TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Yaroslav Hrushchak 172685IVSM

Visual Localization for Iseauto Using Structure from Motion

MSc thesis

Supervisor
Juhan-Peep Ernits, PhD

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Yaroslav Hrushchak 172685IVSM

Visuaalne lokaliseerimine Iseautole kasutades ruumilise info tuletamist kaamera liikumisest

Magistritöö

Juhendaja
Juhan-Peep Ernits, PhD

Tallinn 2019

Declaration

I declare that this thesis is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Yaroslav Hrushchak

May 14, 2019

....
(Signature)

Abstract

This thesis describes an approach of solving a localization problem for Iseauto based on visual data only utilizing an open-source structure-from-motion software (OpenSfM). OpenSfM is a structure-from-motion library written in Python originally intended for reconstructions of 3D objects based on feature matching and tracking among a set of images. This work describes a workflow of performing localization, gives an overview of main difficulties and provides a performance evaluation.

The main unit of interest is the Iseauto project of TalTech. For the data collection, I am utilizing fish-eye and rectilinear cameras which are installed on the vehicle as well as the RTK GNSS device for retrieving geolocation data. The approach described in this work provides a relatively accurate results with an error of 0.5-1.5 meters with a single 3.2 megapixel camera. The precise error estimation is possible due to the high precision of the installed GNSS device (precision up to 2cm). As the software for accessing the data from the GNSS device was in development during the experiments in the current thesis, not all collected data contains high precision coordinates. The computation performance in its current form does not allow to localize the vehicle in real-time, but can be improved in future work. The approach has value as it can be used as an auxiliary localization system or as the only localization system for cost reduction purposes.

The thesis is in English and contains 61 pages of text, 6 chapters, 22 figures, 3 tables.

Annotatsioon

Käesolev magistriöö uurib kaamera liikumisest tuletatud ruumilise kujutise saamise (SfM) meetodi rakendamist isesõitva auto lokaliseerimiseks kasutades vabavaralist teeki OpenSfM. OpenSfM on ruumilise kujutise kaamera liikumisest tuletamise teek, mis on kirjutatud programmeerimiskeeles python ja mis on algselt mõeldud 3D kujutiste tuletamiseks hulgast kujutistest. Käesolev töö kirjeldab töövoogu, kuidas kaamera asukohta kaadri järgi lokaliseerida ning annab ülevaate põhilistest raskustest tulemuse saavutamisel ning kirjeldab jõudlustulemusi.

Käesoleva töö kontekstiks on Tallinna Tehnikaülikooli Iseauto projekt. Andmete kogumiseks kasutasime autole paigaldatud kaameraid lainurkobjektiiviga ning väga väikeste moonutustega objektiiviga. Lisaks kasutasime RTK GNSS seadet, mis võimaldas suure täpsusega salvestada auto asukohta.

Töös kirjeldatud lähenemine annab üsna täpse tulemuse: katsetustes on kaamera asukoha arvutamise viga vahemikus 0,5 - 1,5 m, kasutades üht 3,2 megapikslise sensoriga kaamerat. Üsna täpse veahinnangu saamine oli võimalik, kuna Iseautole on paigaldatud väga suure täpsusega GNSS seade (täpsusega kuni 2 cm). Kuna GNSS seadmega suhtlemise tarkvara oli käesoleva töö kirjutamise ajal samuti arenduses, ei ole kõigis kogutud andmekomplektides täpseid GNSS andmeid.

Loodud tarkvaralahendus ei sobi veel jõudluselt reaalajas lokaliseerimiseks, kuna koordinaadivihjega (nt ebatäpselt GNSS sealdmelt) kaadri lokaliseerimine võtab u. 30 sekundit ja vihjeta mõned minutid sõltuvalt rekonstruktsiooni suurusest, kuid jõudlust saab järgnevates töödes parandada näiteks arvutusi paralleliseerides ning sobivat kaadrite indekseerimissüsteemi kasutades.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 61 leheküljel, 6 peatükki, 22 joonist, 3 tabelit.

Contents

1. Introduction

Self-driving cars are becoming more and more popular (both for research and commercial purposes) as data processing units become smaller, more powerful and financially accessible. The development of sensors and the development of data processing tools and algorithms makes it possible to construct more complex systems.

Self-driving vehicles have some important advantages over the traditional way of driving a car. First of all, a machine using different sensors might perceive more surrounding information than most humans (especially at night time). Secondly, unlike a human, a machine cannot be distracted, cannot become tired or fall asleep (only discharge, but that does not influence perception of the traffic situation) and also cannot be influenced by diseases which cause loss of consciousness or muscle strength (stroke, heart attack, seizure etc.). Any of these factors might lead to serious injuries and even deaths. Thirdly, even if a driver doesn't have any health problems and is able to drive safely, having a machine drive autonomously will ease the life of a passenger. Finally, eliminating the need for car controls and a driver, it is possible to make the vehicle more spacious and carry more people.

In a static environment with no possible obstacles such vehicle could be programmed to move from a departing point to a destination point using special floor marks, but in a dynamic environment (mainly outdoors) the surroundings tend to change unpredictably. Thus, a self-driving machine has to assess the surrounding situation using sensors and make a decision how to move further safely with less time consumption.

1.1 Problem statement and contribution of the thesis

One of the important abilities of a self-driving vehicle is to determine its position in space in order to plan a path to the destination point. Although localization for a self-driving vehicle could be implemented using a LiDaR system or a GNSS system, which provide high accuracy, these devices are relatively expensive. and if it is possible to develop a cheaper alternative with a competitive accuracy, it would allow to reduce expenses on a single car unit and allow smaller researchers or developer teams to contribute to this field. In addition to that, it would be preferably to have a fallback in case if main localization solutions will end up with a failure (from a safety perspective).

Current approach for localization is using LiDaR system and Point Cloud Library to map collected data onto a 3D point cloud map as described in [1]. The sensor measures distances to objects in all directions by illuminating objects with pulsed lasers and estimating the respond pulse time. These collected voxels afterwards are used to define the machine position in space through pattern matching of a local point cloud onto a complex point cloud (which should be preliminarily composed for an area of interest where the vehicle is going to operate).

The main research goal would be to determine how well cameras can be utilized within the Iseauto localization problem.

The expected outcome of this thesis is:

- An analysis of how to get a better reconstruction;
- Description of a visual based localization workflow;
- Analysis of the offered approach;

Expected approach to test if the goal of the developing system is being achieved is to compare the accuracy with the GNSS system whose average error is up to 2cm and would be considered as successful if the error would be not greater than 20cm.

1.2 Autonomous vehicles

In the recent years the development of autonomous vehicles is becoming more and more popular object of development. Different organizations and companies get involved in engineering and assembling of their own autonomous cars, i.e. vehicles which use various sensors for gathering data, a processing unit to process this data for making decisions on further actions and actuators to actually move the whole unit. As summarized in [2],

> There are a number of industrial companies worldwide working on self-driving car projects, such as Volkswagen, BMW, Hyundai Motors, Audi, Ford, General Motors, Honda, Mercedes, Nissan, Nvidia, Tesla, Toyota, etc. It is interesting to note that Silicon Valley companies, including Google, Apple, and Uber, are seeking to become key players in this market even though their history is not directly tied to the production of cars.

There are various reasons which influenced the gain in the number of developers. First of all, the rapid technical progress led to the manufacturing of electronics, whose computation power and data gathering precision is increasing while the size is becoming smaller. Also, as the manufacturing technology improves and the competition between manufactures increases, these electronics are becoming cheaper, which means that engineering projects are becoming more affordable not only for private organizations but also for university researchers.

Secondly, different developer teams and individual developers upload free open-source software which can be reused for creating new functional. For example, in terms of self-driving vehicles there is an open-source software named Autoware [3] which has a rich number of modules for data gathering, computing and mechanisms actuation capabilities. This software might be used as a base for a self-driving vehicle allowing new developers to focus on other aspects, such as safety, usability, objects recognition, optimization of travel path etc.

Thirdly, such vehicles are welcomed by the society as they either eliminate a need of having a driver on public transport, have a higher environment perception capabilities which might provide a better decision making outcome, for individual drivers it might allow to save energy after a hard day or during a long ride or at least cannot be affected by diseases which cause loss of consciousness or muscle strength (stroke, heart attack, seizure etc.).

1.3 TTÜ Iseauto project

The project of designing and construction of a self-driving vehicle at TTU started in June 2017 and already has made a significant progress with a demo drive being conducted on 20th of September. The main software which is used to control the vehicle is an open-source Autoware which is run using Robot Operating System on Linux (Ubuntu). As stated in [4], “next sensors are used in the ISEAUTO project to observe the environment and mapping, localization and navigation:

- LiDaR Velodyne VLP-16x2;
- Ultrasonic sensors at the front and back x8;
- Ultrasonic sensors at the door side x6;
- A short-distance radar;
- Cameras x8;
- An RTK-GNSS;
- An IMU sensor."

The RTK-GNSS is a real-time satellite based navigation system which is very precise (error up to 1 cm), although is expensive. The IMU sensor is an inertial measurement unit used to measure a body's angular rate (pitch, roll, yaw).

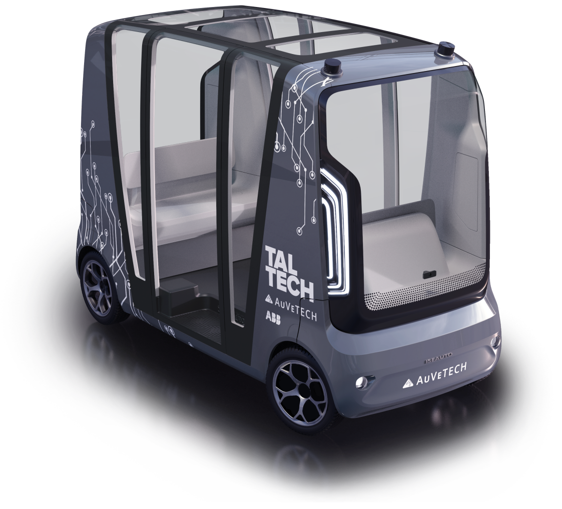


Figure 1.1: Design of ISEAUTO last-mile bus [4]

1.4 Outline of the thesis

Chapter 2: Related work

This chapter overviews the existing approaches to a visual based localization problem, like an approach using network flows.

Chapter 3: Structure from Motion and available implementations

This chapter introduces the principle of the structure from motion technique, covers the existing implementation and describes the reconstruction stages of the OpenSfM software.

Chapter 4: Factors which influence the accuracy

Considering the localization problems based on camera vision it is important to take into account that there are various types of lenses and factors like the weather, lighting conditions, environment might continuously change. This chapter will cover these factors as each of them might influence the final reconstruction.

Chapter 5: Geographic coordinate system and georeferencing

Usually a reconstruction is bound to a local coordinate system where coordinates represent only relative positions between objects (voxels and cameras). To be able to locate cameras and objects, it has to be bound to a geographical coordinate system. This section will cover the main geographic coordinate systems and also compare available georeferencing methods available in the OpenSfM.

Chapter 6: Results

This chapter will provide a test scenario and provide an analysis of this approach.

Chapter 7: Conclusion

The final chapter will conclude the thesis, its possible application and provide suggestions for further improvements.

2. Related work: Existing approaches to the visual localization problem

This chapter gives an overview of the existing approaches to solving the localization problem based on visual data.

2.1 Robust Visual Robot Localization Across Seasons Using Network Flows

The work written by Tayyab Naseer, Luciano Spinello, Wolfram Burgard and Cyrill Stachniss [5] proposes a solution of visual based localization across seasons which is supposed to work at low frame rate and doesn't require season-based feature learning. To achieve the localization, they compute network flows in an association graph and compute minimum cost flow. For this they use an image description which is computed using a dense grid of Histogram of Gradients (HOG) descriptors [5]:

> Instead of relying on keypoints, we compute for each image a fixed quantity of descriptors. To achieve this, we tessellate the image into a grid of regular cells, see Fig. 3. For each cell, we compute a HOG descriptor and combine all descriptors into a single, dense description of the whole image. The overall image descriptor h is a vector composed of the concatenation of all the histograms of gradients computed on all the cells.

There are various factors described in the paper which make the process non-trivial, which include the difference in weather, illumination, change of the scene constructions and also the difference in vehicle positions and speed of movement.

The data association graph is shown on Figure 2.1

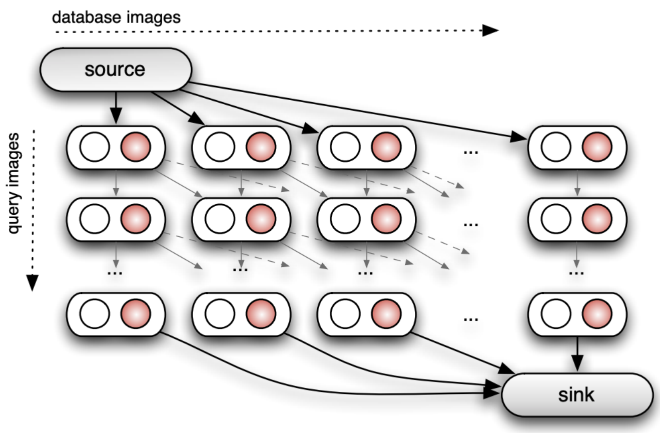


Figure 2.1: Flow network graph [5]

Because the image is tessellated into small cells and each cell is used independently to define a match, it is more likely that seasonal changes wouldn't affect the resulting accuracy much, as even if some cells might differ across seasons (covered by leaves or revealed), other cells would look equal and would allow to recognize the place.

