Scene mapping using a RGB-D sensor from an UAV

Author: Katrina Ashton

Supervisor: Viorela Ila

Course: ENGN4712 Engineering Research and Development Project

Submission: October 26, 2018

Acknowledgments

I'd like to thank my supervisor Viorela Ila for her help and guidance. I would also like to thank Jean-Luc Stevens for helping me set-up my quadcopter and associated code, as well as Alex Martin who created the custom parts for my quadcopter and helped me to assemble it and fix it after crashes.

I would also like to thank all of the OpenCV contributors whose registration algorithm implementations I used in this report.

Abstract

This report investigates data registration techniques for use with data captured from a RealSense D415 RGB-D camera mounted to a quadcopter. Data registration techniques are the backbone of mapping algorithms that use visual odometry, and thus choosing a good registration algorithm is extremely important for these applications.

The RGB-D data for this project is collected from the RealSense camera using the official robot operating system (ROS) package. The estimated trajectory is then evaluated using ground truth data obtained from the Vicon system, which uses a number of infra-red (IR) cameras to track markers attached to the quadcopter. The first part of this project was spent setting up this data collection system – including building the quadcopter, setting up the software, and calibrating the Vicon system.

Three different trajectories are investigated – a circle, a rectangle and a back-and-forth "lawnmower" trajectory. The circle trajectory provides smooth motion and a high level of overlap between frames, so it is expected that the registration algorithms will perform well for it. The rectangle and lawnmower trajectories are investigated as they decouple the translation and rotation, which allows how well the registration algorithms perform on each type of motion to be investigated.

Three different registration methods are then investigated – the Essential Matrix method, Kabsch and Perspective-n-points (PnP). All of these methods use sparse data where the point correspondences are known, these correspondences are found using a feature detector and matcher on the RGB images. The Essential Matrix method registers two sets of RGB data. Kabsch registers two sets of 3D data. PnP registers one set of 2D data with a set of 3D data.

It was found that Kabsch performed very poorly, which is likely at least partially due to the fact that the depth data is quite noisy, and could also be due to poorly chosen RANSAC thresholds. The Essential Matrix method and PnP have similar accuracy, although PnP seemed slightly better at coping with corners. PnP is slower, although that could be partially due to the noisy depth data. Overall if paired with a method for reducing drift the Essential Matrix method seems the best choice when time is a factor. PnP may be better if the depth sensor is less noisy or when time is not critical.

Contents

1 Glossary and Notation

1.1 Abbreviations

ICP: iterative closest point – a data registration method.

IR: infra-red

PnP: perspective-n-point

RANSAC: random sample consensus (algorithm)

RGB-D: red, green, blue, depth — an RGB-D sensor gives a colour image (red green and blue channels) as well as the associated depth image.

ROS: robot operating system.

SLAM: simultaneous localization and mapping

UAV: unmanned aerial vehicle.

1.2 Glossary

Pose: position and orientation.

1.3 Notation

${}_iT_j$: transformation between frame i and frame j, defined in the coordinate system for frame i.

${}^fT_j$: pose of frame j, defined in the coordinate system for frame f.

2 Introduction

Being able to create a map of a scene using a robot is useful for a variety of applications – for example, in aiding the robot in navigating, or in task execution. Using an RGB-D sensor allows depth information to be obtained directly, rather than using techniques such as optical flow or structure from motion. However, even when using an RGB-D sensor, creating an accurate scene map is not always a trivial task due to difficulties in aligning the data and noise in the depth measurements.

A key step in creating scene map from RGB-D data is data registration. Data registration is the process of combining two frames of data, when the relative pose of the camera is not known (or not with certainty). In order for this to work properly there needs to be sufficient degree of overlap between the frames for the algorithm the align them. Ideally data registration should discard noisy or redundant points. There are a number of data registration techniques that work well with noise-less data and will be used as a starting point for this report. These include finding the Essential Matrix (Section 4.6) and using the Kabsch algorithm (Section 4.7) and using a method for solving the Perspective-n-points (PnP problem) (Section 4.8).

This report will focus on data registration for data captured using unmanned aerial vehicles (UAVs). UAVs are very maneuverable and the fact that they travel through the air means that their mobility is independent of terrain. Thus UAVs are highly suitable for mapping tasks, especially when the environment being mapped has rough or uneven ground (e.g. most outside locations) or many obstacles on the ground (e.g. many indoors locations which often have things such as desks and chairs).

A quadcopter was thus built for the purpose of obtaining the necessary experimental data (see Section 5), with an attached RealSense camera used to take the RGB-D measurements. The RealSense camera was chosen due to its small size (allowing it to fit on the quadcopter without impacting mobility or flight-time too much) and relatively good quality images. This UAV was then flown under a few different trajectories for different scenes (see Section 6), with ground truth data on the UAV's pose will be obtained using a Vicon localization system (see Section 5.4).

Data from a sensor mounted to a UAV is often very noisy due to the fact that the

sensor experiences a large amount of shake and rapid motions. The rapid motions can also make it difficult to accurately estimate the difference in pose between measurements. These factors make it difficult to accurately register data. Thus this report aims to design and implement a data registration technique that can perform well in this situation, so that the registered data can be used to form an accurate scene map.

3 Literature Review

3.1 Mapping – SLAM

Simultaneous Localization and Mapping (SLAM) is a problem for which the aim is for a mobile robot to create a map of its environment while simultaneously determining its location within this map (without prior knowledge of its environment or location).

There are three main parts to SLAM. The first is obtaining measurements of how the robot is moving (rotation and translation), as well as data it sense from the environment. These measurements then need to be used to update both the robot's map of the environment and its estimate of its position in it. Finally, almost all SLAM implementations have some way of detecting when the robot senses a feature or returns to a place more than once, and using this to update the estimate of the map and the robot's location. This last step is important for reducing the effect of drift that generally occurs when mapping.

The first step, obtaining the measurements of how the robot is moving, may seem relatively simple, and indeed some robots are equipped with sensors such as rotary encoders and gyroscopes that give relatively accurate and precise measurements of the robot's movements with minimal calibration. However, not all robots can be equipped with such sensors (e.g. quadcopters do not have wheels and thus cannot use rotary encoders) and so have to rely on sensors such as cameras, which require much more computation to get useful measurements. How to estimate the motion of a camera from sequential frames (known as data registration) is an ongoing area of research.

Henry et al [17] use a Kinect RGB-D camera to perform dense 3D modeling of indoor environments using SLAM. For their data registration, they use a variant of the iterative closet point (ICP) algorithm, which they call RGB-D ICP, as part of their RGB-D mapping system. This algorithm first finds an estimated transform, and then refines it using

ICP. This initial estimate is found using sparse point clouds, created by extracting features from the RGB images, matching them between frames, and then using them along with their corresponding depth measurements to form 3D points. It proposes two ways to find the transform from these sparse point clouds, depending on the type of sensor used to capture the RGB-D data. For time-of-flight cameras, it optimizes for point-to-point squared distance error, using the method of Horn [18], which uses quaternions to find the optimal rotation. For active stereo RGB-D cameras it optimizes for re-projection error, using two-frame sparse bundle adjustment (SBA). In both cases, Random Sample Consensus (RANSAC) is used in conjunction with these methods to try to remove outliers. This estimated transform is then used as the initial estimate for ICP, again in conjunction with RANSAC, but this time on dense point clouds rather than the sparse ones. The version of RANSAC used is point-to-plane, with minimization using Levenberg-Marquardt. They found that doing the ICP step only gave marginal improvements if sufficient RANSAC inliers were found during the initial estimate. Thus they only performed ICP when the number of inliers found was below a certain threshold.

Kerl et al [29] perform dense visual SLAM on RGB-D data from the RGB-D SLAM Dataset and Benchmark [55], which is also captured using a Kinect, and contains a few different indoor environments. They use joint photometric and geometric error minimization for their data registration. It does this by modeling the photometric error and depth error as a bivariate random variable which follows a bivariate t-distribution. This defines an error function which can then be linearized using a first order Taylor expansion, and then solved iteratively.

Mur-Artal et al's ORB-SLAM [36] performs feature-based monocular SLAM on three different datasets, one from a robot, one hand-held and one from a car. (The NewCollege dataset [53], TUM RGB-D benchmark [56] and KITTI dataset [13], respectively). This method uses ORB features [48] as they are fast and rotation invariant. For data registration, they compute two geometrical models: a homography assuming a planar scene and a fundamental matrix assuming a non-planar scene. It chooses between these scenes by calculating a score for each and using a heuristic a ration between them. ORB-SLAM tracks the camera's pose in the local map. When tracking is lost it uses the PnP algorithm to relocalize the camera.

3.2 RGB-D data registration – ICP

One of the most common methods for data registration is the iterative closest point (ICP) method, of which there are many variations [45]. ICP is only used on two 3D models at a time, and an initial estimate of the relative pose between these two models must be known [49]. ICP can be used on either dense or sparse point clouds.

Rusinkiewicz and Levoy [49] identify six stages of the ICP algorithm:

1. **Selection** of some set of points in one or both models
2. **Matching** the selected points between the two models
3. **Weighing** the corresponding point pairs
4. **Rejecting** certain pairs (can be done by either looking at pairs individually or considering the entire set of pairs)
5. Assigning an **error metric** based on the pair points
6. **Minimizing** the error metric

The variants of ICP affect one or more of these stages.

The matching step is generally done in the Euclidean space using kd-tree to accelerate the search [45]. The most common distance measures used for the matching step are point-to-point and point-to-plane [45]. Point-to-plane was created to deal with the fact that taking a measurement of a surface imposes a discretization error, it does this by not penalizing offsets along a surface [50].

Generalized ICP (GICP) [50] combines the point-to-point and point-to-plane methods to model a locally planar surface in both scans, a technique they say can be thought of as plane-to-plane. GICP focuses on the error metric and minimization steps. It uses a probabilistic model where the points in each point cloud are assumed to be drawn from independent Gaussians, then Maximum Likelihood Estimation (MLE) is used to find the transformation.

When used in SLAM, ICP generally seems to be used to refine pose estimations rather than as the sole registration technique [9, 17].

3.3 RGB data registration – Photometric error

One popular method for dense RGB image registration is minimizing photometric error [29, 3, 37]. Photometric error is a measure of photo-consistency, i.e. how well the points in an image and another aligned image agree.

When depth data is available, the geometric error is often used in conjunction with the photometric error. There are also a few variants of geometric error that can be used, the main ones being Euclidean (as in ICP) and re-projection (which is similar to photometric error but for the depth map rather than the RGB image). There are also a few different approaches to combining the photometric error and geometric error into a unified error function. For example, include using a weighted sum of a non-linear function of each error (e.g. sigmoid) [35], and forming a bivariate random variable and using that to define an error function [29].

3.4 RGB data registration – Essential Matrix

The Essential Matrix method works on RGB data from a calibrated camera with known point correspondences. It first uses the point correspondences and the camera calibration parameters to find the Essential Matrix, and then uses the Essential Matrix in conjunction with the point correspondences to find the change in camera pose. The original method was proposed by Longuet-Higgins [33] and gives a closed-form solution for 8 point correspondences. Iterative methods for solving this problem were also developed [15], but it was found that a slight modification to the closed form solution made its results competitive with iterative algorithms while being much faster [16]. More recently, an efficient 5-point closed form algorithm has been developed that is now widely used [38].

ORB-SLAM [36] uses the Essential Matrix, which they find from the Fundamental Matrix, for registration when the scene is non-planar and has high parallax. When the scene is planar, nearly planar or there is low parallax, a Fundamental Matrix can be found but the problem is not well constrained, and thus ORB-SLAM uses homography instead for these cases.

3.5 RGB-D data registration – Kabsch

The Kabsch algorithm was first proposed in 1976 by Kabsch [23] as a way of aligning two sets of matched 3D points, with an amendment two years later which ensures a proper rotation is obtained [24]. The original formulation is in terms of Lagrange multipliers, but modern implementations use singular value decomposition (SVD) [28].

The Kabsch algorithm is fairly widely used for aligning molecules [28, 14] and indexing rotation diffraction patterns [25, 26]. More recently, it has been used by the computer vision community on RGB-D data as part of tasks such as non-rigid reconstruction [59], pose estimation [4] and camera re-localization [52].

3.6 RGB-D data registration – Perspective-n-Point

The Perspective-n-Point (PnP) problem is to determine the pose of a calibrated camera given a set of n correspondences between 3D points and their 2D projections. The term PnP problem was introduced by Fischler and Bolles [11], as reformulation of the Location Determination Problem. The PnP problem is only solvable for $n \geq 3$, however for $n = 3$ there are multiple viable solutions, so a fourth point correspondence is needed to remove ambiguity. A widely accepted solution to the P3P problem is given in Gao et al [12]. For arbitrary $n \geq 4$, there are various iterative methods [41, 15], as well as an $O(n)$ closed-form solution called EPnP [31].

EPnP is used in ORB-SLAM [36] to relocalize the camera when tracking is lost.

4 Background

This section details the necessary background information on rigid body motion, coordinate frames, the pinhole camera model and registration methods. The registration methods described in Sections 4.6-4.8 are key to this report, as it focuses on comparing them for use with data from a quadcopter. These methods are all used in conjunction with the Random Sample Consensus (RANSAC) algorithm in order to reduce the effect of outliers, and so the RANSAC algorithm is described in Section 4.5.

However, before these algorithms can be properly implemented and used for mapping, an understanding of how the quadcopter can move, as well as coordinate frames and

how to convert between the image and camera frames is necessary. Thus, Section 4.1 describes the basics of rigid body motion, Section 4.2 describes the relevant coordinate frames and how to convert between them, and Section 4.3 describes the camera model used, its intrinsic parameters and also derives the equations used to convert between the image and camera frames.

The Essential Matrix (Section 4.6), Kabsch (Section 4.7) and Perspective-n-Points (PnP) (Section 4.8) methods all represent fairly basic algorithms that can be used on sparse data where the point correspondences are known. The simplest way to get such data is to use a feature detector and then match the features between the frames being registered. Each of these methods uses data from different frames: the Essential Matrix uses data from image frames (RGB), Kabsch uses data from camera frames (3D) and PnP takes data from an image frame (RGB) and data from a camera frame (3D).

Many of these algorithms are already implemented in the Open Source Computer Vision Library (OpenCV) and where available these implementations are used in this report. Thus an overview of OpenCV and how to use it is provided in Section 4.4.

4.1 Rigid Body Motion

Rigid body motion is motion of a body that preserves the Euclidean distance between any two points on that body. There are three types of rigid body motion: rotations, translations and reflections, with a combination of the three also being a rigid body motion. Proper rigid body motion also specifies that handedness is preserved, which excluded reflections. This report only considers proper 3D rigid body motion, i.e. combinations of rotation and translation in 3 dimensions.

When applied to a body, a rotation and translation change its orientation and position, respectively. Together orientation and position are called *pose*. Proper rigid body motion and poses are both elements of the Special Euclidean Group $SE(3)$, although they are generally represented as members of its associated Lie Algebra $\mathfrak{se}(3)$ which decouples the rotation and translation.

There are multiple ways of representing a rotation. One way is a rotation matrix, which is an element of the Special Orthogonal group $SO(3)$, that is it is an orthogonal 3×3 matrix with determinant 1. Another way is by axis-angle, which gives an axis about which

the rotation occurs (unit vector in $\mathbb{R}^3$) and the magnitude of that rotation (element of $\mathbb{R}$). Another common way is with Quaternions, which can be written as $(a, b, c, d) \in \mathbb{R}^4$ which represents the Quaternion $a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ where $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are the fundamental Quaternion units with the property that $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$. Rotation matrices and Quaternions are easier to combine and apply to a body than axis-angle, with Quaternions being more compact and generally being faster to compute than rotation matrices. The axis angle representation is useful for comparing rotations, e.g. for finding errors, they are not over parameterized.

Translations are elements of $\mathbb{R}^3$; range and bearing can also be used to represent them.

4.2 Coordinate frames

There are four main coordinate frames relevant to this report: the world-fixed frame, the quadcopter-fixed frame, the camera-fixed frame and the image-fixed frame. The "fixed" part of these names may be dropped (so camera frame refers to the camera-fixed frame). There is also another world-fixed frame that the ground truth position data is given in, this will always be referred to as the quadcopter world-fixed frame (which is not the same thing as the quadcopter-fixed frame). This report seeks to use RGB-D information in the image frame to find the movement of the camera in the world frame, but to do this the camera frame and then quadcopter frame must be passed through. An additional Vicon frame is defined, which is the format that the ground truth poses of the quadcopter are given in (see Section 6). The Vicon frame could be used as the world reference frame, however for ease of plotting and understanding it was decided to use two frames. Figure 1 shows these frames and how they relate to each other.

Before continuing further, notation for poses and transformations is necessary. Let ${}^fT_j$ be the pose of frame j, defined in the coordinate system from frame f. Then let ${}_iT_j$ be a transformation between frame i and frame j, defined in the coordinate system for frame i. These frames can be represented by just a letter W for world, Q for quadcopter, C for camera, I for image, and V for Vicon. However, the camera and quadcopter frames are not fixed with respect to the world frame, so the timestep needs to be given as well. So Qn refers to the quadcopter frame at the n^{th} timestep, and likewise Cn refers to the camera frame at the n^{th} timestep.

The ground truth position data is in the Vicon frame (with the x-axis aligned with the world frame), and so needs to be converted to the world frame. This can be done by multiplying it by a rotation matrix, ${}_{V}R_{W}$ which rotates by 180 degrees around the x-axis.

The registration finds transforms in the camera frame, which then need to be converted into the world frame. To convert from the camera to world frame, the quadcopter-fixed frame is passed through and the pose of the quadcopter-fixed frame in the world frame must be known. To convert from the camera frame to the quadcopter frame another rotation matrix, ${}_{C}R_{Q}$ is used, this rotation matrix first rotates around the x-axis by 135 degrees, then about the (new) z-axis by 90 degrees. Looking at Figure 1, it can be seen that these rotations will take the camera frame (orange) into alignment with the the quadcopter frame (blue). Note that the camera is not actually mounted to the center of the quadcopter, so an accurate transformation would include some translation along with the rotation. However, the center of the quadcopter in the ground truth is defined by the Vicon, and is difficult to locate it and measure the distance to the camera precisely. The Vicon also often needs to be re-calibrated, which may result in the point defined as the center changing slightly. In addition, the magnitude of the translation is also not very large, thus it was decided that the benefit of including a translation in the transform did not outweigh the added complications.

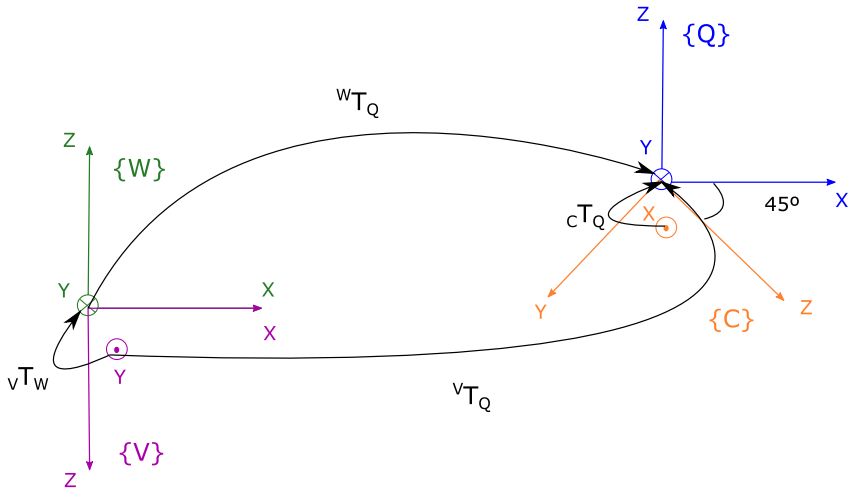


Figure 1: (Best viewed in colour). 3D coordinate frames and how they relate. World frame $\{W\}$ in green, Vicon frame $\{V\}$ in purple, quadcopter frame $\{Q\}$ in blue, camera frame $\{C\}$ in orange.

Then, when ${}^{W}[R|t]_{Qn}$ and ${}_{Cn}[R|t]_{C(n+1)}$ are known, ${}^{W}[R|t]_{Q(n+1)}$ is obtained as follows:

$$
\begin{aligned}
{}^{W}R_{Q(n+1)} &= {}^{C0}R_{Qn}\ {}_{C}R_{Q}\ {}_{Cn}R_{C(n+1)}\ {}_{C}R_{Q}^{T} \\
{}^{W}t_{Q(n+1)} &= {}^{W}R_{Qn} + {}^{W}R_{Qn}^{T}\ {}_{C}R_{Q}\ {}_{Cn}t_{C(n+1)}
\end{aligned}
$$

In order to use these equations, the first pose ${}^{W}[R|t]_{Q0}$ is assumed to be known (in this report it is taken from the ground truth, some other methods initialize at the origin). Then there are two possible methods to find ${}^{W}[R|t]_{Q(n+1)}$. The first is by finding an absolute estimated transform. In this case, the previous estimate is used for ${}^{W}[R|t]_{Qn}$ (so the estimations must be done from the first timestep to the $n+1^{\text{th}}$ timestep in order to get the absolute estimated transform ${}^{W}[R|t]_{Q(n+1)}$). The second is to get a relative estimated transform. In this case, the ground truth is used for ${}^{W}[R|t]_{Qn}$ (so only ${}_{Cn}[R|t]_{C(n+1)}$ needs to be found to get the absolute estimated transform ${}^{W}[R|t]_{Q(n+1)}$). The absolute trajectory is what will be used in practice in most cases (as the ground truth is not known). However the absolute error is prone to drift, so looking at the relative error can give a better insight into how accurate the registration is at each timestep (and thus give insight into possible sources of error).

