Andøya Space Center Internship

Elise Wright Knutsen
Marie Henriksen
Even Nordhagen

August 4, 2018

1 Introduction

This project was done for Andøya Space Center, as a summer internship starting on 25th of June and lasting until the third of August. The purpose of this project was to develop a camera system optimized for cloud detection and monitoring of cloud cover and properties. An fish-eye lens is used with a filter wheel, including a red filter and two polarization filters of different sizes, some additional optics and a 24M pixel Sony A7 III camera. The distorted all-sky images are transformed using a distortion function. Several different methods of cloud detection has been explored, finding that using a clear sky background was the best algorithm for our needs, and from the test images so far the subtraction method has worked well, but the FFT seems very promising if more work is done to adjust for some errors.

At last the binary cloud image is projected on a map with geographical coordinates with cloud cover given in okta.

2 Theoretical background

As rays of light from the Sun moves towards us through the atmosphere, it does not travel in straight lines. They bounce off molecules, and depending on the incident angle of the rays and the size of the molecules, the light may change direction and polarization. There are two main models describing this process, is quickly summarized below.

2.1 Rayleigh scattering

The color of the sky is caused by light from the Sun being scattered off molecules in the atmosphere. This type of scattering is called Rayleigh scattering, and its intensity is wavelength dependent, see equation 1. This causes the scattering to be more effective for shorter wavelengths, hence the color we see is blue during the day and redder in the evening when the light is scattered through more particles.

$$I \propto \frac{1}{\lambda^4} \tag{1}$$

Rayleigh scattering can be considered elastic scattering, since the scattered photon does not lose or gain any energy compared to the incident photon during the interaction with the molecule. Due to the small particle sizes compared to the wavelength, the scatter is close to uniformly distributed in all directions.

2.2 Mie scattering

Mie scattering differs from Rayleigh scattering in that it is almost wavelength independent. It produces the bright glare around the Sun, and the almost white light coming off mist, fog and clouds. The particles that produce Mie scatter are generally larger than the wavelength of the incident photon, which results in forward scatter dominating.

2.3 Polarization

As white light from the Sun travels through the atmosphere to our eyes, one or many collisions may cause it to become polarized. White light *can* be polarized, and colored light need not display polarization at all, there is no straight forward connection between color and polarization. Only linear polarized light is considered here, as circularly polarized light is rarely seen in nature.

The polarization of light around us is affected by how the light is transmitted to us, and also how things are lit. When light comes from a well defines source (as the Sun), in general it becomes more polarized than from a diffuse light source. For this reason light from an overcast sky is less polarized than light from a clear sky.

Still, different clouds yield different polarization effects as it is dependent on illumination, composition (ice/water), density and particle size. Incident polarized light on a thin water-cloud is barely changed at all, while ice-clouds have varying effect on the polarized light. Clouds with aerosols, dust-clouds and sand-clouds behave similarly to ice-clouds, though the variations tend to be less extreme.

The Suns position in the sky is also important, as the highest degree of polarization is usually found around 90° from it. During and shortly after sunset the highest degree of polarization is seen around zenith.

2.4 Ice-clouds and water-clouds

It is thought that by using a polarization filter, and rotating it while photographing the same clouds, one can differentiate between clouds of different phase (water, ice or mixed).

As discussed above, ice- and water particles are though to polarize light differently. Ice particles may come in many different shapes and sizes, and thus are harder to fit into a model. The water clouds however fit Mie scattering theory quite well, assuming spherical droplets. If the results are positive it is then possible to not only deduce cloud composition but also altitude from cloud images. [9]

There is one striking difference between sunlit water-clouds and sunlit ice-clouds however. At about 145° from the Sun water-clouds show a sharp rise in polarization, even surpassing its value at 90° from the Sun. The same is not observed for any other type of cloud. [5]

2.5 Viking hypothesis

Viking sailors were exceptional navigators, and were able to sail 3000 km on open waters between Norway and Newfoundland long before anyone else in Europe. In Greenland 1948, archaeologists found a wooden half-circle with multiple lines scratched on the edge. Some scholars think this wooden fragment could be part of a sun compass. But the sun compass need direct sunlight, and would not help when the sun was obscured by clouds.

The hypothesis of polarimetric navigation by Vikings states that the old Norse sailors used sunstones, specific crystals found in Scandinavia, to navigate when the Sun was not visible, either due to an overcast sky or it was below horizon. Recent studies have shown that the known polarization pattern of the clear sky remains quite unchanged by overcast skies. The issue is that the amount of polarization is very small, and seems hard to detect. [3]

When the sky was just partly covered, but the Sun still obscured, it seems more plausible to be able to detect the polarization pattern. As well as for more or less clear skies when the Sun is right below the horizon.

But there are disagreements within the scientific community, with several claiming there simply was no need for a sunstone as the Vikings had a long list of other means of navigating even when the sun was not visible. [8]

3 Distortion correction

The image captured directly by the CCD chip will not be a correct visualization of the sky, as the fish-eye lens will enlarge central objects and shrink objects closer to the horizon. The effect is similar to that of magnifying glasses, so to obtain an image with correct size ratio between all objects, we need to project the image onto a spherical surface. Exactly how this is done depends on the distortion function of the fish-eye lens.

The perfect fish-eye lens, called a "tru-theta" lens, has a linear relationship between the distance from zenith to chosen pixel, r, and its actual $3D$ zenith angle θ. Most lenses however, have a non-linear relationship.

There are several ways to correct for the distortion of the image caused by a fish-eye lens. The most convenient is if the lens producer provides the correct distortion function which can be directly applied to the image.

If that is not the case, a manual camera calibration is quite easy, and there are again numerous ways of executing it.

If the manual calibration is not an option, one can always assume the lens is approximately linear, and just do the math.

3.1 Camera calibration

As distortion data for our Mamiya-sekor lens was not found online, a calibration method to correct for this was developed.

The camera was mounted on a table, with the focal point of the lens directly above the center of a large printed out protractor. A $20m$ long string was attached to this point as well. The other end of the string was secured to the center of a small black mark on a white board. The target was moved on a 5° interval from 0° to 170° to cover the cameras entire field of view.

The distortion function relating r from the image to the physical θ is plotted, and a line fitted to the data providing a suitable polynomial. The distortion function is needed when linearizing the fish-eye image. The polynomial found from this calibration only partially linearized the image, and some images were linearized better than others. We tried to find reasons for these inconsistencies, but were unsuccessful. A modified polynomial was found by trial and error, the final one being:

$$r(\theta) = 0.22x^3 - 0.75x^2 + 1.0735x, \tag{2}$$

which dewarps an image as seen in 1

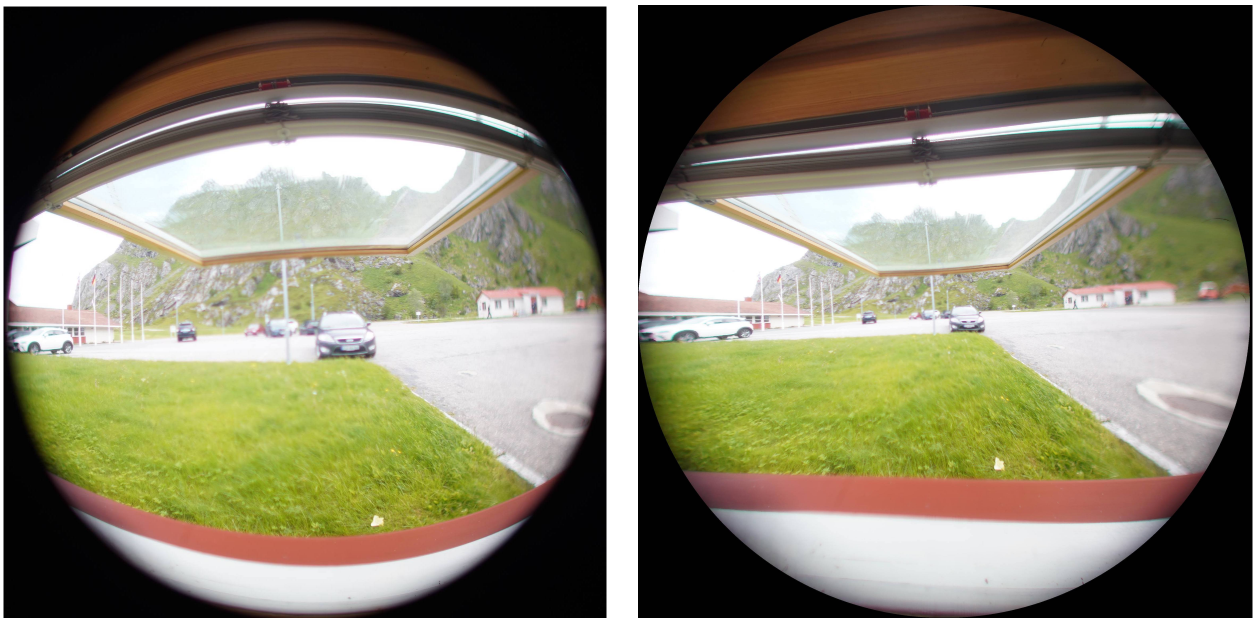

Figure 1: Both images are taken with the Sony camera and the fish-eye lens. The left image is a distorted image, on the left image the polynomial has been applied.

