
West University of Timisoara

Faculty of Mathematics and Computer Science

Study Program:
Computer Science in English


# Master Dissertation


**COORDINATOR:**
Associate Prof. Marc Eduard Frîncu

**GRADUATE:**
Maria Minerva Vonica



Timişoara
2021

West University of Timişoara

Faculty of Mathematics and Computer Science

Study Program:
Computer Science in English

# Master Dissertation

# Name

**COORDINATOR:**
Associate Prof. Marc Eduard
Frîncu

**GRADUATE:**
Maria-Minerva Vonica

Timişoara
2021

# Chapter 1

# Application

## 1.1 Dataset

In order to build the dataset, we made use of the freely available data from the Landsat Archive, specifically from collection 1, level 1. This consists of data products generated from Landsat 8 Operational Land Imager/Thermal Infrared Sensor, Landsat 7 Enhanced Thematic Mapper Plus, Landsat 4-5 Thematic Mapper, and Landsat 1-5 Multispectral Scanner instruments [C1L]. For the purpose of this paper, we will focus only on images collected from the Landsat 8 satellite.

### 1.1.1 Landsat 8

For the purpose of this paper, we will use images collected by the Landsat 8 satellite Figure 1.1, which was launched on an Atlas-V rocket from Vandenberg Air Force Base, California on February 11, 2013. Landsat 8 is the most recently launched Landsat satellite and carries the Operational Land Imager (OLI) and the Thermal Infrared Sensor (TIRS) instruments. Landsat 8 orbits the Earth in a sun-synchronous, near-polar orbit, at an altitude of 705 km, inclined at 98.2 degrees, and completes one Earth orbit every 99 minutes. The satellite has a 16-day repeat cycle with an equatorial crossing time: 10:00 a.m. +/- 15 minutes. It acquires about 740 scenes a day on the Worldwide Reference System-2 (WRS-2) path/row system, with a swath overlap (or sidelap) varying from 7% at the equator to a maximum of approximately 85% at extreme latitudes. A Landsat 8 scene size is 185 km x 180 km [LANa].

**Worldwide Reference System-2** We will be referring to each scene's location based on its path and row coordinates from the worldwide reference system-2, which is a global notation system for Landsat data. It enables a user to inquire about satellite imagery over any portion of the world by specifying a nominal scene center designated by path and row numbers. The WRS has proven valuable for the cataloguing, referencing, and day-to-day use of imagery transmitted

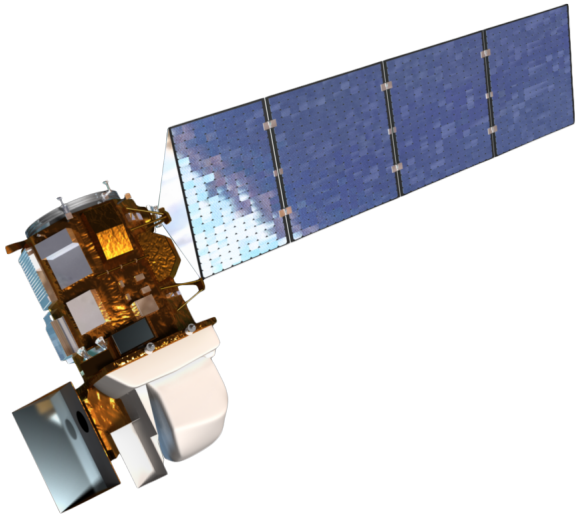

Figure 1.1: Landsat 8 satellite [LANb]

from the Landsat sensors [wrs]. Landsat's trajectory projected on the world map can be seen in Figure 1.2.

### Landsat scene naming convention

Each Landsat scene is named after a well-defined convention in order to easily check information such as the WRS path, row and the date of acquisition. Having access to this information without the need to download the scene itself or the metadata file which holds this information represents a valuable asset, since we can easily filter the data based on the naming convention itself. In the table 1.1.1 below we represent what each part of a Landsat scene of the form **LXS PPPRRR YYYYDDD GSIVV** means.

### Operational Land Imager

The Operational Land Imager (OLI) is a remote sensing instrument aboard Landsat 8, built by Ball Aerospace & Technologies. The sensor collects moderate resolution data that is used to monitor changing trends on the surface and evaluate how land usage changes over time. The images and data that OLI has helped collect have practical applications today in agriculture, mapping, and monitoring changes in snow, ice, and water [KK14]. The OLI operates in the visible (VIS) and short wave infrared (SWIR) spectral region, having a width of 185 km. It uses nine channels, which range from wavelengths of 443 nm to 2,200 nm. Of these nine channels, eight are multispectral and one is panchromatic. The eight multispectral channels have a 30-meter spatial resolution, and the panchromatic channel has a 15 meters one. The OLI generates

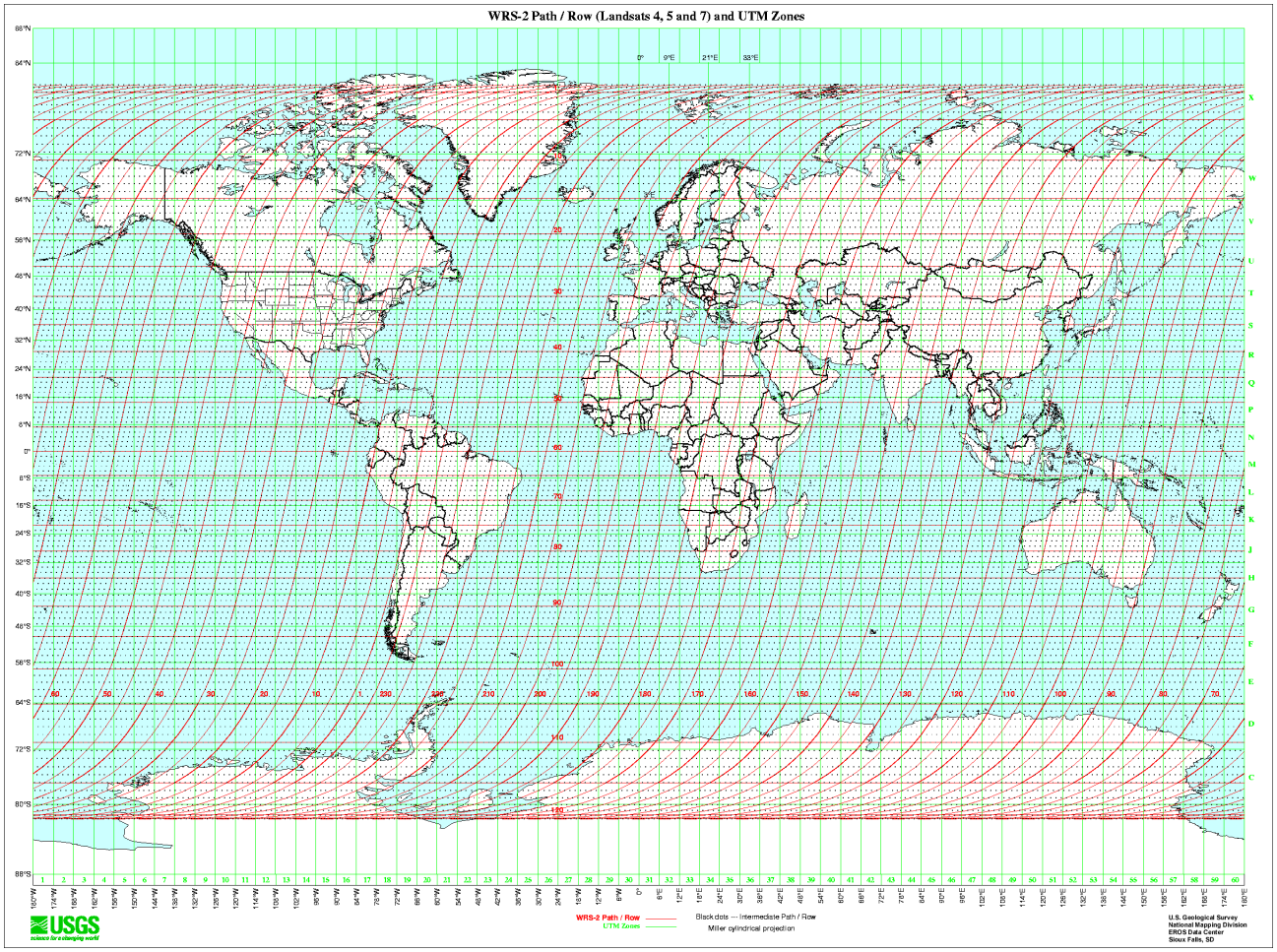


Figure 1.2: WRS-2 Path/Row for Landsat [wrs].

9 bands for Landsat as shown in Figure 1.3.

### Thermal Infrared Sensor

The Thermal Infrared Sensor (TIRS) measures land surface temperature in two thermal bands with a new technology that uses Quantum Well Infrared Photodetectors to detect long wavelengths of light emitted by the Earth whose intensity depends on surface temperature. These wavelengths, called thermal infrared, are well beyond the range of human vision [lgn]. The thermal infrared sensor generates 2 bands for Landsat as shown in Figure 1.4.

