Technical Documentation

GoPro Trails

Images and Graphics, Project Course CDIO

TSBB11

Group Members

Daniel Cranston, dancr948

Carl Ekman, carek025

Lisa Eriksson, liser858

Freja Fagerblom, frefa105

Filip Skarfelt, filsk543

Version 1.0

December 19, 2018

Abstract

Construction of 3D models with correct scale from video is useful for creating a realistic scene of the recorded surroundings. This together with a generated trajectory can be used to track movement of people wearing GoPro cameras and simultaneously build a 3D map of the environment.

This project was about reconstructing 3D models with correct scale along with a georeferenced camera trajectory from a GoPro video and embedded IMU and GPS data. This is a difficult problem due to the fact that the GoPro camera has a wide-angle lens and a rolling shutter, meaning that rows in the image are captured at different points in time.

The strategy used in the project was to first track keypoints. These keypoint tracks were then used to perform rolling shutter aware visual inertial structure from motion. This results in a camera trajectory that can be georeferenced using GPS data and 3D points, which can be used to perform surface reconstruction.

Overall, the produced software *GoPro Trails* works well. However, some recordings are more sensitive to different parameter values. In general, the generated trajectory can be seen as a refinement of the GPS data in terms of resolution and the 3D reconstruction is generally satisfactory.

Contents

1 Introduction **4**

2 System Description **4**
2.1 Components . . . 5
2.1.1 Camera Calibration . . . 5
2.1.2 GoPro Data Extraction . . . 6
2.1.3 Tracking . . . 6
2.1.4 Kontiki Trajectory and 3D Point Estimation . . . 7
2.1.5 Trajectory GPS Fitting . . . 10
2.1.6 Trajectory Map Visualization . . . 11
2.1.7 Create Dense Correspondences . . . 12
2.1.8 Triangulation and Outlier Removal . . . 14
2.1.9 Point Cloud Processing and Visualization . . . 14
2.2 IMU Calibration . . . 16
2.3 Configuration . . . 18

3 User Interface **18**

4 Results **19**
4.1 Stone . . . 19
4.2 Benches . . . 20
4.3 Bridge . . . 21
4.4 Barn . . . 22
4.5 Small House . . . 23
4.6 Road . . . 24

5 Known Problems **26**

6 Future Work **26**

7 Conclusions **27**

References **28**

A Requirement Specification **29**

B User Manual **32**

1 Introduction

The software GoPro Trails turns GoPro videos into 3D models using structure from motion (SFM) techniques and produces trajectories shown in an interactive map service using georeferencing. This document explains the different components and the code structure of GoPro Trails. The requirements for the program are specified in Appendix A.

The aim of the project was to create a georeferenced camera trajectory and a 3D model of the scene, given a GoPro video with embedded IMU and GPS data. The IMU data consists of accelerometer data in the form of a three dimensional acceleration vector for every sample. It also consists of gyroscope data in the form of a three dimensional angular velocity vector. Every sample in the GPS data consists of longitude and latitude. The main problem statement was whether the Kontiki library [1] can be used to reconstruct 3D models for different kinds of video sequences successfully, despite rolling shutter effects. Another problem statement was whether the generated trajectory can be used as a refinement of the trajectory described by GPS data.

2 System Description

The system generates a 3D model and a georeferenced trajectory given a GoPro video and embedded IMU and GPS data. Camera calibration and data processing that generates keypoint tracks and prepares data for the SFM module build up the system pipeline. The generated trajectory is georeferenced and visualized in a map while the point cloud is densified and visualized in *Meshlab* [2] after surface reconstruction. Figure 1 shows the components of the system.

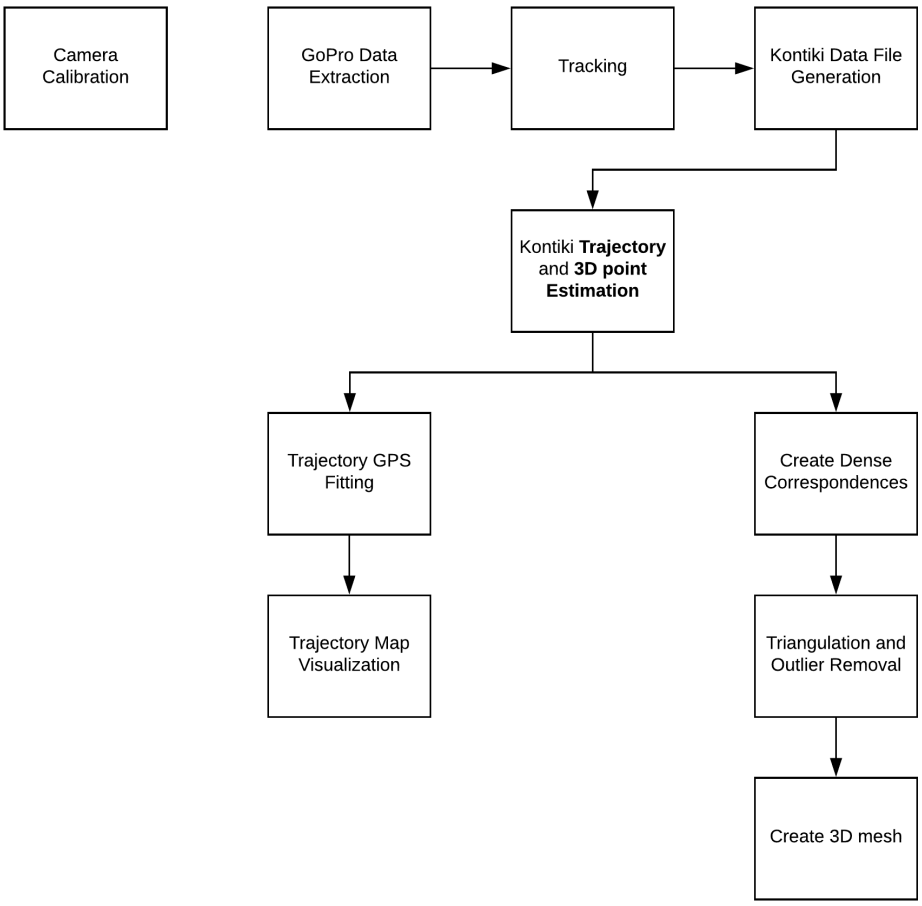


Figure 1: Flowchart of the system components.

The *Camera Calibration* component handles the calibration of the camera which is performed once before all the other steps. Extraction of IMU and GPS data from the GoPro camera is handled by the *GoPro Data Extraction* module. The *Tracking* component creates tracks using the software *TrackRetrack* [3]. Desired input format to Kontiki is ascertained by the *Kontiki Data File Generation* module. The *Kontiki Trajectory and 3D Point Estimation* component performs the optimization process run by the Kontiki software. This results in a trajectory and 3D points. Visualization of the trajectory is handled by the modules *Trajectory GPS Fitting* and *Trajectory Map Visualization*. The first, *Trajectory GPS Fitting*, projects the 3D trajectory points and georeferences the points using the GPS points. At the end in this branch is the component *Trajectory Map Visualization*. It visualizes the created trajectory together with the GPS points in *OpenStreetMap* [4]. The other branch handles visualization of the 3D points. The *Create Dense Correspondences* component uses the *Patchmatch* [5] algorithm to find a dense set of putative correspondences. These correspondences are made into 3D points in the *Triangulation and Outlier Removal* component, which also removes any outliers to improve the point cloud. At the end of this branch is the module *Create 3D Mesh*. It uses the 3D points to create a mesh which is visualized in *Meshlab* [2]. All these components are further explained in Section 2.1.

The directory hierarchy for the project is shown in Figure 2.

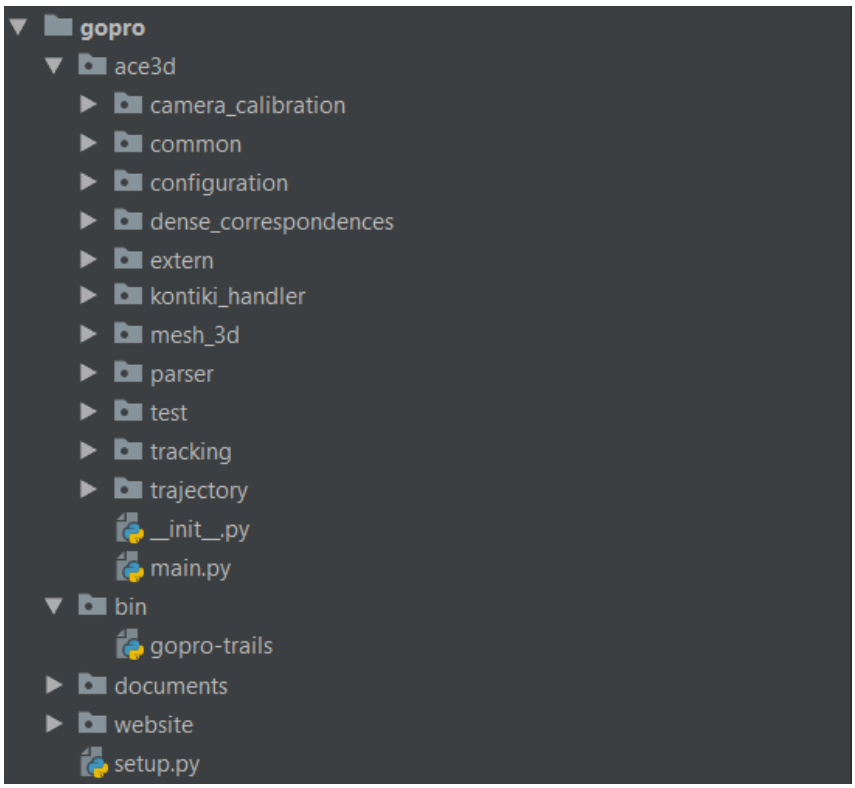


Figure 2: Directory hierarchy of the project.

2.1 Components

All components within the product are described in this section.

2.1.1 Camera Calibration

This section will explain how camera calibration is handled in the program. Project module `ace3d/camera_calibration`, see Figure 2, contains the code regarding the camera calibration.

The camera calibration is performed separately from the rest of the program, before the program is run. Ovrén has created the project `atan_calib` [3], which was used to perform the calibration. This calibration program is available in the project as `ace3d/extern/atan_calib.py`.

The library uses an arctan camera model, which is suitable for a camera such as the GoPro with wide angle characteristics [1]. Two parameters are used in the model, distortion parameter γ and distortion center $\boldsymbol{\omega}_c = [\omega_x \ \omega_y]^T$. The model transforms normalized coordinates $\mathbf{y}_n$ to distorted coordinates $\mathbf{y}_d$ according to

$$\mathbf{y}_d = \boldsymbol{\omega}_c + \frac{\arctan(r\gamma)}{\gamma} \frac{(\mathbf{y}_n - \boldsymbol{\omega}_c)}{r} \tag{1}$$

where $r = ||\mathbf{y}_n - \boldsymbol{\omega}_c||$. The inverse distortion model is

$$\mathbf{y}_n = \boldsymbol{\omega}_c + \frac{\tan(r\gamma)}{\gamma} \frac{(\mathbf{y}_d - \boldsymbol{\omega}_c)}{r} \tag{2}$$

where r in this case is $||\mathbf{y}_d - \boldsymbol{\omega}_c||$. [1]

The calibration is performed by taking multiple photos of a calibration pattern with known geometry and distinct corner points. Actual pixel coordinates of the corners are compared to predictions of the arctan camera model for the current parameters. The parameters are then optimized using non-linear optimization to minimize the residuals for the calibration photos.

A calibration HDF file containing these optimized parameters is the result from the calibration process. This file is then augmented with the additional camera parameters height, width, readout time and fps. The files are stored in the directory `ace3d/camera_calibration`. When GoPro Trails is run the user can select one of the built-in calibration files or supply a custom file. The parameters are then loaded from the file by the `camera_calibration` module into a camera object.