3.2 Projection on spherical image dome

3.2.1 Find location of a pixel on new image

The most intuitive way to correct for lens distortion is to go through all the pixels in the old image and find out where they should be in the new image, using spherical coordinates. In other words, we want to find a function $f(x, y)$ which transforms a set of coordinates from the old image to the new one.

To find this transformation function, we need to take a closer look at how the camera works. The all-sky camera captures light from a dome, where the size depends on the FOV. All the captured light is then placed on a plane surface. To get the correct proportions, the idea is to map the plane surface onto a sphere surface and then flatten it out. The method is illustrated in figure(2) for a camera with FOV of 180°.

To find an expression for the transformation function, we will start from the spherical coordinates:

$$x = r \sin\theta \cos\phi \tag{3}$$

$$y = r \sin\theta \sin\phi \tag{4}$$

$$z = r \cos\theta \tag{5}$$

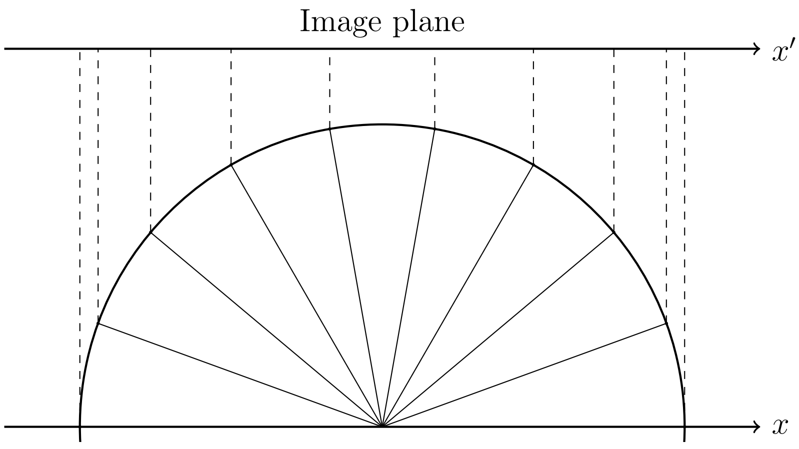

Figure 2: Projection from a place surface onto a spherical surface, in 2D (cross section). In this example the FOV is 180°, for smaller angles the method is the same, but we need to map onto a smaller fraction of the sphere surface.

where ϕ is the azimuth angle and θ is the zenith angle (inclination angle). Initially we know the location of the pixels given in Cartesian coordinates, so we need the angles as functions of x and y:

$$\phi = \arctan(y/x) \tag{6}$$

$$\theta = \arcsin\left(\frac{\sqrt{x^2+y^2}}{y}\right) \tag{7}$$

For a FOV angle of 180° a length in the new image will be proportional to θ, where the leftmost and rightmost points need to be the same. From this we find the relation

$$\frac{r'}{r} = \frac{\theta}{\pi/2} \tag{8}$$

where r' is the distance from a point (x', y') in the new image to the center and r is the maximum radius of the image. Since the azimuth angle, ϕ, is conserved throughout the transformation, we can now easily find the new coordinates using

$$x' = r' \sin\phi \tag{9}$$

$$y' = r' \cos\phi \tag{10}$$

However, often the FOV angle, α, is smaller than 180°, which makes the computations slightly more complex. A distance in the transformed image is no longer proportional to the inclination angle, θ', in the initial image, but it will be proportional to the angle θ between a line staying

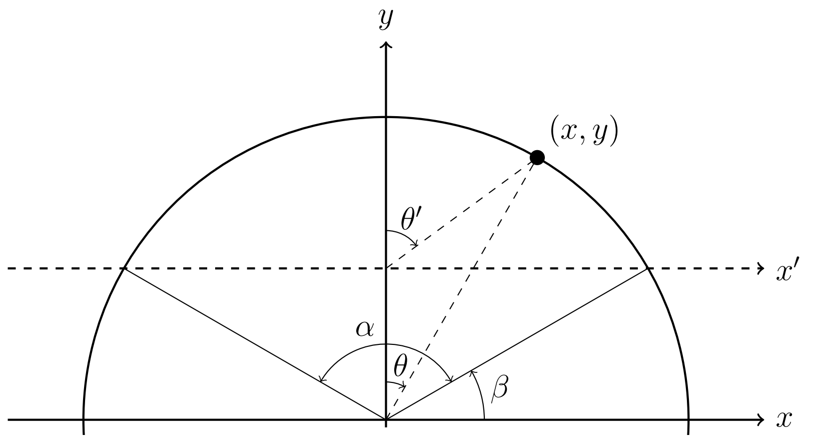

Figure 3: Dome

perpendicular on the sphere and the y-axis, as shown in figure(3). Given an α, this angle θ can be found by

$$\theta = \theta' - \arcsin(\sin\beta \cdot \sin\theta') \tag{11}$$

where

$$\beta = (180^\circ - \alpha)/2 \tag{12}$$

A distance r' in the new image will now be proportional to θ in the same way as in equation (8):

$$\frac{r'}{r} = \frac{\theta'}{\theta_{\text{max}}} \tag{13}$$

where θ_{max} is the angle θ at the point where θ' is 90°, which is directly dependent on β:

$$\theta_{max} = 90$$

$^\circ - \beta$(14)

The point of interest will have the distance r' from the center, and since ϕ again remains untouched, we can easily transform back to Cartesian coordinates using the relations

$$x' = r' \cos\phi \tag{15}$$
$$y' = r' \sin\phi. \tag{16}$$

As the pixels slightly overlap towards the center, they get pulled apart at the edges. This leads to the space in between them being filled with black pixels.

3.2.2 Find location of a pixel on old image

To avoid the stretching problem, we can try the other way around. In the previous section we derived the transformation function

$$x', y' = f(x, y), \tag{17}$$

what if we invert this such that we find a coordinate in the old picture given a new coordinate? This will make it possible to find all the pixels on the new image without getting stretches. The inverse transformation reads

$$r' = \sqrt{x'^2 + y'^2} \tag{18}$$

$$\phi = \arcsin\left(\frac{y'}{x'}\right) \tag{19}$$

$$\theta = \frac{r'}{r}\left(\frac{\pi}{2} - \beta\right) \tag{20}$$

$$\theta' = -\arctan\left(\frac{\sin\theta}{\sin\beta - \cos\theta}\right) \tag{21}$$

$$x = r\sin\theta'\cos\phi \tag{22}$$

$$y = r\sin\theta'\sin\phi \tag{23}$$

4 Cloud detection

To distinguish clouds from clear skies, several methods have been proposed through the years. From the simplest looking only at the intensities of the red channel, evolving into fixed threshold methods for the red-blue ratio. Later came the adaptive threshold method along with background subtraction. All these methods have both strong sides and problematic areas. Here follows our trial and review of several of them.

4.1 Threshold methods

Threshold methods are generally divided into to categories; fixed and adaptive. The fixed methods set a limit for all images, sorting the pixels into "cloudy" or "clear". The adaptive methods change their threshold for each picture, trying to compensate for varying lighting conditions etc.

4.1.1 Red channel intensity

A simple fixed threshold method for cloud detection is to exclusively look at the red color channel, where the sky would appear dark and the clouds more illuminated, and then look at the change in intensity across rows and columns in the image.

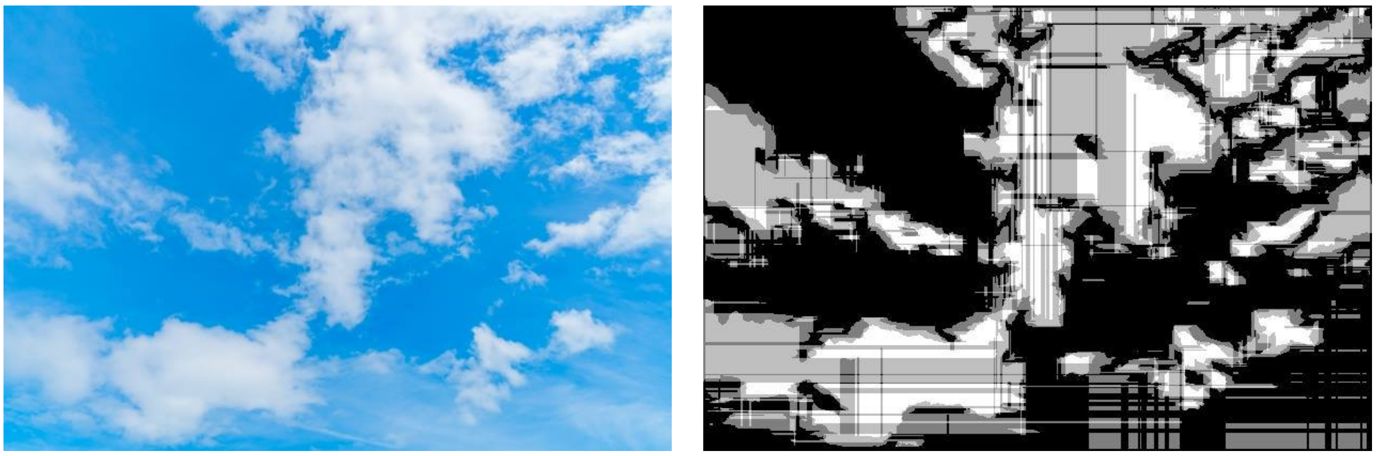
Figure 4: Image of the left shows the test image of a cloudy sky. On the right is the result image, using both horizontal and vertical, back and forth checks for intensity changes greater than 3.