### OLI and TIRS bands

Landsat 8 acquires data from these sensors in in 11 bands, as following in Table 1.2.

| | |
|---|---|
| **L** | Landsat |
| **X** | Sensor |
| **SS** | Satellite |
| **PPP** | WRS path |
| **RRR** | WRS row |
| **YYYY** | Acquisition year |
| **DDD** | Julian day of the acquisition year |
| **GSI** | Ground station identifier |
| **VV** | Archive version number |

Table 1.1: Landsat 8 scene naming convention [sn].

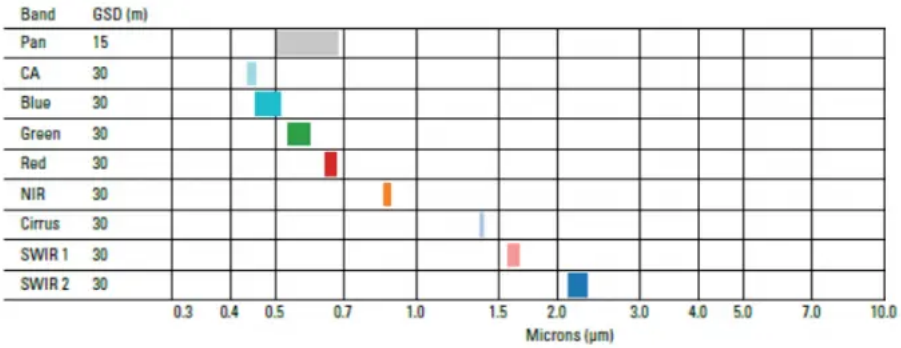


Figure 1.3: Landsat 8 OLI generated bands [l8o]

### 1.1.2 World Glacier Inventory

The World Glacier Inventory (WGI) proves to be a useful resource for building our dataset, since it contains information for over 130,000 glaciers. Inventory parameters include geographic location, area, length, orientation, elevation, and classification. The WGI is based primarily on aerial photographs and maps with most glaciers having one data entry only. Hence, the data set can be viewed as a snapshot of the glacier distribution in the second half of the 20th century. It is based on the original WGI (WGMS 1989) from the World Glacier Monitoring Service [WGI]. There are a number of ways to retrieve data from the inventory:

- download the entire database in a single ASCII text file (wgi_feb2012.csv);
- search by parameter using the Search Inventory interface;

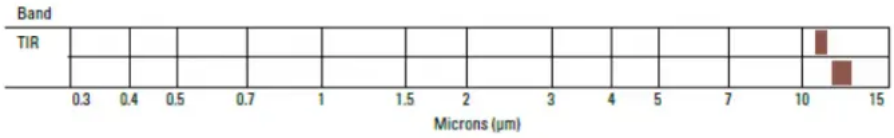


Figure 1.4: Landsat 8 TIRS generated bands [l8o]

| **Bands** | **Wavelength (micrometers)** | **Resolution (meters)** |
|---|---|---|
| Band 1 (Coastal aerosol) | 0.43-0.45 | 30 |
| Band 2 (Blue) | 0.45-0.51 | 30 |
| Band 3 (Green) | 0.53-0.59 | 30 |
| Band 4 (Red) | 0.64-0.67 | 30 |
| Band 5 (Near Infrared) | 0.85-0.88 | 30 |
| Band 6 (SWIR1) | 1.57-1.65 | 30 |
| Band 7 (SWIR2) | 2.11-2.29 | 30 |
| Band 8 (Panchromatic) | 0.50-0.68 | 15 |
| Band 9 (Cirrus) | 1.36-1.38 | 30 |
| Band 10 (TIR 1) | 10.6-11.19 | 100 |
| Band 11 (TIR 2) | 11.50-12.51 | 100 |

Table 1.2: Landsat 8-9 Operational Land Imager (OLI) and Thermal Infrared Sensor (TIRS) bands [ban].

- extract regions through the Extract Selected Regions interface.

The ASCII text file will be used with the purpose to define which are the glaciers to be included in the dataset to be built. An example of how this file looks like can be found in Figure 1.5.

The *parameters* which will be extracted for the dataset construction are the following:

- **wgi_glacier_id**: unique id representing one glacier (or part of it, if the coverage area is larger);
- **glacier_name**: name of the glacier (if it has one);
- **lat**: latitude of the glacier;

| | wgi_glacier_id | political_unit | continent_code | drainage_code | free_position_code | local_glacier_code | glacier_name | lat | lon |
|---|---|---|---|---|---|---|---|---|---|
| 2 | SU5X1430909 | SU | 5 | X143 | 9 | 90 | Zyuryuzamin | 38.92 | 71.272 |
| 3 | AT4J143OE00 | AT | 4 | J143 | OE | 6 | ZWISELBACH W | 47.112 | 11.038 |
| 4 | AT4J143OE00 | AT | 4 | J143 | OE | 5 | ZWISELBACH | 47.11 | 11.052 |
| 5 | CH4L01200008 | CH | 4 | L012 | 0 | 8 | ZWISCHBERGEN GL | 46.108 | 8.041 |
| 6 | CN5N236I0001 | CN | 5 | N236 | I0 | 1 | Zuxuehui | 31.828 | 94.675 |
| 7 | CH4J14304001 | CH | 4 | J143 | 4 | 1 | ZUORT VADRET DA | 46.738 | 10.271 |
| 8 | CN5O282B002 | CN | 5 | O282 | B0 | 23 | Zuoqiupu | 29.212 | 96.893 |
| 9 | CN5O282A047 | CN | 5 | O282 | A0 | 476 | Zuoguzasan | 29.958 | 95.92 |
| 10 | SU5X1430831 | SU | 5 | X143 | 8 | 310 | ZULUMART | 39.13 | 72.78 |
| 11 | CN5N224E001 | CN | 5 | N224 | E0 | 12 | Zuima | 29.839 | 96.456 |
| 12 | SU5X1430948 | SU | 5 | X143 | 9 | 489 | Zotkin | 38.649 | 71.244 |
| 13 | SU5X1430949 | SU | 5 | X143 | 9 | 490 | Zotkin | 38.649 | 71.244 |
| 14 | SU5X1430932 | SU | 5 | X143 | 9 | 326 | Zordi-Birauso | 38.673 | 71.664 |
| 15 | NZ6B868B000 | NZ | 6 | B868 | B0 | 7 | ZORA | -43.739 | 169.823 |
| 16 | SU5T09106366 | SU | 5 | T091 | 6 | 366 | ZOPKHITO | 42.88 | 43.43 |
| 17 | IT4L01104020 | IT | 4 | L011 | 4 | 20 | ZOCCA S | 46.285 | 9.647 |
| 18 | IT4L01104021 | IT | 4 | L011 | 4 | 21 | ZOCCA E | 46.292 | 9.653 |
| 19 | AQ7SSI00012 | AQ | 7 | SSI0 | 0 | 125 | Znosko Glacier | -62.1005 | -58.4865 |
| 20 | SU4X0300190 | SU | 4 | X030 | 1 | 903 | ZNAMENITYY | 80.53 | 61.02 |

Figure 1.5: World glacier inventory ASCII text file, as CSV

- **lon**: longitude of the glacier.

### 1.1.3 Asset Acquisition

Since Landsat 8 acquires over 700 scenes per day, this means that there are over two million scenes available for download, either making use of already built user friendly tools or by simply querying for them directly.

#### USGS Earth Explorer

One of the most popular services for satellite imagery downloading is USGS Earth Explorer. This is used for querying and ordering of satellite images, aerial photographs, and cartographic products through the U.S. Geological Survey. The tool is particularly useful when the main focus is to analyse a specific area rather than trying to acquire a large dataset of scenes. One can easily search for assets based on criteria such as world reference system path and row variables, latitude, longitude, cloud coverage, capture date and many others [USG].

However, downloading a large set of assets proves to be rather difficult by using this tool alone, since the parameters for each scene need to be manually set. On top of this, the query results have to be picked by hand and then passed for downloading through another application which handles their bulk download. This makes the process of building the dataset rather slow, frustrating and error prone. Such an example can be viewed in Figure 1.6, for the Belvedere glacier (45.942, 7.908).