2.1.2 GoPro Data Extraction

This section explains the GoPro data extraction component (see Figure 1). The component is implemented by the project module `ace3d/parser` (see Figure 2).

The data extraction is implemented using the `gpmfstream` project created by Ovrén [3]. Timestamps for the IMU data are generated using a calibrated (assumed constant) sampling frequency and a time offset relative to the image stream. Section 2.2 describes the calibration process. The GPS data is not evenly sampled, so the timestamps are computed by spreading the samples evenly over the time interval corresponding to the data package in which the points appears.

2.1.3 Tracking

The code for the tracking phase is placed in `ace3d/tracking`, see Figure 2. Points to track are found and tracked using the software `trackretrack` written by Ovrén [3]. This library uses the FAST corner detection algorithm [6] to detect new keypoints. The keypoints are then tracked frame by frame using a pyramid implementation of the Lucas Kanade feature tracker [7]. This tracker uses a coarse-to-fine pyramid approach. The feature tracks are regularly verified by tracking the points in reverse and checking that the reverse tracking is consistent with the forward tracking. Inconsistent tracks are discarded. New keypoints are detected whenever the number of active tracks drops below a threshold. At the end, tracks that span a number of frames that is less than a threshold are discarded. An example of detected keypoints can be seen in Figure 3.

Figure 3: Example of detected keypoints during tracking.

The color for a track is defined as the color of the first observation of that specific track. All tracks are stored in a HDF file and each track contains the following fields

- frames
- points
- color
- start
- backtrack length
- min distance
- min points
- min length

2.1.4 Kontiki Trajectory and 3D Point Estimation

This section will explain the parts of the project that are directly connected to Kontiki. Kontiki, or the Continuous-Time Toolkit, is a toolkit for optimization-based, continuous-time, structure from motion. In short, it can estimate a camera trajectory and 3D structure from a set of measurements and it does so using a continuous-time trajectory [8]. The files contained in `ace3d/kontiki_handler` are relevant for this section. A few things need to be prepared in order to utilize the Kontiki pipeline:

- Tracks in 2D images corresponding to 3D points
- A color for each track
- Information about which frames each track exists in
- Parameters for the arctan camera model
- IMU data
- A true IMU rate and time offset from the camera
- A plethora of parameters that will effect the final output

The file `ace3d/kontiki_handler/kontiki_classes.py` contains attributes and methods directly relating to what Kontiki needs to run. The pipeline consists of an initialization and four optimization phases.

First of all a camera object from the camera calibration is loaded into the object. Then a Kontiki IMU-object is initialized. In this stage the true rate of the IMU is provided, along with the accelerometer bias, if any.

In the next stage the SFM-data is initialized. The tracks from the tracking stage are loaded and made into landmarks (3D points with several observations at different times) and views are constructed from data explaining the life time of the tracks. Due to reasons explained in the IMU Section (2.2), a culling of views has to be performed in order to have accompanying IMU data for all views. A landmark in Kontiki does not explicitly hold the cartesian coordinates for the 3D points, but rather holds the image coordinates in a reference frame, together with an inverse depth. The 3D coordinates can be obtained by unprojecting using the inverse depth. The selection of the reference observation can be set to always choose the first observation of the landmark, or simply choose a valid observation at random. An observation holds information about which landmark it observes, in which view and what the image coordinates are in that view.

In the final initialization stage the trajectory is initialized. The positions and orientations are modelled with R3-splines and SO3-splines respectively. Together they form a time continuous split trajectory. In order to find a suitable knot distance for the splines, a quality measure [9] for each spline is used as a parameter. In order to be able to use the residuals from the IMU data compared to the trajectory, combined with the image residuals, a weighting needs to be done as well. This process is called Spline Error Weighting and was developed by Ovrén and Forssén and the details are described in their paper [9].

In general, a lot of tracks will have been generated in a large amount of frames (views). This gives rise to a huge amount of residuals to be included in the optimization problem. Therefore the next stage is to choose keyframes such that a certain ratio of common tracks are retained between the keyframes. Adaptive non-maxima suppression using disk coverage is a method of choosing tracks in such a way that they are evenly distributed in the image [10]. This method is used to simplify the optimization problem by reducing the problem size. The image residuals corresponding to the views and landmarks selected from these algorithms are stored and used in the estimation phase. In order to reduce the impact of any inevitable outliers, the residuals are calculated with a Huber norm, which acts quadraticaly up to a certain threshold, after which it acts linearly.

An estimator object is created, containing all the relevant data for the optimization process. It holds all the residuals and the Kontiki solver itself, as well as methods for visualizing the residuals and 3D points in different stages of the pipeline. Residuals for views not currently in the optimization problem are also used to visualize how well the result generalizes. In the first phase the residuals discussed above are optimized for a set amount of iterations. The solver minimizes the sum of the residuals which is formulated in the following way;

$$\begin{aligned} J(\mathcal{T},\mathcal{X}) = & \sum_{\mathbf{y}_{k,n}\in\mathcal{Y}} \underbrace{\left\| \mathbf{y}_{k,n} - \pi(\mathbf{T}^{-1}(t_{k,n})\mathbf{x}_k) \right\|^2 \mathbf{w}_v}_{\text{Visual}} \\ + & \sum_m \underbrace{\left\| \omega_m - \nabla_\omega \mathbf{T}(t_m) \right\|^2 \mathbf{w}_g}_{\text{Gyroscope}} + \sum_l \underbrace{\left\| \mathbf{a}_l - \nabla_a^2 \mathbf{T}(t_l) \right\|^2 \mathbf{w}_a}_{\text{Accelerometer}} \end{aligned} \tag{3}$$

where $\mathcal{T}$ is the time continuous trajectory transformation function, the function that transforms a 3D point in camera coordinates to world coordinates, $\mathcal{X}$ is the set of landmarks and $\mathcal{Y}$ is the set of observations. The projection function is denoted as π. This is explained in further detail in the PhD dissertation by Ovrén [1].

In the next two phases outlier removal is performed. This is done by locking the trajectory and adding all landmarks in the current keyframes to the problem and thus calculate their residuals. Landmarks with a mean residual above a certain threshold are removed from the optimization problem. The new set of residuals are optimized for a set amount of iterations. This process is repeated in the next phase, but the threshold for removing outliers is more strict.

In the final phase, all views are added to the optimization problem for the selected landmarks. This stage is considerably slower due to the larger problem size, but due to the initial solution provided by the previous steps, a high amount of iterations should be avoidable.

Figure 4: A single frame of a barn from a used sequence.

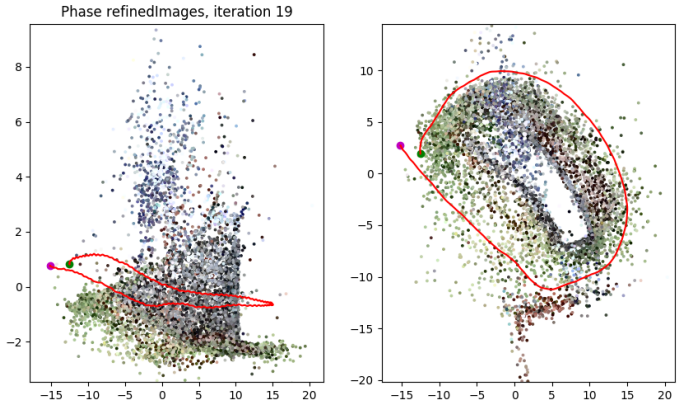


Figure 5: Estimated trajectory and 3D points from two different angles, from mid-optimization.

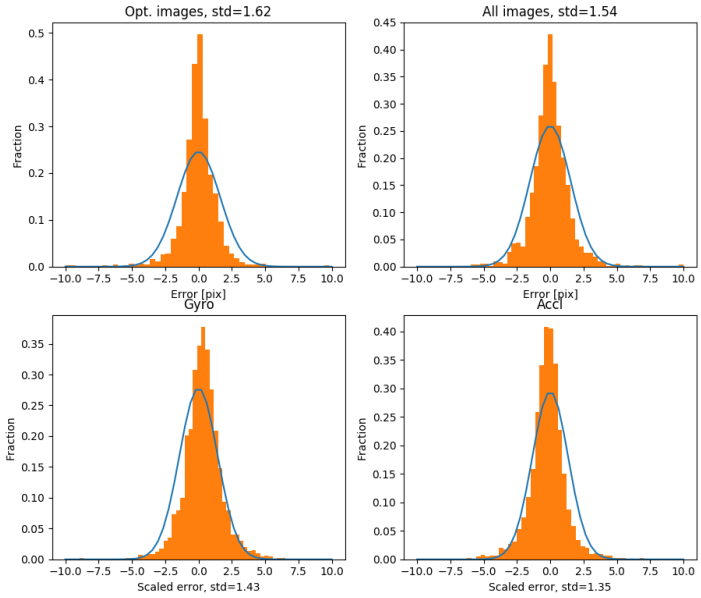


Figure 6: Mid-optimization residuals. The blue line overlays a Gaussian with the standard deviation calculated from the residuals in the plot. Top left: image residuals of the views currently optimized. Top right: image residuals of all views, including those not yet added to the optimization. Bottom left: residuals of the gyroscope measurements. Bottom right: residuals of the accelerometer measurements.

Figure 5 and 6 shows the optimization results printed out during the Kontiki pipeline after the second phase, i.e. the first of the outlier removal phases. This output is mostly used to make sure that a promising result is being outputted and to allow the user to halt the optimization early and change parameters if the results seem poor. In this example, the loading ramp of a barn was filmed (Figure 4) and 310 keyframes were used out of a total of 1649 views.

In Figure 5 it is apparent that the 3D points form some structure (you can clearly make out the ramp) and the trajectory goes in a loop, which in this case is an indicator of a promising result, since the object was filmed while walking one lap around it. Figure 6, which displays the residuals, shows that there are no outliers for the measurements currently included in the optimization problem and the residuals are roughly normally distributed. In this case, the same can be said for the residuals in the top right plot, which includes residuals from views not yet included in the optimization problem. This means that the currently optimized subproblem generalizes very well to the other views.

This result will be improved further in the following phases and complete examples from different sequences can be found in the result section (Section 4).

2.1.5 Trajectory GPS Fitting

This section will explain the trajectory GPS fitting component (see Figure 1). The component is implemented by the project module `ace3d/trajectory` (see Figure 2).

As input this component takes the camera GPS points, corresponding time stamps and the previously generated Kontiki trajectory object. The component outputs coordinates (longitude latitude) generated from the trajectory and georeferenced using the GPS points.

Implementation

1. Project GPS points to the Universal Transverse Mercator coordinate system (UTM) with the WGS84 ellipsoid model [11], using the python module `UTM` [12].

2. Remove GPS outliers:

(a) Remove duplicated points at the beginning and the end.

(b) For each point, compute the median x and y coordinates for points in a configurable time interval around the current point and remove it if the distance to the median is larger than a configurable threshold.

3. Sample Kontiki trajectory at time points corresponding to the GPS timestamps.

4. Project the 3D trajectory points into the ground plane. The world coordinate system defined by the SFM component is assumed to have one coordinate axis perpendicular to the ground plane (vertical axis) and the remaining two axes parallel to the ground plane. This means that the projection is performed by simply discarding the coordinate corresponding to the vertical axis.

5. A rigid transformation $(\mathbf{R}(\alpha), \mathbf{t})$ is estimated (scale is assumed to already be correct) by minimizing

$$\varepsilon(\alpha, \mathbf{t}) = \sum_i ||\mathbf{R}(\alpha)\mathbf{y}_{\text{traj},i} + \mathbf{t} - \mathbf{y}_{\text{gps},i}||^2 \tag{4}$$

over the angle α and the 2D translation $\mathbf{t}$. The points involved are the UTM GPS coordinates and the projected trajectory points. The error ε is minimized using the `scipy.optimize.least_squares` python module [13]. Since the UTM coordinates are represented relative to the origin of the current UTM zone, the translation required can be quite large. To help the optimization process the translation is initialized to the mean value of the GPS UTM points.

