



Universitetet
i Stavanger

Faculty of Science and Technology

BACHELOR'S THESIS

Study program/specialization: Bachelor in Computer Technology	Spring semester 2022 <u>Open</u> /Restricted access
Author(s): Darya Baturyna, Christer Constantin Selbach Iordanescu	
Subject responsible: Karl Skretting Supervisor(s): Karl Skretting	
Title of the bachelor's thesis: Visualisering av stor mengde kartdata for Hafrsfjord English title: Visualization of a large amount of the map data for Hafrsfjord	
Study credits: 20	
Keywords: Python, OpenCV, Qt5, Image processing, big data, GIS, visualization	Number of pages: 54 + attachments/other: 9 Stavanger, 15 May 2022

Abstract

The problem statement is to create an application that can process large and redundant data files received as coordinates of Hafrsfjord along with depth level, such that depth for each point can be shown expediently; with further data visualization as images and depth simulation to show drying of the whole Hafrsfjord as well as the part of it.

The workload is split into two parts: one for the handling of the raw data, and one for the visualization. The raw data files containing the coordinates with their respective depth levels were processed so that they could be stored in two-dimensional data structures. This was done in order for the data structures to be converted into images. Instead of converting the depth coordinates into a single big image, the decision was made to create the image through splitting it into tiles. Received images in the form of tiles of specified size are displayed in the application. All loading is happening in the background to avoid lags due to the amount of images. To show drying of the Hafrsfjord, a depth simulator is created, allowing to change the water level based on the chosen sea level.

The solution to the problem was to generate data structures for each tile, so that when reading through the data files, each coordinate could be appended to its belonging data structure. This resulted in each data structure could easily be converted into a tiled image. The final application includes an easy-to-use user interface, that automatically arranges multiple photos with thought to effectivity, provides depth level and drying features and includes a built-in map editor e.g., grid, pointer, extra map signs, etc.

Contents

Abstract	i
Acknowledgements	vi
1 Introduction	1
1.1 Universal Transverse Mercator (UTM)	2
1.2 What's different about our app?	3
1.3 Technologies	3
1.3.1 Python	3
1.3.2 NumPy	4
1.3.3 PyQt5	4
1.3.4 OpenCV	5
1.3.5 GitHub & Overleaf	5
1.3.6 Collections/Deque	6
1.3.7 Classes provided by Karl Skretting	6

CONTENTS

1.3.8	AppImageViewer1	7
1.4	Further development	8
1.5	Outline	8
2	Manipulation of raw data	10
2.1	Introduction	10
2.2	Constructing a tiled map	11
2.2.1	A tiled image	11
2.2.2	Zoom Level	11
2.2.3	Reading the Files	12
2.2.4	Two-dimensional Numpy Arrays	12
2.2.5	The offset	13
2.2.6	asc/z01FilesMeta	13
2.2.7	A tiled Map	13
2.3	The class: HafrsTiles	15
2.3.1	tileDimension()	15
2.3.2	hdpToTileAndPoint()	15
2.3.3	nTilesInFilesMeta()	16
2.3.4	addTilesToArray()	16
2.3.5	addTilesToDict()	16

CONTENTS

2.3.6	populateDict()	17
2.3.7	curOpen(), curRead(), curClose()	17
2.3.8	addPointsToTiles()	17
2.3.9	createImage()	18
2.4	How to run the program	18
2.5	Results	18
2.5.1	Amount of generated images	18
2.5.2	Time and memory	19
3	Visualization and Graphical User Interface	21
3.1	GUI layout	22
3.2	List of methods	22
3.3	Limitations	26
3.4	Program	27
3.4.1	Zooming	27
3.4.2	Mouse events	28
3.4.3	Navigation/Position information, extra signs & Grid	29
3.4.4	Depth simulation and scaling dialogue	31
3.5	Visualization of images	33
3.6	Testing	37

CONTENTS

4 Discussion	38
4.1 The depth values	38
4.2 The tiled images	39
4.3 Missing values	40
4.4 Building the image	41
4.5 Depth visualization and time complexity	41
4.6 Placement of multiple pixmaps and extra objects	43
4.7 Economy and environmental accounting	45
5 Conclusion	46
Bibliography	48
Attachments	48
A Software	49
A.1 Directory tree	49
A.2 Download links & Requirements	52
A.2.1 Installation	52
B User manual	53
B.1 Application startup	53
B.2 IVE Keyboard Shortcuts	55

Acknowledgements

We would like to thank our supervisor, Karl Skretting, for valuable guidance and expertise throughout this thesis process.

Chapter 1

Introduction

The ancient Greek scientist Ptolemy, in his famous "Guide to Geography" [9], thought that the map, using mathematics, allows us to survey the entire Earth in one image. A depth map, as it can be guessed from the name alone, an image or image channel that contains information relating to the distance of the surfaces of scene objects from a viewpoint [10]. It is needed so that fishermen, travelers can navigate a section of water, also frequently used in hydrogeological observations to track aquatic environment changes and for understanding the depth at the given location, as well as in other parts of the reservoir. The main issue is that most of the information comes in large and difficult to read files, making it almost impossible to analyze or simulate potentially upcoming changes.

Figure 1.1: Thesis

Vi har flere GB med dybde data fra to ulike innsamlinger (fra kartverket) liggende på vårt Unix-system. Disse er til dels overlappende, redundante. Det vi ønsker her er at dere lager et program som kan lese, tolke og ordne alle disse data slik at en for hvert punkt (med oppløsning ned til centimeter-nivå både på posisjon og dybde) kan vise dybder for et valgt område på en hensiktsmessig måte. Jeg tenker at en god måte å visualisere dette på er at en lager (store) bilder med dybde data for hvert pixel, ett bilde ca 8000x6000 pixel stort med et pixel for hver kvadratmeter og så gjerne 100 bilder med detaljer (10 cm oppløsning). I en GUI ønsker vi så å visualisere at Hafrsfjorden tømmes for vann, både for hele fjorden eller for en del av fjorden. Jeg ser for meg at programmeringsarbeidet gjøres i **Python**, med **OpenCV** for bildebehandling og kanskje **Qt** for GUI-program. Det er tilgjengelig noe kode fra tidligere lignende prosjekt, og noe som jeg har laget (for å lese og komprimere datafilene).

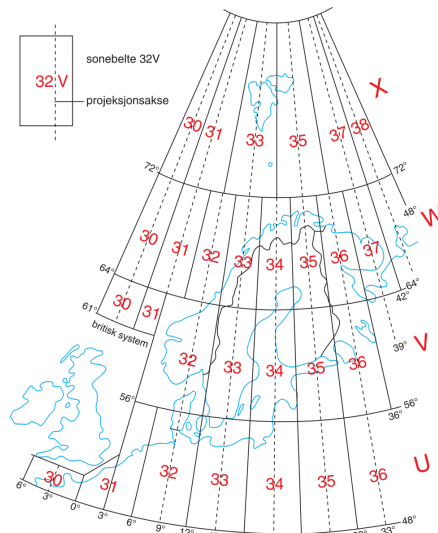
1.1 Universal Transverse Mercator (UTM)

The objective of this thesis (fig. 1.1) is to build a program that can read both large and redundant depth data files, interpret and visualize the data in a clear way. Also, allowing to track and simulate changes in the water level, e.g., simulating devastation of the water in Hafrsfjord.

1.1 Universal Transverse Mercator (UTM)

In our thesis, Hafrsfjord data comes in .asc or .z01 files containing both north and east coordinates given in Universal Transverse Mercator (UTM) coordinates [6]. Universal Transverse Mercator is an international system for map-references and is a two-dimensional coordinate system. The UTM-system covers the world's entire surface and is sectioned into two parts. The first part is east-west and is divided into 60 zones that are 6 longitudes wide. The next part is north-south and is divided into 20 belts, which extend for 8 latitudes. These 20 belts has been given the letters ranging from C-X, excluding the letters I and O. The surface of the world has with this been divided into 1200 (60×20) zone belts [6]. The UTM coordinates included in this project starts with 32V, where 32 is the zone and V is the belt (as shown on fig. 1.2).

Figure 1.2: UTM



1.2 What's different about our app?

1.2 What's different about our app?

The problem statement split into two main parts: raw data and GUI. One of the main goals for the project is to keep everything at a good productivity and efficiency level. In the earlier work with Hafrsfjord, coordinates were stored in a database which made it harder to work with, now all coordinates are stored in separate .asc files. Still, the time it takes to go through hundred million coordinates in .asc files is long, to keep up with the set goal, create a productive and efficient app, the expected running time for all the data to process was set to 14 hours. The time it takes to process all raw data files from point to point was optimized to 12.5 hours.

In case of image visualization, Google Map's[7] concept of splitting image into tiles used to make image processing at different levels more comfortable. The concept goes out with starting at a default level and splitting images into given tiles size while zooming in and zooming out. This technique allows providing more details per tile and increases efficiency during different image manipulations provided in GUI part (image display, color change, depth level simulation, etc.).

And last but not least, both raw data part and GUI are independent and can be run separately, which significantly increases the program's runtime.

1.3 Technologies

This section aims to describe the different technologies and libraries used to present raw data into viewable tiled pictures with further visualization.

1.3.1 Python

Python is an object-oriented programming language that is easy to use, that aims to help users construct clear and logical code.

Python comes with a large array of libraries and frameworks that comes in handy for both big and small-scale projects.[4] In the course of our study in

1.3 Technologies

the university of Stavanger, Python has been the most used programming language, and is a natural choice for us to use in the development of our thesis.

1.3.2 NumPy

NumPy is the central package for scientific computing in Python. It is a Python library for multidimensional arrays objects, which is faster, more flexible and more complex than the built-in Python arrays. It also provides a collection of routines for fast operations on arrays, including logical, mathematical, shape manipulation, sorting, basic linear algebra, selecting and a lot more. [8]

For our project, we were given a set of files containing UTM coordinates. After the files were compressed to 5 percent of their original size, we were left with files that were 1.25 gigabytes large in total, containing 887141602 UTM coordinates.