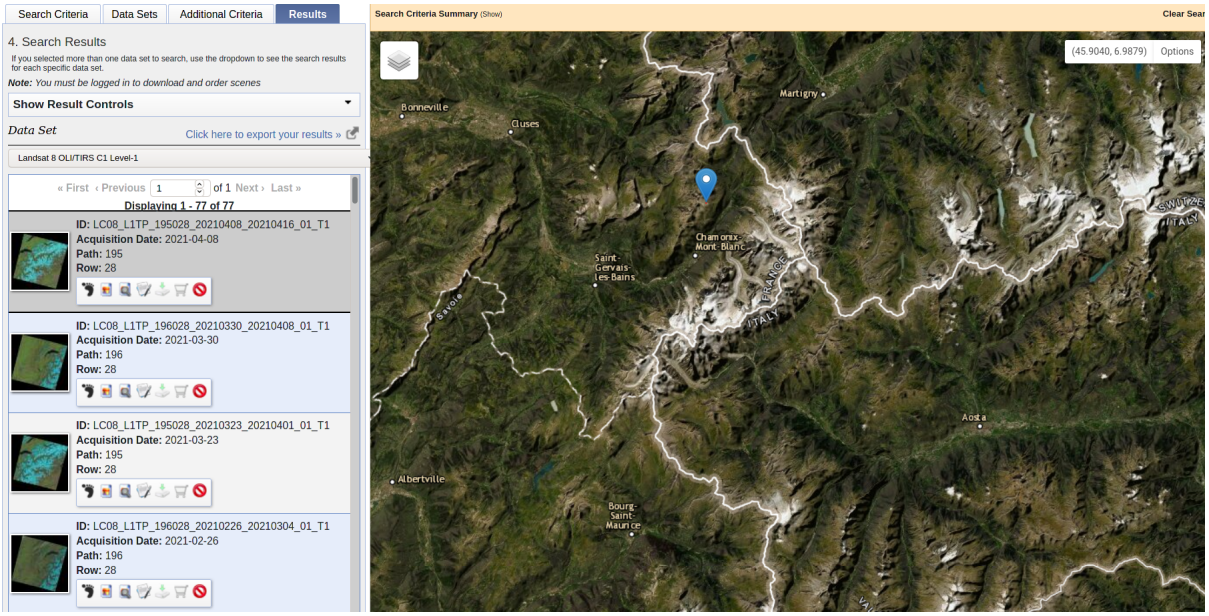


Figure 1.6: EarthExplorer

### SpatioTemporal Asset Catalog API

In order to fix the problem of excessive manual labour which appeared by using the USGS Earth Explorer, we rather implemented an endpoint of the SpatioTemporal Asset Catalog API, specifically, the following: `http://nsidc.org/data/glacier_inventory/index.html` [STA]. The main idea of searching by using parameters still remains, but instead of manual inputting data for the search data, we rely on using the above-mentioned World Glacier Inventory ASCII text file, since it already has all the required information for each glacier.

By using this method we can pick which glaciers we want to download based on their coordinates and calculate a bounding box representing the area we want to search, required for the STAC API query. Since there might be clouds which could obfuscate the area of interest in the image, we also add a maximum allowed cloud coverage along the bounding box.

The STAC API query also requires a name for the collection of assets we want our queries to be made on, which for us is landsat-8-l1 (Landsat 8 Collection 1, Level 1). Using these three parameters we can now easily acquire a large number of assets with minimal manual labour, as compared to the more user friendly tool provided by USGS.

The downloaded assets will be stored at a user specified disk location and they will be structured as shown in the Figure 1.7.

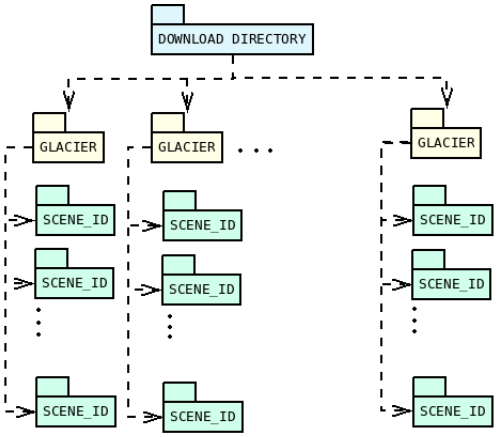


Figure 1.7: Download Directory

**Landsat 8 Collection 1, Tier 1**

To support analysis of the Landsat long-term data record, the Landsat data archive was reorganized into a formal tiered data collection structure, which ensures that all Landsat Level 1 products provide a consistent archive of data quality to support time-series analysis and data "stacking", while controlling continuous improvement of the archive, and access to all data as they are acquired. The implementation of collections ensures consistent and known radiometric and geometric quality through time and across instruments while improving control in the calibration and processing parameters [lc1]. By using data from this collection, we ensure that our images are fit for accurate pixel-to-pixel processing.

## 1.2 Dataset entities

TODO better make this intro... We will further describe which are the bands necessary for the image generation and what are their uses. On top of this, we will further explain what is the normalized snow difference index and what is the optical flow used for.

### 1.2.1 Bands

Each Landsat 8 band is represented by a 16 bit grayscale image with a resolution between 7000 and 10000 pixels, each pixel representing 30 meters. We can conclude, therefore, that one scene covers around 200 and 300 km of Earth. Only the green and SWIR1 bands will be used for the purpose of this thesis and below we will discuss the specifications of each.

| | |
|---|---|
| **Wavelength** | 0.53- 0.59 micrometers |
| **Spacial resolution** | 30 meters |
| **Resolution** | between 7000x7000 pixels and 10000x10000 pixels |
| **Depth** | 16-bit |
| **Format** | grayscale |

Table 1.3: Landsat 8 green band specifications.

### Band 3 - Green Band

The green band, alongside with the red and blue ones, fall in the visible spectrum and it is usually used for mapping peak vegetation. Figure 1.8 is an example of the green band for the Belvedere glacier, specifically taken from scene LC81950282015098LGN01.

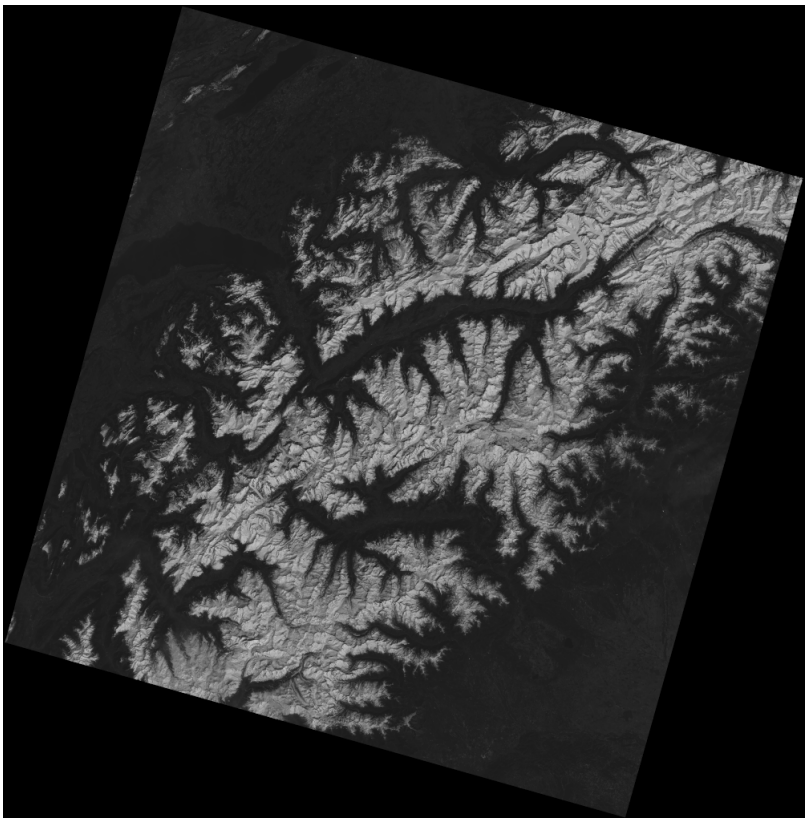

Figure 1.8: Green band of scene LC81950282015098LGN01 from the Belvedere glacier, Italy.

### Band 6 - SWIR1 Band

The shortwave infrared 1 band is particularly useful for enhancing geology objects such as rocks and soils which look similar in other bands [bd]. Alongside this, it also discriminates moisture

| | |
|---|---|
| **Wavelength** | 1.57 - 1.65 micrometers |
| **Spacial resolution** | 30 meters |
| **Resolution** | between 7000x7000 pixels and 10000x10000 pixels |
| **Depth** | 16-bit |
| **Format** | grayscale |

Table 1.4: Landsat 8 SWIR1 band specifications.

content of soil and vegetation and penetrates thin clouds [ban]. Figure 1.9 is illustrated below as an example of a SWIR1 band taken from the same scene as the green one above.

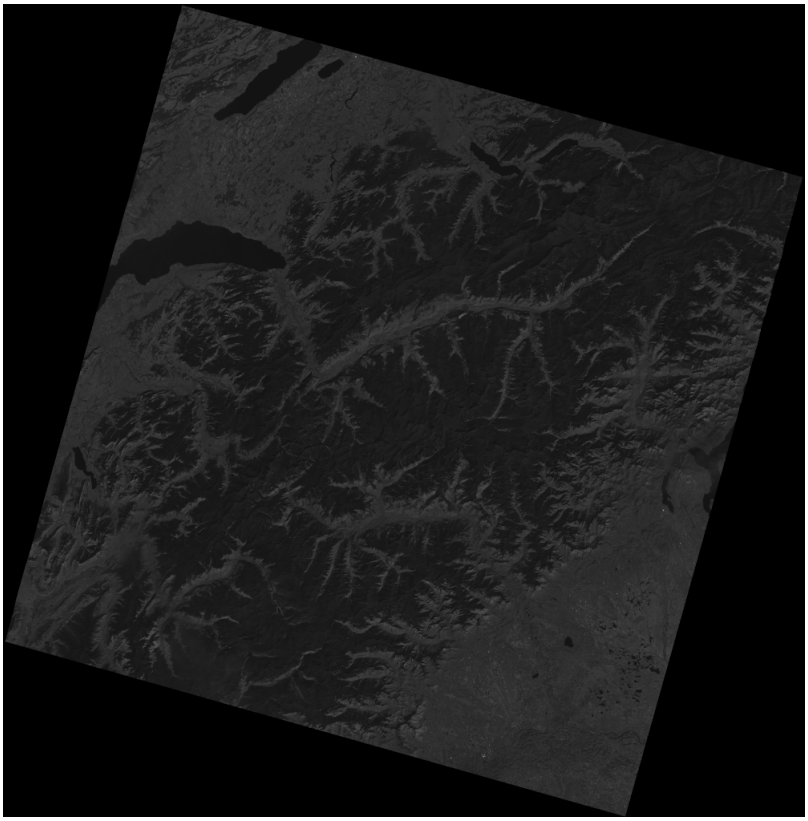

Figure 1.9: SWIR1 band of scene LC81950282015098LGN01 from the Belvedere glacier, Italy.

