Berner
Fachhochschule

Open Source SLAM Library for Embedded Systems

Stefan Eichenberger
<eichest@gmail.com>

January 2020

Advisor: Prof. Marcus Hudritsch
Expert: Dr. Harald Studer (Optimo Medical AG)
Departement: TSM CPVR Lab, BFH

Abstract

Simultaneous location and mapping (SLAM) is a technology used for robot navigation and augmented reality. Today most SLAM libraries are proprietary or not ready for embedded systems. In this thesis, we write and analyse a library based on Semi-Dense Visual Odometry (SVO), which runs on embedded systems. Compared to the publicly available implementation of SVO, our version presents two advantages: it uses stereo cameras as input and it has fewer dependencies to third-party libraries.

Executive Summary

Today we use Simultaneous Location and Mapping (SLAM) for robot navigation and augmented reality. These algorithms often need a high performance graphics card or a high performance processor. For industrial and robotics purposes we face constraints in space, temperature and power consumption. Therefore, we use devices with less power consuming CPUs, which in turn are less powerful. In this thesis, we implement and analyse an algorithm called SVO, which is capable to run on embedded devices. The reference implementation uses monocular cameras, while we use a stereo camera. We show that the algorithm is based on optical flow, a well-known principle in computer vision. The implemented algorithm performs four tasks. First, it maintains a 3D point cloud which is used to estimate the camera pose. Second, it performs a pose estimation based on the 3D cloud. We call this sparse image alignment. It uses a changed version of optical flow, which directly outputs a pose instead of a warp matrix. Third, to make the guess more accurate, it refines the pose by using standard optical flow, followed by a minimization of the re-projection error. Finally, in a last step, the algorithm performs a 3D point cloud update where it refines the cloud by taking the new pose into account.
We show that the algorithm works as outlined and can estimate the 3D pose as described by the authors of SVO. We see that the algorithm performs faster than a comparable SLAM algorithm called ORB SLAM. However, the current implementation is less accurate than ORB. We conclude that SVO shows great promise as a SLAM algorithm for embedded devices.

Contents

Chapter 1

Introduction

Humans use different sensors to estimate the pose of their head in a room. Everyone who has ever tried to stand on one leg with their eyes closed knows that closed eyes make finding the balance harder. This experiment shows that our eyes are an import factor for us humans to balance. In this thesis, we use optical information from a camera to estimate its angle and position (pose). We call this visual odometry (VO). Most algorithms that implement visual odometry also require SLAM (Simultaneous Localization and Mapping). Such algorithms create a map of the environment while constantly estimating the camera pose. Therefore, we use this terms interchangeably in this document. Knowing the position of an object is important in various applications and becomes always more important for autonomous navigation. Examples of such applications are e.g. drone navigation, robot navigation or augmented reality. Those applications require a flexible and, ideally, mobile device. Therefore, we can't use a PC with a huge power supply. In consequence, such an algorithm should ideally be able to run on embedded devices without the necessity of powerful GPUs or CPUs.
The lab for Computer Perception and Virtual Reality (CPVR) at the Bern University of Applied Science uses an algorithm called ORB SLAM [16] for projects related to augmented reality. It tweaked the algorithm to achieve an acceptable frame rate of 10-20 frames per second(fps) on mobile phones. However, a faster frame rate would mean that no interpolation is needed or that the algorithm could run on lower end devices. Therefore, we test a different SLAM algorithm, which is called Semi-Dense Visual Odometry (SVO) [8]. The second SVO paper [9] from the same author states that the duration to process one frame on a modern processor is below 5 ms when running on two CPUs. Even though SVO SLAM is open source, the reference implementation is over four years old and has many third party dependencies. A newer version, based on the second paper, is available but its implementation is closed source. In this document, we describe our own implementation of the SVO algorithm but we focus on stereo camera SLAM while the original implementation uses monocular cameras. We analyse the robustness of SVO and how suitable it is to run on embedded systems.

1.1 SLAM

SLAM is a technology that takes images as input and generates a map of its environment while simultaneously estimating the pose of the camera. There are two groups of SLAM algorithms. On the one side, indirect SLAM algorithms extract features from an image and then match these features based on descriptors. For each image it is necessary to detect features, extract descriptors, compare the descriptors to previous images and triangulate the camera pose. On the other side, direct SLAM algorithms directly match one frame to another by their intensity differences. They optimize the pose and minimize the intensity difference over the whole image (dense) or at some sparse points (sparse). Figure 1.1 shows an overview of the different SLAM types.

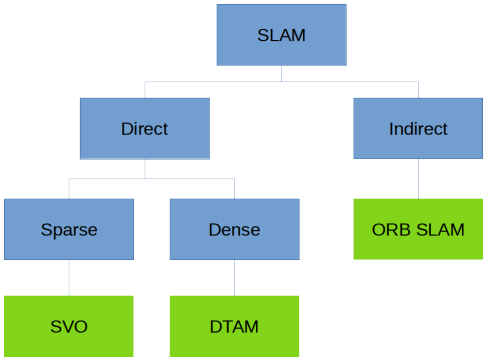


Figure 1.1: SLAM Types

As mentioned, we implement an SVO-like algorithm in this thesis. It is part of the direct sparse group.

1.2 Stereo and Monocular SLAM

In this thesis we focus on stereo SLAM. Many recent papers focus on monocular SLAM because monocular cameras are cheaper and available on mobile phones. Monocular SLAM has two problems that do not prevail with stereo SLAM. First, an initialization process needs to estimate an initial point cloud over several images. It is not possible to calculate depth with only one image. Second, monocular SLAM cannot guess the scale of the point cloud. To understand this, we imagine a small model of a city. With only one camera we cannot distinguish between a camera movement in the city and a movement in the model of the city. With stereo SLAM we do not have this issue. For each camera pose we get two images with a known distance between the two camera sensors (baseline). This enables us to calculate the depth of the points based on a known scale. Based on these points we can

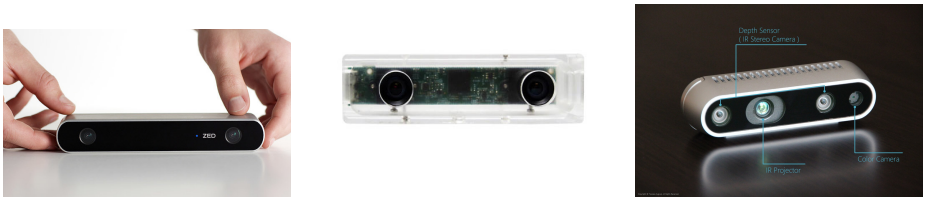
now generate a 3D point cloud. We therefore have an instantaneous initialization and can calculate the pose in a known unit (e.g. meters). Tracking between two camera poses is however identical for monocular and stereo SLAM.

1.3 Camera

To perform stereo computer vision, we need a stereo camera. There are a few stereo cameras available on the market. It is also possible to build a stereo camera with two monocular cameras. However, we decide against this option because it would require a mechanism to synchronize the images from both streams. We analyze three stereo cameras that are available on the market. An ideal camera would have a high frame rate, a high resolution, a global shutter, USB3.0 with UVC driver and a high baseline while being cheap. Table 1.1 shows a comparison of a ZED camera from Stereolabs [14], a Tara from Econ [7] and a Realsense D435 from Intel [13].

Camera	Resolution	FPS	Shutter	Driver	Baseline	IMU	Price
ZED	1920x1080	30	rolling	UVC	120 mm	6DoF	449
Tara	752x480	60	global	UVC	60 mm	6DoF	149
D435	1280x720	90	global	UVC	50 mm	6DoF	179

Table 1.1: Tested Camera Types



(a) Stereolabs ZED (b) Econ Tara (c) Intel Realsense D435

Figure 1.2: Tested Stereo Cameras

For this thesis, we use the Econ Tara because it has an outstanding price ratio and because it is a conventional stereo camera.
The Intel RealSense D435 camera can calculate depth images directly on the camera. For this thesis this is not required because we calculate the depth on our own. Because this camera has a smaller baseline we prefer the Econ Tara.
The ZED camera has a huge baseline of 120 mm. This allows depth estimation over higher range. We decide against this camera mainly because of its high price.

1.4 Purpose of this Thesis

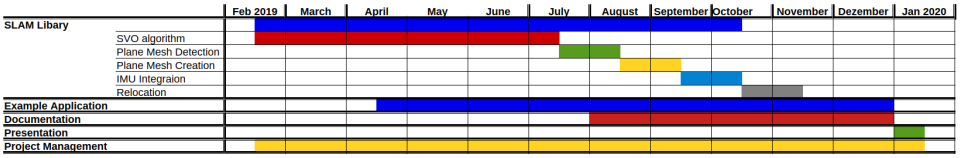
This thesis first implements SVO SLAM as an open source library. Next, based on this implementation, we evaluate how well the algorithm is suited for embedded systems. Finally, we benchmark the implementation against the open source reference implementation, with regard to speed and accuracy. Furthermore, we provide a 3D viewer and a demo application showing the possibilities of the SLAM library.

1.5 Outline

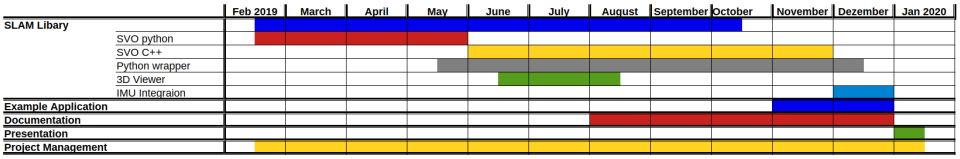
This document is organised as follows. We describe the SVO algorithm in chapter 2. We keep the chapter succinct by providing the mathematical details in later chapters. Chapter 3 describes how we receive depth information from a stereo camera. In chapter 4 we describe optical flow. We first start with an intuitive section and then move forward to the mathematical details. In chapter 5 we present the mathematics for refining the pose after a first estimate. The implementation of the algorithm is the subject of chapter 6. We then compare the results of the implementation with previous work in chapter 7. Ultimately, in chapter 8, we discuss the final results.

1.6 Planning

The master thesis is honored with 27 ECTS. 1 ECTS consumes 30 hours which results in 810 hours. Assuming 8.5 working hours per day, we have to spend 95.29 working days. Because the thesis is done part time, we can use a full year. We assume one year has 48 working weeks. During the first 16 weeks, we can only spend 1.5 days per week for the thesis (because of additional modules). This means that during the first 16 weeks we only work 24 days on the project. 71.29 days are left for the last 32 weeks. This results in 2.228 days of work during these weeks. Figure 1.3 shows the initial and the final planning.
Implementing the library took more time than expected. Therefore, plane detection and mesh creation have been removed. If this is required an additional library like the open source "Point Cloud Library" [15] can be used.



(a) Inital planning



(b) Final planning

Figure 1.3: Planning

As the final planning shows, we first started with a Python based implementation of the algorithm. The idea was, that Python code is more readable and better suited for experimenting. However, the problem was that the code was slow even when using Numpy to speed up matrix multiplications. Therefore, we implemented the library completely in C++. However, in parallel we developed a Python wrapper for debugging. The final version still provides the wrapper but we didn't document it.
Initially, there was no plan to write a 3D viewer. However, to have a flexible tool for debugging, we developed a viewer which can receive data over network. This allows us to analyse 3D data remotely and to separate the code base of the algorithm from the code base of the viewer. Because of the issue with Python and implementing the additional viewer, we reduced the scope and set the focus on the library.

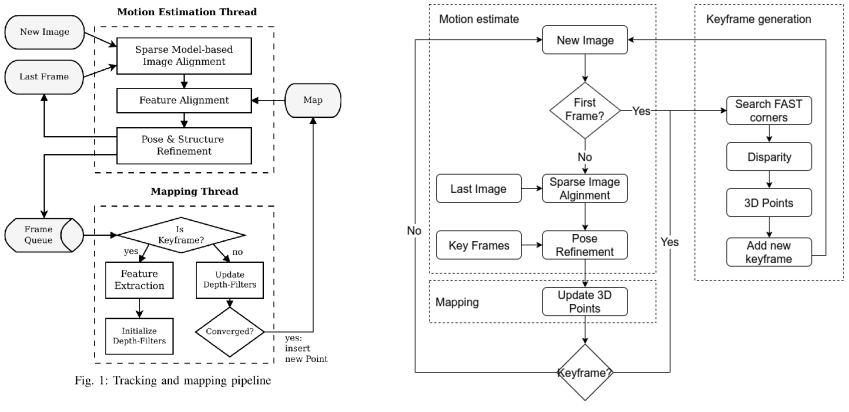
Chapter 2

SVO

The subject of this chapter is the description of the basic idea behind SVO SLAM. We do not dive into the mathematical details yet. We will see that most parts of the algorithm are based on the idea of optical flow. Therefore, we will handle this topic in chapter 4.

2.1 Architecture

Figure 2.1 shows the architecture of SVO compared to our implementation. It depicts many differences between the original implementation and what we do. However, the two main steps, sparse model based image alignment and pose refinement, are the same. Our implementation does not use multi-threading, such that everything is done sequentially. This decision makes the duration for processing one frame more deterministic. Because we focus on stereo cameras, we do not have to implement the mapping task in the same complexity as in the monocular implementation.