4.3 Camera model

This report uses a pinhole camera model, which is equivalent to assuming no lens or aperture distortion. It is also valid when distortion is removed from image data during preprocessing. Under the pinhole model, the scene view is formed by projecting 3D points into the image plane using the perspective transformation [6]

$$
sm' = K[R|t]M'
$$

which is

$$
s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}
$$

where

- (X, Y, Z) are the coordinates of a 3D point in the world frame

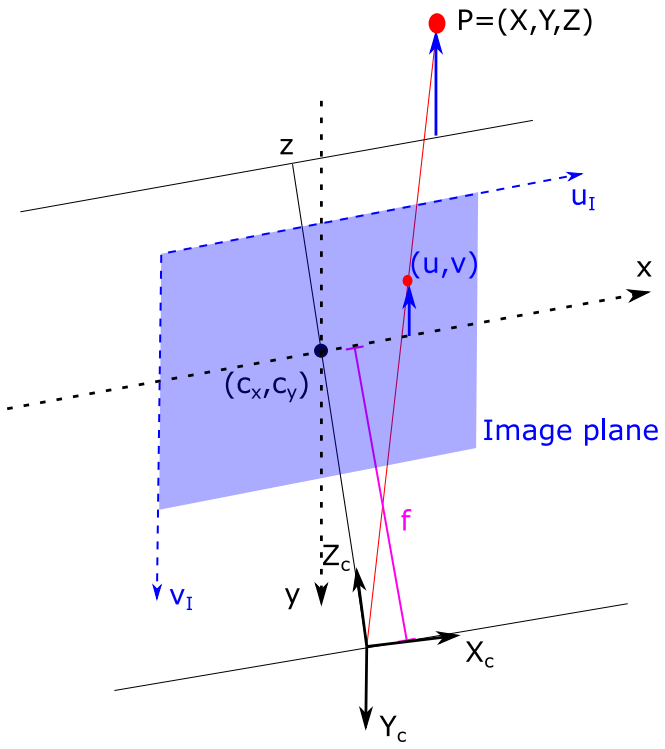


Figure 2: (Best viewed in colour). Pinhole camera model

- (u, v) are the coordinates of the corresponding projection point in pixels (image coordinates)
- s is a scaling term
- R and t are a rotation and translation matrix of the camera, respectively
- K is a matrix of intrinsic camera parameters (note that there is no skew term due to the fact that a pinhole camera model is used)
- (c_x, c_y) is the principal point (usually at the image center), see Figure 2
- f_x and f_y are the focal lengths expressed in pixel units. Figure 2 shows the case where $f = f_x = f_y$ as in an ideal pinhole camera, but this is not necessarily the case in practice. When $f_x \neq f_y$, the resulting image has non-square pixels.

This can be written in a way that is easier to switch between the image and camera frames. Start by inverting K:

$$K^{-1} = \begin{bmatrix} 1/f_x & 0 & -c_x/f_x \\ 0 & 1/f_y & -c_y/f_y \\ 0 & 0 & 1 \end{bmatrix}$$

And $[R|t][XYZ1]^T = R[XYZ]^T + t$, so

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

$$s\begin{bmatrix} 1/f_x & 0 & -c_x/f_x \\ 0 & 1/f_y & -c_y/f_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = R\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$s\begin{bmatrix} u/f_x - c_x/f_x \\ v/f_y - c_y/f_y \\ 1 \end{bmatrix} = R\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$\begin{bmatrix} s(u - c_x)/f_x \\ s(v - c_y)/f_y \\ s \end{bmatrix} = R\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

Note that $R\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$ is the position of a 3D point in the camera frame. So this gives

$$x = s(u - c_x)/f_x$$

$$y = s(v - c_y)/f_y$$

$$z = s$$

and thus

$$u = xf_x/z + c_x$$

$$v = yf_y/z + c_y$$

for a point (x, y, z) in camera coordinates with projected image coordinates (u, v).

This can be written as the projection function

$$\pi({}^I\mathbf{p}) = (s(u - c_x)/f_x, s(v - c_y)/f_y, s)$$

and inverse projection function

$$\pi^{-1}({}^C\mathbf{p}) = (xf_x/z + c_x, yf_y/z + c_y)$$

for a point ${}^c\mathbf{p} = (x, y, z)$ in the camera frame with projected image coordinates ${}^i\mathbf{p} = (u, v)$. These functions require that the principal point, focal lengths and scaling factor of the camera are known. This report only works with calibrated cameras, so these functions can be used.

4.4 OpenCV

The Open Source Computer Vision Library (OpenCV) is a library with C++, Python and Java interfaces which provides a number of useful functions for working with images and performing computer vision tasks [44]. It implements the Essential Matrix method and PnP, as well as a feature descriptor and matcher. OpenCV is well established, widely used and has many contributors, thus these functions are likely well optimized. OpenCV is also implemented in C/C++ code, with an interface provided for use with other languages, thus their implementation will also likely be faster than a purely Python implementation would be. Therefore when OpenCV implementations of functions are available they will be used instead of coding them from scratch.

In Windows, OpenCV for Python can be installed via pip (`pip install opencv-python`). There are also anaconda distributions as well as official releases with pre-built binaries and source code. After OpenCV is installed, it can be imported using `import cv2`.

The main OpenCV functions used in this report are described in the Camera Calibration and 3D Reconstruction documentation [42]. OpenCV also provides various feature descriptors (such as SIFT, SURF, ORB, BRIEF, BRISK) and functions for matching features (such as FLANN and a brute force matcher) [43]. This report uses the SIFT feature descriptor, and the matches are found using a brute force matcher.

4.5 RANSAC

The RANdom SAmple Consensus (RANSAC) algorithm is method of fitting a model to data with a high proportion of outliers. Previous techniques for dealing with outliers often started by using all of the data and then trying to remove wrong correspondences, RANSAC takes the opposite approach of starting with a small number of points and adding data that is consistent with the model generated from these points [11]. It begins by sampling the minimum number of points needed to find parameters for the model, and

finds parameters based only on those points. It then finds all the points that are consistent with the model using those parameters (up to some predefined tolerance). If there are enough points (i.e. the proportion is over some other predefined threshold), the parameters are re-estimated using all of the consistent points. If there are not enough points, the process begins again by sampling new points. A maximum number of repetitions is set beforehand, this number is chosen to ensure a high enough probability p (usually set to 0.99) that at least one of the sets of (initial) random samples does not include an outlier [5]. When the probability of a point being an outlier v is known, N can be specified in terms of v, p and the minimum number of points needed for the model, m [5]

$$N = \frac{\log(1-p)}{\log(1-(1-v)^m)}$$

4.6 Registration Methods: Essential Matrix

The essential matrix arises from the basic epipolar constraint [57]

$$\hat{x}_1^T E \hat{x}_0 = 0$$

Where $\hat{x}_j = K_j^{-1} x_j$ are the local ray direction vectors (K_j are the calibration matrices, x_j are the image coordinates) and

$$E = [t]_x R$$

is the essential matrix.

The essential matrix can be found in OpenCV via the function findEssentialMat, using matched points and the focal length and principal point of the camera. This is done using Nister's five-point algorithm [38]. Note that in practice more than five points may be required to ensure the estimation is robust and accurate. With more than five points, the five-point algorithm can be used as a hypothesis generator within a random sample consensus scheme (RANSAC) (see Section 4.5). That is, five point correspondences are chosen at random, then the five-point algorithm is used to estimate the essential matrix from them, this estimate is a hypothesis. This process is repeated to get a number of hypotheses, which are scored using a robust statistical measure over all points. This gives an overall estimate of the best hypothesis, which can be improved iteratively as more sets of five point correspondences are used. In OpenCV this process does not necessarily

use all possible sets of five point correspondences; it stops once the probability of the hypothesis being correct has reached the specified probability threshold.

The algorithm itself uses the fact that a real non-zero 3×3 matrix E is an essential matrix if and only if it satisfies:

$$EE^T E - \frac{1}{2} trace(EE^T)E = 0$$

There are additional constraints imposed by each point correspondence: $\tilde{q}^T \tilde{E} = 0$ where

$$\tilde{q} \equiv \begin{bmatrix} q_1 q'_1 & q_2 q'_1 & q_3 q'_1 & q_1 q'_2 & q_2 q'_2 & q_3 q'_2 & q_1 q'_3 & q_2 q'_3 & q_3 q'_3 \end{bmatrix}^T$$
$$\tilde{E} \equiv \begin{bmatrix} E_{11} & E_{12} & E_{13} & E_{21} & E_{22} & E_{23} & E_{31} & E_{32} & E_{33} \end{bmatrix}^T$$

The $\tilde{q}^T$ vectors for the five points can then be stacked to get a 5×9 matrix. Then four vectors $\tilde{X}, \tilde{Y}, \tilde{Z}, \tilde{W}$ that span the right nullspace of this matrix are found (when there are multiple possibilities, the ones corresponding to the smallest singular values are used). These vectors correspond to four 3×3 matrices X, Y, Z, W, which can be used to find the Essential Matrix as

$$E = xX + yY + zZ + wW$$

for some scalars x, y, z, w, it is assumed $w = 1$ as these scalars are only defined up to a common scale factor. This equation can then be inserted into the 9×5 constraint matrix. This can then be used to find x, y and z (see [38] for details), and thus E can be found from $E = xX + yY + zZ + wW$.

The OpenCV function recoverPose can then be used to recover the relative pose difference of the camera using the above information and the Essential Matrix. The process for doing this is also described in [38]. This is done by setting

$$D = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

then the singular value decomposition of the essential matrix is $E \sim Udiag(1, 1, 0)V^T$. Which means $t \sim t_u \equiv \begin{bmatrix} u_{13} & u_{23} & u_{33} \end{bmatrix}^T$ and R is equal to $R_a \equiv UDV^T$ or $R_b \equiv UD^T V^T$. Any combination of R and t that according to this satisfies the epipolar constraint [38]. Thus these ambiguities need to be resolved. This starts by assuming the pose of the first camera, as without additional information it is only possible to recover the relative pose

between the two cameras. Thus to simplify the maths, the first camera is assumed to be located at the origin and aligned with the axes (identity rotation). Then there are four possible poses for the second camera: $P_A \equiv [R_a|t_u]$, $P_B \equiv [R_a|-t_u]$, $P_A \equiv [R_b|t_u]$ and $P_D \equiv [R_b|-t_u]$. Only one of these is the true configuration. Which one it is can be found by imposing a cheirality constraint (i.e. all scene points should be in front of the cameras) with at least one point [38].

4.7 Registration Methods: Kabsch

Kabsch [23] is an algorithm that is used to align matched 3D points. That is, it can only be used for two point clouds for which each point in one cloud has a known matching point in the other cloud.

The algorithm minimizes the weighted sum of squared deviations between paired points. That is, it attempts to find an orthogonal matrix R which minimizes the function

$$E = \frac{1}{2}\sum_{i=1}^{N} w_i |Rx_i - y_i|^2,$$

where x_i and y_i are two vectors of N points, and w_i are the weights corresponding to each pair x_i, y_i (for the data used in this report all of the points are assumed to be of equal weight, and so the w_i are set to $1/N$). The original papers by Kabsch [24, 23] use Lagrange multipliers to find this R. However there is an equivalent method that uses the Singular Value Decomposition (SVD) [28] that will be used for this report.

The first step is finding the optimal translation for aligning the point clouds. It turns out that the optimal translation occurs by translating one point cloud so that its centroid coincides with the other point cloud's centroid [28]. Where the centroid x_c of a set of points $\{x_1, x_2, ..., x_N\}$ is the average over all of the points:

$$x_c = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

Thus the translation to get from a point cloud X to a second point cloud Y is given by $y_c - Rx_c$, where R is the rotation that is found in the following steps. To find the rotation between, the point clouds need to be aligned with the origin. This is done by subtracting the centroid from each one,

$$X_0 = X - x_c, Y_0 = Y - x_c.$$

Now we go back to considering minimizing E, but with equal weights

$$E = \frac{1}{2N}\sum_{i=1}^{N}(Rx_i - y_i)^2.$$

This can be written in terms of traces [28][1]. First, let $Rx_i = x'_i$, which gives

$$2NE = \sum_{i=1}^{N}|x'_i - y_i|^2.$$

Now writing this in terms of $X' = RX$ and Y (where these are $3 \times N$ matrices with a point per row) gives

$$\begin{aligned}
2NE &= \sum_{n=1}^{N}|x'_n - y_n|^2 \\
&= Tr((X' - Y)^T(X' - Y)) \\
&= Tr((X'^T - Y^T)(X' - Y)) \\
&= Tr(X'^T X') + Tr(Y^T Y) - 2Tr(Y^T X') \\
&= \sum_{i=1}^{N}(|x'_i|^2 + |y_i|^2) - 2Tr(Y^T X') \\
&= \sum_{i=1}^{N}(|x_i|^2 + |y_i|^2) - 2Tr(Y^T X')
\end{aligned}$$

The last line holds as R is a rotation matrix, and thus Rx has the same length as x. The first term $\sum_{n=1}^{N}(|x_n|^2 + |y_n|^2)$ is thus not affected by the choice of R. So when minimizing E it can be ignored, while $Tr(Y^T X')$ must be maximized. Recall that $X' = RX$ so $Tr(Y^T X') = Tr(Y^T RX)$. Using the fact that $Tr(AB) = Tr(BA)$, this gives

$$Tr(Y^T RX) = Tr((Y^T R)X) = Tr(X(Y^T R)) = Tr((XY^T)R)$$

(Recall that X and Y are $3 \times N$ and R is 3×3, so $Y^T R$ is $N \times 3$ and XY^T is 3×3 so these products are all defined).

Since XY^T is a 3×3 square matrix, it can be decompose via SVD into

$$XY^T = VSW^T$$

[1]Relevant properties of traces are for matrices A, B and scalar k, $Tr(A + B) = Tr(A) + Tr(B)$, $Tr(AB) = Tr(BA)$ when BA is defined, $Tr(A^T) = Tr(A)$ and $Tr(kA) = kTr(A)$

where V and W^T are the matrices of left and right eigenvectors (which are orthonormal) and S is a diagonal matrix whose elements are the eigenvalues of XY^T in decreasing order. Thus

$$Tr(XY^TR) = Tr(VSW^TR) = Tr(SW^TRV) = \sum^{3} -i = 1 s_i w_i^T R v_i$$

(V, S, W^T, R are all 3×3, so all of these products are defined).

Letting $T = W^TRV$, this can be written as

$$Tr(X^TR) = \sum_{i=1}^{3} s_i T_{ii} \leq \sum_{i=1}^{3} s_i$$

The inequality holds because T is the product of orthonormal matrices, and so it itself an orthonormal matrix, and for an orthonormal matrix, the absolute value of each element cannot be greater than 1.[2] This inequality means that $\sum_{i=1}^{3} s_i T_{ii}$ is maximized when it equals $\sum_{i=1}^{3} s_i$. This occurs $T_{ii} = 1$ for all i, and as T is orthonormal this means that $T = I$. Which means

$$T = I$$
$$W^TRV = I$$
$$WW^TRVV^T = WIV^T$$
$$R = WV^T$$

as W and V are orthonormal. Note that as W and V are orthonormal, this construction of R must also be orthonormal. However, R is a proper rotation matrix, so the further constraint that $\det R = 1$ is imposed [28]. So if $\det(WV^T) = -1$ then another value has to be chosen for R. As it turns out, the second largest value of $\sum_{i=1}^{3} s_i T_{ii} = s_1 T_{11} + s_2 T_{22} + s_3 T_{33}$ occurs when $T_{11} = T_{22} = 1$ and $T_{33} = -1$ as $s_1 \geq s_2 \geq s_3 \geq 0$ and $|T_{ii}| \leq 1$. So $T = diag(1, 1, -1)$. Thus the optimal rotation R can be expressed as

$$R = W \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & d \end{bmatrix} V^T$$

where $C = XY^T$ with SVD $C = VSW^T$ and $d = \det WV^T$.

[2] For an orthonormal matrix $P = [p_1, p_2, ..., p_m]^T$ where p_i are row vectors, $PP^T = I$.

So $p_i \dot{p}_i = 1$. $p_i \dot{p}_i = \sum_j p_{i_j} p_{i_j} = \sum_j p_{i_j}^2$. As the terms of the sum are squared real numbers, they must be non-negative. So for their sum to equal 1, there cannot be an element $p_{i_j}^2$ which is greater than 1, which means that none of the the p_{i_j} can be greater than 1.

4.8 Registration Methods: Perspective-n-Point

The Perspective-n-Point (PnP) algorithm aims to determine the pose of a camera given its intrinsic parameters and a set of n correspondences between 3D points and their 2D projections. Generally the 3D object points are given in the world frame, for use in data registration the 3D points and 2D points need to come from different frames. Thus in this report, for registering frames t and $t+1$, the object points are given in the frame of the camera at time t, and the image points are given in the time $t+1$ image frame. Thus the pose found is the pose of the camera at time t in the frame of the camera at time $t+1$. Thus inverting this pose will get the transformation that turns the the camera pose at time t into the camera pose at time $t+1$ (in the frame of the camera at time t).

There are both iterative and non-iterative methods for solving this problem. Non-iterative methods are efficient but prone to instability when noise is present, especially for $n \leq 5$ as there is no redundant information available [32]. There are non-iterative methods that have complexity $O(n)$ [31, 32]. When redundant points are available, there are iterative methods based on the minimization of nonlinear cost functions that are generally more accurate than non-iterative methods, but they are prone to instability due to local minima of the cost functions, and tend to have a higher computational cost [32].

This report uses OpenCV's solvePnPRansac function, which has the option of using an iterative method based on Levenberg-Marquardt optimization [34], a linear-time non-iterative method called EPnP [31] and a method that works with exactly four object and image points [12]. The last method was not considered for use as in order for RANSAC to work properly the algorithm needs to be able to use a variable number of points. Thus both the iterative method and EPnP are tried in this report.

EPnP attempts to retrieve the coordinates of the 2D image projection points in the camera coordinate frame corresponding to that image. It then aligns both sets of coordinates using an algorithm like Kabsch. This approach is shared by most proposed solutions to the PnP problem [31]. Note that this can still produce better results than just using Kabsch even when depth data is available for both frames, as it may find better inliers and/or produce a better estimate of the 3D points (if the depth data is noisy).

It expresses the estimated coordinates of the reference points in the camera frame as a weighted sum of virtual control points. 4 non-planar control points are needed for general

configurations, and 3 for planar ones. The general case with 4 points will be discussed here.

First, denote the control points $\mathbf{c}_i$ and the reference points $\mathbf{p}_i$, then the reference points can be written in terms of the control points as

$$^W\mathbf{p}_i = \sum_{j=1}^{4} \alpha_{ij} {}^W\mathbf{c}_j, \; ^C\mathbf{p}_i = \sum_{j=1}^{4} \alpha_{ij} {}^C\mathbf{c}_j$$

with $\sum_{j=1}^{4} \alpha_{ij} = 1$.

Now, given a camera internal calibration matrix K, and 2D points $^I\{\mathbf{p}_i\}_{i=1,\dots,n}$ with corresponding 3D points $^C\{\mathbf{p}_i\}_{i=1,\dots,n}$, the following holds [31]

$$\forall i, w_i \begin{bmatrix} ^I\mathbf{p}_i \\ 1 \end{bmatrix} = K {}^C\mathbf{p}_i = K \sum_{j=1}^{4} \alpha_{ij} {}^C\mathbf{c}_j$$

where w_i are the scalar projective parameters. This can be expanded to

$$\forall i, w_i \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & x_c \\ 0 & f_y & y_c \\ 0 & 0 & 1 \end{bmatrix} \sum_{j=1}^{4} \alpha_{ij} \begin{bmatrix} ^Cx_j \\ ^Cy_j \\ ^Cz_j \end{bmatrix}$$

This defines three linear equations (one for each row). The last one is that $w_i = \sum_{j=1}^{4} \alpha_{ij} {}^Cz_j$. Substituting this into the first two equations gives the following two equations [31]

$$\sum_{j=1}^{4} \alpha_{ij} f_x {}^Cx_j + \alpha_{ij}(x_c - u_i) {}^Cz_j = 0$$
$$\sum_{j=1}^{4} \alpha_{ij} f_y {}^Cy_j + \alpha_{ij}(y_c - v_i) {}^Cz_j = 0$$

Then these equations can be concatenated for all n reference points, to generate a linear system of the form $\mathbf{Mx} = \mathbf{0}$ with $\mathbf{x} = ({}^C\mathbf{c}_1^T, {}^C\mathbf{c}_2^T, {}^C\mathbf{c}_3^T, {}^C\mathbf{c}_4^T)^T$. So the solution belongs to the kernel of $\mathbf{M}$ and can be expressed as [31]

$$\mathbf{x} = \sum_{i=1}^{N} \beta_i \mathbf{v}_i$$

where $\mathbf{v}_i$ are the columns of the right-singular vectors of $\mathbf{M}$, corresponding to the N null singular values of $\mathbf{M}$. These $\mathbf{v}_i$ can be found as the null eigenvectors of the matrix $\mathbf{M}^T\mathbf{M}$, which is of size 12×12. An estimate of the β_i is then performed, with the method

varying depending on N (which ranges from 1 to 4). This estimate is then refined using the Gauss-Newton algorithm to minimize [31]

$$\text{Error}(\beta) = \sum_{(i,j)|i<j} \left(||^{C}\mathbf{c}_i - {}^{C}\mathbf{c}_j||^2 - ||^{W}\mathbf{c}_i - {}^{W}\mathbf{c}_j||^2 \right)$$

with respect to β.

5 Quadcopter Set-up

A quadcopter was built for the purpose of acquiring the data (see Figure 3). Its' design is the same as an existing one built by Jean-Luc Stevens (PhD candidate) except for using a slightly different frame to try to dampen vibrations, and the fact that it uses a different sensor with different mounting. A full component list can be found in Section 5.1. Details about the main computational platform (Jetson TX2) and the camera (RealSense D415) are provided in Sections 5.2 and 5.3, respectively.

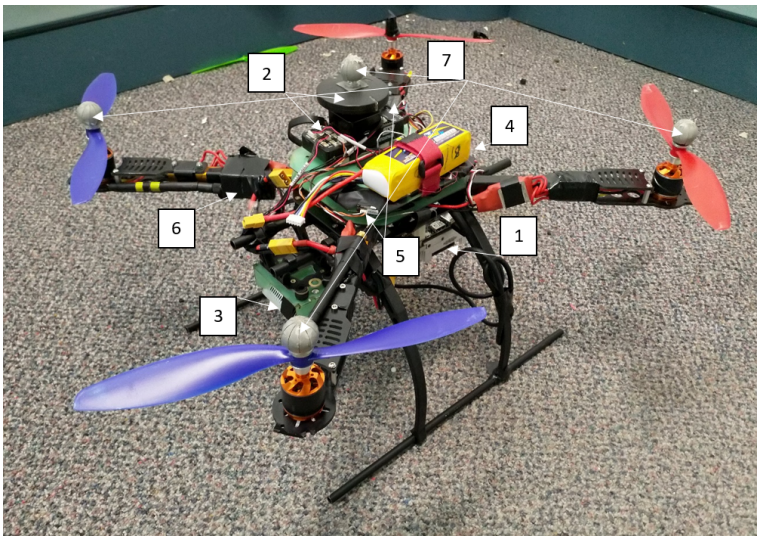


Figure 3: Quadcopter

Major components:

1. TX2 with Orbitty carrier board
2. Pixhawk 4 flight controller and GPS
3. RealSense camera with fixed mount
4. lithium polymer battery
5. Receiver
6. Radio
7. Vicon markers

5.1 Quadcopter Components

Flight controller	Pixhawk 4
On-board computer	NVIDIA Jetson TX2, mounted to Orbitty carrier board
GPS	Here GNSS
Propellers	254mm x 114mm, 10 x 4.5L/R
Vicon markers	Custom, round and coated in reflective cloth
Motors	Turnigy D2836/9 950KV
Speed controllers	Turnigy PLUSH - 30A, BEC - 2A/5V ver 3.1, DC 5.6V-16.8V
Radios	3DR radio v2 (915 MHz, SiK firmware)
Power distribution board	
Frame	Custom, fiber glass, vibration dampening
Arms	No longer in production
Legs	
RealSense mount	Custom, fiber glass, attached to quad via mounting platform with vibration dampening from mini 3D pros 3 axis gimbal
Battery	Lithium polymer, 2200 mAh
Cables	2 telemetry cables, power cables.

