] Kyllingstad S. et al. "Climate, environment and frost damage of architectual heritage". In: (2013).

[25] scikit-image. *skimage.measure.regionprops*. 2018. URL: http://scikit-image.org/docs/dev/api/skimage.measure.html?highlight=regionprops#skimage.measure.regionprops.

[26] scikit-image.org. *Entropy*. 2018. URL: http://scikit-image.org/docs/dev/api/skimage.filters.rank.html#skimage.filters.rank.entropy.

[27] scikit-image.org. *Label*. 2018. URL: http://scikit-image.org/docs/dev/api/skimage.measure.html?highlight=label#skimage.measure.label.

[28] Claude Elwood Shannon. "A mathematical theory of communication". In: *ACM SIGMOBILE Mobile Computing and Communications Review* 5.1 (2001), pp. 3–55.

[29] Spyder. *Spyder documentation*. 2018. URL: https://pythonhosted.org/spyder/index.html.

[30] Bill Triggs et al. "Bundle adjustment—a modern synthesis". In: *International workshop on vision algorithms*. Springer. 1999, pp. 298–372.

[31] Compton J Tucker. "Red and photographic infrared linear combinations for monitoring vegetation". In: *Remote sensing of Environment* 8.2 (1979), pp. 127–150.

[32] Gabriel Versaci. *Facade Inspection in Messina Using Pix4Dmapper Orthoplane*. 2018. URL: https://pix4d.com/facade-inspection-pix4dmapper-orthoplane/.

[33] Stéfan van der Walt et al. “scikit-image: image processing in Python”. In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: http://dx.doi.org/10.7717/peerj.453.

[34] Kesheng Wu, Ekow Otoo, and Arie Shoshani. “Optimizing connected component labeling algorithms”. In: *Medical Imaging 2005: Image Processing*. Vol. 5747. International Society for Optics and Photonics. 2005, pp. 1965–1977.

A Pix4D mapper processing options

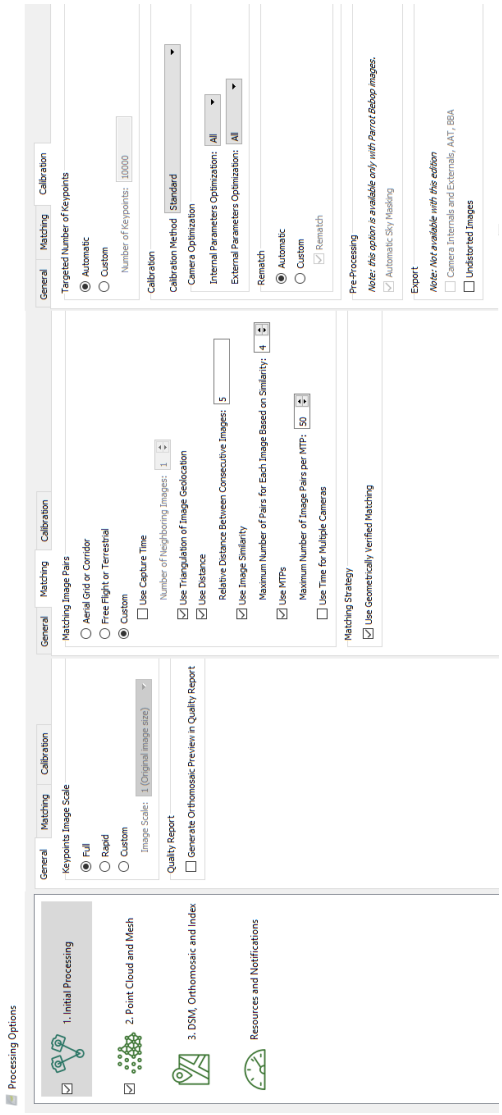

Figure A.1: Settings for step 1: initial processing.