(a) Original SVO (b) Our SVO

Figure 2.1: SVO Flowchart Comparison

2.2 Keyframe Generation

The original SVO implementation uses a monocular camera by which it is not possible to estimate depth with the first image. In order to estimate the

depth at a given point in the image, a movement along x or y axis is required. In this thesis, we use a stereo camera with a known baseline. Therefore, we can estimate the depth of a point from the first image.
We only calculate the depth at certain points in the image. Those are called keypoints. Keypoints are pixels that have an outstanding characteristic compared to their neighbouring pixels (e.g. corners). We add new keypoints when we do not see enough keypoints in the current image. We call images where we insert new keypoints keyframes. For each keyframe we perform the following steps:

1. Search FAST corners in the image [18]
2. Do Sobel [10] filtering of the image in horizontal direction
3. Divide image in $n \cdot m$ blocks (e.g. 16x10)
4. Select one FAST corner in each block as keypoint
   (a) If we cannot find a FAST corner, we use the point with the biggest gradient from Sobel filtering as keypoint
5. Reuse keypoints found in previous keyframes. Insert only new keypoints in empty blocks
6. Calculate depth for each new keypoint (see 3)
7. Transform the keypoint depth to world coordinates by using the camera pose

Chapter 3 shows the details on how we calculate the depth and the world coordinates from a stereo image.

2.3 Sparse Image Alignment

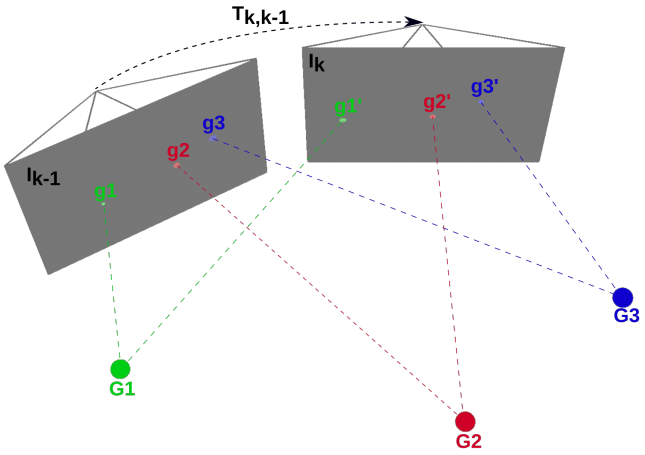


Figure 2.2: Sparse Image Alignment
Search minimal intensity difference between g in previous image I_{k-1} and g' in the current image I_k by optimizing pose $T_{k,k-1}$

From the keyframe generation step we get an initial point cloud. We use this point cloud to estimate the pose and motion of the camera. We try to find a pose that minimizes the photometric error of small patches around the keypoints. The photometric error is the intensity difference between two subsequent images. We do this by minimizing the formula shown in equation 2.2. We use equation 2.1 to project the 3D point cloud to the current frame. This step is shown in figure 2.2. T corresponds to our extrinsic matrix written as rotation and translation matrix in equation 2.1. By using Gradient Descent we can minimize the intensity difference by changing the parameters of the extrinsic matrix r_{ij} and $t_{x,y,z}$. As an initial guess for the pose, we use the current pose and add the motion model. The motion model is the difference between the pose in the previous frame $t-1$ and the pose in frame $t-2$. If we have an additional IMU available, we can also improve the motion model by adding information from this sensor (e.g. angle velocity from gyroscope).

$$\begin{pmatrix} u \\ v \\ s \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$
$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{u}{s} \\ \frac{v}{s} \end{pmatrix} \tag{2.1}$$
$$=> g = h(C \cdot T \cdot G) = h(P \cdot G)$$

f_x, f_y : Focal length
c_x, c_y : Principal point (z axis of camera goes through)
X, Y, Z : Point coordinates with local camera coordinates
u, v, s : Image coordinates with scale s
x, y : Keypoint projection from 3D cloud
r_{ij} : Camera rotation parameter
$t_{x,y,z}$: Camera translation
g : 2D point x,y
G : 3D point X,Y,Z
C : Intrinsic Camera Matrix
T : Extrinsic Camera Matrix (Pose)
h : Camera coordinates (u,v,s) to pixel coordinate (x,y) function
P : Projection Matrix

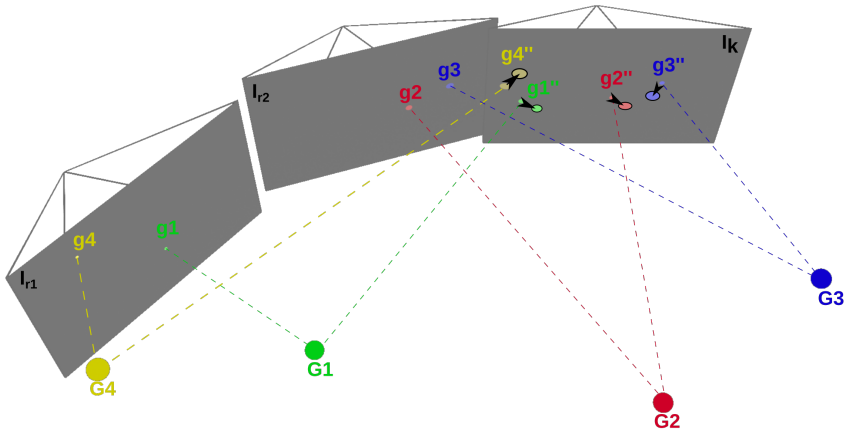
$$E_p = \min_P (\sum_{a \in A} \sum_{x=a_x-\frac{m}{2}}^{m} \sum_{y=a_y-\frac{n}{2}}^{n} (I_{k-1}(x'', y'') - I_k(x'(P), y'(P)))^2) \tag{2.2}$$

m, n : Patch size
x'', y'' : Warped pixel position in the template
x', y' : Pixel position in the current image
I_{k-1} : Template image
I_k : Current image
A : For all keypoints
E_p : Remaining photometric error

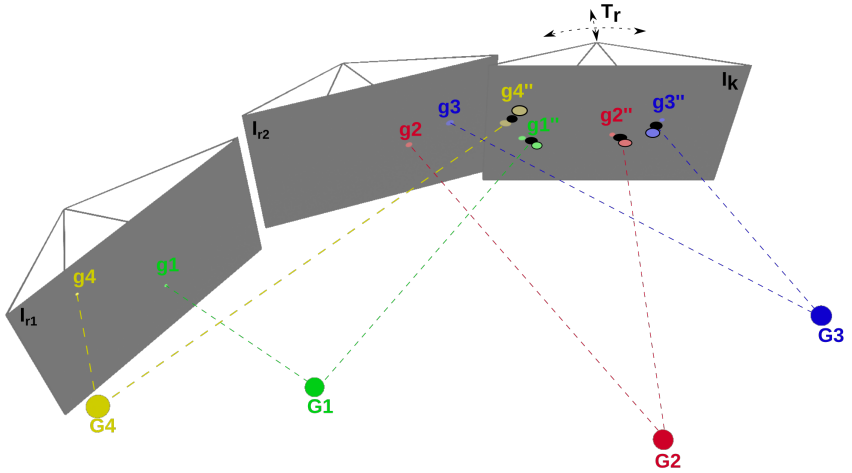
The SVO paper [8] describes this step in section IV.A. By performing the sparse image alignment, we get a first estimate of the pose. However, because we estimate the current pose by using the previous frame, we get a drift in

the long term. Therefore, we need to refine the pose by using the keyframes as reference. This is what we describe in the next section. In section 4.4 we describe sparse image alignment in more detail.

2.4 Pose Refinement



(a) Optical Flow: 2D keypoint position g'' found with optical flow from keyframe $I_{r1,r2}$ to current frame I_k



(b) Pose Refinement: Minimize re-projection error between g'' found by optical flow and projection with T_r

Figure 2.3: Pose Refinement

The refinement step is necessary to reduce the drift over time by using the keyframe where we first saw the keypoint. We use the Inverse Compositional Lucas Kanade algorithm to find an updated position of the point in the current image. We show this step in figure 2.3a. By using the pose from sparse image alignment, we get an estimate of the 2D point position in the current image. After that, we use optical flow to get a more accurate 2D point position. Then we start to optimize the pose by minimizing the re-projection error shown in equation 2.3. We show this process in figure 2.3b.

$$E_r = \min_P \Big(\sum_{a \in A | x' = a_x, y' = a_y} ((x' - x(P))^2 + (y' - y(P))^2) \Big) \tag{2.3}$$

x', y' : 2D Point position found by optical flow
x, y : 2D Point projected from 3D (see 2.1)
P : Projection matrix
A : For all keypoints
E_r : Remaining re-projection error

Chapter 4 describes optical flow and Lucas Kanade in more details. Chapter 5 describes how we calculate the gradient for minimizing the re-projection error in equation 2.3.

2.5 Keyframe Insertion

In every frame we should see one keypoint for each grid. Because some keypoints are weak or do not project to the current frame, we normally have less keypoints available for pose estimation. If the number drops below 66% of the theoretic count, we insert a new keyframe. The process for adding a new keyframe is described in section 2.2.

2.6 Update 3D Points

The update of 3D points is different from what SVO [8] describes. The reason is again that we can measure the keypoint depth in the first frame by using the stereo image. SVO uses a bayesian filter approach to estimate the keypoint depth over time. Instead, we use a mechanism to detect outliers and a simple Kalman Filter to update the 3D position of 3D keypoints.

2.6.1 Outlier Detection

We base outlier detection on two steps. First, we check the intensity difference returned by optical flow during pose refinement. If it exceeds a value

of 20, which was determined empirically, we drop the corresponding point from the point cloud. This can happen if points get occluded by objects and are therefore no longer visible in the new image. Another mechanism that does not mark a point as an outlier, but masks it during refinement, is when a point moves more than 9 pixels. This happens for points in areas with low texture. We do not use such points during refinement. However, such points can still help to improve the position during sparse image alignment. We use a second mechanism to detect outliers after pose refinement. We use the stereo image and calculate the disparity for a keypoint $d_{current}$. Then we transfer the global 3D keypoint to local coordinates. By calculating $d_{expected} = z_{local}/baseline$, we get the expected disparity. We consider the estimate as an inlier if the difference is lower than 2.5 pixel $abs(d_{current} - d_{expected}) < 2.5$. We take the standard deviation of 0.5 pixel for disparity and include 99.99%, which corresponds to 5 Tau. If a point has more outliers than inliers, we remove it from the keypoints. This mechanism eliminates weak keypoints with low contrast. Chapter 3 will explain the term disparity in more detail.

2.6.2 Point Refinement

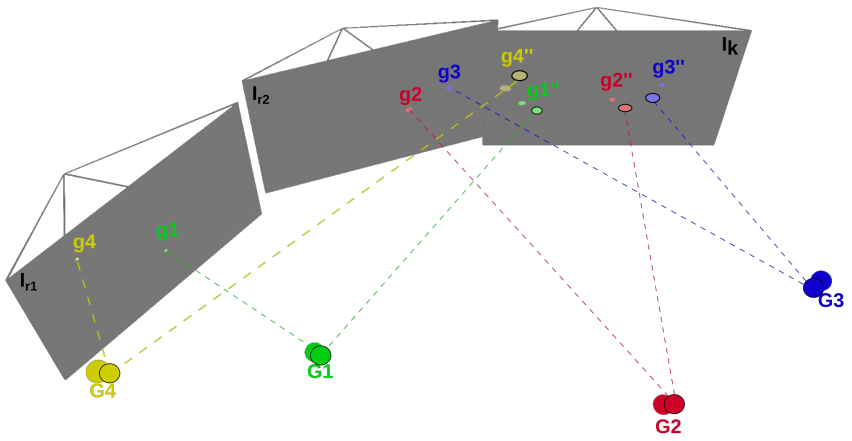


Figure 2.4: 3D Point Refinement
Change 3D keypoint position G to minimize distance between point from optical flow g'' and point from projection

In section 2.4 we refined the keypoint position in the image with optical flow. We then refined the pose by minimizing the re-projection error. If we use the refined pose to project the points to the image, we still have a small error between these points and the points from optical flow. This

happens because of some inaccuracy in the depth estimation. We use the movements of the camera that are bigger than the baseline of the camera in order to improve the accuracy of the 3D point cloud (figure 2.4). To find a new estimate for the 3D point we do a triangulation [6] based on the pose of the keyframe and the pose of the current frame. We update the 3D points by using a Kalman filter with an update and state equation shown in 2.4. We use the same definitions for the Kalman filter as [11]. The larger the movement, the smaller the error of our depth estimation. For the first measurement, we know that the movement has the length of the baseline of the camera. This means that instead of $\sqrt{\Delta x^2 + \Delta y^2}$, we just use the baseline in meter. The disparity has a close to normal distribution with a standard deviation of 0.5 pixel. For this reason, we use the inverse depth to update our Kalman filter. The standard deviation of 0.5 pixel results from the fact that we have a maximum resolution for disparity of one pixel. The variance for new information w should be a small number close to zero.

$$\begin{aligned} z^{-1}(t) &= z^{-1}(t-1) + w(t) \\ y(t) &= z^{-1}(t) + v(t) \\ w(t) &= 0.001 \\ v(t) &= (\frac{0.5}{\sqrt{\Delta x^2 + \Delta y^2}})^2 \end{aligned} \tag{2.4}$$

$\Delta x, \Delta y$: Movement of the camera

w : New information

v : Measurement error/noise

z : New point depth in keyframe local coordinates

Chapter 3

Depth Estimation

To find the depth at a certain point (x,y) we match the intensities of the left and right images based on a patch of pixels (e.g. a square of 31x31 pixel) [6]. The algorithm matches the right image at position (x,y) to the left image at the same position. Because we know the stereo camera is vertically aligned, it then increases x for the right image and matches again. We repeat this step up to the maximum allowed disparity (e.g. 50 pixels). The position with the smallest intensity difference between left and right is what we call disparity. An object far away has less disparity than an object close to the camera. This is what we show in figure 3.1. We use an empiric patch size of 31x31 pixel for the Econ Tara at 752x480.