For this large amount of elements, it would have been too slow to use Python's built-in arrays, so NumPy was a much better choice for the computation.

1.3.3 PyQt5

Qt itself is a large and popular framework on C++, used for creation of graphical user interfaces on all major desktop, mobile and embedded platforms such as Windows, Linux, macOS, Android, iOS and many more.

PyQt is an open source Python binding for the Qt widget toolkit, functions as a cross-platform application development environment. It consists of over five hundred classes covering a decent range of features such as XML processing, networks, Web toolkits, SQL databases and GUI. PyQt is widely used by many large companies from various industries: LG, Mercedes, AMD, Panasonic, Harman, etc.

Most of the documentation written for PyQt is either for C++ language or PyQt4 version, while in this application we are using the newest version PyQt5. Despite not that simple searching work, PyQt5 is a good choice for

1.3 Technologies

GUI due to use of NumPy and OpenCV for image processing.

Another graphical framework that was considered - Tkinter, which is simple and fast in development. But in contrast to Tkinter Qt has its own designer, the set of Qt components is wider and better than in tkinter and last but not the least Qt libs may be more suitable for GUI (e.g. due to asynchrony).

1.3.4 OpenCV

Open Source Computer Vision Library builds on numpy where images are stored in memory as numpy-arrays, which makes it a good choice for image processing. Besides that, OpenCV is widely used in companies, research groups and government agencies. But most of the information is written for C++ same as PyQt5 documentation.

OpenCV has extremely simple user interface, contains a library for Python with a bunch of CV functions and has excellent performance.

1.3.5 GitHub & Overleaf

GitHub is an open source code hosting platform for version control and collaboration. Due to its pros as code storage and sharing, easy team communication and analyzing of activities, it was used as our group's main social network.

The Kanban board technique used in a project management tool by GitHub allows having a full control over each member's tasks and track of progress. On the board itself, each member can drag tasks, add bugs issues or note cards.

The beauty of GitHub version control, lies in easy access to a previous version of the project at any stage of the process, which gives space for secure updates without worrying about losing or destroying any data. Programmers can easily roll back the application and run it locally.

Overleaf is another useful tool to make collaboration on a project easier and

1.3 Technologies

is an analogue of online LaTeX. It provides built-in templates for different document types, allows real-time edition by several members and provides full range of different text modifications, file attachments, adding of code and lists and even more.

1.3.6 Collections/Deque

Deque in python is implemented using the module “collections”. Deque is a double-ended queue, and is preferred over the Python list structure in cases where operations such as pop and append are needed. There is $O(n)$ complexity for pop and append operations on python's lists. As for deque, there is $O(1)$ complexity for the pop and append operations. [3]

Because of the large amount of data in our project, it made it a natural choice to use deque over lists, as this would reduce the time it takes to read all entries by a lot.

1.3.7 Classes provided by Karl Skretting

This section will contain classes that Karl Skretting has provided us to build our program.

HafrsDepthPoint

HafrsDepthPoint is a class used to represent our UTM coordinates in a neatly fashion. It takes three values as parameters; North, East and Depth, and depending on the method chosen, it can represent the point in various ways. For our project, we only needed the point to be represented with the offset subtracted from the north and east values.

1.3 Technologies

HafrsDepthPointList

This class builds on `HafrsDepthPoint` in that instead of listing one point at a time, it appends multiple points into a python list, with an append method that is more memory efficient.

HafrsFiles

The `HafrsFiles` is a class that makes it easier to use the depth data files. The asc-files we were given were uncompressed text-files and are way too large. By using two modules created by Karl Skretting; `clsMyBAzip1.py` and `funMyCompress.py`, which is used for compressing and decompressing files, our asc-files were compressed and converted to z01-files, which are 5 percent of the size of the asc-files.

`HafrsFiles` contains a number of handy methods, but the ones we used were for opening, closing and reading files. Because the asc-files were decompressed and were converted to z01-files, it was not possible to process them without the use of these methods.

1.3.8 **AppImageViewer1**

`AppImageViewer1` is a simple Qt program made as example solution to parts of assignments in *ELE610: Robotic technology* course, which was provided by our mentor as a template to build the application on. The program contains two main classes: `MyGraphicsView` and `MainWindow`.

`MyGraphicsView` class contains basic examples of the processed events and controllers for a mouse such as `mousePressEvent`, `mouseMoveEvent` and `mouseReleaseEvent`; all of the events take cursor position according to the scene into consideration. This class uses some variables that belong to `MainWindow` object.

`MainWindow` is the main class that contains settings for a window, view and scene setup, and some basic image processing methods such as `open`,

1.4 Further development

close and save a file and a more complex method for image cropping.

1.4 Further development

The project we have worked on is done within the frames of our thesis goals. The finished program is buildable and has a potential for further development. There are some features that are left uncompleted due to time restrictions that will be mentioned in a later chapter and could be resolved in further updates.

One of the extra features that could be added is building maps with different geological map references. A more flexible solution could be implemented such that map references could be converted into other map references, making the program independent on receiving only the UTM coordinates. Many famous real-time location sharing apps such as Google Maps, Gule Sider, etc. use World Geodetic System WGS84 [6] standard as the default reference coordinate system and provide conversions to other coordinate systems.

The program is currently not capable of distinguishing heights that are above water levels, known as altitude. In case of wider range of heights and depths, altitude and sea level, an extension allowing to manipulate height data alongside with depth level would be useful. As well as, it can be expanded with GUI visualization of heights based on altitude and special objects or formations (rocks and boulders).

1.5 Outline

The rest of the thesis is outlined as follows:

Chapter 2 presents raw data used in the project, including methods to manipulate data and files alongside with results and solution to solve data processing.

1.5 Outline

Chapter 3 introduces visualization goals, GUI folder tree and program with proposed solution on visualization of processed data.

Chapter 4 presents discussions and evaluation of results against problem statement alongside with alternative solutions.

Chapter 5 summarizes and concludes the work done.

Bibliography and Attachments includes a list of resources used in researching work, directory tree, user manual and required installations and a downloadable zipped file with application.

Chapter 2

Manipulation of raw data

2.1 Introduction

Hafrsfjord is a fjord that resides in Stavanger and Sola. The distance from its most northern to its most southern point stretches for 9 kilometres.[5] We were handed data files containing UTM coordinates that covers the entire fjord.

This chapter will cover what a tiled image is and how to build a program that can generate tiled images by using the large set of data files containing UTM coordinates with corresponding depth values. The goal is to create a program that enables the end application a way to view the whole fjord or sections of it, by enabling zooming.

First comes a description of what a tiled image and map is and mentioning some key features used in the development process. Secondly there will be a description of what the program contains of and how it works.

2.2 Constructing a tiled map

2.2 Constructing a tiled map

2.2.1 A tiled image

Instead of rendering a single image, we broke our image into smaller even sized squared pieces (tiles), that could be put back together, just like mosaic. As an example, think of an image of size $m \times m$. The amount of tiles that can be generated (n) is dependant on the image size (m) and the tile dimension (d). To compute n , the equation used is: $n = m^2/d^2$. To calculate the tile dimension (d), we decided on using a zoom level as input in this equation

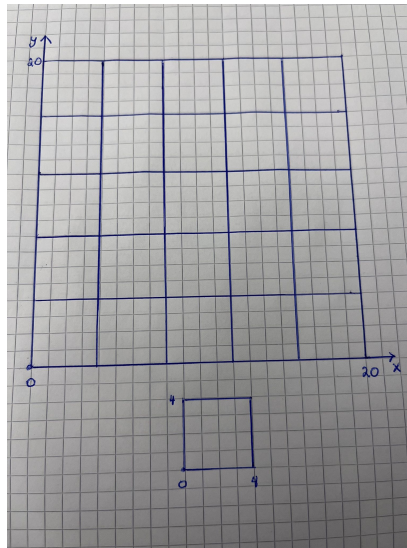


Figure 2.1: 20x20 image split into 25 4x4 tiles

2.2.2 Zoom Level

The zoom level is a metric for how much or little the application is able to increase or decrease the size of the image. The equation for calculating the tile dimensions: $2^{(14 - ZoomLevel)}$

We chose zoom levels that fit into a range between 1 and 9 which returns appropriate tile dimensions for our project. The base or default zoom level,

2.2 Constructing a tiled map

where each pixel represent 1x1 meters, is zoom level = 5. So if you put 5 as ZoomLevel into the equation: $2^{(14 - 5)}$ you will get 512 as a result, which is our base tile dimension.

For zoom level 5, each pixel is 1x1 meters. If you increase the zoom level, you will get smaller tile dimensions with corresponding pixel sizes. For example, if you chose zoom level 9, each pixel will correspond to 1/16x1/16 of a meter or 6.25x6.25 cm. On the other hand, if you decrease the zoom level, the tile dimensions will get bigger. For zoom level 1, the tile dimension will be 8192x8192, and each pixel will correspond to 16x16 meters.

The tile dimension given by our equation also helps to determine how many tiles are required for a given area.

2.2.3 Reading the Files

Before we could start working with our UTM coordinates, we needed a way to browse the text-files. Luckily, Python has an easy and short way to handle files. To read through a file, we used the built-in `open()` function.

The way Python's `open()` functions works, is that it takes a file, and a mode (in our case, we used the mode "r", which is for reading) and in a sequential order, goes through the file, line by line. [2] The beauty of this, is that we now have our UTM coordinates extracted from the file, and ready to be processed and added to Numpy arrays.

2.2.4 Two-dimensional Numpy Arrays

As pictures are most commonly visualized as pixel matrices [1], we chose to build ours as a two-dimensional matrix, using numpy's two-dimensional array.