### Normalized Snow Difference Index

The normalized snow difference index (NDSI) is an index which relates to the presence of snow/ice in a pixel. Snow and ice usually have a very high reflectance in the visible spectrum and very low one in the shortwave infrared one, which is useful for mapping out most types of clouds from the scene [nds]. We can therefore use the formula from Equation 1.1 in order to

highlight the snow and ice pixels from a Landsat 8 image.

$$NDSI = \frac{green - SWIR1}{green + SWIR1} \tag{1.1}$$

By combining the snow reflectance behaviour of snow and ice for each of these bands, we can create a normalized snow difference index image which has values in the range [-1, 1]. We can then apply a threshold on the pixels which states that if the NDSI value for a pixel is larger than 0.0, then that pixel represents snow/ice covered land; similarly, if its value is smaller than 0.0, that pixel represents snow/ice free land [nds], [Hal15], as represented in Equation 1.2. Of course, this represents the general value for the threshold, but with the increase of its value, we can more accurately differentiate between snow and ice. With larger threshold values we can state that a pixel represents ice covered land rather than just snow fall land [Hal15]. We have tried different values for the threshold and came to the conclusion that a value of 0.3 is best suited for our needs.

$$\begin{cases} \text{snow/ice land} & \text{if } NDSI \geq 0.0 \\ \text{snow-free land,} & \text{NDSI ¡ 0.0} \end{cases} \tag{1.2}$$

An example of a generated NDSI for the Belvedere glacier, scene LC81950282015098LGN01, can be viewed in Figure 1.10.

Figure 1.10: NDSI of scene LC81950282015098LGN01 from the Belvedere glacier, Italy.

## 1.3 Alignment

Landsat's trajectory orbiting Earth is not fully precise, therefore not all images will be pixel-to-pixel aligned, which is a problem for image processing. Since we want to track each pixel's movement, we must be sure that its coordinates in the image do not change between any two given scenes. Figure 1.11 highlights the misalignment from scene between scenes LC81950282013316 and LC81950282013364, as an example. Since the bands we work with have a spacial resolution

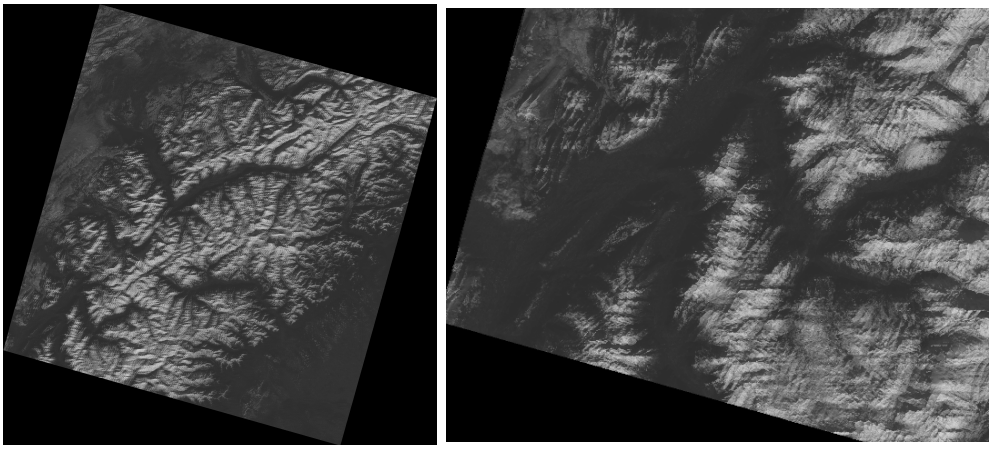

Figure 1.11: Overlapped unaligned green band of scene LC81950282013316 and reference LC81950282013364 of the Belvedere glacier (195/028 WRS-2).

of 30 meters, which means that each pixel in the image represents 30 meters. Even with a misalignment of just 50 pixels we would end up with a 1.5 km difference between two scenes. Tracking pixels through the image without aligning them first would mean that we could never be sure that what we are looking at is indeed the same location. We have used different approaches on alignment which will be described in Section 2.2.1. Figure 1.12 highlights the alignment corrected scene from Figure 1.11.

#### NDSI's Optical Flow

Since our dataset represents a **time series of satellite images**, as described in Section 1.1.1, we thought that in order to generate a new image of this series, we could extract the **motion of each pixel** from one scene to another. This information could then be applied between any two consecutive (date-wise) scenes from the set and we could update a pixel's coordinates based on the value its motion vector and store it in a new image. Extracting the motion vectors between two consecutive frames can be achieved by calculating their optical flow. **Optical flow** is defined as the motion of objects between consecutive frames of sequence, produced by the relative movement between the object and camera. By using computer vision algorithms which calculate the optical flow of two scenes, we could track the motion of melting glaciers across

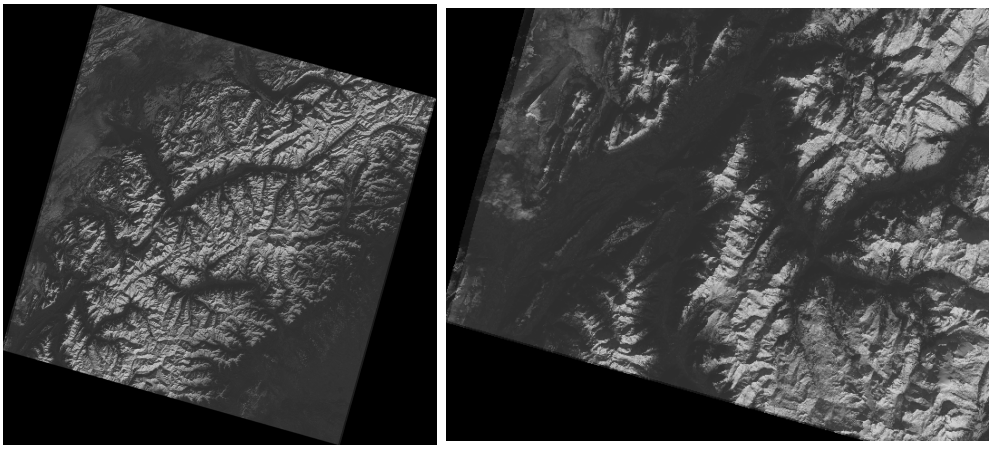

Figure 1.12: Overlapped aligned green band of scene LC81950282013316 and reference LC81950282013364 of the Belvedere glacier (195/028 WRS-2).

them in order to estimate their current velocity and possibly **predict their position** in the next frames [opt].

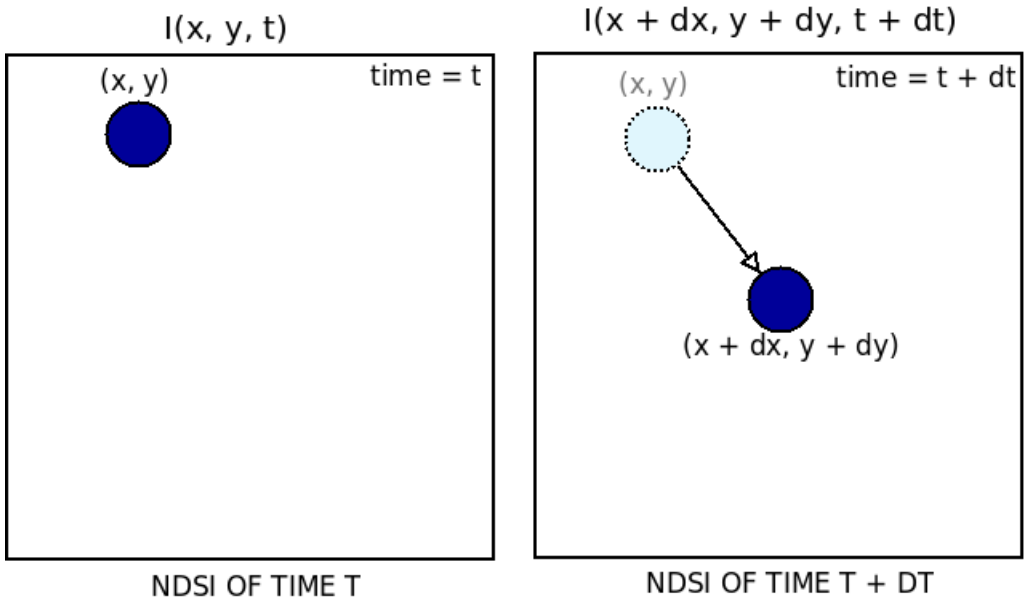


Figure 1.13: Optical flow problem visualisation.

Figure 1.13 emphasizes the problem visually, where we can express an image as a **function of space**, with the coordinates $(x, y)$, and **time** $t$. If we take the first image $I(x, y, t)$ and we move its pixels by a distance of $dx, dy$ over a timestamp $dt$, we obtain the new image as follows: $I(x + dx, y + dy, t + dt)$ [orb].