This worked quite well for some regions, but showed serious problems in the horizon and around the Sun, as well as producing "cloud lines" in between separate clouds which were hard to eliminate, see 4.

4.1.2 Red-blue ratio

A fixed threshold method for RBR was tried, but the results were highly variable for different images as well as having trouble near the horizon, detection the entire region as cloudy. We tried using several thresholds for different cloud opacities, see 5, but still the horizon remained a tricky region. different from the clear blue of the region around zenith, the sky dome is divided into these three regions to obtain more accurate thresholds. The Sun's position is given in azimuth and elevation by UiOCam every minute.

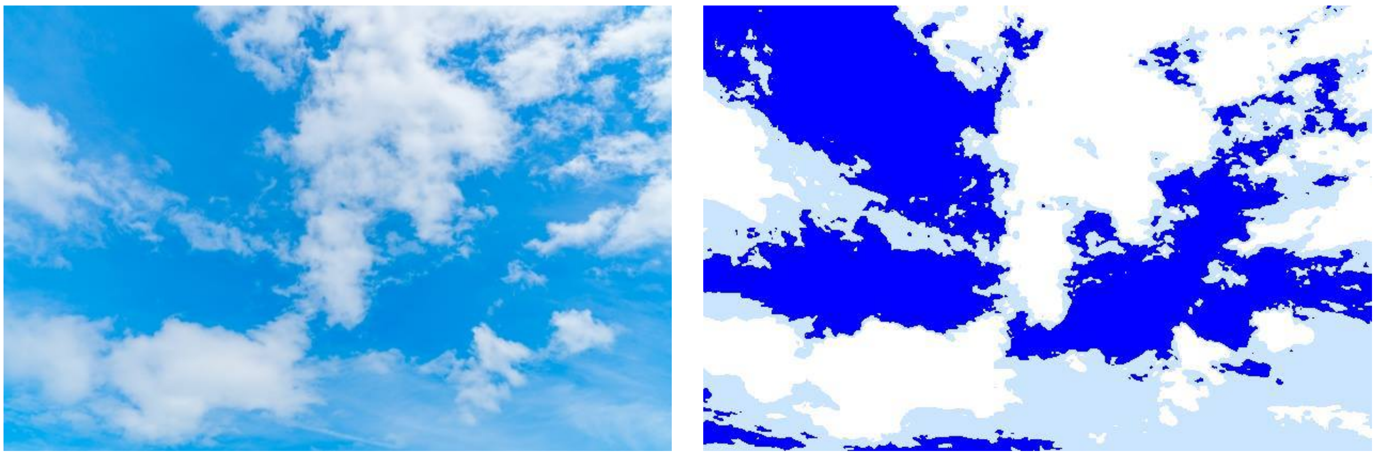

Figure 5: Image of the left shows the test image of a cloudy sky. On the right is the result image, using two thresholds for the RBR, 0.2 for thick cloud and 0.6 for thin cloud.

As the color scheme of the horizon and circumsolar regions are quite different, separate regions were defined with independent thresholds for each region, see 6

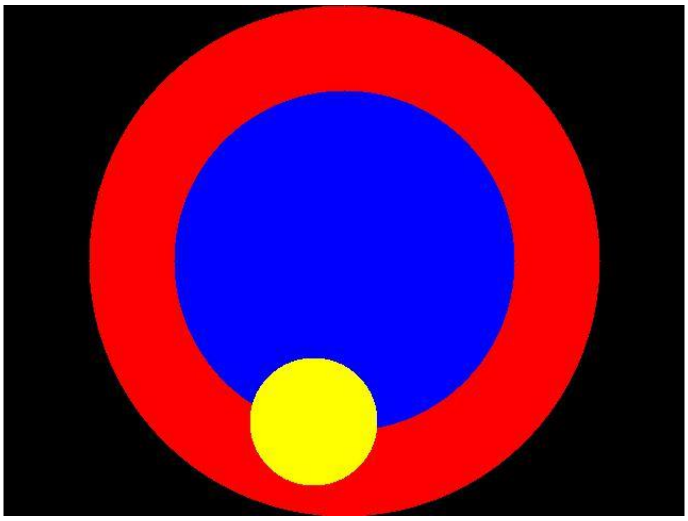

Figure 6: Model of how the three regions were divided. Red region is the horizon, yellow is the glare area around the Sun and blue is the large area around zenith.

While this seemed to fix some of the issues at least in the circumsolar region, it created more problems as a clear line between the regions became visible.

4.1.3 Adaptive threshold

We start out with the original 24 megapixel image, the RBR is then calculated for each pixel, and a grey-scale image is produced. The standard deviation σ of this image is found, and compared to a threshold T_s. If $\sigma < T_s$ the image is classified as bimodal, otherwise the image is classified as unimodal.

A histogram of the grey-scale image is produced. Unimodal images usually produce histograms with only one clearly defined peak, as all the pixels are roughly the same color. This applies for completely overcast skies as well as totally clear skies.

The location of the single peak of the unimodal image reveals if the image is of a clear sky or of overcast conditions. A threshold of $T_f = 40$ is chosen after rigorous testing. If the peak is located below this value, the image is classified as "clear", otherwise it is set to "cloudy".

Bimodal images contain blue skies as well as clouds and therefore have several distinct colors. The histogram therefore has two or more peaks, but often for sky images there is one clearly defined peak for the blue, and an elevated elongated area for all the color tones of the clouds in the image. The standard deviation test is therefore better at checking for uni/bimodal images than looking for the number of distinct peaks. [6]

A threshold for each individual image is chosen by Otsu's method, which aims to minimize the spread of pixels on each side. [2]

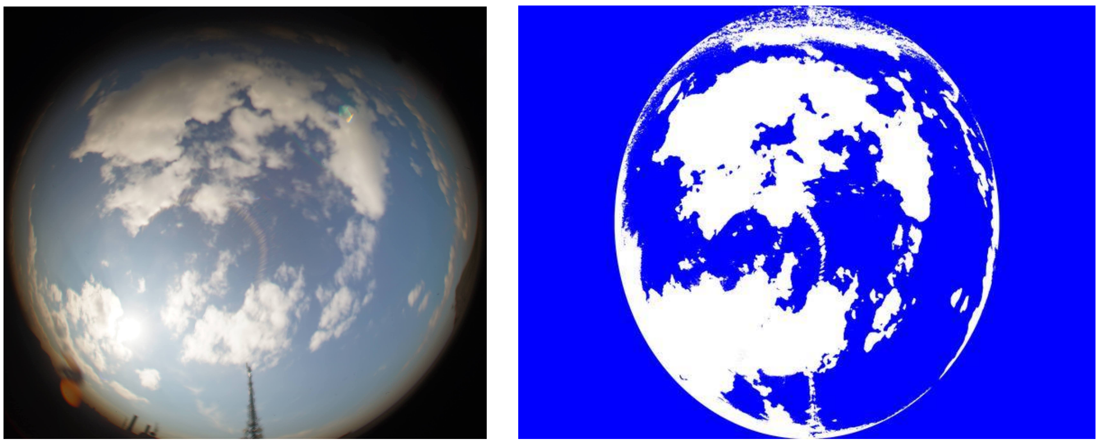

Figure 7: Image of the left shows the test image taken by our Sony camera. The image on the right is the result image using a standard deviation test as well as Otsu's thresholding technique.

The problem of cloud detection in the circumsolar region and close to the horizon was found now as well, as figure 7 shows. Again we tried to divide the image into the same three regions. Two different RBR's are used, $\lambda = \frac{B}{R}$ is the regular ratio while $\lambda_N = \frac{B-R}{B+R}$ is the normalized ratio. λ is less sensitive to thin clouds, but as a consequence makes fewer mistakes around the Sun and the horizon and is therefore used in these regions. The normalized ratio is used in the zenith region.

This did not seem to help much, and the borders between regions were easily spotted.

4.2 Clear sky backgrounds

4.2.1 Virtual

Instead of dividing the image further into many more regions to avoid the boundary problem of the fixed threshold method, a model of the clear sky was attempted. In a clear sky, the RBR is largest near the sun and decreases with increasing sun-pixel-angle (SPA) in the image dome. The RBR also increases near the horizon, (large PZA) due to increased optical path length and larger aerosol concentrations near the surface.

To better distinguish clouds from the sky near the horizon, a background intensity image is generated based on the equation below:

$$L(PZA, SPA) = \Big(1 - \exp\Big(\frac{-0.32}{\cos(PZA)}\Big)\Big) \times \Big(0.91 + 10 \cdot \exp(3SPA) + 0.45 \cdot \cos^2(SPA)\Big). \quad (24)$$

To correct for increased RBR close to the Sun, an exponential decrease in intensity is set as a function of SPA. [1] The Suns position is given by UiOCam.