One telemetry cable connects the Pixhawk to the 3DR radio, the other connects the Pixhawk to the TX2. For the Pixhawk to radio cable, a 6-pin cable is needed, which slots into the Pixhawk's telem 1 connector. The cable between the Pixhawk and TX2 has a ribbon cable to connect to the Orbitty Carrier which then connects to the 6-pin telem 2 connector on the Pixhawk. This cable connects the TX on the Pixhawk to the RX on the Orbitty Carrier and vice versa, note that the Orbitty carrier uses UART1. It also connects the grounds.

5.2 Jetson TX2

The Jetson TX2 is an embedded platform made by NVIDIA. It is used as the control board and main computational platform for the UAV used in this report. It was selected due to the fact that it has high computational power while still being light and power-

efficient enough to be mounted on a quadcopter [39]. It is also has a good development community (e.g. JetsonHacks [22]).

The Jetson TX1 is very similar to the TX2, and could likely be substituted if a TX2 is not available. However the TX1 has less computing power so running some programs on-board (such as an ICP implementation) may not be possible. The TX2 has the following specifications [40]:

- GPU: NVIDIA Pascal, 256 CUDA cores
- CPU: HMP Dual Denver 2/2 MB L2 + Quad ARM A57/2 MB L2
- Video: 4K x 2K 60 Hz Encode (HEVC), 4K x 2K 60 Hz Decode (12-Bit Support)
- Memory: 8 GB 128 bit LPDDR4, 59.7 GB/s
- Display: 2x DSI, 2x DP 1.2 / HDMI 2.0 / eDP 1.4
- CSI: Up to 6 Cameras (2 Lane), CSI2 D-PHY 1.2 (2.5 Gbps/Lane)
- PCIE: Gen 2 — 1x4 + 1x1 OR 2x1 + 1x2
- Data Storage: 32 GB eMMC, SDIO, SATA
- Other: CAN, UART, SPI, I2C, I2S, GPIOs
- USB: USB 3.0 + USB 2.0
- Connectivity: 1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth
- Mechanical: 50 mm x 87 mm (400-Pin Compatible Board-to-Board Connector)

The TX2 comes with a development board, but this is too big to be mounted on a quadcopter, thus the orbitty carrier board is used. installation process of the TX2 for use with an Orbitty carrier board is described in Appendix 10.1.

5.3 Intel RealSense RGB-D Sensor

The Intel RealSense line of Cameras capture red, green, blue and depth channels. They connect to a computer via USB 3.0. The experiments for this report began by using the

R200 model, but it was discovered that the depth reading were not of sufficient quality so the D415 is used instead.

librealsense, a SDK for the Intel RealSense camera is available on git [21]. Note that the older cameras such as the R200 only work with the legacy SDK, whereas the D415 uses the current SDK (version 2). The TX1 and TX2 are not officially supported [20]. Jetson hacks [27] has created a script which will install librealsense on a TX1 or TX2, however this is for a legacy version of the RealSense SDK. After installing this hack the old SDK can be removed and the new SDK installed with no issues.

ROS packages are available for the RealSense cameras (for the Kinetic ROS distribution), these packages are split in the same way as the SDK, with old cameras using realsense_camera [47] and new ones using realsense2_camera [51]. Note that the SDK has to be installed first, then the ROS package can be installed using catkin as described in the installation instructions on the package's GitHub page [19].

Using a ROS package makes synchronizing the data much easier and also allows for easier integration into a mapping pipeline. Thus the ROS package is used to acquire the necessary data, with the results being saved in a rosbag.

The realsense node can be launched using roslaunch realsense2_camera rs_camera.launch, then the RGB images are published to camera/color/image_raw and the depth images are published to camera/depth/image_rect_raw. The intrinsic parameters are published to camera/color/image_info and camera/depth/image_info for the RGB and depth cameras, respectively.

5.4 Vicon system

The Vicon system is used to get the ground truth pose of the quadcopter. It does this by tracking Vicon markers, which are round objects covered in a shiny material. It does this using IR cameras, which connect to a computer. In order to track the quadcopter, four Vicon markers are attached (see Figure 3 at the start of Section 5) one on top of the Pixhawk and one on top of each of three propellers. This configuration is chosen so that when just visualizing the markers it is clear where on the quadcopter they are.

In order to set up the Vicon system to track the quadcopter, it first has to be calibrated so that the markers on the quadcopter are visible and fairly stable. This is done from

the System tab, under the Vicon Cameras option. From there, the strobe intensity and threshold have to be adjusted. If one of the markers is not showing up, the threshold should be decreased. If a maker is shaking or switching between a few positions, try reducing the strobe intensity or increasing the threshold. If just one of the markers is shaking slightly it should still be able to track properly, although ideally all of them should be stable.

Once the markers are stable, the quadcopter needs to be defined as an object. This is done in the objects tab. All of the markers should be selected, and then the create object option chosen. Make a note of the name chosen, as it needs to be used for the vicon_bridge rospackage. A marker can also be added later by selecting it then clicking add and then clicking on the object to add it to. Markers can be removed by right clicking them and selecting detach, either on the 3D model or from the markers list for the object. Once an object has been defined, its alignment needs to be set. This is done by clicking the pause button under the objects tab (top right), and then clicking and dragging the axes into the desired alignment.

Note that the Vicon system should be checked each time before the quadcopter is flown, and recalibration may be necessary. Usually tuning the strobe intensity and threshold is enough, but sometimes (especially after a crash) the markers will need to be updated too.

5.5 Setting up ground station

The ground station used for this report runs Ubuntu 16.04 and ROS kinetic.

This report uses QGroundControl as the main ground control station, however in order to connect to the ground station via radio the radios must first be set up in a different program, this report uses Mission Planner. Installation instructions for ROS can be found in Appendix 10.2, and installation instructions for Mission Planner and QGround Control can be found in Appendix 10.3.

5.5.1 Connecting to Vicon and PX4

The ground station will communicate with the Vicon system via vicon_bridge [1] the code for which can be found at [10]. The code needs to be compiled from source using catkin

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ethz-asl/vicon_bridge.git
$ rosdep install -r --from-paths vicon_bridge/
$ cd ~/catkin_ws/
$ catkin_make
```

The ground station communicates with the PX4 using mavros [8], this can be installed using the package manager:

```
$ sudo apt-get install ros-kinetic-mavros
↪ ros-kinetic-mavros-extras
```

5.5.2 Setting up 3DR radios in Mission Planner

1. Connect one of the 3DR radios[3] to the computer running Mission Planner via USB.
2. Open the Hardware tab, and select 3DR radio.
3. In the top right, select the correct COM port for the 3DR radio and the select its current baud rate (if you don't know, try the next step with different ones until it works). Do not click connect.
4. Click the green "load settings" button at the top.
5. Set the desired settings. In particular choose the desired baud rate and Net ID.
6. If the 3DR radio is already paired (and its pair is powered), you can copy over the appropriate settings using the "copy required items to remote" button at the bottom.
7. Click the "save settings" button at the top and disconnect the radio from the computer.
8. If the radio is not paired, connect and set-up another 3DR radio in the same way. Ensure that its baud rate and Net ID are the same as the first radio.

[3]Other radios with SiK firmware will likely be able to be set up using these instructions, but have not been tested

5.5.3 Connecting Pixhawk to QGroundControl

The Pixhawk can be connected to QGroundControl directly via USB or by 3DR radio. Usually it connects automatically, but sometimes the connection needs to be set-up when it is connected via 3DR radio. These connections are managed in the application settings tab (Q at the top left), and selecting comm links. Then the add button at the bottom can be used to set-up a new connection. The baud rate will need to be set correctly, it can be found via Mission Planner as described in Section 5.5.2.

5.5.4 Configuring sensors in QGroundControl

QGroundControl will probably throw an error message about the mag sensors not being properly calibrated. This can be solved by going through its sensor calibration procedure, under the calibrate tap in settings. There are a number of sensors that need to be calibrated – compass, gyroscope and accelerometer. If the quadcopter has a bad crash and/or the Pixhawk is moved, the quadcopter will likely have to be re-calibrated. If the sensors are not calibrated the quadcopter will not arm.

5.6 Setting up transmitter and receiver

The transmitter used is the Spektrum DX8, which has 8 channels, 30 modes and 2 types. The receiver is the Spektrum AR8000. The specific choice of transmitter and receiver is not important, as long as they can bind together and have 8 channels. This section will describe the set-up and binding procedure for the Spektrum DX8 and AR8000. For more details see the Spektrum DX8's manual [54].

To set-up the transmitter, hold down the side-scroll button while turning the transmitter on. The following settings are used for the switch select:

Trainer: Inh Mix: Inh F Mode: Inh Aux2: Aux2 Gear: Aux1

R Trim: Inh L Trim: Ihn Knob: Aux3 Flap: Gear

Note that the transmitter and receiver need to be re-bound after changing the transmitter settings.

To bind the transmitter and receiver they need to be put in binding mode. First, turn both the transmitter and receiver off. The receiver can be put in binding mode by inserting a binding plug into the bnd/dat port before turning it on. The receiver can be

powered by connecting it to the quadcopter, via the Pixhawks' "RC in" port, or by using an external battery. Either way the power source can be connected using any channel other than the bnd/dat. The satellite receiver should be connected to the SPKT port on the main receiver or on the Pixhawk if the quadcopter is being used for power. The LED on the receiver will blink when it is in binding mode. Once the receiver is in binding mode, the transmitter should be put in binding mode. This is done by holding down the trainer bind button while turning it on.

The receiver and transmitter should now be bound, as indicated by the LED on the receiver becoming solid. After the receiver and transmitter than have been bound, the binding plug is removed from the receiver. Both the receiver and transmitter can now be powered down and will remain bound. Note that the LED on the receiver will light up when the bound transmitter is turned on.

The transmitter can be configured in the radio tab under settings in QGroundControl (see Section 5.5). It can be used to assign channels and reverse controls.

5.7 Flying quadcopter

The quadcopter is flown using automated trajectories, as following predetermined trajectories when flying the quadcopter manually requires a significant amount of training.

These trajectories are programmed as part of a ROS package using C++ (see Appendix 10.6). The desired pose of the quadcopter (x, y and z coordinate and a yaw) is defined in terms of time. Then the difference between the quadcopter's actual pose (found using the Vicon system) and the desired pose is calculated (i.e. actual - desired) for the x, y and z coordinates, these values are denoted xic, yic and zic, respectively. zi and the desired yaw can be used directly, but changing the yaw means that the x and y axes move. Thus the change in x and y is worked out by first finding the difference in the current yaw and the angle between xic and yic (angle_diff = yaw-atan2(yic, xic)). Then the magnitude of this movement is found (vxy_mag=$\sqrt{xic^2 + yic^2}$). Then the desired change in x is given by vxy_mag*cosf(-ang_diff) and the desired change in y is given by vxy_mag*sinf(-ang_diff). These desired changes are sent to the Pixhawk which adjusts the rotor speeds accordingly.

Even though the trajectories are automated the transmitter is still needed to arm and land the quadcopter, as well as to stop it if necessary.

5.8 System

Figure 4 shows a high-level overview of the data collection system. The computational platforms are marked in blue, and the method of communication is marked on the arrows.

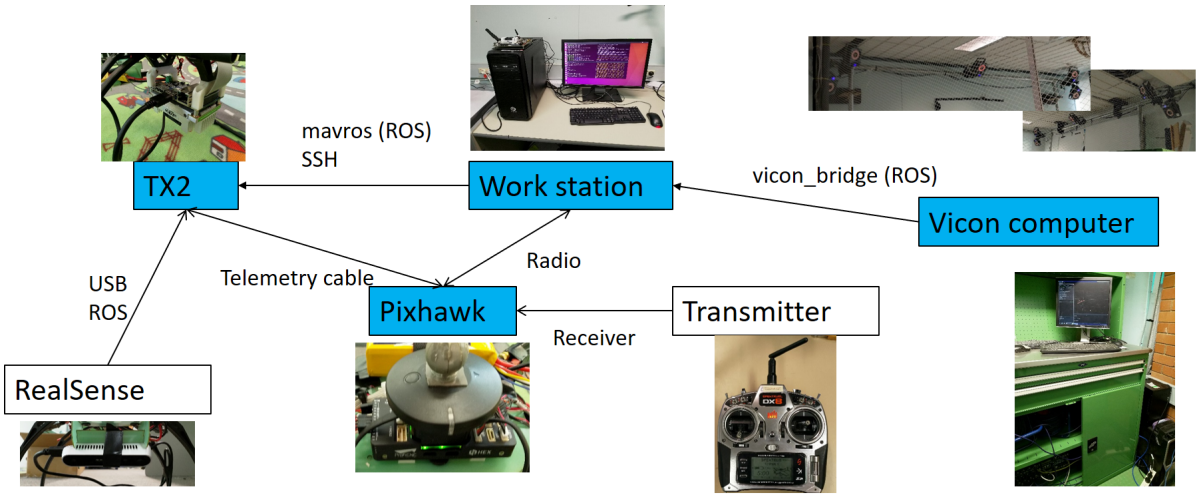


Figure 4: Overview of data collection system

6 Data collection

There are three types of data that need to be collected while flying:

- Images from the RealSense camera (both RGB and depth).
- Control inputs from the TX2 to the Pixhawk (flight controller for quadcopter).
- Ground truth data: position from Vicon, orientation from the gyroscope on the Pixhawk.

In addition the dimensions and layout of the scene are needed to use as ground truth data for the registered point cloud. Ideally odometry data would also be captured for use in a SLAM algorithm with the images. However, getting the position of the quadcopter can only be done using the Vicon (which is used for ground truth, but is not available in most environments), GPS (which is generally not used for SLAM and is not available in some environments) or via aerodynamics (which is difficult and not very accurate). Thus only visual odometry will be used.

The data is acquired via ROS and saved into a rosbag on the TX2. The RGB and depth data is acquired using the realsense2_camera package. The ground truth data is acquired using a combination of the Vicon data (transmitted using the bridging code described in Section 5.5.1) and the gyroscope data from the Pixhawk. The control data is published from the code which controls the quadcopter's trajectory.

Some initial experiments were done with an older RealSense camera, the R200, which was found to have very noisy depth readings. These experiments also found that shiny boxes are not picked up well by the RealSense cameras, which was used to choose suitable scenes later. More details can be found in Appendix 10.4.

6.1 Method for capturing data using quadcopter with automated trajectory

The quadcopter is flown using a Vicon assisted velocity controller program. In order for this program to work the following steps must be completed:

1. Turn on Vicon system.
2. Set-up scene and quadcopter. The quadcopter should be facing the scene 1m back (at the $-1x, 0y$ position as defined by the Vicon), it should also be pointing forwards as defined by the Vicon system.
3. Ensure that the quadcopter is showing up on the Vicon system. Tune the strobing and threshold if necessary to remove reflected landmarks. If tuning is not sufficient then try using electrical tape to cover reflective parts of the quadcopter.
4. Turn on and arm the quadcopter.
5. Connect ground station computer to the Vicon via ethernet and ensure that the radio is plugged in and connected to its pair on the quadcopter.
6. Turn on the controller, ensuring that everything is in the zero position, be especially careful to check that it is manual mode.
7. Run `roslaunch mavros px4.launch`
8. In a new terminal run `roslaunch vicon_bridge vicon.launch`.

9. In a new terminal run `sh connect_tx2_realsense.sh` (ensure that the TX2 and ground station computer are connected to the same network).

10. In the ssh terminal run `sh run_vicon.sh`. This will open up a number of terminal windows including ones for capturing relevant data using ROS and ones for flying the quadcopter. The last one should say something along the lines of "in hover mode". This has started the velocity controller, so the quadcopter needs to be launched within 30 seconds.

11. Use the controller to arm the quadcopter and flick it into autonomous mode to launch.

12. Once the quadcopter has completed its loops, it will hover in place for a bit and then descend. Once it has reached the ground the controller should be switched back into manual mode and then used to disarm the quadcopter.

6.2 Datasets

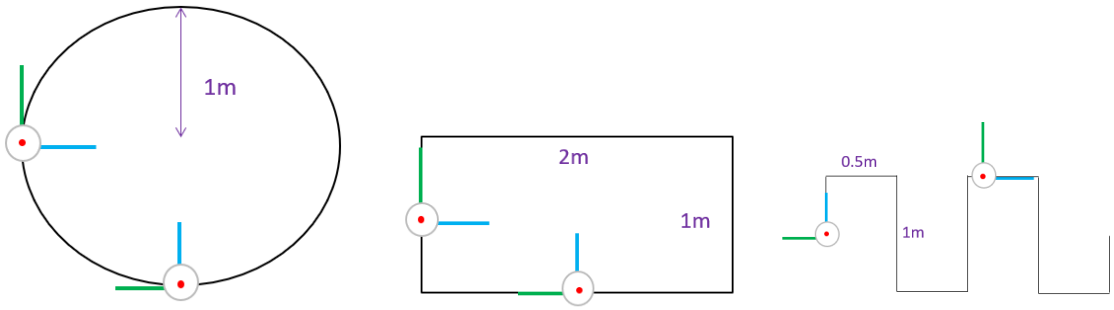
There are two scenes, shown in Figure 5 and three trajectories: circle, rectangle and lawnmower, shown in Figure 6.

Scene 1 has a few objects that are highly textured, with not much texture elsewhere. Scene 2 has multiple objects, most of which are highly textured, and the ground is also very textured. Scene 2 is more textured than most environments that would be encountered in practice, but it was deemed necessary in order to ensure enough RGB features are visible in all frames for the rectangle and lawnmower trajectories. Scene 1 is more similar to

(a) Scene 1

(b) Scene 2

Figure 5: Scenes used for data collection



(a) Circle trajectory, two loops

(b) Rectangle trajectory, two loops

(c) Lawnmower trajectory, mirrored trajectory back

Figure 6: (Best viewed in colour). Trajectories used for data collection. The orientation of the quadcopter is shown using the quadcopter frame, with the blue axis being the front of the quadcopter. All trajectories have a height of 1.5m.

environments that would be encountered in practice, although how similar it is depends on the application.

A circle (with the quadcopter looking inwards) is chosen as the starting trajectory due to the fact that it is simple while still providing coverage of the entire scene from a variety of angles. There is a high level of overlap between frames and the boxes are always in view. The scene is circled twice as it is expected that some details will be missed on each pass and this also allows for the possibility of multiple loop closures. Initially a radius of 1.2m and a height of 1.2m was used for the trajectory, but it was found that the quadcopter flew too close to the walls of the flying space, increasing the likelihood of a crash, and the top of the boxes were not visible. Thus this was changed to a radius of 1m and a height of 1.5m.

A rectangle is chosen as the second trajectory as they decouple the translation and rotation, which allows the performance of the algorithms for each type of motion to be evaluated separately. For this trajectory, the quadcopter also looks inwards, perpendicular to the direction of motion. However, unlike with the circle trajectory the quadcopter does not look at the same point for the whole trajectory, and the boxes in scene 1 do not remain in the field of view when the quadcopter is close to the corners of the rectangle. Thus this trajectory is only used with scene 2.

Dataset	Total amount of translation (m)	Total amount of rotation (rad)	Total time (s)
Circle 1	11.87431077970471	11.928861468329714	69.88999987
Circle 2	13.404915561036152	12.419477223200264	75.93000007
Rectangle	14.972810856470433	17.207008968439762	150.51999998
Lawnmower	20.51718719703204	38.382665866625544	194.96000004

Table 1: Amount of motion and time for each dataset (only including the part of trajectory used for registration).

Four datasets are collected:

- Circle 1: Circle trajectory with scene 1
- Circle 2: Circle trajectory with scene 2
- Rectangle: Rectangle trajectory scene 2
- Lawnmower: Lawnmower trajectory with scene 2

The total amount of translation (sum of Euclidean distances between frames), total amount of rotation (sum of magnitude of angle of rotation between frames) and total time are given in table 1.

7 Implementation Details

This Section gives a high-level overview of the code used for the data registration. Section 7.1 explains the choice of language, and gives the versions of the libraries used. Section 7.2 gives an outline of the registration code; further details and the code itself are given in Appendix 10.5.

7.1 Language and libraries for registration

There were three main choices of language considered for implementing the registration algorithms in:

1. MATLAB: easy to prototype in, built-in registration tools, cannot run on-board, limited audience due to cost

2. Python: easy to prototype in, OpenCV support, theoretically could run on-board (but slow), widely used in computer vision

3. C/C++: hard to prototype in, OpenCV support, good for running on-board, widely used in computer vision

Thus MATLAB is only used for the initial investigation into the data using its exiting tools. The registration algorithms are implemented in Python, with OpenCV implementations used where available (see Section 4.4).

Python 3.5.5 is used through anaconda. The following libraries are used:

1. OpenCV 3.1.0

2. numpy 1.15.1

3. matplotlib 2.2.2

4. pyquaternion [30]

7.2 Code outline

For the registration, all methods followed the same basic structure:

1. Get the RGB image names (which give their timestamps), set starting point and amount of frames to skip

2. Extract the position and orientation data from the ground-truth

3. Align timestamps by finding closest depth and ground truth timestamps to each RGB timestamp

4. Initialize trajectory using first aligned ground truth position and orientation

5. Loop over RGB images, registering each with the next one using the following steps:

   (a) Extract and match RGB features using OpenCV (SIFT features)

   (b) If registration method uses 3D points, convert features to 3D using depth image

(c) Do registration, timing how long it takes

(d) Find new pose of quadcopter in global frame (see Section 1) and add it to the trajectory

6. Save trajectories and times

For more detail see Appendix 10.5.

8 Results

This section compares the following three registration methods: Essential Matrix, Kabsch and PnP. They are evaluated on the four RGB-D datasets described in Section 6, all of which are captured from a D415 RealSense camera mounted to a quadcopter. The methods require a certain amount of movement between frames in order to work properly – too much and there is not enough overlap to get good point matches, too little and numerical errors have a large impact. Thus before the methods can be compared, the optimal processing frequency for each needs to be found, the results can be found in Section 8.2. Then the methods can be compared in terms of their absolute and relative error (see Section 8.1 for an explanation of these errors and Section 8.3 for the comparison), as well as their computational time (see Section 8.4).

8.1 Relative and absolute error

Before the results can be be discussed, the relative and absolute error have to be defined. Relative error is the error for a single step of the registration, it is found by applying the estimated transform to the previous ground truth pose. Absolute error is the error for the estimated trajectory at each time-step, it is found by applying the estimated transform to the previous estimated pose. See Figure 7 for visualization.

The errors themselves are calculated separately for the rotation and translation, as this allows for better intuition as to what they mean. The translation error is the L2 norm of the translation (which is the Euclidean distance between the start and end points). For a translation $t = (t_x, t_y, t_z)$ this error is given by

$$E_t = \sqrt{t_x^2 + t_y^2 + t_z^2}$$

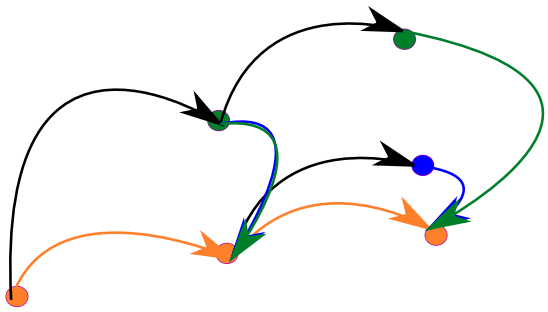


Figure 7: (Best viewed in colour). Relative (blue) and absolute (green) error. The black arrows show the estimated transform and the orange arrows show the ground truth transform.

The rotation error is the magnitude of the angle of the rotation in axis-angle form. This is calculated using the pyquaternion package.

8.2 Circle 1 dataset: finding optimal data frequency

The quadcopter traveled at a fairly constant velocity when capturing the data, so the captured images can be sub-sampled to vary the amount of motion between images being registered. The relative pose error is more suitable for use in determining an appropriate processing frequency, as the absolute error accumulates drift and numerical errors which are likely affected by the number of frames registered and thus the processing frequency. These factors are important to take into consideration for a mapping application. However, this section is solely focused on selecting a processing frequency that works best for each method, as there are other ways for mapping algorithms to deal with drift and other accumulated errors (e.g. closing the loop).

The processing frequencies are varied by recording data at a high frequency, then skipping a certain number of frames during registration. To see how this skip number corresponds to the amount of motion and time for each dataset, refer to table 1 in Section 6.2, page 34.

Table 2 shows the errors for the relative trajectories found using the Essential Matrix, Kabsch and PnP (iterative) methods. None of the methods have the best mean rotation and the best mean translation at the same frequency. The frequencies are relatively close for the Essential Matrix method, but far apart for PnP. PnP uses both RGB and depth data, whereas the Essential Matrix and Kabsch only use one type each. Incorporating

Method	**Images skipped**	**Mean rotation (rad)**	**Standard deviation rotation**	**Mean translation (m)**	**Standard deviation translation**
Essential Matrix	20	0.4894	0.1564	**0.1860**	0.0545
	30	**0.3210**	0.1866	0.2715	0.0768
	40	0.9839	0.5943	0.3231	0.0893
	50	0.4587	0.1806	0.3986	0.0978
	60	0.7676	0.2862	0.5786	0.1229
	70	0.4189	0.1678	0.3671	0.0912
	80	1.1206	0.2977	0.4390	0.1082
	90	1.2945	0.6580	0.4803	0.1184
Kabsch	20	1.0833	0.4862	0.5733	0.8449
	30	1.3750	1.1153	0.6569	0.7853
	40	1.6761	0.9486	0.6968	0.9587
	50	1.0069	0.2896	0.5875	0.6919
	60	0.8842	0.3332	0.5836	0.4071
	70	1.6081	0.7557	**0.5121**	0.2411
	80	1.4819	1.0692	0.9134	1.1902
	90	**0.7386**	0.4512	0.9596	0.8742
PnP (iterative)	20	0.6246	0.1879	**0.1654**	0.1000
	30	0.5321	0.1672	0.2498	0.1534
	40	0.7845	0.2815	0.3059	0.1689
	50	0.7922	0.2451	0.3926	0.2385
	60	0.9364	0.3424	0.4287	0.1390
	70	**0.3606**	0.0971	0.5367	0.3136
	80	1.1450	0.9139	0.5945	0.2970
	90	1.5099	0.9690	1.0761	0.9652

Table 2: Realtive errors for each registration method on circle 1 dataset, with varying numbers of frames skipped. The rotation error is in radians, and the translation error is in meters.

depth data can give a larger spread of points than just using RGB data, especially when the features are clustered close together in the image plane. Also, the Essential Matrix method needs 5 point correspondences, but both Kabsch and PnP can work with 3. Thus methods which incorporate depth information may require less overlap than those that only use RGB data, and thus work better for larger motions. Also, the depth data is much noisier than the RGB data (see Section 8.5). So it could be the case that high processing frequencies work best when the data is relatively noise-free but lower processing frequencies are better for noisy data. This would make sense as size of the error relative to the motion decreases as the motion gets bigger.

As the best mean rotation and translation do not coincide, it makes deciding on the best frequency for each method a bit difficult. For the Essential Matrix, the mean rotation for skip size 20 is 1.5246 times the best, and the mean translation for skip size 30 is 1.4597 times the best. Thus, skip size 30 is selected as the best. For Kabsch, the mean rotation for skip size 70 is 2.1772 times the best, and the mean translation for skip size 90 is 1.8739 times the best. For skip size 60, the mean rotation is 1.1971 times the best, and the mean translation is 1.1396 times the best. These are quite close to the best values, whereas the rotation for skip size 70 and the translation for skip size 90 are quite far off. Thus skip size 60 is selected as the best. For PnP, the mean rotation for skip size 20 is 1.7321 times the best, and the mean translation for skip size 70 is 3.2449 times the best. For skip size 30, the mean rotation is 1.4756 times the best, and the mean translation is 1.5103 times the best. Thus the skip size of 30 is selected as the best, but it is much closer than for the other methods.

A few variations on these methods were also investigated, an alternate implementation of PnP (EPnP) and Kabsch but using inliers from the Essential Matrix method or PnP (iterative) instead of using RANSAC. The alternate inliers for Kabsch are used in order to investigate if the RANSAC implementation is (one of) the cause(s) of Kabsch performing poorly. The results are shown in table 3.

It can be seen that both variations of PnP produce extremely similar results, as their mean and standard deviation for the rotation and translation error is the same up to 4 decimal points.

This table also shows that the Kabsch algorithm can produce better results when given points found to be inliers, indicating that RANSAC is not finding suitable inliers. However

the results are still worse than PnP for the translation, and the rotation is not much better (1.1174 times). Also the standard deviation is much worse for the translation, indicating a that the results are not conclusive. Kabsch produces better results when using the inliers from PnP than the Essential Matrix. This is somewhat expected as the PnP method uses 3D data like Kabsch, whereas the Essential Matrix method only uses RGB data. Also, note that the optimal processing frequency is different for Kabsch with different inliers (following the same procedure as in Section 8.2). This difference in optimal frequency likely arises from the quality of the inliers being affected by the frequency. Although interestingly the optimal frequency for Kabsch with the Essential Matrix inliers is higher than the optimal frequency for the Essential Matrix method, even though Kabsch works better with a lower processing frequency.

8.3 Circle 1 dataset: Investigating trajectories

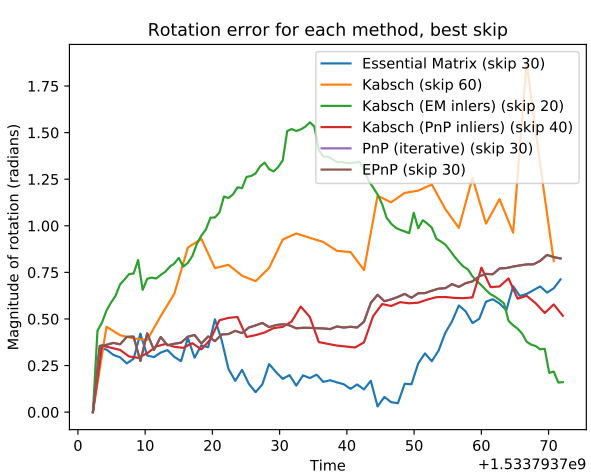
Now that the best processing frequency has been found for each method (see Section 8.2), the trajectories can be plotted (Figures 9), as well as the associated errors over time (Figure 8). Note that the errors for the iterative PnP method and EPnP are so similar that they overlap on the error graphs. For the trajectories, only the Essential Matrix method, Kabsch, Kabsch with PnP inliers and iterative PnP are shown. This is because EPnP has been found to have almost identical results to iterative PnP, and as can be seen

Method	**Images skipped**	**Mean rotation (rad)**	**Standard deviation rotation**	**Mean translation (m)**	**Standard deviation translation**
Kabsch	60	0.8842	0.3332	0.5836	0.4071
Kabsch (EM inliers)	20	0.9214	0.3664	**0.4775**	0.8699
Kabsch (PnP inliers)	40	**0.4762**	0.1397	0.5817	0.7004
PnP (iterative)	30	0.5321	0.1672	0.2498	0.1534
EPnP	30	0.5321	0.1672	0.2498	0.1534

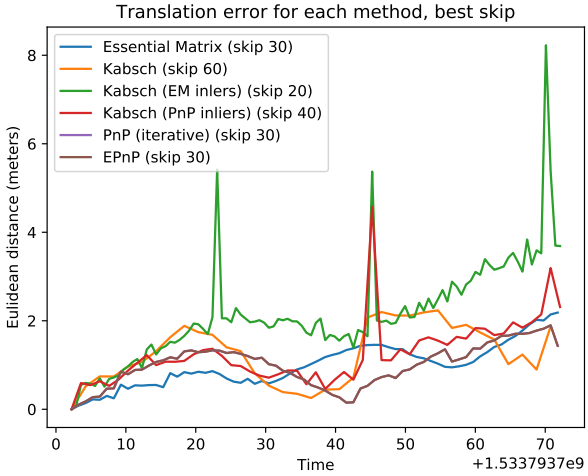
Table 3: Relative errors for variations on each registration method on circle 1 dataset, with best amount skipped. The errors for the main methods are also repeated here to ease comparison. The rotation error is in radians, and the translation error is in meters.

in Figure 8, Kabsch with PnP inliers has less error than Kabsch with Essential Matrix inliers.

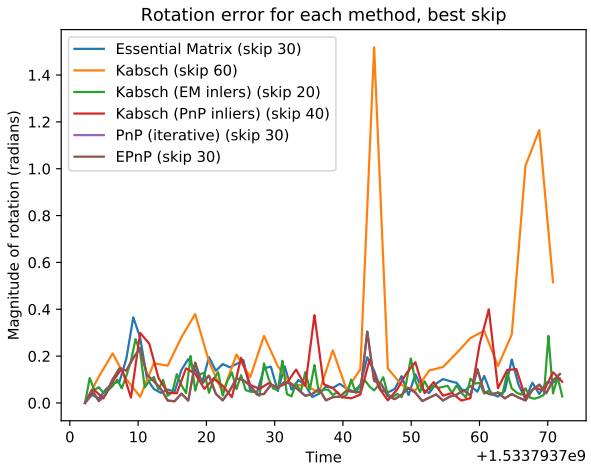
In Figure 8, the presence of drift over time can be seen in the absolute translation error. It is slowly increasing, although although all of the methods have some dips. The Essential Matrix method has two, whereas the Kabsch and PnP methods have one in about the same spot. Kabsch with inliers from the Essential Matrix method or PnP also has some large spikes in the absolute where the relative translation error is bad. Interestingly, the error comes down again after these spikes which indicates that these methods are then finding the wrong transformation for the following frames, but that it is somehow correcting for the previous wrong estimate. Upon closer inspection, the



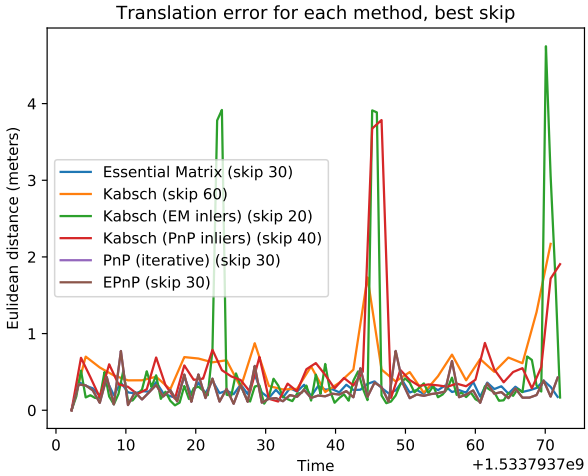
(a) Absolute rotation error



(b) Absolute translation error

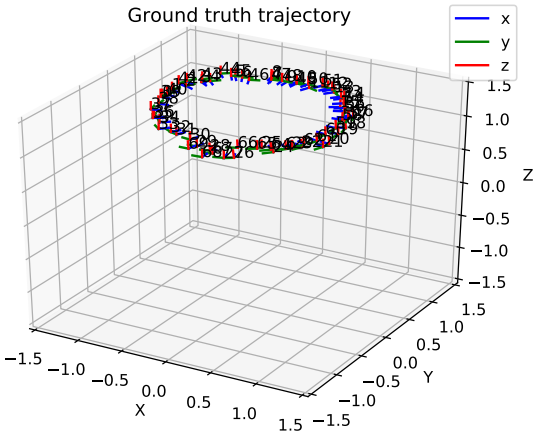


(c) Relative rotation error

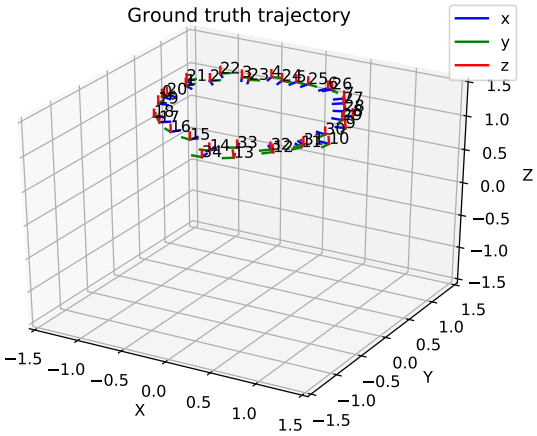


(d) Relative translation error

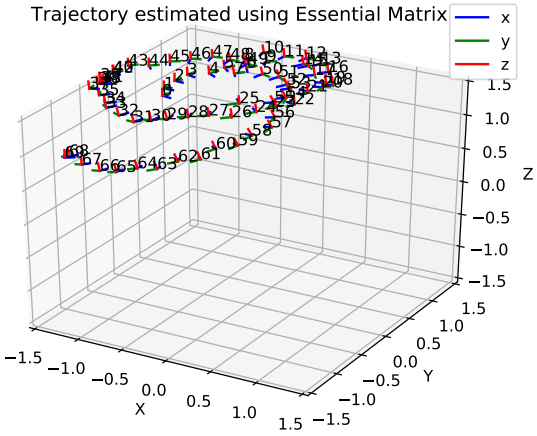
Figure 8: (Best viewed in colour). Vision error for each registration method on the circle 1 dataset, with best skip amount.



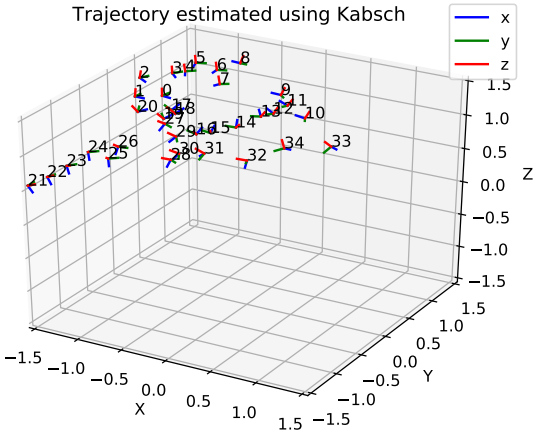
(a) Ground truth trajectory (skip 30)



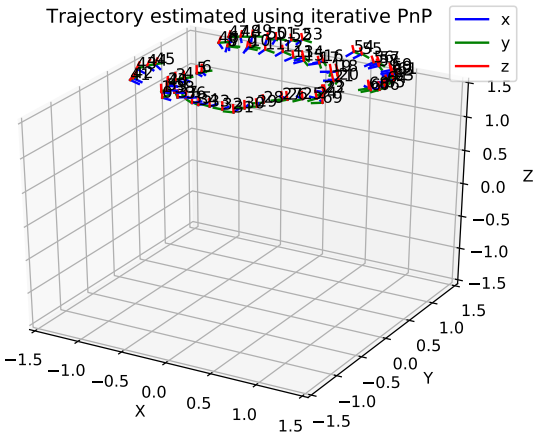
(b) Ground truth trajectory (skip 60)



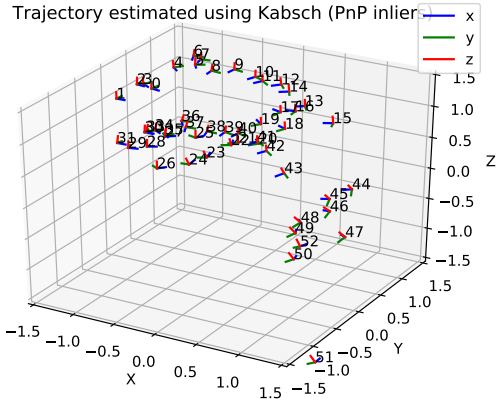
(c) Essential Matrix (skip 30)



(d) Kabsch (skip 60)



(e) PnP (skip 30)



(f) Kabsch (PnP inliers) (skip 40)

Figure 9: Estimated trajectories for each registration method on the circle 1 dataset, with best skip amount.

Kabsch method that uses RANSAC also increases around this point, but it does not decrease much afterwards. Also, looking at the spikes in the relative translation error, it can be seen that two out of three of them occur in roughly the same spot for all three variations of Kabsch, indicating that they occur due to the Kabsch algorithm as applied to those frames. Looking at the relative rotation error reveals spikes in the rotation error corresponding to those in the translation error for the Kabsch method that uses RANSAC but not the two that use other inliers. Note that the Kabsch algorithm finds the translation first, and then the rotation. Thus it is expected that a poor estimate of the translation will coincide with a poor estimate of the rotation if the data is not too noisy.