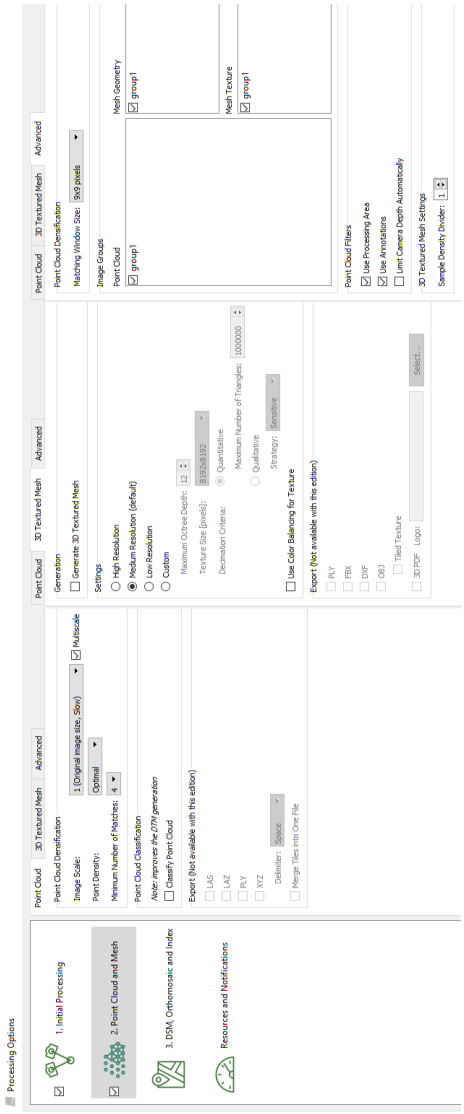

Figure A.2: Settings for step 2: Point Cloud and Mesh.

B Python script

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  5 13:25:31 2018

@author: Espen Johnsen
"""

import matplotlib.pyplot as plt
import numpy as np
from scipy.misc import imread
from skimage.filters import threshold_adaptive
from skimage.morphology import remove_small_holes, remove_small_objects
from skimage.measure import regionprops, label
from skimage.segmentation import relabel_sequential
from skimage.filters.rank import entropy
from skimage.morphology import disk
from skimage import img_as_float

#Read Orthophoto
img = imread('01_East_040.tif')

#Copy img for processing
img = img[:,:,:]
img_edit = img.copy()
img_edit = img_edit[:,:,0]

#Threshold orthophoto
block_size = 17
binary = threshold_adaptive(img_edit, block_size, offset=0)

#Remove small holes in binary image
binary_remove_holes = remove_small_holes(binary, min_size=35)

#Remove small objects in binary image
binary_remove_objects=remove_small_objects(binary_remove_holes, min_size=8)


def horizontal_lines_detection_split(img, th=[], split_x=[]):

    """
    img :: Binary bool image
    th  :: List of thresholdvalues
    split_x :: List of image coordinates
    """

    img_copy = img.copy()
    shape = np.shape(img)

    for j in range(len(split_x)-1):
        for i in range(shape[0]):
            if np.mean(img[i,split_x[j]:split_x[j+1]]) <= th[j]:
                img_copy[i,split_x[j]:split_x[j+1]] = False

    return img_copy

#Detect horizontal lines of mortar in the image
horizontal_mortar = horizontal_lines_detection_split(binary_remove_objects,
                                        th=[0.41, 0.5, 0.42, 0.42, 0.45],
                     split_x=[0, 1225, 1670, 2457, 2921, np.shape(img)[1]])

```

```
#Remove small objects again
remove_objects = remove_small_objects(horizontal_mortar, min_size=8)


def grow_and_split(img_binary_input):

    """
    Input :: A binary image.
    Values should be adjusted for brick size and orthophoto cm/px resolution
    """
    # Gives each region an unique ID.
    labeled = label(img_binary_input, background=0, connectivity=1)
    labeled = relabel_sequential(labeled)[0]

    # Brick that does not conform to the cm/px size of bricks.
    mock = labeled.copy()
    mock[:,:] = 0

    # Binary image that conforms to the cm/px size of bricks.
    iteration = mock.copy()

    #Loops over each region detected
    for region in regionprops(labeled):
        miny, minx, maxy, maxx = region.bbox
        x_length = maxx-minx
        y_length = maxy-miny