There are multiple types of optical flow algorithms, but for the purpose of our thesis, we have chosen the **dense optical flow** one, specifically **Gunnar Farneback's**. Even if dense implementations have higher cost we chose to make this trade mainly because it calculates the

motion for each pixel of the frame and it also has a higher accuracy [orb] compared to methods such as Lucas-Kanade (sparse) [LK81].

We have used optical flow as a tool to generate **distance vectors** based on motion between the NDSI($time = t$) and NDSI($time = t + dt$). These vectors will be used in order to create the **motion predicted NDSI**.

### Motion Predicted NDSI

Whilst optical flow allows us to see movement of the ice front which already happened in the past, we wanted to use this information and apply it on the most recent images such that we can estimate further glacier movements.

We obtain the motion predicted generated NDSI by relocating each pixel value from the NDSI($time = t + dt$) to a **predicted location**. For now, this location is generated by adding the distance vectors obtained by optical flow as described in Section **??** to the pixels of the same image, therefore shifting them by twice as much as they originally did, in their respective directions, as it can be observed in Figure 1.14. More methods of calculating the predicted location can be found in Section 4.1.

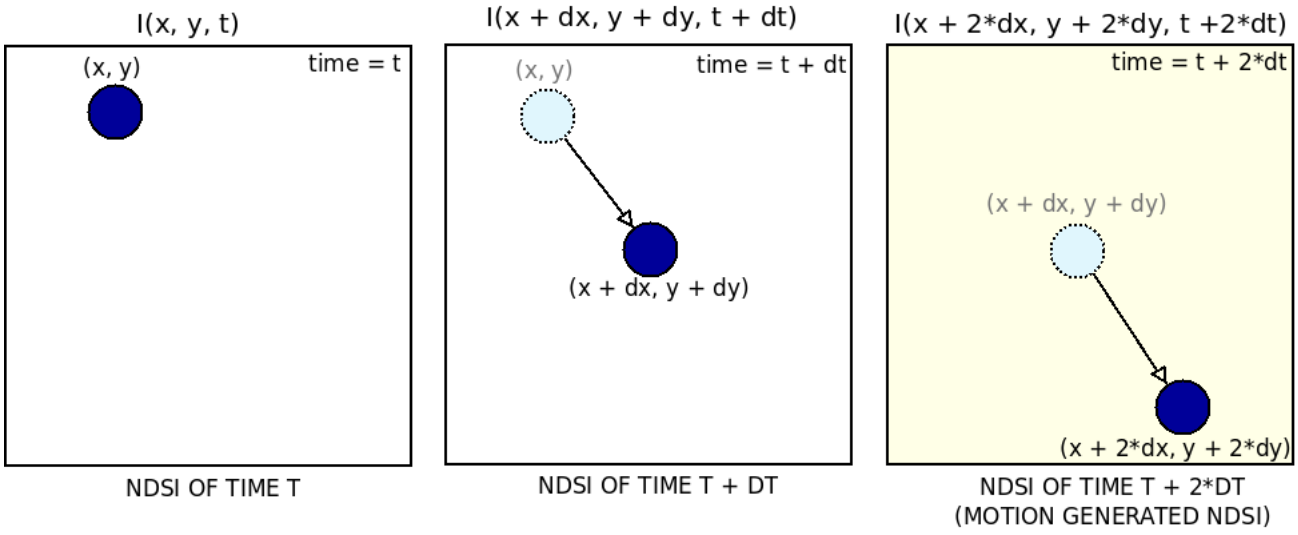


Figure 1.14: Motion generated NDSI algorithm.

By applying this function to each pixel of the NDSI ($t + dt$) image and by making some adjustments which are further described in Section 2.2.4, for the Belvedere glacier we have generated the motion predicted NDSI as shown in Figure **??**. TODO SHOW THE IMAGE AFTER CROPPING THE EDGES.

## 1.4 Graphical User Interface

# Chapter 2

# Design and Implementation

The design of the application will be discussed while focusing on two different aspects:

- **search and download**
- **processing**

The first section is optional and can be run independently from the processing, since the user can have an already created database of images. However, we have focused on simplifying the process of satellite imagery downloading as described in 1.1.3 through using the sat-search library as a helper for searching and downloading assets. More information on querying can be found in 2.1. Given an existing set of data, the next step is to pass the root folder which contains the glacier directories to the processing unit, also designed as a plug-in mechanism which will trigger the graphical user interface to pop up and allow for the processing options to appear. More information on this section can be found at 2.2. From there, one can start processing by simply making use of the predefined buttons described in 1.4.

## 2.1 Search and Download

Searching as well as downloading have been implemented on top of the sat-search library and it is used directly from the command line interface by running the script described in Section **??**. As an input we will be using a CSV file which will have the form as described in Section 1.1.2. Mainly we will need just four attributes to be specified for each desired glacier in order to create a query for searching, as follows: *wgi_glacier_id, glacier_name, lat and lon.* The CSV file will be intercepted by the **Download** class and sent for processing row by row (glacier by glacier) to the **GlacierFactory** class. This one is responsible for parsing each row of the input CSV file and transforming the information in **Glacier** objects, which will be passed back to the **Download** class. Using the newly created Glacier object we can construct its **search query**

by calculating its bounding box, specifying the asset collection from which we request data and setting other parameters (maximum allowed cloud coverage, in our case). Listing 2.1 represents an example of querying for glacier Belvedere, with the following parameters set in the CSV file: *"IT4L01211009","BELVEDERE","45.942","7.908"*.

Listing 2.1: Search query created by sat-search

```
{"page": 1, "limit": 170, "bbox": [7.907990000000001, 45.94199, 7.90801,
    45.94201], "query": {"eo:cloud_cover": {"lt": 10}}, "collection":
    "landsat-8-l1"}
```

The result of each glacier query will be automatically saved in a **JSON file** which is used as a data buffer between the search and download. The downloader takes each JSON file generated by the search engine and sends the command for getting that asset. The technical specifications of the Download, GlacierFactory and Glacier classes can be found in Figure 2.1.

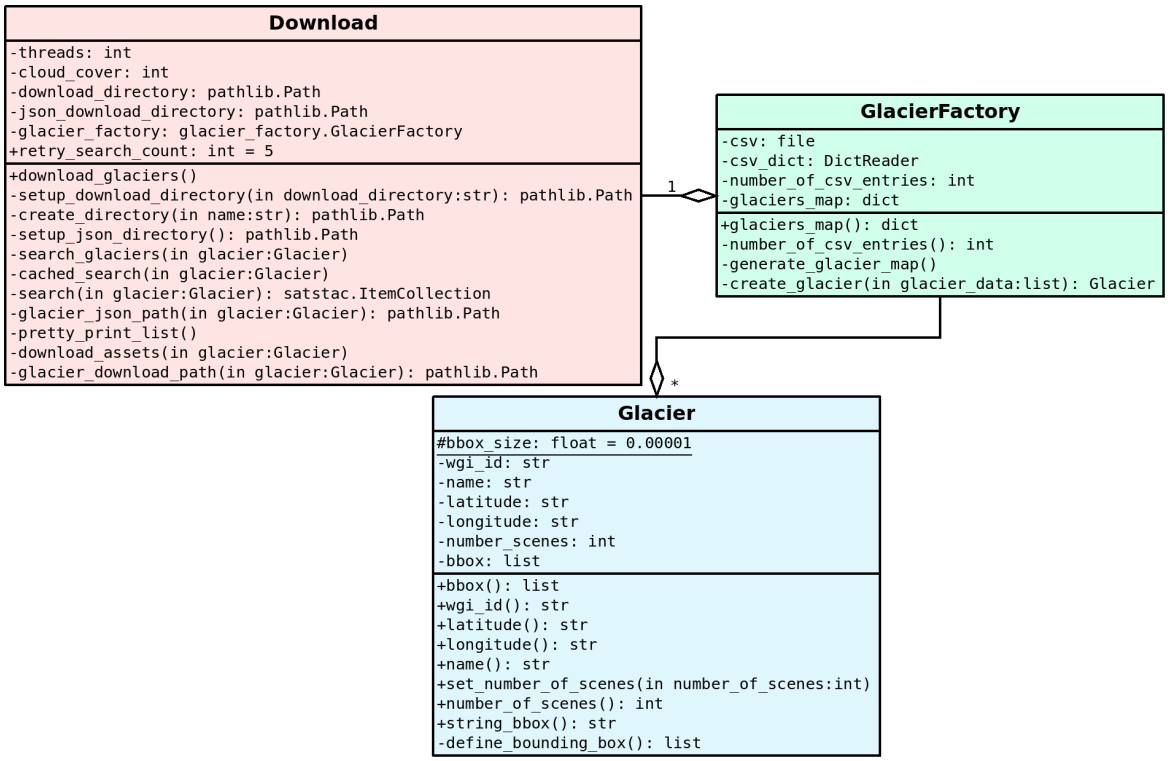


Figure 2.1: Technical specifications of Download, GlacierFactory and Glacier classes.

**Data corruption verification** Both the searching and downloading functions are executed concurrently, since they are **time intensive** tasks. Each band has around 60 MB, which would make one scene approximately 360 MB. For the Belvedere glaciers, we found 78 scenes with a cloud coverage of 10%. This means that the size of the entire glacier data will be around

28 GB. Given that one asset's size is quite large, it takes a lot of time to finish the download. In this time, even a small connection interruption might result in corrupted data files, which would interfere with processing. Therefore, we implemented an **extra security measure** in the sat-stac library which verifies whether a file with the same name already exists at the download location. If it does, its size will be compared to the size taken from the STAC-API servers. In case the two file sizes don't match, it means that the file got corrupted during download and it will be downloaded again. If the sizes match, the downloader will skip the file and continue when it finds one requested file which was not yet found on the disk, such that we do not end up unnecessarily overwriting the already existing files in case of failure.