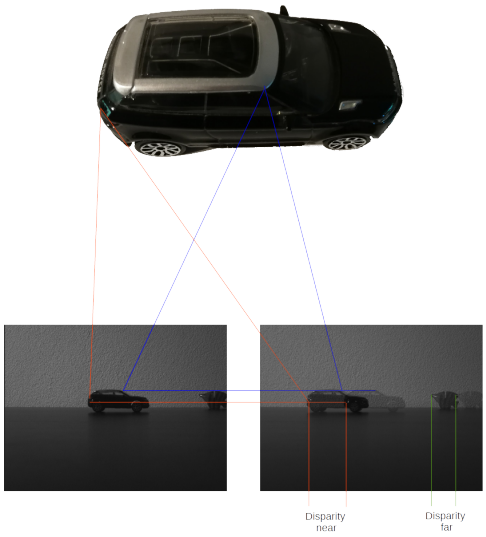


Figure 3.1: Sparse Disparity
The 3D object (top) shown as left image (bottom left) and right image (bottom right)
The right image includes a transparent copy of the left image. The disparity is the distance of a point in the left image to the same point in the right image.

From the disparity we can calculate the depth by using equation 3.1.

$$Z = \frac{f_m \cdot b}{s_{px} \cdot d} = \frac{f_x \cdot b}{d} \tag{3.1}$$

Z : Depth/Distance from camera
f_x : Focal length in pixel
d : Disparity
b : Baseline in meter
f_m : Focal length in meter
s_{px} : Pixel size in meter

Then we need to calculate the X and Y position in local camera coordinates. We show this in equation 3.2.

$$\begin{aligned} X &= Z \cdot \frac{x - c_x}{f_x} \\ Y &= Z \cdot \frac{y - c_y}{f_y} \end{aligned} \tag{3.2}$$

X, Y, Z : Point coordinates with local camera coordinates
f_x, f_y : Focal length in pixel
c_x, c_y : Image center (where z axis goes through)
x, y : 2D point in image

Next we transform the 3D point from the camera's local coordinate system to the global coordinate system by using the camera pose. We show this in equation 3.3. Note that we set camera pose = extrinsic camera matrix. This can be different in other documents and implementations.

$$\begin{pmatrix} X' \\ Y' \\ Z' \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}^{-1} \left(\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} - \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \right) \tag{3.3}$$

X, Y, Z : Point coordinates with local camera coordinates
X', Y', Z' : Global point coordinates
r_{ij} : Camera rotation parameter
$t_{x,y,z}$: Camera transition

Finally, we use these coordinates to create and maintain our point cloud. If we know the physical characteristics of our camera, we can calculate the camera properties in equation 3.1 (f_x and b). If the characteristics are unknown, we can use OpenCV to calibrate the camera [17] with a checkerboard.

Chapter 4

Optical Flow

In this chapter, we describe Lucas Kanade Optical Flow in more detail. This algorithm searches the movement of one or several points from one image to another. Lets assume that we have an image I1 at time t-1 and an image I2 at time t. We know the position of a point P1 in image 1. We use Lucas Kanade Optical Flow to find the position P2 of the point in image 2. Figure 4.1 shows the problem we want to solve. We call the position change of the point "Optical Flow".

Figure 4.1: Optical Flow
Where did the point at position P1 in image I1 move in image I2?

We divide this chapter into four sections. The first section describes Lucas Kanade in an intuitive but oversimplified way. Section two describes the original Lucas Kanade algorithm while section three focuses on the Inverse Compositional Lucas Kanade algorithm. In the last section we see an extension to guess a 3D pose. We use this method in the sparse image alignment step.

4.1 Lucas Kanade Intuitive

Lucas Kanade Optical Flow identifies where a point moves between two images, as shown in figure 4.1. It does that efficiently by using intensity gradients. This section describes how this works in an intuitive but simplified way, while the next section describes it mathematically. In this section, we use the term "intensity gradient" for gradients that are defined by a Sobel

filter and we call the gradient pointing towards the direction of movement "position gradient".
Let us assume the simplest scenario where we can only move in one direction (left or right). Figure 4.2 shows this simple scenario. Let us first focus on the case where we have a small movement 4.2a. We calculate the intensity gradients in the template for a small patch of 1x5 pixel (first row). Then we calculate the intensity difference (last row) between the current image (second row) and the first image. Next we multiply the intensity gradient by the intensity difference, which gives us the direction in which we have to move (position gradient). Here we assume that moving the current patch to left means - and moving it to right +. Now we move in the position gradients direction and calculate the intensity difference again. If it increases, we moved too much and we have to reduce the step size. On the contrary, if it decreases, we found a better position. We can now start a new iteration by calculating the intensity difference at the new position of the patch. After several iterations, the intensity change from one iteration to the other will be minimal. We found the horizontal movement of our patch.
Figure 4.2b shows that we cannot track points that moved a lot. If the patch in the reference image does not connect to where the patch moved, we cannot find the correct gradient. To solve this problem we down-sample the image as shown in figure 4.2c. We again find a valid gradient. For optical flow we use image pyramids [6], which store down-sampled images by factor 1,2,4,8,.... We then optimize at the lowest level. If we converge, we move one level up, etc. With this idea, it is possible to track movements over longer distances.

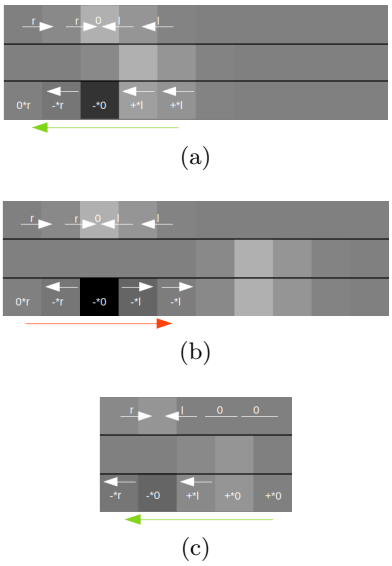


Figure 4.2: Optical Flow Intuitive see table 4.1 for description

Image a	Small move where we find the correct gradient green
Image b	Bigger move where we find an invalid gradient red
Image c	Bigger move as in b down-sampled in horizontal direction
First row	Reference image with gradients
Second row	Current image
Last row	Diff image (current-reference)
Color	Black -128, White +128
l	Gradient points left
r	Gradient points right
+	Current pixel - reference pixel > 0
-	Current pixel - reference pixel < 0
Multiply	-·l=r,-·r=l
Long arrow	Sum of gradients (e.g. l+l+l=l, r+r+l=r)

Table 4.1: Optical Flow Intuitive Description

We extend the above scenario from a one-dimensional movement to a two-dimensional movement as shown in figure 4.3. We use a patch size of 3x3 pixels. This time, we also calculate the gradient in the vertical direction. By doing so, we can track movements in x and y direction.

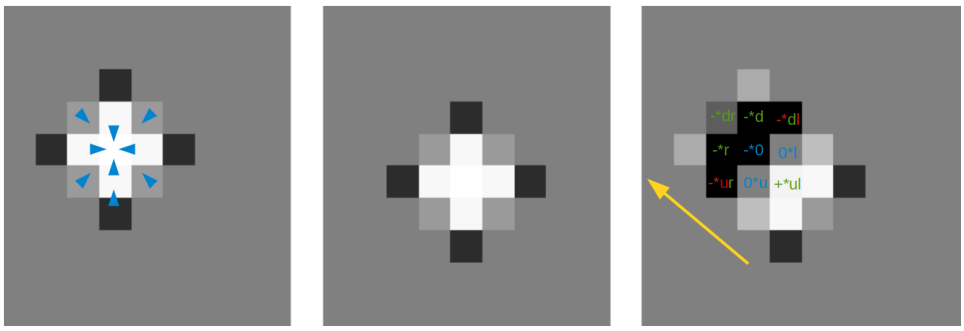


Figure 4.3: Optical Flow Intuitive 2D see table 4.2 for description

First column	Reference image with gradients
Second column	Current image
Third column	Diff image (current-reference)
Color	Black -128, White +128
+	Current-previous intensity > 0
-	Current-previous intensity < 0
u	Gradient points up
d	Gradient points down
l	Gradient points left
r	Gradient points right
Yellow arrow	Final gradient

Table 4.2: Optical Flow Intuitive 2D Description

We showed that optical flow can track two dimensional movements. However, it does not track rotations and shearing effects. This will introduce dependencies between x and y movements, which are hard to show. We therefore describe the further derivations mathematically.

4.2 Lucas Kanade Mathematical

In the previous section, we showed the basic idea of optical flow. It is oversimplified and can only track movements in x and y directions. In this section, we extend the idea to more complicated movements. Instead of linear movements in x and y directions, we want to find a warp matrix which describes an affine transformation of a patch from one image to another (equation 4.1).

$$P = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \end{pmatrix} \tag{4.1}$$

P : Warp matrix

p_{ij} : parameter of warp matrix

This warp matrix describes movements along the x and y axis(p_{13}, p_{23}), a rotation $(p_{11}, p_{12}, p_{21}, p_{22})$ and a shearing (p_{12}, p_{22}). The goal of Lucas Kanade is to find this warp matrix. It does that by minimizing the intensity difference between a patch of pixels in the template (reference image) and a patch of pixels with the same size in the current image as shown in equation 4.2.

$$E_p = \min_P (\sum_x^m \sum_y^n (I_k(x', y') - I_{k-1}(x(P), y(P)))^2) \tag{4.2}$$

m, n : Patch size
x, y : Pixel position in template
x', y' : Pixel position in current image
I_{k-1} : Template image
I_k : Current image
E_p : Remaining photometric error

We describe the transformation of a pixel from the reference image to the current image as a matrix multiplication as shown in equation 4.3. Because 4.2 is a non-linear problem, we have to solve it with a non-linear solver (Gradient Descent). The problem is solved by iteratively changing the warp matrix in direction of the gradient until the difference of the intensities are minimal:

$$\begin{aligned} \begin{pmatrix} x' \\ y' \end{pmatrix} &= W(x; P + \Delta P) \\ => \begin{pmatrix} p_{11} + \Delta p_{11} & p_{12} + \Delta p_{12} & p_{13} + \Delta p_{13} \\ p_{21} + \Delta p_{21} & p_{22} + \Delta p_{22} & p_{23} + \Delta p_{23} \end{pmatrix} &\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \\ => \begin{pmatrix} x(p_{11} + \Delta p_{11}) + y(p_{12} + \Delta p_{12}) + p_{13} + \Delta p_{13} \\ x(p_{21} + \Delta p_{21}) + y(p_{22} + \Delta p_{22}) + p_{23} + \Delta p_{23} \end{pmatrix} & \end{aligned} \tag{4.3}$$

W : Warp function
Δp_{ij} : change of p_{ij} per iteration

After each iteration we set $P = P_{n-1} + \Delta P$, where P stands for all p_{ij}. We can solve this problem by approximating the gradient heuristically. However, this is slow. Therefore, we need a direct computational approach to find the gradient. This is what Lucas Kanade describes.
Equation 4.2 describes the Lucas Kanade problem, which is non-linear. It is not possible to find ΔP, our gradient, analytically. Therefore, we need to linearise the problem by doing a first order Taylor approximation, as shown in equation 4.4 [19]. ∇I is the exterior derivative when using the chain rule, while $\frac{\sigma W}{\sigma P}$ is the inner derivative.

$$E = \sum_{x}^{m} \sum_{y}^{n} (I_k(x'_{i-1}, y'_{i-1}) + \nabla I_k \frac{\sigma W}{\sigma P} \Delta P - I_{k-1}(x, y))^2 \tag{4.4}$$

∇I_k : Intensity gradient in current image at position x',y'
E : Error

We want to find ΔP. To do so, we minimize the equation 4.4 with respect to ΔP by setting its derivative to zero:

$$\frac{\sigma E}{\sigma \Delta P} = 2 \sum_{x}^{m} \sum_{y}^{n} \left[\nabla I_k \frac{\sigma W}{\sigma P} \right]^T \left[I_k(x'_{i-1}, y'_{i-1}) + \nabla I_k \frac{\sigma W}{\sigma P} \Delta P - I_{k-1}(x, y) \right] = 0 \tag{4.5}$$

We now solve 4.5 for ΔP.

$$\Delta P = (\sum_{x}^{m} \sum_{y}^{n} \left[\nabla I_k \frac{\sigma W}{\sigma P} \right]^T \left[\nabla I_k \frac{\sigma W}{\sigma P} \right])^{-1} \sum_{x}^{m} \sum_{y}^{n} \left[\nabla I_k \frac{\sigma W}{\sigma P} \right]^T \left[I_{k-1}(x, y) - I_k(x'_{i-1}, y'_{i-1} \right] \tag{4.6}$$

We find $\frac{\sigma W}{\sigma P}$ by using equation 4.3 and do a partial derivation for each p. This gives us the Jacobian matrix shown in equation 4.7.

$$\frac{\sigma W}{\sigma P} = \begin{pmatrix} x & y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & y & 1 \end{pmatrix} \tag{4.7}$$

We know all parameters in equation 4.6 and can calculate ΔP. However, the following part of equation 4.6 is time consuming to calculate:

$$H^{-1} = (\sum_{x}^{m} \sum_{y}^{n} \left[\nabla I_k \frac{\sigma W}{\sigma P} \right]^T \left[\nabla I_k \frac{\sigma W}{\sigma P} \right])^{-1} \tag{4.8}$$

Where H is the Hessian matrix. It is the same formulation used by Gauss-Newton optimization. Therefore, this is the optimization method of choice for finding the minima. We need to calculate the Hessian matrix for each iteration. Because this is time consuming, we normally use the Inverse Compositional Lucas Kanade algorithm. It solves the problem by warping the template to the current image instead of the current image to the template. We see that in the next section.