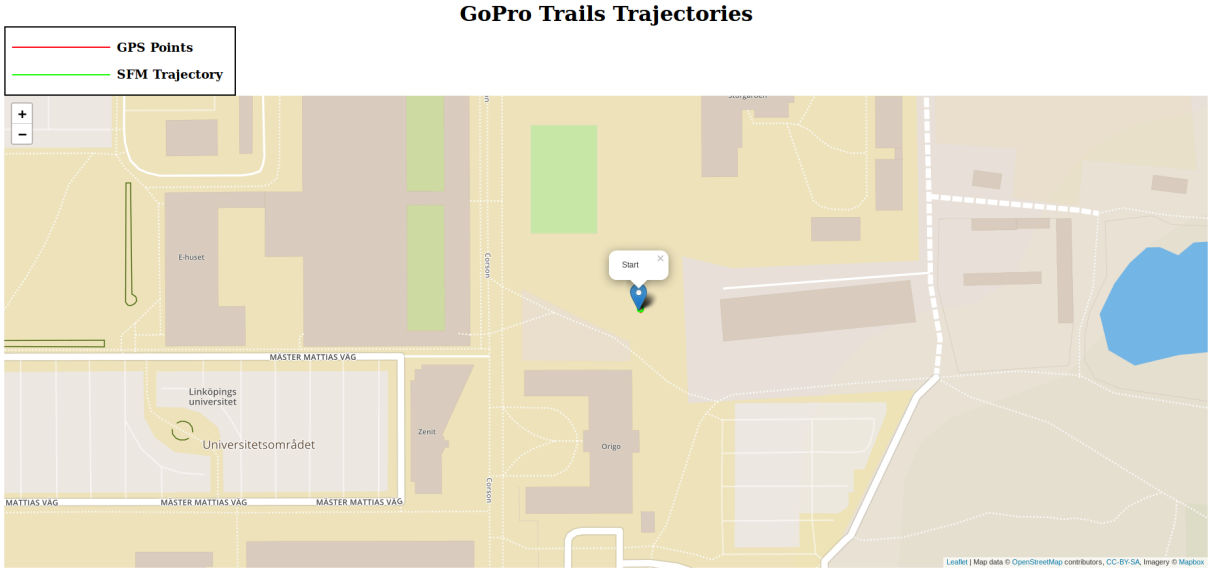
6. The Kontiki trajectory is sampled more densely using a configurable sampling frequency.

7. The estimated transformation is applied to the densely sampled points.

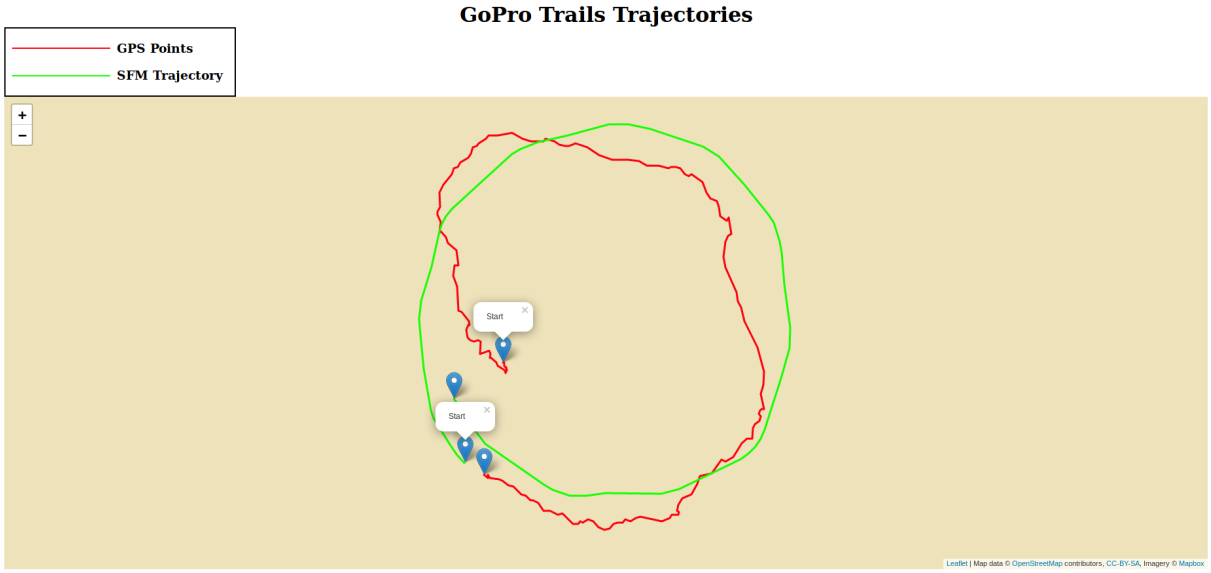
8. The points are then finally transformed to longitude latitude coordinates by inverting the UTM projection.

2.1.6 Trajectory Map Visualization

This section will explain the trajectory visualization in a map service. The code regarding this part is found in `ace3d/trajectory` (see Figure 2). The file `ace3d/trajectory/visualize.py` contains a function for saving both the created SFM trajectory and the GPS coordinates in a json file. This file can then be used in the function `show_trajectories`, located in the same file, which opens the default web-browser and displays the trajectories in *OpenStreetMap* [4]. An example trajectory visualization can be seen in Figure 7. More information and code regarding the web-application can be found in the files `ace3d/trajectory/trajectory_map.css`, `ace3d/trajectory/trajectory_map.html` and `ace3d/trajectory/trajectory_map.js`. An example of the trajectory visualization is shown in Figure 7.



(a) The trajectories' positions on the map.



(b) Zoomed variant showing the difference between the trajectory constructed by GPS points and the SFM trajectory.

Figure 7: Example of trajectory visualization.

2.1.7 Create Dense Correspondences

This section explains how dense correspondences are computed and used to create a better point cloud. The directory `ace3d/dense_correspondences` contains the relevant files for this section.

These steps are performed after Kontiki has created a stable trajectory. The Kontiki object from `ace3d/kontiki_handler` will then contain not only the trajectory itself, but also where each frame (view) exists along the trajectory. This means that frames (image data) can be related to poses in the constructed 3D world. Given two frames, their respective poses can be queried from the trajectory. Using these poses and corresponding 2D points, new 3D points can be triangulated.

Finding new correspondences in pairs of frames becomes the first problem. To solve this, the *PatchMatch* [5] algorithm is used. The algorithm is implemented in C++ and *Pybind11* [14] is used to expose the algorithm to Python. It is composed of a random search step and a propagation step.

PatchMatch takes a pair of images and produces a Nearest Neighbour Field (NNF) that maps each pixel in the first image to a pixel in the second, by comparing the local area (image patch) around the

pixels. Initially the NNF is randomly initialized, meaning that each pixel is mapped to a pixel in the second image at random. In the following explanation, a pixel in the first image is called a 'target pixel', and a pixel in the second image is called a 'neighbour candidate pixel' or simply 'candidate pixel'.

For each pixel in the first image (target pixel) a random search is performed over pixels in the second image (neighbour candidates). The goal is to find a candidate pixel whose local area (patch) resembles that of the target pixel. A 'Distance' or 'Likeness' between these patches is calculated in accordance with (5).

$$D = \sum_{\text{Pixels in Patch}} (dr^2 + dg^2 + db^2) \tag{5}$$

Where D is the Distance measure, and dr, dg and db are the differences in intensity of the red, green and blue channels respectively. In short, the difference in intensity for each pixel in the patch is compared and summed together. The smaller D is, the more 'alike' the two patches are. A target pixels match is chosen as the candidate pixel which has the lowest Distance measure. As this is computed for each new candidate pixel iteratively, there is a need to remember the smallest distance (best match) found so far. Therefore, a Distance Mapping is used to keep track of this. See Figure 8 (middle).

Another important part of PatchMatch is its propagation step. The naive random search step is not guaranteed to find good matches for every target pixel. The propagation step is designed to remedy this, by propagating good matches to adjacent target pixels. There are two observations that illustrates why this is a good idea:

1. At least some target pixels are highly likely to have found very good matches.
2. If a target pixel has found a good match, it is likely that adjacent target pixels will also find good matches by looking at the same area.

A more detailed explanation of PatchMatch can be found in the related paper. [5]

For an image pair, *PatchMatch* is performed "both ways", followed by Left/Right Consistency Checking to extract only the matches that the two *PatchMatch* executions agree with. The result is a dense set of *putative* correspondences between the images. See Figure 8 (right).

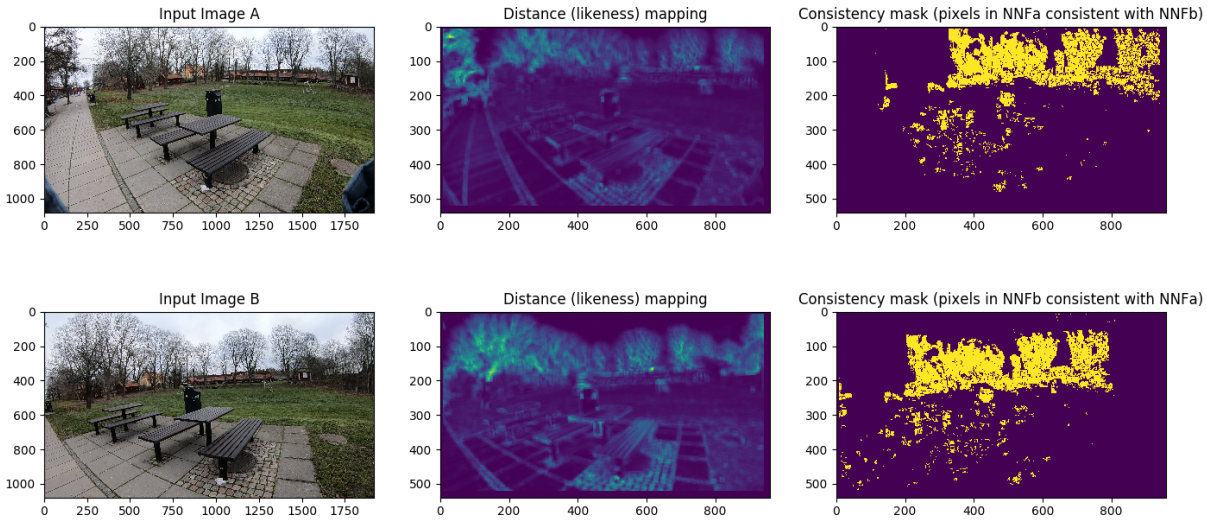


Figure 8: Example output from PatchMatch and Left/Right Consistency Checking. Left: input image pair. Middle: resulting Distance mapping after PatchMatch. Right: resulting putative correspondences after Left/Right Consistency Checking.

By repeating the above steps for several sets of image pairs along the video sequence, point correspondences (and subsequently new landmarks) will be created from many different viewpoints. The current method of choosing image pairs is based on trajectory positions and two parameters (*keyframe_stride* and *baseline_dist*) and can be described as follows:

1. Start at the first frame in the video sequence.
2. Step forward *keyframe_stride* frames.
   (a) Choose this as the first frame of the pair.
3. Step forward one frame.
   (a) Compare trajectory positions for this frame and the one chosen in Step 2.
   (b) If the relative translation is above *baseline_dist* meters: choose this as the second frame of the pair.
   (c) Else: Repeat this step and try again.
4. Repeat from Step 2 until the end of the video.