Here is an example of a UTM coordinate: 6534883.10 305802.71. The first number is a north coordinate, and the second is an east coordinate, and together they form a point. This point has a depth value assigned to it, and is what is added to the matrix. One can define a numpy array as such: `np.zeros(X, Y)`, which will create an X times Y matrix, where X are rows and Y are columns. In our case, north is represented as rows and east is represented as columns.

2.2 Constructing a tiled map

Figure 2.2: A 2-dim Numpy array

```
two_dimensional_numpy_array = np.array([
    [0, 0, 0], [0, 0, 0], [0, 0, 0],
    [0, 0, 0], [0, 0, 0], [0, 0, 0],
    [0, 0, 0], [0, 0, 0], [0, 0, 0]
])
```

2.2.5 The offset

In this project, we use an offset of: (6530000, 300000), which is a point located south-west of Hafrsfjord.

What we did with the offset was to subtract it from the UTM coordinates to get more manageable numbers to work with, as its only the 4 last digits in the coordinates that will vary within the area that Hafrsfjord is located.

2.2.6 asc/z01FilesMeta

ascFilesMeta or ascZ01FilesMeta are lists containing metadata from all the asc or z01 files. Each entry in the list has a file name, number of points within the file and maximum and minimum points.

Figure 2.3: The 15 first elements in the asc/z01FilesMeta

```
142 ascZ01FilesMeta = [['110asc.z01', 6537695, 6538191, 305913, 306226, 15553086],
143                    ['111asc.z01', 6537686, 6538199, 306201, 306519, 21142843],
144                    ['113asc.z01', 6537312, 6537708, 305927, 306202, 13926131],
145                    ['114asc.z01', 6537126, 6537696, 306179, 306454, 23198325],
146                    ['119asc.z01', 6536584, 6536798, 305825, 306121, 16081483],
147                    ['124asc.z01', 6536563, 6537128, 306344, 306557, 14695190],
148                    ['125asc.z01', 6537113, 6537611, 306428, 306693, 18272300],
149                    ['130asc.z01', 6536039, 6536599, 305781, 305890, 12467892],
150                    ['131asc.z01', 6536026, 6536596, 305861, 306142, 18317744],
151                    ['132asc.z01', 6536013, 6536573, 306116, 306401, 21287973],
152                    ['133asc.z01', 6536000, 6536570, 306372, 306657, 7890590],
153                    ['134asc.z01', 6535991, 6536557, 306628, 306912, 11454592],
154                    ['135asc.z01', 6534886, 6535128, 307348, 307816, 20698073],
155                    ['136asc.z01', 6535108, 6535406, 307353, 307880, 18495573],
156                    ['137asc.z01', 6535368, 6535631, 307879, 308118, 13942152],
```

2.2.7 A tiled Map

The design process of creating a tiled map is to first choose a map size and tile size. For a map with 20x20 as dimensions and a tile size of 4x4 would

2.2 Constructing a tiled map

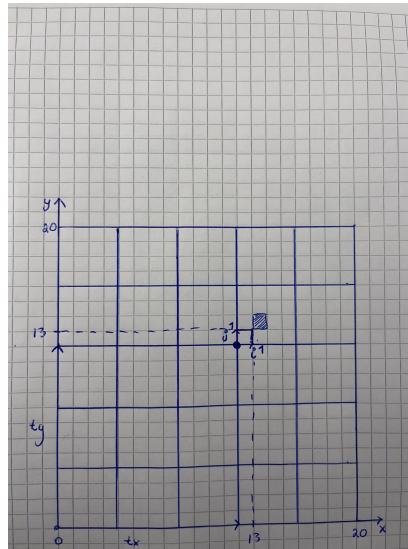
generate 25 tiles. A map with these dimensions has 400 pixels, which is equal to the amount of pixels contained in 25 4x4 tiles ($4*4=16$ pixels in one tile, and $16*25 = 400$ pixels).

This can be thought of as an x/y coordinate system with x and y values ranging from 0 to 20, which is filled by 25 4x4 quadrants. Now that the map is split into quadrants or tiles, it is possible to assign each tile a coordinate. Assume that each tile has its own x/y coordinate system, and that its (0,0) value lies at the bottom left side. This makes it possible to assign each tile a coordinate at its (0,0) value, which makes it possible to navigate the map by tiles.

So if we want to plot for example p1 that has the values (13, 13) in this map, we look for the tile which corresponds to p1, and that is tile (3,3). Within this tile, the point resides at (1,1).

The tile coordinate is calculated by floor dividing each element in a point with the tile dimension; $13 // 4 = 3$. The point within the tile is calculated by taking each element in a point and using the modulus operator with the tile dimension; $13 \text{ modulus } 4 = 1$.

Figure 2.4: Tile coordinates and points within the tile



2.3 The class: HafrsTiles

2.3 The class: HafrsTiles

HafrsTiles is a class made to manipulate the raw data, so that they can be viewed as tiled images by creating methods for extracting relevant information from the different data files. The class has three instance attributes; tileDict for storing the different tiles as numpy arrays, tileArray for storing all the different tiles as tuples, and level which is used to set the tile dimension.

2.3.1 tileDimension()

tileDimension() is a simple helper method that returns the tile dimension by using the value of level, which by default is set to 5. It uses a simple equation to calculate the tile dimension: $\text{tileDimension} = 2^{(14 - \text{level})}$

2.3.2 hdpToTileAndPoint()

The hdpToTileAndPoint() method is in a way the engine behind the program. It takes an UTM coordinate with its respective depth value as a parameter along with the offset and converts the point into tile coordinates, and where within this tile the point resides.

Take this point as an example which has the form (North, East, Depth): (6534882.66,305802.76,11.18). To convert the point into a tile coordinate, first off the offset needs to be subtracted from the north and east values and then floor divided by the tile dimension. Secondly, to get the values from the point within the tile, the procedure is almost the same, except instead of using floor division, we use the modulus operator. This would result in our point being converted into two tuples; (9,11) which is the tile coordinate and (274,170) which is the coordinate within the tile. Lastly, the depth value is multiplied by 100 so that it is converted from meters to centimetres. So for level 5 which has tile dimension = 512, the point (6534882.66,305802.76,11.18) lies in tile (9,11) which is $9 \cdot 512$ north of origo (0,0) and $11 \cdot 512$ east of origo. This tile covers an area of $512 \cdot 512$, and within this tile, our point is located at $(6534882.66 - 6530000(\text{north offset}))$

2.3 The class: HafrsTiles

- 9*512 which gives its north coordinate of 274, the same procedure is done for the east coordinate with the east offset, and will give the point 170.

2.3.3 nTilesInFilesMeta()

This method uses `hdpToTileAndPoint()` as a helper method to calculate the number of tiles that an asc or z01 file contains, by taking the max/min entry in the asc/z01FilesMeta as a parameter. I created this equation that calculates how many tiles a file consists of using the asc/z01FilesMeta entries: $n = (\text{maxNorth} - \text{minNorth} + 1) * (\text{maxEast} - \text{minEast} + 1)$, where n is the number of tiles within a file, `maxNorth` is the highest north value, `minNorth` is the minimum north value, `maxEast` is the highest east value and `minEast` is the minimum east value.

2.3.4 addTilesToArray()

`addTilesToArray()` is a method that calculates all existing tiles contained in a asc or z01 file and appends them to the `tileList` attribute. By taking an asc/z01 meta value as a parameter, the method looks at the max and min values for its north and east coordinates to determine every individual tile that the file covers.

2.3.5 addTilesToDict()

This method adds the tile values from `tileList` to the attribute `tileDict`. `addTilesToDict()` iterates through the `tileList` attribute and converts them into key:value pairs. The keys are made up of the current level appointed to the class and the north and east coordinate that the tile belongs to. The keys are assigned a Numpy array with dimensions that corresponds to the level assigned to the class. Here is an example of how the key:value pairs look like for level 5 within the dictionary; `'HafrsNi5N4608E5632' : 'np.zeros((512,512),np.dtype=int16)'`

2.3 The class: HafrsTiles

2.3.6 populateDict()

This is a simple method that iterates through the asc/z01 meta list and populates tileDict by using addTilesToDict() for each iteration.

2.3.7 curOpen(), curRead(), curClose()

These are methods created by Karl Skretting for opening, reading and closing files. CurOpen() prepares a file for reading by using python's open() function, and curClose() closes a file by setting the dFile attribute to None which just tells the program that there is currently no active file open. For .z01 files, curRead() will read a minimum of 3000 lines from a file and append them to the deque, as for .asc files, curRead() can be set to more wide variety of values.

2.3.8 addPointsToTiles()

This method starts by adding the correct file-path to the catalogs list, where the z01 files are stored.

By calling the populateDict() method, all previous mentioned methods from the HafrsTiles class will be called to form the tileDict.

The method iterates through the asc/z01FilesMeta, calls curOpen() and reads through each file by calling curRead(), adding its data (UTM:depth value pairs) to the deque. From here the data are removed by using the function popleft() which returns the deque's leftmost item. The item returned gets processed by hdpToTileAndPoint() that generates information required to access the correct key in the tileDict and adds points to its respected numpy arrays. This is done as long as there are points left in the deque, and when the file has been read through, a call to curClose() is initiated, and a new iteration can commence.

After a successful call to this method, all of the Numpy arrays in tileDict should have been populated with depth values.

2.4 How to run the program

2.3.9 createImage()

This method iterates through tileDict and uses OpenCv's imwrite method to convert all the numpy arrays into images. It takes an image file extension and a file path as parameters.

2.4 How to run the program

The program can be run by creating an instance of the class and passing it a level, or if the level is not specified it will be run with the default level of 5.

A call to the method addPointsToTiles() with the asc/z01FilesMeta as parameter will populate all numpy arrays with points as long as all the .z01/.asc files are within the specified file path, and a call to createImage() with a file-extension as parameter will produce all the tiled images and store them in the current file-path. If no file-path is entered into createImage(), the images will be saved in the current working directory.