4.3 Inverse Compositional Lucas Kanade

The problem which the Inverse Compositional Lucas Kanade algorithm solves is the same as before [1]. However, we switch the role of the template and the image as shown in equation 4.9.

$$E_p = \min_P (\sum_{x}^{m} \sum_{y}^{n} (I_{k-1}(x''(P), y''(P)) - I_k(x', y'))^2) \tag{4.9}$$

m, n : Patch size
x'', y'' : Warped pixel position in the template
x', y' : Pixel position in the current image
I_{k-1} : Template image
I_k : Current image
P : Warp matrix see 4.1
E_p : Remaining photometric error

This time, we warp the template with ΔP, as shown in equation 4.10.

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = W(x; \Delta P)$$
$$=> \begin{pmatrix} 1+\Delta p_{11} & 0+\Delta p_{12} & 0+\Delta p_{13} \\ 0+\Delta p_{21} & 1+\Delta p_{22} & 0+\Delta p_{23} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{4.10}$$
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

x, y : Original pixel position in the template
x'', y'' : Warped pixel position in the template
p_{ij} : Parameter of warp matrix
Δp_{ij} : Change of p_{ij} per iteration

We separate the update step from the warping of the image. We now show the update step in equation 4.11. Note that the update step is not additive like for Lucas Kanade. We can argue this is because the new warp depends on the old warps rotation. We can express this with the matrix multiplication shown in equation 4.11.

$$W(x; p) = W(W(x; \Delta P); p)$$
$$=> \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \end{pmatrix} \begin{pmatrix} \Delta p_{11} & \Delta p_{12} & \Delta p_{13} \\ \Delta p_{21} & \Delta p_{22} & \Delta p_{23} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{4.11}$$

W : Warp function
ΔP : Update of warp matrix

From equation 4.9 we need to do the first order Taylor approximation as we did for the normal Lucas Kanade algorithm (equation 4.12).

$$E = \sum_x^m \sum_y^m (I_{k-1}(x, y) + \nabla I_{k-1} \frac{\sigma W}{\sigma P} \Delta P - I_k(x', y'))^2 \tag{4.12}$$

∇I_{k-1} : Intensity gradient in template at position x,y

E : Error

We derivate equation 4.12 with respect to ΔP and set it to zero to minimize the problem (equation 4.13).

$$\frac{\sigma E}{\sigma \Delta P} = 2\sum_{x}^{m}\sum_{y}^{n}\left[\nabla I_{k-1}\frac{\sigma W}{\sigma P}\right]^{T}\left[I_k(x'_{i-1}, y'_{i-1}) + \nabla I_{k-1}\frac{\sigma W}{\sigma P}\Delta P - I_{k-1}(x,y)\right] \\ => 0 \tag{4.13}$$

This allows us to find the gradient ΔP (equation 4.14) for the non-linear optimization.

$$\Delta P = (\sum_{x}\sum_{y}\left[\nabla I_{k-1}\frac{\sigma W}{\sigma P}\right]^{T}\left[\nabla I_{k-1}\frac{\sigma W}{\sigma P}\right])^{-1}\sum_{x}^{m}\sum_{y}^{n}\left[\nabla I_{k-1}\frac{\sigma W}{\sigma P}\right]^{T}\left[I_k(x'_{i-1}, y'_{i-1}) - I_{k-1}(x,y)\right] \tag{4.14}$$

Compared to the normal Lucas-Kanade algorithm, the Hessian Matrix H is now constant for all iterations.

$$H = \sum_{x}^{m}\sum_{y}^{n}\left[\nabla I_{k-1}\frac{\sigma W}{\sigma P}\right]^{T}\left[\nabla I_{k-1}\frac{\sigma W}{\sigma P}\right] \tag{4.15}$$

Because we only have to calculate the Hessian matrix once, the Inverse Compositional Lucas Kanade is more efficient than the Lucas Kanade algorithm. We use the Inverse Compositional Lukas Kanade in SVO to calculate the optical flow for pose refinement. In the next section, we describe how we can use optical flow to find a 3D pose instead of a warp matrix. This is what we need for sparse image alignment in SVO.

4.4 Sparse Image Alignment

We use a slightly different optical flow method to get a first estimate of the pose in SVO. Instead of searching a warp matrix, we directly optimize a 3D pose. We start again with the idea of minimizing the intensity differences over a patch of pixels, as shown in equation 4.16.

$$E_p = \min_{P}(\sum_{a\in A}\sum_{x=a_x-\frac{m}{2}}^{m}\sum_{y=a_y-\frac{n}{2}}^{n}(I_{k-1}(x''(P), y''(P)) - I_k(x', y'))^2) \tag{4.16}$$

m, n : Patch size
x'', y'' : Projected pixel position in the template
x', y' : Pixel position in the current image
I_{k-1} : Template image
I_k : Current image
A : For all keypoints a
P : Projection matrix 4.17
E_p : Remaining photometric error

This time, we use 3D points instead of pixel positions. Therefore, we use a projection matrix instead of a warp matrix. The projection matrix P is shown in equation 4.17.

$$\begin{aligned} \begin{pmatrix} u \\ v \\ s \end{pmatrix} &= \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \\ &=> \begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \\ &=> P \cdot G \end{aligned} \tag{4.17}$$

u, v, s : Scaled pixel position
X, Y, Z : 3D Point position
x, y : 2D Point position in image
p_{ij} : Parameters of projection matrix P
r_{ij} : Rotation matrix
$t_{x,y,z}$: Translation
f_x, f_y : Focal length
c_x, c_y : Principal point
P : Projection matrix
G : 3D Point position (vector)

We need to find 12 gradients if we use the formulation in equation 4.18 for the inverse compositional Lucas Kanade. However, the rotational factors r_{nm} in equation 4.17 depend on each other. We could express the whole matrix based on 6 unknowns which are 3 times rotation and 3 times translation. It is easy to see that this formulation gets rather complex and that

it is untractable to find a Jacobian for such a matrix.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \frac{u}{s} \\ \frac{v}{s} \end{pmatrix} = \begin{pmatrix} \frac{p_{11} \cdot X + p_{12} \cdot Y + p_{13} \cdot Z + p_{14}}{p_{31} \cdot X + p_{32} \cdot Y + p_{33} \cdot Z + p_{34}} \\ \frac{p_{21} \cdot X + p_{22} \cdot Y + p_{23} \cdot Z + p_{24}}{p_{31} \cdot X + p_{32} \cdot Y + p_{33} \cdot Z + p_{34}} \end{pmatrix}$$

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = W(X, Y, Z; \Delta P) \tag{4.18}$$

$$=> \begin{pmatrix} \frac{u''}{s''} \\ \frac{v''}{s''} \end{pmatrix} = \begin{pmatrix} \frac{\Delta p_{11} \cdot X + \Delta p_{12} \cdot Y + \Delta p_{13} \cdot Z + \Delta p_{14}}{\Delta p_{31} \cdot X + \Delta p_{32} \cdot Y + \Delta p_{33} \cdot Z + \Delta p_{34}} \\ \frac{\Delta p_{21} \cdot X + \Delta p_{22} \cdot Y + \Delta p_{23} \cdot Z + \Delta p_{24}}{\Delta p_{31} \cdot X + \Delta p_{32} \cdot Y + \Delta p_{33} \cdot Z + \Delta p_{34}} \end{pmatrix}$$

Δp_{ij} : Update of projection parameters

ΔP : Update of projection matrix

We reformulate the problem by using Lie algebra [2]. We use the two terms SE(3) and se(3). SE(3) describes our "normal" projection matrices with a rotational and transitional part, as shown in equation 4.17. If we use Lie algebra, we write se(3) instead of SE(3). We move from se(3) to SE(3) by using the exponential map, and from SE(3) to se(3) by using the logarithm map. The problem we need to solve is the same as in equation 4.16. However, we need to find x'' and y'' differently. We use the following formulation:

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = h(e^{\xi} \cdot G)$$

$$\xi = \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{4.19}$$

h : Mapping function from u,v,s to x,y

e : Exponential map

ξ : Pose gradient as vector in se(3)

v_x, v_y, v_z : Position change in se3

$\omega_x, \omega_y, \omega_z$: Angle change in se3

ξ in 4.19 is the change in angle and position in se(3). We can transfer from se(3) back to SE(3) by using the exponential map e. The exponential map is the pendant to the Euler function for matrices. In our case, it has the closed form solution shown in 4.20 [6].

$$\omega_{skew} = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}$$

$$\begin{pmatrix} \Delta t_x \\ \Delta t_y \\ \Delta t_z \end{pmatrix} = I + (1 - cos(1))w_{skew} + (1 - sin(1))w_{skew}^2 \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{4.20}$$

$$\begin{pmatrix} \Delta r_x \\ \Delta r_y \\ \Delta r_z \end{pmatrix} = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}$$

$\Delta t_{x,y,z}$: Translational update step

$\Delta r_{x,y,z}$: Rotational update step

We use $\Delta t_x, \Delta t_y, \Delta t_z, \Delta r_x, \Delta r_y, \Delta r_z$ as our gradient to update the pose. We need to rotate the gradient by the current pose because we use the Inverse Compositional Lucas Kanade algorithm (see 4.3). We show the update step in equation 4.21.

$$\begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} + \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \begin{pmatrix} \Delta t_x \\ \Delta t_y \\ \Delta t_z \end{pmatrix}$$
$$\begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix} + \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \begin{pmatrix} \Delta r_x \\ \Delta r_y \\ \Delta r_z \end{pmatrix} \tag{4.21}$$

To find ξ, we again use the first order Taylor approximation as shown in equation 4.22.

$$E = \sum_x^m \sum_y^n (I_{k-1}(x, y) + \nabla I_{k-1} \frac{\sigma h(e^{\xi} G)}{\sigma \xi} \xi - I_k(x', y'))^2 \tag{4.22}$$

x, y : Original pixel position in template

∇I : Gradient in image I at position x',y'

E : Error

We again derivate equation 4.22 with respect to ξ and set it to zero, as shown in equation 4.23.

$$\frac{\sigma E}{\sigma \Delta P} = 2 \sum_x^m \sum_y^n \left[\nabla I_{k-1} \frac{\sigma h(e^{\xi} G)}{\sigma \xi} \right]^T \left[I_k(x'_{i-1}, y'_{i-1}) + \nabla I_{k-1} \frac{\sigma h(e^{\xi} G)}{\sigma \xi} \xi - I_{k-1}(x, y) \right] = 0 \tag{4.23}$$

Finally, we solve 4.23 for ξ, which leads us to equation 4.24.

$$\xi = (\sum_{x}^{m}\sum_{y}^{n}\left[\nabla I_{k-1}\frac{\sigma h(e^{\xi}G)}{\sigma\xi}\right]^{T}\left[\nabla I_{k-1}\frac{\sigma h(e^{\xi}G)}{\sigma\xi}\right])^{-1}\sum_{x}^{m}\sum_{y}^{n}\left[\nabla I_{k-1}\frac{\sigma h(e^{\xi}G)}{\sigma\xi}\right]^{T}\left[I_{k}(x'_{i-1}, y'_{i-1}) - I_{k-1}(x,y)\right] \tag{4.24}$$

We have to find a solution for $\frac{\sigma h(e^{\xi}G)}{\sigma\xi}$ to calculate ξ. This Jacobian matrix is shown in equation 4.25 (see [2] for derivation).

$$\frac{h(e^{\xi}G)}{\delta\xi} = \begin{pmatrix} f_x & f_y \end{pmatrix}\begin{pmatrix} \frac{1}{Z} & 0 & -\frac{X}{z^2} & -\frac{X\cdot Y}{Z^2} & 1+\frac{X^2}{Z^2} & -\frac{Y}{Z} \\ 0 & \frac{1}{Z} & -\frac{Y}{z^2} & -1-\frac{Y^2}{Z^2} & \frac{X\cdot Y}{Z^2} & \frac{X}{Z} \end{pmatrix} \tag{4.25}$$

Given the Jacobian and all 3D points, we can calculate the gradient for a given pose. We then iteratively optimize the pose based on the gradient as we did for the Inverse Compositional Lucas Kanade algorithm. We can use image pyramids to track bigger movements. Instead of one optimization, we do an optimization round for each pyramid level.