Although this method "accurately matches image sequences across seasons and it outperforms two state-of-the-art methods such as FABMAP2 and SeqSLAM" [5], it has some disadvantages: firstly, it requires to store a large amount of images (thousands) along the movement path which further would be processed to localization; secondly, the vehicle has to move along a predefined route in order to recognize its location on this route and the accuracy might dramatically fall if the vehicle would deviate from the route as moving along a deviated route would provide images which would not match with the ones stored in the database: "The car drove approximately along the same route but with different speeds and due to traffic, it had to stop a few times at different locations while acquiring the query images" [5].

2.2 Visual SLAM

A paper written by José Ruiz-Ascencio [6] considers a localization approach, similar to the approach used and evaluated in this work, based on images as the only data source, but it considers an approach when a map creation and localization in space is being done simultaneously. In addition, the paper only briefly describes the idea without going into any details or analysis.The authors divided the visual SLAM problem into three groups: probabilistic filters (maintenance of a probabilistic representation of a robot pose and locations of environment landmarks), a technique employing a Structure from Motion and bio-inspired models.

The standard procedure in this technique is to extract environment features, match them and perform Bundle Adjustment (BA) to minimize the re-projection error, [6] :

> SfM allows a high precision in the location of the cameras but does not necessarily intend to create consistent maps. Despite this, several proposals have been made using SFM to locate with precision while creating a good representation of the environment.

The technique might match key features across images and estimate relative positions of the cameras buto depending on the image quality, the reconstructed structure or point cloud might not be good enough for extraction of 3D objects and might not even be recognized by a human. There are several factors which might cause to these inaccurate reconstructions: blurred image due to the movement of a camera or an out-of-focus focusing (small vibrations an image might blur and it might have a significant influence for thin structure element or for elements in the distance), dynamic objects in the environment which might constantly move and therefore lead to false matches across images; repetition of structure in different locations (traffic lights, brick/stone walls, buildings etc.).

2.3 Localization for an Autonomous Vehicle with a Multi-Camera System

A recently published paper by Lionel Heng “Project AutoVision“ [7] (March 2019) proposes the use of a 360 degree camera for visual based localization utilizing multi-view geometry and deep learning. In order to perform localization, the authors have installed 12 Near-Infrared cameras with fisheye lenses (Figure 2.2). The cameras were calibrated using ApriTag markers [8] which allows to calibrate multiple cameras at the same time due to uniqueness of markers.

For estimating camera geolocation, authors propose two approaches. The first one is used for localization at an unmapped area. For that, they have two thread running: tracking and mapping. Initially there should be a starting geolocation (received for example by a GNSS device) and then using the last two frames and a depth information from a stereo-cameras a new position is computed. An average computed geoposition error with this

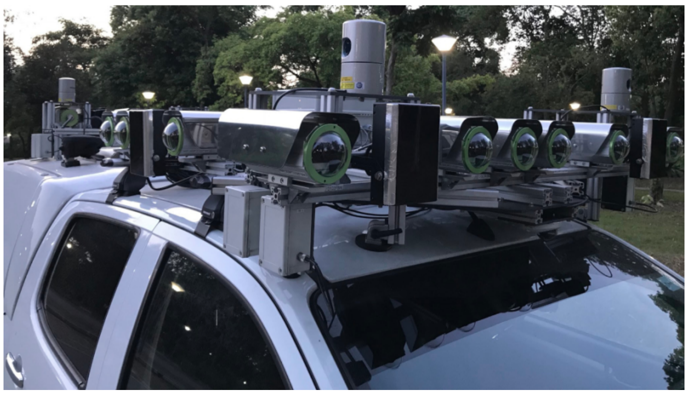


Figure 2.2: Auto Vision vehicle platform [7]

direct visual-inertial odometry approach was about 9.5 meters but with high frequency of 30Hz. The second one is used to localize at a mapped area using a Sparse 3D map reconstruction. Having the pre-made map and an initial geoposition, they perform feature matching between the closest frame to estimate a location. This method has higher accuracy (with error at about 3m), but lower frequency (2Hz).

In general, both approaches seem to have value for localization and the performance is high for localizing a vehicle, the accuracy is not precise enough to be used for navigation purposes or for localization in an urban environment as a 3-9 meters error is significant. Another drawback is that the project uses simultaneously 12 cameras which is a serious load for a CPU and a network.

In order to have a more flexible localization approach, I utilized the SfM technique which is originally used for recreation of environment structure. As this technique involves estimation of a relative camera positions, it is supposed to be able not only localize along the covered route but also in the nearby area. As there are already existing software products for performing reconstruction using SfM, I am going to overview them and using the most suitable software, I will research how to utilize it for localization, evaluate problems, accuracy, efficiency and possible application. The work is oriented on the Iseauto project and therefore I will have to utilize the software and hardware already installed on the vehicle.

3. Structure from Motion and available implementations

A low-cost photogrammetric technique for structure reconstruction is called Structure-from-Motion (SfM) [9] and was first developed in 1990s [10]. The software implementations are to this date under active development and different developers provide various solutions with its own interface, functionality, performance, accuracy. Not all of the implementations are free, but they are all based on the basic principles of tracking common features across images. The SfM approach is mostly being used for geomorphological research and usually is based on data collected by drones. On the other hand, as has been shown in [11], SfM can be customised for visual localization.

3.1 Structure from motion

Structure from motion is a special technique in computer vision for reconstruction three-dimensional models from a sequence of two-dimensional images by analyzing the motion between these images. This method allows computers to estimate distances to objects using camera images in a similar way as humans when they move around some object, as show on Figure 3.1.

The main data source in this method is a set of features. A feature is a piece of information and on images it is represented by points and edges. Usually, a feature is detected in a place where the intensity of an image significantly changes and can be represented in a boolean value (presence or absence) or in a float value (by a corresponding level of certainty). These detected features are then tracked within a set of images and used to estimate interior and exterior parameters of cameras as well as the 3D position of the features itself. The main requirement is that a single feature has to be visible on at least three images but the additional images would provide better accuracy. As for the camera,

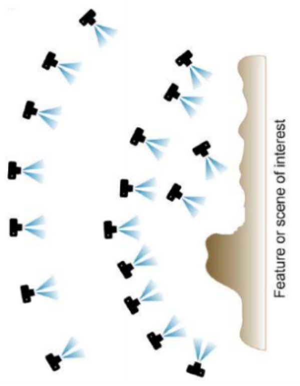


Figure 3.1: Camera positions for an SfM dataset [12]

the best result should be achieved using a digital SLR camera with a fixed focus lens and capturing a static scene.

3.2 Available solutions

The area of image based localization has being studied for several decades and there are various papers in this direction [6], [13], [14]. There are various tools available for PC, smartphones and available as a service, which provide similar reconstruction output, but have different functionality and performance. The web-based solutions include Autodesk 123D (which was replaced with Autodesk Recap [15]) and until 2017 Microsoft Photosynth. For local install there are Agisoft Photoscan, PhotoModeler, VisualSFM (`http://ccwu.me/vsfm/`), SFMToolkit (`http://www.visual-experiments.com/demos/sfmtoolkit/`), Agisoft Metashape [16] and OpenSfM (`https://github.com/mapillary/OpenSfM`) and probably more. Among the available solutions, the OpenSfM was selected as it operates on a Linux system, has a permissive open source license which allows to modify the code to extend functionality. For a comparison purpose on how good this open-source software reconstruct objects, VisualSfM solution was tested, running both utilities on the same dataset. Regarding the feature detection algorithms, VisualSfM uses HAHOG, but have a possibility to use some other algorithms as would be described later.

For performing experiments, two types of images were collected: images extracted from the Iseauto bus fisheye cameras and from a DSLR camera for comparison. The images from the DSLR camera were slightly adjusted in a photo editing software, as this already takes more time and effort to collect images comparing to the recordings from a vehicle which is more automated, so it seems reasonable to try to make the result more accurate. The adjustments included removing highlights, amplifying shadows and increasing sharpness and photo editors allow to apply defined settings to a bunch of photos at once. For the Iseauto camera, the images also need adjustments because with the fisheye lens they have a barrel distortion. Tackling the distortion will be covered in the next chapter.

The results of running a reconstruction if presented on the Figures 3.2-3.5.

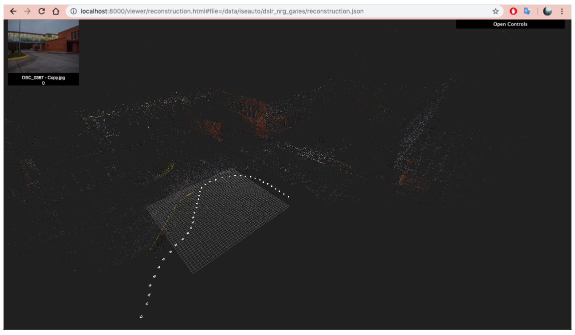

localhost:8000/viewer/reconstruction.html#file=/data/iseauto/dslr_nrg_gates/reconstruction.json
Open Controls
DSC_0387 - Copy.jpg
0

Figure 3.2: DSLR camera with OpenSfM

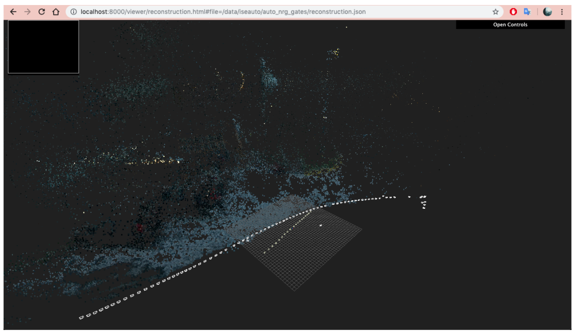

localhost:8000/viewer/reconstruction.html#file=/data/iseauto/auto_nrg_gates/reconstruction.json
Open Controls

Figure 3.3: Fisheye camera with OpenSfM

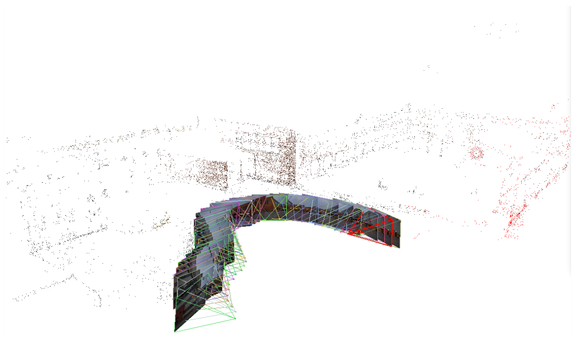


Figure 3.4: DSLR camera with VisualSfM

Although, the results might look similar, on a closer look it is noticeable that OpenSfM provides more 3D points and more precise result. The computation time is presented in Table 3.1.

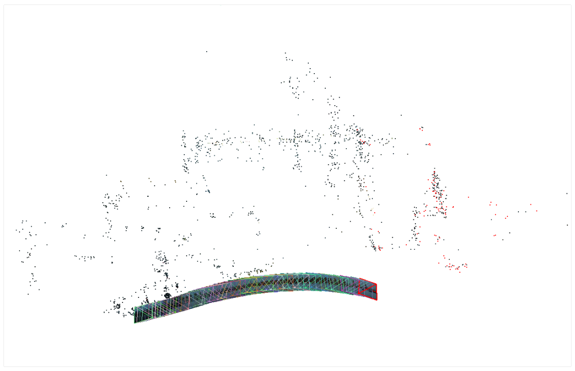


Figure 3.5: Fisheye with VisualSfM

Camera Type	Lens model	Software	Reconstruction time
DSLR	Nikkor 18-55mm	OpenSfM	6m 40s
DSLR	Nikkor 18-55mm	VisualSfM	9m 45s
Fisheye	Sunex DSL315 2.7mm	OpenSfM	5m 25s
Fisheye	Sunex DSL315 2.7mm	VisualSfM	10m 5s

Table 3.1: Computation time comparison between OpenSfM and VisualSfM

It can be concluded that as OpenSfM provides a relatively accurate reconstruction and also can be modified according to one's needs being open-source, it was decided to proceed with this software.

3.3 OpenSfM workflow

OpenSfM [17] is an open-source Structure from Motion library developed in Python and still being developed with new developers willing to contribute to the project. It consists of several modules which perform the estimation of camera poses and point cloud reconstruction from images in a step-by-step manner.

There are two ways to setup the project:

1. Clone the repository, its submodules and dependent libraries:

```
git clone --recursive https://github.com/mapillary/OpenSfM}
cd OpenSfM
git submodule update --init --recursive

python setup.py build
```

Code example 3.1: Installing OpenSfM from a repository

Manually install the dependent libraries:

- OpenCV
- OpenGV
- Ceres Solver
- NumPy, SciPy, Networkx, PyYAML, exifread

2. Using a Dockerfile run a container

The docker file is presented in listing A.1 and was taken from a forum thread [18].