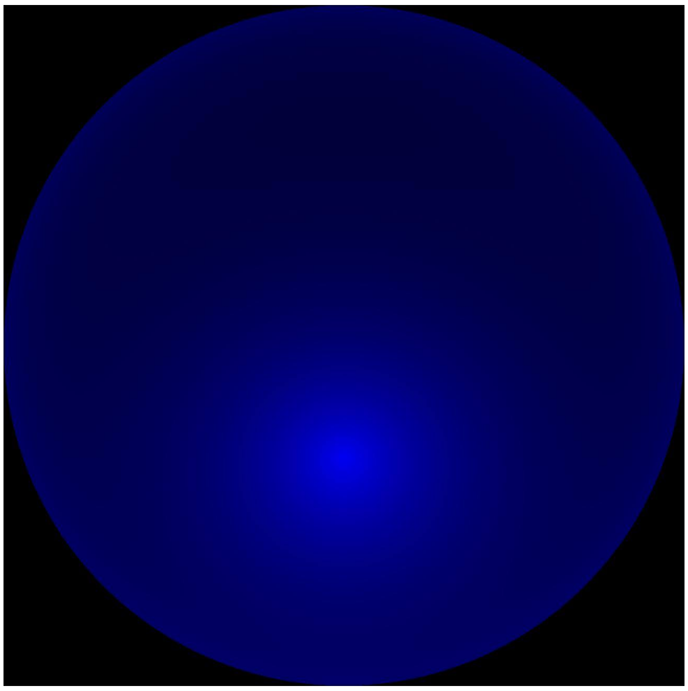
Figure 8: Virtual clear sky background.

The RBR for this clear-sky background (CSB) is calculated and subtracted from the original RBR of the real-sky image. A threshold is then applied to this difference:

$$\Delta_{RBR} = RBR - RBR_{CS} \tag{25}$$

The threshold is set at 0.15. If Δ_{RBR} is above this value, the pixel is cloudy.

Method failed due to not being able to find a wavelength/color dependence. The equation above is supposed to be for the three RGB channels individually, but we do not know the RGB distribution across the sky.

4.2.2 Real

Using a real clear sky image as base, the intensity gradient of the red channel as a function of distance to the Sun and azimuth angle is found. It is assumed that when the Sun is at zenith, the intensity gradient surrounding it is perfectly symmetrical, and as the Sun moves toward the horizon the intensity of a pixel between the Sun and zenith, at a given distance from the Sun, will remain the same. This might not always be the case, but the approximation seems fairly good.

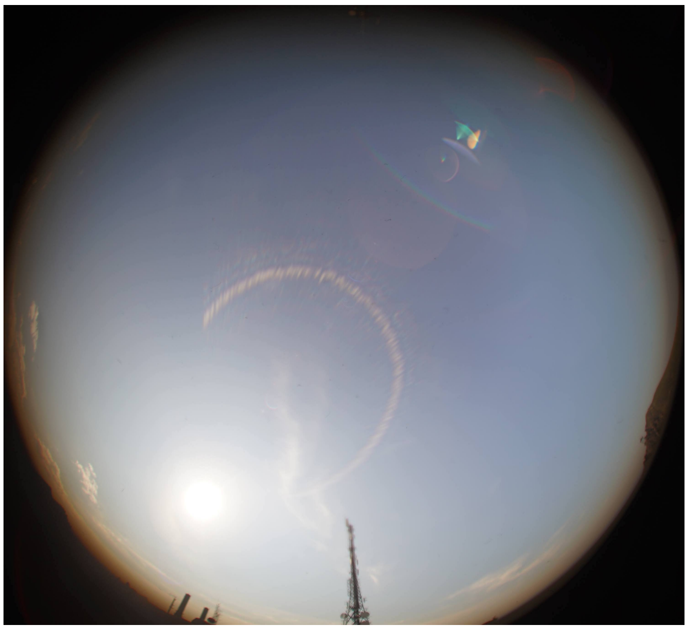

Figure 9: Image used as clear sky background.

The clear sky background image that was used for testing, 9, is not completely free of clouds, but has the closes solar elevation angle that was available to us.

An FFT algorithm is applied to both images, and the frequency distribution in them is compared. The contrast between sky and cloud is enhanced and hopefully removing the Sun and any glare effects from the lens.

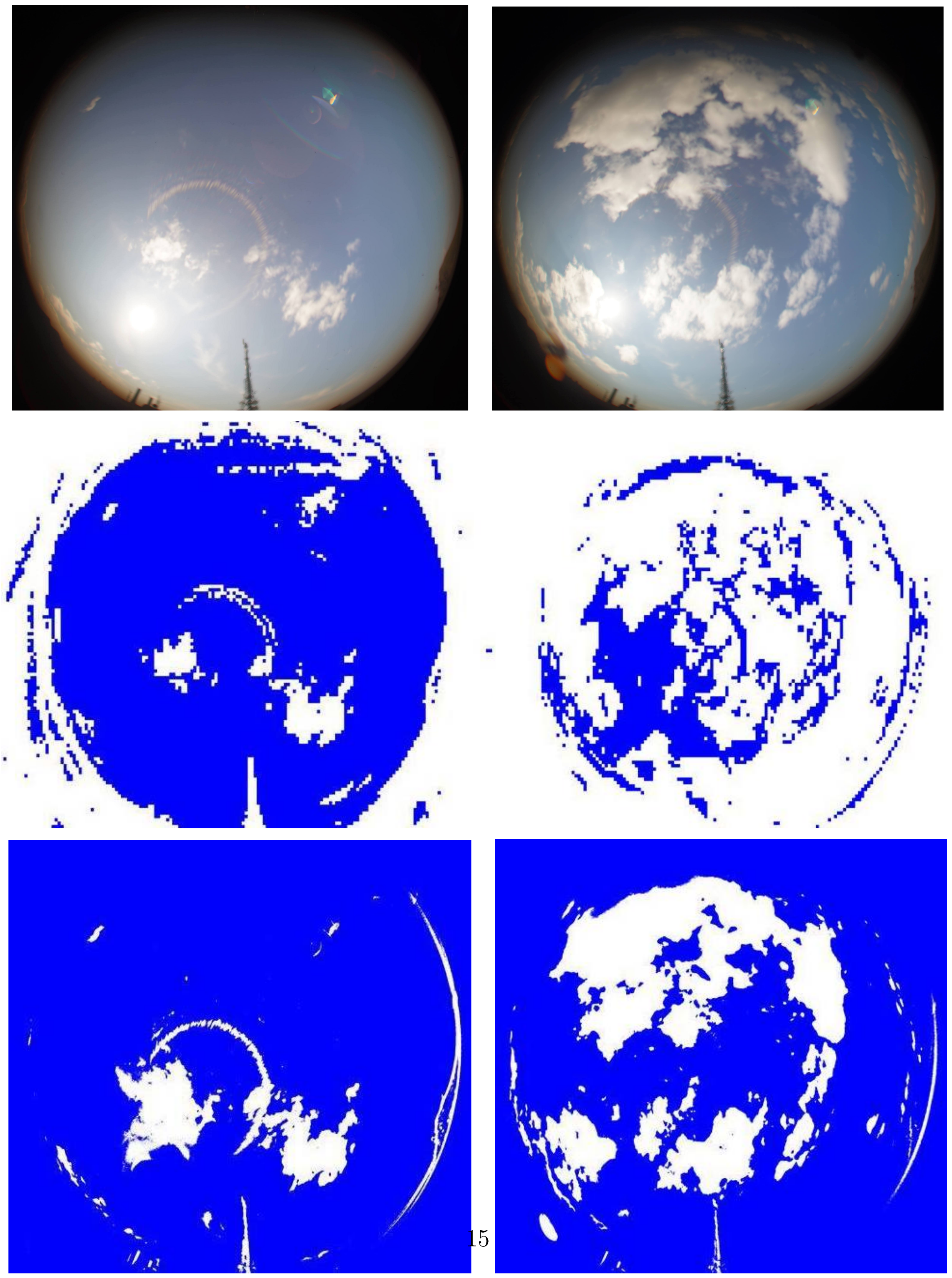

Figure 10: The top row are the original images taken on july 30th. On the second row the FFT algorithm has been applied to both images, and on the third row the subtraction method has been used.

We found that even when the camera was mounted on the instrument, some movement was detected in the images which resulted in the glare and static structures in the FOV not being eliminated. If the camera is kept completely still this should be easily avoided.

Another good method is to simply subtract the CSB from the cloudy image. To subtract all the color channels worked extremely poorly, so as suggested by Yang et al [10], we looked only at the green channel, and got promising results.

Image 10 compares the real cloudy images to both the FFT and subtract methods. The FFT is the second row of images while the subtract method is depicted in the third row. It is clear that the FFT is superior in removing the Sun from the images, but has other obvious problems as seen in the second column. The subtracting method works quite well, but does not quite manage to remove the Sun. The static tower (at the bottom om the images) is almost completely removed however, so it is believed that if a more stable and robust camera system is set up the tower could be eliminated completely.

Ultimately one should have a clear sky background (CSB) for all solar elevation angles, something we did not have time to do. Azimuth does not have to be considered thanks to symmetry. The background image can simply be rotated to fit with the original.

4.3 Cloud types and coverage

Clouds are usually categorized by their cloud base height (CBH), the lower part of their structure. They are divided into three main regions; Low level (2 km) , Mid level (6 km) and High level clouds 13 km). In each region we find puffy white clouds (cumulus) as well as darker more even clouds (stratus) that produce precipitation.

In meteorology, the amount of cloud coverage at any given location is measured in *okta*. Sky conditions are estimated in terms of how many eights of the sky is obscured by clouds. 0 on the scale corresponds to completely clear skies, while 8 is completely overcast. This means that an uncertainty of at least 0.125 is to be expected, even from human observations. The scale does not account for cloud types or thickness, only extent.

The number of cloudy pixels was divided by the total number of pixels and then rounded to the closest eighth to calculate coverage.

5 Cloud projection onto map

When the image has been corrected by a distortion function and the clouds have been detected, the now binary result image is projected over a map of the geographical location of the instrument.