Figure 9 shows the trajectories. For PnP, the trajectory looks fairly similar to the ground truth, but with some sections starting in the wrong spot. For the Essential Matrix the trajectory is also still fairly close to the ground truth, but trajectory does not lie flat on the $x - y$ plane, from the way the trajectory tilts it seems likely that the estimated rotations have some bias which is accumulating. The Kabsch trajectory looks alright until it reaches step 21, at which point it is very far from the ground truth. Kabsch with PnP inliers performs alright at first, but around step 30 it greatly diverges from the ground truth.

8.4 Circle 1 dataset: computational time

Table 4 gives the computational time for each method. Note that these times were only taken over one run, so they could be effected by things like what else the computer is doing at the time and random variations. However, care was taken to keep the level of activity on the computer constant while running all of the algorithms. For the Essential Matrix and Kabsch methods, most of the time is taken in finding the features (71.8% and 77.1%, respectively). The PnP methods are much slower, the average time per registered frame is 5.6 times slower than the Essential Matrix method and 7.2 times slower than Kabsch. This difference comes from the PnP method itself, which takes up 86.77% and 86.76% of the computational time for PnP (iterative) and EPnP respectively.

Note that the PnP method does use RANSAC, which is included in the method computational time. So as the depth data is noisy (see Section 8.5), the method may be

Method	**Images skipped**	**Computational time (s)**				
		Finding features	Finding 3D	Method	Total	Average
Essential Matrix	30	20.5341	0	8.0612	28.5953	0.4085
Kabsch	60	8.5676	1.1311	1.4212	11.1200	0.3177
Kabsch (EM inliers)	20	24.2366	2.8234	0.0180	27.0780	0.25789
Kabsch (PnP inliers)	40	15.5132	2.1325	0.01208	17.6578	0.3396
PnP (iterative)	30	20.5635	0.6215	138.9488	160.1338	2.2876
EPnP	30	20.5765	0.6047	138.7476	159.9288	2.2847

Table 4: Computational time for each registration method. Note that the time for Kabsch using other inliers does not include the time used to find said inlers. There are 2100 frames without skipping.

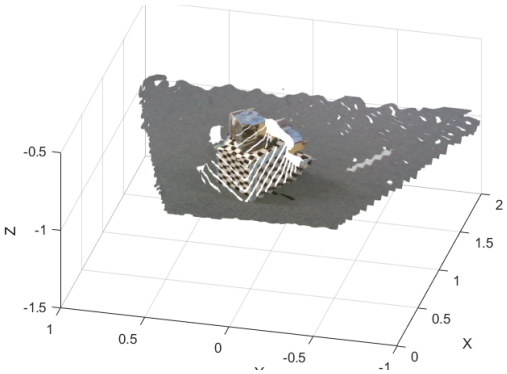
taking a long time in order to find suitable inliers. It would be interesting to see how the computational time changes if used on a dataset with better depth measurements. The fact that this is not happening for Kabsch may be an indication that its associated RANSAC implementation is not functioning as desired, and may achieve more accurate results if given stricter thresholds.

8.5 Circle 1 dataset: Investigating Sources of Error

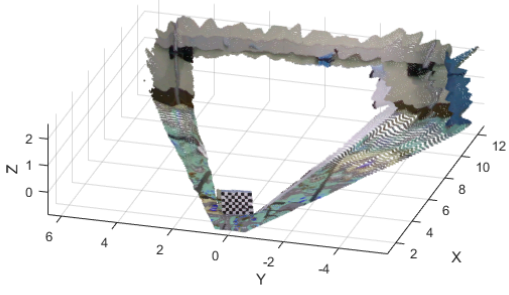
As has been found in Sections 8.2 and 8.3, the Kabsch implementation performs much more poorly than the Essential Matrix or PnP algorithms. There are three main possibilities for the source of this error: the data, the algorithm itself and the RANSAC implementation. It has been found in the previous sections that using the inliers found using either PnP or the Essential Matrix method results in a slight improvement in the error however the results are still far below that of the other methods. Thus the RANSAC implementation needs some tuning but is likely not the only issue. The algorithm implementation was tested by using it to register synthetic data without noise. That is, some points were randomly generated and then a known rotation and translation were applied. The Kabsch algorithm recovered this transformation with error of magnitude 10^{-15} for

both the rotation and translation. Thus, the algorithm itself appears to be implemented correctly. When noise is introduced to the synthetic data, the recovered transform has error of magnitude approximately proportional to the magnitude of the error. This is the case even when the noise is applied to only half of the data-points, indicating that the RANSAC implementation is not correctly filtering out the outliers. Whether this is due to the implementation itself or the thresholds chosen is unclear. The data also needs to be investigated to see how much noise is present and the possible sources of said noise.

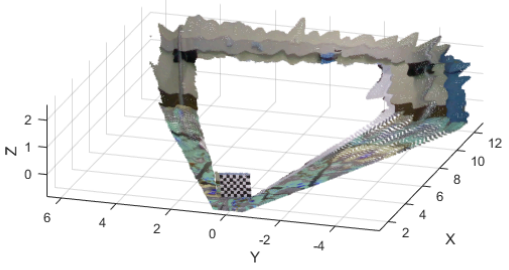
Figure 10 shows various point clouds created using the RGB-D along with the projection algorithms described in Sections 4.2 and 4.3. The scaling seems to be about a factor of 2 off for the point clouds taken with a still RealSense, however that is not important for the purposes of this analysis. Figure 10a shows a point cloud created using data taken from the RealSense while the quadcopter is flying, with the Vicon on. Note the lines going through the boxes, the waves in the floor and the fact that some of the floor near the box



(a) Example point cloud taken from RealSense attached to quadcopter



(b) Point cloud taken of scene from still RealSense camera, with Vicon off



(c) Point cloud taken of scene from still RealSense camera, with Vicon on

Figure 10: Investigating sources of error in depth measurements via visualizing point clouds. Note that all point clouds have been aligned with the quadcopter-fixed frame.

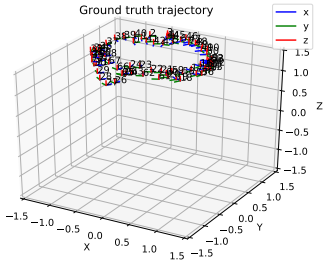
edges has been textured with part of the boxes' texture (on the right). As the RealSense uses IR to get the depth measurements, it is hypothesized that the Vicon, which also uses IR, could be interfering with the measurements. Thus, measurements were taken of a static scene with and without the Vicon on so they could be compared. The RealSense camera was also still for these measurements. As can be seen, the point clouds taken with the RealSense camera still are basically indistinguishable by eye. Thus the issues are not due to the Vicon. Comparing these point clouds to the one taken from the quadcopter it appears that some of the issues are native to the RealSense camera, and some may be due to either the motion or the angle of the the camera relative to the object(s) being captured. The wavy pattern still appears in the point clouds from the still RealSense, it starts about 3m back from the camera. The texturing at the edge of the box is also wrong for these point clouds. As with the point cloud from the moving camera, on the left some of the texture from the floor is on the box, and on the right some of the texture from the box is on the floor. This indicates that the depth and RGB images were not correctly aligned. However, the lines through the boxes that are present in the point cloud from the moving camera are not present in the other two point clouds. This could be due to the movement (especially as the RealSense D145 has a rolling shutter). However the boxes are also at an angle for that point cloud, whereas the other point clouds have the camera facing the box head-on, so it could be due to the angle.

Interestingly, PnP also uses the 3D data but it still produces results comparable to the Essential Matrix. As found in Section 8.4, it does take much longer than either the Essential Matrix method or Kabsch to do so. It is possible that this is due to RANSAC needing to run for a long time in order to find appropriate inliers.

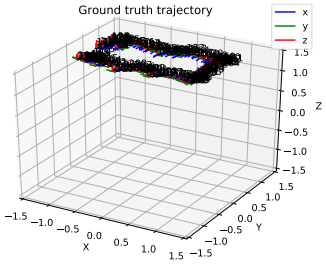
8.6 Different Trajectories

The choice of trajectory for mapping can also be very important. Thus this section investigates how well the registration methods perform with different trajectories, as described in Section 6.2. Note that the scene is also different to the one used for the circle 1 trajectory, so another circle trajectory is also investigated with the new scene to allow for a more easy comparison with the other trajectories.

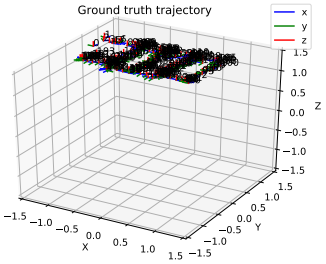
Figure 11 shows the best estimates for each trajectory for each of the methods. Note



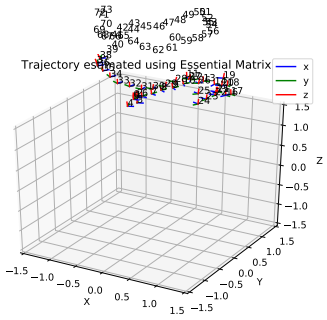
(a) Ground truth trajectory circle 2 (skip 30)



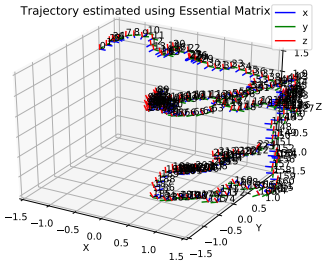
(b) Ground truth trajectory rectangle (skip 20)



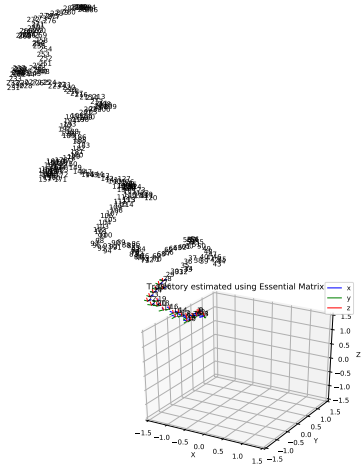
(c) Ground truth trajectory lawnmower (skip 30)



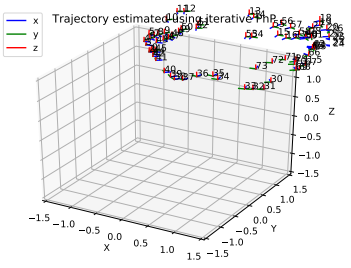
(d) Essential Matrix (skip 30)



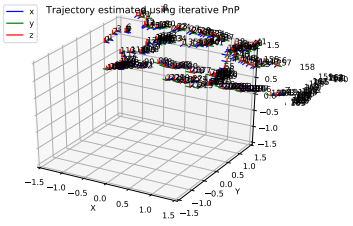
(e) Essential Matrix (skip 20)



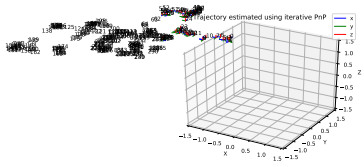
(f) Essential Matrix (skip 20)



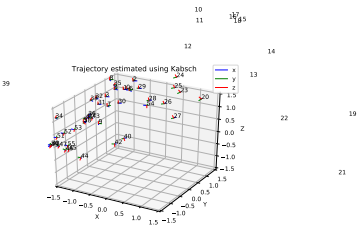
(g) PnP (skip 30)



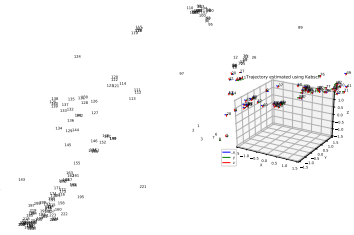
(h) PnP (skip 20)



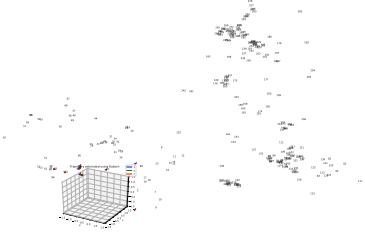
(i) PnP (skip 20)



(j) Kabsch (skip 40)



(k) Kabsch (skip 20)



(l) Kabsch (skip 20)

Figure 11: Estimated trajectories for each registration method on the circle 2, rectangle and lawnmower trajectories dataset, with best skip amount.

that the variations are not shown as the previous sections found that there was no significant difference between iterative PnP and EPnP and that Kabsch does not perform well even with inliers from another method. Kabsch performs very poorly for these new datasets. The parameter tuning for RANSAC was done on the circle 1 dataset, so this may indicate that the parameters found are not very robust.

Looking at the ground truth for the circle 2 trajectory, it can be seen that it is not a perfect circle, in particular it is a bit too far inwards at the start and has a bit of a dip near the end. This helps to explain why the results are not as good as they were for the circle 1 trajectory. Note that the circle 2 dataset also has a different scene to the circle 1 dataset. The scene for the circle 2 dataset has more texture, with similar boxes to the first scene but a highly textured carpet. Generally more texture would be expected to improve results, as it gives the registration algorithms more data to work with. However, in this case the texture is added to the carpet, which is planar and the Essential Matrix does not perform well for planar scenes [36], so the added features from the carpet could be degrading performance.

For the rectangle trajectories for the Essential Matrix method (Figure 11e) and PnP (Figure 11h), it can be seen that the straight-motion sections are estimated relatively well, but the methods fail near the corners.

For the lawnmower trajectories for the Essential Matrix method (Figure 11f) and PnP (Figure 11i), the estimates start heading in the wrong direction almost immediately. A few different starting points were also tried that exhibited the same behaviour, so it was likely due to the scene and/or type of motion. During this part of trajectory none of the objects were in view of the camera, so the methods needed to rely on the carpet for features, the carpet sections have the same pattern so it is possible that part of one section was matches with that part in another section, resulting in the poor estimation. Other than at the start, the straight-motion sections seem to be estimated relatively well, but the corners often go in the wrong direction.

For all of the trajectories in Figure 11, the z value for PnP remains fairly constant, whereas the Essential Matrix method tilts upwards. Thus it seems like having some kind of loop closure method is even more important for mapping when the registration method is the Essential Matrix method than when it is PnP.

Table 5 gives the average computational time for each of methods for the different

Dataset	Method	Images skipped	Average computational time (s)
Circle 2	Essential Matrix	30	0.4819
	Kabsch	40	0.5549
	PnP	30	0.9156
Rectangle	Essential Matrix	20	0.5558
	Kabsch	20	0.6627
	PnP	20	0.7904
Lawnmower	Essential Matrix	20	0.4283
	Kabsch	20	0.5032
	PnP	20	0.7120

Table 5: Computational time for each registration method with the different datasets.

datasets. PnP is slower than the Essential Matrix method and Kabsch for all three datasets. However the difference is not as big as for the circle 1 dataset. The average computational time for PnP is more than 2 times faster than for the circle 1 dataset and the Essential Matrix method and Kabsch are also slightly slower. The fact that scene 2 includes a highly textured carpet should result in more features being present, so it makes sense that the methods would be slower, as with the Essential Matrix method and Kabsch. However PnP is actually faster than it was for scene 1. It is possible that RANSAC is converging faster, so it could be that many of these new features are likely to be inliers for PnP but not the other methods. The Essential Matrix method does not work well for planar data [36], so it makes sense that these new features are less likely to be inliers. The RANSAC parameters for Kabsch were tuned on the circle 1 dataset, so it also makes sense that the RANSAC implementation will not perform as well for the other datasets.

9 Conclusion

The Essential Matrix method appears to be the best method to use for registering the circle 1 and 2 datasets. It has similar relative error to PnP, but with a much lower computational time. However, when used with the other trajectories PnP seems to be more accurate than the Essential Matrix method, although it is still slower. Thus PnP

seems to be the best choice for trajectories with corners when time is not a critical factor. The Essential Matrix method does produce good enough results to use if a faster method is needed (e.g. for real-time mapping) if used with a method for correcting for drift.

One of the reasons the Essential Matrix method worked well compared to Kabsch and PnP is that the depth data was much noisier than the RGB data. This will also likely be the case in most applications, as RGB sensors have been around for far longer and are much more widely used than depth sensors, so research into creating more accurate, affordable and compact RGB sensors has progressed further than for depth sensors and this is not likely to change in the foreseeable future. However, PnP achieved a similar level of error even using the noisy depth data. Thus, in an application with better depth data and/or noisier RGB data PnP may be a more suitable method. If there was also a large amount of movement between frames this becomes even more likely, as the Essential Matrix method performed best when there was not much movement between frames, whereas Kabsch performed better with larger movements for the circle 1 dataset.

It would be interesting to see how well the methods perform with less noisy depth readings. Unfortunately, doing so would require access to a camera with a better depth sensor. Some other interesting factors to test would be the effect of structure and texture. It would be expected that the methods that incorporate depth would perform better when structure is added, and the RGB-only methods would perform better when texture is added (although adding texture and/or structure should improve the results of all of the methods at least a bit).

As all of the registration methods tested used feature-matching to get the point correspondences, it would also be interesting to test how well they performed with different feature detectors and/or matchers.

This report was mainly focused on investigating different registration techniques. In order to use these techniques to produce an accurate mapping algorithm, many parts still need to be added. Some kind of filtering of the data to decide which frames to register is necessary. If the motion is not constant, then ideally this method should be able to dynamically alter the processing rate. At a minimum it should filter out blurry images. Then in order to reduce the effects of drift some kind of loop closure should be utilized. Finally, in order to create the map itself some kind of suitable representation of the scene needs to be constructed.

References

[1] M. Achtelik. vicon bridge, 2015. URL http://wiki.ros.org/vicon_bridge.

[2] ArduPilot dev team. Installing mission planner (windows), 2016. URL http://ardupilot.org/planner/docs/mission-planner-installation.html.

[3] A. Bartoli. Groupwise geometric and photometric direct image registration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(12):2098–2108, 2008.

[4] E. Brachmann, A. Krull, F. Michel, S. Gumhold, J. Shotton, and C. Rother. Learning 6d object pose estimation using 3d object coordinates. In *European conference on computer vision*, pages 536–551. Springer, 2014.

[5] K. G. Derpanis. Overview of the ransac algorithm. *Image Rochester NY*, 4(1):2–3, 2010.

[6] O. dev team. Camera calibration and 3d reconstruction, 2014. URL https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[7] DHood. Ubuntu install of ROS kinetic, 2017. URL http://wiki.ros.org/kinetic/Installation/Ubuntu.

[8] Dronecode. MAVROS. URL https://dev.px4.io/en/ros/mavros_installation.html.

[9] N. Engelhard, F. Endres, J. Hess, J. Sturm, and W. Burgard. Real-time 3d visual slam with a hand-held rgb-d camera. In *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*, volume 180, pages 1–15, 2011.

[10] ETHZ ASL. vicon bridge, 2015. URL https://github.com/ethz-asl/vicon_bridge.

[11] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[12] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng. Complete solution classification for the perspective-three-point problem. *IEEE transactions on pattern analysis and machine intelligence*, 25(8):930–943, 2003.

[13] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[14] L. Goldschmidt, P. K. Teng, R. Riek, and D. Eisenberg. Identifying the amylome, proteins capable of forming amyloid-like fibrils. *Proceedings of the National Academy of Sciences*, 107(8):3487–3492, 2010.

[15] R. M. Haralick, H. Joo, C.-N. Lee, X. Zhuang, V. G. Vaidya, and M. B. Kim. Pose estimation from corresponding point data. In *Machine Vision for Inspection and Measurement*, pages 1–84. Elsevier, 1989.

[16] R. I. Hartley. In defence of the 8-point algorithm. In *Computer Vision, 1995. Proceedings., Fifth International Conference on*, pages 1064–1070. IEEE, 1995.

[17] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. Rgb-d mapping: Using kinect-style depth cameras for dense 3d modeling of indoor environments. *The International Journal of Robotics Research*, 31(5):647–663, 2012.

[18] B. K. Horn, H. M. Hilden, and S. Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. *JOSA A*, 5(7):1127–1135, 1988.

[19] Intel. ROS wrapper for Intel RealSense devices, 2018. URL https://github.com/intel-ros/realsense/#installation-instructions.

[20] Intel RealSense team. Jetson TX2 installation, 2017. URL https://github.com/IntelRealSense/librealsense/blob/master/doc/installation_jetson.md.

[21] Intel RealSense team. Intel realsense SDK, 2018. URL https://github.com/IntelRealSense/librealsense.

[22] JetsonHacks. Jetsonhacks — developing for nvidia jetson, 2018. URL https://www.jetsonhacks.com/.

[23] W. Kabsch. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A: Crystal Physics, Diffraction, Theoretical and General Crystallography*, 32(5):922–923, 1976.

[24] W. Kabsch. A discussion of the solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A: Crystal Physics, Diffraction, Theoretical and General Crystallography*, 34(5):827–828, 1978.

[25] W. Kabsch. Automatic indexing of rotation diffraction patterns. *Journal of Applied Crystallography*, 21(1):67–72, 1988.

[26] W. Kabsch. Xds. *Acta Crystallographica Section D: Biological Crystallography*, 66 (2):125–132, 2010.

[27] kangalow. Intel realsense camera librealsense - NVIDIA jetson TX dev kits, 2017. URL http://www.jetsonhacks.com/2017/08/14/intel-realsense-camera-librealsense-nvidia-jetson-tx-dev-kits.

[28] L. E. Kavraki. Geometric methods in structural computational biology. 2009.

[29] C. Kerl, J. Sturm, and D. Cremers. Dense visual slam for rgb-d cameras. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 2100–2106. IEEE, 2013.

[30] KieranWynn. pyquaternion, 2018. URL https://github.com/KieranWynn/pyquaternion.

[31] V. Lepetit, F. Moreno-Noguer, and P. Fua. Epnp: An accurate o (n) solution to the pnp problem. *International journal of computer vision*, 81(2):155, 2009.

[32] S. Li, C. Xu, and M. Xie. A robust o (n) solution to the perspective-n-point problem. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1444–1450, 2012.

[33] H. C. Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828):133, 1981.

[34] J. J. Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[35] L.-P. Morency and T. Darrell. Stereo tracking using icp and normal flow constraint. In *null*, page 40367. IEEE, 2002.

[36] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[37] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. Dtam: Dense tracking and mapping in real-time. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2320–2327. IEEE, 2011.

[38] D. Nister. An efficient solution to the five-point relative pose problem. In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 2, pages II–195. IEEE, 2003.

[39] NVIDIA. NVIDIA Jetson TX2 Module, 2018. URL https://developer.nvidia.com/embedded/buy/jetson-tx2.

[40] Nvidia. NVIDIA JETSON, 2018. URL https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/.

[41] D. Oberkampf, D. F. DeMenthon, and L. S. Davis. Iterative pose estimation using coplanar feature points. *Computer Vision and Image Understanding*, 63(3):495–511, 1996.

[42] OpenCV dev team. Camera calibration and 3D reconstruction, 2014. URL https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[43] OpenCV team. Feature matching. URL https://docs.opencv.org/3.4.3/dc/dc3/tutorial_py_matcher.html.

[44] OpenCV team. OpenCV library, 2018. URL https://opencv.org/.

[45] F. Pomerleau, F. Colas, R. Siegwart, and S. Magnenat. Comparing icp variants on real-world data sets. *Autonomous Robots*, 34(3):133–148, 2013.

[46] QGroundControl dev team. Download and install. URL https://docs.qgroundcontrol.com/en/getting_started/download_and_install.html.

[47] M. H. M. H. S. D. Rajvi Jingar, Reagan Lopez and I. Carpis. realsense_camera, 2018. URL http://wiki.ros.org/realsense_camera.

[48] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE international conference on*, pages 2564–2571. IEEE, 2011.

[49] S. Rusinkiewicz and M. Levoy. Efficient variants of the icp algorithm. In *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, pages 145–152. IEEE, 2001.

[50] A. Segal, D. Haehnel, and S. Thrun. Generalized-icp. In *Robotics: science and systems*, volume 2, page 435, 2009.

[51] D. H. Sergey Dorodnicov. realsense2_camera, 2018. URL http://wiki.ros.org/realsense_camera.

[52] J. Shotton, B. Glocker, C. Zach, S. Izadi, A. Criminisi, and A. Fitzgibbon. Scene coordinate regression forests for camera relocalization in rgb-d images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2930–2937, 2013.

[53] M. Smith, I. Baldwin, W. Churchill, R. Paul, and P. Newman. The new college vision and laser data set. *The International Journal of Robotics Research*, 28(5):595–599, 2009.

[54] Spektrum. Spekturm DX8 instruction manual. URL https://www.spektrumrc.com/ProdInfo/Files/SPM8800-Manual_EN.pdf.

[55] J. Sturm. RGB-D SLAM dataset and benchmark, 2017. URL https://vision.in.tum.de/data/datasets/rgbd-dataset.

[56] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 573–580. IEEE, 2012.

[57] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[58] WilliamWoodall. Creating a workspace for catkin, 2017. URL http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

[59] M. Zollhöfer, M. Nießner, S. Izadi, C. Rehmann, C. Zach, M. Fisher, C. Wu, A. Fitzgibbon, C. Loop, C. Theobalt, et al. Real-time non-rigid reconstruction using an rgb-d camera. *ACM Transactions on Graphics (TOG)*, 33(4):156, 2014.

10 Appendices

10.1 Setting up Jetson TX2

The following content in this section is copied from document sent to me by Jean-Luc on 13/03/2018.

This document briefly explains how to setup the NVIDIA Tegra TX1 or TX2 on either the Development Board or the Orbitty Carrier Board. While the processes for both are similar, we add small changes for doing the Orbitty Carrier board in red.

Setup NVIDIA Jetson L4T software

This describes the process to setup the NVIDIA Jetson L4T software. This process assumes that your computer has more than 1 network card. On laptops, the native wifi card and ethernet port is sufficient. This process also assumes you have Ubuntu (tested on 14.04 and 16.04) as an OS on the computer.

1. Download the NVIDIA Jetson L4T software from https://developer.nvidia.com/embedded/jetpack. You are required to have an NVIDIA account. I recommend you use your student/staff email to register.

2. Install the Jetson L4T software on the PC. **DO NOT FINALISE THE PROCESS AND FLASH THE TX1 or TX2, if you plan on using the Orbitty Carrier board**. Only select components to install relating to your project. Please refer to section 2 on dependancies.

3. Download the Orbitty Carrier components from http://connecttech.com/product/orbitty-carrier-for-nvidia-jetson-tx2-tx1/, in the downloads section. Select the version relating to the current Jetson L4T version and Tegra type.

4. Follow the step provided in the orbitty instruction manual to flash the TX1/TX2 with the orbitty drivers.

The next steps are used to install the special components on the TX1/TX2, as they do not install during the flashing process. These steps require 2 network cards (in laptops, the normal wifi and ethernet port are suitable).

1. Connect the TX1/TX2 to the host computer using an ethernet cable.
2. Set the ethernet port to a shared connection on the host computer, this will allow the TX1/TX2 to see the computer.
3. Find out the IP address of the TX1/TX2.
4. Go back into theJetson L4T software on the host PC, and go through the process. **DO NOT OPT TO REFLASH THE TX1/TX2**.
5. Choose the option to install the components on the TX1/TX2 that are required (eg. cuda, cudacnn, etc).
6. Using the IP address, install the components.

After this process, the TX1/TX2 will be ready to use as planned.

NVIDIA Jetson dependencies

When install the Jetson software, these dependencies may help you decide what applications are needed. I recommend not install any components on your host pc (laptop/pc), except what is required to flash the TX1/TX2.

- TensorFlow/CudaCNN: Only install if using deep learning - neural networking.
- OpenCV4Tegra: DO NOT INSTALL if you are planing to use ROS on the TX1/TX2
- Cuda: Always install this on the TX1/TX2, otherwise your GPU component will be useless

10.2 Installing ROS and setting up catkin

See [7], current terminal commands as of 8/8/2018 are as follows