Here is the list of main modules with the description of what are they responsible for:

1. extract_metadata

   This module is responsible for extracting all the available metadata from the dataset images. If the dataset provides an exif_overrides file it will read all the values from that file. The most important values which can be overwritten are the focal length of the lens which was used to take the photo and GNSS position of the frames, as the first field might increase accuracy of the resulting reconstruction and the second one allow to georeference the reconstruction either by manually estimating the location or if there is another device which collects GNSS data along with images. Other fields are hight, width, projection type, orientation, capture time. The GNSS and the capture time also allow to decrease the computation time and in some cases increase accuracy by restricting the feature matching pairs to a fixed number of neighbours.

   After the metadata has been read from the dataset images and from the exif_override file it saves this metadata for each image into a separate folder so on the next run it would not repeat the same operation.

2. detect_features

   This module is responsible for detecting features on the dataset images. The features are places on an image where the intensity significantly changes, representing corners, edges and dots. The process of feature detection is needed to transform vision information into a vector space, which further might be used to perform mathematical operations (transformation, rotation, comparison etc.). The features are extracted for each image in using OpenCV implementation of the specified feature extraction algorithm and then also are saved into a separate folder.

There are various feature extraction algorithms like SIFT, SURF, AKAZE, HAHOG, ORB, which differ in accuracy and performance. In general, the most widely used algorithms are SIFT and SURF, where SIFT provides better results and SURF has lower accuracy but works about 3 times faster. In the experiments the default HAHOG feature detector was used which is combination of Hessian Affine feature point detector and HOG descriptor.

3. match_features

   This module is responsible for matching image pairs. First it creates a set of images pairs depending on a configuration file. It can either make a combination of all images, which would make the reconstruction process the most time consuming, create matching pairs by GNSS data, by name order or by a timestamp. Then after the matching pairs have been computed, its goes through each pair and estimates a match value. These matches are then being saved to the dataset folder and later would be used to track common features' motion among images.

4. create_tracks

   This module is for creating tracks from pair-wise matches retrieved in the previous step. A track basically represent a set of feature points which was detected both images of image-pair and correspond to the same place in real world.

5. reconstruct

   This is the main module which runs the incremental reconstruction algorithm and the main goal of this module is to estimate the 3D positions of feature points (tracks from the previous stage) and positions of cameras, which represent dataset images with some translation and rotation. It starts the process by loading the data from the previous steps and creating initial reconstruction made of two images. After that it grows the initial reconstruction using the remaining images. At this step the basic point cloud along with camera positions already can be observed in using the internal web-viewer.

6. mesh

   This module computes a triangular mesh of the scene and is used to simulate a smooth transition between images in the web-viewer. The meshed reconstruction is saved in a separate file and is only intended for demonstration purposes.

7. undistort

   This module attempts to remove radial distortion form the reconstruction as well as from the images (in case if the radial distortion parameters have been provided). The undistorted version of the reconstruction is intended for further computation of depthmaps.

8. compute_depthmaps

   This module allow to extract a dense point cloud of the reconstruction in a format which can be used by viewers which support .ply files (e.g. MeshLab).

In order to be able to perform path planning and path following, the localization process should take as little time as possible. With the default setup, the OpenSfM recomputes everything using the available input dataset and doesn't allow to add new images to an existing reconstruction. The process of feature detection, feature matching, tracks creation and incremental reconstruction might take a significant amount of time depending on the amount of input data. Thus, the code needs a modification which would allow the software to use already computed reconstruction to dynamically add new frames to the reconstruction.

The main idea of performing localization with the OpenSfM is to execute reconstruction process for an area of interest using a georeferenced set of images and then using the reconstruction find a match for only one frame.

The modifications applied to the OpenSfM are listed in Appendix A.4. These changes include:

- dataset auxiliary functions for saving and loading intermediate results (e.g. tracks)
- match_features for each matching method compute matches only for new images
- create_tracks load previously computed tracks, compute tracks only for new images
- reconstruction load existing reconstructions, for each reconstruction execute incremental reconstruction using new images

The Table 3.2 shows the computation time for each reconstruction stage for different scenarios. The dataset taken for this experiment consists of 30 rectified images taken from

Experiment	extract metadata	detect features	match features	create tracks	reconstruct
Initial run without GNSS	1.5 s	11.4 s	56 s	2.7 s	33.4 s
New image without GNSS	0.7 s	4.4 s	49.2 s	3.2 s	7.3 s
Initial run with GNSS	2.8 s	12.9 s	13.6 s	2.5 s	35.9 s
New image without GNSS	0.7 s	6.1 s	46.4 s	2.8 s	11.0 s
New image with GNSS	0.7 s	6.0 s	8.9 s	2.7 s	7.6 s

Table 3.2: Computation time comparison for different scenarios

the Iseauto vehicle fish-eye camera and an additional file which contains GNSS positions for each frame.

To sum up the table, to achieve the best performance the dataset used for reconstruction should have GNSS metadata as it will speed up the feature matching process which takes the most time. Additionally, if a new image has GNSS data, even an approximate value just to reduce the number of matching candidates, it will add it to the reconstruction significantly faster.

When there is no GNSS data available, OpenSfM perform matching process for each possible image pair combinations. Each such pair feature matching with the current setup takes about 1.5 - 2 seconds and the bigger the area of interest is, the more time it would take to add a single image, unless it has a hint of where the car might be at this moment. Such a hint might be received from an Inertial Measurement Unit (IMU), an electronic device which measures a body's specific force and angular rate.

4. Factors which influence the accuracy of camera pose estimation

The structure from motion technique completely depends on the visual data from cameras, namely on the quality of the input images. There are various factors which can influence on the reconstruction process. Considering the localization problems based on camera images it is important to take into account that the environment might look different during different time of a day, during different seasons or even during different years. This chapter will cover the main aspects of the image acquisition process.

4.1 Camera and lenses

All modern cameras are equipped with lenses whose main purpose is to focus the light rays from the objects of interest onto the camera sensor. When a camera sensor is exposed to light, each single point of the sensor would collect a lot of light rays making it impossible to receive an image. There is a projection model called pinhole perspective (proposed by Brunelleschi in 15th century), which solves the problem of overlaying rays by making a very small hole in a surface between an object of interest and an image plane which would let a single ray from the object to be visible only on a single spot on the image plane. The resulting image on the image plane will be inverted horizontally, but would the light rays would not overlay meaning that the image should be sharp.

Nevertheless, in reality, making a hole so tiny that it will allow to pass only a single ray of light is very hard if not impossible and, most importantly, such small hole wouldn't let too much light to go through, making the resulting image really dark. In order to focus the light rays onto the image plane (or sensor) while keeping the amount of light on a sufficient level, lenses are used.

There is a wide variety of lenses which have different sizes, different optical properties and purposes. The cameras which are installed in the Iseauto vehicle have fish-eye lenses which means that the image they provide has barrel distortion. This is a form of optical aberration when magnification decreases from the center of an image to its edges. One of the widely used approach to solve this problem is to use an Open-Source Computer Vision (OpenCV) library and a flat printed pattern (for example, a grid of black and white boxes with known dimensions). With this setup as described in paper [19],

> to calibrate one camera it takes several images of the calibration pattern in various orientations. In the process either the calibration pattern or like in this case the camera must be fixated while moving the other one around. In each image the feature points of the pattern have to be detected by a pattern specific feature detection algorithm like the algorithm for chessboards by Bennett [20]

Applying this algorithm would provide a non-distorted image, but as image has to be rectified, the edges of the image would be cut out.

For the cameras (Basler aca2040-gc35) which are installed in the vehicle with a wide angle lens (Sunex DSL315) and used for the datasets, the barrel distortions parameters has already been estimated and can be found at [21]. The approach to correcting for the non-linear relationship is to fit a suitable polynomial to the data points. A general function for the latitude φ might be $\varphi(r) = a_0 + a_1 r + a_2 r^2 + ... + a_i r^i + a_n r^n$. And for the DSL315 this function is $\varphi(r) = 0.748x - 0.0272x^2 - 0.0032x^3 - 0.0198x^4$

A simple script might be used to rectify a group of images using a command-line ‘’convert” tool, which accepts two arguments (distortion type and distortion parameters)

```
#!/bin/bash

mkdir ./images
for img in ./*.jpg; do
        convert $img -distort Barrel "-0.0198 -0.0032 -0.0272 0.748" ./
    images/$img
done
```

Code example 4.1: Script for rectification of DSL315 frames

The result of tool execution with the mentioned parameters is presented on the Figure 4.1. Although the rectification produced more useful images, a significant part of images was cut out as highlighted on the Figure 4.2.

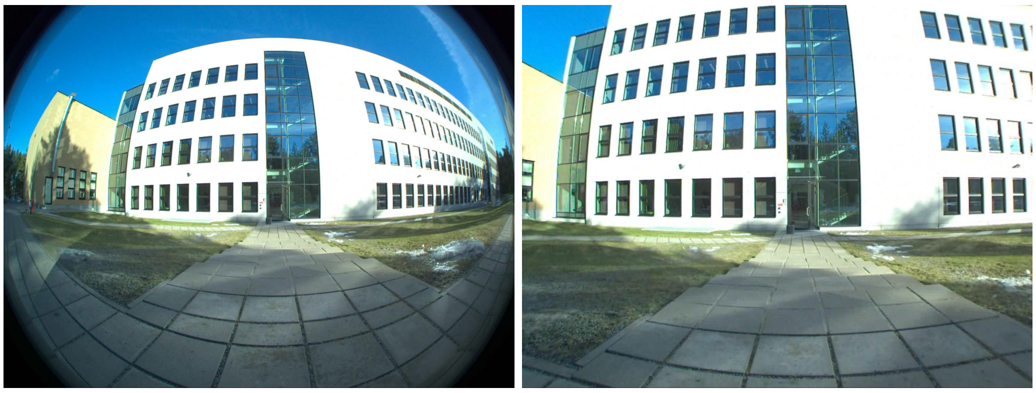

(a) Before rectification (b) After rectification

Figure 4.1: Barrel distortion rectification

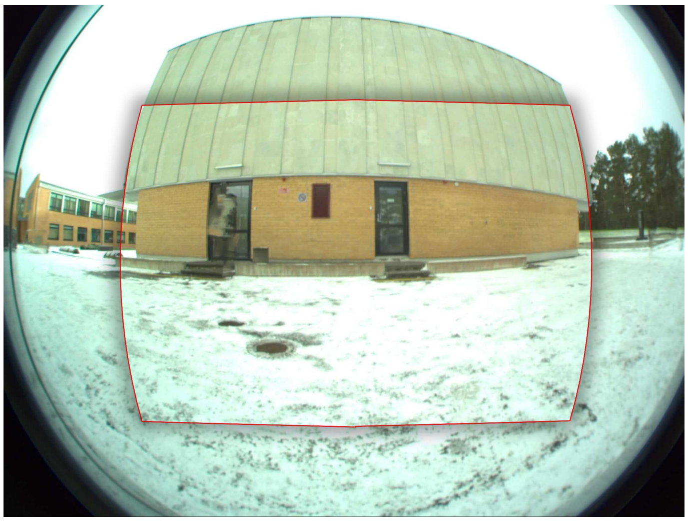


Figure 4.2: Highlight of the area received after rectification process in relation to the original frame

Although such cameras have distortions and have to be calibrated using software, they have an advantage over other cameras as they provide a wider field of view.

Another property which differ lenses is called a focal length, which stands for the distance between a point where light rays converge forming a sharp image and a sensor. The focal point primarily defines the field of view: the shorter the focal length, the wider is the field of view. For the same camera sensor lenses with different focal length would provide images with different magnifications and it would will seem that images were made from different distances while in reality the camera would remain on the same spot.

Taking this into account, it important to understand that the lens used to create an initial reconstruction and a lens, which is going to be used during the localization, have to be of the same focal length. The SfM algorithm estimates position of a camera based on the image features and their translation among a series of images and if an initial reconstruction was made using a camera with low focal length lens, then an image made with a higher focal lens camera would be recognized to be closer to objects than it is.

The last main aspect of a lens is its focus, as even slightly unfocused lens can significantly decrease the resulting quality as blurry image would reduce the amount of detected features. Considering the Iseauto bus, which is the main object of interest within the localization problem, it has three fish-eye lenses placed on the front and on the sides of the car. The lenses are not securely attached to the camera body, but are rather in a floating position where it can be screwed in or out and thus the focus would also change. Initially, the lenses seemed to be in focus, but on a closer look it was noticeable that it is slightly out of focus and objects' edges were not as sharp as they should be, especially for further objects. Slightly adjusted the focus on the lenses although didn't bring much difference in appearance for a common user but noticeably increased the resulting reconstruction quality. Considering that the camera in its current placement is easily reachable, the lens should be fixed in place after being properly adjusted.

4.2 Exposure

Exposure defines how bright or dark a resulting image would be. There are two settings which affect the exposure. These are: the shutter speed and the aperture.

The aperture defines the opening through which light travels to the sensor. The bigger the aperture is the more light a sensor will receive, but at the same time with smaller aperture it is easier to adjust focus on objects at different distances.

The shutter speed defines the amount of time during which a sensor will be exposed to light. The longer the shutter speed the brighter the resulting image would be, but at the same time non-static objects would result to be blurry (motion blur). In case if the camera is not static then the whole image would be blurry and unusable.

There is also a camera ISO, which stands for a sensor sensitivity, meaning how hard each pixel received by the sensor would be brightened. This setting is usually used only in those cases when it is not possible to get a sharp image and at the same time bright image using the two previously mentioned settings. This setting should be the last one to adjust, as the higher the sensitivity is set the more noise the image would get.