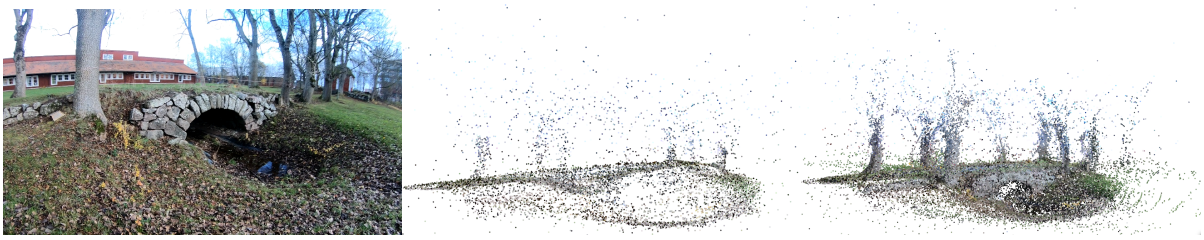
The parameters *keyframe_stride* and *baseline_dist* have default values of 60 and 0.5 respectively, but can also be set in the configuration file. From a triangulation point of view, a large baseline between image pairs is desired. However, since PatchMatch works best with little relative translation and rotation between the image pairs, a compromise between the two has to be made. The above method was chosen for this exact reason.

2.1.8 Triangulation and Outlier Removal

Once *PatchMatch* has produced a set of putative correspondences from a pair of images, these have to be triangulated and subsequently tested to assure that they are not outliers. This is done by making use of the Kontiki trajectory created earlier in the pipeline.

For each pair of corresponding points, a new landmark is created and the points are added as observations. Similarly to Section 2.1.4, a new trajectory estimator object is created and the observations are added to the estimator. In other words, only terms marked by "Visual" in equation (3) in Section 2.1.4 are included. By locking the trajectory and letting the trajectory estimator solve the estimation problem, the landmarks are effectively triangulated. If the resulting mean residual error of a landmark exceeds a certain threshold (default is 0.5 pixels but can be specified in the configuration file), it is assumed to be an outlier and is removed.

Finally, the new landmarks are added to the list of landmarks created from the sparse Kontiki SFM. A comparison of point clouds before and after densification can be seen in Figure 9.



(a) View of the scene. (b) Sparse point cloud from Kontiki (c) Result after densification.

Figure 9: Comparison of point clouds before and after dense triangulation

2.1.9 Point Cloud Processing and Visualization

In this section subjects relating to processing and visualization of the point cloud will be explained. The significant functions for handling point clouds and creating a mesh that are mentioned in this section are located in `ace3d/mesh_3d`, seen in the directory hierarchy in Figure 2. It is divided into `pointcloud_handler` and `meshlab_handler`. The first handles processing and improvements of the point cloud data, while the second handles import and export of the point cloud data in connection with the processing as well as managing tasks related to Meshlab [2].

The point coordinates in 3D and its color information are extracted from the Kontiki Object after dense correspondences have been calculated. An example of how the point cloud might appear is visualized in Figure 10 by applying a draw geometries function from the open source library Open3D [15].

(a) Reference image (b) Input point cloud

Figure 10: Input pointcloud and reference image from video.

Afterwards, point cloud processing begins with down-sampling using Open3D and a regular voxel grid to create a uniformly down-sampled point cloud. This will mainly affect dense areas and leave sparse areas unchanged, but is necessary since PatchMatch creates a large amount of points in the attempt to densify the point cloud that can be considered duplicates. This is followed by statistical outlier removal. The point cloud in this stage tend to include several misplaced points. This step therefore removes points with an average distance to neighbors higher than a threshold based on the standard deviation of the average distances of the point cloud. The default value of the threshold is by default set very low, because points far away from the camera tend to increase the average distance. Figure 11 demonstrates sparse points that will be removed in red color.

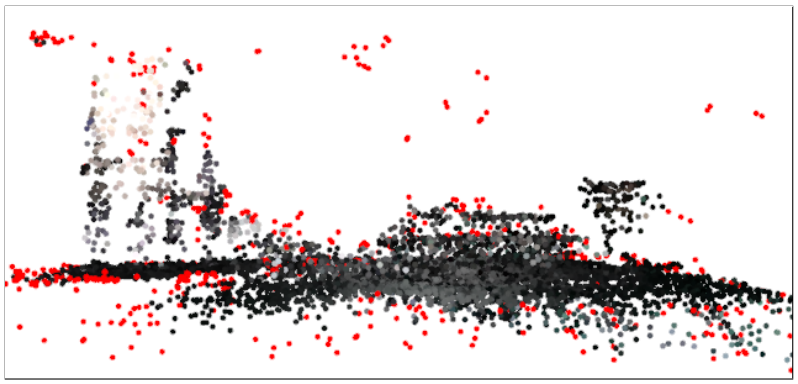


Figure 11: Example of outlier removal with points to be removed in red.

Consecutively, normals are estimated. As the direction of the normal is dubious, functions for refining the normals follow with purpose to aim the normals in the general direction towards the viewpoint. The viewpoint is estimated to be the mean of all reference observation points associated with the landmarks from which the initial point data is extracted after the dense correspondences section. In Figure 12 a few points of the point cloud are chosen to display the normals. The last operation in the point cloud processing is to crop the point cloud to improve the visualization. The median point is computed and points far away from it are removed.

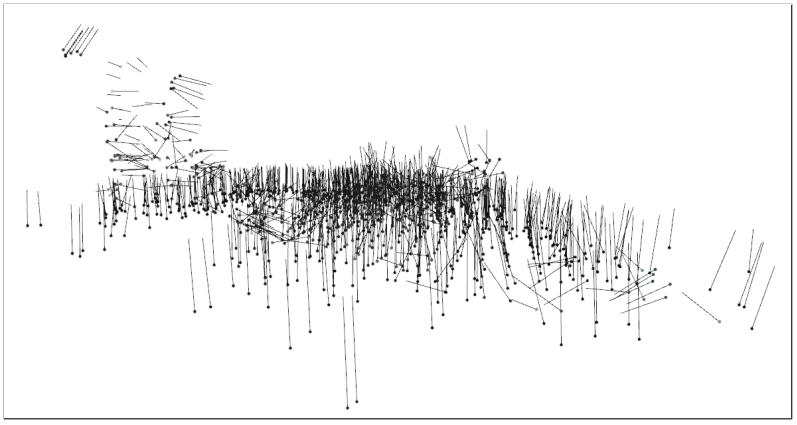


Figure 12: Visualization of normals of a few points of the pointcloud.

The data is stored in the output directory as Polygon File Format (PLY) [16] with point coordinates, colors of RGB values and normals for each point coded in ASCII. If necessary, the input color or normals can be set to `None` and no such data will be included in the file, but in this implementation of the program colors and normals will always be available. It is possible to choose decimal separator for the PLY file as comma or dot according to the computer settings or according to choice with input parameters when running GoPro Trails. The reason for this is because of Meshlab interpreting decimal separators depending on computer settings.

Generating a 3D mesh out of the point cloud is done using Meshlab. A Meshlab Filter Script is written in `filter_script.mlx` which contains the filters *Smooths normals on point set* and *Screened Poisson Surface Reconstruction*. The smoothing is based on the neighboring normals with weights according to their distance. The second filter is a technique for estimating surfaces from an oriented point set. The point set is transformed to a continuous vector field and a scalar function whose gradients is the best match to the field is sought. This minimization problem can be interpreted as solving the so called screened Poisson equation, where the screening factor (in Meshlab the parameter called *Interpolation Weight*) typically trades off the importance of fitting the gradients of the vector field and fitting the points. The surface is then extracted from the optimized scalar function [17]. When running the mesh command, the filters are automatically applied on the point cloud and the project is launched in Meshlab for viewing. An example of a resulting mesh along with a reference image can be seen in Figure 13.

(a) Reference image (b) Resulting mesh

Figure 13: Example of surface reconstruction results.

2.2 IMU Calibration

Even though the IMU was built into the GoPro Hero 6 unit, there was need for calibration of the rate and time offset of the IMU. A lot of simplifications were able to be made due to the fact that it was built in. The average optical flow between neighbouring image pairs were correlated with the norm of the angular speed from the gyroscope [1]. It is important to note that the time stamp for the average

optical flow between image pairs, is the average of the time stamps (i.e. right in between). In order to counteract this, the average optical flow signal was convolved with a small box filter of length 2.

Since the IMU was integrated into the camera, it was assumed that the time offset and rate for one sequence would be usable for other sequences. There was therefore no need to automatically solve the correlation. Instead, correlation was done by a process of visual comparison and iteratively improving estimations of the time offset and rate (see Figures 14 and 15). The result from this appears to have generalized well for other sequences, but a recalibration is probably needed if another GoPro Hero 6 unit is used.

As a result of this calibration, it became clear that the IMU data starts recording late and ends recording early. As mentioned in the Kontiki handler Section 2.1.4, views corresponding to times with missing IMU data had to be culled.

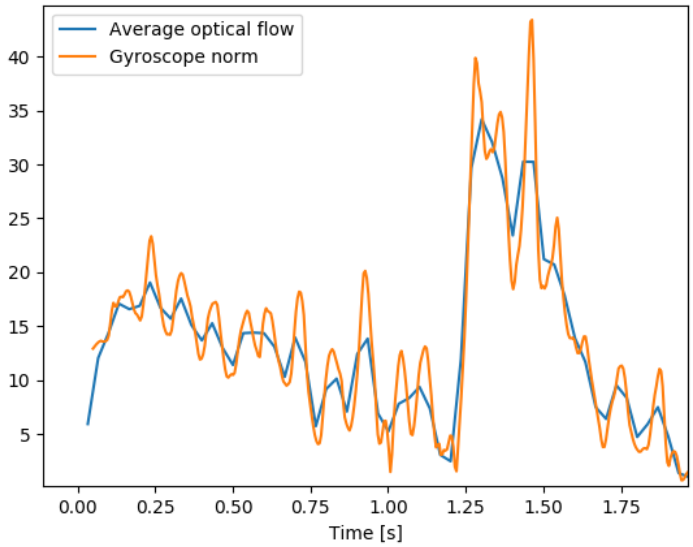


Figure 14: Optical flow offset estimation. Only early values, before a potential rate offset can corrupt the estimation.

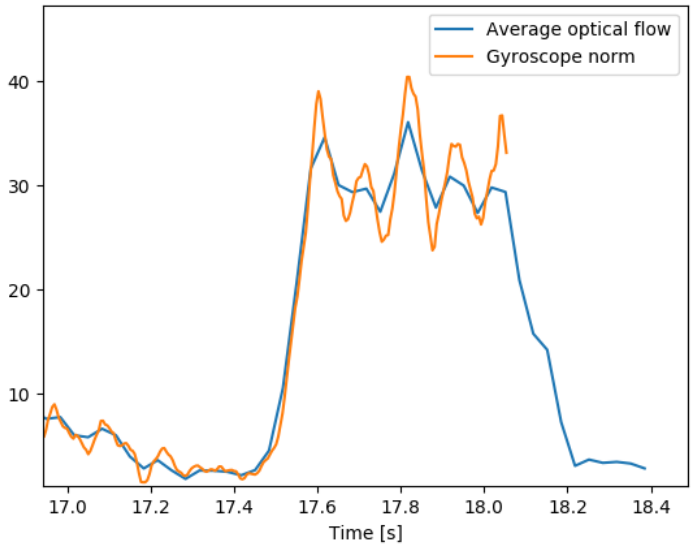


Figure 15: Optical flow rate estimation. A correct rate is essential for good correlation of late values.

2.3 Configuration

There are several parameters that affect the performance of the software. To allow the user to choose internal parameters to the Kontiki, tracking and georeferencing processes, these parameters are collected in a configuration file. The code for generating and parsing the configuration file is located in `ace3d/configuration`. The file `ace3d/configuration/config.py` contains a class for each group of parameters. Currently available parameter classes are *KontikiParams*, *ImuParams*, *GeoreferenceParams*, *TrackingParams* and *PointcloudParams*. New parameters and classes can be added in this file by simply adjusting the already existing classes or adding new ones. Note that the configuration class, *Config*, handles everything regarding the configuration file. If a new parameter class is added, then it also has to be added in the initialization method within the configuration class.