Chapter 5

Pose Refinement

We use pose refinement in SVO to decrease the drift over time and to increase the accuracy of the sparse image alignment. In a first step, we use Inverse Compositional Lucas Kanade to locate keypoints in the current image. Then we try to minimize the re-projection error shown in equation 5.1.

$$\begin{aligned} E_r &= \min_P \Big(\sum_{a \in A | x'=a_x, y'=a_y} ((x' - x(P))^2 + (y' - y(P))^2) \Big) \\ &=> \min_P \Big(\sum_{a \in A | g=a} ((g' - g(P))^T (g' - g(P))) \Big) \end{aligned} \tag{5.1}$$

x', y' : 2D Point position found by optical flow
x, y : 2D Point projected from 3D (see 2.1)
g, g' : x,y and x',y' as vector
P : Projection matrix
A : For all keypoints
E_r : Remaining re-projection error

We can achieve that with heuristic gradient descent but it is not efficient. Therefore, we use an approach similar to the one we used for optical flow. We try to find the gradient mathematically. Based on the camera model, we use the formulation shown in equation 5.2 for optimization. As we saw for pose estimation, we cannot use this formulation directly because we cannot find a Jacobian.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{u}{s} \\ \frac{v}{s} \end{pmatrix} = \begin{pmatrix} \frac{(p_{11}+\Delta p_{11}) \cdot X + (p_{12}+\Delta p_{12}) \cdot Y + (p_{13}+\Delta p_{13}) \cdot Z + (p_{14}+\Delta p_{14})}{(p_{31}+\Delta p_{31}) \cdot X + (p_{32}+\Delta p_{32}) \cdot Y + (p_{33}+\Delta p_{33}) \cdot Z + (p_{34}+\Delta p_{34})} \\ \frac{(p_{21}+\Delta p_{21}) \cdot X + (p_{22}+\Delta p_{22}) \cdot Y + (p_{23}+\Delta p_{23}) \cdot Z + (p_{24}+\Delta p_{24})}{(p_{31}+\Delta p_{31}) \cdot X + (p_{32}+\Delta p_{32}) \cdot Y + (p_{33}+\Delta p_{33}) \cdot Z + (p_{34}+\Delta p_{34})} \end{pmatrix} \tag{5.2}$$

u, v, s : Pixel position scaled with s
X, Y, Z : 3D Pixel position
p_{ij} : Projection parameters
Δp_{ij} : Projection parameter change per iteration

We formulate the problem with the help of Lie algebra, as shown in equation 5.3.

$$\begin{pmatrix} x \\ y \end{pmatrix} = h(e^{\xi} G)$$
$$\xi = \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{5.3}$$

P : Current projection matrix
e : Exponential map
ξ : Pose gradient as vector in se(3)
G : 3D point
h : Mapping function from u,v,s to x,y

If we formulate the problem of equation 5.1 with the knowledge of equation 5.3, we end up with equation 5.4.

$$\begin{aligned} E_r &= \min_P (\sum_n ((g' - g)^T (g' - g))) \\ &=> \min_\xi (\sum_n (g' - h(e^{\xi} G))^2) \end{aligned} \tag{5.4}$$

We do a Taylor approximation of the inner part as shown in equation 5.5.

$$E = (g' - h(e^{\xi} G) + \frac{h(e^{\xi} G)}{\xi}(\xi))^2 \tag{5.5}$$

Now we derive the Taylor approximation of equation 5.5, which is shown in equation 5.6.

$$\begin{aligned} &\frac{\sigma E}{\sigma \xi} = 2(\frac{h(e^{\xi} G)}{\xi})^T (g' - h(e^{\xi} G) + \frac{h(e^{\xi} G)}{\xi}(\xi)) = 0 \\ &=> (\frac{h(e^{\xi} G)}{\xi})^T (g' - h(e^{\xi} G)) + \frac{h(e^{\xi} G)}{\xi} \frac{h(e^{\xi} G)}{\xi} \xi \\ &((\frac{h(e^{\xi} G)}{\xi})^T (\frac{h(e^{\xi} G)}{\xi}))^{-1} (\frac{h(e^{\xi} G)}{\xi})^T (h(e^{\xi} G) - g') = \xi \end{aligned} \tag{5.6}$$

We use the same definition for the Jacobian as we did for Sparse Image Alignment in equation 4.25. To get the gradient in SE(3), we use equation 4.20. However, the resulting $\Delta t_x, \Delta t_y, \Delta t_z, \Delta r_x, \Delta r_y, \Delta r_z$ are already our update steps. We do not have to multiply the gradient with the current rotation matrix as we did in section 4.4.

Chapter 6

Implementation

In this chapter, we discuss how we implement the algorithm during this thesis. It should give a rough overview of the code base. All the source code written during this thesis is available on Github under the following link:
`https://github.com/eichenberger/stereo-svo-slam`
We organized the repository as follows:

Directory	Description
doc	Documentation (current document)
src	The source code
src/app	The source code for the test application
src/ar-app	The source code for the demo augmented reality app
src/lib	The source code for the library
src/python	The source code for the python wrapper and python demo
src/qt-viewer	The source code for the Qt 3D Viewer
test	Some test and helper scripts

Table 6.1: Repository organization

The source code uses GNU Makefiles for compilation. Table 6.2 shows the environment variables used to control the make process.

Variable	Description
OPENCV_INC_DIR	Include directory for OpenCV (opencv2/..)
OPENCV_LIB_DIR	Where to find the opencv libraries if not in standard path
QT_INC_DIR	Where to find the Qt include files if not in standard path
QMAKE	Which qmake should be used
CXX	Which compiler should be use
CXXFLAGS	Additional compiler flags (e.g. -O0/-O3)
LDFLAGS	Additional linker flags (e.g. additional library paths)

Table 6.2: Environment variables for GNU Make

The following make commands are available under the src directory:

```
# Compile library, app (test applicaiton) and ar-app (demo application)
make
# Compile the python wrapper
make python
```

```
# Compile the qt-viewer
make qt-viewer
```

All applications besides the qt-viewer accept some command line arguments. It is possible to query the help text by calling the application with -h:

```
./slam_app -h
Usage: ./slam_app [options] camera
SVO stereo SLAM application

Options:
  -h, --help                      Displays this help.
  -v, --video <video>             Path to camera or video (/dev/videoX,
                                  video.mov)
  -s, --settings <settings>       Path to the settings file (Econ.yaml)
  -r, --hidraw <hidraw>           econ: HID device to control the camera
                                  (/dev/hidrawX)
  -i, --hidrawimu <hidrawimu>     econ: HID device to control the imu
                                  (/dev/hidrawX)
  -e, --exposure <exposure>       econ: The exposure for the camera 1-30000
  -t, --trajectory <trajectory>   File to store trajectory
  -m, --move <move>               video: skip first n frames
  -d, --hdr                       econ: Use HDR video

Arguments:
  camera                          The camera type to use can be econ, video or
                                  euroc
```

Most applications require a camera settings file describing the camera parameters and algorithm settings. We provide an example file for Blender, including comments, under src/app/Blender.yaml.

6.1 Library

We write the stereo SVO Library (libstereosvo) as a shared library. We can use it in other projects by dynamically linking against it.

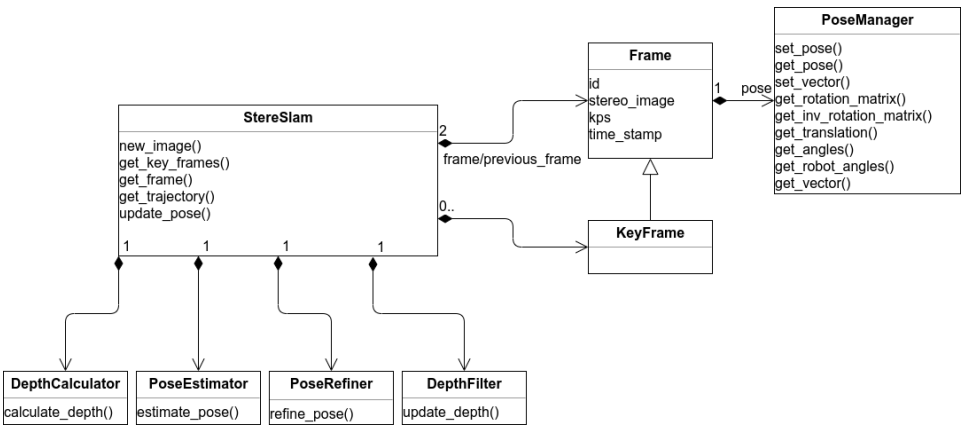


Figure 6.1: Class diagram for the Stereo SVO Library

In figure 6.1, we show the most important classes of the library. To start the SLAM system, we create an instance of StereoSlam. When creating the object, we need to specify the camera parameters (see software documentation in appendix). After that, we feed new images by calling the method new_image. The getter functions are used to get the current frame (with pose information), all keyframes (including keypoints), the trajectory or to update the pose.

6.1.1 Stereo SLAM

The Stereo SLAM class implements the basic algorithm described in chapter 2 and shown in figure 2.1. Figure 6.2 shows a slightly changed flow chart to align better with the naming in the implementation. It calls the depth calculator if new keyframes are needed, estimates the pose and updates the point cloud. However, it delegates the work to its sub-objects. It also provides an interface to receive information about the current state (see figure 6.1).

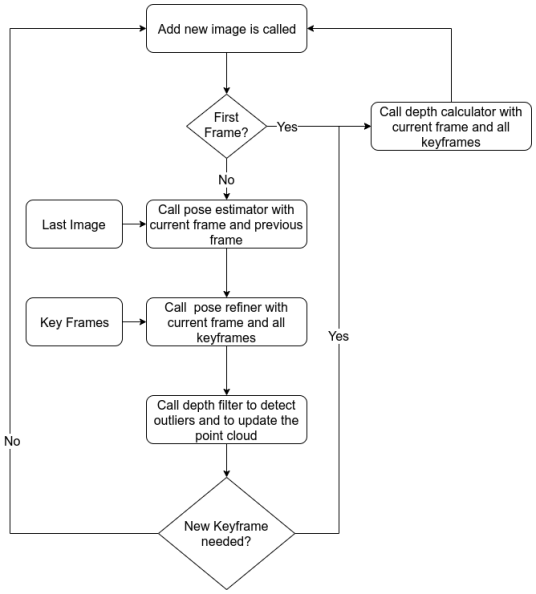


Figure 6.2: Stereo SLAM Flowchart

6.1.2 Depth Calculator

The depth calculator divides the image into grids and searches for new keypoints, merges keypoints from previous keyframes and estimates the depth. It is only used when a new keyframe is required. Figure 6.3 shows the flowchart for the depth calculator.

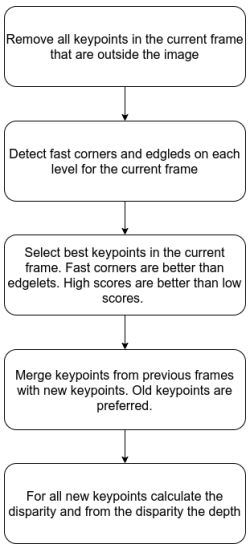


Figure 6.3: Depth Calculator Flowchart

6.1.3 Pose Estimator

The pose estimator does a first estimate of the pose, based on the previous image. It does that by using the sparse image alignment described in section 2.3. Figure 6.4 shows the flowchart for the pose estimator.

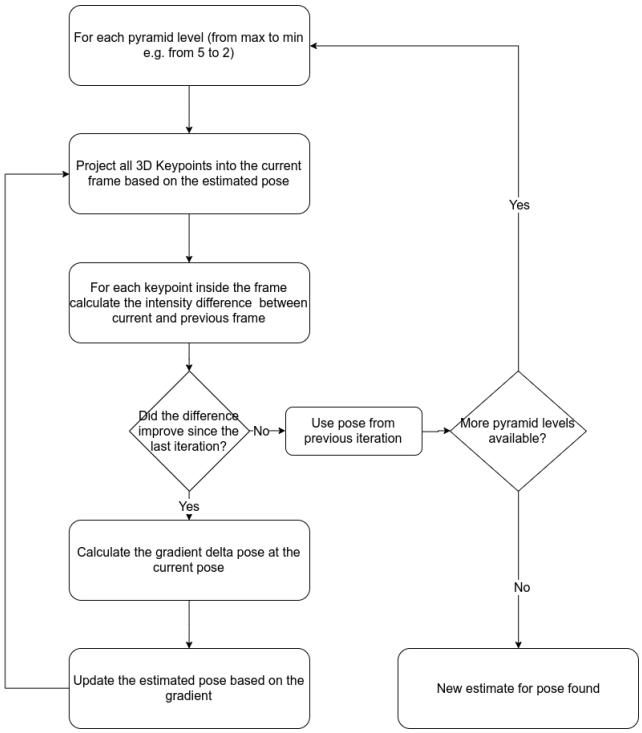


Figure 6.4: Pose Estimator Flowchart

6.1.4 Pose Refiner

The pose refiner updates the pose by using the keyframes in which a keypoint was first found. This reduces the drift over time. It uses the algorithm described in chapter 5. Figure 6.5 shows the flowchart of the pose refiner.