Figure 2.5: How to run the program

```
def runProgram():
    h = HafrsTiles(2)
    h.addPointsToTiles(ascZ01FilesMeta)
    h.createImage(".png", '/Volumes/KINGSTON/ImageFilesLevel2')
```

2.5 Results

2.5.1 Amount of generated images

The amount of images generated is dependent on the level used in the program. For higher levels more images will be generated and visa versa.

The different amounts of images generated by the different levels is shown below:

2.5 Results

Level 2: 5
Level 3: 11
level 4: 21
level 5: 57
level 6: 183
level 7: 660

2.5.2 Time and memory

Because the program had to read and process very big amounts of data, the time it took to run the program took around 750 minutes or 12 and a half hours. The dominant factor for the high run-time is reading through all the files and adding points to the numpy arrays. To convert all the numpy arrays into images using opencv had a negligible impact on the run-time.

To process all the data files for all 9 different levels, the run-time would be at a total of around 112.5 hours.

The amount of memory the images occupy is dependent on the level. There are a lot more images generated at high levels, but these images takes less memory as they are of lower dimensions. At lower levels, fewer images are generated, but each image is of a greater size and therefor occupy more memory. The amount of memory for all the level 7 images comes to a total of 10 MB, as for level 2, the total amount is 337 KB.

2.5 Results

Figure 2.6: Some run times for reading and appending points.

```
(py38) christers-mbp-2:clshafrs christeriordanescu$ python clshafrsfiles.py
It took 744 seconds to read 15553086 points from 110asc.z01
It took 1011 seconds to read 21142843 points from 111asc.z01
It took 668 seconds to read 13926131 points from 113asc.z01
It took 1118 seconds to read 23198325 points from 114asc.z01
It took 761 seconds to read 16081483 points from 119asc.z01
It took 724 seconds to read 14695190 points from 124asc.z01
It took 887 seconds to read 18272300 points from 125asc.z01
It took 594 seconds to read 12467892 points from 130asc.z01
It took 862 seconds to read 18317744 points from 131asc.z01
It took 1056 seconds to read 21287973 points from 132asc.z01
It took 398 seconds to read 7890590 points from 133asc.z01
It took 582 seconds to read 11454592 points from 134asc.z01
It took 985 seconds to read 20698073 points from 135asc.z01
It took 911 seconds to read 18495573 points from 136asc.z01
It took 683 seconds to read 13942152 points from 137asc.z01
It took 303 seconds to read 6265165 points from 138asc.z01
It took 414 seconds to read 8563676 points from 140asc.z01
It took 718 seconds to read 14528361 points from 143asc.z01
It took 646 seconds to read 13082540 points from 144asc.z01
It took 923 seconds to read 18227611 points from 149asc.z01
It took 498 seconds to read 9847489 points from 150asc.z01
```

Chapter 3

Visualization and Graphical User Interface

The main goal for the visualization part is to efficiently display processed data in form of tiles and show that Hafrsfjord is emptied of water.

Visualization and Graphical User Interface section provides full information about the final program layout, features provided for a user-friendly interaction, methods and techniques for visualization of the processed data and more. Most part of the visualization concentrates around image display, storing, manipulation and mainly on simulation of water level. Hafrsfjord is emptied of water, which makes visualization of expected drying out an important feature.

In creation of the visual part of an application, the QT library uses. The Qt library includes the Qt resource system, which is a convenient way of adding binary files such as icons, images, translation files, and other resources to applications. For image manipulation such as conventions from *numpy array to QImage*, *QImage to QPixmap* and other way, PyLab library including NumPy was used. Features such as depth simulation, colormaps and change of image channels done with help of OpenCV.

More detailed discussion and overview of what was used and removed, alternative solutions will be taken in section 4.

3.1 GUI layout

3.1 GUI layout

Full list of folders and files is provided under, full dictionary tree of the whole application and detailed information can be found in Appendix A.1:

```
bachelor_2022
├── ...
└── GUI
    ├── Images
    ├── QImagesMethods.py
    ├── app.py
    ├── clsColorDialog.py
    ├── stack.py
    └── stylesheet.qss
```

3.2 List of methods

Contains a list of features that are included in the program and description on how they are connected and contribute to a total solution and how they are built up.

app.py: The main program for the application contains all methods to open file, show tiles and different image manipulation methods such as cropping, zooming, color change, depth value simulation, etc, Contains two classes: *QGraphicsView* class with mouse events and *MainWindow* class with the main program.

* Not all methods are listed in the table, details on *QGraphicsView* will be taken in section 3.4.2 as well as methods for the grid and pointer.

3.2 List of methods

main_actions(), main_menu(), setMenuItems()	sets actions on the main menu; setup main menu; setup for the menu items.
toolbar(), showToolBar(), statusBar()	setup for the toolbar; enables/disables toolbar; setup for the status bar.
addExtra(), assignSign(), undoSigns(), deleteAll()	enables/disables add extra feature; creates a pixmap of the sign which will be used in <i>QGraphicsView.mousePressEvent()</i> and updates cursor; deletes last added sign from the scene; removes all added signs to the scene.
sceneMoved()	enabled whenever scene is scrolled, updates visible tiles by calling <i>chooseImage()</i> method, draws back grid if the one existed.
openFile(), openFileDialog()	allows choosing an .asc file and to display processed image on the scene.
closeWin(), save/printFile()	closes the window, saves or prints the scene (only the visible tiles will be saved/printed).

3.2 List of methods

<p>showHafrsfjord()</p>	<p>Main method to display processed tiles, also used to display tiles when <i>zoomIn/-out</i> is called. Uses <i>pixmap2image2np_multi()</i> to update qimage and npImage array. Calls <i>chooseImage()</i> that checks and updates the scene if image should be displayed. Resizes scene <i>sceneResize()</i> and updates menu items <i>setMenuItems()</i>.</p>
<p>pixmap2image2np_multi(), np2image2pixmap_multi()</p>	<p>used to convert between pixmap-qimage-nparray when one of them is updated.</p>
<p>removeAllPixmaps()</p>	<p>removes all pixmaps from the scene.</p>
<p>changeDepth(), tryDepth(), greyColormap() and changeCmap()</p>	<p>First a <i>changeDepth()</i> creates an object of class that triggers <i>tryDepth()</i> for a quick update of the visible tiles on the scene. Beforehand, images turned to grey by <i>greyColormap()</i>. Chosen colormap is applied by <i>changeCmap()</i>. Depending on results from created object, <i>changeDepth()</i> updates or resets the scene.</p>

clsColorDialog.py: Contains a program for dialogue window called in **app.py** by *MainWindow* class in the *changeDepth()* method. The dialogue allows quick test of the changed values by *tryDepth()* method called by **clsColorDialog.py** when either slider value is changed, colormap assigned

3.2 List of methods

or scaling factor is updated.

scaleUp(), scaleDown(), scaleReset(), tryScale()	called on buttons when clicked, either scales an image up, down or resets. When <i>tryScale()</i> is called, it updates scaling factor and calls <i>tryClicked()</i> method inside <i>clsColorDialog.py</i> .
okClicked(), cancelClicked()	either sends 1 if accepted (OK), 0 if rejected (Cancel) in <i>changeDepth()</i> when <i>ColorDialog().result()</i> is called.
tryClicked()	gets threshold value, colomap index and calls <i>tryDepth()</i> method with those values for a quick visualization of the image/-s.
sliderMoved(), spinChanged()	updates threshold value based on slider value; updates slider value based on the spin; both call <i>tryDepth()</i> method with updated threshold value.
getValues()	return threshold value, called by <i>changeDepth()</i> method to get the threshold value.

stack.py: Class object created and called by **app.py** *MainWindow* class, used to delete all or last added sign to the scene.

3.3 Limitations

push()	adds an element to the top (end of the array).
peek()	returns 1st top/last element of the array.
pop()	removes top element and returns its value.
isEmpty() and size()	return True if no elements in a stack; returns stack size.

QImageMethods.py: Contains methods such as conversion between QImage and image numpy array. Called by *MainWindow* class in **app.py** for methods: `QPixmap2image2np_multi`, `np2image2pixmap_multi` (for Hafsfjord tiles) and `QPixmap2image2np`, `np2image2pixmap` (for one image).

qimage2np()	takes QImage as input and returns numpy array of an image if successful.
np2qimage()	takes numpy array of an image and converts to QImage, return QImage if successful, else empty QImage.

Rest of **GUI** chapters provide feature limitations, program overview and finally testing.

3.3 Limitations

Zooming Due to time consumption to create tiles for each level, the application is using pre saved image folders, two levels over default and two less than default level. In the first place, **Zooming** was not expected to have any limitations, resizing of an image is now limited by 2 zooming levels both ways.

3.4 Program

Scaling allows user to choose the wanted scaling factor from the checkbox. The scale factor >1 indicates shrinking, and the scale factor <1 stands for stretching. For the best scaling results that do reflect the general case, it's common not to use nice numbers such as 0.5, 2 or 4, etc. as they can edge cases. To find unique factors, a current resolution of an image, $w \times h$, would be divided by the expectedly resized width and height returning `scale_x` and `scale_y` factors. But in this project it's hard to predict the next best resolution mainly because of the variety of image modification, therefore basic scaling factors of 2, 6, 14, 18 and 26 in square root are used.

Scaling can be easily activated with *Ctrl+wheel*. When setting up a wheel, a problem occurred, the wheel event wasn't able to scale when the scrollbar appeared. To solve this problem, the wheel event is tracked by event filtering.

3.4 Program

Let's take a look at the program itself.

3.4.1 Zooming

The amount of information available and size of a tile on each zooming depends on a zoom level, activate from the *View* or by pressing *Ctrl + +/-*. As a default start point, zoom level is 5 which corresponds to a tile of a size 512 pixels. Percent visualization is used to track changes in zoom level due to its common use in mostly all application. Percentage is displayed on the status bar at the button of an application.