3 User Interface

The user interface consists of a command line interface (CLI) implemented using the python `argparse` module [18]. The top-level of the CLI consists of the following sub-commands:

- **run**: Runs the main part of the program. This command will perform tracking, SFM and georeferencing.
- **map**: Opens the trajectory visualization web application in the system default browser.
- **mesh**: Performs surface reconstruction on the pointcloud using Meshlab and then opens Meshlab to display the results.
- **create-config**: Generates a configuration file filled with deafult values.

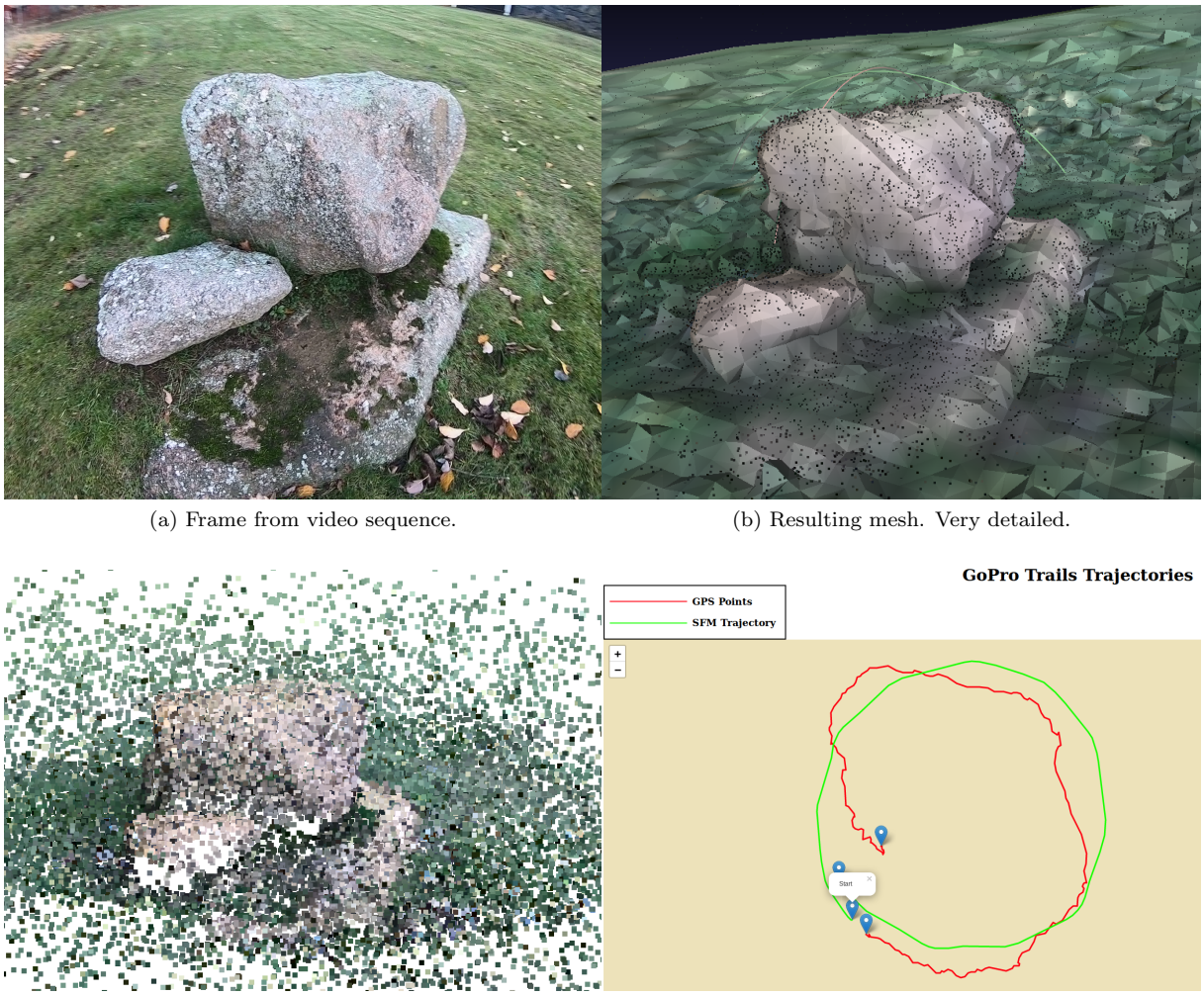
Each sub-command takes a set of optional and positional arguments. The CLI is described in more detail in the user manual, see Appendix B.

4 Results

Results for different sequences are presented in this section. All sequences use a Huber norm parameter value of 0.2 in the first phase, and values above 10 for the following phases. This is in order to be less affected by outliers before they are culled.

4.1 Stone

The result for the sequence of a stone is presented here. Figure 16 shows the stone in the different processing steps. This sequence produces the most recognizable reproduction of the object as a mesh.



(a) Frame from video sequence.

(b) Resulting mesh. Very detailed.

(c) Point cloud. Almost all parts of the stone are dense in points.

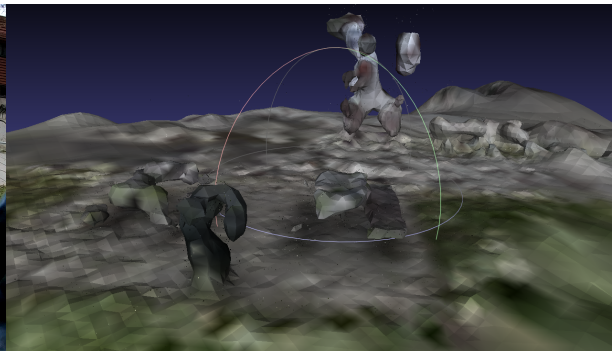
(d) Resulting trajectories. GPS data less reliable since the distance walked is very short.

Figure 16: Results for the sequence of a stone.

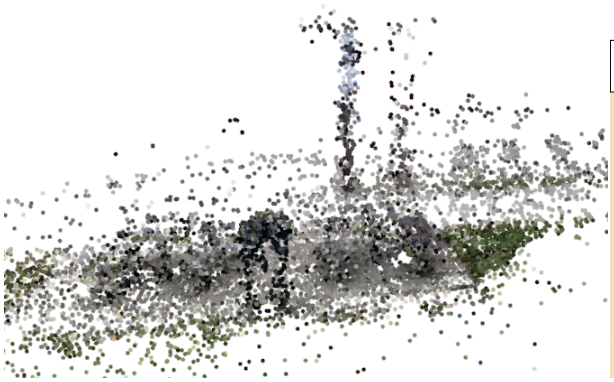
4.2 Benches

The result for the sequence of benches is presented here. Figure 17 shows the bridge in the different processing steps. It is difficult to make out the benches in the mesh, since the densification step has trouble finding good correspondences in those areas.

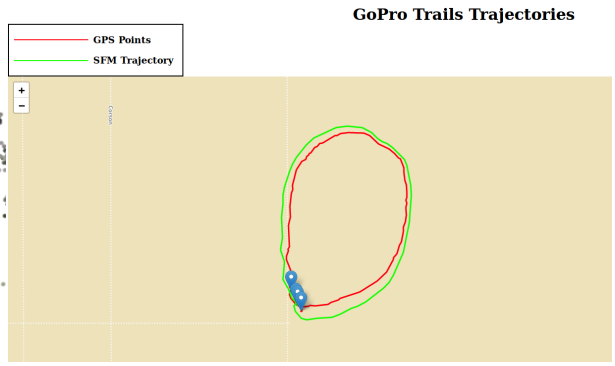
(a) Frame from video sequence.



(b) Resulting mesh. The benches are difficult to make out due to the sparsity of the point cloud the mesh is based on.



(c) Point cloud. The benches are very sparse. The trash can is easy to make out.



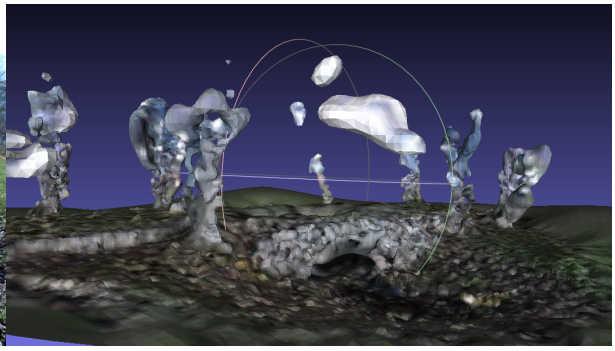
(d) Resulting trajectories. Good fit.

Figure 17: Results for the sequence of benches.

4.3 Bridge

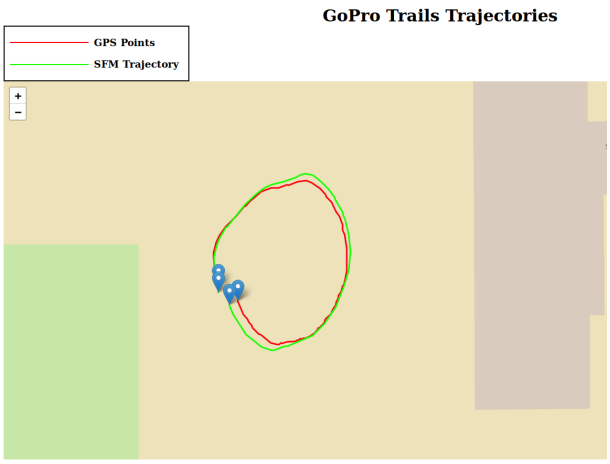
The result for the sequence of a bridge is presented here. Figure 18 shows the bridge in the different processing steps. Initially there was a sparse point cloud above the bridge that came from the branches of the trees. These were removed in the mesh preprocessing step. Other parts of the tree crowns were not removed and instead produce large blobs instead of thin branches in the mesh.

(a) Frame from video sequence.



(b) Resulting mesh. Parts of the tree crowns become large blobs instead of thin branches.

(c) Point cloud. The bridge is very easy to make out.



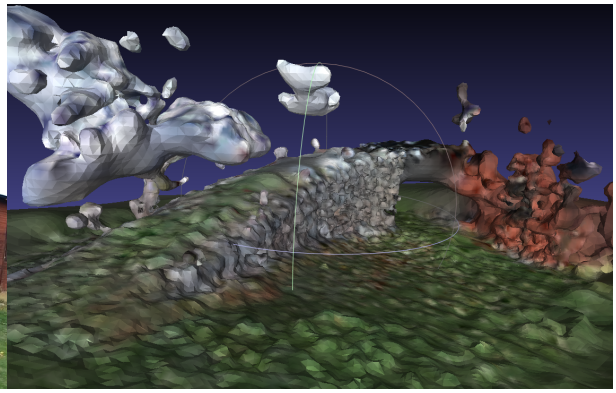
(d) Resulting trajectories. Both trajectories match very well.

Figure 18: Results for the sequence of a bridge.

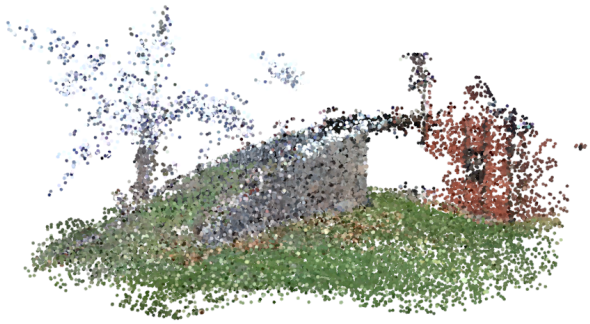
4.4 Barn

The result for the sequence of a barn is presented here. Figure 19 shows the barn in the different processing steps. The walls of the barn leak out as can be seen in the mesh. This is most likely due to a problem with the triangulation in the densification step, as described in the known problems Section 5. A large amount of these points were removed in the mesh preprocessing step.