        #Big connected regions
        if 8 < y_length <= 16:
            if 300 < x_length:
                mock[miny+1:maxy-1, minx+10:maxx-10]= 1

        #Connected Region of length that corresponds to 5 successive bricks
        if 8 < y_length <= 16:
            if 250 < x_length <= 300:
                split = int((maxx-minx)*(1/5.0))
                iteration[miny:maxy, minx:minx+split-2] = 11
                iteration[miny:maxy, minx+split+2:minx+(split*2)-2] = 11
                iteration[miny:maxy, minx+(split*2)+2:minx+(split*3)-2] = 11
                iteration[miny:maxy, minx+(split*3)+2:maxx-split-2] = 11
                iteration[miny:maxy, maxx-split+2:maxx] = 11


        #Connected Region of length that corresponds to 4 successive bricks
        if 8 < y_length <= 16:
            if 230 < x_length <= 250:
                split = int((maxx-minx)*(1/4.0))
                iteration[miny:maxy, minx:minx+split-2] = 2
                iteration[miny:maxy, minx+split+2:minx+(split*2)-2] = 2
                iteration[miny:maxy, maxx-(split*2)+2:maxx-split] = 2
                iteration[miny:maxy, maxx-split+2:maxx] = 2


        #trippels and a half
        if 8 < y_length <= 16:
            if 198 < x_length <= 230:
                mock[miny:maxy, minx-2:maxx+2]= 3


        #Connected Region of length that corresponds to 3 successive bricks
```

```
        if 8 < y_length <= 16:
            if 148 < x_length <= 198:
                split = int((maxx-minx)*(1/3.0))
                iteration[miny:maxy, minx:minx+split-2] = 4
                iteration[miny:maxy, minx+split+2:maxx-split-2] = 4
                iteration[miny:maxy, maxx-split+2:maxx] = 4

        #doubles and a half
        if 8 < y_length <= 16:
            if 128 < x_length <= 148:
                mock[miny:maxy, minx-2:maxx+2]= 5

        #Connected Region of length that corresponds to 2 successive bricks
        if 8 < y_length <= 16:
            if 99 < x_length <= 128:
                split = int((minx+maxx)/(2.0))
                iteration[miny:maxy, minx+1:split-2] = 6
                iteration[miny:maxy, split+2:maxx] = 6

        #one and a half
        if 8 < y_length <= 16:
            if 69 < x_length <= 99:
                mock[miny:maxy, minx-2:maxx+2] = 7

        #Connected Region of length that corresponds to 1 brick
        if 8 < y_length <= 16:
            if 44 < x_length <= 69:
                iteration[miny:maxy, minx+3:maxx-3] = 1

        #halfs
        if 8 < y_length <= 16:
            if 20 < x_length <= 44:
                mock[miny:maxy, minx-2:maxx+2] = 9

        #under half size
        if 8 < y_length <= 16:
            if 4 < x_length <= 20:
                mock[miny:maxy, minx-2:maxx+2] = 10

    return mock, iteration

def just_split(img_binary_input):

    """
    Input :: A binary image.
    This should be used after the last iteration of grow and split
    """

    # Gives each region an unique ID.
    labeled = label(img_binary_input, background=0, connectivity=1)
    labeled = relabel_sequential(labeled)[0]

    # Binary image that conforms to the cm/px size of bricks.
    iteration = labeled.copy()
    iteration[:,:] = 0

    for region in regionprops(labeled):
        miny, minx, maxy, maxx = region.bbox
        x_length = maxx-minx
        y_length = maxy-miny
```