We assume flat Earth, such that no foreign elements are disturbing the image, and flat sky, which simplifies the geometry. The latter is a good approximation when the CBH is relatively small and the field of view angle is some smaller than 180°. To place the clouds onto the map, the CBH, coordinates of the camera (latitude, longitude and altitude), as well as the field of view need to be given.

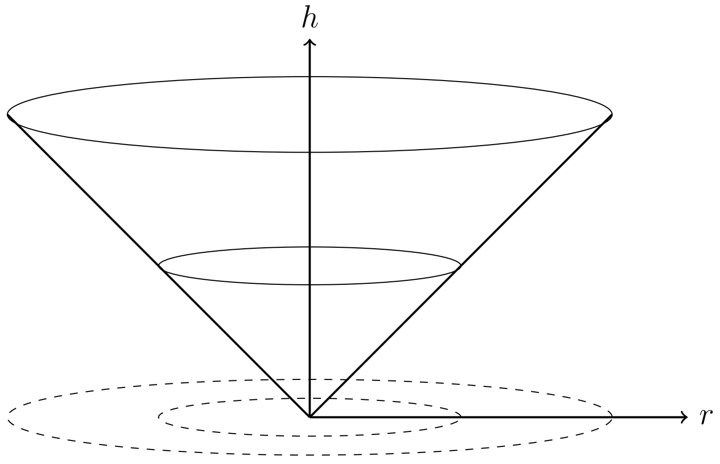

Figure 11: Given a cloud base height, altitude of camera and the field of view angle, one can find which area that is covered by the detected clouds. Here we assume that the Earth is flat and the sky is flat.

5.1 Area occupied by clouds

To find the area occupied by clouds, and then how big fragment of the map we should use, we need to use simple trigonometric identities. It follows easily from figure (11) that the radius of the disc has to be given by

$$r = h \cdot \tan(\text{FOV}/2) \tag{26}$$

where r is the radius of the clouds on the image, with the same unit as h, which is altitude of camera subtracted from the CBH.

Now as we know the area, we just need to find the correct area on a map. The camera position should be the center point of the map, so a map that covers a distance r from the camera position in both the horizontal directions is sufficient. Usually the map packages take coordinates instead of distances, so to get the correct map, the real coordinates of the map edges need to be found.

5.2 Haversine transformation

The coordinates of a distance r from a point is found by the inverse Haversine transformation. To understand why we do this, we will explain the Haversine function briefly. The Haversine function gives the distance between two point on a sphere, given the spherical coordinates of the points,

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\theta_1 - \theta_2}{2}\right) + \cos\theta_1 \cos\theta_2 \sin^2\left(\frac{\phi_1 - \phi_2}{2}\right)}\right), \tag{27}$$

where r is the radius of the sphere (Earth radius), d is the distance on the sphere and (θ_1, ϕ_1), (θ_2, ϕ_2) are the sets of coordinates.

Since we only know one coordinate set and the distance to the other point, we need to invert this. Of course, it will give all possible coordinates on a circumference around the point we know, so we need to specify which one we want. Choosing two points in latitudial and longitudial directions, we find the radius as change in latitude (θ) and longitude (ϕ):

$$\Delta\theta = \frac{d}{r} \tag{28}$$

$$\Delta\phi = 2\arcsin\left(\frac{\sin 2d/3r}{\cos\theta}\right) \tag{29}$$

This is basicly all we need to do to place clouds onto map. The coordinates of the map edges now read

$$\text{right} = \phi_c + \Delta\phi \tag{30}$$

$$\text{left} = \phi_c - \Delta\phi \tag{31}$$

$$\text{up} = \theta_c + \Delta\theta \tag{32}$$

$$\text{down} = \theta_c - \Delta\theta \tag{33}$$

with (θ_c, ϕ_c) as the camera position.

We decided to place clouds onto the map, and then the coastlines onto the clouds such that we could see the coastlines even when it is overcast. An example image is shown in figure (12). The clouds are rotated around the vertical center axis, such that the directions are correct.

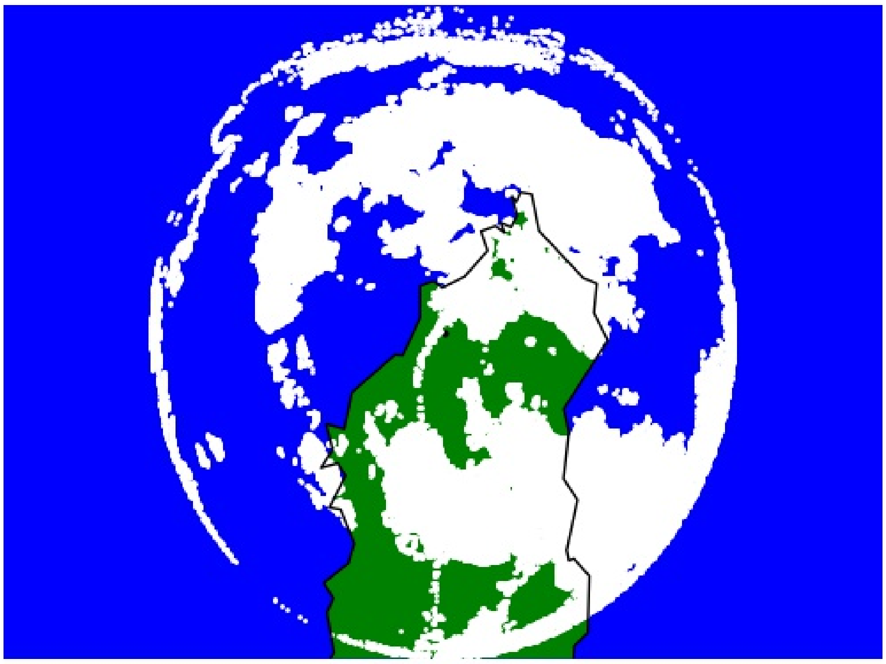
Figure 12: Clouds on map.

6 Image analyzer documentation

First of all, this software is stored in github, but the repository is not public. If you want access to it, please contact Even Marius Nordhagen (email: `evenmn@fys.uio.no`, github account: *evenmn*). If git is already installed on the computer, the repository can easily be cloned by

```
git clone https://github.com/evenmn/Cloud-detection.git.
```

6.1 Requirements

The required packages to run the code are

- NumPy (Numerical Python),
  `pip install numpy`

- MatPlotLib,
  `pip install matplotlib`
- PIL (Python Imaging Library),
  `pip install pillow`
- Basemap,
  `pip install mpl_toolkits`

where PIL is used to transform image to image matrix and basemap is used to project clouds on map.

6.2 Structure

The image editing software is built up such that all communication goes through Main, and subfunctions do not communicate directly with each other. The big advantage of this is that one can easily comment out one of the functions. The code structure can be seen in figure (13).

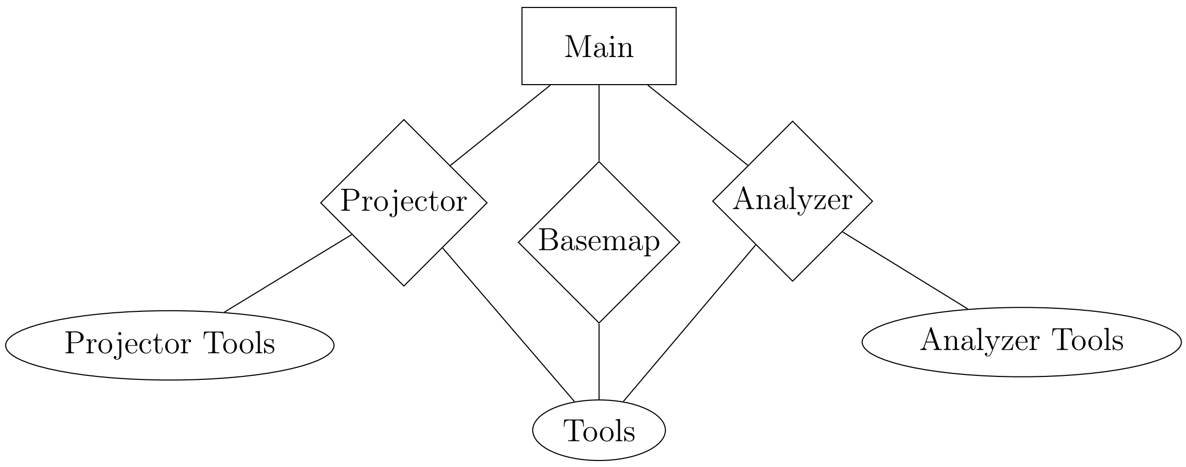

Figure 13: Entity relationship diagram of the code structure.

The main settings are also found in Main, such as camera position, which image to study and so on. To change more specific settings, you might need to enter the functions.

6.3 Functions

Main is where all the magic begins, and to run the program it should be sufficient to run *main.py*. In **Main** the camera position, field of view and cloud base height need to be specified. In addition, this is the place where the user specifies which image to analyze. This can be done in three ways:

- Manually in the script (`'manual'`)
- Manually from the command line (`'cmd'`)
- Automatically (`'search'`)

The latter is probably the most interesting one, which searches for new files in the folder given by path and analyzes new files with extension '.jpg'. This will be useful when the camera takes a sequence of images.

Beside **Main**, the main functions are **Projector, Analyzer and Basemap**, which have their own particular tasks.

6.3.1 Projector

The **Projector** function can be found as *projector.py.* Its main task is to defish the image to get rid of the fish-eye distortion, but it also does a few other things.

1. Remove dark lines in image (**Crop_dark**)
2. Crops the image into square (**Crop_square**)
3. Defishes the image (**defish**)
4. (Draws horizon line on map) (**horizon_line**)
5. Transforming into new format, 4:3 as default (**Crop_format**)

Each of these operations has its own subfunction found in *projector_tools.py.*

6.3.2 Analyzer

The **Analyzer** function can be found as *analyzer.py.* Its main task is to detect clouds on map, using various methods. Currently the implemented methods are

- Surface similarity, (`'FFT'`)
- Subtraction, (`'subtract'`)
- Red-blue ratio + Otsu, (`'RBR'`)
- Normalized red-blue ratio + Otsu, (`'NRBR'`)

Surface similarity is a technique which requires a clear sky image with the sun in the same position as in the cloud image, either real or virtual. Loosely speaking it uses Fourier transformations to find the frequency peaks in both the cloud image and the clear sky image, and subtracts the clear sky peaks from the cloud peaks. This method was stolen from Manoj Raman Kondabathula's master thesis, [4]

Subtraction is another technique which requires a clear sky image. It subtracts the clear sky image from the cloud image directly.

Red-blue ratio + Otsu takes the red-blue ratio of the image and applies the Otsu method to find ideal threshold. It can use the same threshold for the entire image, or it can handle each region separately.

Normalized red-blue ratio + Otsu takes the normalized red-blue ratio of the image and applies the Otsu method to find ideal threshold. It can use the same threshold for the entire image, or it can handle each region separately.

The **Analyzer** also has a function which transforms the image into an image which is transparent around the clouds. To turn on this function, set `transparent` to `True`.

6.3.3 Basemap

The **Basemap** function can be found as *basemap.py*. Its main task is to project images on a map. It has two map settings:

- Simple map, (`'simple'`)
- Satelite map, (`'earth'`)

6.4 Performance

We need the software to be quite fast to analyze images between in real time as they are taken by the camera. The CPU time can be seen in table 1.

Table 1: CPU time usage when projecting and analyzing a 4000pix x 6000pix image with the 'subtract' method.

Intel(R) CORE(TM)	i7-4500U @ 1.80GHz	i5-4300U @ 1.90GHz
Projector	$\sim$ 9s	$\sim$ 8s
Analyzer	$\sim$ 5s	$\sim$ 4s
Basemap	$\sim$ 10s	-

If you want to see the CPU time used by different functions, simply set `show_time` to `True` in **Main**.

7 Instrument

We are modifying an instrument made by KEO Consultants that used to look at the nighttime aurora. The system consisted of a fish-eye lens, a mechanical shutter, light sensor, additional optics, filter wheel, intensifier lens, intensifier, re-imaging optics, camera lens and a camera. When operating in daylight, the light is strong and the intensifier is no longer necessary. The intensifier, the following lens and the light sensor were removed, and the camera was replaced. The new set-up can be seen in table 3.

Fish-eye lens	Mamiya Sekor ULD fish-eye 24mm F4
Mechanical shutter	
Additional optics	
Filter wheel	5-positions, 3-inch filters
Camera lens	Canon FD 85 mm F1.2
Mount adapter	Sony NEX/E-mount to Canon FD
Camera	Sony A7 iii (ILCE-7M3)

Table 2: New hardware set-up.

7.1 Testing (optics)

For testing, a Thorlabs DCC 1645C HQ camera was used. The belonging software, ThorCam, was used to control the camera and capture images.

7.2 Motor control

A motor, SmartMotor from Animatics, is used to control the filter wheel and the shutter. This is controlled from the computer, which is connected to the instrument with a RS-232 cable. The motor can probably be programmed using SmartMotor Interface (SMI), software from Animatics. This software has not been used during this summer internship, but there is a possibility to reprogram the motor to include more functionality. Or destroy the functionality already there. Who knows. Commands that can be sent to the motor from Visual Studio are `g=1..g=5` which will move the filter wheel to filter position 1 to 5, and `d=0, d=1` which will open (1) and close (0) the shutter. More SmartMotor commands can be found in the manual for the finnish system (FMI ASI System).

7.3 2-inch filters

The filter wheel has 5 positions made for 3-inch filters. 2-inch filters are easier to obtain, and will therefore be used instead. Custom adapters from 2-inch to 3-inch are made to fit the filters in the filter wheel. The adapters can be removed if the full 180° field of view is wanted, or if any 3-inch filters are obtained.

7.4 Rotation of polarizing filter

Tentative plan 26.06.2018:
A polarizing filter will be placed in the filter wheel. A mechanism will be placed within the filter wheel so that the polarizing filter will rotate x degrees every time the filter wheel turns.

Update 11.07.2018:
The extra space inside the filter wheel has a height of 2 mm, while our polarizing filters are 3-4 mm outside the adapters. So the polarizing filters does not fit inside the filter wheel unless the can be made shorter, or the adapters can be made deeper.
Update 19.07.2018:
One adapter was made "shorter" on the inside, so that the filter only ended up 1.5-2 mm outside the adapter. When testing, the extra space in the front of the rotating part inside the filter wheel was found smaller than though, close to 0 mm. On the back/inside however, the polarizing filter in the newly made adapter would fit (the old adapter wouldn't). The optimal adapter would be something in between, letting the filter stick approx. 2.5-3 mm out of the filter wheel. A small piece, like a sharp thin screw, could be placed on the edge on the inside/back of the filter wheel case, to turn the polarizing filter a few degrees for each full turn of the filter wheel.
Update 23.07.2018:
It was decided to postpone this functionality.

Note:
A polarizing filter could also be used at a different place in the system. One example could be between the camera lens (Canon FD 85mm F1.2) and the pipe leading to the filter wheel. It would then be easier to fit a small motor close to it to control the turning of the filter. But then the polarizing filter would be present at all times, which could be an unwanted feature when looking at

cloud detection. Some places the contrast between sky and cloud would be better, but other places the contrast could get worse, making it harder to identify the clouds.

7.5 ND filter

A neutral-density (ND) filter could be used to decrease the light reaching the camera chip on sunny days. The Sony A7iii should be sensitive enough to operate during the night, which means that having less light during a sunny (or overcast) day should be no problem at all. Without a ND filter, the pictures could get overexposed by the bright sunlight even with the darkest settings (low shutter speed, low ISO). The ND filter could be placed in the filter wheel if it is not wanted at all times. Else it could be placed between the camera lens (Canon FD 85mm F1.2) and the pipe leading to the filter wheel. How strong/what type of ND filter is needed should be tested.

7.6 Camera mount

With a new camera and the intensifier part removed, a new camera mount must be made.

Tentative plan 20.07.2018:
An ordinary camera mount, a 1/4" screw, will be used to fasten the camera. The other end will be made into something holding onto the body of the system. To get the Canon lens further away from the fish eye lens, a 72mm lens hood will be used and modified to the correct length.

Update 24.07.2018:
The mount keeping the camera body in place has been made, and is fastened with a M6 bolt to the body of the system. Small metal plates can be used to adjust the height of the camera. The camera can also be moved closer to or further away from the filter wheel by moving the whole camera mount.

8 Software

The main software used will be the same as for the old system, uioCam, but an updated version, uioCam v2.0. In addition, software for camera control and data processing is needed. An overview of the different programs used can be seen in table 3.

Main program	uioCam v2.0
Camera remote control	Imaging Edge, Remote (Sony)
Data processing	Various python scripts

Table 3: Software used.

8.1 Operating system (OS)

The old system used Windows XP (32-bit). The main program, uioCam, was written in Microsoft Visual C 2010 on Windows 7. The new software for remote control of the Sony camera needs a 64-bit OS. A laptop with Windows 10 (64-bit) is therefore chosen as the new working station (for this summer), but this caused problems with the old uioCam software.

8.2 Moving uioCam to a new computer

To run the old program on a new computer with updated OS, the easiest way to get it going is to download Visual Studio and create a new project. Visual Studio 2017 was downloaded to the student laptop with Win10, and uioCam was recompiled. After many errors and fixes the program actually compiled successfully. And this is how to do it (in VS 2017):

- Choose New → Project. Choose Installed → Visual C++ → CLR → CLR Empty Project (if this does not exist, download more stash from Visual Studio, make sure "C++/CLI support" is installed).
- Right click on the project, choose Add → New Item. Choose Visual C++ → C++ File (.cpp). Add a .cpp file for the main function, named something like "uioCam.cpp" (use the name of the project you just made).
- Right click on the project, choose Add → New Item. Choose Visual C++ → UI → Windows Form. (Let it be named MyForm.h, it makes life easier later on).
- Copy all files (.cpp and .h) from the original folder of the project (except files like MyForm.h, uioCam.vcxproj etc.) to the folder of your project.
- Right click on "Header Files" and choose Add → Existing Item and select all the .h files in your project folder, and include them. Repeat for the .cpp files and the "Source Files" folder.
- Copy whatever is in the original main file (uioCam.cpp or similar) over to the new uioCam.cpp (or similar) file in your project.
- Copy whatever is in the original MyForm.h file over to the new MyForm.h file. Make sure you have the MyForm.cpp file as well.
- Choose Project -> Properties -> Linker. Then go to System and change Subsystem to "Windows (/SUBSYSTEM:WINDOWS)", and go to Advanced and change Entry point to "main". **Note**: this may give an error in MyForm.h [Design] (to fix: undo this step, restart VS and reopen MyForm.h [Design]). Doing this step prevents the terminal window to pop up when the program is started. So if further development is needed, and the terminal is wanted, this

step can wait. But for the final user, the terminal window will be of no use, so it's better to just remove it. If the terminal window is closed, the application will also be closed, which could be impractical during an experiment.

And that should be it, hopefully it will compile on first try. Also, remember to create and copy the config.txt, time.txt, intensity.txt and sunmoon.txt in the etc folder, and update all paths in the define.h file.

8.3 The folder set-up on the student laptop

The version of uioCam that should be used is uioCam_v2.0 that is found on the desktop in the folder called uioCam (`C:\Users\Student\Desktop\uioCam`). In this folder, several folders exist. An overview of the different folders can be seen in table 4.

data	Should hold the pictures taken.
etc	Has the files config.txt, time.txt, image_compressed.bin, image_uncompressed.bin, intensity.txt and sunmoon.txt which are all used by the program.
graphics	Has the uioCam logo in different file formats.
info	Should include info files for the camera system used (?).
log	Should hold the log files.
pictures	Should hold the last picture taken, in the subfolder last.
tmp	For image and log file to be read during init of program (?).
uioCam_v2.0	Holds all project files, as well as the solution file and all executables.

Table 4: Folder set-up.

8.4 Controlling the camera (options)

Sony has some finished software called Imaging Edge, where Remote is the subprogram for controlling the camera from the computer. To control the system, we have (this far) three options:

1. Use Sony's Imaging Edge and Remote to control the camera. Find a way to talk to Remote, so that either everything can be controlled from uioCam, or so that several sessions can be set up in Remote (and then do a fix in uioCam to synchronize the shutter).

2. Use Sony Camera Remote API to talk with the camera from uioCam over wifi.

3. Port everything over to linux and use gphoto to control the camera (if it's even supported). Then also write everything of motor control and framework so that it can run in linux aswell, and then integrate this with gphoto.

4. Find a way to use gphoto on windows and then use gphoto to control the camera (once again, if the camera is even supported). And find a way for uioCam to communicate with gphoto.

For both option 3 and 4, however, it look like Sony ILCE-7M3 is not yet a supported camera. If this is the case, there's not really any point in trying to get gphoto to work. If the system is ported to Linux in the future (and ILCE-7M3 is supported), this could be used with less problems.
If we know we have wifi and that this can be used (remember to think about the security), then option 2 could be an okay idea to try. It looks like someone already tried to do this from C++, so implementing the code in uioCam could be worth a try. But if communication over wifi is to be avoided, there is no point in working on this yet.
This leaves us with option 1.

Option 1 is therefore used. Remote is controlled from uioCam using Windows System functions. Only simple commands are used to control Remote to take photos, so this was definitely the easiest solution. Commands used can be seen in table 5.

" "	(space)	Turn live view on/off.
"1"		Take picture.
"%{F4}"	(alt+F4)	Close window.
"s"		Shutter speed up (and activate shutter speed).
"+s"	(shift+S)	Shutter speed down (and activate shutter speed).
"i"		ISO up (and activate ISO)
"+i"	(shift+I)	ISO down (and activate ISO).

Table 5: Commands sent from uioCam to Remote.

8.5 New functionality in uioCam_v2.0

With a new camera and a new main objective, new functionality is needed. Most of the old functionality including the light detector, intensifier and the temperature controller was deleted. If this functionality is ever needed again, one may take a look at the old code from the original uioCam and include these functions again. Further, some changes were made in general to make the program run in Visual Studio 2017, instead of 2010. But these were only minor changes and should not be hard to fix again if this should ever be needed.

smartMotorFullTurn()
If polarizing filters are included and these are to be rotated by rotating the filter wheel, a function turning the filter wheel a full turn would be handy. So this function was implemented and can be found in "smartMotorIo.cpp". A button running this function was also made and placed in the smartmotor section of the application.

sonyCameraIo.cpp, sonyCamera.h

The camera is controlled from uioCam through Sony's Remote application. The communication between uioCam and Remote happens in the functions in "sonyCameraIo.cpp". The name of the Remote application and window are defined in "sonyCamera.h", these are needed to use the Window System functionality to control the Remote application window. The commands sent from uioCam to Remote can be found in table 5.

Opening .txt files

New buttons were made for opening the different textfiles: config.txt, time.txt and ReadMe.txt. When setting up experiments some of these files must be edited (ReadMe is just handy if you're new to the program), so having buttons in the gui makes it easier to find the correct files. The path used is the same path the program uses when reading the files. So even though there are several files with the same name in different locations, these buttons will lead to the correct files. The functions used by these buttons are placed in "sonyCameraIo.cpp", mostly because this is a file already heavily edited. They can be moved to a better location, but this can be done later.

initiateCamera()

New code was added to getting the camera ready. Using `System::Diagnostics` for starting processes, getting handles etc., `System::Windows::Forms` for sending keys and `System::Threading` for sleeping, a set up was made to make Sony's Remote start up and shift in to "small mode"/remove live view (so that it doesn't take the whole screen).

Todo:

Must make safeguards. Check if the program starts or not. Does anything happen if it's already open? I think Remote deals with this itself, but could do a proper check. What to do if Remote can't start (not downloaded, path is wrong etc.)?

acquireImage()

Using the same principles as in `initiateCamera()`, the window handle of Remote is found and the program set in the foreground. This makes it active, and the command "1" is sent to Remote which makes it take a picture.

Todo:

Must make safeguards. Check that picture is taken. What if Remote isn't open when picture is taken? How to detect if picture is taken? Anything else that can go wrong?

closeCamera()

Using the same principles as in `initiateCamera()`, the window handle of Remote is found and the program set in the foreground. The command "%{F4}" is sent to close the window of the application.

Todo:

Must make safeguards. Check that Remote is running, if not this command will exit uioCam (it sends the command to whatever window is active). Check that Remote is closed etc.

secTimer_Tick()
The old uioCam was based on the user saying what seconds in the minute pictures were to be taken, and checked each second in `secTimer_Tick()` if this was the case for this second. New functionality is implemented so that the user says what second in the minute to start, what minute in the hour to start and then what the interval time between each picture should be. On the down side, pictures can no longer be taken with uneven intervals. The functionality of changing filter between exposures is not implemented again either, but this could be done in the future if this is necessary. What's good however, is that a picture now can be taken with several minutes interval. Or once an hour. With the old implementation, you got to have at least one picture each minute. When changing this function, some changes were also made in `ReadCamSetupFile()` and `ExposeImageSeqNew()`, in func.cpp and sequence.cpp respectively. The content of sequence.txt was moved to config.txt to make setting up the experiment more intuitive for the user.

Changes were also made in the enabling of `minTimer_Tick()`. Originally this would wait for the first new minute, then one more minute before being activated. A dummy interval is set for the minTimer so that it first tick will be at the first new minute instead. All in all, this means that when the programs is started (or the config files reloaded) the program will start running at the next whole minute.

Changes in config.txt
Since we no longer want pictures every minute with specified filters and exposure times, a new sequence file was created. This was later moved to the config.txt file, to keep the number of files the user must use to a minimum. The main change made was the interval parameter included, which decides the time between each picture. This gives the opportunity to take several pictures a minute, or a picture every 5 minutes. The parameters `sec_start` and `min_start` were included in case an experiment should be started at a specific time within the hour. If the experiment should start at a specified time within the day or at a specified date this can be set in time.txt.

8.6 Even more new functionality in uioCam_v2.0

If time would allow, plenty of functionality could be integrated in uioCam v2.0. Unfortunately, time is running out. This subchapter is meant to describe some of the functionality implemented at the end to make the system more usable for our objectives, looking at cloud coverage and polarization properties. But since these changes probably will be left unfinished, it's nice to have the thought behind them in writing.

The outline goes like this:

We wish to let our camera look at cloud coverage. It must then both manage picturing the clear and overcast sky. To do this, something automatically controlling the shutter speed and/or ISO should be used, to get the pictures as good as possible. One static setting will most probably not be good enough for both cases, and all the cases in between. One choice is to use the auto function on the camera itself. But then we can't control how the auto should work. This could lead to overexposed images on sunny days, which would be nice to avoid. Another option is to implement a function in uioCam that compares either the value of the brightest pixel to a predefined limit to see if anything is overexposed. Or we can look at the lowest value of the 100 highest values or something like that. Or the average value. Then the result can be used to adjust the shutter speed and/or ISO. The camera must then be in manual mode.
Then we wish to let our camera look at the different polarization. We must now have settings giving us a good picture of the day, but also static settings to be able to detect the potential difference in polarization when the polarizing filter is turned. The polarizing filter position must also be chosen, and the filter wheel turned around frequently.

Solution:
Different running modes.
1. Normal
2. Cloud detection
3. Polarization

In normal mode, the system will run as it does at this point in time, before any of this is implemented. The parameters from config.txt and time.txt are what controls uioCam's autocontrol, and no magic happens.
In cloud detection mode, cloud detection parameters are chosen, and the camera should run with auto settings, either decided by the camera itself or by the intensity control function in uioCam (see description below). In polarization mode, polarization parameters are chosen, and the camera should initialize and find an acceptable shutter speed and ISO for the experiment. These camera settings will be kept static, then the filter wheel turned so that pictures with differently positioned polarizing filter are taken. The pictures can then be compared to find the potential difference in intensity, which is what we want to look at.

In the future, a fourth mode can be added:
4. Cloud detection and polarization
The thought behind this one is to have an algorithm looking at the cloud base height (using ceilometer), the cloud coverage and possibly the cloud types (broken or whole) to decide whether or not we should switch over to polarization mode. There is also a possibility to run a combination of mode 2 and 3 at the same time, where the parameters are checked/switched between each image. If the cloud detection mode wants pictures every 3 minutes and the polarization mode every 10 seconds, this can easily be done. Just start polarization at `s=0` and cloud detection at `s=5`, as long as they

have intervals so that they never want a picture at the same time this should be no problem at all.

Further, other modes can also be included if needed:
5. Cloud speed
6. Night mode
or modes for other filters like red/IR filters etc.

What is done:

- Parameters are included in config.txt: `RUN_MODE`, `CLOUD_INTERVAL`, `CLOUD_FILTER`, `POL_INTERVAL`, `POL_FILTER`, `INTENSITY_CONTROL`.

- New struct, `CAMMODES`, made in include.h. Both this one, with params from config file, and an int for `run_mode` and `intensity_control` were added in `CAMSTAT`. `ReadCamSetupFile()` was changed so that values for these params are set from the config file as well.

- The file modes.cpp was made with functions `initCloudMode()`, `initPolMode()` and `initNormalMode`, with include.h as header file. The init functions switches to correct interval time and filter for the mode.

- intensity.txt was made. Limits for the method chosen should be put here. The code must be changed accordingly. For a start a max and a min limit were made. A new struct, `CAMINTENSITY`, was put in `CAMSTAT` and a function `ReadIntensityFile()` (in sonyCamera.cpp) was also made to get the values from the file to the program. The path to the file was also added in define.h.

- More functions implemented in sonyCamera.cpp: `runIntensityControl()`, `compareIntensity()`, `makeBrighter()`, and `makeDarker()`.

- Safeguard made in `runIntensityControl()` catching the case were we have a picture with too much/little intensity but ISO and/or shutter speed are maxed out so that no changes are made. This could give an infinite loop (which we don't want), should go in a "this is not good enough but it's the best we've got" mode, and check again at a later time to see if the conditions have changed. This should be developed further when using the function.

- Night mode was included.

- All params moved to config.txt, it is easier to only have one place to set all the values for the modes. (time.txt still controls the allowed time interval).

- Sun/moon max/min elevation was added for night mode only. Limits for the other modes could also be included in the future if this is needed.

- The modes are tested, but only a tiny bit. They should work in theory, but long time testing should be done.

What is left:

- How to choose if ISO or shutter speed (or both) should be changed between each picture.
- makeBrighter() and makeDarker() must be tested.
- A program checking the desired intensity measure of the picture must be made, and write the result to the intensity.txt file (set `image_intensity`).
- Safeguards, safeguards, safeguards! The modes and intensity control should be tested properly and catch exceptions for funny scenarios so the program doesn't stop running if anything happens in the middle of an experiment.
- Could make a param in config.txt to say if you run in night mode and the sun elevation exceeds max limit, the system should switch to normal mode (or some other mode). Then it can switch back to night mode if the sun goes back down below the limit. And then implement this feature.
- Find a way to check if the camera itself is in auto mode or manual mode (or some other mode). The camera must physically be in manual mode for Remote to be able to control the shutter speed and ISO.
- It could be nice to be able to read the shutter speed and ISO values from Remote. Could be used in finding and setting limits, deciding how many steps in changing shutter speed or ISO is needed, and figure out when they are maxed out and makeBrighter()/makeDarker() no longer will have an effect.
- Proper testing of the whole system.

9 Additional comments

- All the "y_Astro..." files are used to calculate the position of the sun and moon. The "zlib..." files are used for file compression to .png. The latter functionality is not really used in uioCam v2.0.
- There is a lot of "y_Astro..." files included in the project. Only a few of these are actually needed, but they all depend on each other. There is a possibility to exclude most files and examine what functions are actually used to calculate the position of the sun and moon. All

files included in the project are needed for the program to compile without errors, without this clean up being done.

- Most of the functionality handling temperature control, EMCCD gain, light detector and intensifier stuff has been excluded, but a proper clean up is yet to be done. The binning and setting exposure time from gui are also removed.
- The camera must be turned on at all times (when the program is running and supposed to take pictures). There are no safeguards at the moment making sure that the camera is turned on, that it has power etc.
- The camera must physically be switched between auto and manual mode. It must be in manual mode if the shutter speed and/or ISO is to be controlled.
- Remote has no way of setting the shutter speed by number, only giving up/down commands.
- File names (picture prefix and number) as well as where the pictures should be saved (file path) must be specified in Remote. This should be specified by the user before the shoot begins (no big deal if it isn't but it'll probably make your life easier when analyzing the data).
- The intensity control should be made so that all pictures taken within the intensity control function can be discarded, and the real picture taken when the settings are good. How/when the real image will be taken (not sure how long the intensity control will take) is unsure atm, this must be decided. If intensity control is on, a new check will be run every hour. This could also be made into an input parameter, telling how often a check like this should occur.
- Proper testing of the software should be done, and fixes should be made before running any serious experiments.
- Most new comments are marked with //–, these are usually comments to what can be made, or stuff that was included earlier but I didn't want to remove.
- Would recommend a ND-filter for daytime operation. Less chance for overexposed pictures.

10 Future work

Given more time, this project could also grow to include:

10.1 Ceilometer

The plan is to include data from a ceilometer when the instrument is made available. This will accurately give the CBH and make the map projections better.

10.2 Constant structures in image

As the camera has an extremely large FOV, some static structures will probably be visible when the camera is mounted. These needs to be masked out before the cloud detection procedure so that pixels containing non-sky is not interfering with the thresholding/gradient method.

10.3 Ice clouds

Detection of cirrus clouds based on polarization. Only necessary when okta is less than 4/8 and there is a mix of high and low clouds.
- turn polarizing filter 180°
- take pictures every 10° to find min and max
- Take all pictures in quick succession
- observe difference in low water clouds and high ice clouds

10.4 Cloud detection at night

By comparing pictures of the clear night sky, using known star positions, the location and coverage of clouds could be determined at night as well.

10.5 Cloud brokenness

By calculating the cloud brokenness (CB), an "edge-to-area" ratio defined as the number of pixels in the image on cloud/clear boundaries divided by the sum of cloudy pixels in the image, the average cloud size can be determined.

A large edge-to-area ratio relates to broken clouds of small diameter, while a small edge-to-area ratio relates to extended clouds. [7]

The cloud coverage in okta, together with the CBH and CB can be used to infer the cloud type present in the image.

10.6 Cloud forecasting

By comparing consecutive pictures using cross correlation, cloud velocity and direction of motion can be determined.

10.7 Software improvements

The image editing tools can be improved, especially when it comes to projecting clouds onto map. The basemap solution is not ideal, since the map resolution is quite low. A better solution might be the **gmplot** package in Python (`https://github.com/vgm64/gmplot`) which we did not have time to investigate fully.

One can also try other cloud detection methods, but we believe that the implemented methods are quite good with a large image library. For the future one should try to enlarge this library, such that one has at least a clear sky image for all the sun positions (every 0.5 degree should be sufficient). A big data set with cloud images for testing the algorithm will also be needed.

On thing that does not work, is the algorithm that searches for new '.jpg' files in path folder, see *main.py*. The overall performance can also be improved.

References

[1] Rémi Chauvin, Julien Nou, Stéphane Thil, Adama Traore, and Stéphane Grieu. Cloud detection methodology based on a sky-imaging system. *Energy Procedia*, 69:1970–1980, 2015.

[2] Andrew Greensted. The lab book pages, otsu thresholding. `http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html`, (accessed June 17, 2010).

[3] Gábor Horváth, András Barta, István Pomozi, Bence Suhai, Ramón Hegedüs, Susanne Åkesson, Benno Meyer-Rochow, and Rüdiger Wehner. On the trail of vikings with polarized skylight: experimental study of the atmospheric optical prerequisites allowing polarimetric navigation by viking seafarers. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 366(1565):772–782, 2011.

[4] Manoj Raman Kondabathula. Camera based detection and localization of noctilucent clouds.

[5] G. P. Konnen and H. G. Begbie. *Polarized Light in Nature.* Cambridge University Press, 1980.

[6] Qingyong Li, Weitao Lu, and Jun Yang. A hybrid thresholding algorithm for cloud detection on ground-based color images. *Journal of atmospheric and oceanic technology*, 28(10):1286–1296, 2011.

[7] Charles N Long, Jeff M Sabburg, Josep Calbó, and David Pagès. Retrieving cloud characteristics from ground-based daytime color all-sky images. *Journal of Atmospheric and Oceanic Technology*, 23(5):633–652, 2006.

[8] Curt Roslund and Claes Beckman. Disputing viking navigation by polarized skylight. *Applied optics*, 33(21):4754–4755, 1994.

[9] X-M Sun, X-W Shi, and Y-P Han. Reflection of polarized light in ice-water mixed clouds. *Journal of electromagnetic waves and applications*, 20(12):1655–1665, 2006.

[10] Jun Yang, Qilong Min, Weitao Lu, Ying Ma, Wen Yao, Tianshu Lu, Juan Du, and Guangyi Liu. A total sky cloud detection method using real clear sky background. *Atmospheric Measurement Techniques*, 9(2):587–597, 2016.