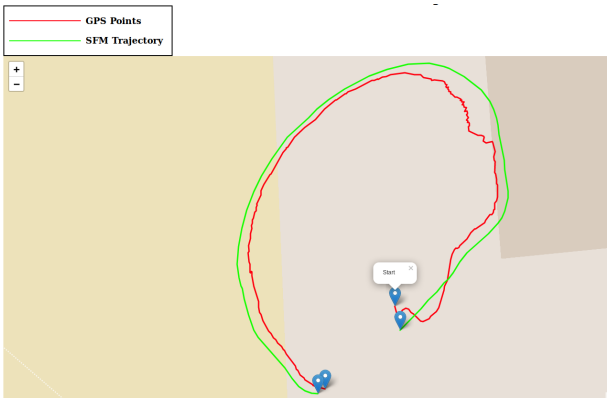
(a) Frame from video sequence.



(b) Resulting mesh. Trees becomes blobs, and the barn it self suffers from bad triangulations.



(c) Point cloud. Part of the wall leaks out due to bad triangulation.

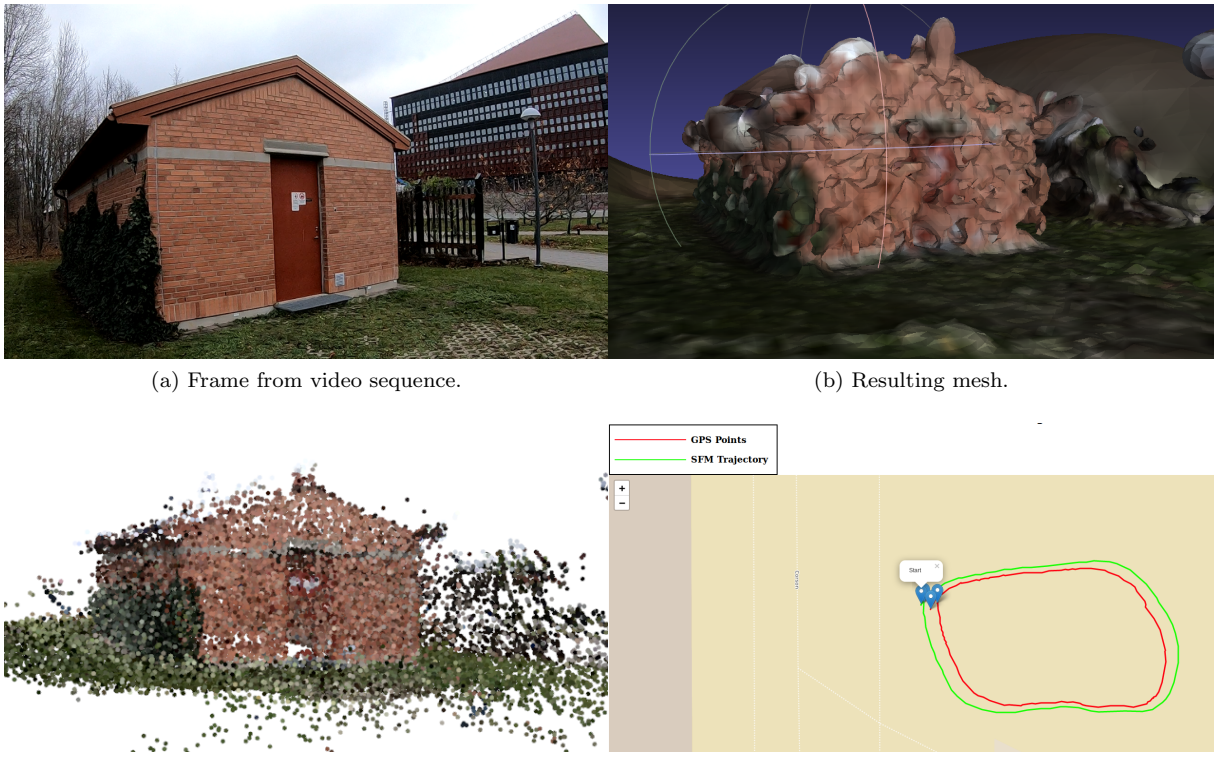


(d) Resulting trajectories. GPS based trajectory makes some odd twists and turns.

Figure 19: Results for the sequence of a barn.

4.5 Small House

The result for the sequence of a small house is presented here. Figure 20 shows the house in the different processing steps. Setting a random valid observation as the reference observation has greatly improved the stability for this sequence. If the reference observation is set to the first observation, the results seem to be more reliant on good tracks.



(a) Frame from video sequence.

(b) Resulting mesh.

(c) Point cloud. A lot of points filtered out in the mesh pre-processing step.

(d) Resulting trajectories. Very good match.

Figure 20: Results for the sequence of a small house.

4.6 Road

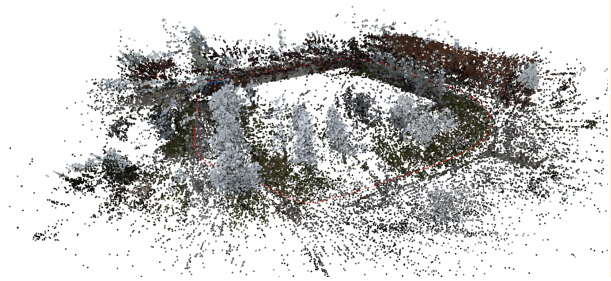
The result for the sequence of road around a building is presented here. Figure 21 shows the road in the different processing steps. Setting a random valid observation as the reference observation has greatly improved the stability for this sequence as well. The reason for this may be related to the car that passes by part way into the sequence as can be seen in Figure 21a.

Figure 22 shows a comparison between this sequence with and without a random reference observation, where the good results use a random reference, and bad results do not. The area in Figure 22b where the trajectory starts overlapping itself is around where the car first enters into view. Having the reference set to the first observation apparently corrupts the trajectory where these points are in view. An explanation for this might be that for views later on in the sequence that view the car, have their reference observation behind the viewing direction, perturbing the trajectory in a much more significant and biased way compared to a more randomly distributed perturbation caused by having a random reference.

Figure 22c and 22d show how almost all points and considered outliers in the latter case, compared to the random reference setup where a significant amount of the residuals can be considered inliers. Figure 23 shows how the random reference trajectory improved between phase 1 and the final phase.

(a) Frame from video sequence. Car enters frame, and remains for a significant amount of time.

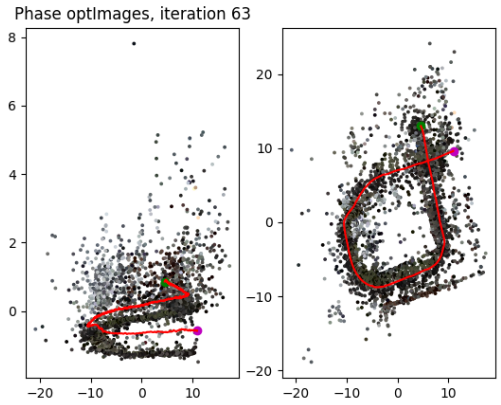
(b) Resulting mesh. Not detailed enough to make out most of the objects, but the road itself it easily identified.



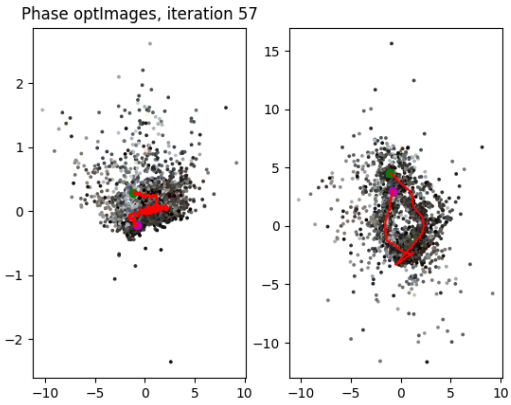
(c) Point cloud. Fairly easy to identify what are trees (gray blobs), and what are buildings (brown rectangular structures).

(d) Resulting trajectories. Good fit, but the SfM trajectory looks slightly small in scale.

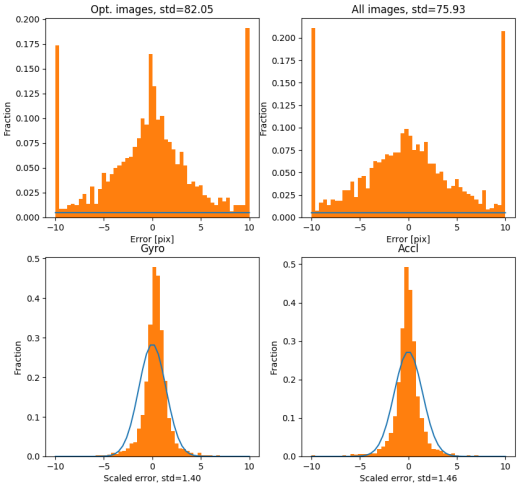
Figure 21: Results for the sequence with road around a building.



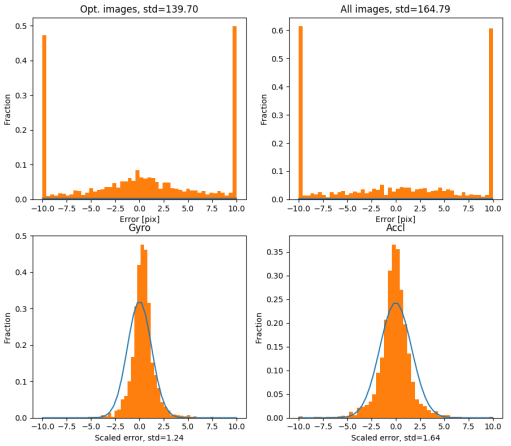
(a) Good points and trajectory from post phase 1. The trajectory goes roughly in a closed loop.



(b) Bad points and trajectory from post phase 1. The trajectory makes some erratic twists and turns that do not correspond with actual movement when filming. In particular a self-overlapping can be seen in the right image, which is where the car first entered the frame.



(c) Good residuals from post phase 1. A significant amount of the residuals are decently small, and can't be labelled as outliers.



(d) Bad residuals from post phase 1. Almost all of the residuals are large and will be considered outliers in subsequent stages, which will corrupt the result if they are wrongfully labelled such.

Figure 22: Comparison between good and bad results for the road sequence.

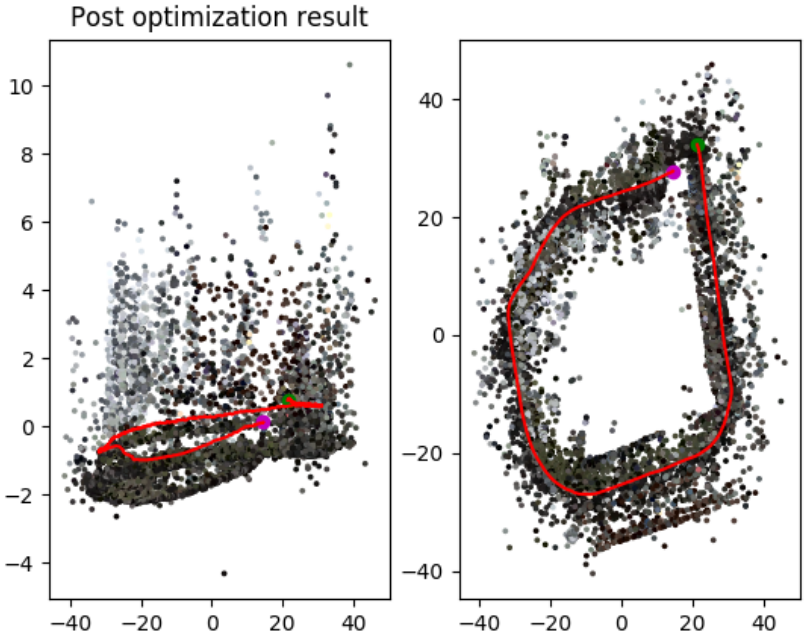


Figure 23: Good points and trajectory from post final phase. The trajectory follows the path walked by the camera man.

5 Known Problems

Triangulation of new landmarks found from PatchMatch is *highly* dependent on the estimated trajectory being of good quality. The trajectory being bad does unfortunately not mean that the densification produces no points; it means that it produces lots of points with the majority being of very low quality.