Keeping in mind these parameters it is important to define how they should be adjusted in order to get images with the highest amount of information without too dark or too bright spots. The main problem is that outdoor in a changing environment, the exposure has to be dynamically adjusted as the brightness can significantly change for example due to changing natural precipitation as illustrated on the Figure 4.3.

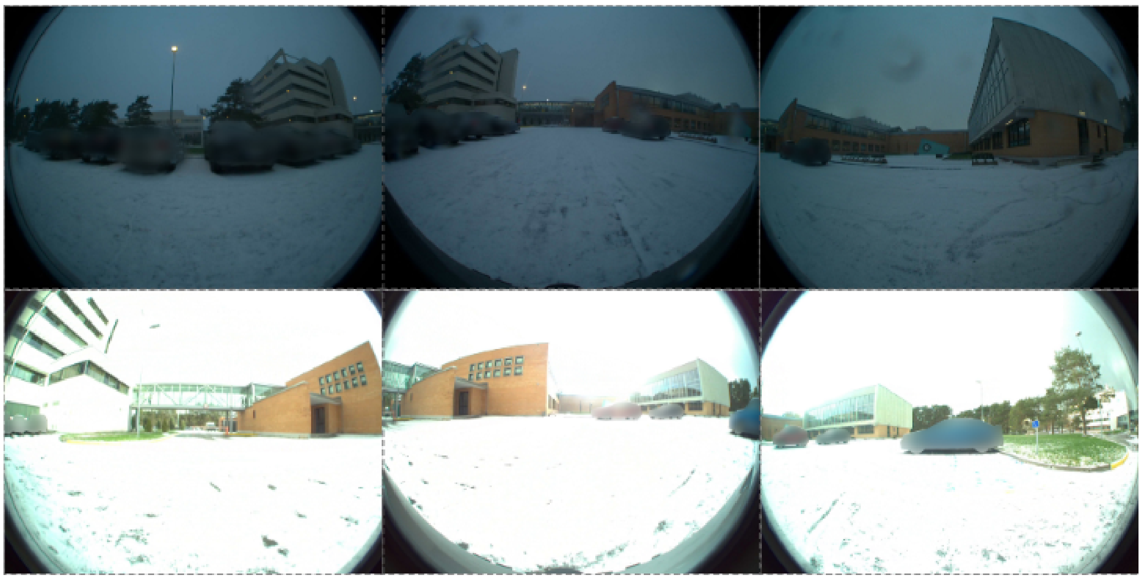


Figure 4.3: Example of over- and underexposed images (frames were collected with the same settings with a 10 minutes difference)

4.3 Weather conditions

During the whole year the weather has a great influence on the resulting image. Firstly, this includes precipitation, e.g. snow flakes, rain or fog, which reduce the general visibility and also after falling on a lens might blur (Figure 4.4), distort or refract the environment. In addition to area of view, the weather change might also change the lighting conditions as was previously shown on Figure 4.3. The fog reduces the viewing range in general and might fluctuate in density.

Secondly, the clouds has a significant influence on how many details it is possible to retrieve from the environment. During a day, when the clouds absent or present in very small amount, the sun as a single lighting point might create a lot of contrast spots where darker ones are harder for feature detection. As opposed to that, during a cloudy day the clouds diffuse the sunlight and serve as a light area. When an environment is illuminated with a light area, it is easier to detect the actual structure features.

We can expect that same environment might produce different images in terms of

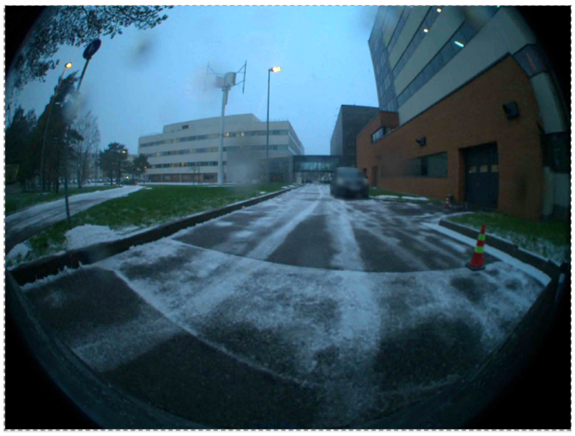


Figure 4.4: Example of precipitation: a snowflake blurred a part of the image

shadows. Assuming we have a flat rectangle surface which has some object on one of its sides. When the main outdoor source of light illuminates this surface without any obstacles, all the surface would look smooth and solid. But when the light source is positioned in such a way that some part of the light doesn't reach the surface, being absorbed by an object nearby, the image of the surface would be separated into 2 parts: lighter one and darker one. The transition between these two parts might be sharp enough for an edge detector to be classified as an edge, although in reality there are no edges. Similarly to this, an edge that exists might not be detected if the color/brightness intensity of the object would be similar to the background. This phenomenon is was mentioned in [7]:

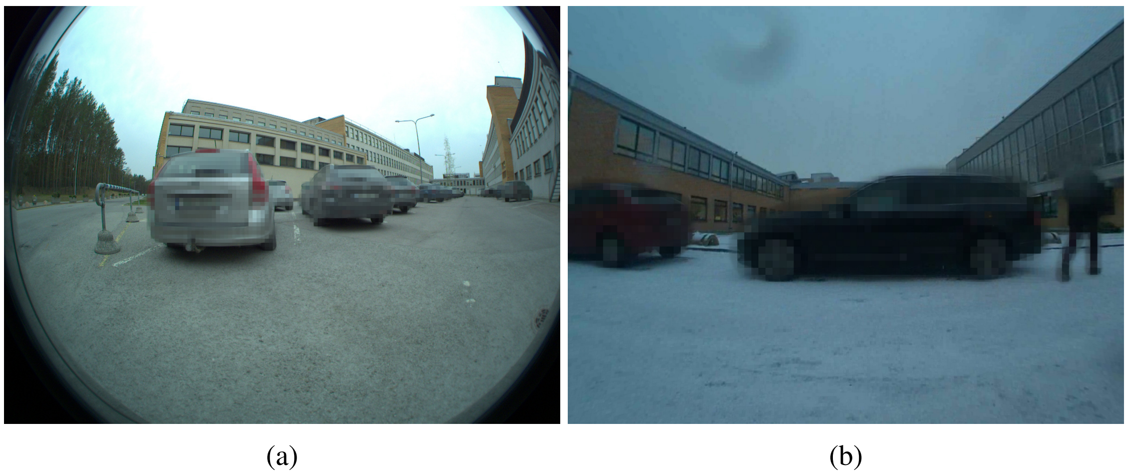
> We think of sharp changes in image intensity as lying on curves in the image, which are known as edges; the curves are made up of edge points. Many effects can cause edges; worse, each effect that can cause an edge is not guaranteed to cause an edge.

These effects mostly occur when the light mostly arrives from a single point source unlike area sources where light is scattered and producing soft shadows (e.g. when sunlight is scattered by clouds). According to [7],

> The sun is usually modeled as a distant bright point. Light from the sun is scattered by the air. In particular, light can leave the sun, be scattered by the air, strike a surface, and be reflected into the camera or the eye. ... A natural model of the sky is to assume that air emits a constant amount of light per unit volume; this means that the sky is brighter on the horizon than at the zenith because a viewing ray along the horizon passes through more sky.

4.4 Physical obstacles

For performing structure from motion algorithm, the scene should be as static as possible and the object of interest should be well visible. There are various possible factors which brings difficulties in visibility. They include moving people, parked vehicles, fences and also foliage. The first two might continuously change during a day or a week. The main problem is that the cameras currently are installed close to the ground and although it can be usable for obstacle detection it blocks a lot of information as soon as any object approaches or is being approached close enough as shown at Figure 4.5.



(a) (b)

Figure 4.5: An example of visibility problem when vehicles cover structure

In general, if not all the structure is covered, OpenSfM should still be able to localize the images matching and tracking the visible features, but not always. For creating an initial reconstruction the scene is still has to be as empty as possible, otherwise it does not produce a usable output (Figure 4.6)

4.5 Structural properties

Problems with structural properties occur when a structure is homogeneous, repetitive, transparent or reflective. This might be affect even if matching is performed by closest neighbours.

Homogeneous means that a surface is uniform in color or texture (Figure 4.7a). When a surface is homogeneous it doesn't have enough distinct features to be tracked between a set of images. When a repetitive structure or objects present, the feature matching

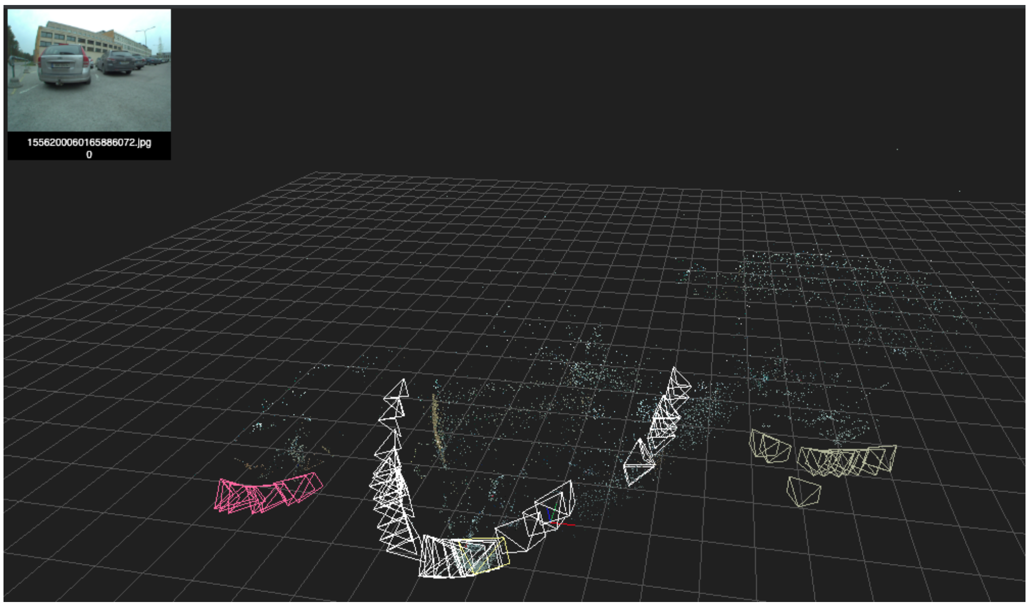
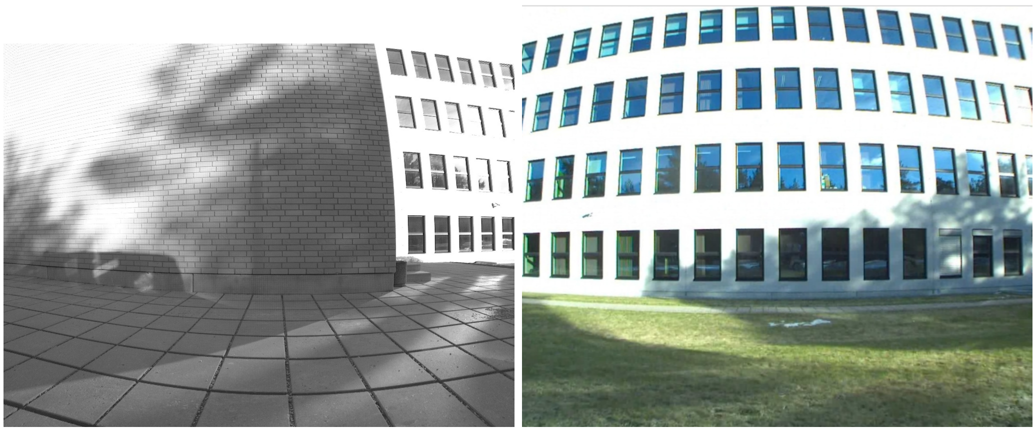


Figure 4.6: Bad reconstruction caused by the presence of parked vehicles which cover the view. The dataset is georeferenced and rectified

process might match images from different locations assuming that as the features match, they correspond to the same object. Figure 4.7b shows an example when it is possible to have repetitive structure along with reflections, which is an another problem because reflections might bring additional features which should not be recognized.



(a) Homogeneous (b) Repetitive with reflections

Figure 4.7: An example of homogeneous and reflective structures

5. Geographic coordinate system and georef-erencing

In order to be able to estimate an image geolocation within a reconstruction, it has to be georeferenced. This section will cover basics about coordinate systems, available options to georeference the OpenSfM reconstruction and also an attempt to extract geoposition from the GNSS device installed on the Iseauto.

5.1 WGS84 and UTM

The WGS84 is a standard definition of a global reference system for geospatial data and serves as a reference system for GPS. It was established in 1984 and last updated in 2004 with the responsible organization called National Geospatial-Intelligence Agency (NGA). This system projects coordinates onto an ellipsoid which has been estimated to best fit the Earth shape with its origin in the Earth's center of mass. The coordinates are usually measured in decimal degrees or degrees, minutes, seconds. The latitude values reside within the -90 degrees (to the South) and the +90 degrees (to the North). The longitude values reside between -180 degrees (to the West) and +180 degrees (to the East).

The UTM is a standard coordinate system which divides the Earth plane into 60 vertical zones each of which is six degree wide. The projection tries to minimize the distortion in each zone, where the error is minimal near the central meridian and increases when moving away. The coordinates itself have an easting range between 167 000 and 833 000 meters and a northing range between 1 100 000 and 9 300 300 [22].