When one of the zooming functions is triggered, image level is then going one level up or down. If zooming is not successful, level and tiles size will reset. A folder with images equivalent to the level received will be handled by `showHafrsfjord()` method. If level is greater than the default one, >5 , the level parameter changes to $+1$ and tiles size is doubled. Same rules apply to lower level, <5 , the level updates to -1 and tiles size is now halved. Zooming is equivalent to basic level 5 visualization of multiple images. Therefore, this part will be taken together with subchapter 3.5, Visualization of images.

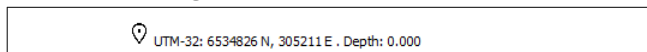
3.4 Program

3.4.2 Mouse events

For simplicity, most of the features provided in the IVE application bound to the mouse, either to start an event or to end it. In this subchapter, only basic interactions of *MouseMoveEvent*, *MousePressedEvent*, *MouseReleased* and *WheelEvent* will be discussed. More details will come in a separate feature subchapter

MouseMoveEvent allows displaying current coordinate position based on a mouse position on a scene and its depth. All information, immediately, shows on a status bar at the bottom of the application. *MouseMoveEvent* is called whenever the mouse moves over the view.

Figure 3.1: Coordinates for cursor



MousePressedEvent is used mostly for all the features, some of them requires direct interaction with the menu bar to start. Also, used for activation of rubber band used in image cropping. Actions to crop an image, add a grid or a point are first triggered from the *View* menu. Grid will be immediately displayed, for the pointer, it requires to be added to the scene by *Left*-interaction with the scene, or *Right-click* to remove a pointer. When mouse is pressed, the pointer can be moved all over the scene til released. If nothing mentioned is activated, *Drag and move* event starts, allowing to scroll with the mouse all over the scene in both directions.

MouseReleasedEvent called whenever a mouse button is released after being pressed in the view. Currently, used for image cropping function to deactivate rubber band triggered by *MousePressEvent* and to get coordinates where the cropping should end. Also, this event takes part in *the drag and move* method, which stops when mouse is released. If the pointer is active, releasing the mouse updates the position of the cursor and calls *QMessageBox* to show detailed information of the chosen location.

And lastly, *WheelEvent* called whenever the mouse wheel is rolled forward or backward, calls *self.scaleUp* and *self.scaleDown* functions when combined with *Ctrl + +/-*. A positive value indicates that the wheel was rotated forwards away from the user, and a negative value indicates that the wheel was rotated backwards toward the user. To mention, mouse wheel intercept with scrollbar which appears when scene is getting larger, therefor wheel

3.4 Program

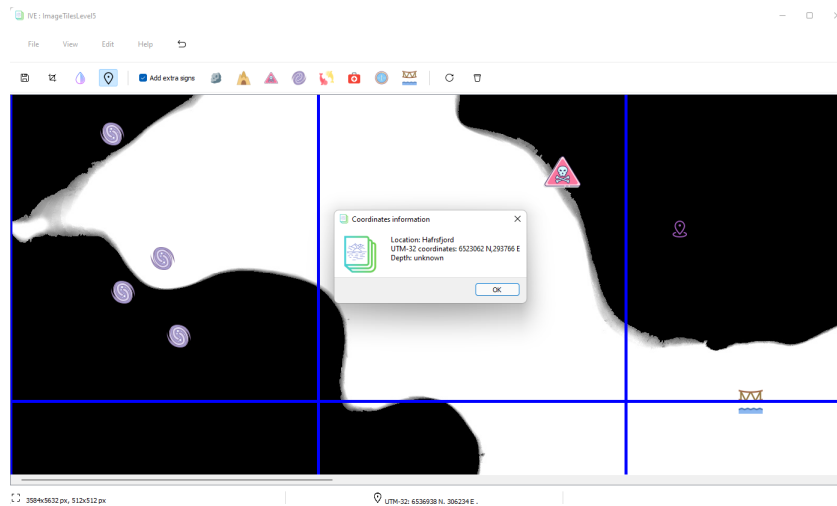
event was modified to *Ctrl+wheelUp* and *Ctrl+wheelDown*. Scaling function scales at the approximate mouse position and is limited to 4 times scaling, but scaling factor can be modified and tested in *Colormap and scaling* dialogue. *angleDelta().y()* method provides the angle through which the mouse rotated since the previous event.

All *MouseEvent*s work only when scene is not empty. And coordinates used in the events are received from *event.pos()*, it returns the position of the mouse cursor, relative to the widget that received an event. Mainly all parameters on the status bar, coordinates with depth value and image pixels with zooming percentage, are updated by *MouseEvent*s.

3.4.3 Navigation/Position information, extra signs & Grid

Position marker: Navigation is possible from the toolbar by clicking the location button. An information box created with *QMessageBox* library will pop up, and a marker will be set at the given position. To close navigation mode, either click on the location button from the toolbar or *right-click* anywhere on a pixmap.

Figure 3.2: Grid, navigation point and extra signs



Another useful feature is a **grid**, available from *View* menu for activation,

3.4 Program

Figure 3.3: Grid, navigation point for one image



deactivates the same way. North-east coordinates are displayed alongside the grid. Grid size updates whenever *self.zoomUp* or *self.zoomDown* is called. To create a more visual friendly and precise grid, *showGrid()* method is called to check whether any grid exists if so make it visible, else draw a new one. By using *max_min()* method that finds minimum and maximum values of the tiles positions and then based on that creates a grid. Minimum and maximum values are updated on each call of *sceneMoved* when scrolled, allowing to create a dynamic grid. To draw a grid, another method used *drawGrid()*, lines are added by *addLine()* and appended to the array that keeps track of the grid lines on the scene.

```
1     qline = self.scene.addLine(line, -self.min_x + ...
      self.tilepx, line, -self.max_x, pen)
2     self.gridLines.append(qline)
```

Extra signs is a new feature to add extra details or information to make tiles. Works similar to the pointer. To activate the menu, *Add extra signs* box should be checked, called by *addExtra()*. When a sign is chosen, a method called *assignSign()*, sign is formatted to the pixmap and the cursor is updated. To reset the cursor, right-click on the scene. Signs can be either removed one by one or remove all items, last added elements will be removed first. Signs are stored in a (First in-Last out), which simplifies removal.

Navigation points, extra signs and grid are created can be displayed together. To create a more precise position marker instead of top-left placement, *translate()* method is used on x and y coordinates, which allows draw-

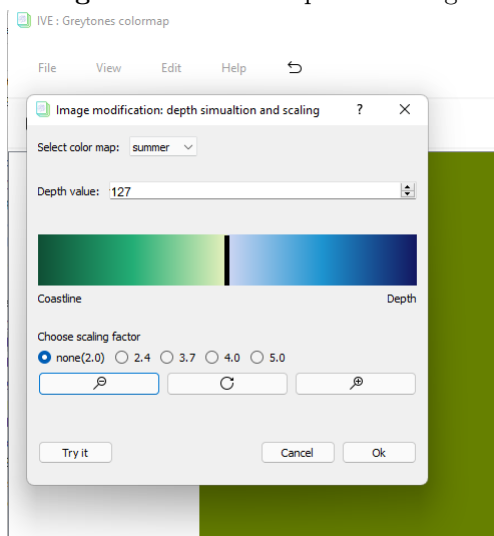
3.4 Program

ing in the center of a cursor. All elements, grid, points and icons, are independent on the scene and can be removed without effect on other elements. To avoid any overlay when zooming or updating an image in *Colormap and scaling* dialogue, *self.resetGridPointer()* method is called which resets grid and point parameters by calling *self.showGrid()* and *self.setPointer()*. Extra signs can be removed by clicking on the basket on the toolbar.

3.4.4 Depth simulation and scaling dialogue

Depth simulation, colormaps and scaling can be accessed from *View* menu - *Colormap and scaling* or simply by key combination *Ctrl+D*. For more efficient work with images, all of them are turned into greytone ones before colormaps are applied. *Colormap and scaling* dialogue provides 4 types of image modifications: choose one of the suggested colormaps, simulate depth level, change scaling parameters and scaling itself.

Figure 3.4: Colormap and scaling dialog



ColorDialog class contains setup and methods for *Colormaps and scaling* dialogue, is a descent of *MainWindows* class and *QWidget*. When dialogue is activated, the object of *ColorDialog* is created. For this class, extra libraries have been imported such as *QSlider*, *QLayouts*, *QSpinBox*, *QLabel*, *QComboBox*, *QRadioButton*, *QButtonGroup*, etc.

3.4 Program

Firstly, the range of colormaps displayed as droplist is created. Each item in a list has its own id. When the user chooses from the list, the object is triggered, calling *self.tryClicked* method. Index of recently changed colormap is then received by *self.selectmap.currentIndex()* which returns id of chosen colormap. Also, whenever the slider changes its value, the same method, *self.tryClicked* is called. Threshold and slider values are equal, therefore when slider value changes, threshold value sets equal to slider in *self.sliderMoved*. As well as, when threshold value manually changes, slider is set equally by *self.spinChanged*.

Both colormap index value and threshold are parameters in the method *self.tryDepth*, called by *self.tryClicked*. This method allows the user to quickly show results of colormap and threshold value, in other words the user can choose colormap or use default one and simulate changes in depth level by simply changing t value (=threshold value). Before any changes may apply, the image is converted to gray singled channelled image. In *self.tryDepth*, t value received from *ColorDialog* class is then set into *cv2.threshold(image, thresh=t, maxval=255, type=cv2.THRESH_BINARY)* where image is numpy array for an image, thresh stands for threshold value, maxval sets all pixels greater than t value to its value and type is a threshold type to be used. For THRESH_BINARY, the threshold value is a unique number, all pixel intensities less that t value will be assigned 0, all pixels greater that t sets to max value, usually 255. This method simulates changes in depth level by manipulating gray tones.