Choosing view pairs for use in PatchMatch is not a trivial task. There must be a sufficient translation between the two views for triangulation to perform as expected. Currently, care is taken so that sufficent translation takes place between view pairs. However, this is not always enough. For instance, if both views have the same viewing direction and the translation is along this same direction, observations in the center of the image will result in highly uncertain triangulations, as is the case in the barn sequence (Figure 19d). Note that correspondences lying in the periphery of the image does not have this issue, so it is not the case that we simply want to discard the entire view pair.

Outliers will greatly affect how the mesh looks. While it is possible that outliers may remain from the Kontiki pipeline, a poor densification is more likely to produce a large set of poorly triangulated points, due to the problems discussed in the paragraphs above.

The GoPro Hero 6 camera used in the project was somewhat unreliable when it comes to IMU and GPS data. For some recordings the GPS data and/or IMU data would be completely missing. Sometimes the accelerometer data is available but not the gyroscope data or vice versa. For some sequences the GPS data is available, but of very poor quality, resulting in bad georeferencing results.

6 Future Work

PatchMatch currently uses the L2 norm. Although fast and providing satisfactory results, other norms should probably be investigated. No illumination or rotation invariance means that PatchMatch performs sub-optimally.

A way to fix the problem where there is a high uncertainty in the triangulation could be to only triangulate points where the unprojection lines (the line that intersects the camera center, image point and the 3D point) are sufficiently non-parallel. This could be done by calculating the scalar product

between the direction vectors of the unprojection lines and only triangulate if this product is below a certain threshold.

There are a lot of configurable parameters available that has a significant effect on the results. The current default values are by no means optimal and needs to be tuned further. One possible way to find good default values could be to record multiple sequences and perform parameter search for parameters that minimize the residuals in the georeferencing over all these sequences.

There is no ground truth data available for the trajectory which means the quality of the GPS data and the generated trajectory cannot be measured. One way to construct ground truth data could be to draw and follow a trajectory on the ground with known/measured geometry. For instance, a circle with a known radius could be drawn and followed. The generated trajectory should then match the shape and radius of the circle, since the scale of the trajectory is supposed to be correct.

7 Conclusions

Overall, the produced software GoPro Trails works well, which can be seen in Section 4. In general, the generated trajectory can be seen as a refinement of the GPS data in terms of resolution and the 3D reconstruction is generally satisfactory. The densification and mesh steps do improve the results, but also produce some faulty data under certain conditions and need further work to remedy this.

In summary, the Kontiki library is suitable for reconstructing 3D models from GoPro videos. However, some parameter tuning might be required for some sequences. Since the generated trajectories are in general of good quality and have better resolution than the GPS data, they are reasonable to use as a refinement of the GPS data. Since the ground truth trajectories were not known, the accuracy for both the GPS data and the generated trajectory could not be measured. Hence, it is not possible to determine which of the GPS data and the generated trajectory that is most precise.

References

[1] H. Ovrén, *Continuous models for cameras and inertial sensors*, PhD dissertation, 2018.

[2] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia, "MeshLab: an Open-Source Mesh Processing Tool," in *Eurographics Italian Chapter Conference*, V. Scarano, R. D. Chiara, and U. Erra, Eds., The Eurographics Association, 2008, ISBN: 978-3-905673-68-5. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[3] H. Ovrén, *Github - Hovren*. [Online]. Available: https://github.com/hovren.

[4] *Openstreetmap*, https://www.openstreetmap.org/.

[5] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman, "PatchMatch: A randomized correspondence algorithm for structural image editing," *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 28, no. 3, August 2009.

[6] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 32, pp. 105–119, 2010. DOI: 10.1109/TPAMI.2008.275. eprint: arXiv:0810.2434[cs.CV]. [Online]. Available: http://lanl.arXiv.org/pdf/0810.2434.

[7] J.-Y. Bouguet, "Pyramidal implementation of the lucas kanade feature tracker description of the algorithm," *OpenCV Document, Intel, Microprocessor Research Labs*, vol. 1, January 2000.

[8] *Kontiki, the Continuous Time Toolkit*, 2018. [Online]. Available: https://hovren.github.io/kontiki/index.html.

[9] H. Ovrén and P.-E. Forssén, "Spline error weighting for robust visual-inertial fusion," in *:*, 2018. [Online]. Available: http://openaccess.thecvf.com/content_cvpr_2018/html/Ovren_Spline_Error_Weighting_CVPR_2018_paper.html.

[10] *EFFICIENTLY SELECTING SPATIALLY DISTRIBUTED KEYPOINTS FOR VISUAL TRACKING*, 2011. [Online]. Available: https://www.cs.ucsb.edu/~holl/pubs/Gauglitz-2011-ICIP.pdf.

[11] *Universal Transverse Mercator coordinate system - Wikipedia*, August 2018. [Online]. Available: https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system.

[12] *utm*, April 2017. [Online]. Available: https://pypi.org/project/utm.

[13] *scipy.optimize.least_squares — SciPy v1.1.0 Reference Guide*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html.

[14] *Pybind11*, https://github.com/pybind/pybind11.

[15] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.

[16] *PLY - Polygon File Format*, http://paulbourke.net/dataformats/ply/.

[17] M. Kazhdan and H. Hoppe, "Screened poisson surface reconstruction," *ACM Trans. Graph.*, vol. 32, no. 3, 29:1–29:13, Jul. 2013, ISSN: 0730-0301. DOI: 10.1145/2487228.2487237. [Online]. Available: http://doi.acm.org/10.1145/2487228.2487237.

[18] *16.4. argparse — Parser for command-line options, arguments and sub-commands — Python 3.6.7 documentation*. [Online]. Available: https://docs.python.org/3.6/library/argparse.html.

Appendices

A Requirement Specification

Requirements Specification
GoPro Trails
Images and Graphics, Project Course CDIO
TSBB11

Group Members
Daniel Cranston, dancr948
Carl Ekman, carek025
Lisa Eriksson, liser858
Freja Fagerblom, frefa105
Filip Skarsfelt, filsk543

Version 1.0

Monday 10th September, 2018

1 Requirements

Table 1 shows the requirements. The requirements with priority 1 must be fulfilled while those with priority 2 will be fulfilled if there is time available.

Table 1: Requirements.

Requirement #	**Functionality**	**Priority**
1	Investigate and if needed correct the transformation between image and intertial measurements	1
2	Verify the intrinsic camera calibration, and if needed recalibrate the camera	1
3	Extract inertial data stream from GoPro format	1
4	Extract GPS data stream from GoPro format	1
5	Feed data streams into Kontiki	1
6	Georeference the SfM solution into a map service	1
7	Visualize SfM trajectories in a map service	1
8	Visualize GPS trajectories in a map service	1
9	Create a website	1
10	Create command line interface to the program	1
11	Create a user manual	1
12	Write technical documentation	1
13	Create a video	1
14	Create dense correspondences	2
15	Apply meshing to the dense 3D point cloud	2
16	Create graphical user interface application	2

B User Manual

User Manual

GoPro Trails

Images and Graphics, Project Course CDIO

TSBB11

Group Members

Daniel Cranston, dancr948

Carl Ekman, carek025

Lisa Eriksson, liser858

Freja Fagerblom, frefa105

Filip Skarfelt, filsk543

Version 1.0

December 17, 2018

Contents

1 Introduction **3**

2 Installation **3**
2.1 Developer Installation 3

3 User Instructions **3**
3.1 Sub-command Run 4
3.1.1 Positional Arguments 5
3.1.2 Optional Arguments 5
3.2 Sub-command Map 6
3.2.1 Positional Arguments 6
3.2.2 Optional Arguments 6
3.3 Sub-command Mesh 6
3.3.1 Positional Arguments 7
3.3.2 Optional Arguments 7
3.4 Sub-command create-config 7
3.4.1 Positional Arguments 8
3.4.2 Optional Arguments 8

4 Configuration File **8**
4.1 Kontiki Parameters 8
4.2 Dense Correspondences Parameters 10
4.3 IMU Parameters 11
4.4 Georeference Parameters 11
4.5 Tracking Parameters 11
4.6 Point Cloud Parameters 12

1 Introduction

This document explains how to use the GoPro Trails software. The software produces a trajectory and a 3D model of a video sequence supplied by the user. The trajectory is visualized in an interactive map service and the 3D model is shown in MeshLab.

2 Installation

Follow the steps below to install the software. Observe that the program only works on Linux.

1. Make sure Ceres is installed. It can be installed using your system package manager. For instance, using Ubuntu you can install Ceres by executing

   ```
   $ sudo apt-get install libceres-dev
   ```

2. Clone the Git repository available at `https://gitlab.ida.liu.se/cdio-gopro/gopro`

   ```
   $ git clone https://gitlab.ida.liu.se/cdio-gopro/gopro
   ```

3. Make sure `pip` is installed and updated (at least version 18.1). Update `pip` by executing the following command

   ```
   $ pip install --upgrade pip
   ```

4. Install GoPro Trails by executing the following

   ```
   $ pip install ./path/to/cloned/gopro/directory
   ```

5. Verify that the program was installed correctly by executing

   ```
   $ gopro-trails --help
   ```

6. To use the surface reconstruction and 3D visualization features, MeshLab needs to be installed and available on the system `PATH`. MeshLab can be installed by downloading the Linux snap from `http://www.meshlab.net/#download`. Observe that at least version 2016.12-2 should be used.

2.1 Developer Installation

When developing the software further the flag `-e` can be used in step 4 in the previous section. This flag enables continuous update of the program while editing is performed. Hence, the developer does not have to uninstall and install the program after each change in the code. Step 4 can therefore be changed to the following

```
$ pip install -e ./path/to/cloned/gopro/directory
```

3 User Instructions

To print an explanation of the command line interface, execute the following command

```
$ gopro-trails --help
```

This prints the following

```
usage: gopro-trails [-h] {run,map,mesh,create-config} ...

Performs structure from motion and georeferencing.

optional arguments:
```

```
  -h, --help            show this help message and exit

Subcommands:
  {run,map,mesh,create-config}
                        Available subcommands.
    run                 Runs the SFM pipeline and produces output files.
    map                 Opens default web browser and displays a map
                        with GPS coordinates and SFM trajectory.
    mesh                Opens Meshlab and displays surface
                        reconstruction of generated 3D points.
                        Surface reconstruction will be
                        performed if not already found in the
                        output directory.
    create-config       Creates a configuration file in current
                        directory or specified directory with
                        tunable parameters.
```

As explained in the help message, the program is executed by supplying one of the sub-commands. To learn more about a specific sub-command execute `gopro-trails <sub-command> --help`. The sub-commands are further explained in the following sections.

3.1 Sub-command Run

This command executes the main part of the program. It will generate tracks (unless pre-computed), perform structure from motion, georeferencing of trajectory and store all output files in the output directory.

To print an explanation of the sub-command `run`, execute the following command

```
$ gopro-trails run --help
```

This prints the following

```
usage: gopro-trails run [-h] [-c CONFIG] [-t TRACKS] [-dd]
                        [-o OUTPUT_DIR] [-cp PARAM_PATH]
                        video-path

Runs the SFM pipeline and produces output files. Generated output files
will be stored in a directory called gopro-trails-output created in the
current directory. NOTE: previous files in the output directory will be
overwritten if the same directory is used again.

positional arguments:
  video-path            Path of the GoPro video file to use.

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        Path to config file. If not given, default path
                        will be used.
  -t TRACKS, --tracks TRACKS
                        Path to hdf file containing pre-computed tracks.
                        If not given, tracks will be generated.
  -dd, --disable-dense  Disable computation of dense correspondences
                        and dense 3D points.
  -o OUTPUT_DIR, --output-dir OUTPUT_DIR
                        Specify path to output directory (will be
                        created if it does not exist). If not
                        specified, output directory will be
```

```
                        created in the current directory
-cp PARAM_PATH, --camera-parameters PARAM_PATH
                        Optional path to an hdf file with custom
                        calibration parameters or one of the built
                        in camera parameter files by specifying
                        "1080p30fps" or "1080p60fps".
                        (Default value: "1080p30fps")
```

3.1.1 Positional Arguments

This sub-command takes one positional argument, supplied after all the optional arguments.

video-path

Path of the GoPro video file to use.