## 2.2 Processing

Processing can be viewed as four different entities, as following:

- Image alignment;
- Normalized snow difference index;
- NDSI's motion matrix;
- Motion generated NDSI.

Before generating any images, we first need to make sure that the pixels between any two scenes are not misaligned. The details of the alignment process are described in Section 2.2.1. After we ensure that out images are fit, we move on to creating the normalized snow difference index image in order to enhance the ice through a set threshold (see Section 2.2.2). We then create the matrix of pixel motions calculated between two consecutive (date-wise) scenes, process described in Section **??**. Based on the information of where each pixel has moved between two consecutive images, we can then create our future NDSI image by applying the motion vectors on each pixel from a scene. More details on the design and implementation of this part can be found in Section 2.2.4. The normalized snow difference index image will be computed for each scene as described in Section 1.2.1.

### 2.2.1 Alignment

In order to keep a high level of abstraction, we have split each scene into aligned and unaligned: Scene and AlignedScene objects. These and their children are created when crawling through the glaciers directory, specifically for each region of interest. However, aligning all the images when crawling would result into a lot of idle time for the user when starting the application and

it would not be overall feasible to do so. Therefore, a scene is aligned only when it is specifically being selected in the graphical user interface (or when another entity needs it, such as optical flow and image generation). By doing this, we ensure that there is no unnecessary extra waiting time for each scene that we have when powering up the interface. As for the actual alignment, we have used an algorithm which **collects features** from each band of a scene and tries to match them with a given reference one's features. The obtained **matches** can be then used to create an **affine transformation matrix** which specifies by how much did the current image **rotated and translated** in comparison to its reference. The affine transformation matrix will be then applied to the current image by a **warping** algorithm with the goal of ensuring as much as possible that there is no misalignment between any pixel of two different images from the same time series. Section 2.2.1 describes more technically how we built this and what were the computer vision algorithms used for that matter. Figure 2.2 contains the technical details of each class which takes part into the alignment process.

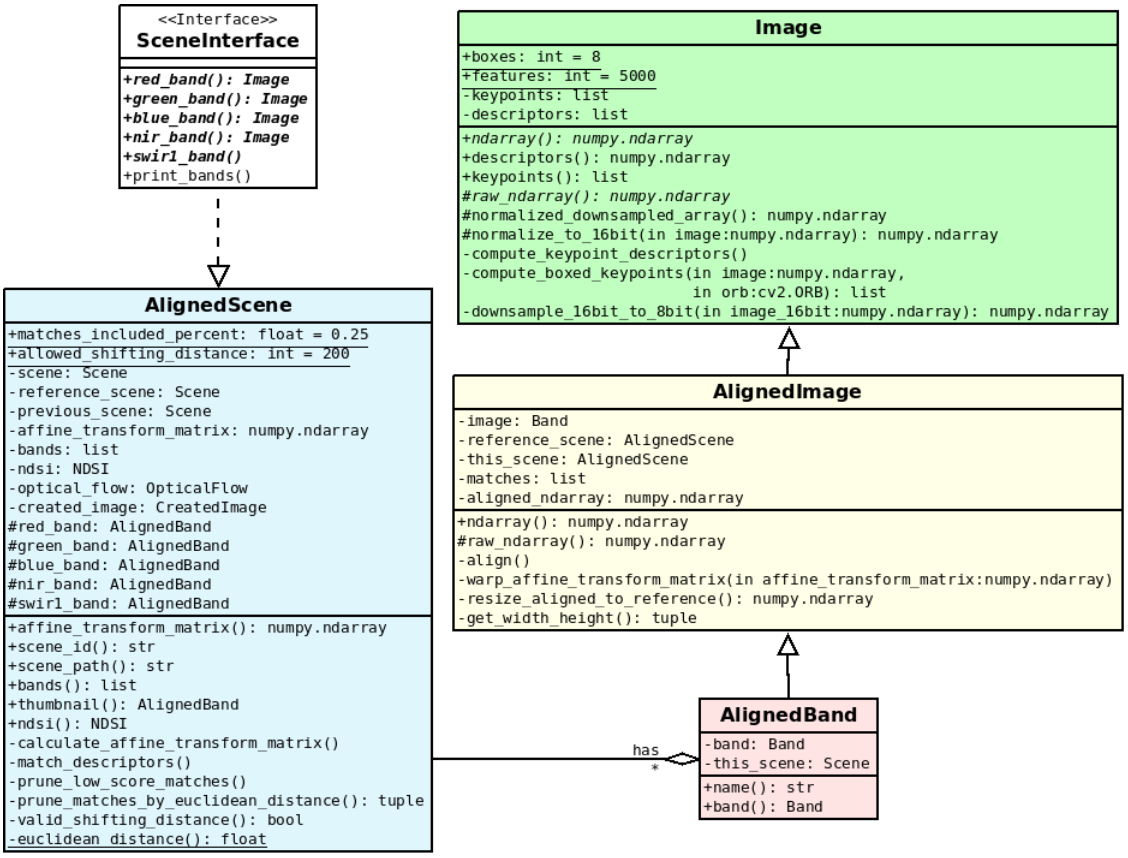


Figure 2.2: Technical diagram for the SceneInterface, AlignedScene, AlignedBand and Image classes.

### Alignment algorithms

As we have seen in Section 1.1.1, the Landsat 8 satellite does not have a perfect trajectory, which results into misaligned images. These changes are not obvious to the naked eye from scene to scene, but overlapping two scenes highlights this problem, as we can see in Figure 1.11. Even so the scenes are almost similar, which means that aligning them does not prove to be very hard and it is done by using a strong **keypoint detection algorithm** which in our case detects edges formed by mountains and other geographical features present in the scenes. The **computer vision algorithms** used for this are **ORB (Oriented FAST and Rotated BRIEF)**, **Harris Corner Detector** and **RANSAC (Random Sample Consensus)** and they are applied on the raw 16bit grayscale image.

**ORB** represents a fusion between the features from accelerated segment test (**FAST**) keypoint detector and binary robust independent elementary features (**BRIEF**) descriptor. The FAST detector finds keypoints in the image and uses Harris corner measure to select the a number of top points from the list (25%, in our case) [orb]. In order to use the detector we must first normalize and downsample the 16bit raw image to 8 bit, as it is required by ORB. Each keypoint is then represented by a circle of 16 pixels. The descriptors must then identify these keypoints and pair with each other. We compute for each scene its **keypoints and descriptors** taken from **each band**, in order to increase the precision of alignment later on. Also, since we are working with satellite imagery there might be cases when the algorithm only finds keypoints in a small part of the image, resulting in erroneous distortion. **Splitting** the image into multiple boxes and applying the orb detect algorithm on each separate part of the image proved to solve this problem and create much better results.

After ensuring that we have good enough keypoints we will be aligning each scene with its reference by simply comparing the keypoint-descriptor pairs between the two scenes and checking which are equal. If two pairs are found to be equal, it means that the algorithm has found the same feature in both images and can calculate by how much it **rotated and shifted**. However, not all of these matches represent good results in our case, since the trajectory Landsat 8 can vary. We have found that selecting only **5000 keypoints** in the **top 25% matches** from the total yielded in the better results overall. We then set the **maximum allowed shifting euclidean distance** between any two pixels to be at most 200, so 6km on the map, therefore getting rid of outlier matches based on the distance. The **matches** from the reference and image to be aligned will be used alongside the **Random Sample Consensus (RANSAC)** algorithm in order to create the **affine transformation matrix** using the cv2 library. RANSAC, proposed by Fischler and Bolles [FB81], is a general parameter estimation approach designed to handle a large proportion of outliers in the input data [Der10]. After we

have the transformation matrix, we can **warp** it on the image to be aligned in order to get the result.

### 2.2.2 Normalized Snow Difference Index

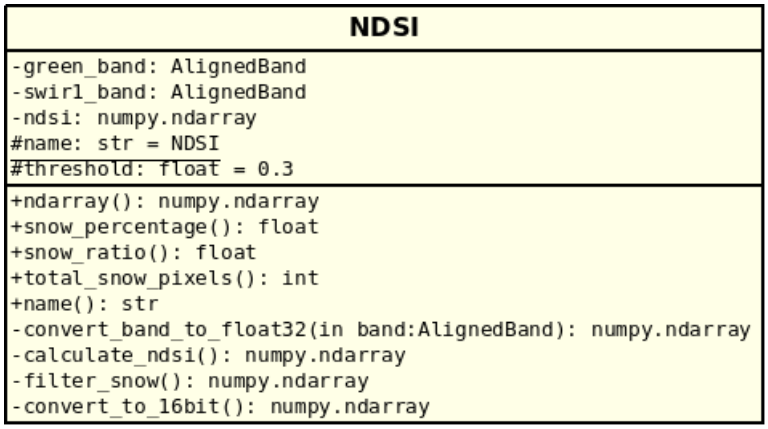


Figure 2.3: Technical diagram for the NDSI class.

The NDSI is generated by using the Formula 1.1 described in section 1.2.1 on the green and shortwave infrared bands from a specific scene. Each band pixel is converted to **float32** in order to increase the **precision** of calculation; as each band is read as a n-dimensional array we can use numpy's optimised processing for generating the NDSI index image, since it converts the data and runs on native code rather than going through Python's interpretor. We then **filter** the generated image such that everything which is snow-free land will be -1 (**black**), but this is only applied for better visual analysis. In the background, we work with the full range of the image for better precision, which is the raw image. Figure 2.3 holds the technical diagram for the NDSI.

### 2.2.3 NDSI's Optical Flow

As described in Section 2.2.4, in order to determine the motion vector for each pixel between two images, we have used **Gunnar Farneback's** algorithm for dense optical flow calculation, implemented by the **OpenCV** library. The MotionVectors class takes as input two NDSI images.

Since the NDSI images have been aligned, depending on the affine transformation matrix, they will not overlap perfectly any more, resulting in artifacts at their borders. Applying optical flow by using two NDSI images as such would result in highly distorted motion vectors at the borders, as it can be seen in Figure **??**. For fixing this, we have created a mask from each image

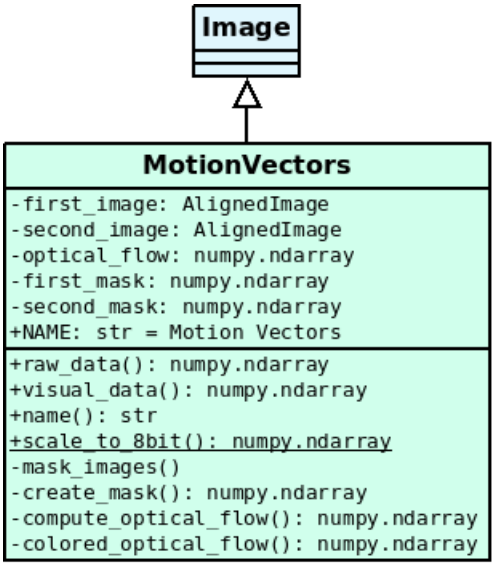


Figure 2.4: Technical diagram for the MotionVectors class.

such that both of them are cropped with the mask of the other. This results in losing a small number of pixels at the borders which could not be taken into consideration anyway.

TODO GIVE EXAMPLE OF OPTICAL FLOW IMAGE

In addition to the two NDSI image inputs, we have used the optical flow algorithm with the parameters specified in Listing 2.2. We have used a pyramid scale of 0.5 and 6 pyramid levels having in mind that the images that we work with are large. By choosing this combination of parameters, we state that at each level, the image is going to be reshaped at half the size of the previous one; therefore the area of search for motion is small enough such that the optical flow is able to track movement.

Listing 2.2: Optical flow parameters

```
cv2.calcOpticalFlowFarneback(..., pyr_scale=0.5, levels=6, winsize=15,
    iterations=3, poly_n=5, poly_sigma=1.2, flags=0)
```

After our images are perfectly overlaid, we can give them as inputs to the optical flow algorithm. The result of this will be the computed motions for each pixel, represented as a tuple of the distance that its coordinates moved from one frame to another. Specifically, as referring to Figure 1.14, each item from the optical flow n-dimensional array will represent the motion distance vector $(dx, dy)$ as calculated between NDSI($time = t$) and NDSI($time = t+dt$).

To be able to visually interpret the optical flow output, we have used a colour coded image. Colour intensity is directly proportional to the motion vector length, while its hue represents the direction of this vector, as represented in Figure 2.5.

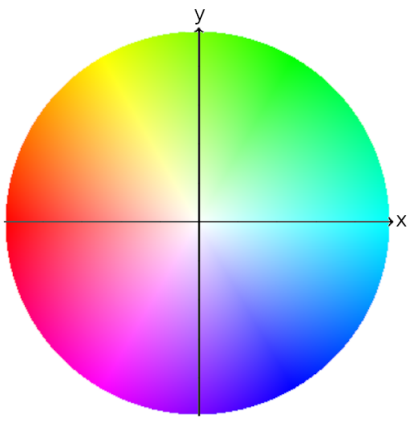


Figure 2.5: Hue representation of the optical flow.

TODO PUT OPTICAL FLOW IMAGE FINISHED

### 2.2.4 Motion Predicted NDSI

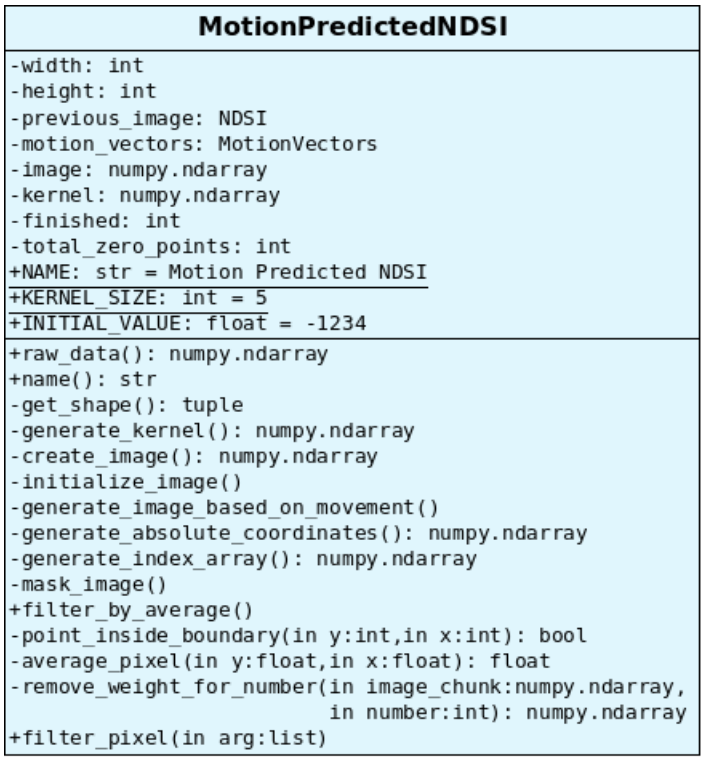


Figure 2.6: Technical diagram for the MotionPredictedNDSI class.

As means to put in practice what we described in Section 1.3, we start by initialising a new numpy n-dimensional array based of the shape of the image NDSI($time = t + dt$). The new array will be filled with the data generated by using the two required entities:

- **NDSI(**$time = t + dt$**)**;
- **motion vectors (**$dx, dy$**)** extracted by optical flow.

Each motion vector $(dx, dy)$ corresponds to the movement of a pixel from the NDSI($time = t + dt$) image. We can use this information to generate the new predicted position $(x + 2 * dx, y + 2 * dy)$ for each pixel of the NDSI($time = t + dt$) image.

**Algorithm 1:** Algorithm used for motion predicted image generation based on the optical flow vectors and NDSI($time = t + dt$)

```
1  function generate (previous_NDSI, motion_vectors);
   Input  : The NDSI(time = t + dt and the motion vectors generated by optical flow
            between NDSI(time = t) and NDSI(time = t + dt)
   Output: The motion predicted NDSI(time = t + 2 * dt))
2  motion_predicted_image ← previous_NDSI.shape;
3  width ← NDSI.width();
4  height ← NDSI.height();
5  for y in [height, width] do
6      for x in [height, width] do
7          dx ← motion_vectors[y][x][0];
8          dy ← motion_vectors[y][x][1];
9          new_x ← x + dx;
10         new_y ← y + dy;
11         motion_predicted_image[new_y][new_x] ← previous_NDSI[y][x];
12     end
13 end
14 return motion_predicted_image;
```

As a first approach of populating the new image, we have simply iterated over all the pixels in the NDSI($time = t + dt$) image and calculated their new coordinates based on their motion vector information, as it can be seen in Algorithm 1. Since this is an iterative approach, it does not scale for images as large as the ones we are using. On top of this, since we are using the Python language, which is an interpreted one, each operation has to go through an interpreter before being run. For very granular data processing this proves to be inefficient. For a typical image of 8543x8039 resolution, it takes as much as 10 minutes for generating the image on the machine that I have used (specifications at Section 3).

Since using the naive approach is not feasible in our application, we had to avoid iterating

over the whole image in the first place. Therefore, instead of looping over each pixel in order to generate the values for the new image, we have made use of the numpy library to heavily optimise our data processing. By using numpy functions and vectorized operations directly instead of iterating we were able to improve the processing time by around 1700%, bringing it down approximately from 10 minutes to 35 seconds.

In the iterative approach, for each pixel we add its motion to its position such that we get its new location. These numbers can be computed ahead of time and transformed into arrays such that we only use numpy operations. By creating an array of the initial coordinates $x, y$ and adding the motion vectors $(dx, dy)$ array to it, we generated the absolute coordinates where each pixel from the NDSI image should be translated. By adding this modification we got rid of lines 5, 6, 7, 8, 9, 10 and 11 from the algorithm and replaced them with the code as it can be seen in Algorithm 2. The $absolute\_coordinates$ and $previous\_NDSI$ can be treated as a sparse array which is then densified using numpy capabilities.

**Algorithm 2:** Algorithm used for motion predicted image generation based on the optical flow vectors and NDSI($time = t + dt$)

1 $index\_array \leftarrow [[[0,0],[1,0]...[width,0]\ [0,1],[1,1]...[width,1]\ ...$
$[0,height],[1,height]...[width,height]]]$;

2 $absolute\_coordinates \leftarrow motion\_vectors + index\_array$;

3 $motion\_predicted\_image[absolute\_coordinates] \leftarrow previous\_NDSI$;

The function used to generate the predicted NDSI does not have a one-to-one domain, thus making it undefined for certain inputs. This results in a very noisy generated image, as it can be seen in Figure TODO. We have implemented a filter which uses the weighted average of the neighbouring pixels values to fill the missing ones. The detailed description of the implementation can be found in Section 2.2.5.

### 2.2.5 Generated image filtering

A first step into creating the filter for the motion predicted NDSI image is to create a mask which will be applied on the black border of the scene such that we are looking for undefined values only inside it. Using numpy, the coordinates of these pixels are extracted into an array which then can be processed in parallel by Pythons' multiprocessing library. We have used multiprocessing instead of threading because Python does not have true parallelization in multithreading, only concurrency, which would not be a real improvement on the processing time. Since all the processes have to write different chunks of the same image and they do not share the same

memory space, we chose to solve this problem by creating a shared memory buffer to hold the image.

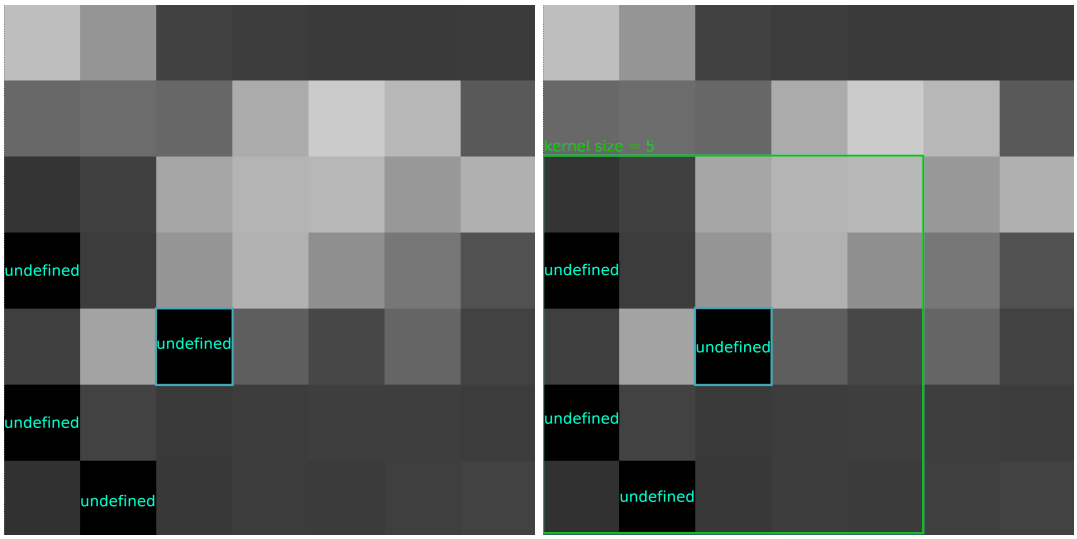


Figure 2.7: Zoomed in part of the predicted NDSI (left). Neighbourhood extracted (right).

The value of each found undefined pixel has to be calculated as an weighted average composed of its neighbouring pixels, as shown in Figure 2.7. We created the kernel which holds the weight of each pixel in the neighbourhood such that pixels closer to the centre have a higher weights than those near the edge. However, we have the case when we have multiple undefined pixels in the same neighbourhood. Since we cannot take their values in consideration, we do not want to include them in the weighted average, thus we set their weight to be 0, as shown in Figure 2.8 left. The result of the weighted average will be then stored as the value of the currently focused undefined pixel, as highlighted in Figure 2.8 right.

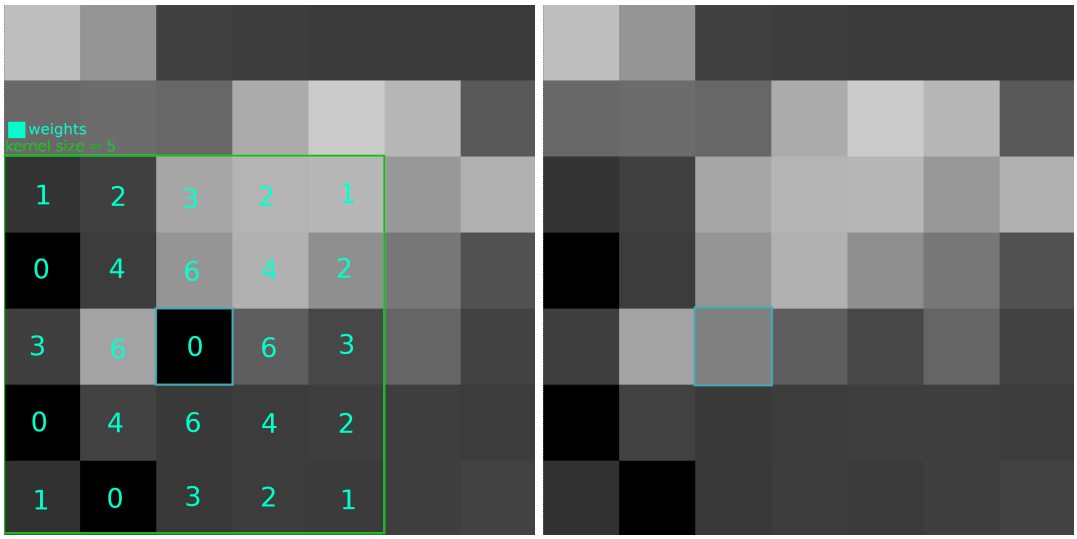


Figure 2.8: Generated weights kernel (left). Filled in undefined value (right).

# Contents

# List of Figures

# List of Tables

# Bibliography


[ban] bands. https://www.usgs.gov/faqs/what-are-best-landsat-spectral-bands-use-my-research Accessed: 2021-06-21.

[bd] bd. https://landsat.gsfc.nasa.gov/landsat-8/landsat-8-bands/. Accessed: 2021-06-22.

[C1L] Landsat collection 1. https://www.usgs.gov/core-science-systems/nli/landsat/landsat-collection-1?qt-science_support_page_related_con=1#qt-science_support_page_related_con. Accessed: 2021-06-20.

[Der10] Konstantinos G. Derpanis. Overview of the ransac algorithm. 2010.

[FB81] M.A. Fischler and R.C. Bolles. Random sample consensus: A paradigm for modelfitting with applications to image analysis and automated cartography. 1981.

[Hal15] Dr. Dorothy K Hall. Viirs snow cover algorithm theoretical basis document (atbd). 2015.

[KK14] Edward J Knight and Geir Kvaran. Landsat-8 operational land imager design, characterization and performance. In *Proceedings of the ECAI 94 Workshop on Applications of Evolutionary Algorithms*. Landsat-8 Sensor Characterization and Calibration, 2014.

[l8o] l8otb. https://gisgeography.com/landsat-8-bands-combinations/. Accessed: 2021-06-22.

[LANa] Landsat. https://www.usgs.gov/core-science-systems/nli/landsat/landsat-8?qt-science_support_page_related_con=0#. Accessed: 2021-06-21.

[LANb] Landsatpic. https://landsat.gsfc.nasa.gov/sites/landsat/files/2013/01. Accessed: 2021-06-21.

[lc1] lc1l1. https://prd-wret.s3.us-west-2.amazonaws.com/assets/palladium/production/atoms/files/LSDS-1656_%20Landsat_Collection1_L1_Product_Definition-v2.pdf. Accessed: 2021-06-22.

[lgn] lgng. https://landsat.gsfc.nasa.gov/. Accessed: 2021-06-21.

[LK81] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. 1981.

[nds] ndsi. https://nsidc.org/support/faq/what-ndsi-snow-cover-and-how-does-it-compare-fsc. Accessed: 2021-06-22.

[opt] opticalflow. https://nanonets.com/blog/optical-flow/. Accessed: 2021-06-25.

[orb] orb. https://docs.opencv.org/4.5.2/d1/d89/tutorial_py_orb.html. Accessed: 2021-06-24.

[sn] sn. https://gisgeography.com/landsat-file-naming-convention/. Accessed: 2021-06-22.

[STA] Stac. https://stacspec.org/STAC-api.html. Accessed: 2021-06-20.

[USG] Usgs. https://earthexplorer.usgs.gov/. Accessed: 2021-06-21.

[WGI] World glacier inventory. http://nsidc.org/data/glacier_inventory/index.html. Accessed: 2021-06-20.

[wrs] wrs. https://landsat.gsfc.nasa.gov/about/worldwide-reference-system/. Accessed: 2021-06-22.

# Chapter 3

# Performance

Here we should also write on what machine I ran the code and try it on different ones as a thing. Also write about the idea of moving the processing on cloud, but there might be traffic overhead because the images are large. Also say that we store it on harddrive, NOT ssd. Make tests with both.

# Chapter 4

# Conclusions

## 4.1 Future Development

# Chapter 5

# Glossary

## 5.1 Acronyms

| | |
|---|---|
| WGI | World Glacier Inventory |
| NDSI | Normalized Snow Difference Index |
| CSV | Comma separated values |
| JSON | JavaScript Object Notation |

Table 5.1: Acronyms table