Human eyes are not built to observe fine changes in grayscale images, therefore a greyscaled image will be recolored. Colormap id parameter received from *self.tryClicked* is then used in calling another method, *self.changeCmap*, which takes in id together with image numpy array and applies colormaps depending on id received. To mention, image numpy array remains the same, no changes will occur til *Ok* is clicked. OpenCV defines 12 colormaps that can be assigned to greyscaled images using *cv2.applyColorMap()*. But in the output color reverses due to OpenCV's sequence being in BGR, therefore *cv2.COLOR_BGR2RGB* method calls on an image to convert bgr to rgb to achieve expected colormap. And to make it all show on a pixmap, *np2image2pixmap()* sets all required parameters but as long as *numpyAlso* is False, image numpy array is not modified.

```
1 self.greyColormap() #updates to grayscale if not yet
```

3.5 Visualization of images

```
2 #update threshold value, updates all but not numpy array
3 (used_thr, B) = cv2.threshold(im, thresh=t, maxval=255, ...
                               type=cv2.THRESH_BINARY)
4 self.np2image2pixmap(B, numpyAlso=False)
5
6 #snippet of the code for colormap (multiple tiles)
7 cmap = cv2.COLORMAP_SUMMER
8 satelight = cv2.applyColorMap(im, cmap)
9 satelight_vr = cv2.cvtColor(satelight, cv2.COLOR_BGR2RGB)
10 if key:
11     self.np2image2pixmap_multi(key, satelight_vr, ...
                                 numpyAlso=num)
12 else: #when only one image
13     self.np2image2pixmap(satelight_vr, numpyAlso=num)
```

When the user is finished with all simulations, one of the three *QPUsh-Buttons* is triggered. Depending on *ColorDialog*. Result, pixmap will be backed up to its previous version or by receiving the final *t.getValues()* threshold, change colormap, update all other features and save new pixmap, `numpyAlso` parameter in *np2image2pixmap()* will be set to `True`, therefore image numpy array will be updated.

Lastly, user can change scaling factor depending on preferred resizing and scale in/out directly from the dialogue. All scaling can be reset with *Reset* button. As mentioned before, scaling both ways is possible up to 4 levels.

3.5 Visualization of images

By accessing *File* menu - *show Hafrsfjord*, *self.showHafrsfjord* method is triggered. In order to get started with visualization of images, a method for iteration of all image files in a specified folder is expected, which requires some extra libraries to import. Depending on the zoom level the amount of files is dynamically increasing or decreasing, those circumstances make any manual path updates inefficient.

In *self.showHafrsfjord*, *os.listdir()* method is called to iterate through the images and print the names in order. To get started some imports have to be done, operating module *os* to interact with the operating system and to get access to the folders *listdir()* are imported. As mentioned, in the application

3.5 Visualization of images

Figure 3.5: Black-white Hafrsfjord image, level 5



all images are saved in .jpg and few in .png format, therefore only .jpg and .png files will be loaded by using the *endswith()* restriction function. The concept of displaying multiple images and furthermore modification of the following lays in adding *QGraphicsPixmapItems* to the scene. Before any pixmap is added to the scene, a check is triggered, in case there is something, *removeAllPxmmaps()* method will take care of it by removing all added items. Hence, method is also used for zooming in and out, everything is set dynamic. All image information is stored in a dictionary with (east, north) as a key and contains an array as a value.

```
1
2     #snippet of the showHafrsfjord()
3     self.imageDict = {}
4     self.visible = {}
5     self.min_y=self.min_x=10000
6     self.max_y=self.max_x=0
7     if len(self.scene.items()) > 0:
8         self.removeAllPxmmaps(all=True)
```

3.5 Visualization of images

```
9         first_el = True
10        x, y = 0,0
11        path = "../Gui/Images/ImageTilesLevel"+str(self.level)+
12              "/ImageTilesLevel"+str(self.level)
13        sceneRect = ...
14              self.view.mapToScene(self.view.rect()).boundingRect()
15        for images in os.listdir(path):
16            image_info = []
17            impath = join(path, images)
18            if (images.endswith(".jpg") and ...
19              isfile(impath)) or ...
20              (images.endswith(".png") and isfile(impath)):
21                image_info = []
22                self.pixmap.load(impath)
23                if self.pixmap.isNull():
24                    print("Couldn't load an image")
25                else:
26                    north = int(images[9:13])
27                    east = int(images[14:18])
28                    self.pixmap2image2np_multi() #updates ...
29                    qimage and npImage array
30                    image_info.append(self.pixmap)
31                    image_info.append(self.prevPixmap)
32                    image_info.append(self.image)
33                    image_info.append(self.npImage)
34                    image_info.append(None) #acts like ...
35                    self.curItem for each image
36                    self.imageDict[f"{east},{north}"] = ...
37                    image_info
38                if first_el:
39                    first_el = False
40                    x, y = east, north
41                    self.firstx, self.firsty = east, north
42                    self.chooseImage(east, north, east, ...
43                                    north, ...
44                                    sceneRect.width(), sceneRect.height())
45                continue
46            self.chooseImage(east, north, x, y, ...
47                            sceneRect.width(), sceneRect.height())
```

All mentioned, main information such as pixmap, previous pixmap, qimage and numpy array and object of the scene item (set to None), for each image is stored in a dictionary, *self.imageDict*. Dictionaries mainly have searching time complexity at $O(1)$ and items can be searched by half key. This feature may be quite useful in case searching by one of the coordinates. When it comes to images received, each image name contains image level, north

3.5 Visualization of images

and east coordinates. A default level is 5 with tiles size of 512 pixels, tiles position can be counted by dividing north and east coordinates by tiles size, in the default case by 512. In case of image display, *QGraphics* coordinates system start in the left-top corner with (0,0) origin. While images were created with the thought of left-bottom corner with (0,0) origin. To handle correct image display, coordinates are scale as (1,-1).

Visualisation of multiple images is built on *GraphicsPixmapItem(self.pixmap)* that creates an item that will display the given pixmap, *self.scene.addItem(...)* adds the given item to the scene. Depending on the amount of images that are expected to load, lags may occur. To solve this problem, images are loaded in the background while showing only tiles inside the visible area. To handle any movement, all changes in scene are caught by *self.view....ScrollBar().valueChanged.connect(self.sceneMoved)* which references to *self.sceneMoved* method. With new x and y, old x and y, visiblewidth and visibleheight, we have all to update tiles. But this will work for later images. In the main function, *showHafrsfjord*, a new element is added, *sceneRect*. *sceneRect* returns origin of current view, width and height. Since the chance for displaying some images from the start is very low, (x,y)-start points from *sceneRect* are too low for values that are used in an app, display of images will be built around the first image in a file. The main goal is to find a north-east pair that is inside the range which is handled by *self.chooseImage()*:

```
1 sceneRect =
2     ...self.view.mapToScene(self.view.rect()).boundingRect()
3 for key in self.imageDict:
4     if key in self.visible:
5         continue
6     east, north = self.key_conversion(key)
7     self.chooseImage(east, north, int(abs(sceneRect.x())),
8                     ... int(abs(sceneRect.y())), sceneRect.width(),
9                     ... sceneRect.height())
```

Looking through *self.imageDict* to find key-pairs. First, the check is done on the key, to find if it's already visible. If not, call *self.chooseImage()* to check if key-pair matches the requirements and if so, display and add to the scene. The Final Hafrsfjord image is shown in 3.5.

3.6 Testing

3.6 Testing

As a starting point in GUI programming and setup, a simple image processing program was used, *AppImageViewer1*. It contains some basic options such as Open file, Save as, Close the program. For the testing purposes, *AppImageViewer1* program was used as a template and rewritten to use. First, when Open file was clicked, a window allowing to choose an image was popped up. The chosen image was then displayed on a pixmap. The image used for testing was of expected in the future image size format, 3000×2500 pixels.

After an update, when Open file was triggered, it asked a user to choose an asc-file and referenced the received information to an App.py file that contained *Hafrsfjord* class. From the given information, a numpy-array is created and converted into an OpenCV image that was then returned to GUI's *MainWindow* class. This feature still remains in the final project as a potential for testing and future update of an app.

The image received is then displayed in a pixmap. Later the application was updated, and extra class *Tiles* were added to split the received image into tiles of a given format by simply calling *zoomUp* function. The idea was to simulate splitting of an image and displaying several tiles on a pixmap.

Similarly, a class for *Colormap and scaling* dialogue created, it contains its own main method to start the dialogue and some basic test methods. Each class has its own tests inside the *main* method. App accepts .asc files, images or just shows Hafrsfjord. All methods remain in the final product folder, with thought for future update.

Chapter 4

Discussion

This chapter will contain a discussion around eventual improvements that could have been implemented to the program, comparing the problem statement with the results and presentation of the possible alternative solutions.

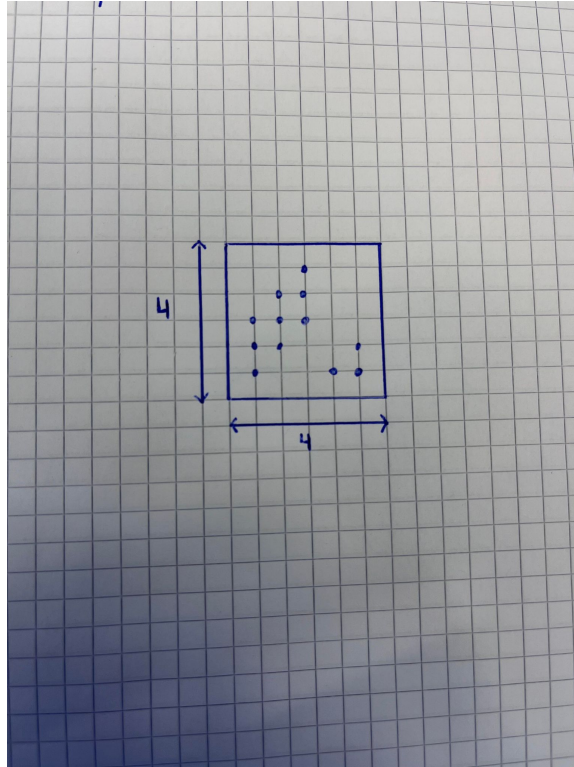
4.1 The depth values

A problem that occurred that was left unaddressed was that there could be multiple depth values added to the same pixel. This would happen more frequently at lower zoom levels, as each pixel would represent a larger area. The result of this was that every time a depth value was added to the same pixel, the program would overwrite the previous value, which makes the representation of some pixels less accurate.