3.1.2 Optional Arguments

This sub-command supports several optional arguments.

--help

The `--help` argument displays the the description for the sub-command and all possible arguments with belonging descriptions. The print after running the program with this argument is displayed above in Section 3.1.

--tracks

Path to an HDF file containing pre-computed tracks. When not given the tracks are generated for the video.

--disable-dense

If this switch is given, the computation of dense correspondences and 3D points will be skipped and the more sparse 3D points from the SFM system will be output.

--output-dir

Specifies an output directory that will be created and hold all the output files. If not given, a folder named `gopro-trails-output` will be created in the current directory. Observe that previous files in the output directory will be overwritten if the same directory is used again.

--camera-parameters

Used to specify a custom HDF file with camera parameters or one of the built in camera parameter files (”1080p30fps” or ”1080p60fps”). The custom file should have the following fields:

- **size**: Video size as [width, height].
- **readout**: Readout time in seconds.
- **K**: Intrinsic camera parameters, 3x3 matrix.
- **wc**: Distortion center ω_c, 2-element vector.
- **lgamma**: Distortion parameter γ, scalar.
- **fps**: Frames per second.

3.2 Sub-command Map

This command opens the default web-browser and displays the created SFM trajectory together with the GPS coordinates in a map service.

To print an explanation of the sub-command `map`, execute the following command

```
$ gopro-trails map --help
```

This prints the following

```
usage: gopro-trails map [-h] [gopro-trails-output]

Opens default web browser and displays a map with
GPS coordinates and SFM trajectory.

positional arguments:
  gopro-trails-output  Path to GoPro-trails output directory.
                       (Default value: ./gopro-trails-output)

optional arguments:
  -h, --help           show this help message and exit
```

3.2.1 Positional Arguments

Sub-command `map` only has one positional argument. It is described below.

gopro-trails-output

Path to GoPro-trails output directory. This only needs to be specified if the user has chosen an own output directory when running the program. If this argument is not specified then the default directory `./gopro-trails-output`, where the output files automatically are stored, will be used.

3.2.2 Optional Arguments

Sub-command `map` only has one optional argument. It is described below.

--help

The `--help` argument displays the the description for the sub-command and all possible arguments with belonging descriptions. The print after running the program with this argument is displayed above in Section 3.2.

3.3 Sub-command Mesh

This command performs surface reconstruction from the generated 3D points and opens Meshlab to display the results. If this command already has been executed, then the previous surface reconstruction will be used and opened in Meshlab.

To print an explanation of the sub-command `mesh`, execute the following command

```
$ gopro-trails mesh --help
```

This prints the following

```
usage: gopro-trails mesh [-h] [-dds] [-cds] [gopro-trails-output]

Opens Meshlab and displays surface reconstruction of
generated 3D points. Surface reconstruction will be
performed if not already found in the output directory.

positional arguments:
  gopro-trails-output   Path to GoPro-trails output directory.
```

```
                        (Default value: ./gopro-trails-output)

optional arguments:
  -h, --help            show this help message and exit
  -dds, --dot-decimal-sep
                        Use dot instead of system separator as floating
                        point decimal separator in ply file for meshlab
                        (Try this if file fails to load in meshlab)
  -cds, --com-decimal-sep
                        Use comma instead of system separator as
                        floating point decimal separator in ply
                        file for meshlab
                        (Try this if file fails to load in meshlab)
```

3.3.1 Positional Arguments

Sub-command `mesh` only has one positional argument. It is described below.

gopro-trails-output

Path to GoPro-trails output directory. This only needs to be specified if the user has chosen an own output directory when running the program. If this argument is not specified then the default directory `./gopro-trails-output`, where the output files automatically are stored, will be used.

3.3.2 Optional Arguments

Sub-command `mesh` has two optional arguments. These are described below.

--help

The `--help` argument displays the the description for the sub-command and all possible arguments with belonging descriptions. The print after running the program with this argument is displayed above in Section 3.4.

--dot-decimal-sep

The `--dot-decimal-sep` flag forces dots to be used in the ply file copy that is read by Meshlab, regardless of system separator. **Note:** if Meshlab is unable to open the ply file with the default setting (system separator), then the problem might be solved by changing the decimal separator. The original ply file from the `run` command is never modified in-place.

--com-decimal-sep

The `--com-decimal-sep` flag forces commas to be used in the ply file copy that is read by Meshlab, regardless of system separator. See notes for `-dds`.

3.4 Sub-command create-config

This command creates a GoPro Trails configuration file in the current directory, or the directory specified. The configuration file contains tunable parameters that affect the results of the program.

To print an explanation of the sub-command `create-config`, execute the following command

```
$ gopro-trails create-config --help
```

This prints the following

```
usage: gopro-trails create-config [-h] [config-file]

Creates a configuration file in current directory or specified
```

```
directory with tunable parameters.

positional arguments:
  config-file  Configuration file destination path.
               (Default value: ./gopro-trails.cfg)

optional arguments:
  -h, --help   show this help message and exit
```

3.4.1 Positional Arguments

Sub-command `create-config` only has one positional argument. It is described below.

config-file

Path to configuration file that will be created. If not specified it will be called `gopro-trails.cfg` and be placed in the current directory.

3.4.2 Optional Arguments

Sub-command `create-config` has one optional argument. It is described below.

--help

The `--help` argument displays the the description for the sub-command and all possible arguments with belonging descriptions. The print after running the program with this argument is displayed above in Section 3.4.

4 Configuration File

The configuration file created via the sub-command `create-config` (see Section 3.4) contains different groups of parameters. The created configuration file parameters will be filled with default values. The default values are not considered optimal in any way and should most likely be tuned to get good performance. The groups and their parameters are described in the subsections below.

4.1 Kontiki Parameters

The group called *KontikiParams* contains parameters which effects the optimization process run by Kontiki. These parameters are described below.

huber_c1

Adjusts the impact of outliers for the first optimization phase (there are 4 phases in total). The larger value, the larger impact of outliers.

huber_c2

Adjusts the impact of outliers for the second optimization phase (there are 4 phases in total). The larger value, the larger impact of outliers.

huber_c3

Adjusts the impact of outliers for the third optimization phase (there are 4 phases in total). The larger value, the larger impact of outliers.

huber_c4

Adjusts the impact of outliers for the fourth optimization phase (there are 4 phases in total). The larger value, the larger impact of outliers.

max_iter1

Sets the maximum number of iterations during the first optimization phase.

max_iter2

Sets the maximum number of iterations during the second optimization phase.

max_iter3

Sets the maximum number of iterations during the third optimization phase.

max_iter4

Sets the maximum number of iterations during the fourth optimization phase.

initial_inv_depth

Specifies the initial inverse depth of all landmarks. This is included as a debug tool and should generally not be changed from the default value.

acc_bias

Specifies the accelerometer bias. Observe that this value is only used if the parameter `zero_bias` is set to `False`.

zero_bias

Specifies if no bias should be used during optimization. If set to `False`, accelerometer bias will be considered during optimization.

rand_reference

Specifies if a random observation should be the reference to a landmark. If set to `False`, then the first observation is used as reference.

gyro_std

Specifies the standard deviation of the gyroscope noise, which is used to estimate a better weighting.

acc_std

Specifies the standard deviation of the accelerometer noise, which is used to estimate a better weighting.

max_error_first_cull

Specifies the threshold for outlier detection in pixels. This threshold is used in the first outlier elimination. If the mean of the residuals for a landmark is above the threshold, the landmark is culled.

max_error_second_cull

Specifies the threshold for outlier detection in pixels. This threshold is used in the first outlier elimination. If the mean of the residuals for a landmark is above the threshold, the landmark is culled.

q_gyro

Specifies a quality measure for how well you want the SO3-spline to model the measured data. This affects the knot spacing of the spline.

q_acc

Specifies a quality measure for how well you want the R3-spline to model the measured data. This affects the knot spacing of the spline.

num_obs_per_frame

Specifies how many observations per keyframe are to be used. The selected observations will be evenly spaced throughout the image.

keyframe_ratio

Specifies the threshold of common tracks between keyframes. A higher value will result in more keyframes.

keyframe_min_dist

Specifies the minimum spacing between keyframes.

precalc_rel_pos

A debug parameter that specifies whether the relative pose between the IMU and the camera should be preprocessed instead of letting Kontiki handle the transformation. Shouldn't affect the result.

plot_progress

Specifies whether plots of the residuals and trajectory together with 3D points should be plotted during the optimization process. This is best used as a debug tool, or to try out new parameters, since it is easy to abort early, but shouldn't be used in the general case since it affects the optimization procedure.

plot_phase

Specifies whether or not to plot the residuals and trajectory with 3D points after optimization phase.

plot_final

Specifies whether or not to plot the end result of the residuals and trajectory with 3D points after the whole Kontiki pipeline. This halts the program until the plots are closed, in order to give the user time to view the plots before the program exits after finishing.

4.2 Dense Correspondences Parameters

The group called *DenseCorrespondencesParams* contains parameters regarding the densification of the point cloud. These parameters are described below.

baseline_dist

Specifies the minimum distance in meters between image pairs used in PatchMatch

keyframe_stride

Specifies how many frames to skip before choosing the next image pair

scale_factor

Specifies how much the images should be downscaled before PatchMatch is executed. A value of 2 results in a down-scaling of half the original size.

kernel_size

Specifies the patch size to be used in PatchMatch.

pm_iter

Specifies how many iterations PatchMatch will run.

max_reproj_error

Specifies the maximum reprojection error (in pixels) tolerated when removing outlier landmarks generated from PatchMatch correspondences. Landmarks with a larger error than this value will be removed.

visualize

Boolean specifying whether or not to visualize the correspondences found by PatchMatch.

4.3 IMU Parameters

The group called *ImuParams* contains parameters regarding the IMU. These parameters are described below.

true_rate

Specifies the sampling rate of the IMU.

offset

Specifies the time offset between when the camera and the IMU starts recording.

4.4 Georeference Parameters

The group called *GeoreferenceParams* contains parameters which effects the georeferencing step. These parameters are described below.

gps_outlier_threshold

Threshold in metres for deviation from short time median GPS point.

gps_outlier_median_interval

Time interval used to compute short time median of GPS points.

trajectory_sample_freq

Sampling frequency (sample/s) used to sample the SFM trajectory.

4.5 Tracking Parameters

The group called *TrackingParams* contains parameters which effects the tracking step. These parameters are described below.

min_track_length

Specifies the minimum amount of frames in which a track has to be alive in order to be saved.

backtrack_length

Specifies the amount of frames in which to backtrack.

min_points

Specifies how many points to track in a frame.

min_distance

Specifies the minimum distance between existing and new tracks when finding new tracks.

win_size

Specifies the patch size of the tracker.

visualize

Specifies whether or not to visualize the tracking procedure.

4.6 Point Cloud Parameters

The group called *PointcloudParams* contains parameters which effects the surface reconstruction step. These parameters are described below.

voxel_size

Sets the voxel size for down-sampling point cloud during processing stage to create a uniformly down-sampled point cloud from a regular voxel grid.

nb_neighbors

This specifies how many neighbors are taken into account in order to calculate the average distance for a given point in the outlier removal stage.

std$_ratio$

This parameter sets the threshold level based on the standard deviation of the average distances across the point cloud. A lower threshold will remove more points.

radius

Specifies search radius for normal estimation. Neighboring points within the radius will affect the normal.

max_nn

This sets the maximum nearest neighbors that are taken into account for normal estimation to save computation time.

min_dist

The minimum distance from reference observation position a landmark needs to have to be extracted from the Kontiki Object to the point cloud.

max_dist

The maximum distance from reference observation position a landmark needs to have to be extracted from the Kontiki Object to the point cloud.