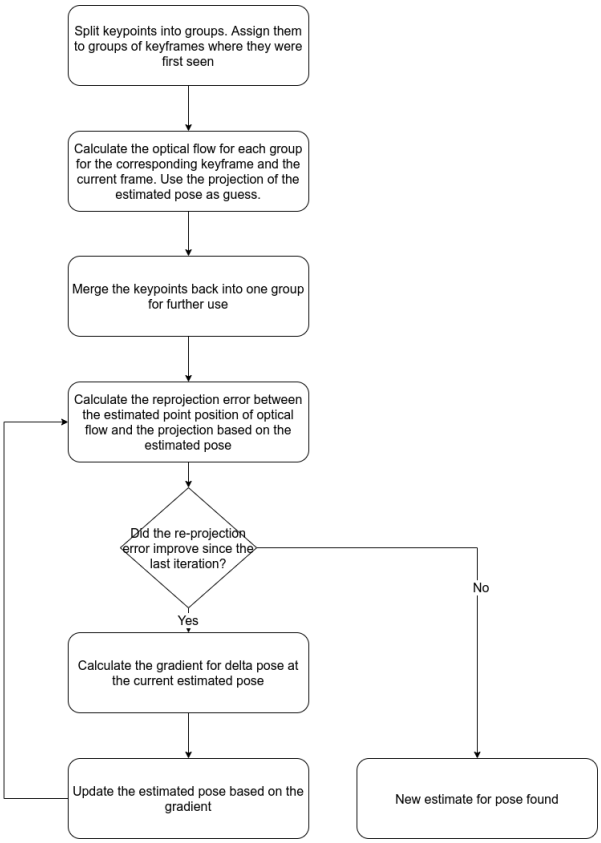
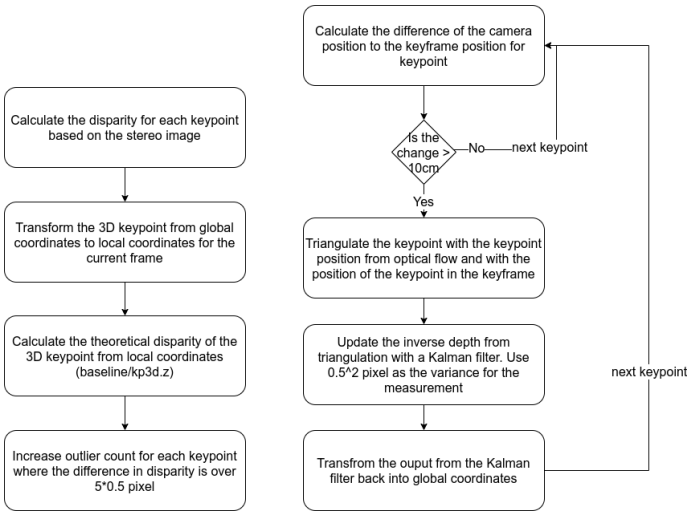


Figure 6.5: Pose Refiner Flowchart

6.1.5 Depth Filter

The depth filter removes outlier and updates the 3D point position of keypoints as described in section 2.6. Figure 6.6 shows two flows. The first flow in figure 6.6a describes the outlier detection, while the second in figure b describes the 3D point update.



(a) Outlier detecction (b) 3D Point Update

Figure 6.6: Depth Filter Flowchart

6.2 Test Application

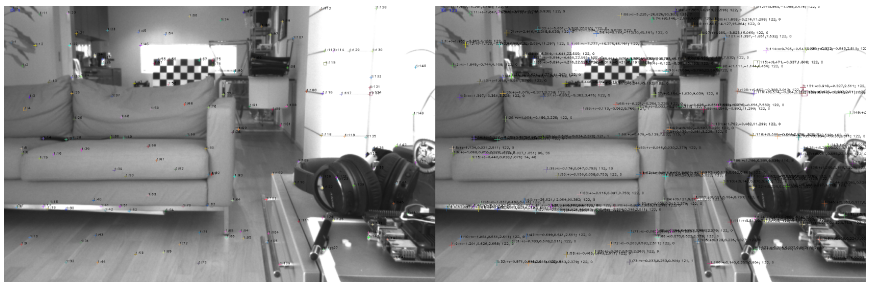


Figure 6.7: Test Application

We can use the test application to test and debug the algorithm. It allows us to use different input sources like video, EuRoC and camera. Figure 6.7 shows an example of the output of the application. The left side shows the last keyframe with its keypoints. The right image shows the current frame. In the keyframe we show two numbers a:b. a is the keyframe id where the

frame was first seen and b is the index of the keypoint in this keyframe. In the current frame it shows the information a:b:+/-:x,y,z;inliers,outliers. a and b are again the keypoint ids. + means that the keypoint is currently used, and - that it is not. x,y,z are the 3D position and the last two numbers indicate how many times the point was counted as an inlier or as an outlier. The numbers, in both images are small, but it is possible to zoom in to make them appear bigger. These numbers are interesting when debugging the application. By pressing a key other than q, we can freeze the output to analyse the current state. By pressing q we can quit the application.
Additionally to the above output, the test application offers a Websocket [20] backend. Through this backend, it is possible for the Qt Viewer to read 3D data like current pose, keyframes and trajectory. We show the Websocket idea in figure 6.8.

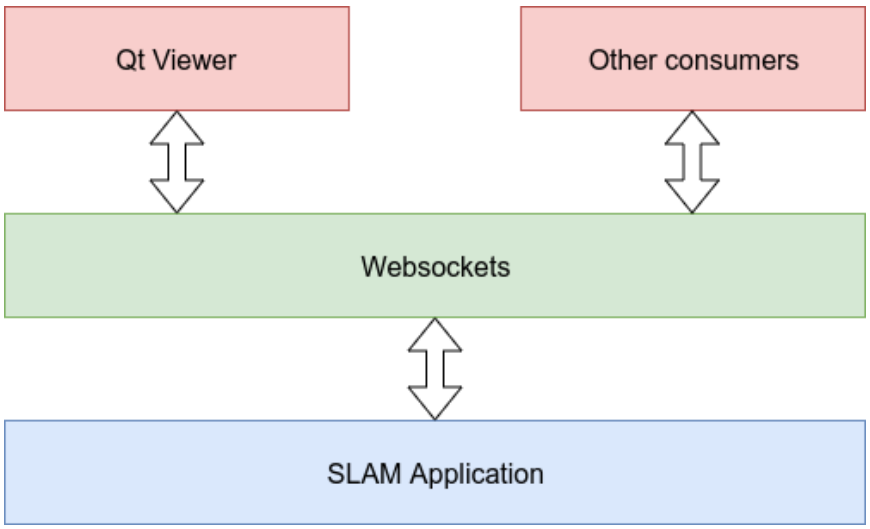


Figure 6.8: Websocket Connection

Table 6.3 shows the implemented Websocket URLs and commands. For trajectory it will send x1, y1, z1, rx1, ry1, rz1, x2, y2, z2, ...The Test Application will open a Websocket on port 8001.

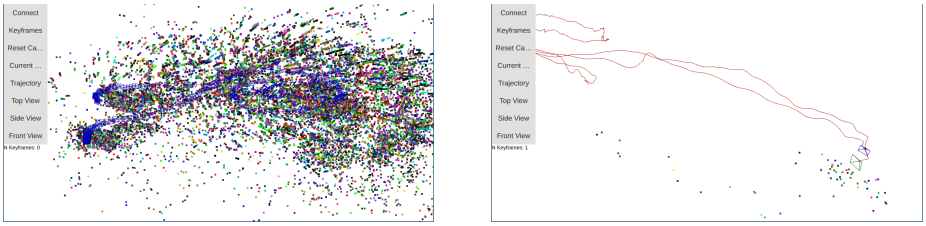
URL	Command	Response

keypoints	get	{
		”colors”:[{”b”:139,”g”:69,”r”:103},...],
		”keypoints”:[{”x”:-4.8,”y”:-0.9,”z”:6.1},...],
		pose”:{”rx”:0.4,”ry”:0.1,”rz”:-0.9,”x”:-1.8,”y”:0.2,”z”:0.1}
		}

pose	get	{”pose”:{”rx”:-9.1,”ry”:0.0,”rz”:-0.0,”x”:-1.2,”y”:0.0,”z”:-2.0}}
trajectory	get	{”trajectory”:[0,0,0,0,0,0,0,0.1,0.1,0.1,0.1,0.1,0.1,0.1,...]}

Table 6.3: Websocket commands

6.3 Qt Viewer

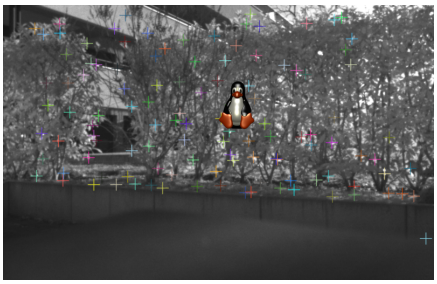


(a) Viewer with reduced keyframes (b) Viewer with all keyframes

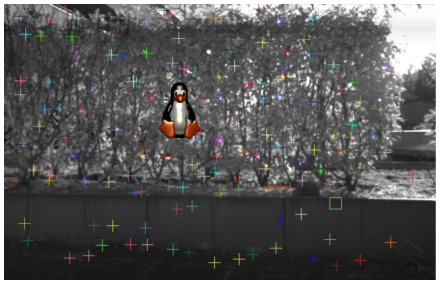
Figure 6.9: Qt Viewer

We can use the Qt Viewer shown in figure 6.9 to display 3D information calculated by the Test Application. We can navigate with the cursor to move trough the 3D room. The data is not received automatically. We need to click connect, get pose, etc. to trigger an action. This allows us to keep a snapshot of a scene and to analyse the current state. All points have the same color as displayed in the test application. Keyframes are shown as blue rectangles and the current frame pose is shown in green. The trajectory is shown as a red line. When pressing the connect button, it will automatically connect to localhost on port 8001, where the Test Application is listening on.

6.4 Demo application



(a) Demo application first view



(b) Demo application different view

Figure 6.10: Demo Application

The demo application shown in figure 6.10 is a simple augmented reality application that demonstrates one usecase. If we move the camera, the Tux should stay at the same place. It is possible to disable the painting of the keypoints by not providing the -p option when starting the application. Only the Econ camera is supported as input.

6.5 Additional tools

There are a few other tools available, but they will not be documented in more detail because they are not intended for common use.

6.5.1 Python wrapper

There is a Python wrapper under src/python which uses Cython to generate bindings. It allows us to use libstereoslam from Python. We provide a demo application as main.py. It offers a similar feature set as the Test Application but it is slower.

6.5.2 Scripts

There are several scripts under test which we can use, in particular to record videos from the econ camera in the right format, to plot results, to export trajectory from Blender, etc.

Chapter 7

Results

For testing the algorithm with a reference implementation we use two synthetic scenes generated by Blender. The advantage of having synthetic scenes is that we can test well defined movements where we know the trajectory. Blender allows to render stereoscopic videos by following the manual [4]. We can also customize the baseline and the focal length of the virtual camera. To export the trajectory into a CSV file, we can use the script test/blender-export-trajectory.py. We can then use this for comparison. The Test Application allows to export the trajectory of a scene by providing the -t argument.
We did all measurements in this chapter on an Intel i7-8550U processor. For our SVO implementation, we use the settings in src/app/Blender.yaml. In SVO and SVO Edglet, we adjusted fx,fy,cx and cy (see Blender.yaml). The rest is left unchanged. For ORB_SLAM2, we use the settings found in test/ORB-Blender.yaml.

7.1 Classroom

Figure 7.1: Blender classroom scene

For the first test, we use the Blender Classroom Scene [3]. We show a snapshot of this scene in figure 7.1. An image for the left and right camera is rendered to have a valid stereo image input. We change the position and angles in the scene as shown in figure 7.2. The test video is part of the repository under test/blender-classroom.mkv. The trajectory file is called test/blender-classroom.csv. We compare our implementation with SVO [8],

SVO edgelet [12] and ORB SLAM [16]. SVO edgelet additionally uses edglets, which are missing in the original SVO sources. Because SVO and ORB SLAM have dependencies to old libraries, we move the implementation to a Docker container. The Dockerfile is available in the test folder.