For example, on figure 5.1 from [23] the UTM zones for Europe is presented.

Both of these systems are used in the OpenSfM: WGS84 for georeferencing and

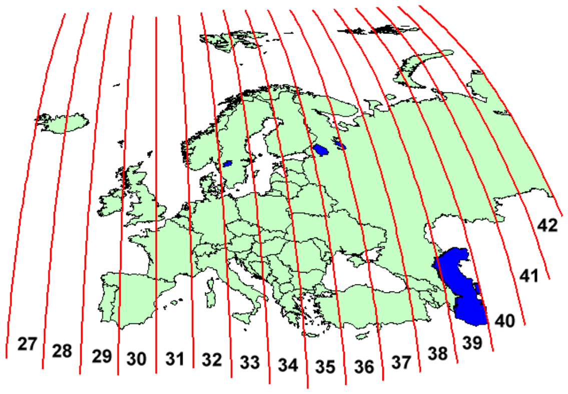


Figure 5.1: Map of Europe divided into UTM zones [24]

UTM is the output format of the images' coordinates.

5.2 Georeferencing the reconstruction

In the resulting reconstruction the position of the 3D points and cameras' positions are stored in world coordinates relatively to a reference frame, where X axis points to the east, Y axis points to the north and Z axis points to the zenith. The camera position here represents a position and rotation of a single shot in relation to other shots and points.

When there is no GNSS data provided for the input frames, the reference frame contains zero coordinates and the reconstruction process tries to rotate the world reference frame so Z is pointing in the vertical direction assuming that images were taken at the similar altitudes and the up vector of the images correspond to the up vector of the world. When GNSS data is provided, the software uses the topocentric reference frame for the world coordinates reference in a form of latitude/longitude/attitude. This reference frame is then used to compute the geo coordinates of all available images.

There are three possible ways to connect the reconstruction with the GPS data:

- Using Ground Control Points (GCP)

  Ground Control Points are specific landmarks which are visible on the provided images and which has a geospatial positions specified. To provide the dataset with GCP it is required to create a file within the dataset which would specify the projection format to be used for the geo coordinates and then row by row GCP information, including image name, pixel coordinate (x,y) and geospatial position (latitude, longitude, altitude). Each GCP can be observed on several frames.

  There are three possible formats in which the geospatial coordinates can be specified:

  - WGS84 - standard latitude, longitude and altitude in degrees.
  - UTM - format which splits the globe into 60 zones and uses coordinates in meters.
  - Proj4 - format which allows to specify a projection with various parameters.

  The GCP points are used by the OpenSfM during two stages:

  - Alignment: points are used to move the reconstruction in such a way that GCP align with the GNSS positions. This stage requires at least two observations for each GCP.
  - Bundle adjustment: all GCP points are used to refine the reconstruction.

  During an experiment I have provided a test dataset with several GCPs defining each of them on several frames, but the resulting GNSS coordinates were not usable. Probably the number of GCP were not enough or the image quality was not good enough, but this approach takes too much effort and time to manually define a GCP pixel coordinate on a frame, discover its geospatial position and save it into a file.

- Using GPS data available within the images' metadata.

  This approach is the easiest if the camera is able to capture a frame and save a GPS position. In this case it is enough to just submit all the frames into the software and launch the computation.

  As long as all the images contain the Global Positioning System (GPS) data it is also possible to boost the performance of the reconstruction process by restricting the number of neighbours used by the software to capture motion, so it would compare frames which correspond to totally different places.

- Overriding metadata of images.

  This approach suggests creating a file in the root folder of a dataset which will override the images' parameters in a form of a json file. There are various parameters including GNSS location in a form of (latitude/longitude/altitude) and Dilution Of Precision (DOP) in meters.

  Providing the GNSS locations for the images might be done manually for each image by identifying the position on a map and saving the coordinates or taking an input from a navigation system.

5.3 Extraction of geopositions from the GNSS

In the Iseauto project it is possible to utilize the GNSS receiver which provides location data with errors as low as 2 cm using the VRS Now GNSS correction service (depending on conditions).

At the time of writing the thesis, there are two ROS topics which contain geospatial data: nmea_sentence and gnss_pose. A message examples are presented in a block 5.1. There are various messages in nmea_sentence, but the most relevant one is GNGGA. It includes such fields as: Coordinated Universal Time (UTC) time, latitude, longitude, number of satellites in use and altitude. The gnss_pose messages contain coordinates in LAMBERT-EST of 1997 (L-EST97) [25] coordinate system and the nmea_sentence messages contain positions projected onto a three-dimensional spherical surface which is more usable as is more widely used.

```
#gnss_pose
---
header:
  seq: 1636
  stamp:
    secs: 1555072188
    nsecs: 564685106
  frame_id: "map"
pose:
  position :
    x: 538012.246977
    y: 6584245.13438
    z: 23.076
  orientation :
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
---
#nmea_sentence
header:
  seq: 1639
  stamp:
    secs: 1555072188
    nsecs: 764707088
  frame_id: ''
sentence: "$GNGGA,122854.00,5923.66687006,N,02440.14221173,E,4,09,2.1,23.078,M
    ,18.400,M,0.8,2012*60"
```

Code example 5.1: Gnss_pose and nmea_sentence sample messages

In order to build an initial georeferenced reconstruction which could be used for localization of new frames, it is necessary to record the area of interest in a form of frames from the side cameras and GNSS positions of the frames. Although modern smart cameras are able to make images along with GNSS positions, the cameras installed in the Iseauto bus are only able to collect visual data and, thus, GNSS location can be collected separately and then merged with the images.

In order to extract frames and GNSS locations, it is enough to subscribe to the nmea_sentence and to one or several camera topics. Each Robot Operational System (ROS) message consists of a Header (sequence ID, Unix timestamp, frame ID) and required data fields. One of the possible ways to synchronize images with GNSS data is to use the timestamp from the messages' headers. It should be also possible to save this data in a real-time but saving these two groups of messages explicitly gives more flexibility.

The script A.2 collects the data from the left/right cameras and from the GNSS. The images are saved into separate folders according to camera positions and the filename corresponds to the timestamp of the message. GNSS data is stored in two formats in two separate files. One of them is used to create a .kml file which is convenient for exploration and another file is used to generate an exif_overrides.json file which would describe GNSS location for each image and would be used by OpenSfM.

The exif_overrides.json file can be generated with the script A.3. It uses the folder with timestamp-named images and a file with timestamp-referenced GNSS locations, generated by the script A.2 and attempts to find for a each image a corresponding GNSS location and save this information in correct format into separate file. Due to the fact that messages from the ROS topics are generated with different nanoseconds value (especially if they have different frequency), it was decided to match images with a seconds precision as difference in nanoseconds might filter out a lot of data. When a match is found, it is written into the output file. If the image doesn't have a corresponding GNSS position it is removed from the folder.

The GNSS device was installed and configured only recently and is still under development. Although it provides accurate geospatial data, it seems that there are some problems with message generation (at the same frequency with cameras (5 Hz)) as the GNSS messages are shifted and stretched in time. For example, we made two test-drives recording data from cameras and GNSS. As can be seen at Figure 5.2, GPS positions from the first file correspond to the path which was covered to the starting point before we started to record anything. The second file contains GNSS positions which correspond to the last part of the first run, but although this path was covered in about 30 seconds, the GNSS positions cover about 90 seconds of recording. Later it was discovered that it is possible to get correct data with correct timestamp at a frequency of 2 Hz, but getting data at higher frequency has to be investigated.

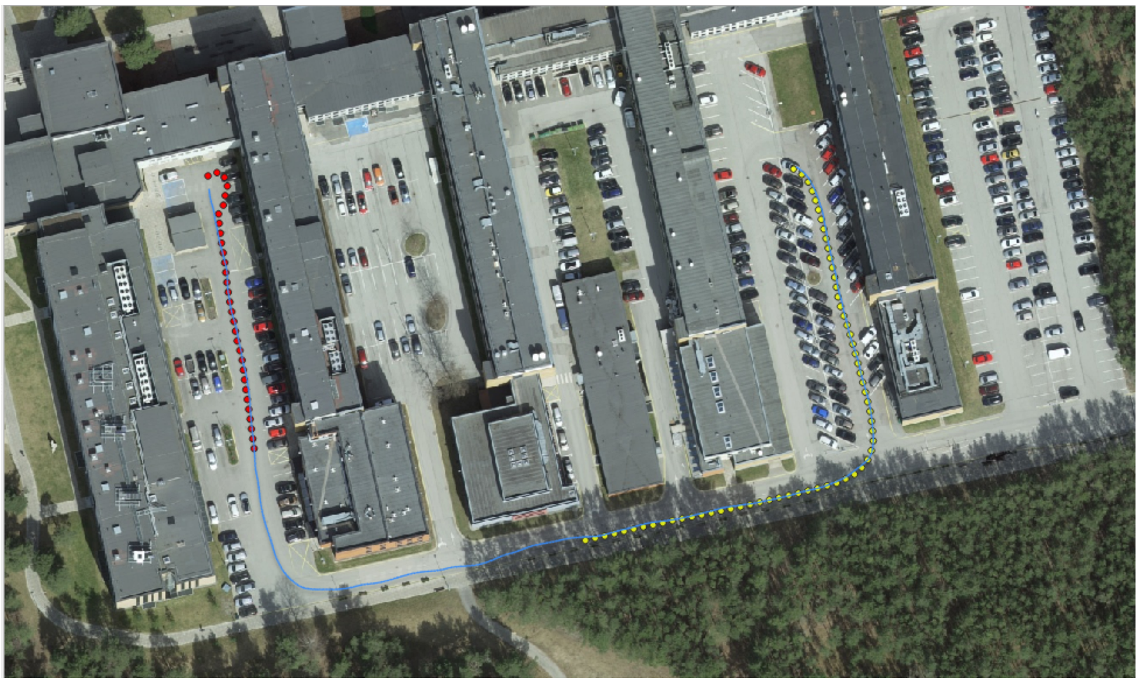


Figure 5.2: Visualization of recorded GNSS locations during two test-drives (red line - first recording, yellow line - second recording, blue line - the actual path covered during the first recording)

6. Experimental Results

In this chapter an account for an experiment of collecting data with Iseauto and performing visual localization on the data is given.

First of all, to be able to perform localization of a new image, an initial reconstruction has to be computed. The dataset required for initial reconstruction is a folder with a set of images of an area of interest and an OpenSfM config file, which overwrites default settings.

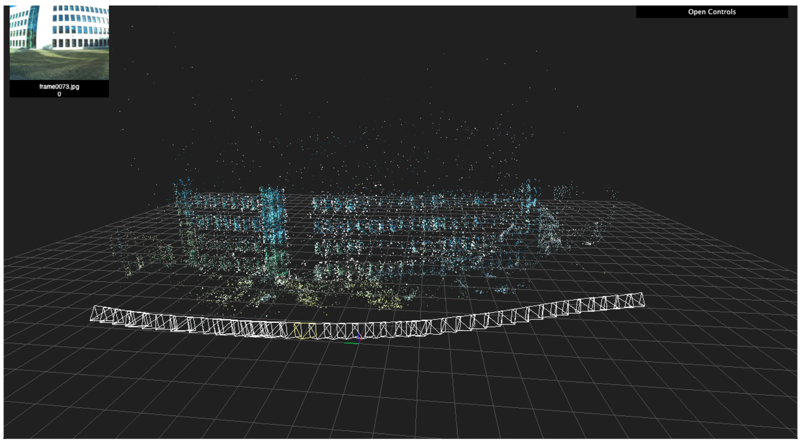
The dataset should also contain GNSS coordinates of the vehicle for the purpose of estimating the localization precision for the images.

In the Iseauto bus, the data extraction can be done in real-time from the vehicle or from a rosbag recording. In both cases it is enough to subscribe to the left/right camera and nmea_sentence topics. A subscriber written in Python was mentioned in previous chapters and is listed in the Appendix A.2. The images from the first topic can be grayscale as feature detection doesn't use the colors and have to be rectified if barrel distortion is present using the "convert" tool as described in the previous chapters.

The last topic is used to retrieve GNSS positions for the images, but at the time of writing the thesis the GNSS position messages were recorded with 2 Hz frequency, while the images are recorded at 10 frames per second.

6.1 Manual georeferencing

For the first experiment the GNSS data for the extracted images was assigned manually using a Google Maps. In either way, the GNSS locations should be saved in a specific .json file in a specific format understandable by OpenSfM.