```
180
181         #Big connected regions
182         if 8 < y_length <= 16:
183             if 300 < x_length:
184                 iteration[miny+1:maxy-1, minx+10:maxx-10]= 1
185
186          #Connected Region of length that corresponds to 5 successive bricks
187         if 8 < y_length <= 16:
188             if 250 < x_length <= 300:
189                 split = int((maxx-minx)*(1/5.0))
190                 iteration[miny:maxy, minx:minx+split-2] = 11
191                 iteration[miny:maxy, minx+split+2:minx+(split*2)-2] = 11
192                 iteration[miny:maxy, minx+(split*2)+2:minx+(split*3)-2] = 11
193                 iteration[miny:maxy, minx+(split*3)+2:maxx-split-2] = 11
194                 iteration[miny:maxy, maxx-split+2:maxx] = 11
195
196
197         ##Connected Region of length that corresponds to 4 successive bricks
198         if 8 < y_length <= 16:
199             if 230 < x_length <= 250:
200                 split = int((maxx-minx)*(1/4.0))
201                 iteration[miny:maxy, minx:minx+split-2] = 2
202                 iteration[miny:maxy, minx+split+2:minx+(split*2)-2] = 2
203                 iteration[miny:maxy, maxx-(split*2)+2:maxx-split] = 2
204                 iteration[miny:maxy, maxx-split+2:maxx] = 2
205
206
207         #trippels and a half
208         if 8 < y_length <= 16:
209             if 198 < x_length <= 230:
210                 iteration[miny:maxy, minx+3:maxx-3]= 3
211
212
213         #Connected Region of length that corresponds to 3 successive bricks
214         if 8 < y_length <= 16:
215             if 148 < x_length <= 198:
216                 split = int((maxx-minx)*(1/3.0))
217                 iteration[miny:maxy, minx:minx+split-2] = 4
218                 iteration[miny:maxy, minx+split+2:maxx-split-2] = 4
219                 iteration[miny:maxy, maxx-split+2:maxx] = 4
220
221         #doubles and a half
222         if 8 < y_length <= 16:
223             if 128 < x_length <= 148:
224                 iteration[miny:maxy, minx+3:maxx-3]= 5
225
226         #Connected Region of length that corresponds to 2 successive bricks
227         if 8 < y_length <= 16:
228             if 99 < x_length <= 128:
229                 split = int((minx+maxx)/(2.0))
230                 iteration[miny:maxy, minx+1:split-2] = 6
231                 iteration[miny:maxy, split+2:maxx] = 6
232
233         #normal and a half
234         if 8 < y_length <= 16:
235             if 70 < x_length <= 99:
236                 iteration[miny:maxy, minx+3:maxx-3] = 7
237
238         #Connected Region of length that corresponds to 1 brick
239         if 8 < y_length <= 16:
```

```
            if 41 < x_length <= 70:
                iteration[miny:maxy, minx+3:maxx-3] = 1

        #halfs
        if 8 < y_length <= 16:
            if 20 < x_length <= 41:
                iteration[miny:maxy, minx+2:maxx-2] = 9

    return iteration

# A list which will contain each iterated binary image.
growth = []

#first run
mocked, iterated = grow_and_split(remove_objects)
growth.append(iterated)

#Second and consecutive runs
number_of_iterations = 6
for i in range(number_of_iterations):
    mocked, iterated = grow_and_split(mocked)
    growth.append(iterated)

#Final run, just split.
final_split = (growth[0] + growth[1] + growth[2] + growth[3] +
                           growth[4] + growth[5] + growth[6])
iterated_final = just_split(final_split)

#Label the image on last time.
final_label = label(iterated_final, background=0, connectivity=1)
final_label = relabel_sequential(final_label)[0]

#This will be the image marked according to entropy values and limits.
entropy_img = img.copy()
entropy_img = entropy_img.mean(axis=2)
entropy_img[:,:] = 0

#This will be the marked image. Center pixels of each region coloured
#according to entropy value and limits.
img_marked = img.copy()

#Size of the structure element to be considered in entropy function.
disk = disk(5)
entropia = [] #This is the entropy distribution.