```
    $ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
↪ $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
    $ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
↪ --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
    $ sudo apt-get update
    $ sudo apt-get install ros-kinetic-desktop
    $ sudo rosdep init
    $ rosdep update
    $ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
    $ source ~/.bashrc
    $ sudo apt-get install python-rosinstall
↪ python-rosinstall-generator python-wstool build-essential
```

Catkin will be installed with ROS, but the workspace still needs to be set up. See [58], current terminal commands as of 8/8/2018 are as follows

```
    $ source /opt/ros/kinetic/setup.bash
    $ mkdir -p ~/catkin_ws/src
    $ cd ~/catkin_ws/
    $ catkin_make
    $ source devel/setup.bash
```

10.3 Installing Mission Planner and QGroundControl

It is recommended to install Mission Planner in Windows, while QGroundControl should be installed on the main ground station which is running Ubuntu.

A .msi installer is available for Mission Planner, so setup is very straightforward. Instructions and a download link are available through ArduPilot [2]. QGroundCOntrol for Linux is available as either an AppImage or Compressed Archive. Instructions and download links are available through the QGroundControl docs [46].

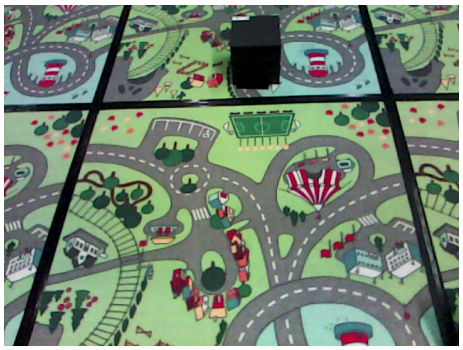
10.4 Initial Experiments

The initial experiments used multiple cardboard boxes for the scene, as shown in Figure 12. They also used an old RealSense camera, the R200, which was later found to have very noisy depth readings. Data capture for these experiments was done as pointclouds saved as .ply files rather than RGB and depth images through ROS.

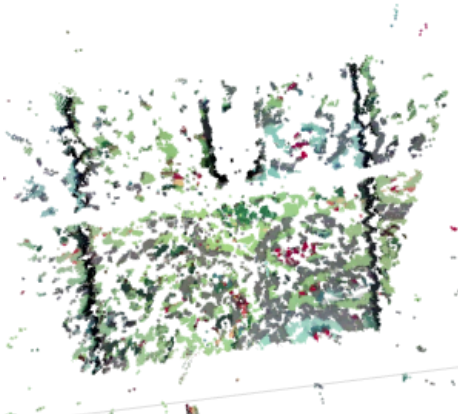
The pointclouds obtained from these experiments were missing a lot of data on all of the boxes except for the checkered box. This was tested by capturing pointclouds of each box individually with RealSense only moving very slowly (handheld). The results show that the data is still very patchy (see Figure 13). Thus the checkered box was chosen to be used in future experiments, and box 1 was also used after being covered in photographs.

To investigate other possible issues with the pointclouds, the depth images from the R200 are viewed (see Figure 14). There were two R200 cameras available, both of which were tested. The were also two platforms tested: the TX1 and a laptop running Ubuntu in case there were issues with the TX1 not having enough computational power. All of these depth images are too noisy to provide accurate results when used in registration, thus it was determined that a different RealSense camera should be used instead. The D415 was selected for the reasons described in Section 5.3.

Figure 12: Set-up of scene for initial flight experiments

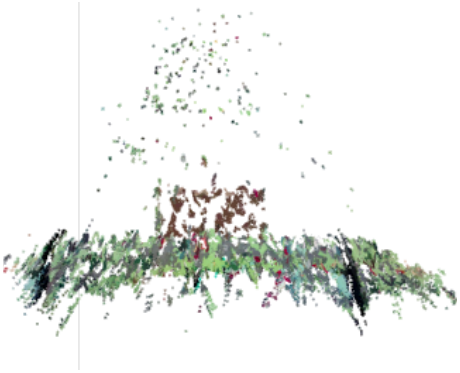


(a) Box 1, RGB image



(b) Box 1, Point cloud

(c) Box 2, RGB image

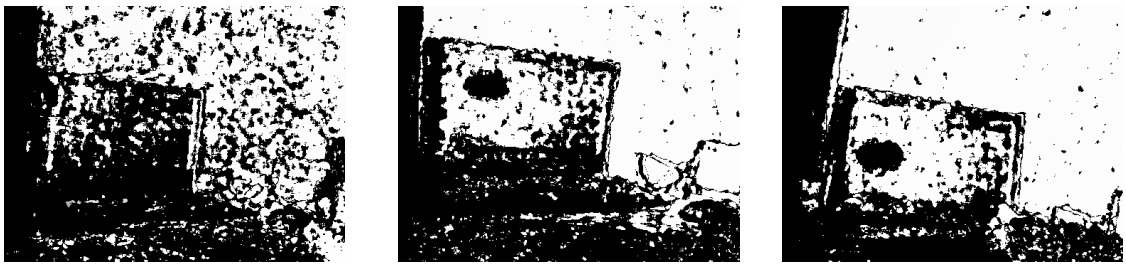


(d) Box 2, Point cloud

(e) Box 3, RGB image

(f) Box 3, Point cloud

Figure 13: RGB Image and Point cloud images for boxes, handheld RealSense R200 camera.



(a) Depth image from R200 RealSense that was mounted to quadcopter, with code running on TX1

(b) Depth image from other R200 RealSense, with code running on TX1

(c) Depth image from other R200 RealSense, with code running on Ubuntu

Figure 14: Comparison of depth images for the two RealSense R200 cameras and for running on different platforms.

10.5 Code: registration

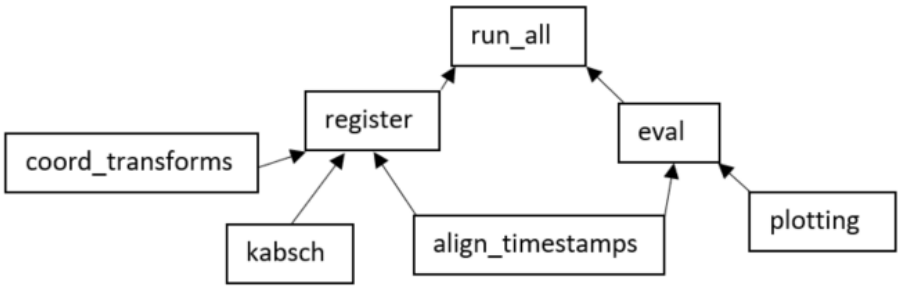


Figure 15: Overview of registration code modules. Arrows show dependencies.

Figure 15 shows the overview of the registration modules and how they relate to each other. run_all is the main script that calls register and then uses functions from eval to display the trajectories (eval calls plotting to do the trajectory plots) and evaluate them. Both eval and register call functions in align_timestamps to align the RGB, depth and ground truth data. align_timestamps also has functions to convert from Euler angles to rotation matrices, as the ground truth orientation is given in Euler angles. register also calls im_to_cam in coord_transforms to transform the 2D feature points and their associated depths into 3D points in the camera frame. kabsch implements the Kabsch data registration algorithm, with RANSAC, and is used by register. Note that the OpenCV implementations of the Essential Matrix method and PnP are used (by register). The code for each module is provided below.

run_all:

```
import numpy as np

import register
from eval import get_trjs, get_rtrjs, get_rotm, save_plot, compare, \
  process_comp_method, process_comp_all, process_comp_best

rgb_scales = [7,4.5,4,3,2,1,1,1]
stepsizes = [20,30,40,50,60,70,80,90]
methods = ['rgb','d','de','dp','pnp','pnpe']

data_path = '../../data/new-lawnmower/'
save_root = './basic-reg-saves-new-lawnmower/'

for i in range(len(stepsizes)):
  step = stepsizes[i]
  rgb_scale = rgb_scales[i]
  print(step)
  for method in methods:
    print(method)
    register.register(method, step, data_path, save_root, rgb_scale)


aeR_rgb = []
aet_rgb = []
aeR_d = []
aet_d = []
aeR_de = []
aet_de = []
aeR_dp = []
aet_dp = []
aeR_pnp = []
aet_pnp = []
aeR_pnpe = []
aet_pnpe = []

reR_rgb = []
ret_rgb = []
reR_d = []
ret_d = []
reR_de = []
ret_de = []
reR_dp = []
ret_dp = []
reR_pnp = []
ret_pnp = []
reR_pnpe = []
ret_pnpe = []

tsa = []

for step in stepsizes:
  save_path = save_root + str(step) + '/'
```

```
trj_gt, qtrj_gt, trj_rgb, qtrj_rgb, trj_d, qtrj_d, trj_pnp, qtrj_pnp, \
  trj_de, qtrj_de, j_dp, qtrj_dp, trj_pnpe, qtrj_pnpe, rgb_ts \
  = get_trjs(data_path, save_path, step)
rot_gt, rot_rgb, rot_d, rot_pnp, rot_de, rot_dp, rot_pnpe \
  = get_rotm(qtrj_gt, qtrj_rgb, qtrj_d, qtrj_pnp, qtrj_de, \
  qtrj_dp, qtrj_pnpe)


lim = 1.5
s = 0.1
save_plot(rot_gt,trj_gt,lim,s,'Ground truth trajectory','atrj_gt',save_path)
save_plot(rot_rgb,trj_rgb,lim,s,'Trajectory estimated using Essential Matrix',\
  'atrj_rgb',save_path)
save_plot(rot_d,trj_d,lim,s,'Trajectory estimated using Kabsch',\
  'atrj_d',save_path)
save_plot(rot_de,trj_de,lim,s,'Trajectory estimated using Kabsch (EM inliers)',\
  'atrj_de',save_path)
save_plot(rot_dp,trj_dp,lim,s,'Trajectory estimated using Kabsch (PnP inliers)',\
  'atrj_dp',save_path)
save_plot(rot_pnp,trj_pnp,lim,s,'Trajectory estimated using iterative PnP',\
  'atrj_pnp',save_path)
save_plot(rot_pnpe,trj_pnpe,lim,s,'Trajectory estimated using EPnP',\
  'atrj_pnpe',save_path)

et_rgb, eR_rgb = compare(rot_gt,trj_gt,rot_rgb,trj_rgb)
et_d, eR_d = compare(rot_gt,trj_gt,rot_d,trj_d)
et_de, eR_de = compare(rot_gt,trj_gt,rot_de,trj_de)
et_dp, eR_dp = compare(rot_gt,trj_gt,rot_dp,trj_dp)
et_pnp, eR_pnp = compare(rot_gt,trj_gt,rot_pnp,trj_pnp)
et_pnpe, eR_pnpe = compare(rot_gt,trj_gt,rot_pnpe,trj_pnpe)

aeR_rgb += [eR_rgb]
aet_rgb += [et_rgb]
aeR_d += [eR_d]
aet_d += [et_d]
aeR_de += [eR_de]
aet_de += [et_de]
aeR_dp += [eR_dp]
aet_dp += [et_dp]
aeR_pnp += [eR_pnp]
aet_pnp += [et_pnp]
aeR_pnpe += [eR_pnpe]
aet_pnpe += [et_pnpe]

tsa += [rgb_ts]

ets = np.vstack([et_rgb, et_d, et_de, et_dp, et_pnp, et_pnpe])
eRs = np.vstack([eR_rgb, eR_d, eR_de, eR_dp, eR_pnp, eR_pnpe])

process_comp_all(rgb_ts, ets, eRs, save_path, 'a')
```

```
  trj_gt, qtrj_gt, trj_rgb, qtrj_rgb, trj_d, qtrj_d, trj_pnp, qtrj_pnp, trj_de, \
    qtrj_de, trj_dp, qtrj_dp, trj_pnpe, qtrj_pnpe, rgb_ts \
    = get_rtrjs(data_path, save_path, step)
  rot_gt, rot_rgb, rot_d, rot_pnp, rot_de, rot_dp, rot_pnpe \
    = get_rotm(qtrj_gt, qtrj_rgb, qtrj_d, qtrj_pnp, qtrj_de, \
      qtrj_dp, qtrj_pnpe)

  lim = 1.5
  s = 0.1
  save_plot(rot_gt,trj_gt,lim,s,'Ground truth trajectory','rtrj_gt',save_path)
  save_plot(rot_rgb,trj_rgb,lim,s,'Trajectory estimated using Essential Matrix',\
    'rtrj_rgb',save_path)
  save_plot(rot_d,trj_d,lim,s,'Trajectory estimated using Kabsch','rtrj_d',\
    save_path)
  save_plot(rot_de,trj_de,lim,s,'Trajectory estimated using Kabsch (EM inliers)',\
    'rtrj_de',save_path)
  save_plot(rot_dp,trj_dp,lim,s,'Trajectory estimated using Kabsch (PnP inliers)',\
    'rtrj_dp',save_path)
  save_plot(rot_pnp,trj_pnp,lim,s,'Trajectory estimated using iterative PnP',\
    'rtrj_pnp',save_path)
  save_plot(rot_pnpe,trj_pnpe,lim,s,'Trajectory estimated using EPnP','rtrj_pnpe',\
    save_path)

  et_rgb, eR_rgb = compare(rot_gt,trj_gt,rot_rgb,trj_rgb)
  et_d, eR_d = compare(rot_gt,trj_gt,rot_d,trj_d)
  et_de, eR_de = compare(rot_gt,trj_gt,rot_de,trj_de)
  et_dp, eR_dp = compare(rot_gt,trj_gt,rot_dp,trj_dp)
  et_pnp, eR_pnp = compare(rot_gt,trj_gt,rot_pnp,trj_pnp)
  et_pnpe, eR_pnpe = compare(rot_gt,trj_gt,rot_pnpe,trj_pnpe)

  reR_rgb += [eR_rgb]
  ret_rgb += [et_rgb]
  reR_d += [eR_d]
  ret_d += [et_d]
  reR_de += [eR_de]
  ret_de += [et_de]
  reR_dp += [eR_dp]
  ret_dp += [et_dp]
  reR_pnp += [eR_pnp]
  ret_pnp += [et_pnp]
  reR_pnpe += [eR_pnpe]
  ret_pnpe += [et_pnpe]

  ets = np.vstack([et_rgb, et_d, et_de, et_dp, et_pnp, et_pnpe])
  eRs = np.vstack([eR_rgb, eR_d, eR_de, eR_dp, eR_pnp, eR_pnpe])

  process_comp_all(rgb_ts, ets, eRs, save_path, 'r')


save_path = save_root + 'methods/'
process_comp_method(tsa, aet_rgb, aeR_rgb, save_path, "Essential Matrix", \
  'rgb' ,pre='a')
process_comp_method(tsa, aet_d, aeR_d, save_path, "Kabsch", 'd' ,pre='a')
```

```
process_comp_method(tsa, aet_de, aeR_de, save_path, \
  "Kabsch (Essential Matrix inliers)", 'de' ,pre='a')
process_comp_method(tsa, aet_dp, aeR_dp, save_path, "Kabsch (PnP inliers)", \
  'dp' ,pre='a')
process_comp_method(tsa, aet_pnp, aeR_pnp, save_path, "PnP (iterative)", \
  'pnp' ,pre='a')
process_comp_method(tsa, aet_pnpe, aeR_pnpe, save_path, "EPnP", 'pnpe', pre='a')

print('\n')

process_comp_method(tsa, ret_rgb, aeR_rgb, save_path, "Essential Matrix", 'rgb' ,pre='r')
process_comp_method(tsa, ret_d, aeR_d, save_path, "Kabsch", 'd' ,pre='r')
process_comp_method(tsa, ret_de, aeR_de, save_path, \
  "Kabsch (Essential Matrix inliers)", 'de' ,pre='r')
process_comp_method(tsa, ret_dp, aeR_dp, save_path, "Kabsch (PnP inliers)", \
  'dp' ,pre='r')
process_comp_method(tsa, ret_pnp, aeR_pnp, save_path, "PnP (iterative)", 'pnp' ,pre='r')
process_comp_method(tsa, ret_pnpe, aeR_pnpe, save_path, "EPnP", 'pnpe' ,pre='r')

eta = [aet_rgb, aet_d, aet_de, aet_dp, aet_pnp, aet_pnpe]
eRa = [aeR_rgb, aeR_d, aeR_de, aeR_dp, aeR_pnp, aeR_pnpe]

etr = [ret_rgb, ret_d, ret_de, ret_dp, ret_pnp, ret_pnpe]
eRr = [reR_rgb, reR_d, reR_de, reR_dp, reR_pnp, reR_pnpe]

idx = [1,4,0,2,1,1]
names = ['Essential Matrix (skip 30)', 'Kabsch (skip 60)', \
  'Kabsch (EM inlers) (skip 20)', 'Kabsch (PnP inliers) (skip 40)', \
  'PnP (iterative) (skip 30)', 'EPnP (skip 30)']

process_comp_best(tsa, idx, eta, eRa, save_path, names, 'a')
process_comp_best(tsa, idx, etr, eRr, save_path, names, 'r')

suff = ['rgb','d','de','dp','pnp','pnpe']
idx = [1,4,0,2,1,1]
tf = 2100
n = []

for i in range(len(idx)):
  n += [np.floor(tf/((idx[i]+2)*10))]

print(n)
for i in range(len(idx)):
  t = np.load(save_root + str(stepsizes[idx[i]]) + '/t' + suff[i] + '.npy')
  print(t)
  print(np.sum(t))
  print(np.sum(t)/n[i])
```

register:

```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
```

```
from pyquaternion import Quaternion
from mpl_toolkits.mplot3d import Axes3D
import time

import coord_transforms
import align_timestamps
import kabsch

import scipy.io
import math

def register(method, step_size, data_path, save_root, rgb_scale):
  save_path = save_root + '/' + str(step_size) + '/'
  rgb_path = data_path + 'rgb/'
  depth_path = data_path + 'depth/'

  rgb_ims, depth_ims = align_timestamps.get_names(step_size, data_path)

  fo  = open(data_path + 'groundtruth.txt', 'r')
  fo.readline()
  trj_gt = []
  qtrj_gt = []
  t_gt = []
  for line in fo:
    ele = line[:-1].split(' ')
    if len(ele)==14:
      rotm = align_timestamps.eulerAnglesToRotationMatrix\
        ([float(ele[1]),float(ele[2]),float(ele[3])])
      q = Quaternion(matrix=rotm)
      t_gt += [float(ele[0])]
      trj_gt += [[ float(ele[7]), float(ele[8]), float(ele[9])]]
      qtrj_gt += [q]
  t_gt = np.vstack(t_gt)
  trj_gt = np.vstack(trj_gt)#.astype(np.float)
  qtrj_gt = np.vstack(qtrj_gt)

  rgb_ts = [x[:-4] for x in rgb_ims]
  rgb_ts = np.vstack(rgb_ts).astype(np.float)

  trj_gt = align_timestamps.gt_with_rgb(t_gt, rgb_ts, trj_gt)
  qtrj_gt = align_timestamps.gt_with_rgb(t_gt, rgb_ts, qtrj_gt)

  sp = trj_gt[0,:]
  so = qtrj_gt[0][0]

  Rgt2q = align_timestamps.Rx(math.pi)
  sp = np.dot(Rgt2q,sp.transpose()).transpose()

  R1 = align_timestamps.Rx(math.pi/2+math.pi/4)
  R2 = align_timestamps.Rz(-math.pi/2)
  Rc2q = np.dot(R2,R1)
```

```
rot = so.rotation_matrix

rot0 = rot.transpose()

so = Quaternion(matrix=rot)

pos = sp
trj = [pos]
qtrj = [so]
rtrj = [pos]
rqtrj = [so]


mat = np.eye(3)

fx = 616.9660034179688
fy = 616.8399047851562
cx = 328.9248962402344
cy = 230.74755859375
focal = np.sqrt(fx**2+fy**2)
pp = (cx,cy)

p = 0.9999

cmat = np.array([[fx,0,cx],[0,fy,cy],[0,0,1]])

tm_m = 0
tm_fd = 0
tm_ff = 0


for i in range(len(rgb_ims)-1):
  img1 = cv2.imread(rgb_path + rgb_ims[i],0)   # queryImage
  img2 = cv2.imread(rgb_path + rgb_ims[i+1],0) # trainImage

  mask = np.array(img1.shape, dtype=np.uint8)


  tm = time.time()
  # Initiate SIFT detector
  sift = cv2.xfeatures2d.SIFT_create()

  # find the keypoints and descriptors with SIFT - image frame
  kp1, des1 = sift.detectAndCompute(img1,None)
  kp2, des2 = sift.detectAndCompute(img2,None)

  des1 = des1.astype(np.uint8)
  des2 = des2.astype(np.uint8)


  # create BFMatcher object
  bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

```
# Match descriptors.
matches = bf.match(des1,des2) #train, query ?

# Sort them in the order of their distance.
matches = sorted(matches, key = lambda x:x.distance)

#Get points
pt1 = []
pt2 = []

for j in range(len(matches)-1):
  pt1 += [kp1[matches[j].queryIdx].pt]
  pt2 += [kp2[matches[j].trainIdx].pt]

pt1 = np.vstack(pt1)
pt2 = np.vstack(pt2)

tm_ff += time.time() - tm

imgm = cv2.drawMatches(img1, kp1, img2, kp2, matches, None)
plt.imshow(imgm)
plt.savefig('matches/{}_before.png'.format(rgb_ims[i][:-4]))
plt.close()

#Set up 3d points
if(method=='d'):
  tm = time.time()
  d_img1 = cv2.imread(depth_path + depth_ims[i],0)   # queryImage
  d_img2 = cv2.imread(depth_path + depth_ims[i+1],0) # trainImage

  pt1 = pt1.astype(np.uint32)
  pt2 = pt2.astype(np.uint32)

  pt2f = []

  P = []
  Q = []

  rgb_img1 = cv2.imread(rgb_path + rgb_ims[i])   # queryImage
  rgb_img2 = cv2.imread(rgb_path + rgb_ims[i+1]) # trainImage
  Cp = []
  Cq = []

  matches3 = []

  for j in range(pt1.shape[0]):
    #convert to camera frame 1 first
    d1 = d_img1[pt1[j,1], pt1[j,0]]
    d2 = d_img2[pt2[j,1], pt2[j,0]]

    if d1 > 0 and d2 >0:
      P += [coord_transforms.im_to_cam(pt1[j,0], pt1[j,1], d1)]
      Q += [coord_transforms.im_to_cam(pt2[j,0], pt2[j,1], d2)]
```

```
            Cp += [rgb_img1[pt1[j,1], pt1[j,0],:]]
            Cq += [rgb_img2[pt1[j,1], pt1[j,0],:]]

            matches3 += [matches[j]]

        P = np.vstack(P)
        Q = np.vstack(Q)
        Cp = np.vstack(Cp)/255.0
        Cq = np.vstack(Cq)/255.0

        tm_fd += time.time() - tm

    elif(method=='dp'):
        tm = time.time()
        d_img1 = cv2.imread(depth_path + depth_ims[i],0)   # queryImage
        d_img2 = cv2.imread(depth_path + depth_ims[i+1],0) # trainImage

        pt1 = pt1.astype(np.uint32)
        pt2 = pt2.astype(np.uint32)

        pt2f = []

        P = []
        Q = []

        rgb_img1 = cv2.imread(rgb_path + rgb_ims[i])   # queryImage
        rgb_img2 = cv2.imread(rgb_path + rgb_ims[i+1]) # trainImage
        Cp = []
        Cq = []

        matches3 = []

        for j in range(pt1.shape[0]):
            #convert to camera frame 1 first
            d1 = d_img1[pt1[j,1], pt1[j,0]]
            d2 = d_img2[pt2[j,1], pt2[j,0]]

            if d1 > 0 and d2 >0:
                P += [coord_transforms.im_to_cam(pt1[j,0], pt1[j,1], d1)]
                Q += [coord_transforms.im_to_cam(pt2[j,0], pt2[j,1], d2)]

                pt2f += [[float(pt2[j,0]), float(pt2[j,1])]]

                Cp += [rgb_img1[pt1[j,1], pt1[j,0],:]]
                Cq += [rgb_img2[pt1[j,1], pt1[j,0],:]]

                matches3 += [matches[j]]

        P = np.vstack(P)
        Q = np.vstack(Q)
        Cp = np.vstack(Cp)/255.0
        Cq = np.vstack(Cq)/255.0
```

```
      pt2f = np.vstack(pt2f)

      tm_fd += time.time() - tm

    elif(method=='pnp' or method=='pnpe'):
      tm = time.time()
      d_img1 = cv2.imread(depth_path + depth_ims[i],0)   # queryImage
      d_img2 = cv2.imread(depth_path + depth_ims[i+1],0) # trainImage

      pt1 = pt1.astype(np.uint32)
      pt2 = pt2.astype(np.uint32)

      pt2f = []

      P = []

      matches3 = []

      for j in range(pt1.shape[0]):
        #convert to camera frame 1 first
        d1 = d_img1[pt1[j,1], pt1[j,0]]
        d2 = d_img2[pt2[j,1], pt2[j,0]]

        if d1 > 0 and d2 >0:
          P += [coord_transforms.im_to_cam(pt1[j,0], pt1[j,1], d1)]
          pt2f += [[float(pt2[j,0]), float(pt2[j,1])]]

          matches3 += [matches[j]]

      pt2f = np.vstack(pt2f)
      P = np.vstack(P)

      tm_fd += time.time() - tm

    #Do registration
    if(method=='rgb'):
      tm = time.time()
      E = cv2.findEssentialMat(pt2, pt1, focal, pp, cv2.RANSAC, p, 1.0, None)
      R = np.zeros((3,3))
      t = np.zeros((3,1))
      cv2.recoverPose(E[0], pt2, pt1, R, t, focal, pp, None) #camera frame 1??

      tm_m += time.time() - tm

      matches_good = []
      for m in range(len(E[1])):
        if E[1][m] == 1:
          matches_good += [matches[m]]

      imgm = cv2.drawMatches(img1, kp1, img2, kp2, matches_good, None)
      plt.imshow(imgm)
      plt.savefig('matches/{}_essential.png'.format(rgb_ims[i][:-4]))
```

```
    plt.close()

    R = np.transpose(R)
    t = -t/rgb_scale

  elif(method=='d'):
    tm = time.time()
    R, t, inliers = kabsch.kabsch(P, Q, p=p, v=0.5, dist_thresh = 0.1)
    tm_m += time.time() - tm

    matches_good = []
    if not (inliers is None):
      for m in inliers:
        matches_good += [matches3[m]]

      imgm = cv2.drawMatches(img1, kp1, img2, kp2, matches_good, None)
      plt.imshow(imgm)
      plt.savefig('matches/{}_kabsch.png'.format(rgb_ims[i][:-4]))
      plt.close()

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(P[:,0], P[:,1], P[:,2], c='b')
    ax.scatter(Q[:,0], Q[:,1], Q[:,2], c='g')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.legend(['P', 'Q'])
    # plt.show()
    plt.savefig('PCs/{}_PQ.png'.format(rgb_ims[i][:-4]))
    plt.close()

    Pa = np.dot(P, R.transpose()) + np.repeat(t.transpose(), P.shape[0], axis=0)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(Pa[:,0], Pa[:,1], Pa[:,2], c='b')
    ax.scatter(Q[:,0], Q[:,1], Q[:,2], c='g')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.legend(['P aligned', 'Q'])
    # plt.show()
    plt.savefig('PCs/{}_PQ-aligned.png'.format(rgb_ims[i][:-4]))
    plt.close()

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(P[inliers,0], P[inliers,1], P[inliers,2], c='b')
    ax.scatter(Q[inliers,0], Q[inliers,1], Q[inliers,2], c='g')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
```

```
    ax.legend(['P', 'Q'])
    # plt.show()
    plt.savefig('PCs/{}_PQ-inliers.png'.format(rgb_ims[i][:-4]))
    plt.close()

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(Pa[inliers,0], Pa[inliers,1], Pa[inliers,2], c='b')
    ax.scatter(Q[inliers,0], Q[inliers,1], Q[inliers,2], c='g')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.legend(['P aligned', 'Q'])
    # plt.show()
    plt.savefig('PCs/{}_PQ-aligned-inliers.png'.format(rgb_ims[i][:-4]))
    plt.close()

  elif(method=='de'):
    E = cv2.findEssentialMat(pt2, pt1, focal, pp, cv2.RANSAC, p, 1.0, None)

    if len(E[1])<5:
      R = np.eye(3)
      t = np.array([0,0,0])
    else:

      tm = time.time()
      d_img1 = cv2.imread(depth_path + depth_ims[i],0)   # queryImage
      d_img2 = cv2.imread(depth_path + depth_ims[i+1],0) # trainImage

      pt1 = pt1.astype(np.uint32)
      pt2 = pt2.astype(np.uint32)

      Pe = []
      Qe = []

      for j in range(pt1.shape[0]):
        #convert to camera frame 1 first
        d1 = d_img1[pt1[j,1], pt1[j,0]]
        d2 = d_img2[pt2[j,1], pt2[j,0]]

        if d1 > 0 and d2 >0:
          if E[1][j]==1:
            Pe += [coord_transforms.im_to_cam(pt1[j,0], pt1[j,1], d1)]
            Qe += [coord_transforms.im_to_cam(pt2[j,0], pt2[j,1], d2)]

      Pe = np.vstack(Pe)
      Qe = np.vstack(Qe)

      tm_fd += time.time() - tm

      tm = time.time()
      R, t= kabsch.find_transform(Pe, Qe)
      tm_m += time.time() - tm
```

```
elif(method=='dp'):
  v = 0.8
  m = 5
  N = np.int(np.ceil(np.log(1-p)/np.log(1-(1-v)**m)))
  dist_thresh = 1 #0.05

  P = P.reshape((P.shape[0],1,3))
  pt2f = pt2f.reshape((pt2f.shape[0],1,2))

  _, R_vec, t, in_pnp = cv2.solvePnPRansac(P, pt2f, cmat, None, \
    iterationsCount=N, reprojectionError=dist_thresh, confidence=p, \
    flags=cv2.SOLVEPNP_ITERATIVE)
  if in_pnp is None:
    R = np.eye(3)
    t = np.array([0,0,0])
  else:
    Pi = P[in_pnp[:],:]
    Qi = Q[in_pnp[:],:]
    Pi = Pi.reshape((Pi.shape[0],3))
    Qi = Qi.reshape((Qi.shape[0],3))
    # print(i)
    tm = time.time()
    R, t = kabsch.find_transform(Pi, Qi)
    tm_m += time.time() - tm

elif(method=='pnp'):
  v = 0.8
  m = 5
  N = np.int(np.ceil(np.log(1-p)/np.log(1-(1-v)**m)))
  dist_thresh = 1 #0.05

  P = P.reshape((P.shape[0],1,3))
  pt2f = pt2f.reshape((pt2f.shape[0],1,2))

  tm = time.time()
  _, R_vec, t, in_pnp = cv2.solvePnPRansac(P, pt2f, cmat, None, \
    iterationsCount=N, reprojectionError=dist_thresh, confidence=p, \
    flags=cv2.SOLVEPNP_ITERATIVE)
  tm_m += time.time() - tm

  R = np.zeros((3,3))
  cv2.Rodrigues(R_vec,R)

  matches_good = []
  if in_pnp is not None:
    for m in in_pnp:
      matches_good += [matches3[m[0]]]

    imgm = cv2.drawMatches(img1, kp1, img2, kp2, matches_good, None)
    plt.imshow(imgm)
    plt.savefig('matches/{}_pnp.png'.format(rgb_ims[i][:-4]))
    plt.close()
```

```
elif(method=='pnpe'):
  v = 0.8
  m = 5
  N = np.int(np.ceil(np.log(1-p)/np.log(1-(1-v)**m)))
  dist_thresh = 1 #0.05

  P = P.reshape((P.shape[0],1,3))
  pt2f = pt2f.reshape((pt2f.shape[0],1,2))

  tm = time.time()
  _, R_vec, t, in_pnp = cv2.solvePnPRansac(P, pt2f, cmat, None, iterationsCount=N, \
    reprojectionError=dist_thresh, confidence=p, flags=cv2.SOLVEPNP_EPNP)
  tm_m += time.time() - tm

  R = np.zeros((3,3))
  cv2.Rodrigues(R_vec,R)

  matches_good = []
  if in_pnp is not None:
    for m in in_pnp:
      matches_good += [matches3[m[0]]]

    imgm = cv2.drawMatches(img1, kp1, img2, kp2, matches_good, None)
    plt.imshow(imgm)
    plt.savefig('matches/{}_pnpe.png'.format(rgb_ims[i][:-4]))
    plt.close()

#Add to trajectory

Rq = qtrj_gt[i][0].rotation_matrix.transpose()
pq = np.dot(Rgt2q,trj_gt[i,:].transpose()).transpose()

R = np.dot(np.dot(Rc2q,R),Rc2q.transpose())
t = np.dot(Rc2q,t)

q_c = np.dot(rot, R)
pos = pos + np.transpose(np.dot(rot.transpose(),t))
rot = q_c

if (len(pos)!=3):
  pos = pos[0]
trj += [pos]
qtrj += [Quaternion(matrix=q_c.transpose())]


t = np.dot(Rc2q.transpose(),t)

qr = np.dot(Rq, R)
rpos = pq + np.transpose(np.dot(Rq.transpose(),t))

if (len(rpos)!=3):
  rpos = rpos[0]
```

```
    rtrj += [rpos]
    rqtrj += [Quaternion(matrix=qr.transpose())]


  #Save
  trj = np.vstack(trj)
  qtrj = np.vstack(qtrj)
  rtrj = np.vstack(rtrj)
  rqtrj = np.vstack(rqtrj)

  np.save(save_path + 'trj_' + method, trj)
  np.save(save_path + 'qtrj_' + method, qtrj)
  np.save(save_path + 'rtrj_' + method, rtrj)
  np.save(save_path + 'rqtrj_' + method, rqtrj)

  np.save(save_path + 't' + method, np.array([tm_ff, tm_fd, tm_m]))
```

coord_transforms:

```
# def im_to_cam(u, v, d, fx = 520.9, fy = 521.0, cx = 325.1, cy = 249.7, factor = 1):
def im_to_cam(u, v, d, fx = 616.9660034179688, fy = 616.8399047851562, \
  cx = 328.9248962402344, cy = 230.74755859375, factor = 20):
    Z = d / factor
    X = (u - cx) * Z / fx
    Y = (v - cy) * Z / fy

    return [X, Y, Z]
```

kabsch:

```
import numpy as np
import cv2

def kabsch(P,Q,p,v, dist_thresh = 0.03):
  np.random.seed(0)
  m = 5
  N = np.int(np.ceil(np.log(1-p)/np.log(1-(1-v)**m)))

  frac_thresh = max(0.05, m/N)

  num_points = P.shape[0]

  best_dist = 1e10
  best_good_i = None

  for n in range(N):
    i = np.random.randint(0, num_points, size=m)

    Pn = P[i,:]
    Qn = Q[i,:]

    R_d, t_d = find_transform(Pn, Qn)

    Pa = np.dot(P, R_d.transpose()) + np.repeat(t_d.transpose(), num_points, axis=0)
```

```
    # Pa = np.zeros(P.shape)

    # for j in range(P.shape[0]):
    #   Pa[j,:] = np.dot(R_d, P[j,:].transpose()) + t_d.transpose()

    dist = np.sqrt(np.sum(np.square(Pa - Q), axis=1))

    # print(min(dist))

    good_i = np.where(dist<dist_thresh)[0]

    # print(dist)

    # print(len(good_i))

    if (len(good_i)/N) > frac_thresh:
      Pn = P[good_i,:]
      Qn = Q[good_i,:]

      R_d, t_d = find_transform(Pn, Qn)
      Pa = np.dot(Pn, R_d.transpose()) + np.repeat(t_d.transpose(), len(good_i), axis=0)

      dist = np.sum(np.sqrt(np.sum(np.square(Pa - Qn), axis=1)))
      if dist < best_dist:
        best_dist = dist
        best_R_d = R_d
        best_t_d = t_d
        best_good_i = good_i
      # print(len(good_i))


  if (best_good_i is None):
    print(dist_thresh)
    if(dist_thresh>20):
      return np.eye(3), np.array([[0,0,0]]).transpose(), None
    return kabsch(P,Q,p,v, dist_thresh =dist_thresh*2)


  return best_R_d, best_t_d, best_good_i


def find_transform(Pn, Qn):
  mat = np.eye(3)

  p0 = (np.sum(Pn, axis=0)/Pn.shape[0]).reshape(1,3)
  q0 = (np.sum(Qn, axis=0)/Qn.shape[0]).reshape(1,3)




  P1 = Pn - np.repeat(p0,Pn.shape[0],axis=0).astype(np.float32)
  Q1 = Qn - np.repeat(q0,Qn.shape[0],axis=0).astype(np.float32)
```

```
  C = np.dot(P1.transpose(),Q1) #Nx3, not 3xN

  S, V, Wt = cv2.SVDecomp(C)

  Vt = V.transpose()
  W = Wt.transpose()

  d = np.round(np.linalg.det(np.dot(W,Vt)))

  mat[2,2] = d

  R_d = np.dot(np.dot(W,mat),Vt)

  t_d = q0.reshape(3,1)-np.dot(R_d,p0.reshape(3,1))

  return R_d, t_d
```

eval:

```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
from pyquaternion import Quaternion
from mpl_toolkits.mplot3d import Axes3D

import align_timestamps
import plotting

import math


def get_rot(q, R0):
  # Rc2q = align_timestamps.Rc2q()
  # return np.dot(Rc2q, q.rotation_matrix)
  return q.rotation_matrix

def align_trj(trj, R0):
  # Rc2q = align_timestamps.Rc2q()

  # trj = np.dot(Rc2q,trj.transpose()).transpose()
  return trj

def align_gt(trj_gt, qtrj_gt):
  R1 = align_timestamps.Rx(math.pi)
  trj_gt = np.dot(R1,trj_gt.transpose()).transpose()
  return trj_gt, qtrj_gt


def get_trjs(data_path, save_path, step):
  rgb_path = data_path + 'rgb/'
  depth_path = data_path + 'depth/'
```

```
rgb_ims = os.listdir(rgb_path)
depth_ims = os.listdir(depth_path)

rgb_ims = align_timestamps.sample_rgb(rgb_ims, step)

rgb_ts = [x[:-4] for x in rgb_ims]
rgb_ts = np.vstack(rgb_ts).astype(np.float)


fo  = open(data_path + 'groundtruth.txt', 'r')
fo.readline()
trj_gt = []
qtrj_gt = []
t_gt = []
for line in fo:
  ele = line[:-1].split(' ')
  if len(ele)==14:
    rotm = align_timestamps.eulerAnglesToRotationMatrix([float(ele[1]),\
      float(ele[2]),float(ele[3])])
    q = Quaternion(matrix=rotm)
    t_gt += [float(ele[0])]
    trj_gt += [[ float(ele[7]), float(ele[8]), float(ele[9])]]
    qtrj_gt += [q]
t_gt = np.vstack(t_gt)
trj_gt = np.vstack(trj_gt)#.astype(np.float)
qtrj_gt = np.vstack(qtrj_gt)

suff = ''
trj_rgb = np.load(save_path + 'trj_rgb' + suff + '.npy')
qtrj_rgb = np.load(save_path + 'qtrj_rgb' + suff + '.npy')
trj_d = np.load(save_path + 'trj_d' + suff + '.npy')
qtrj_d = np.load(save_path + 'qtrj_d' + suff + '.npy')
trj_pnp = np.load(save_path + 'trj_pnp' + suff + '.npy')
qtrj_pnp = np.load(save_path + 'qtrj_pnp' + suff + '.npy')
trj_pnpe = np.load(save_path + 'trj_pnpe' + suff + '.npy')
qtrj_pnpe = np.load(save_path + 'qtrj_pnpe' + suff + '.npy')
trj_de = np.load(save_path + 'trj_de' + suff + '.npy')
qtrj_de = np.load(save_path + 'qtrj_de' + suff + '.npy')
trj_dp = np.load(save_path + 'trj_dp' + suff + '.npy')
qtrj_dp = np.load(save_path + 'qtrj_dp' + suff + '.npy')

trj_gt = align_timestamps.gt_with_rgb(t_gt, rgb_ts, trj_gt)
qtrj_gt = align_timestamps.gt_with_rgb(t_gt, rgb_ts, qtrj_gt)

trj_gt, qtrj_gt = align_gt(trj_gt,qtrj_gt)

trj_rgb = align_trj(trj_rgb, qtrj_gt[0,0].rotation_matrix)
trj_d = align_trj(trj_d, qtrj_gt[0,0].rotation_matrix)
trj_de = align_trj(trj_de, qtrj_gt[0,0].rotation_matrix)
trj_dp = align_trj(trj_dp, qtrj_gt[0,0].rotation_matrix)
trj_pnp = align_trj(trj_pnp, qtrj_gt[0,0].rotation_matrix)
trj_pnpe = align_trj(trj_pnpe, qtrj_gt[0,0].rotation_matrix)
```

```
  return trj_gt, qtrj_gt, trj_rgb, qtrj_rgb, trj_d, qtrj_d, trj_pnp, qtrj_pnp, \
    trj_de, qtrj_de, trj_dp, qtrj_dp, trj_pnpe, qtrj_pnpe, rgb_ts

def get_rtrjs(data_path, save_path, step):
  rgb_path = data_path + 'rgb/'
  depth_path = data_path + 'depth/'
  rgb_ims = os.listdir(rgb_path)
  depth_ims = os.listdir(depth_path)

  rgb_ims = align_timestamps.sample_rgb(rgb_ims,step)

  rgb_ts = [x[:-4] for x in rgb_ims]
  rgb_ts = np.vstack(rgb_ts).astype(np.float)


  fo  = open(data_path + 'groundtruth.txt', 'r')
  fo.readline()
  trj_gt = []
  qtrj_gt = []
  t_gt = []
  for line in fo:
    ele = line[:-1].split(' ')
    if len(ele)==14:
      rotm = align_timestamps.eulerAnglesToRotationMatrix([float(ele[1]),\
        float(ele[2]),float(ele[3])])
      q = Quaternion(matrix=rotm)
      t_gt += [float(ele[0])]
      trj_gt += [[ float(ele[7]), float(ele[8]), float(ele[9])]]
      qtrj_gt += [q]
  t_gt = np.vstack(t_gt)
  trj_gt = np.vstack(trj_gt)
  qtrj_gt = np.vstack(qtrj_gt)

  suff = ''
  trj_rgb = np.load(save_path + 'rtrj_rgb' + suff + '.npy')
  qtrj_rgb = np.load(save_path + 'rqtrj_rgb' + suff + '.npy')
  trj_d = np.load(save_path + 'rtrj_d' + suff + '.npy')
  qtrj_d = np.load(save_path + 'rqtrj_d' + suff + '.npy')
  trj_pnp = np.load(save_path + 'rtrj_pnp' + suff + '.npy')
  qtrj_pnp = np.load(save_path + 'rqtrj_pnp' + suff + '.npy')
  trj_pnpe = np.load(save_path + 'rtrj_pnpe' + suff + '.npy')
  qtrj_pnpe = np.load(save_path + 'rqtrj_pnpe' + suff + '.npy')
  trj_de = np.load(save_path + 'rtrj_de' + suff + '.npy')
  qtrj_de = np.load(save_path + 'rqtrj_de' + suff + '.npy')
  trj_dp = np.load(save_path + 'rtrj_dp' + suff + '.npy')
  qtrj_dp = np.load(save_path + 'rqtrj_dp' + suff + '.npy')

  trj_gt = align_timestamps.gt_with_rgb(t_gt, rgb_ts, trj_gt)
  qtrj_gt = align_timestamps.gt_with_rgb(t_gt, rgb_ts, qtrj_gt)

  trj_gt, qtrj_gt = align_gt(trj_gt,qtrj_gt)

  trj_rgb = align_trj(trj_rgb, qtrj_gt[0,0].rotation_matrix)
```

```
  trj_d = align_trj(trj_d, qtrj_gt[0,0].rotation_matrix)
  trj_de = align_trj(trj_de, qtrj_gt[0,0].rotation_matrix)
  trj_dp = align_trj(trj_dp, qtrj_gt[0,0].rotation_matrix)
  trj_pnp = align_trj(trj_pnp, qtrj_gt[0,0].rotation_matrix)
  trj_pnpe = align_trj(trj_pnpe, qtrj_gt[0,0].rotation_matrix)

  return trj_gt, qtrj_gt, trj_rgb, qtrj_rgb, trj_d, qtrj_d, trj_pnp, qtrj_pnp, \
    trj_de, qtrj_de, trj_dp, qtrj_dp, trj_pnpe, qtrj_pnpe, rgb_ts

def get_rotm(qtrj_gt, qtrj_rgb, qtrj_d, qtrj_pnp, qtrj_de, qtrj_dp, qtrj_pnpe):

  rot_gt = np.dstack([x.rotation_matrix for x in qtrj_gt[:,0]])
  rot_rgb = np.dstack([get_rot(x, qtrj_gt[0,0].rotation_matrix) for x in qtrj_rgb[:,0]])
  rot_d = np.dstack([get_rot(x, qtrj_gt[0,0].rotation_matrix) for x in qtrj_d[:,0]])
  rot_pnp = np.dstack([get_rot(x, qtrj_gt[0,0].rotation_matrix) for x in qtrj_pnp[:,0]])
  rot_pnpe = np.dstack([get_rot(x, qtrj_gt[0,0].rotation_matrix) for x in qtrj_pnpe[:,0]])
  rot_de = np.dstack([get_rot(x, qtrj_gt[0,0].rotation_matrix) for x in qtrj_de[:,0]])
  rot_dp = np.dstack([get_rot(x, qtrj_gt[0,0].rotation_matrix) for x in qtrj_dp[:,0]])

  return rot_gt, rot_rgb, rot_d, rot_pnp, rot_de, rot_dp, rot_pnpe


def save_plot(rot,pos,lim,s,title,name,save_path):
  f = plt.figure()
  ax = f.add_subplot(111, projection='3d')
  for i in range(pos.shape[0]):
    plotting.plot_axis(ax,rot[:,:,i],pos[i,:],s, i)
  ax.set_xlabel('X')
  ax.set_ylabel('Y')
  ax.set_zlabel('Z')
  ax.set_title(title)
  ax.set_xlim(-lim,lim)
  ax.set_ylim(-lim,lim)
  ax.set_zlim(-lim,lim)
  f.savefig(save_path + name + '.pdf', bbox_inches='tight')
  plt.close()

def compare(rot_gt,pos_gt,rot,pos):
  #get transform from gt to estimated:

  if len(rot.shape) < 3:
    t = -pos + pos_gt
    et = np.sqrt(np.sum(np.square(t)))
    R = np.dot(rot_gt[:,:],rot[:,:].transpose())
    q = Quaternion(matrix=R)
    ar = q.radians
    ar = ar%math.pi
    eR = ar
  else:
    t = -pos + pos_gt
    et = np.sqrt(np.sum(np.square(t),axis=1))
    eR = []
    for i in range(rot.shape[2]):
```

```
      R = np.dot(rot_gt[:,:,i],rot[:,:,i].transpose())
      q = Quaternion(matrix=R)
      ar = q.radians
      ar = ar%math.pi
      eR += [ar]

  return et, eR

def process_comp(ts, et,eR,pre,suff,name,save_path):
  f = plt.figure()
  plt.plot(ts, et)
  plt.title('Translation error for trajectory estimated using ' + name)
  plt.xlabel('Time')
  plt.ylabel('Eulidean distance')
  f.savefig(save_path + pre + 'et_' + suff + '.pdf', bbox_inches='tight')
  plt.close()

  f = plt.figure()
  plt.plot(ts, eR)
  plt.title('Rotation error for trajectory estimated using ' + name)
  plt.xlabel('Time')
  plt.ylabel('Magnitude of rotation')
  f.savefig(save_path + pre + 'eR_' + suff + '.pdf', bbox_inches='tight')
  plt.close()

  print(name, np.sum(et), np.sum(eR))

def process_comp_all(ts, ets,eRs,save_path,pre=''):
  f = plt.figure()
  for i in range(ets.shape[0]):
    plt.plot(ts, ets[i,:])
  plt.title('Translation error for all estimation methods')
  plt.xlabel('Time')
  plt.ylabel('Eulidean distance (meters)')
  plt.legend(['Essential Matrix', 'Kabsch', 'Kabsch (EM inliers)', \
    'Kabsch (PnP inliers)', 'iterative PnP', 'EPnP'])
  f.savefig(save_path + pre + 'et_all' + '.pdf', bbox_inches='tight')
  plt.close()

  f = plt.figure()
  for i in range(eRs.shape[0]):
    plt.plot(ts, eRs[i,:])
  plt.title('Rotation error for all estimation methods')
  plt.xlabel('Time')
  plt.ylabel('Magnitude of rotation (radians)')
  plt.legend(['Essential Matrix', 'Kabsch', 'Kabsch (EM inliers)', \
    'Kabsch (PnP inliers)', 'iterative PnP', 'EPnP'])
  f.savefig(save_path + pre + 'eR_all' + '.pdf', bbox_inches='tight')
  plt.close()


def process_comp_method(ts, ets,eRs,save_path,name,suff,pre=''):
  s = 'skip'
```

```
  f = plt.figure()
  for i in range(len(ets)):
    plt.plot(ts[i], ets[i])
  plt.title('Translation error for ' + name + ' method')
  plt.xlabel('Time')
  plt.ylabel('Eulidean distance (meters)')
  plt.legend([s+' 20',s+' 30',s+' 40',s+' 50',s+' 60',s+' 70',s+' 80',s+' 90'])
  # plt.savefig(save_path + pre + 'et_' + suff)
  f.savefig(save_path + pre + 'et_' + suff + '.pdf', bbox_inches='tight')
  plt.close()

  f = plt.figure()
  for i in range(len(eRs)):
    plt.plot(ts[i], eRs[i])
  plt.title('Rotation error for ' + name + ' method')
  plt.xlabel('Time')
  plt.ylabel('Magnitude of rotation (radians)')
  plt.legend([s+' 20',s+' 30',s+' 40',s+' 50',s+' 60',s+' 70',s+' 80',s+' 90'])
  # plt.savefig(save_path + pre + 'eR_' + suff)
  f.savefig(save_path + pre + 'eR_' + suff + '.pdf', bbox_inches='tight')
  plt.close()

  for i in range(len(ets)):
    print("& %d & %.4f & %.4f & %.4f & %.4f \\\\" % \
      ((i+2)*10, np.mean(eRs[i]), np.std(eRs[i]), np.mean(ets[i]), np.std(ets[i])))

def process_comp_best(ts, idx, ets,eRs,save_path,names,suff,pre=''):
  f = plt.figure()
  for i in range(len(idx)):
    j = idx[i]
    plt.plot(ts[j], ets[i][j])
  plt.title('Translation error for each method, best skip')
  plt.xlabel('Time')
  plt.ylabel('Eulidean distance (meters)')
  plt.legend(names)
  f.savefig(save_path + pre + 'et_best_' + suff + '.pdf', bbox_inches='tight')
  plt.close()

  f = plt.figure()
  for i in range(len(idx)):
    j = idx[i]
    plt.plot(ts[j], eRs[i][j])
  plt.title('Rotation error for each method, best skip')
  plt.xlabel('Time')
  plt.ylabel('Magnitude of rotation (radians)')
  plt.legend(names)
  f.savefig(save_path + pre + 'eR_best_' + suff + '.pdf', bbox_inches='tight')
  plt.close()
```