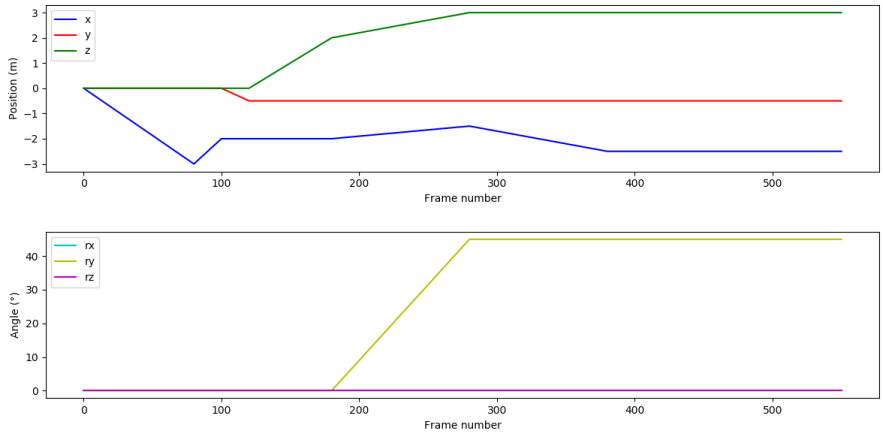


Figure 7.2: Blender classroom trajectory

Figure 7.2 shows the trajectory difference measured by our implementation, the SVO edgelet reference and the orignal SVO implementation. We scale both reference implementations to make a comparison possible. Table 7.1 shows the maximum error, table 7.2 the average error and table 7.3 the average frames per seconds when doing the measurement. Figure 7.3 shows our expectations. Monocular SLAM is considerably less accurate than Stereo SLAM. While ORB SLAM matches the trajectory almost perfectly, our implementation of SVO has a slight offset when tracking the transverse movement (figure 7.3a).

impl	err x	err y	err z	err rx	err ry	err rz
Our SVO	0.031	0.013	0.073	0.205	0.610	0.285
Edgled SVO	0.344	0.099	0.273	0.989	12.941	1.430
Original SVO	0.300	0.120	0.488	1.169	6.123	1.390
ORB SLAM	0.023	0.015	0.023	0.222	0.302	0.204

Table 7.1: Maximum errors in meter (x,y,z) and Degrees (rx,ry,rz)

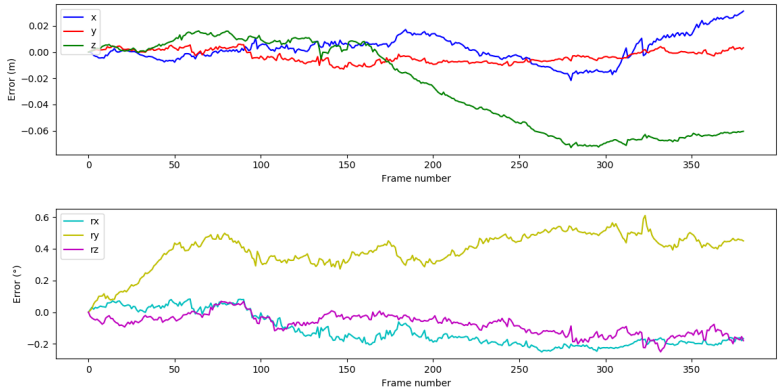
impl	err x	err y	err z	err rx	err ry	err rz
Our SVO	0.008	0.005	0.032	0.110	0.390	0.107
Edgled SVO	0.158	0.050	0.093	0.477	5.349	0.344
Original SVO	0.060	0.058	0.202	0.396	3.132	0.557
ORB SLAM	0.006	0.004	0.011	0.074	0.084	0.123

Table 7.2: Average errors in meter (x,y,z) and Degrees (rx,ry,rz)

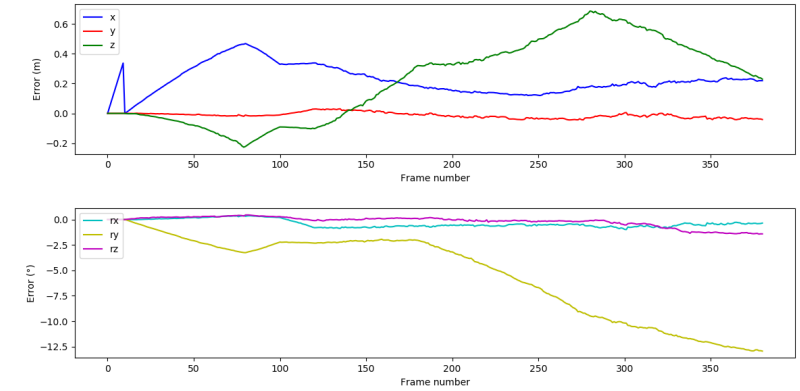
impl	FPS
Our SVO	52.38
Edgled SVO	0.93
Original SVO	115.81
ORB SLAM	17.38

Table 7.3: Frame rate

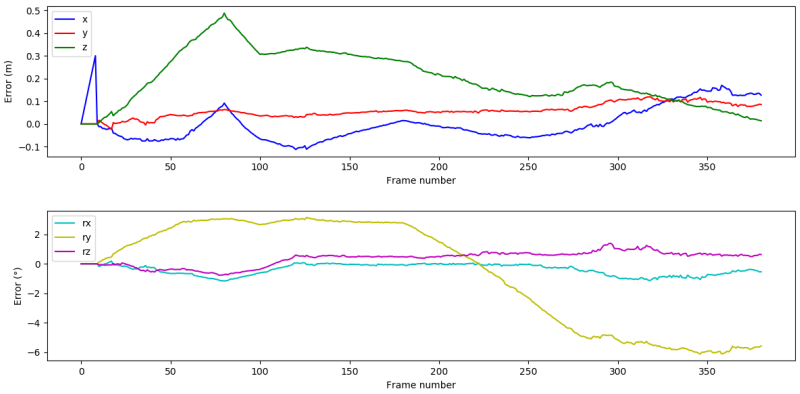
As we see in the results, tracking ry and z is especially challenging for monocular SLAM. This is expected because by changing ry we can often correct small movements along the x axis. Further, when moving along the z axis we are not able to add new 3D points because we do not have a change along the x or y axis



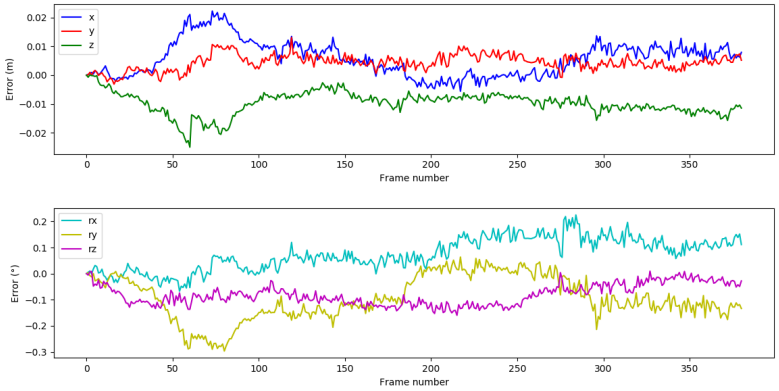
(a) Our Implementation



(b) SVO Edgled Implementation

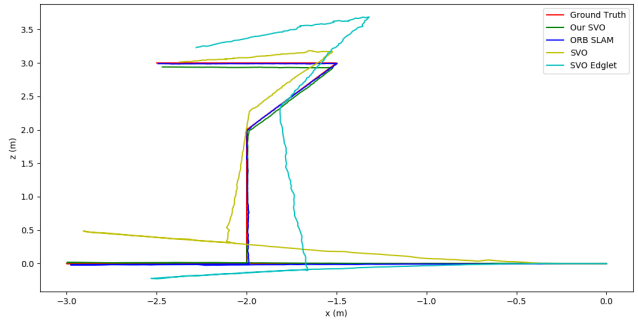


(c) SVO Original Implementation

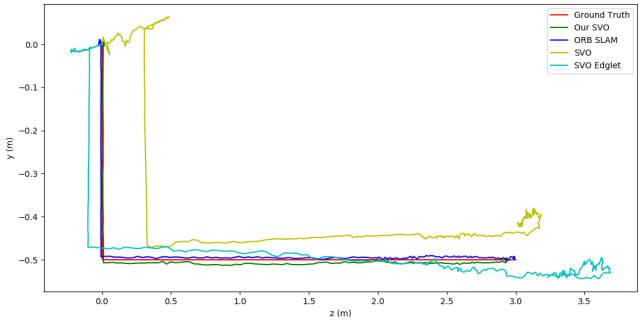


(d) ORB SLAM

Figure 7.2: Blender Classroom Scene Errors



(e) Top View



(f) Side View

Figure 7.3: Blender Classroom scene trajectory of our implementation, ORB, SVO and SVO Edglet

7.2 Barcelona

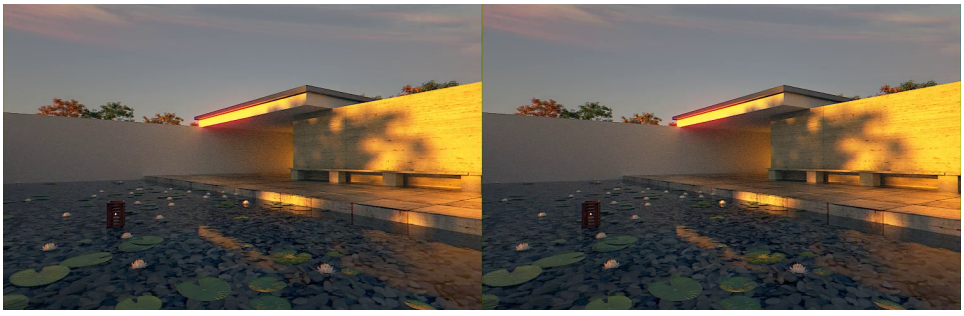


Figure 7.4: Blender Barcelona Scene

For the second test, we use an outdoor synthetic scene called Barcelona [3]. Here we compare ORB SLAM with our SVO implementation because the

monocular SLAMs are unable to track these movements. Figure 7.5 shows the planned trajectory. The challenges for the tracker are the movements along the z axis and the changes in the angle. We can find the test video in the repository under test/blender-barcelona.mkv and the trajectory under test/blender-barcelona.csv.

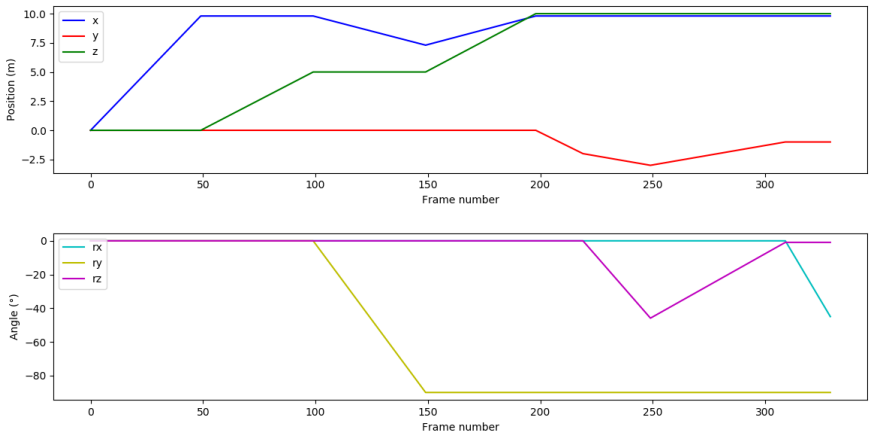


Figure 7.5: Blender Barcelona Trajectory

As we see in figure 7.6, both algorithms struggle for substantial angle changes without movement. When checking table 7.4 and table 7.5, we see that ORB SLAM performs slightly better but with a lower frame rate than our SVO SLAM implementation (table 7.6). Figure 7.7b shows that it is more delicate for our algorithm to track movements along the y axis.

impl	err x	err y	err z	err rx	err ry	err rz
Our SVO	0.622	0.537	0.956	10.417	11.222	14.682
ORB SLAM	0.993	0.189	0.592	7.491	19.315	10.542

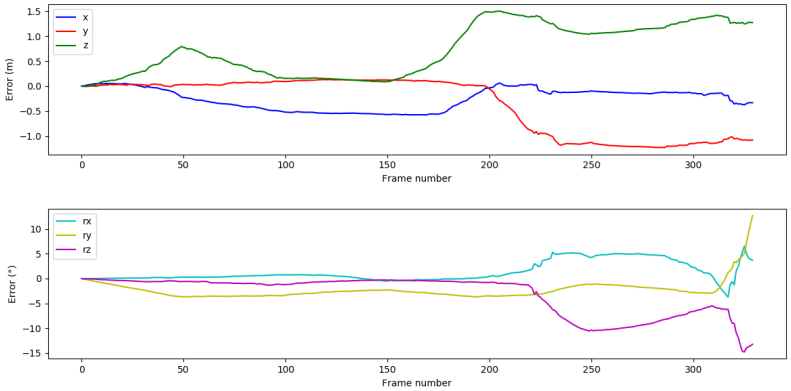
Table 7.4: Maximum error in meter (x,y,z) and Degrees (rx,ry,rz)

impl	err x	err y	err z	err rx	err ry	err rz
Our SVO	0.396	0.166	0.451	1.012	2.753	2.446
ORB SLAM	0.734	0.088	0.416	0.711	2.717	1.955

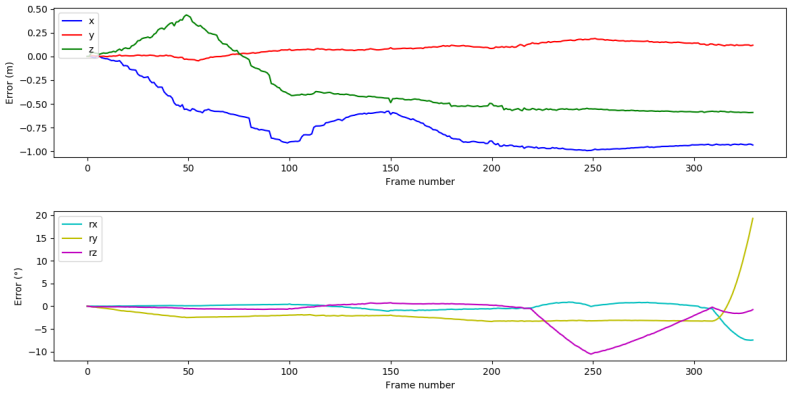
Table 7.5: Average error in meter (x,y,z) and Degrees (rx,ry,rz)

impl	FPS
Our SVO	39.56
ORB SLAM	22.53

Table 7.6: Frame rate

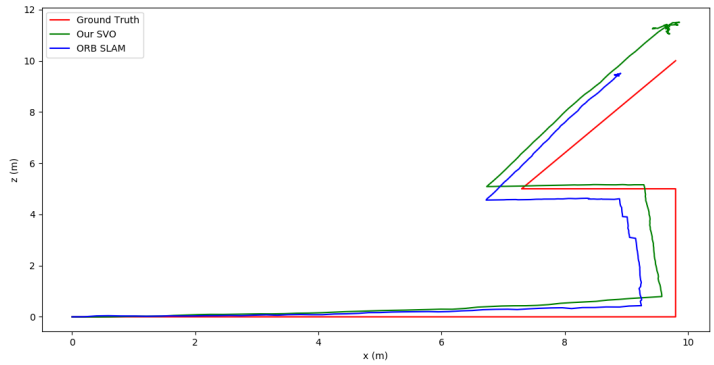


(a) Our implementation

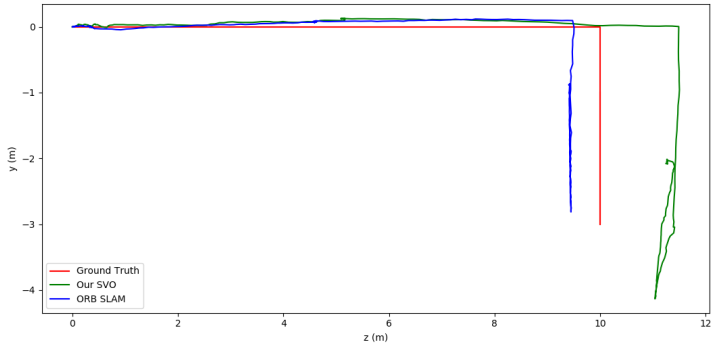


(b) ORB SLAM

Figure 7.6: Blender Barcelona Scene Errors



(a) Top View



(b) Side View

Figure 7.7: Blender Barcelona scene trajectory of our SVO implementation and ORB

7.3 EuRoC Machine Hall 2

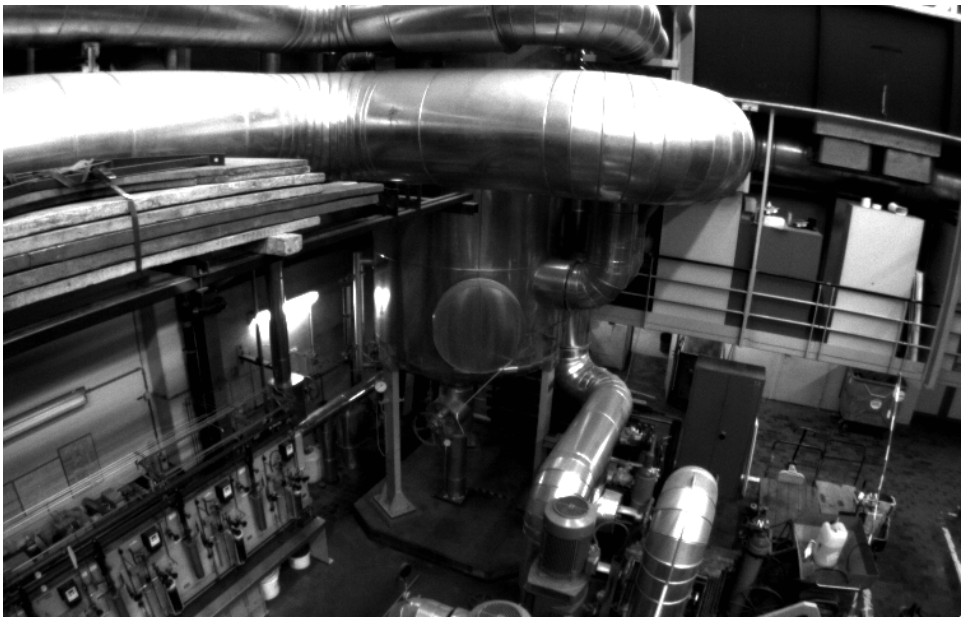
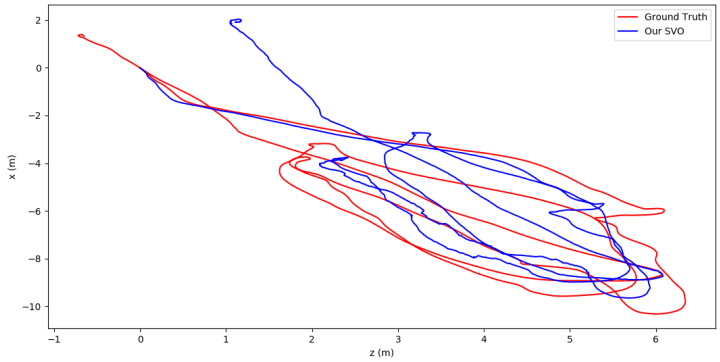
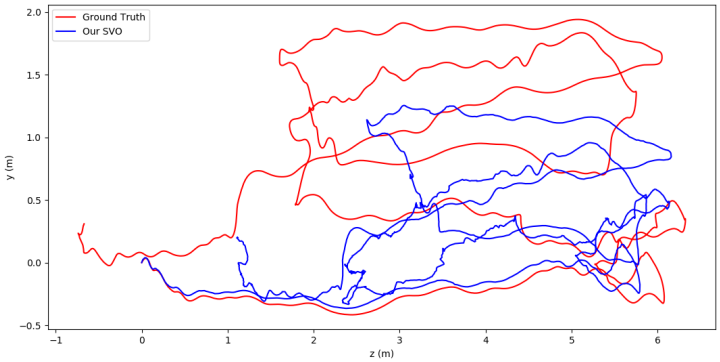


Figure 7.8: EuRoC Machine Hall 2 Scene

EuRoC is a widely used dataset for SLAM algorithms [5]. Figure 7.8 shows an image from EuRoC Machine Hall 2. The SVO2 paper [9] compares some measurements based on EuRoC. Our results are not in the range of those of other modern SLAM algorithms. The reason is that the current implementation does not do a relocation (local or global). Therefore, it cannot improve if it sees the same scene more than once, what increases the drift. For completeness, an initial result is shown anyhow for EuRoC Machine Hall 2.



(a) Top View



(b) Side View

Figure 7.9: EuRoC

As we see in figure 7.9, we have a huge drift over the whole sequence. We measure a final error of more than one meter in each direction. However, this was measured without doing a lot of parameters tuning. SVO2 states that it can achieve an RMSE of only 8 cm. SVO one, on which this thesis is based, cannot follow the sequence at all. It is possible that we could achieve better results with our implementation by tweaking the parameters but it is not expected to get close to what SVO2 measured. The average frame rate was 37.13 fps.

Chapter 8

Discussion

We showed that SVO works very effectively and is faster than e.g. ORB SLAM. Our implementation does not yet achieve the same frame rate as the reference implementation. However, by tweaking the source code, we should achieve higher frame rates. Some initial tests with OpenMP and TBB show that the single-threaded version is faster. The task size per iteration is too low, such that the overhead of multiprocessing is too large. By increasing the task size, we could probably improve this. However, what seams more important is to actively use vectorizations. By consistently using OpenCV vectors and matrices, the compiler can better optimize the code. We already gain at least a factor two by doing so. The depth filter implementation is the slowest part in the current implementation. However, it was also optimized the least. Therefore, we could gain the most by optimizing this part for the moment.

First tests on an ARM iMX8QM (Cortex A72) based system showed frame rates of 10 fps for our SVO implementation and 5 fps for ORB SLAM. However, here we didn't do any optimization or analysis yet. ARM NEON vectorizations is used by the compiler but we didn't do any function tracing on the library so far.

The accuracy of our implementation is lower than those of modern SLAM implementations. We could improve this by projecting old keypoints into the current frame to have some kind of local re-localization. We do not do that at the moment because of speed concerns and to keep the algorithm simple.

We implemented a 3D viewer, which we can use over a network connection. This allows us to debug devices that are further away e.g. via Wifi or Ethernet. The current experience shows that this is a good approach because we can also separate the code base.

We showed an easy way to generate test scenes by using Blender animations. This allows us to debug easily. We know all movements and can add strict movements in one direction. Further, it allows us to compare results between different algorithms and implementations. Generating scenes is simple, cheap and fast.

A demo application shows the possibilities of SLAM. It is a simple augmented reality application which we can use to showcase our algorithm.

The current library is in the state of a proof of concept. The implementation showed to be faster than ORB SLAM but it is not as robust yet. We can use the current state as a starting point for further development. The library is open source and has only dependencies to OpenCV. This makes it easier to compile it for other platforms and systems.

The CPVR lab currently uses ORB SLAM for its augmented reality projects. The benefit of SVO would only be performance-wise and not in terms of accuracy. Therefore, the effort to port its current projects to SVO does not seem to be justifiable. However, for future projects, a combination of SVO and ORB SLAM could be an efficient solution for mobile devices. It could use SVO for tracking between frames, while ORB SLAM could do relocation. Inserting new keyframes and computing descriptors would still be done by ORB SLAM but the pose estimation would only involve sparse image alignment and pose refinement without triangulation. This could speed up the frame processing by a factor two.

We conclude that SVO shows great potential for use on embedded devices. The current implementation would need more tuning to achieve higher frame rates. However, we already showed that SVO runs faster than indirect SLAM algorithms like ORB SLAM.

Bibliography

[1] Simon Baker and Iain Matthews. "Lucas-Kanade 20 Years On: A Unifying Framework". In: *International Journal of Computer Vision* 56.3 (Feb. 2004), pp. 221–255. ISSN: 1573-1405. DOI: 10.1023/B:VISI.0000011205.11775.fd. URL: https://doi.org/10.1023/B:VISI.0000011205.11775.fd.

[2] José-Luis Blanco. *A tutorial on SE(3) transformation parameterizations and on-manifold optimization*. Tech. rep. University of Malaga, 2010. URL: http://ingmec.ual.es/~jlblanco/papers/jlblanco2010geometry3D_techrep.pdf.

[3] Blender. *Blender Demo files*. accessed January 02, 2020. 2019. URL: https://www.blender.org/download/demo-files/.

[4] Blender. *Blender Stereo Camera*. accessed January 08, 2020. 2020. URL: https://docs.blender.org/manual/en/latest/render/output/multiview/usage.html#stereoscopy-setup.

[5] Michael Burri et al. "The EuRoC micro aerial vehicle datasets". In: *The International Journal of Robotics Research* (2016). DOI: 10.1177/0278364915620033. eprint: http://ijr.sagepub.com/content/early/2016/01/21/0278364915620033.full.pdf+html. URL: http://ijr.sagepub.com/content/early/2016/01/21/0278364915620033.abstract.

[6] Peter Corke. *Robotics, Vision and Control*. Springer. ISBN: 978-3-319-54413-7.

[7] Econ. *Tara Stereo Camera*. accessed December 27, 2019. 2019. URL: https://www.e-consystems.com/3D-USB-Stereokamera-de.asp.

[8] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. "SVO: Fast Semi-Direct Monocular Visual Odometry". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2014.

[9] Christian Forster et al. "SVO: Semi-Direct Visual Odometry for Monocular and Multi-Camera Systems". In: *IEEE TRO'17*. 2017.

[10] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing Third Edition*. Pearson. ISBN: 978-0-13-234563-7.

[11] Monson H. Hayes. *Statistical Digital Signal Processing And Modeling*. Wiley. ISBN: 0-47159431-8.

[12] HeYijia. *SVO Edgled*. accessed January 02, 2020. 2017. URL: https://github.com/HeYijia/svo_edgelet.

[13] Intel. *Intel RealSense*. accessed December 27, 2019. 2019. URL: https://realsense.intel.com/depth-camera/#D415_D435.

[14] Stereo Labs. *ZED Stereo Camera*. accessed December 27, 2019. 2019. URL: https://www.stereolabs.com/zed/.

[15] Point Cloud Library. *Point Cloud Library*. accessed January 03, 2020. 2019. URL: https://github.com/PointCloudLibrary.

[16] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. "ORB-SLAM: a Versatile and Accurate Monocular SLAM System." In: *CoRR* abs/1502.00956 (2015). URL: http://dblp.uni-trier.de/db/journals/corr/corr1502.html#Mur-ArtalMT15.

[17] OpenCV. *Camera Calibration*. accessed January 13, 2020. 2019. URL: https://docs.opencv.org/4.1.1/d9/d0c/group__calib3d.html.

[18] Miroslav Trajkovic and Mark Hedley. "Fast Corner Detection". In: *Image and Vision Computing* 16 (Feb. 1998), pp. 75–87. DOI: 10.1016/S0262-8856(97)00056-5.

[19] Wikipedia. *Taylor Series*. accessed December 27, 2019. 2019. URL: https://en.wikipedia.org/wiki/Taylor_series.

[20] Wikipedia. *WebSocket*. accessed January 06, 2020. 2019. URL: https://en.wikipedia.org/wiki/WebSocket.