We came up with two theories on how to solve this problem. The first theory was to calculate the average value of all depth values within one pixel. This would represent the depth value in a more accurate manner than the method used in our program. The second theory was to use an 80 percentile that would result in a more accurate representation than our previous theory. This is because some values could be more spread out, and by using 80 percentiles we would acquire a point that was in the more concentrated area, instead of the average value that could be a less good representation if some of the values deviated a lot from the majority of the

4.2 The tiled images

Figure 4.1: A pixel represented as 4x4m with multiple values



values.

4.2 The tiled images

The goal for generating the tiled images was to visualize the depths as grayscale. OpenCV has a color-range ranging from 0 to 255. If a pixel has the value 0, it will be presented in the image as black, on the other hand, if a pixel has the value 255, it will be presented as white and values in between 0 and 255 will be grayscale.

The depth values from the data files were originally presented in meters. If we were to use the depth values in this way, the image would become almost completely black, because the water levels in Hafrsfjorden are low.

4.3 Missing values

Figure 4.2: Tiled image



By converting the depth values into centimeters, the images became more detailed with better gray-scale, but this also lead to a problem. As previously mentioned, the color values range from 0 to 255, and because of this, depth values greater than 255 centimeters or 2.55 meters all became white. This is usually not a problem when viewing the map close to land, where the depths decline, but further away from land, where depths become greater than 2.55 meters, details fade away and pixels become all white.

4.3 Missing values

All black areas in the images represent either land or that no values are present in the matrix.

There can occur black dots within the image, that represents a dropped value, or a value that has not been recorded for some reason. A solution to

4.4 Building the image

this, which we have not had time to address, is to use the pixel value of its adjacent point.

4.4 Building the image

To build all the tiled images for all the 9 different levels took around 112 hours, and is a non practical way to generate the images. A solution for not needing to run the program 9 different times (one for each level), which we did not have time to implement, is to run the program at level 9, where the tile dimensions are at the lowest ($1/32 \times 1/32$)m and build the other images from these images. This could have been done by combining 4 level 9 tiles which correlates to 1 level 8 tile, and by repeating this process, all the tiled images for all levels could be generated, and this would take a fraction of the time compared to re-reading through all the files for each level.

4.5 Depth visualization and time complexity

As given in the problem statement 1.1, the main goal for the visualization was to create an application that can show Hafrsfjord processed data in a user-friendly way, alongside with visualization of the whole or partial Hafrsfjord dry out.

One of the main decisions made was to make *Hafrsfjord* class for the raw data and visualization of the processed data, *GUI*, independent. It is quite time-consuming to process raw data, in order to reduce the running time for the application, all methods that may require tiles dependent on the level were pre saved in the folder *Images* for a quicker access. On the other hand, this technique requires some extra storage memory. The final application is created the way that it wouldn't require much effort to connect both parts for further collaboration.

All the processed data is in the format of a big amount of tiles of a given size that creates a lot of lags while first display. Which in the first place was not that efficient due to the *QGraphicsScene* being emptied before getting images for each new zoom in/out levels. With each zoom in level the amount

4.5 Depth visualization and time complexity

of tiles increases which causes even more delays when all of them are being loaded and displayed. Therefore, all tiles are loaded in the background and only tiles inside the visible area are being loaded for display. This method also shows good timing while colormap and depth level modifications. When *Colormap and scaling* dialogue are activated, quick changes extend only to the visible area, which fastens the process drastically. As well as, when the user decides to save the changes, then and only then all the tiles will be updated, in the background. This method prevents any possible delays to appear.

An alternative method for displaying images on zooming in and out, could be creating different scene layers, which may reduce the time it takes to switch between zooming levels. As explained in the GUI chapter 3.5, while zooming in and out all the previous items stored in the scene are removed and then new items in form of *QGraphicsPixmapItem* been added. In case of the current application, this display technique doesn't affect the application in a drastic way, but may delay in the future.

One of the main features in GUI is *Colormap and scaling* dialogue which is used to simulate depth level to show drying out, change colormaps and update scaling factor. To solve the issue with drying out, threshold image processing (segmenting images) method used. With help of threshold t (thresh) value, each pixel in an image depending on its intensity will be replaced. If pixel value is less than fixed thresh value, then by calling *cv2.threshold()*, the pixel will be given a value of 0, else if pixel intensity is greater than the threshold, the pixel will get white. In case of depth visualization and drying, this results in the darker colors (assigned to coastline) being darker, and the gray-white pixels assigned to water becoming whiter, allowing to manipulate pixels both way making darker pixels to show more on an image, simulating drying. Or if going the other way, showing higher depth level. To make changes even more visible, the user can choose one of the provided colormaps. When it comes to application of colormaps, CV2 has a good range of 12 different colormaps, but due to CV2 segment being BGR format, a little conversion *cv2.cvtColor(heat, cv2.COLOR_BGR2RGB)* is required.

In the early stage of application, instead of threshold and colormaps were seaborn library used. Seaborn is an analogue of Matplotlib, which spec-

4.6 Placement of multiple pixmaps and extra objects

ifies mainly in graph visualization. This method was not that efficient in subsequent perspectives. First of all, the definition of dots per inch, known as dpi, is required in order to resize an image. The main issue with using this method was lags and delays when larger images were added, also all images over 3000x3000px. This method was optimized by plotting on the same figure, but once again such modifications on larger images have little to say, we can conclude that plotting was inefficient in this case. As another alternative for one image updates, *PyQtGraph* could have been used, as it's more comparable with PyQt and is more frequently used for real-time graph modifications.

4.6 Placement of multiple pixmaps and extra objects

In the previous section, visualization and display of multiple tiles taken to look at. This part discusses methods used to store and further placement of tiles on the scene before displaying as placement of extra items such as grid, pointer and map signs. In order to processed data, it should be stored somewhere in a form that when the visible area is moved, the program would know what tiles to be shown next. For that purpose, all tiles stored as pixmaps in the dictionary, converted to *QGraphicsPixmapItems* and whenever *QGraphicsPixmapItem* tiles are in the visible area, add those to the scene.

At the same time, there are several alternative methods on how to place multiple pixmaps for further visualization. One of them is to use *QGraphicsGridLayout*, by first adding pixmap to the scene *self.scene.addPixmap(pixmap)* and creating *QWidget*. The only problem would be *QGraphicsLayout* that takes only *QGraphicsLayoutItem*. But again could be solved by creating a container class for the pixmap item of type *QGraphicsLayoutItem*. And then implement the required function. This could be seen as another way of placing tiles similar to the one used.

Secondly, one view can have only one scene. Then it is possible to add pixmap to the scene, creating an item of type *QGraphicsPixmap* and adding it to *QGraphicsScene*, assign scene to the view, and by adding view to the layout create a layout of views. In the application, for each image, a pixmap

4.6 Placement of multiple pixmaps and extra objects

is created and then added to the scene. Before any pixmap is added, all items previously assigned to the scenes are removed. This alternative is not that efficient as it requires several views to be displayed and manipulated, which causes a lot of dictionary manipulations and movements.

Therefore, the best alternative or potential upgrade is to use the layered method discussed in section 4.5.

Another issue to mention is conversion of the tiled images back to the depth array, this is not *npImage* array used in *pixmpa2qimage2np*. *GUI* class contains conversion between pixmap, QImage and *npImage* array but due to images being stored as .jpg and .png, it is not possible to convert those to back to depth array. The reason is that .jpg type cannot carry more than 8 bits per channel and 8 to 16-bit integer for .png.

Beside main modifications, the application allows adding grid (array of lines that create a grid), point and extra map signs to the scene. For application of mentioned features, an extra item in the form of a grid, point marker or stack of extra signs been added to the *QGraphicsScene*. Under will be provided an old version for comparison.

Previously all items have been draw on a pixmap by *QPainter* which was not a good practice in terms of access and update of the elements. Grid updated *self.npImage* and could be backed up with *self.prevPixmap* while pointer didn't change an image array. Both grid and pointer could be called together. For more precise drawing *painter.translate(.5, .5)* and *painter.setRenderHints(painter.Antialiasing)* been added.

Updated version all changes happens independently and doesn't require any drawing to be called on the pixmaps. Meaning that any new items added to the scene can be easily removed or edited without pixmaps being involved, which make application more efficient and editions more productive. A new feature for grabbing and dragging the point marker have been added, allowing quick and simple location check. Display of an image created by ".asc" file allows displaying the coordinates alongside the grid. This feature was removed for the tiles, for a better overview of the depth tiles, also can be easily restored by just adding *addText()* method inside *drawGrid()* function.

4.7 Economy and environmental accounting

The last thing to mention, is extra signs that can be added to the map. The application allows, by enabling **Add extra signs** from the toolbar, to add map signs to the tiles. This is the feature that can be modified and updated by threshold image processing as well as it is done for the depth values, if altitude values would be added.

4.7 Economy and environmental accounting

Due to the low energy consumption and the inappropriateness of specifying the material used, the decision was made not to take into consideration both economy and environmental accounting, as those are irrelevant.

Chapter 5

Conclusion

As part of the thesis work mentioned in 1.1, we have implemented an application that read, interpret and arrange large and redundant data so that one for each point (with resolution down to centimeter level both in position and depth) can display depths for a selected area appropriately. It was achieved by visualizing large images with depth data for each pixel.

With the development of zoom levels ranging from 1 to 9 and creating a method that could generate tile dimensions by taking the zoom levels as a parameter, we achieved not only visualization down to 1×1 meters but a more varied range, up to 16×16 meters and down to 6.25×6.25 centimeters. This made zooming more smooth and flexible and makes the program more user-friendly. As for the memory consumption, the compression of the .asc files into .z01 files made a huge impact. The data files after the compression had been reduced to 5 percent of what they were while uncompressed.

Achieved images with depth data visualized in the GUI which contains a simulator that shows depth level change in both ways. User can visualize that Hafrsfjord is emptied of water, change color of the map for a better similarity. The application is expanded with finds, special objects / formations (rocks and boulders) for a more complex view. All changes appear effectively without delays due to optimization of the image load in the background.

Bibliography

- [1] analyticsvidhya.com. How images are stored in the computer? <https://www.analyticsvidhya.com/blog/2021/03/grayscale-and-rgb-format-for-storing-images/>. Last accessed: May 12, 2022.
- [2] docs.python.org/. Built-in functions, open(). <https://docs.python.org/3/library/functions.html#open>. Last accessed: May 12, 2022.
- [3] docs.python.org. collections - container datatypes, deque objects. <https://docs.python.org/3/library/collections.html#collections.deque>. Last accessed: May 12, 2022.
- [4] docs.python.org. General python faq, what is python? <https://docs.python.org/3/faq/general.html#what-is-python>. Last accessed: May 12, 2022.
- [5] fjordnorway.com. Hafrsfjord. <https://www.fjordnorway.com/no/se-og-gjore/hafrsfjord>. Last accessed: May 12, 2022.
- [6] GISGeography. How universal transverse mercator (utm) works. <https://gisgeography.com/utm-universal-transverse-mercator-projection/>, October 29 2021. Last accessed: May 01, 2022.
- [7] Antin Harasymiv. Prototyping a smoother map. <https://medium.com/google-design/google-maps-cb0326d165f5>, August 27 2018. Last accessed: May 10, 2022.
- [8] <https://numpy.org>. What is numpy? <https://numpy.org/doc/stable/user/whatisnumpy.html>. Last accessed: May 12, 2022.

BIBLIOGRAPHY

- [9] Claudius Ptolemy. *The Geography of Ptolemy*. around AD 150.
- [10] Wikipedia. Depth map. https://en.wikipedia.org/wiki/Depth_map.

Attachments A

Software

This chapter contains a directory tree with description, required installations and a downloadable zipped file with application.

A.1 Directory tree

After detailed close up at each folder, a directory tree of the full application will be provided.

App folder contains files created for testing GUI for one image. Allowing to create an image from .asc file received from MainWindow class in GUI.

ELE610pyfiles provides a variety of AppImageViewer programs, dialogs and startups which provides simple image processing applications and were used as a template for visualization of GUI.

GUI consists of **Images** folder which contains icons, tiles for 5 zoom levels, application logo and several images used for testing of splitting and depth color change; **QImagesMethods** methods for conversions of qimages and numpy arrays; **app** which contains a full visualization program including two main classes MainWindow and QGraphicsView; **clsColorDialog** is a dialog called by **app** method whenever user decides to simulate depth level,

A.1 Directory tree

change colormap or scaling factor; **stack** provides a class of type stack (First in—Last out) used to undo changes to the scene whenever extra items are added; **stylesheet.qss** is a Qt Style Sheet analog of Cascading Style Sheets (css) stylesheet used to customize the appearance of an app as the whole or by changing the look of each element on its own.

HafrsfjordFiles is part of the raw data program which provides a module **clsHafrsDepthPoint.py** for presenting UTM coordinates in various ways. **clsHafrsFiles.py** contain a class for opening, reading and closing files and the class **HafrsTiles** that is responsible for manipulating the UTM coordinates so that they can be converted into tiled images. **clsMyBAzip1.py**, **funMyCompress.py** are modules for compressing and decompressing the .asc and .z01. **1_N.z01** contains all the .z01 files which stores the depth points.

README.md contains “about” section of the application.

A.1 Directory tree



Figure A.1: Full directory tree

A.2 Download links & Requirements

A.2 Download links & Requirements

Downloadable zipped file with application ([Click to download](#))

Due to the rights reserved and contract assigned where our group is restricted with any sharing of Hafrsfjord files, GitHub link is not provided as it requires publicity of the repository.

A.2.1 Installation

For proper use, the application requires several libraries and packages to be installed beforehand. For the project, **Python 3.x** has been used, where the needed packages usually work well on all platforms. The main packages mentioned in this document are: numpy, OpenCV, PyQt5.

Python can be installed by accessing it from the official [Python download page](#). An administrator access to the computer may be required. The program also requires an Integrated Developer Environment (IDE) such as **Anaconda** or **PyCharm**. Packages should then be installed by PIP Installs Packages, also known as pip. Code used for installation is listed under:

```
1 :\> pip install numpy
2 :\> pip install opencv-python
3 :\> pip install qimage2ndarray
4 :\> pip install pyqt5 or conda install pyqt (for anaconda)
```

The application doesn't have any specific minimum requirements, but the raw data part of the program is CPU demanding.

Attachments B

User manual

B.1 Application startup

The program starts with running GUI's **app.py** file. As shown in the figure A.1 in Attachments A.1, the main **app.py** file can be found in **bachelor_2022 ⇒ GUI ⇒ app.py**.

As show in the figure B.1, there are two visualization possibilities, either to show pre saved Hafrsfjorden or import an .asc file and display the outcome as an image.

Regardless the choice, **Show Hafrsfjord** or **Open .asc file**, all special features are available and are provided in a list under:

File menu:

- **Show Hafrsfjord** displays tiles of different parts of Hafrsfjord as a set.
- **Open file**, allows opening .asc file and displaying image generated based on coordinates and depth values.
- **Save file**, saves only visible area of an image (for Hafrsfjord), else saves a whole image (for .asc files).

B.1 Application startup

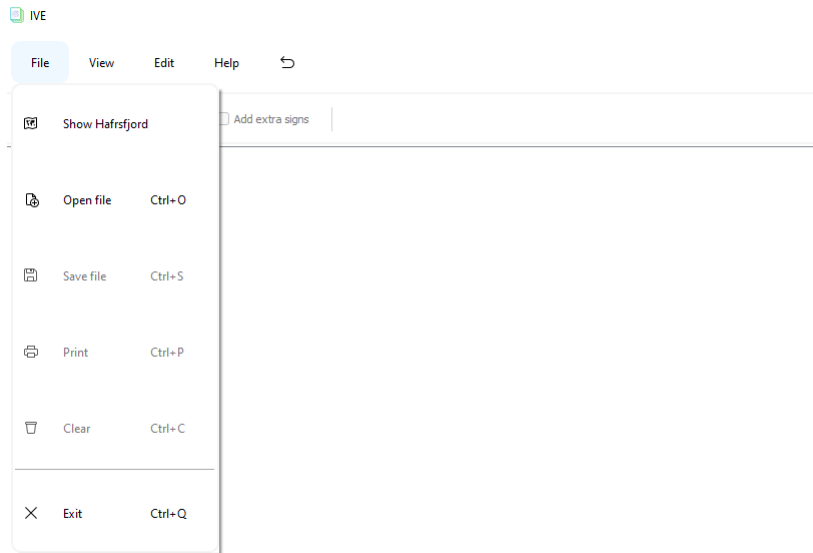


Figure B.1: Application starting page

- **Print** prints only the visible area of an image (for Hafrsfjord), else prints the whole image (for .asc files).
- **Clear** clears the scene (for Hafrsfjord) or pixmap (for .asc).
- **Exit** exits the application.

View menu:

- **Zoom in/Zoom out** zooms in/out by updating tiles to the given level, only for Hafrsfjord.
- **Default size**, allows resetting all window/view size, scaling/-zooming, etc. changes back to the default ones.
- **Toolbar**, enables/disables visualization of the toolbar.
- **Grid**, enables/disables the grid.

Edit menu:

- **Crop** allows cropping of the given rectangle with further display of the outcome.
- **Undo**, undoes last change.
- **Colormap and scaling**, displays colormap and scaling dialogue allowing to:

B.2 IVE Keyboard Shortcuts

Change colormap.

Depth level simulation with colored slider.

Change the scaling factor with testing of scaling in/out and reset functions.

Help menu:

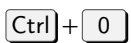
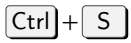
- **Help** provides overall description of the application.
- **About** displays creator information and version.

Toolbar provides a short path to the most used features such as saving, cropping, editing an image and setting a marker together with extra signs to the scene.

B.2 IVE Keyboard Shortcuts

Overview over shortcuts used in an application. All keyboard shortcuts are available on non-NOB keyboards and on Mac. If a shortcut isn't available on your keyboard, you might trigger it from the menu itself.

Some features are also available through the toolbar.

Windows	Description
	Open file.
	Save file as.

(continued on the next page)

B.2 IVE Keyboard Shortcuts

(from previous page)

Windows	Description
Ctrl + P	Prints image with location and app name.
Ctrl + C	Removes image and clears pixmap.
Ctrl + Q	Quite the program.
Ctrl + Z	Undo the latest changes.
Ctrl + R	Crop an image by indicating rectangle.
Ctrl + D	Color bar and scaling dialogue.
Ctrl + +	Zoom in.
Ctrl + -	Zoom out.
Ctrl + mousewheelUp	Scale in.

(continued on the next page)

B.2 IVE Keyboard Shortcuts

(from previous page)

Windows	Description
<code>Ctrl</code> + <code>mousewheelDown</code>	Scale out.
<code>Ctrl</code> + <code>1</code>	Returns to the default size.