(a) Point-cloud loaded using a web-viewer

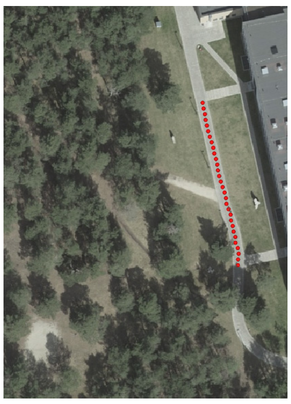

(b) GPS locations of the images

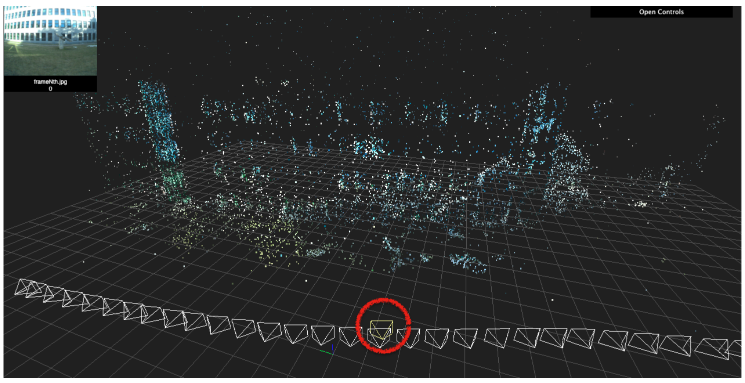
Figure 6.1: Initial reconstruction

The config file is optional and is used to override the default settings. The most useful parameters which should be overwritten are: a number of threads for parallel computations (should be configured according to the hardware), a number of image neighbours for feature matching process (by timestamp, by name or by GNSS location) and maximal distance to the neighbours. Depending on the number of threads, quality of images, number of images and applied matching limitation this step might some amount of time (starting from 1-2 minutes for a very small dataset). After an initial reconstruction has been done and it is of a good quality it should be possible to perform localization.

For the actual localization, a one need to extract an image from a camera, rectify it (in the case of a wide angle lens) and add it to the dataset. If possible, it would be very useful to provide the image with an approximate geoposition (manually or, for example, using IMU to estimate possible location based on the previous location). This rough position would be used by the OpenSfM only to restrict the number of images for feature matching and would give a boost in performance.

A manually georeferenced sample dataset of a 65-meter area as shown at Figure 6.1 was reconstructed within about a minute. The dataset for the initial reconstruction was taken from a one direction run and the image to localize was taken 10 minutes later on the way back.

To extract the geolocation coordinates the following command is used:



(a) An updated reconstruction with a new image

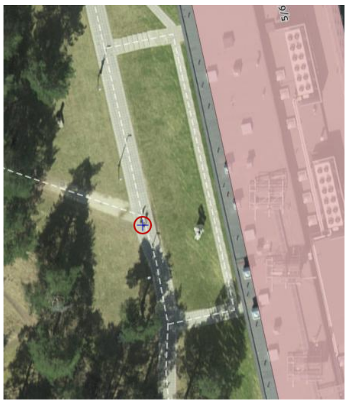

(b) Coordinate of the newly added image produced by the software

Figure 6.2: Result of localization of an image

```
opensfm export_geocoords  --proj="+proj=utm +zone=35 +north +ellps=
    WGS84 +datum=WGS84 +units=mm +no_defs" --image-positons
    path_to_the_dataset
```

Code example 6.1: Script for rectification of DSL315 frames

This generates a list of coordinates by a frame name in a UTM format for the 35th zone. As shown on the Figure 6.2a the position of the camera is estimated correctly.

The process of estimating of a new image within the initial reconstruction took 65 seconds without a hint (takes about 25 seconds with a GNSS hint). A bigger reconstruction would take more time but as the functionality of getting the GNSS locations for images automatically emerged only at the time of finishing the thesis, bigger areas were not tested.

6.2 Automatical georeferencing

For the next experiment a bigger dataset was recorded along with GNSS data from the GNSS device. The images were taken from two cameras for comparison: one was directed to the left (fisheye lens) and another one to the forward-left (non-distorted lens lifted about 150 cm above the ground). As can be seen on Figure 6.3 rectified images from a fisheye lens produced much worse results even with GNSS data and it does not seem reasonable to attempt to localize with respect to such reconstruction. Utilizing images

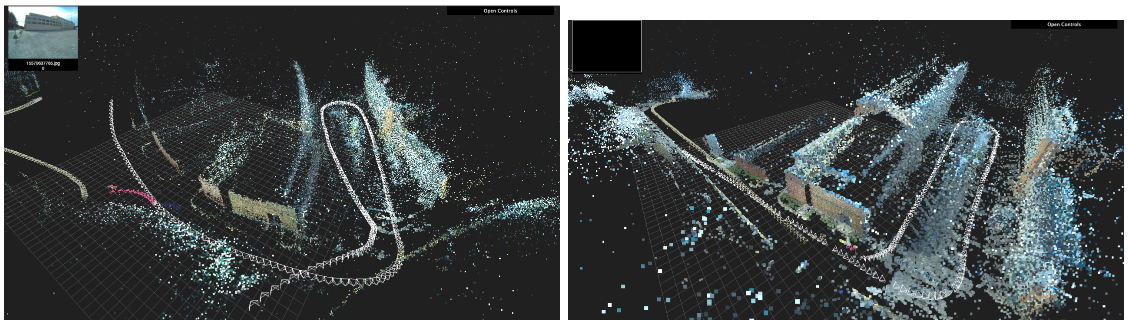

(a) Using rectified images from a fisheye camera (b) Using images from a rectilinear camera

Figure 6.3: Reconstruction of the same area with different lenses

from fish eye lenses requires the SfM approach to be elaborated as in e.g. [7].

In order to estimate the localization error, some images with GNSS data were not included in the initial reconstruction and were used as test images. This way there would be no direct image match with the dataset and the position should be computed in relation to other images. Although it could be more useful to take data from different recordings and, with under different environmental and light conditions, it was out of scope of the current thesis because of time limits and the fact that subsystems such as GNSS were developed in parallel. In the data sets collected during many of the early experiments, it often happened that the coordinate data was completely unusable. In Figure 6.4 two similar routes were recorded with a 1 minute interval and only one of the recordings contains usable GNSS data. The localization result of 15 test images is summarized in the Table 6.1 and visualized on the Figure 6.5

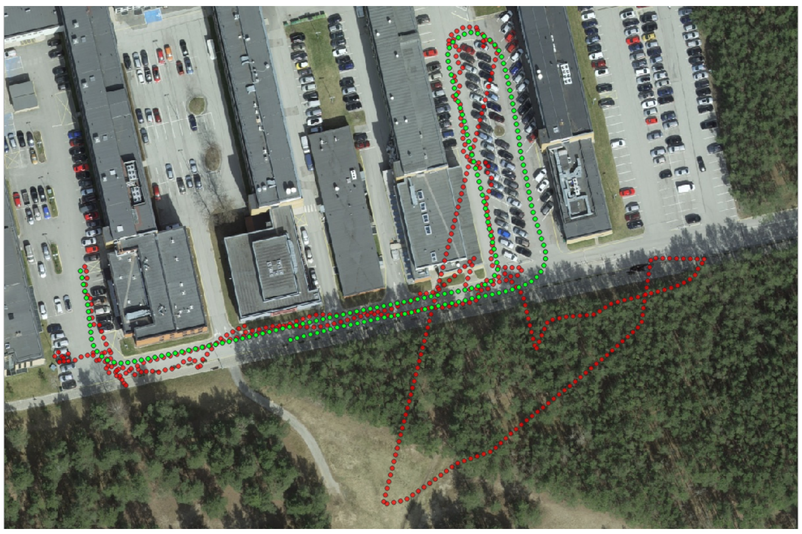


Figure 6.4: Illustration of two GNSS data recordings from the Iseauto GNSS device

An average error according to the test results is fluctuating between 0.3 and 1.5

Image seq	GNSS geoposition	Estimated geoposition	Error
1	59.3937204442, 24.6707128648	59.393723, 24.670715	0.3 m
2	59.3937692218, 24.6711889497	59.393774, 24.671185	0.578 m
3	59.3938169573, 24.6716236708	59.393821, 24.671618	0.554 m
4	59.3939723272, 24.671755331	59.39397, 24.671702	3.02 m
5	59.3943693802, 24.6716013055	59.394361, 24.671579	1.574 m
6	59.393743884, 24.6709495815	59.393747, 24.670944	0.470 m
7	59.3937936787, 24.6714144602	59.393796, 24.671407	0.497 m
8	59.393879209, 24.6717783962	59.393876, 24.671762	0.998 m
9	59.3940674252, 24.6717060073	59.394056, 24.671688	1.633 m
10	59.3942685987, 24.6716375402	59.394237, 24.671652	3.615 m
11	59.3944661977, 24.6715488305	59.394457, 24.67153	1.481 m
12	59.394516873, 24.6714009375	59.394511, 24.671397	0.691 m
13	59.3944589815, 24.6712752842	59.394464, 24.671277	0.568 m
14	59.394369202, 24.6713067248	59.394367, 24.671282	1.426 m
15	59.394274858, 24.6713594573	59.394283, 24.67136	0.908 m

Table 6.1: Comparison of actual and computed geopositions with an error in meters

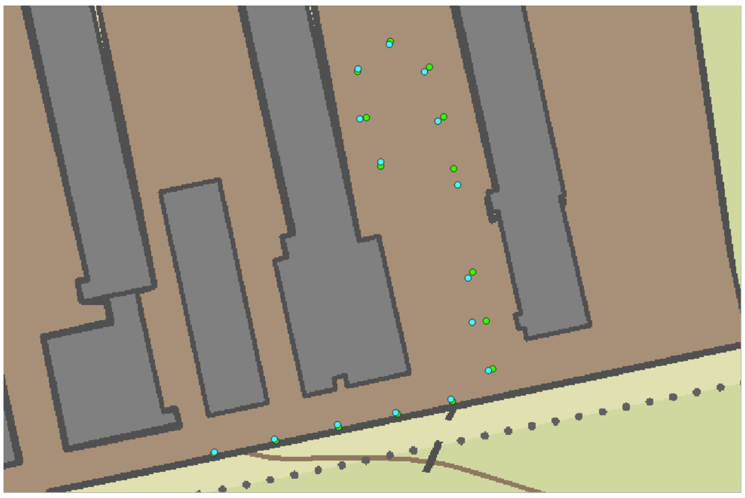


Figure 6.5: Illustration of actual and estimated image geopositions (green and cyan respectively)

meters, and only 2 times the error is above 3 meters. In almost all cases, the estimated position has a shift towards the direction of the camera, which is consistent with the camera positioning within the vehicle. In general, the conditions for the dataset were good (despite the presence of reflections both from the windows of the buildings and from the vehicle window in front of the lens). As part of the future work it would be useful to collect data the accuracy during different seasons, in more locations and perform a more complete evaluation of various combinations of settings. Within this dataset size, localization takes about 10 minutes, but if a hint is provided, the time can be reduced up to 30 seconds (depending on the precision of the hint). The feature matching process across image pairs takes the most time, so when it comes to improving the performance, the matching stage has to be optimized in the first place.

With the average localization error of 1.5 m and computation time for localizing each frame in 30 seconds, the approach not yet usable for real time navigation purposes, but the experiment demonstrates that visual localization is a worth while topic to be researched further.

7. Conclusion

The main purpose of the localization functionality is to enable the vehicle to be aware of where it is in space (on the map) and to be able to follow a path. A path is the result of path planning between the starting and destination point. The ability to carry out path following depends heavily on the performance and accuracy of localisation.

In the current thesis we investigated how visual localisation could be realized using structure from motion. The goal was to run experiments where we could measure the precision of camera pose estimation with the SfM approach based on the sensor set available in Iseauto. We benefitted from the fact that Iseauto has an RTK GNSS positioning solution with localisation precision of down to 2 cm, thus giving us a precise reference framework.

The visual localisation solution developed within the thesis is currently not yet fit for real time localisation as the reconstruction process that is used for camera pose estimation takes on average about 30 seconds if a new image has a rough hint of where it might be (something that can be estimated from inertial tracking and cheap GNSS receivers) and might take minutes if no hint is provided. The computation time also depends on the size of the reconstruction, i.e. how many frames the current part of the map contains. The number of images can be limited by the functionality of dividing the map into submodels (a feature present in OpenSfM). An obvious optimisation would be to utilize the GPU for critical computations (as has been done in VisualSfM). The most time consuming operation is currently feature matching. There are approaches in the literature to optimise the matching as well. Currently the software performs feature matching of all-to-all or neighbour image-pairs depending on the settings.

Considering the experiments, it can be concluded that although the computation time of the camera pose for a frame does not allow to perform localization in real time, the error is fairly small (we observed an error of 0.5-1.5m) considering that we only used a single 3.2 megapixel camera. The result confirms that it is worth while to investigate visual localisation further and try combining information from multiple cameras with the information from the inertial tracking unit. As part of the future work it would be worth while to investigate how to reduce the computation time of the next position relative to the previous position despite the size of the initial reconstruction or the way the reconstruction is divided into submodels.

Additionally, it is worth while to ask how to optimize the overall reconstruction pipeline to allow to perform localization in real time. For example, it should be possible make the feature matching process more parallel or modify the code not to save the intermediate results to the disk but keep critical data in memory.

The benefit of the visual localization approach is that the setup is much cheaper than the use of a GNSS or a LiDAR, therefore can be used in the context where cost needs to be optimized. For the Iseauto project it can be used as an auxiliary localization system.

References

[1] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 287–296, Piscataway, NJ, USA, 2018. IEEE Press. ISBN 978-1-5386-5301-2. doi: 10.1109/ICCPS.2018.00035. URL `https://doi.org/10.1109/ICCPS.2018.00035`.

[2] Raivo Sell, Anton Rassõlkin, Mairo Leier, and Juhan Ernits. Self-driving car iseauto for research and education. 06 2018. doi: 10.1109/REM.2018.8421793.

[3] Autoware repository. URL `https://github.com/autowarefoundation/autoware`. Accessed 2019-01-09.

[4] Anton Rassõlkin, Raivo Sell, and Mairo Leier. Development case study of the first estonian self-driving car, iseauto. *Electrical, Control and Communication Engineering*, 14:81–88, 07 2018. doi: 10.2478/ecce-2018-0009.

[5] Tayyab Naseer, Luciano Spinello, Wolfram Burgard, and Cyrill Stachniss. Robust visual robot localization across seasons using network flows. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 2564–2570. AAAI Press, 2014. URL `http://dl.acm.org/citation.cfm?id=2892753.2892907`.

[6] Jorge Fuentes-Pacheco, José Ruiz-Ascencio, and Juan Manuel Rendón-Mancha. Visual simultaneous localization and mapping: a survey. *Artificial Intelligence Review*, 43(1):55–81, Jan 2015. ISSN 1573-7462. doi: 10.1007/s10462-012-9365-8. URL `https://doi.org/10.1007/s10462-012-9365-8`.

[7] Lionel Heng, Benjamin Choi, Zhaopeng Cui, Marcel Geppert, Sixing Hu, Benson Kuan, Peidong Liu, Rang M. H. Nguyen, Ye Chuan Yeo, Andreas Geiger, Gim Hee

Lee, Marc Pollefeys, and Torsten Sattler. Project autovision: Localization and 3d scene perception for an autonomous vehicle with a multi-camera system. *CoRR*, abs/1809.05477, 2018. URL `http://arxiv.org/abs/1809.05477`.

[8] John Wang and Edwin Olson. Apriltag 2: Efficient and robust fiducial detection. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016*, pages 4193–4198. IEEE, 2016. ISBN 978-1-5090-3762-9. doi: 10.1109/IROS.2016.7759617. URL `https://doi.org/10.1109/IROS.2016.7759617`.

[9] J. L. Schönberger and J. Frahm. Structure-from-motion revisited. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4104–4113, June 2016. doi: 10.1109/CVPR.2016.445.

[10] Carlo Tomasi and Takeo Kanade. Shape and motion from image streams under orthography: a factorization method. *International Journal of Computer Vision*, 9 (2):137–154, 1992. doi: 10.1007/BF00129684. URL `https://doi.org/10.1007/BF00129684`.

[11] Torsten Sattler, Michal Havlena, Filip Radenovic, Konrad Schindler, and Marc Pollefeys. Hyperpoints and fine vocabularies for large-scale location recognition. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 2102–2110. IEEE Computer Society, 2015. ISBN 978-1-4673-8391-2. doi: 10.1109/ICCV.2015.243. URL `https://doi.org/10.1109/ICCV.2015.243`.

[12] N Micheletti, Jim Chandler, and Stuart Lane. Structure from motion (sfm) photogrammetry. pages 1–12, 01 2013.

[13] Wei Zhang and Jana Kosecka. Image based localization in urban environments. In *Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, 3DPVT '06, pages 33–40, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2825-2. doi: 10.1109/3DPVT.2006.80. URL `https://doi.org/10.1109/3DPVT.2006.80`.

[14] Zhiwei Zhu, Taragay Oskiper, Supun Samarasekera, Rakesh Kumar, and Harpreet S Sawhney. Real-time global localization with a pre-built visual landmark database. In *2008 ieee conference on computer vision and pattern recognition*, pages 1–8. IEEE, 2008.

[15] Autodesk Recap: Structure from motion utility from Autodesk, . URL https://www.autodesk.com/products/recap/overview. Accessed 2019-04-22.

[16] Agisoft Metashape: Structure from motion utility from Agisoft, . URL https://www.agisoft.com/features/professional-edition/. Accessed 2019-04-22.

[17] Pau Gargallo. OpenSfM Structure from motion utility. URL https://github.com/mapillary/OpenSfM. Accessed 2019-04-22.

[18] Walter Morawa. OpenSfM Dockerfile, 2017. URL https://stackoverflow.com/questions/42742067/opensfm-setup-py-file-returning-error-after-building-via-docker. Accessed 2019-01-07.

[19] Kühling C. Fisheye Camera System Calibration for Automotive Applications., 2017. URL http://www.mi.fu-berlin.de/inf/groups/ag-ki/Theses/Completed-theses/Master_Diploma-theses/2017/Kuehling/Master-Kuehling.pdf. Accessed 2018-12-15.

[20] Stuart Bennett and Joan Lasenby. Chess - quick and robust detection of chess-board features. *CoRR*, abs/1301.5491, 2013. URL http://arxiv.org/abs/1301.5491.

[21] DSL315 Specification Sheet. URL http://www.optics-online.com/OOL/DSL/DSL315.PDF. Accessed 2018-11-23.

[22] Universal Transverse Mercator coordinate system. URL https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system. Accessed 2019-04-26.

[23] The UTM zones for Europe., . URL https://www.xmswiki.com/wiki/File:Europe.png. Accessed 2019-04-26.

[24] UTM zones of Europe, . URL https://www.xmswiki.com/wiki/UTM_Coordinate_System. Accessed 2019-03-15.

[25] Raivo Aunap. Estonian coordinate system, 2001. URL http://www.geo.ut.ee/~raivo/ESTCOORD.HTML. Accessed 2019-04-24.

Code examples

List of Figures

List of Tables

List of Acronyms

HOG	Histogram of Gradients
IMU	Inertial Measurement Unit
SfM	Structure-from-Motion
OpenCV	Open-Source Computer Vision
WGS84	World Geodetic System 1984
UTM	Universal Transverse Mercator
NGA	National Geospatial-Intelligence Agency
GCP	Ground Control Points
GPS	Global Positioning System
DOP	Dilution Of Precision
UTC	Coordinated Universal Time
L-EST97	LAMBERT-EST of 1997
ROS	Robot Operational System
SLAM	Simultaneous Localization and Mapping
BA	Bundle Adjustment

A. Appendixes

A.1 Appendix A

```

FROM ubuntu:16.04

# Install apt-getable dependencies
RUN apt-get update \
    && apt-get install -y \
        build-essential \
        cmake \
        git \
        libatlas-base-dev \
        libboost-python-dev \
        libeigen3-dev \
        libgoogle-glog-dev \
        libopencv-dev \
        libsuitesparse-dev \
        python-dev \
        python-numpy \
        python-opencv \
        python-pip \
        python-pyexiv2 \
        python-pyproj \
        python-scipy \
        python-yaml \
        wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*


# Install Ceres from source
RUN \
    mkdir -p /source && cd /source && \
    wget http://ceres-solver.org/ceres-solver-1.10.0.tar.gz && \
    tar xvzf ceres-solver-1.10.0.tar.gz && \
    cd /source/ceres-solver-1.10.0 && \
    mkdir -p build && cd build && \
    cmake .. -DCMAKE_C_FLAGS=-fPIC -DCMAKE_CXX_FLAGS=-fPIC -
   DBUILD_EXAMPLES=OFF -DBUILD_TESTING=OFF && \
    make install && \
    cd / && \
    rm -rf /source/ceres-solver-1.10.0 && \
    rm -f /source/ceres-solver-1.10.0.tar.gz


# Install opengv from source
RUN \
    mkdir -p /source && cd /source && \
    git clone https://github.com/paulinus/opengv.git && \
    cd /source/opengv && \
    mkdir -p build && cd build && \
    cmake .. -DBUILD_TESTS=OFF -DBUILD_PYTHON=ON && \
    make install && \
    cd / && \
    rm -rf /source/opengv

#Clone the OpenSfM Repository
RUN git clone https://github.com/mapillary/OpenSfM.git

#Add additional functions that for some reason didn't come with the
   docker file
Run apt-get update \
    && apt-get install python-networkx \
    python-exif \
    python-xmltodict

#Automatically build OpenSfM so that its prebuilt in the docker
Run cd OpenSfM && python setup.py build
```

Code example A.1: Changes applied to the project to allow adding frames without recomputing everything from scratch

```
#! /usr/bin/python
# Copyright (c) 2015, Rethink Robotics, Inc.

# Using this CvBridge Tutorial for converting
```

```python
# ROS images to OpenCV2 images
# http://wiki.ros.org/cv_bridge/Tutorials/
    ConvertingBetweenROSImagesAndOpenCVImagesPython

# Using this OpenCV2 tutorial for saving Images:
# http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/
    py_gui/py_image_display/py_image_display.html

# rospy for the subscriber
import rospy
# ROS Image message
from sensor_msgs.msg import Image
from nmea_msgs.msg import Sentence
# ROS Image message -> OpenCV2 image converter
from cv_bridge import CvBridge, CvBridgeError
# OpenCV2 for saving an image
import cv2
import os
import math

def gngga_to_decdeg(gngga):
    pos_p1 = int(gngga/100)
    pos_p2 = (gngga - pos_p1*100)/60
    return pos_p1 + pos_p2;

# Instantiate CvBridge
bridge = CvBridge()

def image_callback(msg, camera):
    print("Received an image! #" + msg.header.frame_id)
    try:
        # Convert your ROS Image message to OpenCV2
        cv2_img = bridge.imgmsg_to_cv2(msg, "bgr8")
    except CvBridgeError, e:
        print(e)
    else:
        # Save your OpenCV2 image as a jpeg
        stamp = str(msg.header.stamp)
        cv2.imwrite("./images_" + camera + "/" + stamp[:11] + '.jpg',
    cv2_img)

def nmea_callback(msg):
    stamp = str(msg.header.stamp)
    sentence = str(msg.sentence) #$GNGGA,123640.60,5923.66067385,N
    ,02440.20087100,E,4,12,1.7,25.827,M,18.399,M,1.4,2012*60
    if (sentence.startswith('$GNGGA')):
        print("Position received")
        N_f = sentence.index(',') + 1
        N_f = sentence.index(',', N_f) + 1
        N_s = sentence.index(',', N_f)
        N = sentence[N_f:N_s]

        E_f = sentence.index(',', N_s + 1) + 1
        E_s = sentence.index(',', E_f)
        E = sentence[E_f:E_s]

        Z_f = sentence.index(',', E_s) + 1
        Z_f = sentence.index(',', Z_f) + 1
        Z_f = sentence.index(',', Z_f) + 1
        Z_f = sentence.index(',', Z_f) + 1
        Z_f = sentence.index(',', Z_f) + 1
        Z_s = sentence.index(',', Z_f)
        Z = float(sentence[Z_f:Z_s])

        Z_f = sentence.index(',', Z_s+3)

        Z = Z + float(sentence[Z_s+3:Z_f])


        with open('nmea_out_NEZ.txt', 'a') as nmea_kmz, open('nmea_out_tNE.txt', 'a') as nmea:
                nmea.write(stamp + "," + N + "," + E + "\n")
                nmea_kmz.write(str(gngga_to_decdeg(float(E))) + "," + str(gngga_to_decdeg(float(N))) + ","+ str(Z) + "\n")
#       print(stamp + " " + N + " " + E)

def mkdir_for(camera):
    if not os.path.exists('./images_'+camera):
  os.mkdir("./images_"+camera)

def main():
    mkdir_for("left")
    mkdir_for("right")
    mkdir_for("side")

    rospy.init_node('image_listener')
    # Define your image topic
    image_topic = "/pylon_camera_right/image_raw"
    nmea_topic = "/nmea_sentence"
    # Set up your subscriber and define its callback
    rospy.Subscriber("/pylon_camera_right/image_raw", Image,
   image_callback, callback_args = "right")
    rospy.Subscriber("/pylon_camera_left/image_raw", Image,
   image_callback, callback_args = "left")
    rospy.Subscriber("/pylon_camera_side/image_raw", Image,
   image_callback, callback_args = "side")
    rospy.Subscriber(nmea_topic, Sentence, nmea_callback)
# Spin until ctrl + c
    rospy.spin()

if __name__ == '__main__':
    main()
```

Code example A.2: Python subscriber for ROS topics (Pylon camera and GNSS)

```
import os
import math

def gngga_to_decdeg(gngga):
    pos_p1 = int(gngga/100)
    pos_p2 = (gngga - pos_p1*100)/60
    return pos_p1 + pos_p2;

camera = "side"

images = sorted(os.listdir("./images_"+camera))
nmea = open("nmea_out_tNE.txt", "r")
exif_f = open("exif_overrides.json", "w")

nmea_lines = nmea.readlines()
index = 0
nmea_ptr = nmea_lines[index]


exif_f.write("{\n    ")
for image_ptr in images:
  image_s = int(image_ptr[:11])
  nmea_s = int(nmea_ptr[:11])
  print("searching nmea for "+ str(image_s))
  while(True):
    print(str(image_s) +" ? "+str(nmea_s))
    if (image_s == nmea_s):
#     print("gps for " + image_ptr + " exists")
      N_s = nmea_ptr.index(',') + 1
```

```python
      N_e = nmea_ptr.index(',', N_s)
      exif_f.write("\"" + image_ptr + "\": {\n" +
             "        \"gps\": {\n" +
             "            \"latitude\": " + str(gngga_to_decdeg(float(nmea_ptr[N_s:N_e]))) + ",\n" +
             "            \"longitude\": " + str(gngga_to_decdeg(float(nmea_ptr[N_e+1:]))) + ",\n" +
             "            \"altitude\": 29.0\n        }\n    },")
      if (index < len(nmea_lines)-1):
        index += 1
        nmea_ptr = nmea_lines[index]
        nmea_s = int(nmea_ptr[:11])
      break;
    if ((image_s > nmea_s) and (index < len(nmea_lines)-1)):
      print(str(image_s) +" > "+ str(nmea_s))
      index += 1
      nmea_ptr = nmea_lines[index]
      nmea_s = int(nmea_ptr[:11])
      continue;
    if ((image_s < nmea_s) or (index >= len(nmea_lines)-1)):
      print(str(image_s) +" < "+ str(nmea_s))
      print("NO GPS FOR " + image_ptr + "... deleting...")
      os.remove("./images_"+camera+"/" + image_ptr)
      break;

exif_f.seek(-1, os.SEEK_END)
exif_f.truncate()
exif_f.write("\n}")

print("The ./exif_overrides.json file has been updated with ./images_"+camera+"/* and ./nmea_output_tNE.txt")

#image 1555072912946558476.jpg
#nmea  1555072888664709091,5923.66067385,02440.20087100
```

Code example A.3: Python source code for generating an exif_overrides.json, which reference images with the corresponding GPS coordinates

```diff
diff --git a/bin/clean b/bin/clean
index 2d825fb..c545959 100755
--- a/bin/clean
+++ b/bin/clean
@@ -17,3 +17,4 @@ mv -vf $1/features $trash
 mv -vf $1/undistorted* $trash
 mv -vf $1/tracks.csv $trash
 mv -vf $1/reports $trash
+mv -vf $1/track_sets.pkl $trash
diff --git a/opensfm/commands/create_tracks.py b/opensfm/commands/create_tracks.py
index c3e6ecb..949c1e4 100644
--- a/opensfm/commands/create_tracks.py
+++ b/opensfm/commands/create_tracks.py
@@ -19,14 +19,25 @@ class Command:

     def run(self, args):
         data = dataset.DataSet(args.dataset)
+        images = data.images()

         start = timer()
-        features, colors = tracking.load_features(data, data.images())
+        try:
+            graph = data.load_tracks_graph()
+            tracks, processed_images = tracking.tracks_and_images(graph)
+        except IOError:
+            graph = None
+            tracks = None
+            processed_images = []
+
+        remaining_images = set(images) - set(processed_images)
+
+        features, colors = tracking.load_features(data, images)
         features_end = timer()
-        matches = tracking.load_matches(data, data.images())
+        matches = tracking.load_matches(data, remaining_images)
         matches_end = timer()
         graph = tracking.create_tracks_graph(features, colors, matches,
-                                             data.config)
+                data.config, data)
         tracks_end = timer()
         data.save_tracks_graph(graph)
         end = timer()
diff --git a/opensfm/commands/detect_features.py b/opensfm/commands/detect_features.py
index e15a6ee..657e7f3 100644
--- a/opensfm/commands/detect_features.py
+++ b/opensfm/commands/detect_features.py
@@ -54,10 +54,12 @@ def detect(args):
     log.setup()

     image, data = args
-    logger.info('Extracting {} features for image {}'.format(
-        data.feature_type().upper(), image))
+
+

     if not data.feature_index_exists(image):
+        logger.info('Extracting {} features for image {}'.format(
+            data.feature_type().upper(), image))
         start = timer()
         mask = data.load_combined_mask(image)
         if mask is not None:
diff --git a/opensfm/commands/match_features.py b/opensfm/commands/match_features.py
index 8c0615f..1e95e58 100644
--- a/opensfm/commands/match_features.py
+++ b/opensfm/commands/match_features.py
@@ -90,7 +90,7 @@ def has_gps_info(exif):
             'longitude' in exif['gps'])


-def match_candidates_by_distance(images, exifs, reference, max_neighbors, max_distance):
+def match_candidates_by_distance(images, exifs, reference, max_neighbors, max_distance, data):
     """Find candidate matching pairs by GPS distance.

     The GPS altitude is ignored because we want images of the same position
@@ -113,6 +113,8 @@ def match_candidates_by_distance(images, exifs, reference, max_neighbors, max_di

     pairs = set()
     for i, image in enumerate(images):
+        if data.matches_exists(image):
+            continue
         distances, neighbors = tree.query(
             points[i], k=k, distance_upper_bound=max_distance)
         for j in neighbors:
@@ -121,7 +123,7 @@ def match_candidates_by_distance(images, exifs, reference, max_neighbors, max_di
     return pairs

```

```

-def match_candidates_by_time(images, exifs, max_neighbors):
+def match_candidates_by_time(images, exifs, max_neighbors, data):
     """Find candidate matching pairs by time difference."""
     if max_neighbors <= 0:
         return set()
@@ -135,6 +137,8 @@ def match_candidates_by_time(images, exifs,
    max_neighbors):

     pairs = set()
     for i, image in enumerate(images):
+        if data.matches_exists(image):
+            continue
         distances, neighbors = tree.query(times[i], k=k)
         for j in neighbors:
             if i != j and j < len(images):
@@ -142,7 +146,7 @@ def match_candidates_by_time(images, exifs,
    max_neighbors):
     return pairs


-def match_candidates_by_order(images, max_neighbors):
+def match_candidates_by_order(images, max_neighbors, data):
     """Find candidate matching pairs by sequence order."""
     if max_neighbors <= 0:
         return set()
@@ -150,6 +154,8 @@ def match_candidates_by_order(images, max_neighbors
    ):

     pairs = set()
     for i, image in enumerate(images):
+        if data.matches_exists(image):
+            continue
         a = max(0, i - n)
         b = min(len(images), i + n)
         for j in range(a, b):
@@ -186,14 +192,18 @@ def match_candidates_from_metadata(images, exifs,
     data):
         pairs = combinations(images, 2)
     else:
         d = match_candidates_by_distance(images, exifs, reference,
-                                         gps_neighbors, max_distance)
-        t = match_candidates_by_time(images, exifs, time_neighbors)
-        o = match_candidates_by_order(images, order_neighbors)
+                                         gps_neighbors, max_distance,
    data)
+        t = match_candidates_by_time(images, exifs, time_neighbors,
    data)
+        o = match_candidates_by_order(images, order_neighbors, data)
         pairs = d | t | o

     res = {im: [] for im in images}
     for im1, im2 in pairs:
-        res[im1].append(im2)
+        if not data.matches_exists(im1):
+            res[im1].append(im2)
+        else:
+            #assert not data.matches_exists(im2)
+            res[im2].append(im1)

     report = {
         "num_pairs_distance": len(d),
@@ -213,6 +223,9 @@ def match(args):
     log.setup()

     im1, candidates, i, n, ctx = args
+    if ctx.data.matches_exists(im1):
+        #assert(len(candidates) == 0)
+        return
     logger.info('Matching {}  -  {} / {}'.format(im1, i + 1, n))

     config = ctx.data.config
diff --git a/opensfm/commands/mesh.py b/opensfm/commands/mesh.py
index 7cae57a..c55314b 100644
--- a/opensfm/commands/mesh.py
+++ b/opensfm/commands/mesh.py
@@ -21,6 +21,8 @@ class Command:
         graph = data.load_tracks_graph()
         reconstructions = data.load_reconstruction()

+        logger.debug("Starting calculation of reconstructed mesh")
+
         for i, r in enumerate(reconstructions):
             for shot in r.shots.values():
                 if shot.id in graph:
diff --git a/opensfm/dataset.py b/opensfm/dataset.py
index 52aaeca..4272ee0 100644
--- a/opensfm/dataset.py
+++ b/opensfm/dataset.py
@@ -485,6 +485,19 @@ class DataSet(object):
```

```
                    return im2_matches[im1][:, [1, 0]]
        return []

+    def __track_sets_file(self, filename=None):
+        """Return path of unionfind file"""
+        return os.path.join(self.data_path, filename or 'track_sets.pkl')
+
+    def load_track_sets_file(self, filename=None):
+        """Return unionfind of tracks"""
+        with open(self.__track_sets_file(filename), 'rb') as fin:
+            return load_track_sets_file(fin)
+
+    def save_track_sets_file(self, unionfind, filename=None):
+        with open(self.__track_sets_file(filename), 'wb') as fout:
+            save_track_sets_file(fout, unionfind)
+
     def _tracks_graph_file(self, filename=None):
         """Return path of tracks file"""
         return os.path.join(self.data_path, filename or 'tracks.csv')
@@ -712,3 +725,9 @@ def save_tracks_graph(fileobj, graph):
                 r, g, b = data['feature_color']
                 fileobj.write(u'%s\t%s\t%d\t%g\t%g\t%g\t%g\t%g\n' % (
                     str(image), str(track), fid, x, y, r, g, b))
+
+def load_track_sets_file(fileobj):
+    return pickle.load(fileobj)
+
+def save_track_sets_file(fileobj, unionfind):
+    pickle.dump(unionfind, fileobj, protocol=pickle.HIGHEST_PROTOCOL)
diff --git a/opensfm/reconstruction.py b/opensfm/reconstruction.py
index a599be7..af6def4 100644
--- a/opensfm/reconstruction.py
+++ b/opensfm/reconstruction.py
@@ -1152,12 +1152,33 @@ def incremental_reconstruction(data, graph):
     if not data.reference_lla_exists():
         data.invent_reference_lla(images)

-    remaining_images = set(images)
+    try:
+        existing_reconstructions = data.load_reconstruction()
+        for reconstruction in existing_reconstructions:
+            reconstruction.points = {k: point for k, point in reconstruction.points.items() if k in graph}
+    except IOError:
```

```
+        existing_reconstructions = []
+
+    reconstructed_images = set(image for reconstruction in existing_reconstructions for image in reconstruction.shots.keys())
+
+    remaining_images = set(images) - reconstructed_images
     gcp = None
     if data.ground_control_points_exist():
         gcp = data.load_ground_control_points()
-    common_tracks = tracking.all_common_tracks(graph, tracks)
+
     reconstructions = []
+    for reconstruction in existing_reconstructions:
+        grow_reconstruction(data, graph, reconstruction, remaining_images, gcp)
+        reconstructions.append(reconstruction)
+        reconstructions = sorted(reconstructions,
+                                    key=lambda x: -len(x.shots))
+        data.save_reconstruction(reconstructions)
+
+    if len(reconstructed_images) != 0:
+        common_tracks = tracking.all_common_tracks(graph, tracks, remaining_images=remaining_images)
+    else:
+        common_tracks = tracking.all_common_tracks(graph, tracks)
+
     pairs = compute_image_pairs(common_tracks, data)
     chrono.lap('compute_image_pairs')
     report['num_candidate_image_pairs'] = len(pairs)
diff --git a/opensfm/tracking.py b/opensfm/tracking.py
index 0466545..2b0085a 100644
--- a/opensfm/tracking.py
+++ b/opensfm/tracking.py
@@ -35,10 +35,16 @@ def load_matches(dataset, images):
     return matches


-def create_tracks_graph(features, colors, matches, config):
+def create_tracks_graph(features, colors, matches, config, data):
     """Link matches into tracks."""
     logger.debug('Merging features onto tracks')
-    uf = UnionFind()
+    try:
+        uf, track_ids, max_id = data.load_track_sets_file()
+    except (IOError, EOFError) as e:
```

```
+        uf = UnionFind()
+        track_ids = {}
+        max_id = 0
+
     for im1, im2 in matches:
         for f1, f2 in matches[im1, im2]:
             uf.union((im1, f1), (im2, f2))
@@ -50,13 +56,19 @@ def create_tracks_graph(features, colors, matches, config):
             sets[p].append(i)
         else:
             sets[p] = [i]
+            if p not in track_ids:
+                track_ids[p] = max_id
+                max_id += 1
+
+   track_sets = (uf, track_ids, max_id)
+   data.save_track_sets_file(track_sets)

     min_length = config['min_track_length']
-    tracks = [t for t in sets.values() if _good_track(t, min_length)]
+    tracks = [(track_ids[track_name], t) for track_name, t in sets.items() if _good_track(t, min_length)]
     logger.debug('Good tracks: {}'.format(len(tracks)))

     tracks_graph = nx.Graph()
-    for track_id, track in enumerate(tracks):
+    for track_id, track in tracks:
         for image, featureid in track:
             if image not in features:
                 continue
@@ -107,7 +119,7 @@ def common_tracks(graph, im1, im2):
     return tracks, p1, p2


-def all_common_tracks(graph, tracks, include_features=True, min_common=50):
+def all_common_tracks(graph, tracks, include_features=True, min_common=50, remaining_images=None):
     """List of tracks observed by each image pair.

     Args:
@@ -116,11 +128,19 @@ def all_common_tracks(graph, tracks, include_features=True, min_common=50):
         include_features: whether to include the features from the
```

```
images
        min_common: the minimum number of tracks the two images need to have
            in common
+        param remaining_images: if not none, only find pairs from within this list

    Returns:
        tuple: im1, im2 -> tuple: tracks, features from first image, features
        from second image
    """
+
+    if remaining_images is not None:
+        # We just look at the subgraph comprising of remaining images, and tracks that pass through them
+        tracks = {track for imagename in remaining_images for track in graph[imagename]}
+        filtered_nodes = set(remaining_images).union(tracks)
+        graph = graph.subgraph(filtered_nodes)
+
    track_dict = defaultdict(list)
    for track in tracks:
        track_images = sorted(graph[track].keys())
```

Code example A.4: Changes applied to the OpenSfM project to allow a dynamic addition of new images