#Statistics
status = {'green':0, 'yellow':0, 'orange':0, 'red':0}

for region in regionprops(final_label):

    miny, minx, maxy, maxx = region.bbox
    y, x = region.centroid
    y = int(y)
    x = int(x)

    brick = img_as_float(img[miny:maxy,minx:maxx,0])
    ent = entropy(brick, selem=disk)
    ent = np.mean(ent.flatten())

    entropia.append(ent)
```

```
300
301     #green
302     if   ent <= 4.3:
303         entropy_img[miny:maxy, minx:maxx] = 1
304         img_marked[y-2:y+2,x-2:x+2,0] = 0
305         img_marked[y-2:y+2,x-2:x+2,1] = 255
306         img_marked[y-2:y+2,x-2:x+2,2] = 0
307         status['green'] += 1
308     #yellow
309     if 4.3 < ent <= 4.7:
310         entropy_img[miny:maxy, minx:maxx] = 2
311         img_marked[y-2:y+2,x-2:x+2,0] = 155
312         img_marked[y-2:y+2,x-2:x+2,1] = 155
313         img_marked[y-2:y+2,x-2:x+2,2] = 0
314         status['yellow'] += 1
315
316     if 4.7 < ent <=4.9:
317         entropy_img[miny:maxy, minx:maxx] = 3
318         img_marked[y-2:y+2,x-2:x+2,0] = 255
319         img_marked[y-2:y+2,x-2:x+2,1] = 155
320         img_marked[y-2:y+2,x-2:x+2,2] = 0
321         status['orange'] += 1
322
323     if 4.9 < ent:
324         entropy_img[miny:maxy, minx:maxx] = 4
325         img_marked[y-2:y+2,x-2:x+2,0] = 255
326         img_marked[y-2:y+2,x-2:x+2,1] = 0
327         img_marked[y-2:y+2,x-2:x+2,2] = 0
328         status['red'] += 1
329
330
331 #stats
332 total = sum(status.values())
333 green = status['green'] / total
334 yellow = status['yellow'] / total
335 orange = status['orange'] / total
336 red = status['red'] / total
337
338 green_bricks = status['green']
339 yellow_bricks = status['yellow']
340 orange_bricks = status['orange']
341 red_bricks = status['red']
342
343 #Write stats to file.
344 #Uncomment to enable writing of file.
345 """
346 file = open('01_east_data.txt','w')
347 file.write('Total = {0}\nG = {1:.3f} Y = {2:.3f} O = {3:.3f} R = {4:.3f}\n'
348       'G_no = {5} Y_no = {6} O_no = {7} R_no = {8}'.format(total, green*100,
349               yellow*100, orange*100, red*100,
350        green_bricks, yellow_bricks, orange_bricks, red_bricks))
351 file.close()
352 """
353 #Entropy distribution plot
354 #Uncomment to enable plot and saving of result images
355 """
356 plt.hist(entropia, bins=40, color='gray', alpha=1, zorder=2)
357 plt.title('East', fontsize=20)
358 plt.ylabel('number of regions', fontsize=18)
359 plt.xlabel('entropy value', fontsize=18)
```

```
plt.axvline(x=4.3, linewidth=2.0,
color='green', zorder=3, label='Green threshold')
plt.axvline(x=4.7, linewidth=2.0,
color='yellow', zorder=3, label='Yellow threshold')
plt.axvline(x=4.9, linewidth=2.0,
color='orange', zorder=3, label='Orange threshold')
plt.axvspan(2, 4.3, color='green', alpha=0.15, zorder=1)
plt.axvspan(4.3, 4.7, color='yellow', alpha=0.15, zorder=1)
plt.axvspan(4.7, 4.9, color='orange', alpha=0.15, zorder=1)
plt.axvspan(4.9, 6, color='red', alpha=0.15, zorder=1)
plt.legend(prop={'size': 16})
plt.xlim(xmin=2, xmax=6)
plt.tick_params(labelsize=16)
plt.savefig('01_east_entropy.png', bbox_inches='tight')

#Save result images.
coverage = entropy_img > 0.5
plt.imsave('01_01east_coverage.png', coverage, cmap='viridis')

plt.imsave('01_east_binary.png', entropy_img, cmap='binary')

plt.imsave('01_east_marked.png', img_marked)
"""
```

Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway