Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR THESIS

Study program/specialization: Bachelor of engineering / Data technology	Spring semester, 2021 Open
Author: Joachim Andreassen	
Subject manager: Erlend Tøssebro Supervisor: Steve Jothen	
Title of bachelor thesis: World-wide cloud data compiled from satellite imagery	
Credits: 20	
Keywords: clouds, satellites, wavelengths, projections, caching, world-wide, image processing	Number of pages: 105 + attachments/other: 4 Stavanger - May 15, 2022

Contents

Contents **i**

Glossary **vii**

Summary **x**

1 Introduction **1**

1.1 Problem 1

1.2 About the Company 2

1.3 Structure 2

1.4 Technologies 3

1.4.1 Programming language 3

1.4.2 Satpy 3

2 Theory **5**

2.1 Satellites 5

2.1.1 Weather satellites 5
2.1.2 Earth coverage 6
2.1.3 Resolution verification 11
2.1.4 Scanning intervals 15
2.1.5 Future replacements 16
2.2 Satellite frequency bands 17
2.3 Obtaining satellite data 19
2.3.1 Online resources 19
2.3.2 Separation of data 20
2.4 Projections 20
2.4.1 Projection selection 21
2.5 Creating world map 23
2.5.1 Reading satellite data 23
2.5.2 Resampling 24
2.5.3 Combining satellite images 28
2.6 Cloud detection 30
2.6.1 Definition 30
2.6.2 Challenges 31
2.6.3 Methods considered 32
2.6.4 Solution 33

2.7 Product generation 34

2.7.1 Images 35

2.7.2 Video 35

2.7.3 Mapnik 35

2.7.4 OpenCV 36

2.8 Caching 36

2.8.1 Storing downloaded data 36

2.8.2 Caching resampling calculations 36

2.8.3 Creating products from other products 37

3 Design and construction of software 38

3.1 File structure 39

3.1.1 Files 39

3.1.2 Data 40

3.1.3 Data types 40

3.1.4 Domains 41

3.1.5 Helpers 41

3.2 CLI 42

3.2.1 Products 43

3.2.2 Resolution 43

3.2.3 Utctime 43

3.2.4 Hours 43

3.2.5 Iph 43

3.2.6 Fps 44

3.3 Product creation 44

3.3.1 Data image 46

3.3.2 Visual image 47

3.3.3 Video 49

3.4 Obtaining satellite data 52

3.4.1 Satellite collection 52

3.4.2 Satellite enum 54

3.4.3 Satellite type 55

3.4.4 Downloaders 55

3.5 Processing 63

3.5.1 Loading 64

3.5.2 Resampling 65

3.5.3 Combining 69

3.5.4 Cloud extracting 84

4 Results and discussion **87**

4.1 Image products 87
4.1.1 Cloud accuracy 89
4.1.2 Combining 90
4.1.3 GOES-17 noise 91
4.2 Video product 92
4.2.1 Cloud movements 93
4.2.2 Day night cycle 93
4.2.3 Consistent frame change 93
4.3 Resource usage 94
4.3.1 Benchmark 95

5 Economic overview 98

5.1 Direct expenses 98
5.2 Indirect expenses 99
5.3 Caching 99

6 Environmental accounting 100

6.1 Emissions 100
6.2 Benefits 101

7 Conclusion 102

Bibliography **103**

A Email from The Norwegian Meteorological Institute **106**

B External links **108**

B.1 Source code 108

B.2 Video product demonstration 108

C How to use **109**

Glossary

AWS Amazon Web Services. 19, 20, 55–61

band Wavelength portion, used for satellite imagery. 8, 10, 11, 17, 19, 20, 23, 24, 33, 46, 55, 58, 59, 61, 65, 66, 69, 71, 96

EQC Equidistant Cylindrical. 22–24, 28, 66, 67

ESA European Space Agency. 8, 10, 15, 19, 23, 55, 60

EUMETSAT European Organisation for the Exploitation of Meteorological Satellites. 8, 15, 19, 20, 23, 31, 32, 34, 40, 55, 60, 61

geostationary Equatorial orbit at an altitude where satellite's speed is the same as the earth's. 6, 8, 9, 14, 20, 21, 25, 33, 69, 106

GeoTIFF Image format that contains geographic data. 35, 37, 46, 86, 106

GOES-16 A geostationary satellite. 8, 16, 17, 19, 20, 55, 58, 92

GOES-17 A geostationary satellite. 8, 16, 19, 20, 55, 58, 91, 92

Himawari 8 A geostationary satellite. 8, 10–12, 16, 19, 20, 55, 58, 59, 92

HSD File format used for storing satellite data. 23

JMA Japan Meteorological Agency. 8, 15, 23, 55, 56, 59

latitude Geographic axis used to specify positions north or south. 21, 22, 48, 70, 75–77, 80–82

longitude Geographic axis used to specify positions east or west. 6, 8–11, 14, 21, 22, 28–30, 48, 67, 69, 70, 72–75, 79–82

LWIR Long-wavelength infrared. 18

Mapnik Software used for creating image of the earth. 35

Meteosat 11 A geostationary satellite. 8, 10, 11, 16, 19, 60

Meteosat 8 A geostationary satellite. 8, 10–13, 16, 19, 60

MWIR Mid-wavelength infrared. 17, 18

NASA National Aeronautics and Space Administration. 8, 15, 19, 23, 55

native format File format used for storing satellite data. 23

netCDF File format used for storing satellite data. 23, 106

NIR Short-wavelength infrared. 17, 18, 31

NOAA National Oceanic and Atmospheric Administration. 6–8, 15, 16, 19, 23, 55

NWCSAF Software for generating cloud data from satellite imagery. 33, 106, 107

OpenCV Open-source library that includes multiple computer vision algorithms. 36, 51

opencv-python Python module for OpenCV. 36, 48, 51

projection Method used for showing three-dimensional objects in the two-dimensional plane. 21–25, 28, 65

Satpy Python library for handling satellite data. 3, 4, 23–25, 27, 28, 33, 36, 54, 56, 64, 65, 69, 72, 83, 106

Scene Satpy object that handle satellite data. 3, 24, 54, 64–70, 72, 83, 84

SWIR Short-wavelength infrared. 17, 18, 31

thermal infrared light The infrared light with lower frequency. 18

Glossary

TIR Thermal infrared. 17

VIS Visible light. 17

zenith angle Vertical angle of an object from straight above a position on earth. 6, 31

Summary

The goal of this project is to create visualizations of world-wide cloud coverage, using satellite imagery. Visualizations created are being stored in various formats, including images with raw cloud data, images with background of the earth, and videos showing the motion of the clouds. The assignment is given by Time and Date AS, which is the company behind the world's top ranked website for time and time zones: timeanddate.com.

The visualizations of the world-wide cloud coverage are being created by executing multiple consecutive steps. First satellite imagery is downloaded from multiple sources. The imagery is then being resampled into having similar attributes. With the resampled imagery, a combined image is being made, which covers the whole globe. Using the intensity of different frequencies of light, clouds are being detected and extracted, and then used to create the final visualizations.

Due to the complexity of cloud detection, a simplified solution is being used. This leads to reduced accuracy, as only some kinds of clouds are detected. With further development, the cloud detection could be improved and thereby increase the amount of clouds being detected.

Chapter 1

Introduction

1.1 Problem

Multiple governmental entities throughout the world provide publicly available images captured by weather satellites at regular intervals. Sourcing these images, processing them, and storing them in a accessible formats allows querying for cloud coverage for specific locations on Earth. The goal of this project is to be able to render visualisations of world-wide cloud coverage by combining and normalising partial satellite imagery provided by different sources.

The produced world-wide visualisations should contain the most recent satellite data available. It is therefore critical that the final program's execution time is kept low.

Static images at specified times should be made to visualize the global cloud coverage. By combining these images, time-lapses should be made to visualize the movement of the clouds.

This thesis is going to be a proof of concept, where the discovery of problems and possibilities is in focus. Discoveries made in this thesis will later be used as a foundation for the final program. Because this is a proof of concept, the execution time is not as critical as it will be in the final program.

1.2 About the Company

Time and Date AS is a company based just outside Stavanger, Norway. The company operates timeanddate.com, which is the world's top ranked website for time and time zones. More than a million users access the website every day. In addition to time and time zones, timeanddate.com also provide services within weather, astronomy, calculators and much more. [30]

1.3 Structure

This thesis starts off introducing the theory and decisions made in chapter 2. The chapter includes assessments and explanations of why decisions where made, as well as calculations supporting the different conclusions. Central concepts that are used in this project is also explained in this chapter. A lot of the work in this chapter is based on previous work from multiple sources.

The design and construction of software is presented in chapter 3. This chapter shows how the theory explained in chapter 2 is implemented, as well as presenting design choices and algorithms.

Chapter 4 presents the results, and discusses what can be done to further improve this project. This is followed by an economic overview in chapter 5, and environmental accounting in chapter 6, which discusses the economic and environmental consequences of the project's result. Lastly, a short conclusion is presented in chapter 7; following with the bibliography and attachments.

1.4 Technologies

1.4.1 Programming language

One of the programming languages considered for this project is Python. Python benefits from the great amount of packages available for handling satellite data, such as Satpy (see subsection 1.4.2 below). This reduces the complexity and development time of the project. Speed is a concern though when using Python, as it is a relatively slow language. Many of the libraries that perform the heavy computations in this project reduce this concern, as they use underlying libraries that are implemented in faster languages like C.

Rust is another programming language that could be used in this project. This is the language that is primarily used by Time and Date's backend, and therefore easily integrates with the rest of the backend. Rust is well suited for programs that requires low runtime. Compiling directly to machine code, and the memory management system is some of many features that reduces Rust's runtime. One of the disadvantages with Rust, compared to Python, is that it has no library that handles satellite data as well as Satpy does. The development time when using Rust will also be greater, due to the fact that Rust is a low-level programming language.

Due to the fact that this project is a proof of concept, discovering possibilities and potential problems is a greater focus than runtime. This leads to Python being the natural choice when choosing programming language, because of its low development time.

1.4.2 Satpy

A central technology in this project is Satpy. Satpy is a Python library for reading, manipulating, and writing data from satellites. Satpy reduces the concern of different file formats, as multiple file readers are implemented. This allows reading satellite data from many different file formats into Python objects called Scene. The satellite data in Scene objects can be altered in multiple ways; including changing projection, image resolution and much more. Combinations of satellite frequencies can also be made easily with Satpy. This is useful when extracting necessary data, based on the differences between the different satellite frequencies (see 2.2 at page 17). [17]

Chapter 2

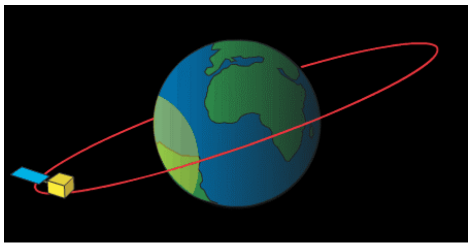
Theory

2.1 Satellites

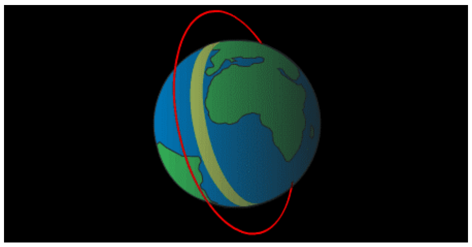
Satellite imagery is used in this project to gather information about the cloud coverage. To cover the whole earth, multiple satellites are needed. The satellites utilized need to be designed for weather monitoring, so the cloud cover can be extracted from the satellite images. Because of these statements, it is important to choose the right satellites for this project's purpose.

2.1.1 Weather satellites

There are multiple satellites circling the earth in many different orbits. The orbit is chosen based on the satellites' intended applications. Figure 2.1 below contains two illustrations of orbits used by satellites monitoring the earth's weather:



(a) Geostationary orbit



(b) Polar orbit

Figure 2.1: Illustration of two different orbits obtained from NOAA [18]. The red circles represents the orbit, while the yellow area on the earth represents the satellite's coverage.

Satellites in geostationary orbits are used in this project. Geostationary satellites circle the earth at the exact same rate as the earth rotates. An illustration of a geostationary orbit is shown above in figure 2.1a. This makes the satellites stay above one particular location on earth. Because of this feature, geostationary satellites are useful for obtaining weather data, and analysing weather changes. [10]

One problem with geostationary satellites is the lack of the ability of monitoring the poles. Physics makes it only possible to have geostationary satellites above the equator. Due to the angle of the satellites, they do only cover from 81.3° south to 81.3° north [28], which decreases at longitudes that differs from the satellite's. The resolution close to these latitudes is also pretty low, as the satellite's zenith angle is high. This problem is solved by using polar orbiting satellites to monitor the weather at the poles. An example of a polar orbit is illustrated in figure 2.1b above. The weather data from the polar orbiting satellites is updated relatively rarely compared to the geostationary weather satellites, due to their orbit [18]. Because of this and the fact that there will be few queries on the weather at the poles, this project will exclude the poles.

2.1.2 Earth coverage

As the geostationary satellites only cover one part of the earth, multiple satellites must be used. Because of earth's curvature, the resolution of the satellite images will be lower near the edges. Figure 2.2 below show this phenomena, where the resolution is defined as square meters per pixel:

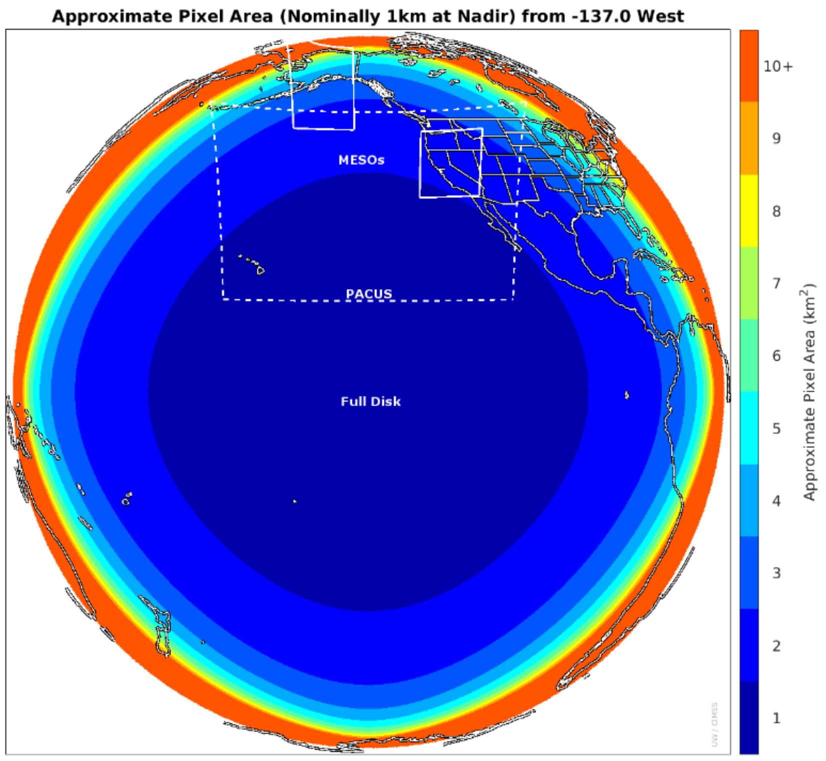

Approximate Pixel Area (Nominally 1km at Nadir) from -137.0 West
MESOs
PACUS
Full Disk
10+
9
8
7
6
5
4
3
2
1
Approximate Pixel Area (km^2)
UW / CIMSS

Figure 2.2: Illustration from NOAA showing the falling resolution of satellite images near the edge [20]

.

As a consequence of the lower resolution near the edge of the satellite images (shown in figure 2.2 above), it's beneficial to utilize other satellites for these areas. The more satellites being used, the higher the mean resolution will be. A downside of using more satellites, is that many of the satellite images have to be downloaded from different places. There is also some differences between the data the satellites provide. These differences includes file format, resolution and other image attributes. Because of these facts, each satellite implemented adds complexity and development time. The number of satellites is therefore kept at a reasonable level. It is beneficial to use satellites from the same operators, as the operators often use the same solutions for their satellites . The satellites utilized in this project are shown in table 2.1 below:

Name	Longitude	Operator	Bands	Lowest band resolution
GOES-16	$75.2°W$	NASA/NOAA	16	2000m
GOES-17	$137.2°W$	NASA/NOAA	16	2000m
Himawari 8	$140.7°E$	JMA	16	2000m
Meteosat 8	$41.5°E$	EUMETSAT/ESA	12	3000m
Meteosat 11	$0°$	EUMETSAT/ESA	12	3000m

Table 2.1: Overview over the satellites utilized in this project. Data for GOES-16 and GOES-17 is obtained from NOAA [20]. Continuously the data for the Himawari 8 satellite is obtained from JMA [22], while the EUMETSAT provided the data for Meteosat 8 and Meteosat 11 [15]. Information about the bands is received from the University of Twente. [23]

The chosen satellites shown in table 2.1 above are spread around the globe at different longitudes. These satellites are chosen as a starting point, because they are all somewhat similar. They do all have 12 or 16 bands (ranges of frequencies used for taking images), with pretty similar resolutions. There are also similarities between the operators, which leads to lower complexity. By drawing the tangents from each satellite to the earth, it can be shown if all places at equator is reached with the chosen satellites. To illustrate this, both earth's radius and the satellites' orbit altitude is needed.

Earth's radius is gotten from NASA Planetary Science Division [9]:

$$earth_{radius} = 6371km \tag{2.1}$$

Altitude of geostationary satellites is gotten from ESA [10]:

$$satellite_{altitude} = 35786km \tag{2.2}$$

An illustration of the satellites from table 2.1 above with the correct measurements of the earth and the satellites is shown in the figure below 2.3:

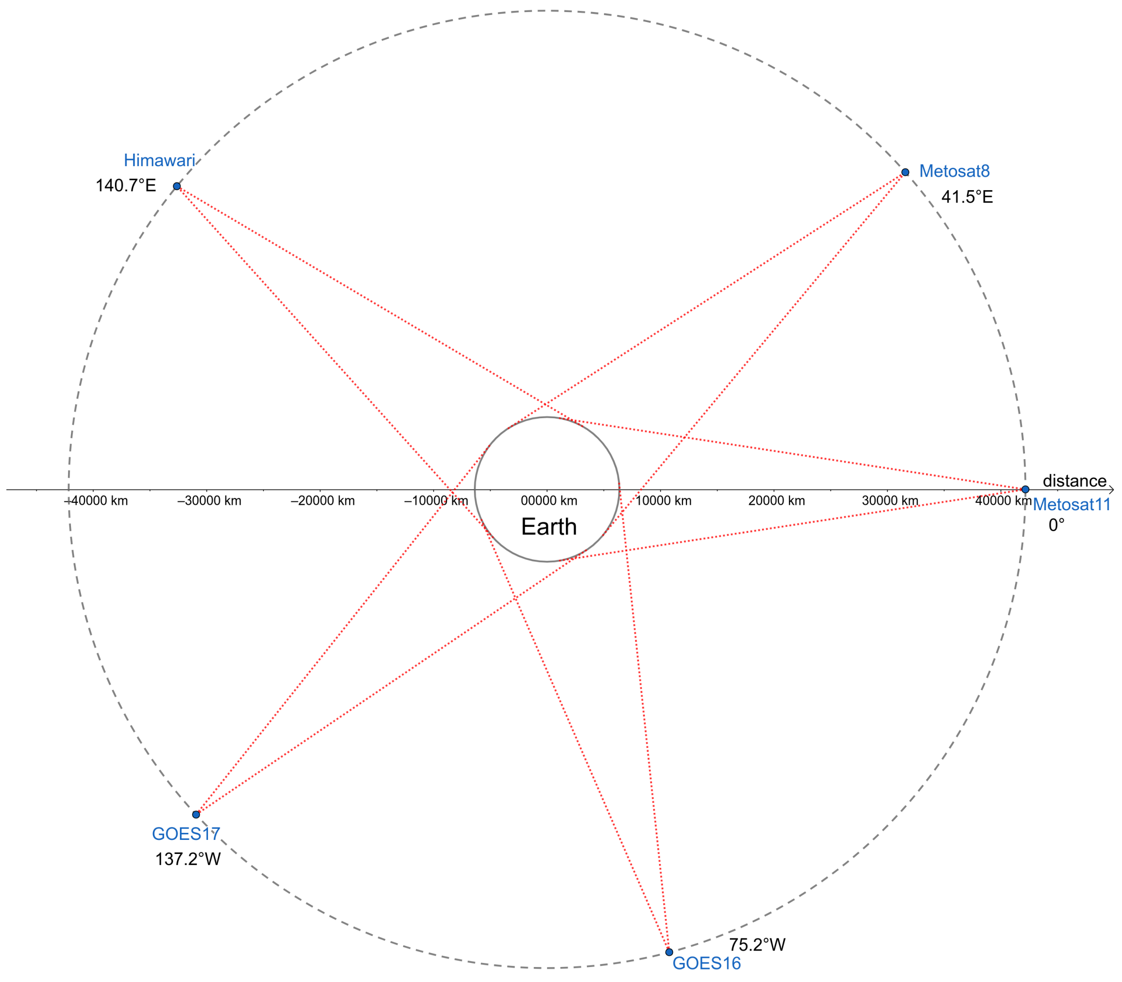


Figure 2.3: Illustration made with GeoGebra showing earth with the chosen satellites. The inner circle represents earth, while the outer represents the geostationary orbit. The red dotted lines shows the outermost view of earth for each satellite.

Figure 2.3 above reveal that every point on earth is covered using the five chosen satellites. As illustrated in figure 2.2 at page 7 the resolution at the outermost view each satellite have of the earth is greatly reduced. By calculating the biggest longitude difference between two adjacent satellites, the point on earth with the lowest resolution can be found. The biggest longitude difference is calculated in the equations below:

$$A = |metosat8_{long} - metosat11_{long}| = |41.5°E - 0°| = 41.5° \tag{2.3}$$

$$B = |himawari_{long} - metosat8_{long}| = |140.7°E - 41.5°E| = 99.2° \tag{2.4}$$

$$\begin{aligned} C &= |goes17_{long} - himawari_{long}| = |137.2°W - 140.7°E| \\ &= 222.8°E - 140.7°E = 82.1° \end{aligned} \tag{2.5}$$

$$D = |goes16_{long} - goes17_{long}| = |75.2°W - 137.2°W| = 62.0° \tag{2.6}$$

$$E = |metosat11_{long} - goes16_{long}| = |0° - 75.2°W| = 75.2° \tag{2.7}$$

By inspecting the equations above (2.3 - 2.7), it's clear that B in equation 2.4 has the biggest angle. This is the longitude difference between the Himawari 8 and Meteosat 8 satellites. In contradiction to this, the difference between Meteosat 8 and Meteosat 11 is the smallest. This means that it would be beneficial if the Meteosat 8 satellite where exchanged with another satellite that is closer to the Himawari 8 satellite. The optimal placement of this satellite is calculated below:

$$\begin{aligned} optimal\_longitude &= (meteosat11_{long} + himawari_{long})/2 \\ &= (0 + 140.7°E)/2 \\ &= 70.35°E \end{aligned} \tag{2.8}$$

Some satellites closer to $70.35°E$ (from equation 2.8) than Meteosat 8 at $41.5°E$ is listed below:

Name	**Longitude**	**Operator**	**Bands**
INSAT-3D	$82°E$	ISRO	6
Electro-L N3	$76°E$	RosHydroMet/Roscosmos	10
FY-2H	$79°E$	China	5

Table 2.2: Overview over satellites considered in this project, that was not used. Data for INSAT-3D is received from ESA [11]. The World Meteorological Organization [24] provided the data for Electro-L N3. Lastly, the data for FY-2H was received from NSMC [6]. Information about the bands is received from the University of Twente. [23]

The satellites in table 2.2 above are all located in a longitude closer to the center of Himawari 8 and Meteosat 11 than Meteosat 8. There are some disadvantages with them though, compared to Meteosat 8. All of the satellites listed has less bands than Meteosat 8. This leads to less data being available for cloud recognition, and other potential products.

Electro-L N3 suffers the least from having a lower amount of bands, with its 10 bands (compared 12 bands of Meteosat 8). This satellite is a Russian satellite, which further leads to ethical issues, due to the recent invasion (at the time of writing) of Ukraine. Insecurity of data availability and quality is a concern due to potential sanctions that may occur.

An important benefit with the Meteosat 8 satellite, is that it is similar to the Meteosat 11 satellite. As explained in page 7 this reduces both complexity and development time of the program. Due to these facts, the Meteosat 8 satellite is used, even though there exists satellites placed at better longitudes.

2.1.3 Resolution verification

Table 2.1 at page 8 shows that the Meteosat satellites has 3000 metres as the lowest resolution (at a perfect angle). This means that the part where the Meteosat 8 satellite overlaps the Himawari 8 coverage is the covered location that has the lowest resolution. The longitude with the lowest resolution is calculated below:

$$\begin{aligned} worst\_longitude &= (meteosat8_{long} + himawari_{long})/2 \\ &= (41.5 + 140.7°E)/2 \\ &= 91.1°E \end{aligned} \tag{2.9}$$

It is beneficial to calculate the resolution difference at the worst point, and verify that it is good enough. A comparison between the resolution at a perfect angle versus the worst angle at equator is illustrated in figure 2.4 below:

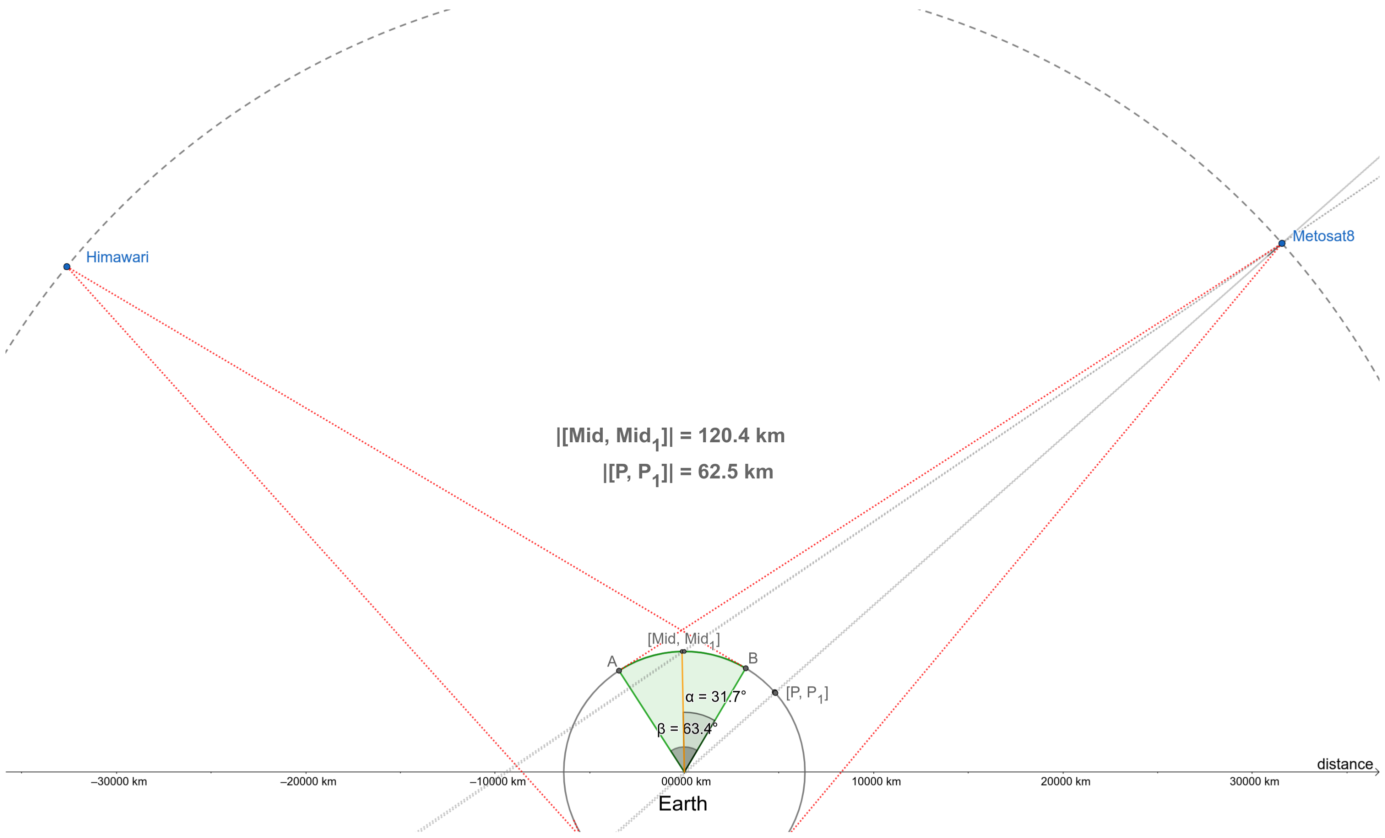


Figure 2.4: Comparison made with GeoGebra of the resolution at a perfect angle versus resolution at the worst point. Both measurement points is made of a vector with two points that is 0.1° apart from each other, from Meteosat 8's perspective. $[P, P_1]$ is the vector from a perfect angle, while $[Mid, Mid_1]$ is the vector at the worst point. The illustration also shows the length of these vectors.

In figure 2.4 above there exists two named vectors. These are named $[Mid, Mid_1]$ and $[P, P_1]$, and represents Meteosat 8's accuracy at the given location. $[Mid, Mid_1]$ is the accuracy at the worst location, which is the location in the middle of the edge of Meteosat 8 and the edge of Himawari 8. These locations are marked as A and B in figure 2.4, respectively. $[P, P_1]$ is the accuracy at the best location. With the use of GeoGebra, the lengths of the two named vectors can be calculated and displayed. These are displayed in the middle of figure 2.4 above. The following values are observed:

$$|[Mid, Mid_1]| = 120.4km \tag{2.10}$$
$$|[P, P_1]| = 62.5km \tag{2.11}$$

With the values from equation 2.10 and 2.11, the drop in accuracy (AD_{PMid}) from $[P, P_1]$ to $[Mid, Mid_1]$ can be calculated:

$$\begin{aligned} AD_{PMid} &= |[P, P_1]| \ / \ |[Mid, Mid_1]| \\ &= 62.5km/120.4km \\ &= 0.5191 \\ &= 51.91\% \end{aligned} \tag{2.12}$$

Equation 2.12 above shows that the accuracy of $[Mid, Mid_1]$ is 51.91% the accuracy of $[P, P_1]$. Dividing the resolution of Meteosat 8 with AD_{PMid} gives the resolution at the location with the worst accuracy:

$$\begin{aligned} Mid_{resolution} &= P_{resolution} \ / \ AD_{PMid} \\ &= 3000m/0.5191 \\ &= 5779m \end{aligned} \tag{2.13}$$

As calculated above in equation 2.13, the worst resolution at equator with the satellites utilized is $5779m$. The consideration of this accuracy being good enough depends on the definition of which clouds are considered above a specific coordinate. Clouds from 30° degrees vertically, relative to the horizontal plane at any location, is in this case defined as clouds above that location. Figure 2.5 below uses the distance from equation 2.13 above, to show how low the measured clouds can be, before the measurement gets too inaccurate:

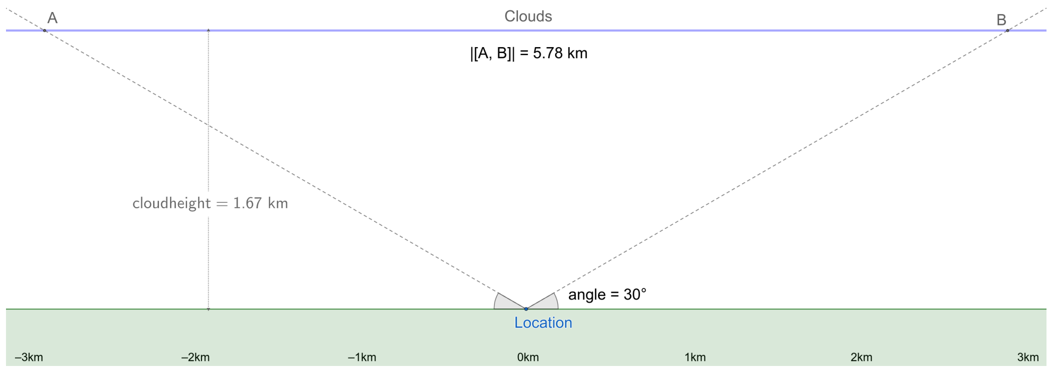


Figure 2.5: Illustration made with GeoGebra calculating the lowest cloud height that is accurate with 5.78km resolution and an angle of 30° degrees.

By inspecting figure 2.5 above, it is noticeable that the lowest cloud height at the location with the worst resolution at equator is $1.67km$. This cloud height represents the top of the cloud. As most cloud tops has an altitude higher than $1.67km$, this resolution is considered acceptable.

The calculations above only apply to the equator, as the satellites' resolution decreases closer to the poles. As mentioned earlier (see page 2.1.1) the geostationary satellites utilized in this project doesn't cover the poles at all. Because of this the locations above 81.3° north and below 81.3° south are not covered at all. Latitudes close to these values will as a side effect have lower resolution, due to the falling resolution close to the satellite images' edges (illustrated in figure 2.2). A more practical upper limit is 75° north and south, according to Planetary [25]. This applies to the points with longitude equal to a satellite's. To keep the upper limit good enough for every longitude, a limit of 70° north and south is being used in this project. By using this limit, the latitudes with low resolution close to the poles is removed.

By using the upper limit of 70° north and south from last paragraph, with the worst longitude of $91.1°E$ from equation 2.9 at page 11, the points with the worst resolution can be found. These points are $70.0°N, 91.1°E$ and $70.0°S, 91.1°E$. These points are located respectively in the northern Siberia and Antarctica. Both of these point are in sparsely populated regions, where the resolution is not that important. These regions will therefore have good enough resolution.

2.1.4 Scanning intervals

The satellites takes images of the earth at different intervals. The intervals are often the same on satellites by the same operators. The table below shows the scanning intervals that the chosen satellites' operators use:

Operator	**Scanning interval (minute)**
EUMETSAT/ESA	15
NASA/NOAA	15
JMA	10

Table 2.3: Table showing what scanning intervals operators use for the chosen satellites.

Table 2.3 above shows that two operators use scanning intervals of 15 minutes, while one scans every 10 minute. To find how often the program needs to be run to always have the latest data, the union of the scanning minutes in the hour of both scanning intervals must be found:

$$\begin{aligned} scanning\_minutes &= A \cup B \\ &= \{a \in 10\mathbb{N} : a < 60\} \cup \{b \in 15\mathbb{N} : b < 60\} \\ &= \{0, 10, 20, 30, 40, 50\} \cup \{0, 15, 30, 45\} \\ &= \{0, 10, 15, 20, 30, 40, 45, 50\} \end{aligned} \tag{2.14}$$

As seen in equation 2.14 above, the steps between the scanning minutes is either 5 or 10 minutes. By using this information, it is known when the program should be run, to keep the data updated at any time.

As the JMA operator with one satellite utilized in this project, has a scanning interval of 10 minutes, it is not that important if the program is not ran every time it is scanning. It could be considered to only use the scanning times for the 15 minute intervals, which reduces executions per hour from 8 ($|scanning\_minutes|$ from equation 2.14) to 4 ($|B|$ from equation 2.14). By doing this, the program's products will contain updated data for 4 out of 5 satellites at all times, while halving the number of executions.

2.1.5 Future replacements

Because of different factors, satellites are replaced at regular intervals. The satellites chosen in subsection 2.1 above are satellites that are operational at the time of writing. The end of life of the satellites utilized in this project, as well as planned replacements are listed in the table below:

Utilized satellite	**End of life**	**Replacement satellite**
GOES-16	$\geq$ 2027	*NA*
GOES-17	$\geq$ 2029	GOES-18*
Himawari 8	$\geq$ 2030	*NA*
Meteosat 8	Nov 2022	Meteosat 9
Meteosat 11	$\geq$ 2033	*NA*

**GOES-18 is sent out to replace GOES-17, because of malfunctions. The replacement is planned to happen in early 2023 [21].*

Table 2.4: Table of utilized satellites with their end of life, as well as planned replacements. Data is provided by The World Meteorological Organization [24].

As shown in table 2.4 above, satellites are replaced at regular intervals. The Meteosat 8 satellite is for example replaced with Meteosat 9 in November 2022, which is months after the time this project carried out. Even though the Meteosat 8 satellite is replaced in November 2022, its still being used in standby. This makes it less crucial, as the satellite still can be used a while after the replacement. Because of the frequent replacements, it is important that a satellite easily can be replaced in the program, without adding a lot of code.

As mentioned in the footnote of table 2.4, GOES-17 suffers from a malfunction. According to NOAA the malfunction is in the cooling system, which leads to the sensor providing imagery, not working optimally at high temperatures. The problem is occurring during the warm season. The side effect from this problem, is that some of the infrared imagery has reduced operating time during the warm season. The solution to this problem is the replacing GOES-17 with a new satellite named GOES-18. NOAA says that GOES-18 will be operational early 2023, at the location of GOES-17. [21]

2.2 Satellite frequency bands

Satellites used for monitoring the weather on earth take images using multiple frequency bands (ranges of frequencies). The frequencies used range from visible to infrared light. An example of a satellite's frequency bands is shown in the table below:

Band	**Type**	**Wavelength (μm)**	**Resolution (m)**
Band 1	VIS	0.45 to 0.49	1000
Band 2	VIS	0.59 to 0.69	500
Band 3	NIR	0.846 to 0.885	1000
Band 4	SWIR	1.371 to 1.386	2000
Band 5	SWIR	1.58 to 1.64	1000
Band 6	SWIR	2.225 to 2.275	2000
Band 7	MWIR	3.8 to 4	2000
Band 8	TIR	5.77 to 6.6	2000
Band 9	TIR	6.75 to 7.15	2000
Band 10	TIR	7.24 to 7.44	2000
Band 11	TIR	8.3 to 8.7	2000
Band 12	TIR	9.42 to 9.8	2000
Band 13	TIR	10.1 to 10.6	2000
Band 14	TIR	10.8 to 11.6	2000
Band 15	TIR	11.8 to 12.8	2000
Band 16	TIR	13 to 13.6	2000

Table 2.5: Table of the bands GOES-16 has. Received from the University of Twente. [23]

The information in table 2.5 shows the frequency bands of the GOES-16 satellite. All the satellites utilized in this project, have similar frequency bands, with some variations in wavelengths and resolutions. Each band represents an image taken from the given satellite, capturing light in the specified wavelength range. As shown in table 2.5 these frequencies ranges from Visible light (VIS) to Thermal infrared (TIR). An illustration of the light spectrum is shown the figure below:

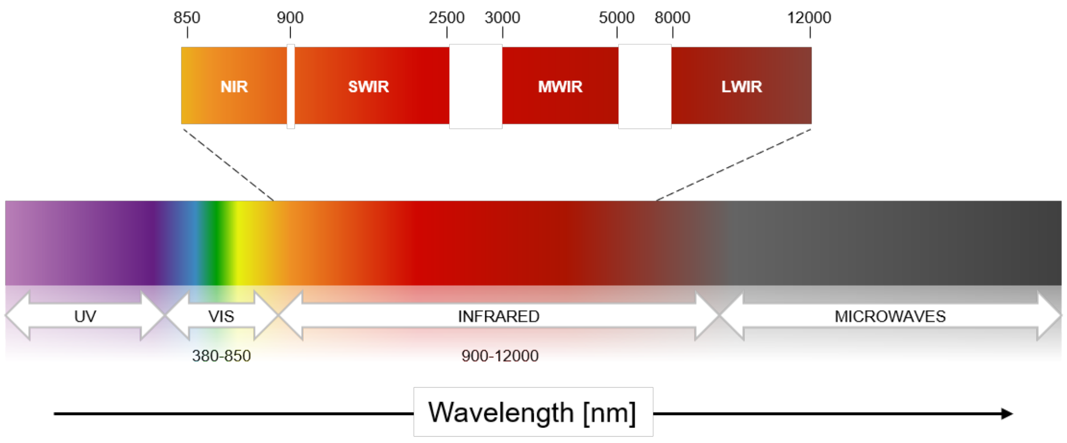


Figure 2.6: Illustration that shows frequency ranges of different categories of light. Obtained from Sunex [29].

As illustration in figure 2.6 above shows, the light spectrum is divided into multiple categories. The categories used by the satellites in this project is both visible and infrared light. Visible light is the light that is visible to the human eye. Infrared light on the other hand is light that has a larger wavelength, which the human eye cannot see.

The infrared light is divided into multiple categories: Short-wavelength infrared (NIR), Short-wavelength infrared (SWIR), Mid-wavelength infrared (MWIR) and Long-wavelength infrared (LWIR). These categories are again put into two sectors: reflected and thermal infrared. [29]

Reflected infrared light is photons reflected by an object. Both NIR and SWIR is reflected infrared light. This leads to lower activity at night, as the sun is not acting as a light source then. [29]

Thermal infrared light is according to Sunex all light ranging from MWIR to LWIR. This light is often light that is emitted from a object, such as thermal energy. As thermal energy is stored in objects over time, thermal infrared light is not affected as much by the time of the day, as reflected infrared light is. [29]

Using the information obtained in this subsection, it is clear that thermal infrared light is more convenient than reflected infrared light when working with global satellite imagery, as it is affected little by the time of the day. Each frequency has different use cases though. Useful information can be extracted by filtering specific brightness values of both single and combined frequency sets. How the clouds are detected is explained in greater detail in section 2.6.

2.3 Obtaining satellite data

As seen in table 2.1 at page 8, the satellites are managed by three different operators. Satellites managed by the same operator can usually be implemented using the same code. This is because operators typically put the image data from satellites on the same site as the other satellites they manage. Other factors like file format and resolution is also often standardised for the satellites managed by an operator. Multiple steps must be carried out to obtain the necessary satellite data. The goal is to download the most recent data with the highest available resolution. It is beneficial if the data is separated into multiple files, so only the files needed can be downloaded. An example of this is satellite images separated based on frequency bands (explained in section 2.2 at page 17). Not every frequency band is needed, which makes it superfluous to download them all.

2.3.1 Online resources

The two satellites operated by NASA/NOAA (GOES-16 and GOES-17) and the Himawari 8 satellite have their data available on Amazon Web Services (AWS). According to Amazon, AWS should provide the most recent data that is available for these satellites, for free [2][3]. An advantage of receiving the data from these three satellites out of the same resource, is that a lot of the code can be used for all the satellites, which reduces the time spent developing, as well as the complexity of the code. This leads to the application being less prone to bugs.

Satisfying the reasoning above, the two satellites operated by EUMETSAT/ESA (Meteosat 8 and Meteosat 11) are also downloaded from the same resource. Both of these satellites' data is obtained directly from EUMETSAT, as it is not available on AWS. The most recent data available can be obtained without any cost [14]. The only requirement is that a user is registered on EUMETSAT's Earth Observation Portal.

2.3.2 Separation of data

Great separation of data at the online resource where it is received from is beneficial to reduce the amount of data to be downloaded. This is important to reduce download time. Download time is a concern, resulting from the great amount of data needed for the high resolution images, that is gathered from every satellites.

As mentioned earlier in the start of this section at page 19, not all frequency bands are necessary when retrieving cloud data. The data from GOES-16, GOES-17 and Himawari 8 is separated into multiple files, based on the frequency bands. By only downloading the frequency bands needed, the amount of data to be downloaded is greatly reduced.

Data from the Meteosat satellites is not separated in any way (except by time). This results in superfluity when downloading the data, as even the irrelevant frequency bands are downloaded. As there is no other resource available to download this data from directly, there is no way to get past this problem.

2.4 Projections

Satellite imagery from geostationary satellites is images taken of the earth, where the earth is represented as a two-dimensional sphere. This representation of earth is called Geostationary Satellite View. A figure visualizing the Geostationary Satellite View is shown below:

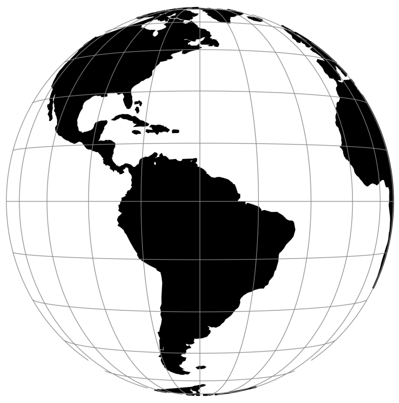


Figure 2.7: An illustration of Geostationary Satellite View, obtained from the PROJ documentation [26].

Figure 2.7 above shows how the images retrieved from the satellites look like. This is called a projection of the earth, as it is a method used for showing the three-dimensional earth in the two-dimensional plane. According to GISGeography this is exactly what a map projection does [16]. A projection is generally a representation of the earth in the two-dimensional plane.

2.4.1 Projection selection

The produced products in this project needs to be represented in a convenient way. As the goal of this project is to show the world-wide cloud coverage, a projection that shows the entire earth is needed. It is also beneficial if the projection is ordered by latitude and longitude. This is because it makes it simpler to work with programmatically. Cylindrical projections are a group of projections that shows the entire earth, while having the latitude and longitude ordered. The projection used to represent the final result in this project will therefore be a cylindrical projection. A visualization of a cylindrical projection is shown below:

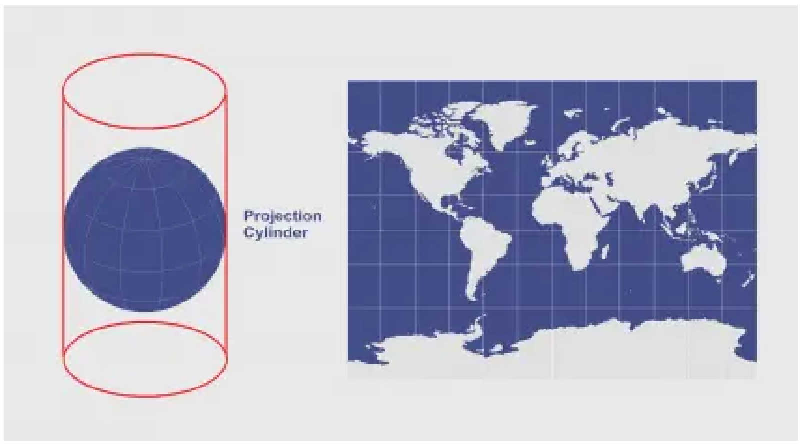


Figure 2.8: Cylindrical projection illustration by GISGeography [16].

A cylindrical projection is shown above in figure 2.8, where the spherical earth is projected onto the cylinder as shown to the left. This results in the projection shown to the right. Unlike the resulting projection shown in figure 2.8 above, the projection utilized should have linear latitude changes. This makes it simpler to find the latitudes mathematically. A cylindrical projection that has linear latitude changes is shown below:

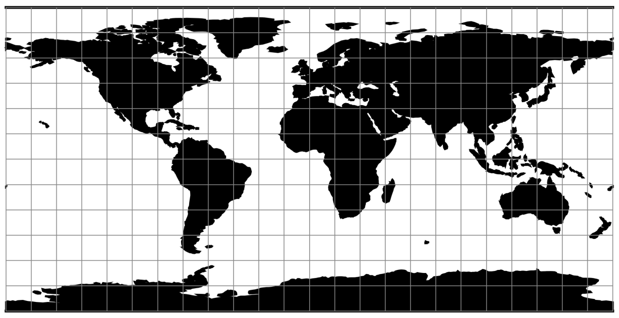


Figure 2.9: Equidistant Cylindrical projection received from the PROJ documentation [26]

The Equidistant Cylindrical (EQC) projection is shown above in figure 2.9. This is a cylindrical projection that has linear latitude changes. In addition to this, the latitude and longitude scale is equal, which makes it act like a grid, where indexes is easily convertible to coordinates. This fact simplifies calculations, and therefore the programmatic implementation. Because of these features, Equidistant Cylindrical projection is chosen as the projection for the final products.

2.5 Creating world map

The creation of the world map from the downloaded data is central in this project. This includes both resampling individual satellite images (see subsection 2.5.2), as well as combining the images into an image of the entire earth (see subsection 2.5.3). As explained in section 2.2, starting at page 17, not all frequency bands are needed. Both the resampling and combination phase are performed for every frequency band utilized, as a world map is needed for every frequency range that is used.

2.5.1 Reading satellite data

Before the satellite data can be used, the data must be read. A lot of code is needed to perform this task, as the satellite data is provided in different file formats. The file format is chosen by the operator, which means that it is often similar for satellites that have the same operator. A table showing the file format used by each operator is listed below:

Operator	**File type**
EUMETSAT/ESA	native format [13]
NASA/NOAA	netCDF [1]
JMA	HSD (bz2 compressed) [22]

Table 2.6: Table showing what file type operators use for their satellites.

The file types in table 2.6 above do all need separate readers. Reading the satellite data in this project is straightforward thanks to the python package Satpy (see section 1.4.2). Satpy provides multiple readers, where all the file types from table 2.6 is supported. The readers read the satellite data into Satpy objects called Scene, which act the same independent of which reader was used.

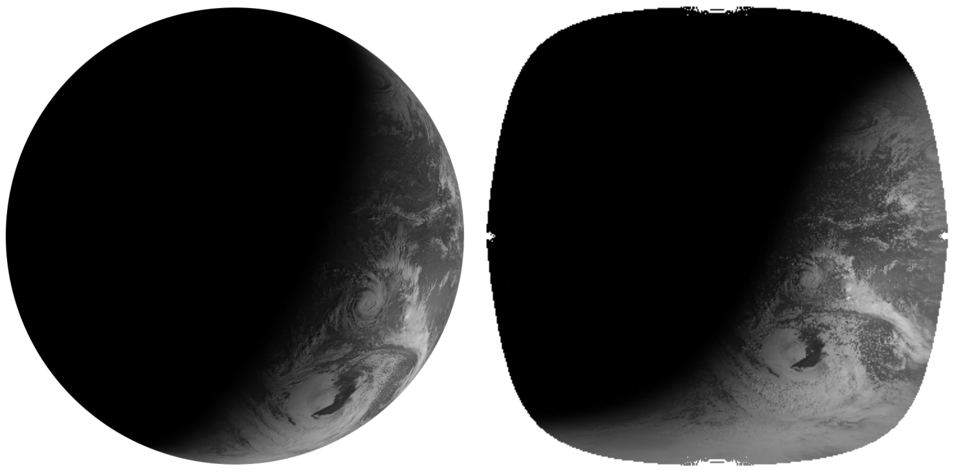
Another benefit with Satpy's Scene object is that it does not load the frequency bands before it is told to. Which frequency bands being loaded can also be chosen. These facts reduce unnecessary data loading, as only the needed bands are the ones that are loaded.

2.5.2 Resampling

Before the world map can be generated, images from each satellite must be resampled. This will alter the images so they are having the same resolution and projection as the final world map. As concluded at page 23, Equidistant Cylindrical projection will be used. When all the satellite images are resampled, the process of combining them into a world map will be a lot simpler. The combining process is explained in detail in subsection 2.5.3 below.

Resampling is an advanced process that depends on multiple calculations that needs to be accurate. As this process is as advanced as it is, potential debugging is also harder. Resampling is straightforward in this project, because of the usage of Satpy [27]. With few parameters, Satpy is able to resample the satellite data into a wide range of projections at any resolution. This also reduces the concern of the hard debugging that follows with mathematical errors.

Below is an illustration showing the satellite image before and after the resampling:



(a) Before

(b) After

Figure 2.10: Before and after resampling. Resampled from geostationary to projection.

Choosing resample algorithm

One of the necessary parameters for Satpy resampling is the resampling algorithm that is going to be used. Satpy provides multiple resampling algorithms that each have their own benefits and disadvantages. Information about these algorithms is documented in the Satpy documentation [17]. Some of the algorithms considered for this project are listed below:

- Nearest
- Bucket Average
- Bilinear

The *nearest* algorithm is the simplest of the ones listed above. An illustration of how it works is shown below:

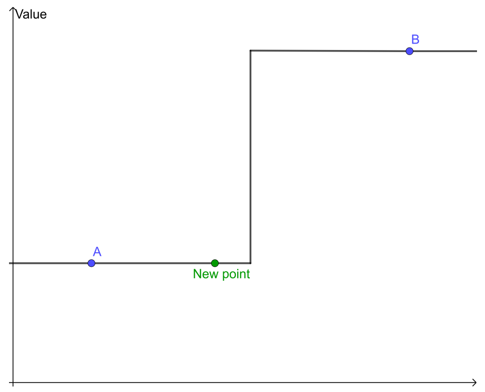


Figure 2.11: 1 dimensional nearest neighbour sampling. The green point shows the new value, made from point A and B.

The illustration in figure 2.11 above shows how the *nearest* algorithm works. The new point's value is found from the value of the point that is nearest to the new point.

Another algorithm considered is the *bucket average* algorithm:

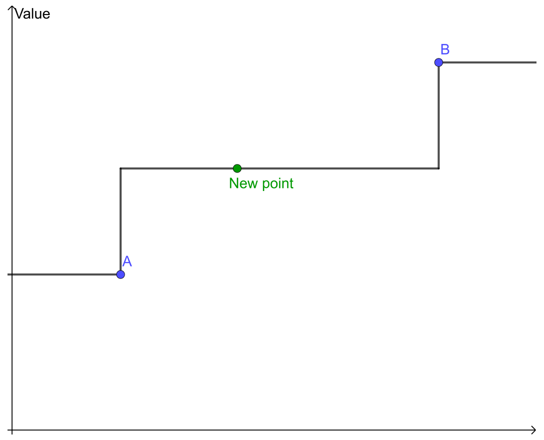


Figure 2.12: Bucket average sampling. The green point shows the new value, made from point A and B.

As figure 2.12 illustrates above, the bucket average sampling sets the new value to the average of the closest points' values.

The *bilinear* algorithm is the last considered algorithm. The one dimensional version of this algorithm is illustrated below:

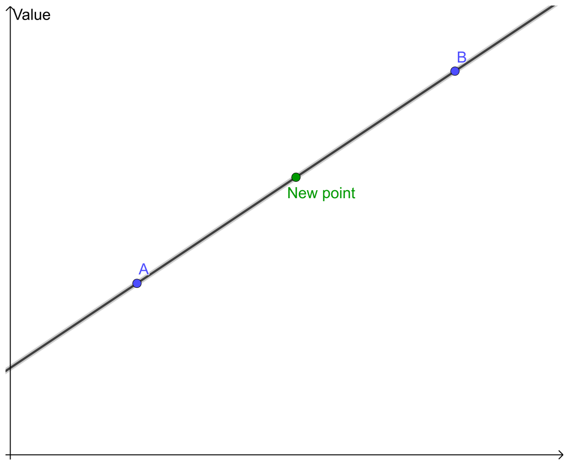


Figure 2.13: Linear sampling. The green point shows the new value, made from point A and B.

Figure 2.13 illustrates *linear sampling*, which shows the principle of *bilinear sampling*, only in one dimension. Bilinear sampling calculates the new point's value based on the old points' values and their distances from the new point. Values at the old points closest to the new point have higher influence on the new value.

The values in the satellite images contain pixels that has a value of the average light coming from that direction. If the pixel were located a little bit more to the left, it would contain more of the light that appears to the left, and less of the light from the right. This is similar to what is happening in *bucket average* and *bilinear sampling*. The value of a point generated in these algorithms is in the same way the average of the light coming from the close points. This does not apply to the *nearest sampling*. *Nearest sampling* is therefore not used.

In this project *bilinear sampling* is used as the resample algorithm. This is because of the fact that it also weightens how close it is to the old points, unlike *bucket average*.

Another benefit with *bilinear sampling* is that Satpy has implemented the ability to cache for it [17]. Caching is explained in detail in section 2.8.

2.5.3 Combining satellite images

Combining satellite images into one world-wide map is a central part of this project. As mentioned in subsection 2.5.2 above, the resampled images is helpful when combining the images. The resampled images is all in Equidistant Cylindrical projection, while having the same resolution as the final products will have. Satpy has functionality built in that handles multiple satellite images. This does not include combination of satellite images of different locations though. Because of this, the combining algorithm is manually programmed.

General description of algorithm

The goal of this algorithm is to create a world-wide map from the satellite images. Each point covered by any satellite should contain a value based on the data available for that particular point. If multiple satellite provides data for a point, all the relevant satellites should affect the resulting value, based on the quality of the satellites' imagery at that point. The algorithm is only going to add data from locations at latitudes within 70° north and south (explanation of this boundary in subsection 2.1.3).

Merging satellite imagery

As mentioned above, imagery from multiple satellites should be used if available for any point. Because of the fact that data at longitudes closer to a satellite's longitude has better quality, these locations should be weighted more than locations further away. As a consequence of this, only the two satellites closest to the specific location is considered.

When choosing how much a satellite influence the final value at a point, the sigmoid function is used together with the distance from the middle of the two influencing satellites' longitudes. The sigmoid function is expressed mathematically as following:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.15}$$

The sigmoid function from equation 2.15 above is shown graphically below:

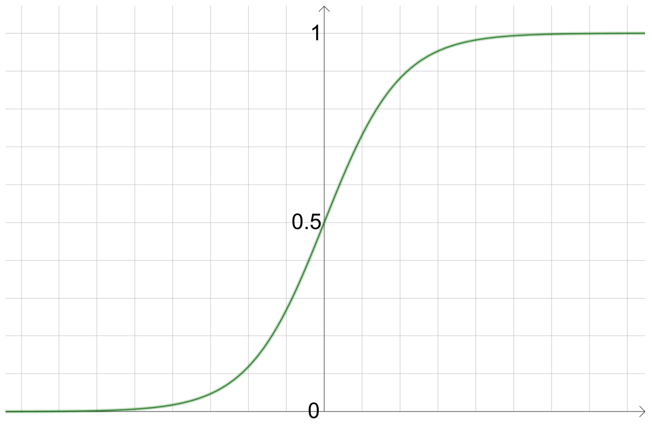


Figure 2.14: Graph of the sigmoid function

Figure 2.14 above shows a graph of the sigmoid function. As shown in the figure, $S(x)$ converges towards 0 if x is less than 0, and towards 1 if x is more than 0. $S(x)$ is 0.5 when x is 0. This function is used to combine values from two satellites. If a point's longitude is in the middle of two satellites' longitudes, the sigmoid function will calculate the following:

$$\begin{aligned} x &= 0 \\ S(0) &= \frac{1}{1 + e^{-0}} = 0.5 \end{aligned} \tag{2.16}$$

Equation 2.16 above shows that the value from both satellites is going to be used 50% each (from 0.5), if a points longitude is in the middle of the satellites' longitudes.

The equation below shows what happens if the satellite is closer to satellite A than satellite B:

$$\begin{aligned} x &= 1 \\ S(1) &= \frac{1}{1+e^{-1}} = 0.73 \end{aligned} \tag{2.17}$$

In equation 2.17 x is bigger than 0, which means that it is closer to satellite A than B. When this is put into the sigmoid function, 0.73 is returned, which tell that the value from satellite A should count for 73% of the value, while satellite B should count for the remaining 27%.

If x is big, the sigmoid function will return a number close to 1, which will almost only weighten the closest satellite. This means that only the points with longitudes that is close to the center of two satellites' longitudes will merge the satellites' values in a noticeable scale. This leads to a smooth transition between the weighing of satellites' values.

2.6 Cloud detection

Detecting clouds from satellite imagery taken with multiple frequencies is a central part of this project. By filtering light intensity between different boundaries, clouds can be detected. Frequencies can also be used together, where the differences between the light intensity is used to detect the clouds. The time of the day, as well as variances in the temperature are factors that increases the difficulty of detecting clouds. Clouds do also appear in different shapes and altitudes, which leads to their properties being different. Because of this, handling clouds with different properties is needed to get all the clouds.

2.6.1 Definition

What a cloud is does not have a clear definition. This has been defined together with Time and Date (the company this project is done for). A cloud is defined in this case as any cloud-like object that can be observed from space. This includes clouds in the sky, as well as fog, which is closer to the ground. The result is going to show how much cloud is at any point, instead of a boolean value. The final product is going to show an image of the earth, with clouds as seen from space at daytime.

2.6.2 Challenges

As mentioned above, there are many challenges and factors that plays a role in cloud detection. Some of these are mentioned below.

Detection based on time of day

As EUMETSAT describes, clouds behaves differently depending on the time of day. This is due to the lack of sunlight and colder temperatures at nighttime [12]. As mentioned in section 2.2 the lack of sunlight at nighttime makes reflective light (visible, NIR and SWIR) not very useful. The thermal light is still useful though, as it is not affected much by the time of day. This means that thermal light is more straightforward to use.

When using reflected light, the amount reflected must be taken into account. The solar zenith angle can be used for this. EUMETSAT shows that their calculations are using the solar zenith angle [12], which makes it possible to gradually change the mathematical expression, based on the amount of reflective light.

Detection over sea

Reflection and temperature differences between sea and land are other challenges that must be handled. Because of this, the earth's seas must be mapped, so it is known where to use algorithms for sea and land. EUMETSAT mention another problem introduced with the high reflectivity of water. Sunglints may disturb the result at some locations over water. This means that it is necessary to detect these sunglints, and handle them with respective algorithms. [12]

Separating clouds from snow and ice

Snow and ice is made of the water and ice crystals, just like clouds. According to EUMETSAT, this leads to difficulties differentiating clouds from snow and ice [12]. A solution to this, is mapping areas with snow and ice, due to the fact that snow and ice usually remain in the same place over some time. This can be done with light at wavelength $1.6\mu m$, because snow/ice absorbs this sunlight, in contrast to clouds [19]. The wavelength of $1.6\mu m$ is reflective light (see section 2.2), which means that it is only usable at daytime. To overcome this problem, three satellite images with 8 hours between could be used to get imagery of the whole globe with daytime.

Other methods separating clouds from snow and ice do also exist. These depends on texture difference, and the movement of clouds. These methods are hard to implement programmatically though, as they cannot determine based on single pixels.

Low clouds

Low clouds is another challenge. Low clouds do often have similar properties as cloud free areas at low altitudes. This means that they are more affected by the environment changes at night, which again leads to the need of different algorithms based on the time of day.

2.6.3 Methods considered

Machine learning

Because of the highly complex classification of clouds from satellite imagery, machine learning is often used [7]. Machine learning is well suited for problems like cloud detection, as there exists a lot of data that can be used for training. The authors of this report has little knowledge of machine learning. This method is therefore not considered, to reduce the scope of the project.

NWCSAF

While researching methods available for cloud detection, an email was sent to The Norwegian Meteorological Institute, asking for tips (see attachment A). The responsible for running the algorithms at The Norwegian Meteorological Institute, Trygve Aspnes, responded by recommending a software named NWCSAF. This software uses imagery from geostationary satellites to detect cloud data. It has been decided together with Time and Date that this software is not going to be used, as it raises some concerns. Copyright and licences is a potential issue when using external software like this. Support for all current and future utilized satellites is also not guaranteed, which could lead to an unusable program.

Algorithmic

Another method that is considered is algorithmic detection. This method depends on manually filtering and combining of frequencies. Combining frequencies is well supported by Satpy, as it supports arithmetic between frequency bands. Subsection 2.6.2 above mentions some of the problems that needs to be handled manually with the algorithmic detection method. This introduces a great deal of manual work, where every factor that influence clouds, and satellite images in general, needs to be handled.

2.6.4 Solution

Detecting clouds from satellite imagery is a complex task, that is broad for a project at this scale. Aspnes mention that The Norwegian Meteorological Institute has been developing their solution for more than 20 years (see attachment A). Because of the great complexity of detecting clouds, a simplified solution is used.

Many methods for detecting clouds from satellite imagery has been considered, which are mentioned above in subsection 2.6.3. As mentioned, both machine learning and the NWCSAF software are methods that are decided to not be put to use. Because of this the algorithmic method is chosen as the utilized method.

As the cloud detection is simplified, the challenges mentioned in subsection 2.6.2 above is avoided. This leads to multiple things:

- Reflective light will not be used, as it depends on the time of day
- Detection over sea is not handled differently than over land
- Sunglint is not considered
- Separating clouds from snow and ice is not done, which leads to snow and ice being detected as clouds
- Low clouds is not detected

The program's product is on the other hand containing high clouds. According to EUMETSAT high clouds may be detected at $10.8\mu m$. To make this wavelength fitting for every satellite, it is changed to $10.6\mu m$.

Changing the cloud detection is made to be straightforward when designing the program, as the algorithm may be changed in the future, so it is able to detect more clouds. This also makes it straightforward to change the program into producing information about e.g. cloud fires or gas emissions, if this is wanted at some point.

2.7 Product generation

As mentioned in the problem description in section 1.1, the cloud coverage is going to be visualized as static images, as well as time-lapse videos. This can be done with just the cloud coverage data generated, or with a background of the world showing the relative location of the clouds.

2.7.1 Images

The static image is useful as a data-only image, as well as a visual image with the background of the earth. The data-only image is useful when extracting cloud data for a specific position, while the visual image is useful for visualizing the cloud coverage.

As the data-only image is dependent of precise positions, it is stored in the GeoTIFF format. GeoTIFF is an image format that contains geographic data, which makes it useful for geographic uses. In addition to the GeoTIFF format, the data-only image may also be stored as a PNG image, which is used in the creation of the visual image. The GeoTIFF format does not support transparency by default, which makes PNG images more suitable for further use in the code.

The visual image is stored as a PNG image, as it is only used for visuals. The PNG format has lossless compression, which makes the image's file size smaller, while retaining the quality.

2.7.2 Video

The time-lapse video is made from the visual image, as it is made to get a visualization of the clouds movement, and not to extract data. By using multiple visual images, a video can be made with a chosen amount of hours with a chosen amount of images per hour. Number of frames per second shown is also chosen when executing the program.

2.7.3 Mapnik

Creating the background image of the earth is done with the help of the Mapnik software. Mapnik is software made for this purpose. As the image is only going to be used for visualization, it is not created at every time the background image is needed. Instead, one image is created, which is being scaled to fit cloud data at any resolution, before the visual image is created.

2.7.4 OpenCV

A lot of the creation of the visual image and the video product is done with OpenCV. OpenCV is an open-source library that includes several hundreds of computer vision algorithms [5]. To make it convenient to use with Python, a module named opencv-python has been made. This is the module that is being used in this project.

2.8 Caching

Caching is an important concept when dealing with great amount of data, or time consuming procedures. By storing reusable data on the hard drive, runtime may be minimized. Caching is used in many areas in this project.

2.8.1 Storing downloaded data

Downloading the satellite data is a time consuming process, as it is a great amount of data that is downloaded. It is therefore beneficial to keep the downloaded data stored for future uses.

2.8.2 Caching resampling calculations

As mentioned in subsection 2.5.2 about resampling, Satpy is using caching when refactoring satellite imagery. Satpy saves a lot of the calculations done when resampling, which can be used at a later time, when a similar dataset at the same resolution is resampled. This reduces the resampling runtime by a great deal.

2.8.3 Creating products from other products

When running the program, the products specified are produced. By storing the data-only product, regardless of what product is wanted, higher level products can be made out of that lower data-only product. As the GeoTIFF format is not officially supporting alpha values (transparency), the data-only product is also stored in the PNG format. This makes it more convenient to load the image later, when creating higher level products.

By creating higher level products from lower level products, most of the heavy calculations are skipped. Runtime is especially reduced when creating videos, where lower level products are already made for the previous time stamps. Situations where products are generated regularly, so recent world-wide cloud data is always available, are predicted to be a regular use case of the program. In these scenarios, video generation will be fast, as lower level products are already generated.

Chapter 3

Design and construction of software

This chapter shows and explains how the software is designed and constructed. Both code examples and diagrams are used to illustrate the software's structure. A lot of the software is based on theory explained in chapter 2.

3.1 File structure

Understanding the file structure is a good place to start, to understand how the software is constructed. The software is created in a directory named `wwclouds` (short for world-wide clouds). The directory, as well as some of its files and sub-folders is shown below:

```
wwclouds
├── __main__.py
├── config.py
├── credentials.ini
├── data/
│   ├── downloads/
│   ├── product/
│   └── satpy/
├── data_types/
├── Dockerfile
├── domains/
│   ├── product/
│   ├── satellite/
│   └── processing/
├── helpers/
└── requirements.txt
```

As shown in the directory tree above, the software is structured in sub-folders. The sub-folders right below `wwclouds` is `data`, `data_types`, `domains` and `helpers`. The files and sub-folders in the directory tree above is explained in greater detail below.

3.1.1 Files

The files in the top-level of the `wwclouds` directory are files that are general for the whole software. In the case of this project, the top-level files can be categorized into three categories: *entry point*, *configuration* and *setup*.

The only file in the *entry point* category is `__main__.py`. This is the entry point of the software. The filename given is a special filename for python projects, which tells the interpreter that this is the file that is going to run when the `wwclouds` directory is called on by the python interpreter.

The two files in the *configuration* category are `config.py` and `credentials.ini`. The `credentials.ini` file contains credentials which is specific for each user. The only content of this file is credentials for the EUMETSAT API. In addition to this, the `config.py` file is also being used for configuration. This file stores all configuration for the software, including file paths, urls, and so on.

Files used for setting up the software are put in the *setup* category. This includes `Dockerfile` and `requirements.txt`. `Dockerfile` is used for building a docker image. This is useful, as the application has a lot of complexity with dependencies. The docker image makes sure that the software is easy to run. `requirements.txt` is the file that contains modules and packages used in the project. This file simplifies the installation of modules and packages.

3.1.2 Data

The data sub-folder is the directory responsible for storing cached data, as well as products created. All the different types of cached data mentioned in section 2.8 at page 36, are being stored in separate directories. Both the desired products and the cached products are stored in the `product` directory.

3.1.3 Data types

The `data_types` directory contains generic data types that are not directly linked up to a specific domain. Data types that are linked to a specific domain are placed in the directory with the associated domain, as this increases separation of concerns.

3.1.4 Domains

The `domains` directory contains a sub-folder for each domain. Each domain contains code for different concerns.

A central domain in this project is the `product` domain. This domain handles product creation. An in-depth explanation of the product creation can be found in section 3.3.

The `satellite` domain handles problems related to the satellites, which includes downloading, as well as other satellite related problems (explained in greater detail in section 3.4).

The problems related to processing of satellite data, happens in the `processing` domain. This includes resampling (section 3.5.2, combining (section 3.5.3) and cloud image creation (section 3.5.4).

3.1.5 Helpers

General classes that do not have a direct connection to a specific domain are placed in the helpers folder. This includes classes helping with math, lists and so on.

3.2 CLI

A CLI (command-line interface) has been developed to run the software. As the program is intended to be executed by the backend, and not by the end user, a simple and concise CLI is beneficial compared to a GUI. Information about the CLI can be found by executing the following command:

```
$ python wwclouds --help
usage: wwclouds [-h] [--utctime UTCTIME] [--hours HOURS]
                [--iph IPH] [--fps FPS]
                {imagedata,imagevisual,video}
                [{imagedata,imagevisual,video} ...]
                resolution

positional arguments:
  {imagedata,imagevisual,video}
                        the desired output
  resolution            resolution of the product

optional arguments:
  -h, --help            show this help message and exit
  --utctime UTCTIME     timestamp (defaults to current time)
  --hours HOURS         hours of video, going backwards (only
                        applicable to video output)
  --iph IPH             images per hour (only applicable to
                        video output)
  --fps FPS             frames per second (only applicable to
                        video output)
```

As shown above, there are two arguments required when executing the program: **products** and **resolution**. Additionally there is one optional argument for specifying **time**. In addition to these arguments, there are three more required arguments if video is one of the chosen products: **hours**, **iph** (image per hour) and **fps** (frames per hour). These arguments are explained in greater detail below.

3.2.1 Products

The **products** argument is one of the required arguments, which represents the desired output. Multiple values are possible, which will lead to the program producing multiple products. The available products are representing the products introduced in section 2.7: *imagedata*, *imagevisual* and *video*.

3.2.2 Resolution

The second required argument is **resolution**. This argument accepts an integer, which represents the resolution of the product. The resolution is the products' desired length and width in meters, of each pixel.

3.2.3 Utctime

Utctime is an optional argument, which represents what time the products should be generated for. This time should be specified in unix time. If the argument is not given, the program will use the current time. The program do not support timestamps that goes beyond the satellites' operating time. This would lead to an exception being raised.

3.2.4 Hours

One of the three video arguments is **hours**. This argument is used for specifying the number of hours the video should contain. The last frame in the video will be at the time related to the **utctime** argument, while the first frame is the specified number of hours before the last frame.

3.2.5 Iph

Iph (images per hour) is another argument required when creating the video product. The number of images within each hour is specified by this argument.

3.2.6 Fps

To specify the speed of the video, the **fps** (frames per second) argument is added. The update frequency of videos is usually specified by frames per second. Frames per second describes how many frames there is within one second of video. In the case of this project, the frames per second is also specifying the speed of the video, as the total amount of frames is already set by the **hours** and **iph** arguments.

3.3 Product creation

The software is centralized around creating various products, which are chosen by the user through the command-line interface, mentioned above in section 3.2. As mentioned in subsection 2.8.3, higher level products are created out of lower level products, where lower level products are cached for future requests. The logic of product creation is centralized around creating the desired products from lower level products. The flowchart below shows the product creation logic:

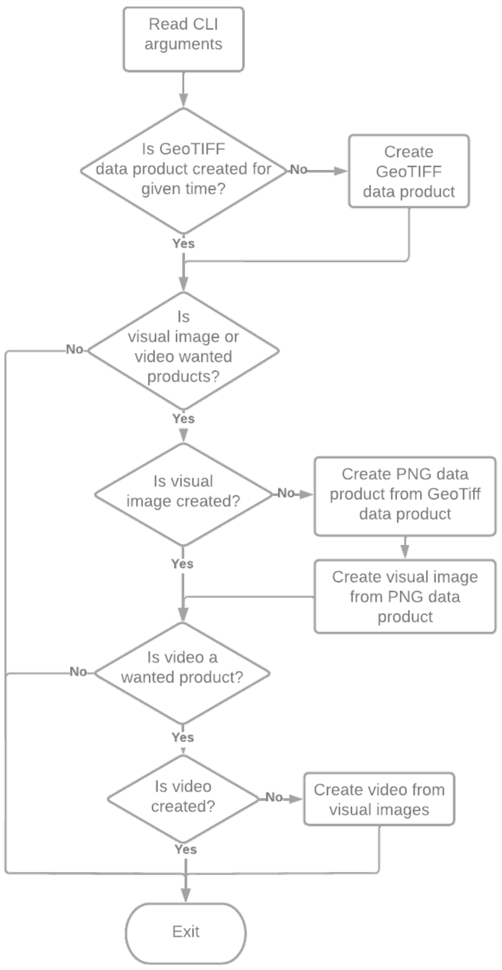

Read CLI arguments
Is GeoTIFF data product created for given time?
No
Create GeoTIFF data product
Yes
Is visual image or video wanted products?
No
Yes
Is visual image created?
No
Create PNG data product from GeoTiff data product
Yes
Create visual image from PNG data product
Is video a wanted product?
No
Yes
Is video created?
No
Create video from visual images
Yes
Exit

Figure 3.1: Flowchart showing a general overview over the product creation

The flowchart in figure 3.1 above gives a general overview over how the products are generated. This includes the logic where higher level products are generated from lower level products. The code below shows how the general product creation is implemented, excluding the handling of already created products:

Code 3.1: domains/product/product_creator.py. General product creation

```
177     print("Creating imagedata")
178     self.__create_imagedata_for_products()
179     if self.product_enum & (ProductEnum.IMAGEVISUAL | ...
            ProductEnum.VIDEO):
180         print("Creating imagevisual")
181         self.__create_imagevisual()
182     if self.product_enum & ProductEnum.VIDEO:
183         print("Creating video")
184         self.__create_video()
```

Code 3.1 above and the flowchart in figure 3.1 shows that the GeoTIFF data image is created, regardless of what products is specified (line 178). It is only created if it does not exist though, for the given time. The reason why it is created at all times, is because of the fact that it is the lowest level product. Every other product depends on the GeoTIFF data image. Using the same reasoning, it is shown that the visual image is created every time the video product is desired (line 179-181). This is also only if the visual image has still not been created for the given time. How each product is created is explained in greater detail in subsection 3.3.1 to 3.3.3 below.

3.3.1 Data image

As explained in subsection 2.7.1, the data image is a product that contains only the generated cloud data, without any background. Because this is the lowest level product, the process of creating the image from scratch is necessary. This process contains multiple sequentially executed steps:

1. Downloading the necessary satellite data (see 3.4).
2. Create a combined world-wide dataset, containing data for each satellite band utilized (see 3.5.2 and 3.5.3).
3. Create cloud image from the world-wide dataset (see 3.5.4).

The process of creating the data image that is explained above, is a time consuming process. This is the reason why it is beneficial to cache the data image, as explained in subsection 2.8.3.

3.3.2 Visual image

The visual image is, as mentioned in subsection 2.7.1, useful as a visualization of the clouds. A background image of the earth is added to visualize the relative position of the clouds. To create the visual image, both the data image and the background image is needed. The data image is created as described in 3.3.1 above, while the background image is being statically available, as it has already been created (subsection 2.7.3 describes how the background was initially created). The background image is shown below:

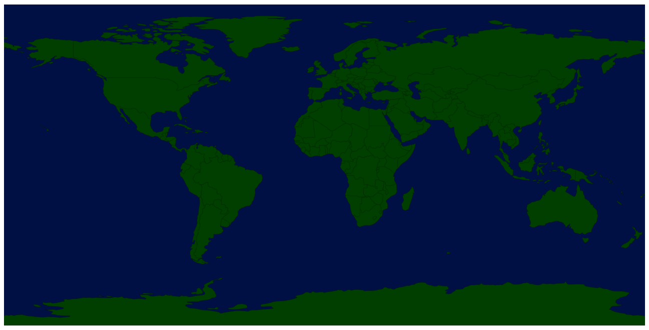


Figure 3.2: Image of the earth, used as a background image

The background image of the earth, shown in figure 3.2 above, shows the earth's landmass in dark green, and oceans/lakes in dark blue. Having the image dark makes the white clouds more visible when they are added to the image.

Essentially, what the creation of the visual image is doing, is adding the data image on top of the background image. Before this can be done, the background image must first be resized to match the data image's resolution. The following code shows how the background image is loaded and resized:

Code 3.2: domains/product/image_visualimage_visual.py. Loading and resizing background image.

```
    image = cv2.imread(self.world_map_filepath, ...
        cv2.IMREAD_UNCHANGED)
    image_resized = cv2.resize(image, self.resolution)
    self.__image = image_resized
```

As shown in code 3.2 above, the cv2 module (opencv-python) is being used for both reading and resizing the background image. When reading the image at line 48, `IMREAD_UNCHANGED` is used as read mode [5]. This tells cv2 to read the image as it is. After loading the image, it is resized (at line 49) to have the same resolution the data image. When the image is resized it is assigned to the `self.__image` attribute, which is the attribute that contains the image used for the visual image creation.

After loading and resizing the background image, the data image is being added. The code showing how the data image is added is shown below:

Code 3.3: domains/product/image_visualimage_visual.py. Adding data image onto background image.

```
80      for c in range(0, 3):
81          self.__image[y1:y2, :, c] = (alpha_s * s_img[:, :, ...
                c] + alpha_l * l_img[y1:y2, :, c])
```

The code in 3.3 is generalized to work with images of different resolutions. Because of this, `s_img` and `l_img` is used to represent the smaller and larger image. In this case, `s_img` is the data image, and `l_img` is the background image. Line 80 start off by iterating over the indexes of the three channels that contains colors. For each iteration, a channel is added to the `self.__image` attribute. This is shown in line 81. As the `l_img` variable may be larger vertically than the `s_img`, a range with the vertical size of the `s_img` (`y1` to `y2`) is used. This is not done horizontally on the x-axis, as both images has previously been resized to have the same length on the x-axis. The reason why the images has been resized to have the same length horizontally, but not vertically, is because both images must contain the whole range of longitudes, while it is acceptable if they do not contain latitudes close to the poles.

As shown in line 81, the channels of each image is multiplied by alpha matrices (`alpha_s` and `alpha_l`). These matrices represents the desired alpha values (transparency) of each image. The `alpha_l` matrix is created to be the inverted of the `alpha_s`, which is shown in the code below:

Code 3.4: domains/product/image_visualimage_visual.py. Creating alpha_l.

```
77      alpha_l = 1.0 - alpha_s
```

The alpha matrices in code 3.4 contains floats between 0 and 1. As the matrices is the inverted of each other, adding them together would make a matrix containing only ones. Going back to code 3.3, it is noticeable that each channel is being multiplied with the the alpha matrices. As the alpha values adds up to ones, a image without any transparency is created, where each value is weighted based on the associated alpha matrix. This leads to a combined image where the data image, is added to the top of the background image.

3.3.3 Video

The highest level product is the time-lapse video product, which is explained in subsection 2.7.2. As mentioned, the video product is a video made for visualization purposes, created from visual images. The properties of the video is decided by the **hours**, **iph** and **fps** CLI arguments (explained in section 3.2). Using the properties specified, the video creation is done by executing the following method:

Code 3.5: domains/product/product_creator.py. Method that create video.

```
171     def __create_video(self) -> None:
172         image_paths = ...
                self.__create_imagevisuals_for_video( ...
                self.hours, self.images_per_hour)
173         VideoMaker(self.__video_path, image_paths, ...
                self.fps).create()
```

Code 3.5 shows a general overview over how the video is generated. As seen in line 172, the visual images are first created. With the paths of each visual image created, the video can be generated at line 173. These methods are explained in greater detail below.

Creating visual images for video

As mentioned, the method `__create_imagevisuals_for_video` is responsible of creating the images used in the video. This method is displayed below:

Code 3.6: domains/product/product_creator.py. Method that create visual images for video.

```
159     def __create_imagevisuals_for_video(self, hours: int, ...
            frames_per_hour: int) -> list[str]:
160         image_paths = []
161         for hour in range(hours):
162             for hour_frame in range(frames_per_hour):
163                 minute = int((60 / frames_per_hour) * ...
                        hour_frame)
164                 time_stamp = self.utctime - ...
                        time∆(hours=hour, minutes=minute)
165                 product_creator = self.__copy( ...
                        product_enum=ProductEnum.IMAGEVISUAL, ...
                        utctime=time_stamp)
166                 product_creator.create_products()
167                 image_paths.append( ...
                        product_creator.imagevisual_path)
168         image_paths.reverse()
169         return image_paths
```

The method shown in code 3.6 above starts of at line 161 by iterating over a range with length equal to the desired number of hours. The same logic continues on line 162, with the number of frames wanted per hour. Using the position of both of these iterators, the time stamps going backwards, is being calculated at line 163-164. The time stamp is traversing backwards in time at a constant speed at each iteration.

For every time stamp, a `product_creator` object of the class `ProductCreator` is created at line 165. The `ProductCreator` is the class that handles the creation of product at a specific time. The new `product_creator` object is made as a copy of the existing one, which leads to attributes like the resolution being transferred. The `utctime` and `product_enum` attributes are being changed in the new object though, as it is only the visual image at the time of the `time_stamp` variable, that needs to be created. Using the `product_creator` object, the visual image product is being created with the method call on line 166.

After the visual image at a time has been created, its path is being added at line 167 to a list containing all the paths of the newly created visual images. When all the visual images has been created, the list containing their paths is being reverted at line 168. This makes the list sorted after the associated time of each `ProductCreator` object.

Creation of the video itself

With the visual images created, the creation of the video itself is ready to be started. This is done with the `VideoMaker` class, which is used in code 3.5 on page 49. The `VideoMaker` class accepts three arguments: the video's destination path, the images' source paths and the number of frames per second. The object of the `VideoMaker` class is used with the `create` method, which creates the video from the images' source paths, and put them into a video at the given destination path. The `create` method is shown below:

Code 3.7: domains/product/video_maker/video_maker.py. Method that create video from a list of images.

```
10      def create(self):
11          start_frame = cv2.imread(self.image_paths[0])
12          height, width , _ = start_frame.shape
13          fourcc = cv2.VideoWriter_fourcc('m', 'p', '4', ...
                'v')
14          video = cv2.VideoWriter(self.dest_path, fourcc, ...
                self.fps, (width, height))
15          for image_path in self.image_paths:
16              frame = cv2.imread(image_path)
17              video.write(frame)
```

The method shown in code 3.7 above, starts of at line 11-12 to receive the height and width of the first image. This will be the shape of the created video. Line 13 continues with getting the opencv-python (see 2.7.4 for more information about OpenCV) object for writing mp4 videos. This object is used together with the shape of the image, the specified fps and the destination path, to create an object of the `VideoWriter` class. The `video` object of the `VideoWriter` class is being used to write the image frames into the video destination path. This is done at line 15-17. The source image' paths is iterated over, before they are read and finally written to the destination path, using the `video` object of the `VideoWriter` class.

3.4 Obtaining satellite data

Obtaining the satellite data is necessary, before the creation of the data image can begin. This is done in the satellite domain (see 3.1.4 at page 41). The theory behind obtaining the satellite data is explained in section 2.3 on page 19. The code is centralized around having the possibility of replacing the utilized satellites, in a simple way. Why this is important is explained in subsection 2.1.5. Another focus has been to keep the complexity of obtaining satellite data hidden from other parts of the code. Doing this makes the data easier to work with outside of the satellite domain. How the cases mentioned above are handled is explained in detail in the following subsections.

3.4.1 Satellite collection

As mentioned, keeping the complexity of obtaining the satellite data is a priority in this domain. To collect the handling of all satellites collected, the `SatelliteCollection` class has been created. The `SatelliteCollection` class is working as a collection that interacts with multiple satellites.

To create a new `SatelliteCollection` object, the following `__init__` method is being called:

Code 3.8: domains/satellite/satellite_collection.py. SatelliteCollection __init__ method.

```
11      def __init__(self, satellite_enums: list[SatelliteEnum]):
12          self.satellites: [SatelliteType] = ...
                list(map(SatelliteMapping.get_satellite_type, ...
                satellite_enums))
```

The `__init__` method of the `SatelliteCollection` class shown above in code 3.8, accepts a list of `SatelliteEnum` enum objects. Each of these enums represents different satellites (explained in subsection 3.4.2 below). At line 12, these enums are used with the static `SatelliteMapping` class to get an associated `SatelliteType` object. The `SatelliteType` class is a class used for grouping satellites of the same type. This class is explained in greater detail in subsection 3.4.3.

Using the `self.satellites` list containing `SatelliteType` objects (see line 12 in code 3.8 above), the `SatelliteCollection` object can easily interact with all the different satellites. Two public methods has been created doing this.

One of the two public methods implemented for the `SatelliteCollection` class is the `get_scan_times_strings` method shown below:

Code 3.9: domains/satellite/satellite_collection.py. Method for SatelliteCollection that retrieve strings representing the scanning start times.

```
    def get_scan_times_strings(self, frequencies: ...
        list[float], utctime: datetime) -> tuple[str, str]:
```

What the `get_scan_times_strings` method in code 3.9 above does, is finding the times for when each satellite starts to scan (starts to take the image). A tuple of two strings is then returned. The first string is representing the specific day, while the second string represents the times of the day that each satellite started to create the imagery. Finding the times is done by looking backwards from the time specified in the argument. The strings returned by this method is useful when storing the downloaded data. These strings can then later be used to identify downloaded satellite data.

The second public method the `SatelliteCollection` class has is called `download_all`, and is shown below:

Code 3.10: domains/satellite/satellite_collection.py. Method for SatelliteCollection that download satellite data for all utilized satellites.

```
    def download_all(self, frequencies: ...
        Optional[list[float]], utctime: datetime) -> ...
        [downloader.FileReader]:
```

To make downloading of all the desired satellite data easier, the `download_all` shown in code 3.10 above has been made. This method accepts time and a list of frequencies as arguments. Using these arguments, the method is downloading satellite data of the given time and frequencies, for every satellite specified in the constructor (see code 3.8). As described in subsection 2.3.2 at page 20, the satellites separate the data in multiple files, in different file formats (see subsection 2.5.1). This problem is handled by returning a list of the `FileReader` class.

The `FileReader` class accepts file paths and the name of a file reader as arguments. This class has one task: reading the file paths, with the reader specified. It returns a Satpy Scene object, which is useful later when processing the satellite data.

3.4.2 Satellite enum

The `SatelliteEnum` class is implemented to make satellite selection simpler. This is a subclass of the `Enum` class, which makes it act like an enum. As it is desirable to use every satellite, a class method named `all` has been implemented:

Code 3.11: domains/satellite/satellite_enum.py. SatelliteEnum all method.

```
    @classmethod
    def all(cls) -> ["SatelliteEnum"]:
        return [satellite_enum for satellite_enum in cls]
```

The `all` method shown above in code 3.11 iterates through `SatelliteEnum`, and creates a list of every element in it. This method allows for cleaner code in cases where every satellite is necessary. An example where the `SatelliteCollection` class is used together with the `all` method is shown below:

Code 3.12: domains/product/product_creator.py. SatelliteEnum all method, used together with the SatelliteCollection class.

```
    self._satellite_collection = ...
        SatelliteCollection(SatelliteEnum.all())
```

As shown in code 3.12 above, the `SatelliteCollection` class and the `SatelliteEnum.all` method makes it simple for other domains to initialize a collection of satellites. It is then, as mentioned in subsection 3.4.1, easy to use code that handles all satellites.

3.4.3 Satellite type

To group similar satellites, the `SatelliteType` class has been made. Grouping satellites is done with satellites from the same operator that have similar properties, as well as similar ways to download its data. The `SatelliteType` class acts as a superclass for underlying classes, which is implemented for every group of satellites.

The subclasses of the `SatelliteType` class are useful for two things. This includes mapping frequencies and bands, as well as creating a downloader object which will be used for downloading data for the specific satellite (explained in depth in subsection 3.4.4 below).

The mapping between frequencies and bands is being hard coded for every subclass, even though this is usually considered bad practice. This is because it will not change at any time for any satellite type. Mapping frequencies and bands is necessary, as the bands are used to find the files online. The bands are not something that should be exposed to domains other than the satellite domain though, as the frequencies are what decides what the data may be used for. The bands are therefore being found with the mapping, before the bands are being used for anything. The code that gets bands from frequencies are made in the `SatelliteType` superclass, which makes them useful for every `SatelliteType` subclass.

In contrast to the subclasses for the satellites from NASA/NOAA and JMA, mappings are not being implemented for the subclass for satellites from EUMETSAT/ESA. The mappings are not necessary for these satellites as the files are not separated by bands (see subsection 2.3.2 at page 20).

3.4.4 Downloaders

The actual downloading of the satellite data is done with help of the `Downloader` class, and subclasses originating from it. As told in section 2.3 at page 19, the satellite data is received from two providers: AWS and EUMETSAT. This is handled by creating a subclass of the `Downloader` class for both of them. AWS provides data for both the GOES satellites (GOES-16 and GOES-17) from NASA/NOAA and the Himawari 8 satellite from JMA. A subclass from the AWS class is thereby created for both of these operators. The structure of the downloader classes is shown below:

```
Downloader
├── Aws
│   ├── Himawari
│   └── NoaaGoes
└── Meteosat
```

Splitting the downloaders into multiple classes is beneficial, as it increases code reusability. Code reusability is important in this case, as satellites will be replaced regularly (see subsection 2.1.5). Facilitating code reusability makes it simple to replace satellites at later occasions. An explanation of the classes shown in the tree above is provided in greater detail in the following subsections.

Downloader superclass

The superclass of the downloaders is helpful as it provides a generic interface for all the downloaders. This makes it more convenient to use the downloaders. As shown below at line 13-16 in code 3.13, the arguments that every downloader subclass needs to provide to the superclass's constructor is `subdir`, `reader` and `update_frequency`.

Code 3.13: domains/satellite/downloader/downloader.py. Downloader superclass constructor.

```
12 class Downloader(metaclass=abc.ABCMeta):
13     def __init__(self,
14                  subdir: str,
15                  reader: str,
16                  update_frequency: timeΔ):
17         self.subdir = subdir
18         self.reader = reader
19         self.update_frequency = update_frequency
```

The `subdir` argument is the sub directory, relative to the download directory (mentioned in section 3.1), that the data from the downloader is going to be downloaded to. Which Satpy reader is going to be used is specified with the `reader argument`. Lastly, the update frequency of the data is being specified in the `update_frequency` argument.

Every subclass of the `Downloader` class must also implement two methods:

Code 3.14: domains/satellite/downloader/downloader.py. Downloader abstract methods

```python
    @abc.abstractmethod
    def _download(self, bands: Optional[List[str]], time: ...
        datetime) -> [str]:
        pass

    @abc.abstractmethod
    def _get_previous_scan_start_time_for_band(self, band: ...
        str, time: datetime) -> datetime:
        pass
```

As shown above in code 3.14 above, the methods named `_download` and `_get_previous_scan_start_time_for_band` are abstract methods, which must be implemented by every subclass. The `abc.abstractmethod` decorator makes sure that these methods are implemented, as it raises an error otherwise.

Aws

As mentioned, the `Aws` class is one of the subclasses of the `Downloader` class. It is also the superclass of both the `NoaaGoes` and `Himawari` class. This means that it acts as a layer between the `Downloader` and `NoaaGoes`/ `Himawari` classes. It provides both the two methods in code 3.14, as well as methods interacting with its subclasses. The methods interacting with its subclasses includes both abstract methods and methods that helps its subclasses interact with Amazon Web Services (AWS). In addition to these methods, it requires two more arguments in its constructor, in addition to the ones that the `Downloader` superclass needs. These additional arguments are the strings `bucket` and `product`.

The strings `bucket` and `product` are two variables that are used for finding files in AWS. The illustration below gives a visualization of AWS stores data:

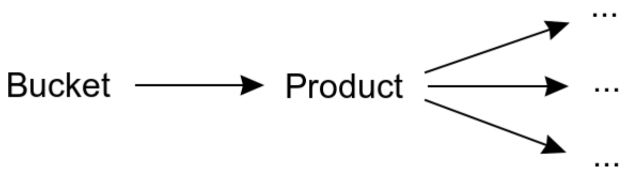


Figure 3.3: Illustration of how data is stored in aws.

Figure 3.3 shows how the bucket and product is used to find data in AWS. Every collection of data in AWS is stored in buckets. In the case of this project, GOES-16, GOES-17 and Himawari 8 do all have their own bucket. The buckets stores data in multiple variants. It is the variant of the full disc that is needed for all the utilized satellites. This is therefore chosen as the AWS product. The files are then stored in a hierarchy of directories based on time, with filenames that reflect the products. How the hierarchy is organized is different for the GOES-16/GOES-17 and Himawari 8 satellites. Because of this, the following abstract method is being made for all subclasses of the `Aws` class:

Code 3.15: domains/satellite/downloader/aws.py. Aws abstract method for obtaining prefix of files' path in AWS.

```
    @abc.abstractmethod
    def _get_aws_prefix_for_band(self, band: str, time: ...
        datetime) -> str:
        pass
```

The abstract method introduced in code 3.15 above is implemented for both the `NoaaGoes` and `Himawari` subclasses. It returns a string containing the prefix of the path to files of the specified band, at the specified time. If no file is found with the given arguments, the program is searching backwards in time for matching files. This leads to searching for files becoming a time consuming process. The results from the method that searches for files are therefore being cached in memory, using `lru_cache`.

The prefixes received for specific bands at specific times, are extended into receiving all the paths to files with that prefix. Using this further, paths to files of all the utilized bands are found for the specified time. When all the paths are found, the following code is executed:

Code 3.16: domains/satellite/downloader/aws.py. Code in Aws class that downloads the imagery from AWS

```
87      for key in keys:
88          file_path = self._get_local_file_path(key)
89          if not self._file_is_downloaded(key):
90              s3t.download(self.bucket, key, file_path)
```

The code in 3.16 above shows how files are downloaded from AWS. It starts of by iterating through every key (Amazon Web Services (AWS) path) at line 87. Continuing on line 88, the local file path is generated from the key. Line 89 checks thereafter if the file has been downloaded before. If it is not downloaded before, the code at line 90 starts the downloading.

As mentioned in table 2.6, the Himawari 8 satellite by JMA is compressed as a bz2 file. Because of this, post handler method is being run after downloading:

Code 3.17: domains/satellite/downloader/aws.py. Post handler method call in the Aws class that runs on every downloaded file.

```
93      return list(map(self._file_posthandler, file_paths))
```

The post handler method, `self._file_posthandler`, in code 3.17 above is executed on every downloaded file. If the method is not implemented in any subclass, it is just returning the file path. This method is useful for the `Himawari` subclass though. The method is being overwritten by the subclass to decompress the bz2 files that is downloaded. It is then deleting the compressed bz2 files, which at this point is not necessary anymore. Doing this makes the subclasses more consistent. Now all the `Aws` subclasses returns decompressed files.

As mentioned earlier, the `Aws` class implements two abstract methods. The second one is shown below:

Code 3.18: domains/satellite/downloader/aws.py. Aws abstract method for getting the scanning start time from object key

```
    @abc.abstractmethod
    def _get_scan_start_time_from_object_key(self, ...
        object_key: str) -> datetime:
        pass
```

The abstract method shown in code 3.18 above is getting the starting time of the scan from keys (AWS paths). This is useful, as the staring time is necessary when implementing the `_get_previous_scan_start_time_for_band` method (shown in code 3.14) of the `Downloader` superclass. The starting time is not available in any places other than in the key and the actual file. As it is beneficial to not download the whole file, every time the starting time is needed, the key is being parsed instead.

Meteosat

The second direct subclass of the `Downloader` class is the `Meteosat` downloader class. This class is responsible for downloading the satellite data for the EUMETSAT/ESA satellites: Meteosat 8 and Meteosat 11. EUMETSAT has created an API for interacting with its satellites' data, which is being utilized. To download any data from EUMETSAT, a user account at their systems is needed (mentioned in section 2.3 at page 19). The credentials for this user account is being stored in the `credentials.ini` file. This file must be created manually by every user of the software, as it is user specific (see subsection 3.1.1).

Before any data can be downloaded from the API, an access token is needed. This token is fetched with the help of the credentials for the EUMETSAT user. The method below shows how the access token is fetched:

Code 3.19: domains/satellite/downloader/meteosat.py. Start of method that receives access token.

```python
50      def __get_access_token(self) -> str:
51          response = requests.post(
52              url=config.METEOSAT_TOKEN_ENDPOINT,
53              auth=requests.auth.HTTPBasicAuth( ...
                    config.METEOSAT_CONSUMER_KEY, ...
                    config.METEOSAT_CONSUMER_SECRET),
54              data={'grant_type': 'client_credentials'},
55              headers={"Content-Type": ...
                    "application/x-www-form-urlencoded"}
56          )
57          response_json = response.json()
58          access_token = response_json.get('access_token')
```

Method `__get_access_token` in code 3.19 above, shows how the access token for the EUMETSAT API is fetched. The API has a separate endpoint (line 52) for receiving the token, which uses the user credentials (line 53) to verify that the user is legitimate.

When the access token is fetched, the `Meteosat` API needs a collection id to find the desired product. This collection id does the same as the bucket and product in AWS (see figure 3.3). It describes which satellite is going to be used, and that it is going to use the full disc product. In the same way as with the `Aws` class' data, the data is sorted on scanning start time. By using the specified time, the path to the desired file can be found right away. The only exception of this is for times that does not contain any data yet. In these cases, the program is searching for data at other times, going backwards, in the same way as the `Aws` class does. As the data is not separated in multiple files by bands (see subsection 2.3.2), filtering unused bands is not needed.

After the file path has been found, the url for downloading the given file is being created. With the access token and download url, the data is ready to be downloaded. A response steam is created for this purpose:

Code 3.20: domains/satellite/downloader/meteosat.py. Code that creates a stream for downloading the satellite imagery in the Meteosat class.

```
101        if not self._file_is_downloaded(download_url):
102            access_token = self.__get_access_token()
103            stream_response = requests.get(
104                url=download_url,
105                params={"format": "json"},
106                stream=True,
107                headers={"Authorization": f"Bearer ...
                       {access_token}"})
```

Code 3.20 above shows how the downloading of data in the `Meteosat` class starts off. This code starts at line 101 of by checking if the file has been downloaded before, as it is not necessary to download it if it can be found locally. The access token is then fetched at line 102. With the access token and download url, a response stream is created at line 103-107. As shown in line 107, the access token is added to the header. A stream is created, because it allows for downloading in chunks, which is beneficial when downloading big amounts of data. This way, the data can be downloaded without loading the entire file into memory.

The stream created in code 3.20 above is being written in iteration to a local file path. This is shown in the following code:

Code 3.21: domains/satellite/downloader/meteosat.py. Code that reads the stream created in code 3.20.

```
108            with open(filepath, "wb") as f:
109                for chunk in stream_response.iter_content( ...
                       chunk_size=1024):
110                    if chunk:
111                        f.write(chunk)
112                        f.flush()
```

The local file path of the file that is being downloaded, is at line 108 in code 3.21 above being opened in write-binary mode. This makes it possible to write the downloaded binary data to the file. Line 109 continues with iterating through the stream. This downloads the file in chunks of 1024 bytes, in each iteration. These chunks are then being written to the local file at line 111-112. After downloading the file, the file path is returned to the `Downloader` superclass, which creates a file reader of the `FileReader` class (mentioned in 3.4.1) for it. The file is then being ready for use outside of the satellite domain.

3.5 Processing

As mentioned in subsection 3.3.1, the process of creating the data image product, must be done by scratch, as this is the lowest level product. This section explains how the processing is done. This includes, *loading* (subsection 3.5.1), *resampling* (subsection 3.5.2), *combining* (subsection 3.5.3) and *cloud extracting* (subsection 3.5.4). Theory behind the processing of image data is discussed in section 2.5 and 2.6 at page 23-34.

A general overview over the processes done when creating the data image product is shown in the flowchart below:

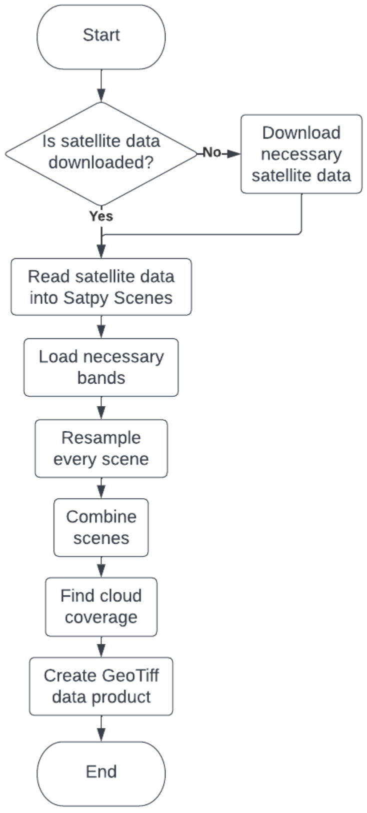

Start
Is satellite data downloaded?
No
Download necessary satellite data
Yes
Read satellite data into Satpy Scenes
Load necessary bands
Resample every scene
Combine scenes
Find cloud coverage
Create GeoTiff data product
End

Figure 3.4: Flowchart of the processing

The flowchart in figure 3.4 above shows a series of subsequent processes, needed to create the data image product. The design and construction behind the downloading of the satellite data is explained in section 3.4, starting at page 52.

3.5.1 Loading

As visualized in figure 3.4, the first processes done after the downloading, is reading and loading of the data. This is easily done, thanks to the `FileReader` objects, and Satpy.

As mentioned in section 3.4, the satellite domain implements a class for reading the satellite data. The file paths for the satellite data, as well as the reader that Satpy uses for reading is stored in the `FileReader` objects. Because of the fact that the objects returned when downloading the satellite data is of the `FileReader` class, reading the satellite data can be done in two lines of code:

Code 3.22: domains/product/product_creator. Code that shows the downloading and reading of satellite data.

```
111      file_readers = ...
             self._satellite_collection.download_all( ...
             frequencies=self._frequencies, utctime=self.utctime)
112      scenes = [reader.read_to_scene() for reader in ...
             file_readers]
```

The first line in code 3.22 above starts off by downloading satellite data for all the utilized satellites, for the specified frequencies and time. The returned `FileReader` objects can be then be read to a Satpy Scene object, with the `read_to_scene` method. This is done for every `FileReader` object at line 112.

Satpy contains a class named `MultiScene`, which is useful for handling multiple `Scene` objects. The `MultiScene` class lacks some functionality for `Scene` objects with satellites at different locations though. To add functionality specific for this project, a subclass of the `MultiScene` has been created. This subclass is named `MultiSceneExt` and acts as a extension of the `MultiScene` class. An extension (subclass) has also been made for the

`Scene` class, named `SceneExt`. Using the `Scene` objects at line 112 in code 3.22, an object of the `MultiSceneExt` class is being constructed:

Code 3.23: domains/product/product_creator. Constructing object of the MultiSceneExt class.

```
    multi_scn_ext = MultiSceneExt(scenes)
```

The constructor for the `MultiSceneExt` class shown above in code 3.23 accepts a list with objects of either the `Scene` or `SceneExt` class. If the objects is of the `Scene` class, they are made into `SceneExt` class.

The `MultiSceneExt` object made in code 3.23 contains various data about the utilized satellites. The bands which contains data has not been loaded though. This has to be done manually. Loading the bands is done in the following line code:

Code 3.24: domains/product/product_creator. Object of the MultiSceneExt class loading bands for all Scene objects.

```
    multi_scn_ext.load(self._frequencies, ...
        resolution=self._legal_resolutions)
```

As seen in code 3.24 above, loading bands are done with the list of the frequencies needed and a list of legal resolutions. The legal resolutions are provided to filter out some unused bands. Because the Meteosat satellites has every band available (see subsection 2.3.2), interfering bands needs to be filtered out. The bands associated to each frequency are then found and loaded for every `SceneExt` in the `MultiSceneExt` object.

3.5.2 Resampling

Resampling the satellite data is as mentioned in subsection 2.5.2, a complex process. This is the process of changing the satellite imagery into having the same resolutions in a given projection (see section 2.4 for an explanation of projections). As mentioned in subsection 2.5.2, Satpy simplifies this problem by a great deal. Resampling with Satpy can be done in one line.

The resampling done in this project is comprehensive. Because of this, the resampling is being done with more than one line of code.

Resampling collection of Scene objects

To integrate the resampling process with the extended classes, `SceneExt` and `MultiSceneExt`, multiple methods for the classes has been made. The entry point of the resampling process is the `resample_loaded_to_eqc` method. This method is shown in the following code:

Code 3.25: domains/processing/multiscene_ext. Resampling loaded bands to EQC projection for MultiSceneExt

```
114     def resample_loaded_to_eqc(self, resolution=None, ...
            **kwargs):
115         start_time = time.time()
116         groups = self.group_loaded()
117         eqc_mscn = self.resample_all_to_eqc(resolution, ...
                **kwargs)
118         eqc_mscn.shared_dataset_ids = groups
119         print(f"Resampled scenes: {round(time.time() - ...
                start_time, 4)} sec")
120         return eqc_mscn
```

The method shown above in code 3.25 resamples all the loaded Scene objects to Equidistant Cylindrical (EQC) projection, with the given resolution. It starts off at line 116 by grouping the loaded bands. The groups created is grouping together bands with similar frequencies, which makes the collection of Scene objects easier to work with. Every Scene is then resampled at line 117 with the `resample_all_to_eqc` method. This is explained in greater detail in the next paragraph. This creates a new object of the `MultiSceneExt` class, with the newly resampled data. At line 118 the previously created groups is added to the `shared_dataset_ids` variable. This variable is storing information about which bands is grouped together.

The actual resampling is done in the `resample_all_to_eqc` method. This method is shown below:

Code 3.26: domains/processing/multiscene_ext. Resampling all scenes for MultiSceneExt

```
105     def resample_all_to_eqc(self, resolution=None, ...
            **kwargs) -> "MultiSceneExt":
106         return MultiSceneExt([
107             scn.resample_to_eqc_area(resolution=resolution, ...
                    reduce_data=False, **kwargs) for scn in ...
                    self.scenes
108         ])
```

Resampling all the Scene objects with the desired resolution and projection, is done by executing the method in code 3.26 is executed. This method iterates through every Scene object at line 107 and resamples them one by one. The `resample_to_eqc_area` method that is being called on at line 107 is the method in the `SceneExt` class that resamples that specific Scene. This method is explained further below.

Resampling single Scene

Resampling of a single Scene is, as mentioned done with the `resample_to_eqc_area` method in the `SceneExt` class. The method starts off with the following lines of code:

Code 3.27: domains/processing/scene_ext. Resampling a single scene with SceneExt, start

```
82      def resample_to_eqc_area(self, *, resolution=None, ...
            **kwargs) -> "SceneExt":
83          projection = {"proj": "eqc", "lon_0": self.lon_0}  ...
                # Equidistant cylindrical projection
```

To start off, the resampling method `resample_to_eqc_area` above in code 3.27 specifies the desired projection. The desired projection is as mentioned Equidistant Cylindrical (EQC), which is specified in the dictionary at line 83 with `"proj": "eqc"`. To keep the Scene's longitude, the longitude is also added to the projection dictionary, with `"lon_0": self.lon_0`.

Before the `Scene`'s resample method can be used, a `AreaDefinition` object must be created. This object is created in the code below:

Code 3.28: domains/processing/scene_ext. Resampling a single scene with SceneExt, AreaDefinition part.

```
85          area_def_args = dict()
86          if resolution is not None: resolution
87              area_def_args["resolution"] = resolution
88
89          area_def = create_area_def(
90              area_id="eqc_area",
91              projection=projection,
92              **area_def_args
93          )
```

Code 3.28 above shows the lines of code in the `resample_to_eqc_area` method that is executed after the lines in code 3.27. This part of the method is as mentioned responsible for creating the `AreaDefinition` object. The first thing that is happening is that a dictionary containing the `AreaDefinition` object's arguments is created. As seen in line 86-87, the desired resolution is added to the dictionary, if the resolution exists. With the `AreaDefinition` objects arguments and the projection, the `AreaDefinition` is created at line 89-93.

When the `AreaDefinition` object is created, the Scene is ready to be resampled by its resample method:

Code 3.29: domains/processing/scene_ext. Resampling a single scene with SceneExt, resampling part.

```
94      return self.resample(
95          destination=area_def,
96          resampler="bilinear",
97          cache_dir=DATA_PATH_SATPY_RESAMPLE_CACHE,
98          **kwargs
99      )
```

As shown in code 3.29 above, the Scene is being resampled right after the `AreaDefinition` object has been created. There is multiple arguments added to the `resample` method shown: `destination`, `resampler`, `cache_dir` and `**kwargs`.

The destination argument in the `resample` method in code 3.29, is where the `AreaDefinition` object is provided. This tells what the desired result of the resampling is. The `resampler` argument specifies the resampling algorithm, which was discussed in section 2.5.2 at page 24. Caching directory is added to the `cache_dir` argument. When this argument is set, caching is enabled for resampling (explained in subsection 2.8.2). The `**kwargs` argument is added in the end to allow for additional arguments to be added, without changing the `resample_to_eqc_area` method.

To make sure that the returned Scene is of the class `SceneExt`, the `resample` method is overwritten. The new `resample` method calls on the original `resample` method, and creates an object of the `SceneExt` class out of the returned `Scene` object. The object of the `SceneExt` class is what is being returned by the new `resample` method.

3.5.3 Combining

When every Scene has been resampled, the next process executed is combining the Scene objects into one combined Scene with imagery of the entire earth. As mentioned in subsection 2.5.3 at page 28, Satpy is not providing functionality to combine satellite imagery from different locations. This means that the algorithm must be made manually. The theory of this algorithm is explained in subsection 2.5.3.

In contrast to Scenes of different locations, combining Scenes with the same location is supported by Satpy. This is being done with the `MulitScene` object's `blend` method. The `blend` method combines every grouped bands (mentioned in subsection 3.5.2) by executing a specified function on the group of bands. The `blend` method is being utilized in this software, by creating a function that combines geostationary imagery, at different longitudes. A method named `combine` has been made in the `MultiSceneExt` class, which utilizes the `blend` method to combine the different Scenes used:

Code 3.30: domains/processing/multiscene_ext. Part of the combine method in MultiSceneExt, where the blend method is called

```
    eqc_blend = EqcBlend(latitude_range=(-max_latitude, ...
        max_latitude))
    combined_scn = self.blend(eqc_blend)
```

As shown in code 3.30 above, the `blend` method at line 126 calls directly on the `EqcBlend` class from line 127. Calling on a class like this executes its `__call__` method. The reason why a class was made, instead of a function, is because it allows for greater code structure. The `__call__` method is what starts the combining algorithm explained in this section.

As shown at line 126 in code 3.30, the `EqcBlend` class is created with an argument, specifying the latitude range. This range is set to $(-70, 70)$, which represents the max latitude explained in subsection 2.1.3 at page 14. Data outside of this latitude range is excluded in the final combined Scene.

Because this software is created with python, speed is a concern when combining the satellite imagery. To speed up the process, the combining algorithm is being parallelized. Parallelism is well suited for image processing like this, as images easily can be split into multiple smaller images. Having every CPU processing one image part each speeds up the process by a great deal. As all of these CPUs are working together to create a single image, a matrix of shared memory is being made. Because the image parts are separated and do not overlap anywhere, mutex locks are not necessary when writing to the shared memory. Mutex locks are being used in this project though, as python is not giving great enough control of the memory. This results in a great deal of redundant processing.

The combining algorithm contains multiple consecutive steps:

1. Create empty shared memory matrix with the desired resolution of the final product.
2. Create longitude sections on map based on longitude differences from satellites.
3. Separate intersections into their own longitude sections.
4. Prepare the data for writing, by creating map portions.
5. Write each map portion to the shared memory matrix.
6. Create DataArray object from shared memory matrix.

As shown above, the algorithm for combining satellite images is quite complex. How each step in the algorithm above is implemented is described in detail below.

Create shared memory matrix

The first process that is done in the algorithm is the creation of the shared memory matrix. As mentioned, this matrix is representing the final image that is being created of the given band. The resolution of this matrix is therefore of the same resolution as the final product, which is specified when executing the program through the CLI. Before the matrix can be created, the length and width must be known. These lengths is calculated with the following equation:

$$axis_{length} = \frac{axis_{degree\_count}}{axis_{delta\_step}} \tag{3.1}$$

The $axis_{degree\_count}$ in equation 3.1 above is the number of degrees on that axis (e.g. 360 for longitude), while $axis_{delta\_step}$ is the number of degrees between each pixel in that axis. The $axis_{delta\_step}$ is found from the datasets of the bands provided. All datasets provides data of the locations of its pixels. As some inaccuracies occur in the datasets, due to inaccurate float values, the biggest delta step found for the given axis is being used. This makes sure that every pixel in the matrix created has a pixel associated to it. The method for creating the shared memory matrix is shown below:

Code 3.31: domains/processing/eqc_blend. Method for creating shared matrix

```
    def __init_shared_earth_array(self) -> ...
        (shared_memory.SharedMemory, np.ndarray):
        size = np.dtype(self.__data_type).itemsize * ...
            np.prod(self.lat_len * self.lon_len)
        shm = shared_memory.SharedMemory(create=True, ...
            size=size)
        dst = np.ndarray(self.__shape, self.__data_type, ...
            buffer=shm.buf)
        dst[:] = np.nan
        self.shared_earth_array, self.__earth_array = shm, ...
            dst
```

The method above in code 3.31 shows that the shared memory matrix is created utilizing multiple properties. These properties are in the same way as the $axis_{length}$ from equation 3.1, created from the datasets of the bands that are being combined. Starting at line 221, the size of the matrix is being calculated, using its desired dimensions multiplied with the size of the data type that is being used. The shared memory is then being created on line 222, using the calculated size. This allocates a buffer of the desired size. A variable pointing to this buffer is then created at line 223. As the data is not initiated yet, every pixel is set to `np.nan`, which is the value used by Satpy for pixels without any value. Finally, the shared memory matrix and the variable pointing to it is returned. These variables are then stored as attributes in the object at line 225.

Due to the way the shared memory works, it is not collected by the garbage collector. It must therefore be freed manually. To make sure that this is done whenever the shared memory is not being used anymore, the `__del__` method of the `EqcBlend` class is being utilized:

Code 3.32: domains/processing/eqc_blend. EqcBlend class's ___del___ method.

```
    def __del__(self):
        self.shared_earth_array.close()
        self.shared_earth_array.unlink()
```

The `__del__` method shown in code 3.32 is a special method that is being run whenever the associated object is being garbage collected. By freeing the shared memory in the `__del__` method, the shared memory is freed when the object is not being used anymore. Freeing the shared memory is done by first calling the `close` method, which closes access to it, then by calling the `unlink` method, which starts the process of removing the object from memory.

Create longitude sections

After creating the shared memory matrix, the process of creating sections for each Scene's dataset is executed. As explained in subsection 2.5.3, the satellites with longitudes closer to a location's longitude should have higher influence on the final result at that location. Because of this, the middle between every two adjacent satellite's longitude is found. The longitudes found are then used as the edges between the satellites' sections. How the sections turns out is illustrated below:

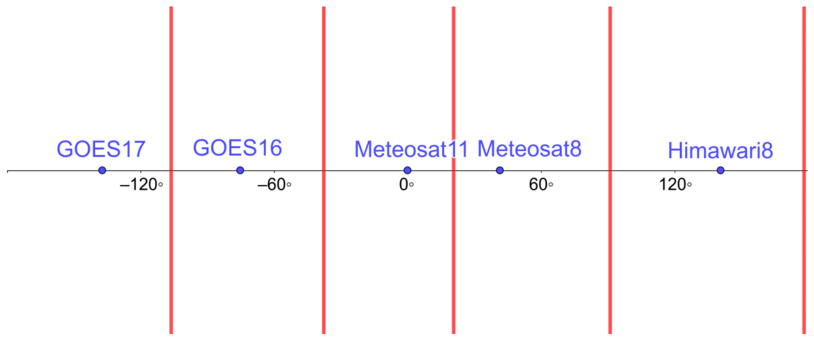


Figure 3.5: Illustration of how the longitude section is created. The red lines represents the edges of the sections. The degrees on the x-axis represents longitudes.

Figure 3.5 above shows an illustration of how the longitude sections turns out. The red lines represents the edges of the longitude sections. Every point in between any two adjacent edges is closest to the satellite in the same section.

The longitude sections are created as a separate class in the code, named `LongitudeSection`. As shown in its constructor, it accepts four arguments:

Code 3.33: domains/processing/eqc_blend. Constructor for the LongitudeSection class

```
20      def __init__(self,
21                   data_array1: xr.DataArray,
22                   from_longitude: float,
23                   to_longitude: float,
24                   *,
25                   data_array2: Optional[xr.DataArray] = ...
                         None
26                   ):
```

The constructor of the `LongitudeSection` class shown in code 3.33 above accepts two arguments of the `DataArray` class, and two floats, which represents longitudes. The longitudes is the edges of the longitude section, while `data_array1` at line 21 and `data_array2` at line 25 is the data that is associated with the longitude section. The `data_array2` argument is optional, as it is not wanted unless specified.

As mentioned earlier, a longitude section contains the data that is closest to the associated satellite, within a range of longitudes. This is only true though if the `LongitudeSection` object does not contain the `data_array2` attribute. If this attribute is set, a combination of the two `DataArray` objects will be used. How this is combined is explained in subsection 2.5.3 at page 28.

Separate intersections

As told above, the `LongitudeSection` objects might contain more than one `DataArray` object. Subsection 2.5.3 explains that locations close to the intersection (edge) between two satellites should gradually transition from one satellite's data to the other's. To prepare for this, `LongitudeSection` objects with two set `DataArray` attributes is being made. The new `LongitudeSection` objects is made so the middle of their longitude ranges is exactly where the intersections is.

Combining the datasets is not being done in the `LongitudeSection` class. There is multiple reasons for this. One is that it is faster to combine the datasets when writing to the shared memory matrix, as the writing happens in parallel. Iterating through the longitude section is also done one less time, as is not being iterated through in the `LongitudeSection`. Another benefit with combining when writing, is that the data arrays can be removed from memory, right after it has been loaded.

The `DataArray` classes uses an underlying library named *Dask*. Dask is a library that is useful for storing big amounts of data. Instead of storing the whole dataset in memory, it stores it on the hard drive. This reduces the memory usage of the program. A downside with storing the dataset on the hard drive, is that it takes longer time to load. The loading of each dataset is therefore only done once, to reduce execution time.

Preparing data for writing

After creating the longitude sections, the process of adding them to the shared memory matrix is starting. This starts off by preparing the data.

The data is prepared by creating objects of the `MapPortion` class. The `MapPortion` class contains geographical data that is going to be written to the shared memory matrix. One `MapPortion` is containing data for only one satellites dataset, with lists with data about its longitudes and latitudes. The constructor for the `MapPortion` class is shown below:

Code 3.34: domains/processing/eqc_blend. Constructor for the MapPortion class

```
    def __init__(self, values: np.ndarray, lon_list: ...
        list[float], lat_list: list[float],
                 lon_indexes: list[int], lat_indexes: ...
                     list[int]):
```

The **`values`** argument shown in code 3.34 above is the data of the associated satellite's dataset. The data has been loaded from the memory here, as the process of adding the data has started. The longitudes and latitudes of the values are provided by the **`lon_list`** and **`lat_list`** arguments. Which data in the **`values`** attribute are being used, is described by the **`lon_indexes`** and **`lat_indexes`** arguments. These arguments provides the indexes to the data in the **`values`** attribute that is being applicable to the associated `MapPortion` object.

The indexes in **`lon_list`** and **`lat_list`** are created from specified ranges. The **`lon_list`** uses the range from a `LongitudeSection` object, while the **`lat_list`** uses the range specified when constructing the `EqcBlend` object. A method has been made, which receives a list of indexes from a list of axis values, within a specified range:

Code 3.35: domains/processing/eqc_blend. Method in EqcBlend class that creates list of indexes for axis

```
    def __get_indexes_from_axis(self, axis_values: ...
        list[float], from_axis_val: float, to_axis_val: float,
                                axis_helper: Type[AxisHelper],
                                edge_size: int = 0,
                                max_length: Optional[int] ...
                                    = None) -> list[int]:
```

The method shown in code 3.35 above, accepts multiple arguments: some required and some optional. The first required argument is **`axis_values`**.

This is a list of axis values in degrees, which the method is using for finding the indexes. All axis values in between the value of `from_axis_val` and `to_axis_val` are wanted as the return value of the method. Depending on what axis the values are of, different functions are needed for calculations. These functions are collected in the `axis_helper` argument of the `AxisHelper` class. The `AxisHelper` class is a superclass which requires its subclasses to implement different methods for calculations on an axis.

The two optional arguments is `edge_size` and `max_length`. The `edge_size` is adding additional indexes to each side of the final indexes. This is useful, as it fixes issues related to imprecise floats. In contrast to the `edge_size` argument, `max_length` is used for specifying the maximum length of the returned indexes. It reduces equally on both sides, which leads to the returned index list having the same center, as without the reduction.

Using the `MapPortion` class instead of the `LongitudeSection` class when writing, is beneficial, as it contains more specific information about the data that is going to be written, than the `LongitudeSection` class does. This leads to the process of writing the data being simpler, which reduces the complexity when writing. Another benefit is that the `MapPortion` objects can easily be split into smaller parts, by changing the index lists.

Splitting the `MapPortion` objects is beneficial, as it creates equal portions to handle for each CPU. By having every CPU available working on datasets of the same size, the parallelization is being optimized. As the CPUs have datasets of the same size, they are finishing at almost the same time. This reduces the overall time spent writing to the shared memory.

Splitting a `MapPortion` is done with calling the following method:

Code 3.36: domains/processing/eqc_blend. Calling method in MapPortion that split it by the latitude axis

```
    map_portion_lists[index] = ...
        map_portion.split_by_lat_axis(CPU_COUNT)
```

With the `MapPortion` class, splitting by the latitude axis can be done with calling one method, as shown in code 3.36 above. This method accepts one argument which tells how many parts the `MapPortion` is going to be split to. This is the number of CPUs in this case. The method is shown in the code below:

Code 3.37: domains/processing/eqc_blend. Method in MapPortion that split it by the latitude axis

```
101     def split_by_lat_axis(self, count: int) -> ...
            list["MapPortion"]:
102         lat_indexes_list = ...
                ListHelper.split_list(self.lat_indexes, count)
103         map_portions = ...
                [self.__copy(lat_indexes=lat_indexes) for ...
                lat_indexes in lat_indexes_list]
104         return map_portions
```

As shown in code 3.37 above, splitting a `MapPortion` object by the latitude axis is simple, because it is only the indexes that needs to change. What happens is that the list of the latitude indexes (`lat_indexes` attribute) is being split into parts of roughly the same size at line 102. Each list of indexes is then being used together with a copy of the current `MapPortion` object, to create a new `MapPortion` with less latitude data. The method is ending by returning all of the newly created `MapPortion` objects.

From code 3.36 it is noticeable that the split `MapPortion` objects is being assigned to a list named `map_portion_lists`. This is a list, containing two lists of the `MapPortion` object. The first index is for the `MapPortion` objects created from the `data_array1` attribute in the `LongitudeSection` class, while the second index is for the optional `data_array2` attribute. By creating this list, the relationship between the data arrays in the `LongitudeSection` is kept, which is needed when combining the values. The `MapPortion` objects for every related data array is returned together as a tuple, as shown below:

Code 3.38: domains/processing/eqc_blend. Line showing how the related MapPortion objects is returned as tuples.

```
329     return [tuple(map_tuples) for map_tuples in ...
            zip(*map_portion_lists)]
```

The code in 3.38 above shows how related `MapPortion` objects of the same geographical location is combined into tuples. If there is only one

MapPortion for the given geographical location, the second value is set to None.

Write map portions

Before the MapPortion objects is written to the shared memory matrix, the processes that does so is being created:

Code 3.39: domains/processing/eqc_blend. Creating processes for writing

```
375        processes = [
376            mp.Process(
377                target=self.__add_value_from_map_portions,
378                args=(map_portion1, map_portion2)
379            ) for map_portion1, map_portion2 in map_portions_list
380        ]
```

Code 3.39 above shows how the processes is being created. One process is created for every tuple of MapPortion objects. The amount of tuples is as mentioned, equal to the number of CPUs available. What method, as well as its arguments, that the CPUs are going to run is specified in the target and args arguments in line 377-378. With the processes created, the following code is being executed:

Code 3.40: domains/processing/eqc_blend. Process methods

```
381        for methodname in ["start", "join", "close"]:
382            for process in processes:
383                getattr(process, methodname)()
```

The Process objects created does three things in their lifespan. As shown in code 3.40 above, every process starts off by running their delegated jobs, when the start method is being called. After every process has started to run their jobs, the join method is called on every process. The join method waits for the process to finish its job, before it lets the program proceed. Lastly, when every process has finished their job, the close method is being called on every process, which closes the processes.

As mentioned, when the `start` methods of the processes is being called, they start to run their designated tasks. It is noticeable in code 3.39 that a method named `__add_value_from_map_portions` being run by every process. This method is shown below:

Code 3.41: domains/processing/eqc_blend. Method that adds values from MapPortion objects to the shared memory matrix.

```
367      def __add_value_from_map_portions(self, map_portion1: ...
             MapPortion, map_portion2: Optional[MapPortion]) -> ...
             None:
368          if map_portion2 is None:
369              self.__add_value_from_single_map_portion( ...
                     map_portion1)
370          else:
371              self.__add_value_from_two_map_portions( ...
                     map_portion1, map_portion2)
```

As shown in code 3.41 above, the code is executed differently when there is two `MapPortion` objects to be added, than if there is just one. If there is one `MapPortion`, method `__add_value_from_single_map_portion` on line 369 is executed, while if there is two `MapPortion` objects, the `__add_value_from_two_map_portions` method on line 371 is executed. The difference between these two methods, is that the method that requires two `MapPortion` objects, is combining the values. The value used is gradually changed from the first `MapPortion` to the second, as the longitude moves towards east.

The most basic of the two methods that writes to the shared memory matrix is shown in the following code:

Code 3.42: domains/processing/eqc_blend. Method that adds single MapPortion to the shared memory matrix

```
331 def __add_value_from_single_map_portion(self, map_portion: ...
         MapPortion) -> None:
332     earth_array_lat_range = ...
             self.__earth_array_latitude_index_range
333     for lat_index in map_portion.lat_indexes:
334         for lon_index in map_portion.lon_indexes:
335             value = map_portion.values[lat_index][lon_index]
336             new_lon_index, new_lat_index = ...
                    self.__translate_coords_to_earth_array_indexes( ..

337                 (map_portion.lon_list[lon_index], ...
                        map_portion.lat_list[lat_index])
338             )
339             if np.isnan(value) or not ...
                    (earth_array_lat_range[0] ≤ new_lat_index ≤ ...
                    earth_array_lat_range[1]):
340                 continue
341             self.__earth_array[ new_lat_index][new_lon_index] ...
                    = value
```

The method shown in code 3.42 above shows the method used to add a single `MapPortion`. The first thing that happens (line 332) is that the indexes of the latitude edges for the shared matrix (`__earth_array`) is retrieved. These indexes is found to provide a clean cut at the latitude edges. Without these indexes, rounded floats may cause an inconsistent cut.

The next thing that happens is that every latitude index and longitude index is iterated over at line 333-334. By iterating over every longitude indexes for every latitude index iterated over, every combination of the two index lists is iterated through.

With the indexes obtained from `lat_index` and `lon_index`, an associated value is retrieved on line 335. This is the value that is going to be written to the shared matrix in the given iteration.

Continuing on line 336-338, the longitude and latitude is being used to find the associated indexes in the shared matrix. This is done by the `__translate_coords_to_earth_array_indexes` method, which uses calculations based on the shared matrix dimensions and the axis's properties.

Before the value is added to the shared matrix, some final checks is first executed at line 339. If the value is of the `nan` type (is empty), or the the latitude index is outside of the range found at line 332, the value is not being added. If none of these expressions is true, the code writes the value to the shared memory at line 341, using the calculated indexes.

The method for writing two `MapPortion` objects to the shared memory matrix is a bit more complex than the method for adding one `MapPortion`. As shown below, the iteration process starts off a bit different:

Code 3.43: domains/processing/eqc_blend. Iteration part in method that adds two MapPortion objects to the shared memory matrix.

```
        for lat_index1, lat_index2 in ...
            zip(map_portion1.lat_indexes, ...
            map_portion2.lat_indexes):
            for i, (lon_index1, lon_index2) in ...
                enumerate(zip(map_portion1.lon_indexes, ...
                map_portion2.lon_indexes)):
```

Code 3.43 above, shows that iterating is done a bit differently when two `MapPortion` objects is being added. The indexes of the two `MapPortion` objects, is iterated over simultaneously. The iterations of the longitude indexes is also being counted, in the `i` variable. This is being used later, when combining the two values received from the `MapPortion` objects.

The indexes of the shared matrix is then being found using the `__translate_coords_to_earth_array_indexes` method, in the same way as in line 336 in code 3.42. The calculated indexes returned from this method is the same independent of which `MapPortion` object's longitude and latitude index is being used.

The part that makes this method differ the most from the method that adds one `MapPortion`, is the part where the two values are being combined:

Code 3.44: domains/processing/eqc_blend. Value calculation in method that adds two MapPortion objects to the shared memory matrix.

```
355     values = [map_portion1.values[lat_index1][lon_index1], ...
            map_portion2.values[lat_index2][lon_index2]]
356     values_filtered = list(filter(lambda val: not ...
            np.isnan(val), values))
357     if len(values_filtered) == 0:
358         continue
359     elif len(values_filtered) == 1:
360         value = values_filtered[0]
361     else:
362         progress = ((i / lon_indexes_length) - 0.5) * ...
                self.merge_intensity
363         weight = MathHelper.sigmoid(progress)
364         value = values_filtered[0] * (1 - weight) + ...
                values_filtered[1] * weight
```

As shown at line 355 in code 3.44 above, the values of the two `MapPortion` objects is retrieved in the same way as with the method for one `MapPortion`. The longitude and latitude indexes for both the objects is used with their associated `values` attribute, which returns the desired value. The list of these values is then filtered at line 356, which removes any value that is `nan`.

How the final value is created depends on how many values there are in the `values_filtered` variable from line 356. If there is zero values, the expression on line 357 becomes true, and line 358 ends up continuing to the next iteration.

If the `values_filtered` variable contains one variable, the expression on line 359 becomes true. This executes line 360, which sets the final value to the only available value. This is useful for locations where only one of the datasets (satellites) have data.

If both the expressions at line 357 and 359 is false, the code in line 362-364 is being used to get the final value. This means that `values_filtered` is containing two values. The code executed in this case uses the theory of merging two values explained in subsection 2.5.3. This subsection tells that a function named `sigmoid` is being used to calculate the weight of each `MapPortion` object's value. It uses the distance from the center of the area's longitudes. When the longitude is west to the center, the value from map_portion1 is weighted the most. If it is east to the center, the value from map_portion2 is weighted the most.

A variable named progress is made at line 362. This calculates the relative distance from the center, and ranges from $-x$ to x. The progress is further used with the *sigmoid* function in line 363, which calculates the weight for the value from map_portion2. This weight is somewhere between 0 and 1. It is then used in line 364 to calculate the final value. This value is then added to the shared memory matrix in the same way as in line 341 in code 3.42.

Create DataArray

When all the processes has finished writing the values in the MapPortion objects to the shared memory matrix, the matrix is not being written to anymore. This means that the dataset of the Satpy Scenes with the given frequency range has been combined. The last thing that needs to be done before the blend method of the MultiSceneExt is finished, is to return the shared memory matrix as a DataArray object. This is to satisfy the return type of functions that may be added to the blend method. Useful information about the data is also being provided by the DataArray class. This includes scanning times, geographical data and projection information. How the data array is created is shown in the method below:

Code 3.45: domains/processing/eqc_blend. Method that creates a DataArray class from the EqcBlend class

```
    def as_data_array(self) -> xr.DataArray:
        start_time, end_time = self.time_range
        attrs = {
            "start_time": start_time,
            "end_time":  end_time,
            "area": self.area_def
        }
        return xr.DataArray(
            data=self.__earth_array.copy(),
            dims=["y", "x"],
            coords=self.coords,
            attrs=attrs
        )
```

The method shown above in code 3.45 is used for creating a `DataArray` object from the associated `EqcBlend` object. Line 407 starts of by getting the start and end scanning times of the satellites that has provided the data. These times are used together with a object of the `AreaDefinition` class, to create a dictionary of attributes in line 408-412. The object of the `AreaDefinition` class, is being created, based on the data's projection, as well as other data that describes the area.

Continuing on line 413-418, the `DataArray` object is created, with some arguments. The first argument added is the `data` argument on line 414. The variable provided to this argument is a copy of the `__earth_array` attribute, which points to the buffer of the shared memory matrix. The reason why this is copied, is because the shared memory matrix is being freed when the `EqcBlend` object is not being used anymore. This leads to the data that the `__earth_array` attribute points to being removed. This issue is avoided by copying the matrix.

A matrix containing the coordinates of the data is being added at line 416 to the `coords` argument. The coordinates needs to be provided in metric form, instead of in degrees. These coordinates is calculated in the `coords` property.

The last two arguments provided is the `dims` and `attrs`. The `dims` argument is given a list with names of the two dimensions the data is in. Lastly the `attrs` argument is given the object of the `AreaDefinition` class, created in line 408-412.

3.5.4 Cloud extracting

Extracting the clouds from the combined Scene which creation is shown in subsection 3.5.3, is as mentioned in section 2.6 a complex task. How this is done is therefore simplified to the solution presented in subsection 2.6.4 at page 33. This solution is algorithmic, and is based on finding every value in a specified boundary for imagery of a specific frequency. Even though only one frequency is being used to extract cloud data, the program is designed to accept more, to support potential future improvements. The method where the clouds are getting extracted is shown below:

Code 3.46: domains/processing/scene_ext. Method that creates image of clouds from a SceneExt object

```
101     def create_cloud_image(self, frequencies: list[float]) ...
            -> XRImage:
102         compositor = CloudCompositor(name="clouds", ...
                transition_min=230.0, transition_max=298.15, ...
                transition_gamma=1.5)
103         composite = compositor([self[frequency] for ...
                frequency in frequencies])
104         return to_image(composite)
```

The method shown above in code 3.46 starts of by creating an object of the `CloudCompositor` class, as shown in line 102. Its `transition_min` and `transition_max` arguments decides what values should be considered as clouds, and not. Values below `transition_min` is being created as clouds, while value above `transition_max` is being created as cloud free. Values in between `transition_min` and `transition_max` is created as a partially cloudy. The transition arguments provided to the `CloudCompositor` constructor is chosen based on the values that gave the best visual result. The image used as reference is obtained from The Meteo Company at sat24.com. The image used is of the same frequency that is being used in this project:

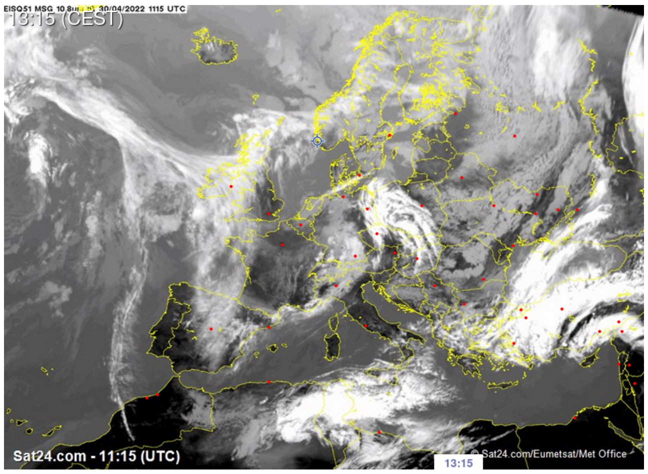


Figure 3.6: Satellite image of Europe at 10.8μm. Obtained from sat24.com. [8]

Figure 3.6 shows the image of Europe taken at 10.8μm, which is used as reference when extracting the cloud data. As mentioned in subsection 2.6.4

at page 33, the 10.8μm frequency is not giving an accurate image of the clouds, as some clouds is not detected at this frequency. It is used as a reference though, because of the fact that it is of the same frequency used in this project.

When the `CloudCompositor` object has been created on line 102 in code 3.46, it is being called on at line 103 to create a composite. This composite includes the cloud data, generated from bands of the specified frequency.

To create the `XRImage` object that is being returned from the method, the `to_image` method is being used. This is shown in code 3.46 at line 104. The `XRImage` object is useful, as it contains methods for saving the object to various formats. The lines of code that shows this is in the following code:

Code 3.47: domains/product/product_creator. Saving of XRImage to various file formats.

```
140     for file_format in file_formats:
141         if os.path.exists(filepath := ...
                self.__get_imagedata_path_for_format(file_format)):
142             continue
143         img.save(filepath)
```

As shown in code 3.47 above, the `img` variable of the `XRImage` class is easily being saved to different file formats. The `img` variable above is the `XRImage` object containing the cloud data. Line 143 shows the final process that is being done before the data image product has been created. The image is at this line being stored in the GeoTIFF format, and optionally the PNG format, depending on which products are desired.

Chapter 4

Results and discussion

This chapter presents the results of the project, and discusses it. How the various processes explained in chapter 3 affects the final product is shown. Discussing what could be done to improve the various results is also done.

4.1 Image products

In this section the final image products are being presented and discussed. This includes both the data image and the visual image. How the processing domain process the image is also shown, and compared to images without the various features.

4.1 Image products

The data image from 30 Apr 2022, 11:15 UTC is shown below, where clouds is represented in black:

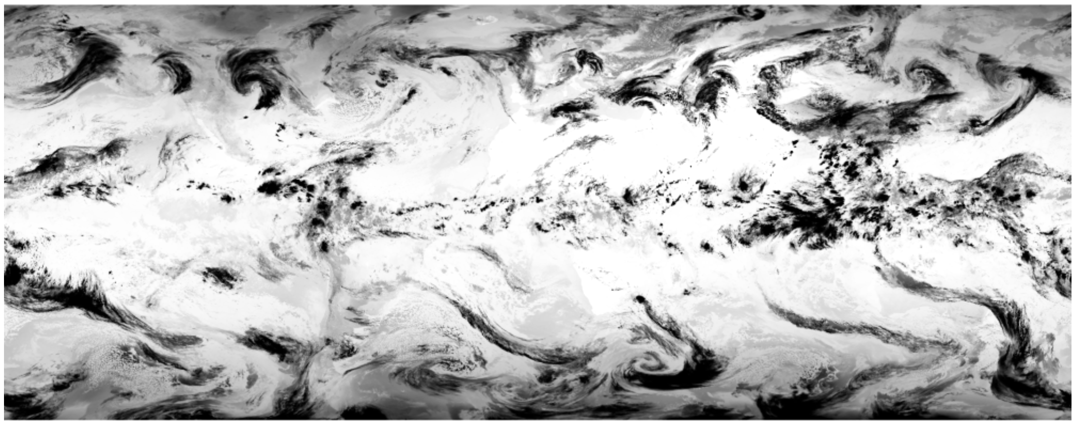


Figure 4.1: World-wide data image of clouds. Clouds is represented by black.

The visual image from 30 Apr 2022, 11:15 UTC is presented below:

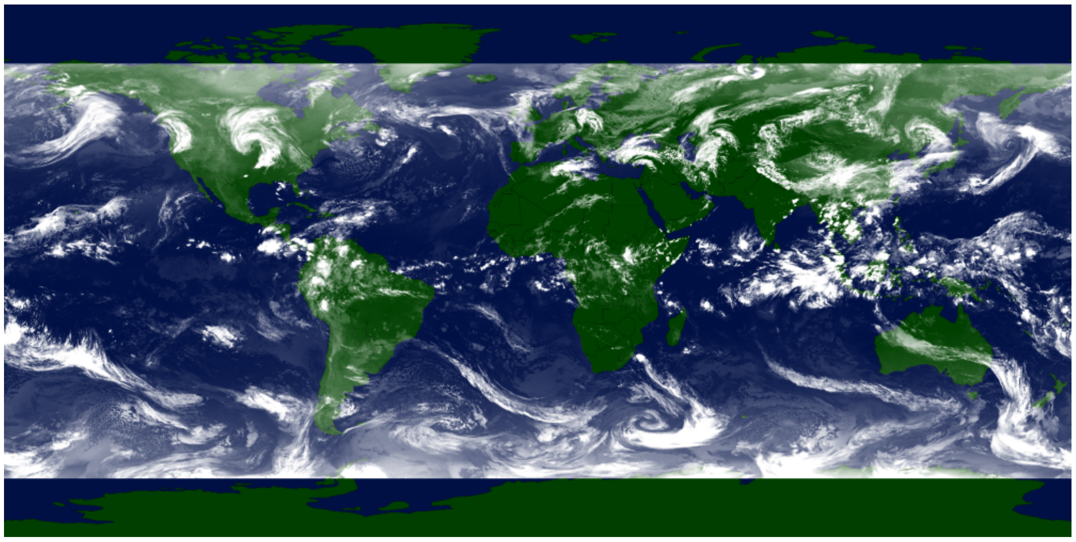
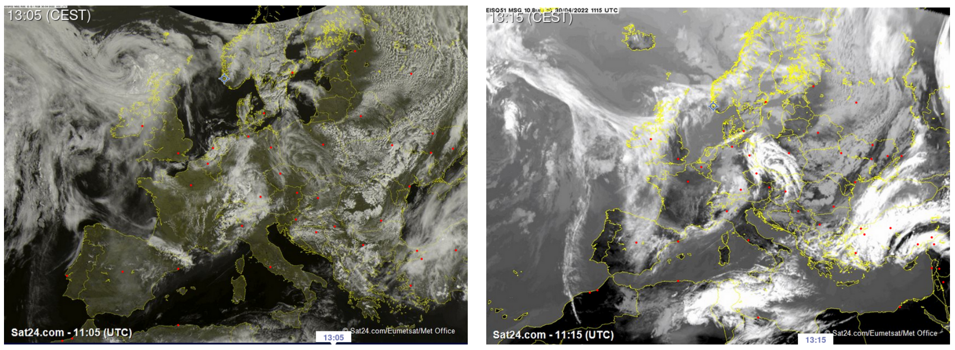


Figure 4.2: World-wide visual image of the clouds.

As shown in the figure 4.1 and 4.2 on the previous page, the images do contain cloud data. The clouds are also spread around the whole globe, with no clear difference between land and ocean. These two images are discussed further in the following subsections.

4.1.1 Cloud accuracy

As mentioned, figure 4.1 and 4.2 on page 88 do contain cloud data. A comparison of the accuracy provided by this projects software is being done with images obtained from The Meteo Company at sat24.com [8]:



(a) Visible light spectrum **(b)** Infrared 10.8μm

Figure 4.3: Images of Europe from sat24.com.

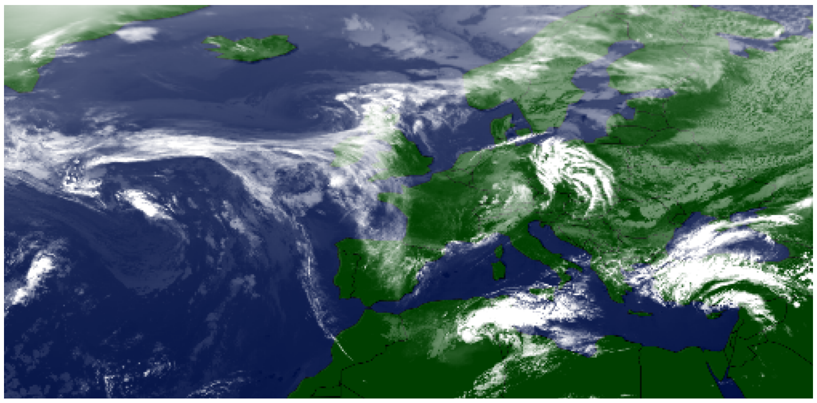


Figure 4.4: Zoomed in part of Europe from figure 4.2.

Using figure 4.3 and 4.4, the product's accuracy can be discussed. Note that figure 4.4 has a different projection than the images in figure 4.3.

It is noticeable that there is multiple similarities between the image in the visible spectrum in figure 4.3a and the image in figure 4.4. The clouds over Germany and Poland can be shown on both images. There is some differences in the intensities though. The clouds over Poland appears very intense in figure 4.4, while they are barely visible in figure 4.3a. The same phenomena is happening in the ocean west for Great Britain. Figure 4.3a shows that it is very cloudy there, while figure 4.4 only shows high intensity in the middle of the cloud.

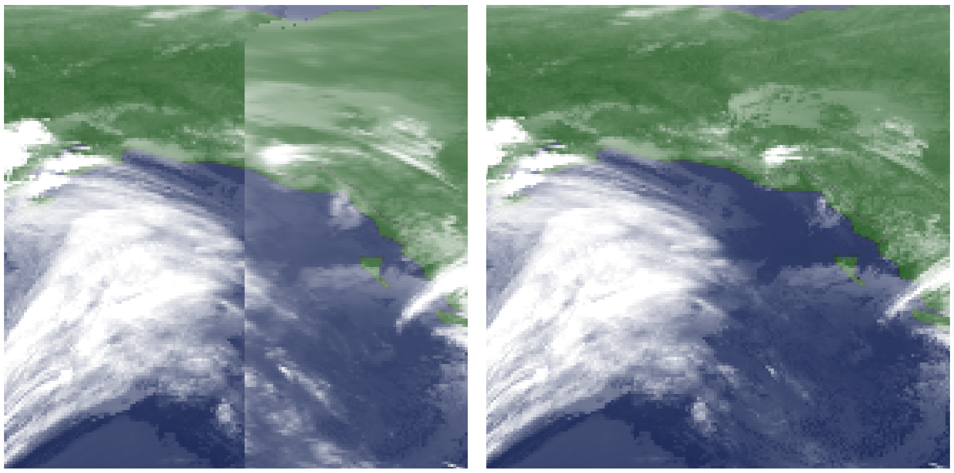
The differences between 4.3a and 4.4 comes from the simplification of cloud detection mentioned in subsection 2.6.4. As mentioned, the only frequencies used is the ones close to $10.6\mu m$. This frequency contains similar data as figure 4.3b does, with its $10.8\mu m$ imagery.

Comparing figure 4.4 with the infrared image in figure 4.3b makes it clear that the similarities between these images is greater than the similarities between the images in figure 4.4 and 4.3a. All the clouds in figure 4.3a can be found in figure 4.4, (with some distortion) with similar intensities. This shows that the cloud detection is working as intended.

As mentioned, the cloud accuracy is not perfect, when comparing to the visible light spectrum in figure 4.3a. Improving this would be one of priorities if lack of time was not an issue. The difficulties of finding accurate cloud products is as explained in section 2.6 comprehensive and time consuming, as clouds behaves differently based on multiple factors.

4.1.2 Combining

Subsection 3.5.3 at page 69 shows that combining the imagery from the utilized satellites is a extensive process. The figures on page 88, which is showing the cloud products, do not show any clear sign of the intersection between the different satellites' imagery. This is because of the way intersections between satellite imagery is handled. The images below shows the effect of the intersection handling:



(a) Hard transition **(b)** Smooth transition

Figure 4.5: Comparison between intersections with and without smooth transition

As shown in figure 4.5 above, the way that the intersections between satellite imagery is handled, makes the transition smooth. Figure 4.5a shows how the intersection is without the smooth transition, while figure 4.5b shows how it is with. The intersection is not being noticeable with smooth transition, as shown in figure 4.5b. Accurate data is also being provided, even though the transition blends the intersection. This is beneficial, as the accuracy of the data is of great importance.

4.1.3 GOES-17 noise

As mentioned in subsection 2.1.5 at page 16, the GOES-17 satellite suffers from some issues, which at some times leads to noise in its imagery:

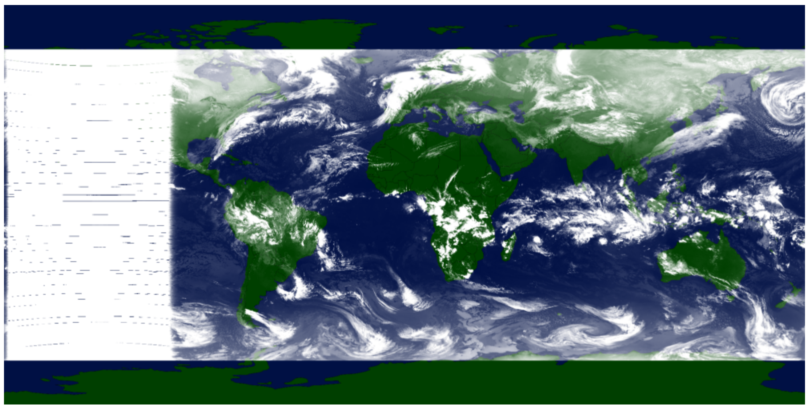


Figure 4.6: Visual image, showing noise from GOES-17.

As shown in the left side of figure 4.6 above, the noise from GOES-17, makes the whole area white. A solution to this problem could be to not use the imagery from GOES-17 when it is noisy, and instead depend on Himawari 8 and GOES-16 to cover its area. This is something that could be implemented in the future.

Subsection 2.1.5 mentions that the GOES-17 satellite is being replaced in early 2023, by the functioning GOES-18 satellite. This means that this problem is only occurring until early 2023. If the software is being used in production before then, the solution suggested above should be implemented. If not used until then, it would not be as necessary to implement this solution. An argument for implementing this solution, is that it would help with other potential satellites that could observe noise. Noise could occur from other sources than a defect satellite, as unforeseen events might happen.

4.2 Video product

The video product is made from the visual images and is therefore having similar results as the ones mentioned in section 4.1. Additional observations that can be done with the video product is being discussed in this section. A link to a YouTube video of the generated video product can be found in appendix B.2. This video contains data starting from 13 December 2020, 19:00 UTC, going 24 hours forward. It contains 4 visual images for every hour of data, which is presented with 15 frames per second.

4.2.1 Cloud movements

The video product shown on YouTube is showing the movement of the clouds, with the background of the earth. As wanted, the clouds is moving in a natural way. An example of this is the cyclone forming in the pacific, west of Canada. It is also noticeable that clouds tend to move eastwards, which is as expected, due to the earth's rotation (The Coriolis Effect). These observations reinforces the claims from subsection 4.1.1, that says the cloud detection process is detecting actual clouds.

4.2.2 Day night cycle

By studying the provided YouTube video, it is noticeable that day and night affects the cloud detection to some degree. Two seconds into the video, the intensity of the clouds over Africa is being lowered. This is because it is changing from day to night time. This is another result originating from the decision of only using the $10.6\mu m$ frequency for cloud detection (see subsection 2.6.4). This effect would therefore be lowered by using an enhanced cloud detection algorithm.

4.2.3 Consistent frame change

The observations done of the video is showing that the video product itself is working as intended. The movement of the clouds is changing gradually, to some degree, which shows that the video have a somewhat consistent time change between each provided image. Some imperfections are noticeable though. As mentioned in subsection 2.1.4 at page 15, the utilized satellites provides imagery in different intervals. By increasing the amount of images included for every hour of imagery in the video, the time between each provided image becomes close to consistent.

A video of 15 frames per second is not considered smooth. It is noticeable in the YouTube video of the video product that the video could benefit from being smoother. As mentioned, in subsection 2.1.4, four out of the five utilized satellites is providing imagery only four times per hour. This means that the images provided to the video for every hour of imagery cannot be increased for more than one of the satellites. This is not that helpful. Another solution to this problem could be to increase the speed of the video, which is not optimal either.

There exists software that could help with increasing the frames per second. Software like *DAIN* could be worth looking into, as it creating additional frames out of the existing ones [4]. This would lead to an increased number of frames shown per second, without speeding up the video.

4.3 Resource usage

Running the software created is a time consuming, as well as a computational heavy task. The runtime of the software do also increase exponentially as the resolution increases, because of the fact that it is a two dimensional image. The big runtime is therefore a great concern if the product is created with the best resolution available, of $2km^2$ per pixel (see table 2.1 at page 8).

Another concern is the memory usage of the software. As the data processed is high resolution images, storing them in memory takes up a lot of space. Running the software on hardware with to little memory will cause unwanted events to occur, depending on which operating system is used. On Linux, the operating system is stepping in and killing the process, which leads to the program not being able to finish.

The concern of the resource usage can be reduced in multiple ways. Some of these is discussed in this section.

4.3.1 Benchmark

To analyze the software's resource usage, a benchmark of the program has been run. Because of caching, the runtime of the program depends on multiple factors. If the data has been downloaded before, the process of downloading is being skipped. This does also apply to products, as it skips creating the product from scratch if any lower level product exists. None of these mentioned cached items is usually being used though, when creating a data image from scratch. What is being used though is caching of resampling calculations, which reduces the runtime of resampling. The figure below shows a benchmark of the most usual execution, where only the resampling cache is being used:

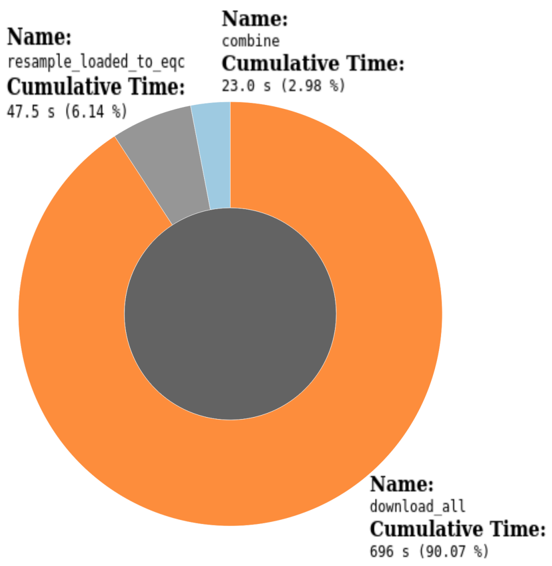


Figure 4.7: Benchmark of creating data image, where resampling cache is being used. The resolution used is $20km^2$.

Figure 4.7 above shows the runtimes of a execution of the program. These runtimes do vary some, as the network connection and CPU resource usage depend on multiple factors. Having a faster network connection or faster CPU would decrease the runtime.

It is noticeable in figure 4.7 above, that downloading the satellite imagery accounts for 90.07% of the runtime. This runtime does not increase as the resolution increases though, as the amount of data downloaded is only available in one resolution. As mentioned in subsection 2.3.2, the Meteosat satellites do not have accessible data separated by bands available. This leads to the files for Meteosat satellites being huge, which again leads to great download time.

The benchmark shown in figure 4.7 above, is made from the creation of a product with a resolution of $20km^2$. As mentioned, the runtime increases exponentially as the resolution gets better. The benchmark below shows the creation of a product with resolution of $2km^2$:

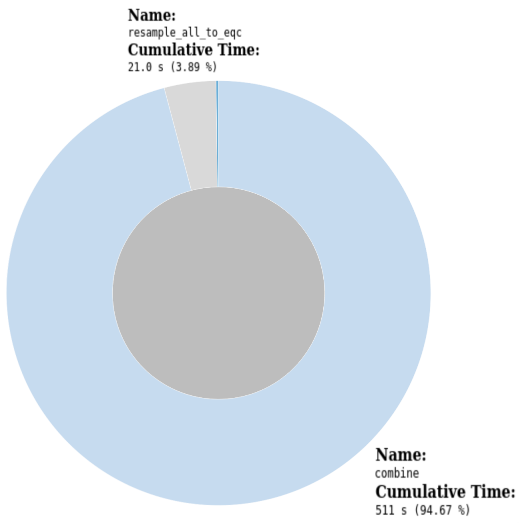


Figure 4.8: Benchmark of creating data image, where resampling, and downloading cache is being used. The resolution used is $2km^2$.

In contrast to the benchmark shown in figure 4.7, the benchmark in figure 4.8 above shows a runtime that uses more time on combining. This is because the better resolution of $2km^2$. Note that the satellite data is being cached here, which removes downloading from the runtime. It is shown in figure 4.8 that the combining accounts for 511 seconds and 94.67% of the runtime. This is an increase of 2120% relative to the 23 seconds of runtime that was gotten in 4.7. This shows that the combining method has longer runtime when the product is created with better resolution.

Digging deeper into the benchmark from figure 4.8 makes it clear that the parallelization in the combining method is not working as intended. The `wait` method that waits for other processors to finish, uses 80.55% (435 seconds) of the runtime. This may be because of the way that python handles the shared memory.

To decrease the runtime of the combine method, lower level programming languages with greater control of the memory could be used. An example of this is the programming language Rust. As this project is a proof of concept, this is not needed. Mapping out the possibilities and problems with creating world-wide cloud coverage imagery, is what is important in this project. Writing the program in a lower level programming language would be useful for the final product though. The runtime could also be lowered by creating an extension module in a lower level language for the combine process. This way, the runtime would be lowered, while the other code created in this project could still be used.

Chapter 5

Economic overview

When mapping out an economic overview, the software it self and the hardware it runs on is taken into account. This includes the economic consequences of downloading, processing and storing the data.

5.1 Direct expenses

The program created in this project does not have that many direct expenses, as it is only using free third party software. This includes all the Python packages and modules used, as well as the sources used for downloading. An expense that is tied to a single execution of the program is storage expenses. Storage must be provided for both the downloaded satellite data and the created products. If the downloaded data is not deleted after some time, the storage needed will increase proportionally for every execution. Where the data is stored also affect the expenses for storage. Self hosting increases the expenses stepwise, while hosting on the cloud increases it linearly.

5.2 Indirect expenses

Indirect expenses also affect the economy behind this product. There are multiple indirect expenses, depending on where the software is being executed. Running the software on self provided hardware leads to higher electricity expenses. The amount of processing power and memory available is also decreasing, as the software is both CPU and memory intensive when it processes the data. This could lead to the necessity of upgrading the hardware.

Downloading of the satellite data may also cause an indirect expense, as this decreases the bandwidth available for other tasks. Upgrading the network connection could be a solution to this, which would again increase expenses.

Running the software in the cloud leads to less indirect expenses. Cloud providers prices often processing based on CPU and memory usage. This leads to the execution of the software becoming more of a direct expense, instead of an indirect. This may be beneficial, as the expenses are tied directly to the processes that creates the expenses.

5.3 Caching

Using caching makes the software use less processing power, while the storage needed is increasing. Reducing the amount of caching being used could be done to lower the cost of storage. Old data could for example be deleted frequently. Storage is cheap though, which should be reflected when deciding how much of the cached data should be kept.

Chapter 6

Environmental accounting

Researching the environmental consequences of any product is important, as every action done has an impact on earth's climate. Being conservative with environmental emissions is more important than ever, with the rapid changes in the climate that is occurring.

6.1 Emissions

Environmental and economic consequences of software often come from the same sources. Both processing and storage of data, which have an economic impact (see chapter 5), also affect the environment. As mentioned, both processing and storage needs hardware that satisfy the required demands. This hardware is being produced in factories, which have a negative impact on the environment. Keeping the processing and storage needed for the software low, is therefore important. As discussed in section 2.8, the caching that is implemented in this project may help with lowering the hardware needed, which again leads to a lower environmental impact.

Caching is also reducing the energy needed to run the software. This is because the CPU requires a lot of energy, which makes it environmentally beneficial to reduce processing needed. In contrast to the CPU, the hard drives that the cached data is stored on does not require that much energy.

6.2 Benefits

The produced products that this software creates comes with some potential for environmental benefits. If other imagery than clouds are being produced, like imagery of gas emissions or forest fires, appropriate actions could be taken faster, which reduces the negative environmental impact. There exists a lot of software that creates products from satellite imagery though. This means that this software will not provide any new information. What this software could do though is to raise accessibility and awareness of the environment, through Time and Date's website. The scale of this website would make the products provided by the software easily accessible globally.

Chapter 7

Conclusion

World-wide satellite imagery of clouds in various formats are being made by the software created in this project. This does solve the given problem presented in section 1.1. The software is downloading satellite data from various satellites, combining them into one unified dataset, before it finally detect and extracts the clouds from the imagery. The global cloud coverage is presented as images containing only the cloud data, as well as visual images which includes a background of the earth behind the clouds. A video product of the cloud coverage may also be created, which shows the motion of the clouds.

Processing and combining the separate satellite images results in an image of the whole globe, with no clear sign of the individual satellite imagery. Detecting clouds accurately from satellite imagery proved to be a complex task though. Lack of time and expertise in this area lead to a simplified cloud detection algorithm, which detects clouds with a moderate accuracy. Time used downloading and processing is another weakness of the software. Having products with the most recent data available are one of the key focuses of this project. The project is a proof of concept though, which leads to the importance of discovering issues and possibilities being more important than the final product. Recreating the software in a lower level language would be beneficial, as it reduces the execution time, which leads to the possibility of having more updated products.

Bibliography

[1] Amazon. Noaa goes on aws. https://github.com/awslabs/open-data-docs/tree/main/docs/noaa/noaa-goes16, 2021.

[2] Amazon. Jma himawari-8. https://registry.opendata.aws/noaa-himawari/, 2022.

[3] Amazon. Noaa geostationary operational environmental satellites (goes) 16 & 17. https://registry.opendata.aws/noaa-goes/, 2022.

[4] Wenbo Bao, Wei-Sheng Lai, Chao Ma, Xiaoyun Zhang, Zhiyong Gao, and Ming-Hsuan Yang. Depth-aware video frame interpolation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[5] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[6] National Satellite Meteorological Center. Fengyun series satellites. https://fy4.nsmc.org.cn/nsmc/en/satellite/index.html, 2022.

[7] cloudflight. Detecting clouds with machine learning. https://www.cloudflight.io/en/project/detecting-clouds-with-machine-learning/, 2022.

[8] The Meteo Company. sat24. https://www.sat24.com, 2022.

[9] NASA Planetary Science Division. Earth - nasa solar system exploration. https://solarsystem.nasa.gov/planets/earth/in-depth/, 2018.

[10] ESA. Types of orbits. https://www.esa.int, 2020.

[11] ESA. Insat-3d. https://earth.esa.int/web/eoportal/satellite-missions/i/insat-3d, 2022.

[12] EUMENTSAT. Algorithm theoretical basis document for the cloud product processors of the nwc/geo. 01 2019.

[13] eumetsat. Eumetsat api. https://api.eumetsat.int, 2022.

[14] Eumetsat. How to access our data. https://www.eumetsat.int/access-our-data, 2022.

[15] Eumetsat. Metosat series. https://www.eumetsat.int/our-satellites/meteosat-series, 2022.

[16] GISGeography. What are map projections? (and why they are deceiving to us). https://gisgeography.com/map-projections/, 2021.

[17] Pytroll group. Satpy documentation. https://satpy.readthedocs.io, 2021.

[18] NWS. Satellites. https://www.weather.gov/about/satellites, 2022.

[19] National Oceanic and Atmospheric Administration. Clouds or snow? here are a few ways to tell the difference. https://www.nesdis.noaa.gov/news/clouds-or-snow-here-are-few-ways-tell-the-difference, 2018.

[20] National Oceanic and Atmospheric Administration. Goes-r series data book. 08 2019.

[21] National Oceanic and Atmospheric Administration. Goes-17 abi performance. https://www.goes-r.gov/users/GOES-17-ABI-Performance.html, 2022.

[22] Meteorological Satellite Center of JMA. Himawari-8/9 spacecraft overview. https://www.data.jma.go.jp, 2022.

[23] University of Twente. Itc satellite and sensor database. https://www.itc.nl/research/research-facilities/labs-resources/satellite-sensor-database/, 2022.

[24] World Meteorological Organization. Observing systems capability analysis and review tool. https://space.oscar.wmo.int/, 2022.

[25] Planetary. Coverage of a geostationary satellite at earth. https://www.planetary.org/space-images/coverage-of-a-geostationary, 2022.

[26] PROJ contributors. *PROJ coordinate transformation software library.* Open Source Geospatial Foundation, 2022.

[27] Martin Raspaud, David Hoese, Panu Lahtinen, Stephan Finkensieper, Gerrit Holl, Simon Proud, Adam Dybbroe, Andrea Meraner, Joleen Feltz, Xin Zhang, strandgren, Sauli Joro, William Roberts, Lars Ørum Rasmussen, BENR0, Jorge Humberto Bravo Méndez, Yufei Zhu, mherbertson, rdaruwala, Pierre de Buyl, Tommy Jasmin, Christian Kliche, Talfan Barnie, Eysteinn Sigurðsson, Sebastian Brodehl, R.K.Garcia, Thomas Leppelt, Taiga Tsukada, and ColinDuff. pytroll/satpy: Version 0.36.0 (2022/04/14), April 2022.

[28] Tomas Soler and David Eisemann. Determination of look angles to geostationary communication satellites. *Journal of Surveying Engineering*, 120:122, 08 1994.

[29] Sunex. Swir imageing optics. https://sunex.com/2021/02/17/swir/, 2021.

[30] Time and Date AS. Timeanddate. https://www.timeanddate.com/company/, 2022.

Attachment A

Email from The Norwegian Meteorological Institute

```
From: Trygve Aspnes <trygveas@met.no>
Subject: Bachelor thesis in collection of cloud data
Date: Thu, 17 Feb 2022 16:23 +0200
To: Joachim Andreassen <jo.andreassen@stud.uis.no>
```

Hey Joachim

It is no small task you have started on.

I am responsible for running various algorithms that do what you describe. These are algorithms that have been developed over more than 20 years.

So unfortunately I have no knowledge of the algorithms themselves.

For the geostationary data you use, you can register at nwcsaf.org and download software that can do this. The result comes in netCDF, but can relatively easily be stored as a GeoTIFF with pytroll / Satpy. NWCSAF can handle Meteosat and I'm pretty sure it can handle GOES 16/17 and Himawari 8 as well.

Email from The Norwegian Meteorological Institute

The disadvantage is that NWCSAF uses model data as input in GRIB format and I am not sure if this is easy to obtain. It can be run without model data as input, but then the quality will not be so good.

NWCSAF also has documentation that describes the various algorithms used.

Trygve Aspenes

Attachment B

External links

B.1 Source code

https://github.com/joffe97/worldwide_cloud_data

B.2 Video product demonstration

https://www.youtube.com/watch?v=zcYJeCNK26E

Attachment C

How to use

1. Add credentials to credentials.ini.dummy file.
2. Rename credentials.ini.dummy to credentials.ini.
3. Build the docker image

```
docker build wwclouds/ -t wwclouds
```

4. Run the docker image

   Note: The program is running with predefined arguments

```
docker run -it wwclouds:latest
```

5. Wait until the program is finished (this might take some time)
6. Get the container id by using the following command

```
docker container ls
```

7. Copy the products into a local directory

   Note: The correct container id must be used

```
docker cp ...
    <container_id>:/usr/src/wwclouds/data/products .
```