plotting:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
import math

def plot_axis(ax,R,t,s,lab):
  x = s*np.array([1,0,0])
  y = s*np.array([0,1,0])
  z = s*np.array([0,0,1])

  x = np.dot(R,x) + t
  y = np.dot(R,y) + t
  z = np.dot(R,z) + t

  ax.plot([t[0],x[0]],[t[1],x[1]],[t[2],x[2]], 'b')
  ax.plot([t[0],y[0]],[t[1],y[1]],[t[2],y[2]], 'g')
  ax.plot([t[0],z[0]],[t[1],z[1]],[t[2],z[2]], 'r')

  ax.text(t[0], t[1], t[2], lab)

  ax.legend(['x','y','z'])

# fig = plt.figure()
# ax = fig.add_subplot(111, projection='3d')
# save_path = './basic-reg-saves/'
# qtrj_rgb = np.load(save_path + 'qtrj_rgb.npy')
# rot = qtrj_rgb[0][0].rotation_matrix
# plot_axis(ax, rot, np.array([0,0.5,0]))
# ax.set_xlabel('X')
# ax.set_ylabel('Y')
# ax.set_zlabel('Z')
# plt.show()
```

align_timestamps:

```
import os
import numpy as np
import math


def Rxo(a):
  return np.array([[1, 0,           0           ],
                   [0,  math.cos(a), -math.sin(a) ],
                   [0,  math.sin(a), math.cos(a)  ]
                   ])

def Ryo(a):
  return np.array([[math.cos(a), 0, math.sin(a)  ],
                   [0,           1, 0           ],
                   [-math.sin(a), 0, math.cos(a)  ]
                   ])

def Rzo(a):
  return np.array([[math.cos(a), -math.sin(a),  0],
                   [math.sin(a),  math.cos(a),   0],
                   [0,           0,         1    ]
                   ])
```

```
def Rx(a):
  return np.array([[1,  0,            0          ],
                   [0,   math.cos(a),  math.sin(a)],
                   [0,   -math.sin(a), math.cos(a)]
                   ])

def Ry(a):
  return np.array([[math.cos(a), 0, -math.sin(a) ],
                   [0,            1, 0            ],
                   [math.sin(a),  0, math.cos(a)  ]
                   ])

def Rz(a):
  return np.array([[math.cos(a), math.sin(a),   0],
                   [-math.sin(a), math.cos(a),   0],
                   [0,            0,             1]
                   ])

def eulerAnglesToRotationMatrix(theta) :

    R_x = Rxo(theta[0])

    R_y = Ryo(theta[1])

    R_z = Rzo(theta[2])


    R = np.dot(R_x, np.dot( R_y, R_z ))


    return R.transpose()

def sample_rgb(rgb_ims, step=40):
  #circle 1: 700, 2800
  #circle 2: 745, 3023
  #rectangle: 752, 5244
  #lawnmower: 825, 6629
  start = 700
  end = 2800
  return rgb_ims[start:end:step]


def get_names(step=40,data_path='../../data/quad3/'):
  # data_path = '../../data/quad3/'
  save_path = './basic-reg-saves/'
  rgb_path = data_path + 'rgb/'
  depth_path = data_path + 'depth/'
  rgb_ims = os.listdir(rgb_path)
  depth_ims = os.listdir(depth_path)

  # rgb_ims = rgb_ims[118:404:2]
  rgb_ims = sample_rgb(rgb_ims,step)
```

```
  rgb_ts = []
  for n in rgb_ims:
    rgb_ts += [float(n[:-4])]

  depth_ts = []
  for n in depth_ims:
    depth_ts += [float(n[:-4])]

  rgb_ts = np.array(rgb_ts)
  depth_ts = np.array(depth_ts)

  rgb_ts_a = []
  depth_ts_a = []

  for n in rgb_ts:
    rgb_ts_a += ['{}'.format(n)+'.png']

    diff_depth = np.abs(depth_ts - n)
    i = np.argmin(diff_depth)

    depth_ts_a += ['{}'.format(depth_ts[i])+'.png']

  return rgb_ts_a, depth_ts_a

def with_gt(ts_gt, ts, data):
  data_a = []
  for t in ts_gt:
    diff = np.abs(ts - t)
    i = np.argmin(diff)
    data_a += [data[i]]

  data_a = np.vstack(data_a)

  return data_a


def gt_with_rgb(ts_gt, ts, data):
  data_a = []
  for t in ts:
    diff = np.abs(ts_gt - t)
    i = np.argmin(diff)
    data_a += [data[i]]

  data_a = np.vstack(data_a)

  return data_a

def gt_with_rgb_start(ts_gt, ts, data, start_rgb, start_gt):
  ts_gt = ts_gt - start_gt + start_rgb
  data_a = []
  for t in ts:
    diff = np.abs(ts_gt - t)
```

```
    i = np.argmin(diff)
    data_a += [data[i]]

  data_a = np.vstack(data_a)

  return data_a
```

10.6 Code: flying

The code for the trajectories is written in C as part of a ROS package. A separate file is written for each trajectory, which will be provided below.

Circle:

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <fstream>
#include <vector>
#include <string>
#include <thread>
#include <mutex>
#include <exception>

#include <ros/ros.h>

#include <GPU_2_Quad_ROS/to_px4.h>
#include <GPU_2_Quad_ROS/from_px4.h>

#define PI 3.14159f

using namespace std;
using namespace ros;




float xic, yic, zic;
float xbc, ybc, zbc;
bool data_recieved;
float yaw_home;
float xvel_sp_speed;

ros::Publisher cmd_pub;

GPU_2_Quad_ROS::from_px4   quad_data;
GPU_2_Quad_ROS::to_px4    cmd_data;



void quadCallback(const GPU_2_Quad_ROS::from_px4 quad_data_msg)
{
```

```
  quad_data = quad_data_msg;
  if (!data_recieved)
  {
    yaw_home = quad_data.yaw;
    data_recieved = true;
  }

}



int move_fb(int forward)
{
  float xyp =   1.0f;
  float zp =  1.0f;

  //xic =  -xyp*(quad_data.x_pos-(-1.2f));
  if (xvel_sp_speed > 1.5f) { xvel_sp_speed = 1.5f; }

  if (forward==1)
  {
    xic =  xvel_sp_speed;
    if (quad_data.x_pos>1.35){ forward = -1; }
  }
  else
  {
    xic =  -xvel_sp_speed;
    if (quad_data.x_pos<-1.35){  forward = 1; } //xvel_sp_speed += 0.25;
  }




  yic =  -xyp*(quad_data.y_pos-(0.0f));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
  cmd_data.yaw_des   =  0;

  return forward;
}
```

```
void hold_1p2m()
{
  float xyp =   0.7f;
  float zp =  0.5f;

  xic =  -xyp*(quad_data.x_pos-(-1.0f));
  yic =  -xyp*(quad_data.y_pos-(0.0f));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
  cmd_data.yaw_des   =  0;
}

void circle(double t)
{
  float xyp =   1.0f;
  float zp =  1.0f;



  float circle_ang =   t/40.0f*2*PI;
  float x_des =     1.0f*cosf(circle_ang+PI);
  float y_des =     1.0f*sinf(circle_ang+PI);

  float yaw_set =    circle_ang;


  xic =  -xyp*(quad_data.x_pos-(x_des));
  yic =  -xyp*(quad_data.y_pos-(y_des));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
```

```
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode     =  7.5f;
  cmd_data.yaw_des   =  yaw_set;
}

void hold_origin()
{
  float xyp =   1.0f;
  float zp =  1.0f;

  xic =  -xyp*(quad_data.x_pos-(-0.0f));
  yic =  -xyp*(quad_data.y_pos-(-0.0f));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
}



int main(int argc, char **argv)
{
  ros::init(argc, argv, "vicon_test");
  ros::NodeHandle nm;
  ros::Rate pub_rate(60);

  double start_time = (double)ros::Time::now().toSec();

  data_recieved = false;
  yaw_home = 0;

  cmd_pub = nm.advertise<GPU_2_Quad_ROS::to_px4>("/px4_to", 2);
  ros::Subscriber quad_data_sub = nm.subscribe("/px4_from", 2, quadCallback);

  xic = 0; yic = 0; zic = 0.2f;
  xbc = 0; ybc = 0; zbc = 0.2f;
```

```
// sleep for 1s to init subscriber
usleep(1000000);

xvel_sp_speed = 0.5f;

int cmd = 0;
int forward = 1;
double time_cmd = (double)ros::Time::now().toSec();

ROS_INFO("Started - launch!");
double t;




while(ros::ok())
{
  t = (double)ros::Time::now().toSec()-time_cmd;
      cmd_data.yaw_des = yaw_home;


  if (cmd==0)
  {
    hold_1p2m();
    if(t > 20.0)
    {
      ROS_INFO("Circle mode!");
              time_cmd = (double)ros::Time::now().toSec();
      cmd++;
    }

  }
  else if (cmd==1)
  {
    circle(t);
    if(t > 80.0)
    {
      ROS_INFO("Hold mode at x=1.2m!  Will then land after 10s!");
      time_cmd = (double)ros::Time::now().toSec();
      cmd++;
    }

  }
  else if (cmd==2)
  {
    hold_1p2m();
    if(t > 10.0)
    {
      //ROS_INFO("Land!");
      cmd_data.z_vel_des = 0.3;
    }
```

```
    }
      if (cmd_data.z_vel_des>0.3)     { cmd_data.z_vel_des=0.3; }
        if (cmd_data.z_vel_des<-0.3)    { cmd_data.z_vel_des=-0.3; }
      if (cmd_data.x_vel_des>1.0)     { cmd_data.x_vel_des=1.0; }
        if (cmd_data.x_vel_des<-1.0)    { cmd_data.x_vel_des=-1.0; }
      if (cmd_data.y_vel_des>1.0)     { cmd_data.y_vel_des=1.0; }
        if (cmd_data.y_vel_des<-1.0)    { cmd_data.y_vel_des=-1.0; }
    //cmd_data.yaw_des = yaw_home;
        cmd_data.mode =      50.0f;
        //ROS_INFO("vz desired: %0.3f", cmd_data.z_vel_des);
        cmd_data.header.stamp =  ros::Time::now();
    cmd_pub.publish(cmd_data);

    ros::spinOnce();
    pub_rate.sleep();
  }



}
```

Rectangle:

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <fstream>
#include <vector>
#include <string>
#include <thread>
#include <mutex>
#include <exception>

#include <ros/ros.h>

#include <GPU_2_Quad_ROS/to_px4.h>
#include <GPU_2_Quad_ROS/from_px4.h>

#define PI 3.14159f

using namespace std;
using namespace ros;



float xic, yic, zic;
float xbc, ybc, zbc;
bool data_recieved;
float yaw_home;
float xvel_sp_speed;

ros::Publisher cmd_pub;
```

```
GPU_2_Quad_ROS::from_px4   quad_data;
GPU_2_Quad_ROS::to_px4    cmd_data;



void quadCallback(const GPU_2_Quad_ROS::from_px4 quad_data_msg)
{
  quad_data = quad_data_msg;
  if (!data_recieved)
  {
    yaw_home = quad_data.yaw;
    data_recieved = true;
  }

}




void turn(double turn_time, double t, float x_set, float y_set, float start_yaw)
{
    float yaw_set =  start_yaw + (PI/2)*(t/turn_time);
  float xyp = 1.0f;
  float zp =  1.0f;

    ROS_INFO("yaw: %f", yaw_set);

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(y_set));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode     =  7.5f;
  cmd_data.yaw_des   =  yaw_set;
}

void fwd(double t, int side)
{
    float speed = 0.1f;
  float xyp = 1.0f;
  float zp =  1.0f;
```

```
    float yaw_set;

    if (side==1) {
        float y_set = -t*speed;

      xic =  -xyp*(quad_data.x_pos-(-1.0f));
      yic =  -xyp*(quad_data.y_pos-(y_set));

        yaw_set = 0.0f;
    }

    else if (side==2) {
        float x_set = -1.0f+t*speed;

      xic =  -xyp*(quad_data.x_pos-(x_set));
      yic =  -xyp*(quad_data.y_pos-(-0.5));

        yaw_set = PI/2;
    }


    else if (side==3) {
        float y_set = -0.5+t*speed;

      xic =  -xyp*(quad_data.x_pos-(1.0f));
      yic =  -xyp*(quad_data.y_pos-(y_set));

        yaw_set = PI;
    }

    else if (side==4) {
        float x_set = 1.0f-t*speed;

      xic =  -xyp*(quad_data.x_pos-(x_set));
      yic =  -xyp*(quad_data.y_pos-(0.5f));

        yaw_set = (PI*3)/2;
    }

    else if (side==5) {
        float y_set = 0.5f-t*speed;

      xic =  -xyp*(quad_data.x_pos-(-1.0f));
      yic =  -xyp*(quad_data.y_pos-(y_set));

        yaw_set = 0.0f;
    }

    else { //shouldn't happen
        xic = 0.0f;
        yic = 0.0f;
        yaw_set = quad_data.yaw;
    }
```

```
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;


  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
    cmd_data.yaw_des   =  yaw_set;

}




void hold_1p2m()
{
  float xyp =   0.7f;
  float zp =  0.5f;

  xic =  -xyp*(quad_data.x_pos-(-1.0f));
  yic =  -xyp*(quad_data.y_pos-(0.0f));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
  cmd_data.yaw_des   =  0;
}

void circle(double t)
{
  float xyp =   1.0f;
  float zp =  1.0f;

  float circle_ang =   t/40.0f*2*PI;
```

```
  float x_des =     1.0f*cosf(circle_ang+PI);
  float y_des =     1.0f*sinf(circle_ang+PI);

  float yaw_set =    circle_ang;


  xic =  -xyp*(quad_data.x_pos-(x_des));
  yic =  -xyp*(quad_data.y_pos-(y_des));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode     =  7.5f;
  cmd_data.yaw_des   =  yaw_set;
}

void hold_origin()
{
  float xyp =   1.0f;
  float zp =  1.0f;

  xic =  -xyp*(quad_data.x_pos-(-0.0f));
  yic =  -xyp*(quad_data.y_pos-(-0.0f));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
}



int main(int argc, char **argv)
{
  ros::init(argc, argv, "vicon_test");
```

```
ros::NodeHandle nm;
ros::Rate pub_rate(60);


data_recieved = false;
yaw_home = 0;

cmd_pub = nm.advertise<GPU_2_Quad_ROS::to_px4>("/px4_to", 2);
ros::Subscriber quad_data_sub = nm.subscribe("/px4_from", 2, quadCallback);

xic = 0; yic = 0; zic = 0.2f;
xbc = 0; ybc = 0; zbc = 0.2f;


// sleep for 1s to init subscriber
usleep(1000000);

xvel_sp_speed = 0.5f;

int cmd = 0;
  int cmd_sqr = 0;
double time_cmd = (double)ros::Time::now().toSec();

  ROS_INFO("Change 2");
ROS_INFO("Started - launch!");
double t;
  int num_laps = 0;
  double turn_time = 5.0f;


while(ros::ok())
{
  t = ((double)ros::Time::now().toSec())-time_cmd;
      cmd_data.yaw_des = yaw_home;


  if (cmd==0)
  {
    hold_1p2m();
    if(t > 20.0)
    {
      ROS_INFO("Square mode!");
              time_cmd = (double)ros::Time::now().toSec();
      cmd++;
    }

  }
  else if (cmd==1)
  {
          if (cmd_sqr==0){
              fwd(t, 1);
              if(t > 5.0) {
      ROS_INFO("First half side done, initiating turn");
```

```
            time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
                }
        }
        else if (cmd_sqr==1){
            turn(turn_time, t, -1.0f, -0.5f, 0.0f);
            ROS_INFO("t: %f", t);
            if(t >turn_time) {
ROS_INFO("Turn finished");
            time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
                }
        }
        else if (cmd_sqr==2){
            fwd(t, 2);
            if(t > 20.0) {
ROS_INFO("Second side done, initiating turn");
            time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
                }
        }
        else if (cmd_sqr==3){
            turn(turn_time, t, 1.0f, -0.5f, PI/2);
            if(t >turn_time) {
ROS_INFO("Turn finished");
            time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
                }
        }
        else if (cmd_sqr==4){
            fwd(t, 3);
            if(t > 10.0) {
ROS_INFO("Third side done, initiating turn");
            time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
                }
        }
        else if (cmd_sqr==5){
            turn(turn_time, t, 1.0f, 0.5f, PI);
            if(t >turn_time) {
ROS_INFO("Turn finished");
            time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
                }
        }
        else if (cmd_sqr==6){
            fwd(t, 4);
            if(t > 20.0) {
ROS_INFO("Fourth side done, initiating turn");
            time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
                }
        }
```

```
            else if (cmd_sqr==7){
                turn(turn_time, t, -1.0f, 0.5f, (PI*3)/2);
                if(t >turn_time) {
      ROS_INFO("Turn finished");
                time_cmd = (double)ros::Time::now().toSec();
      cmd_sqr++;
                    }
            }
            else if (cmd_sqr==8){
                fwd(t, 5);
                if(t > 5.0) {
                num_laps++;
                if (num_laps==1) {
            ROS_INFO("Second half side done, begining new lap");
                    time_cmd = (double)ros::Time::now().toSec();
            cmd_sqr=0;
                }
                }
            }
    if(num_laps==2)
    {
      ROS_INFO("Hold mode at x=1.2m!  Will then land after 10s!");
      time_cmd = (double)ros::Time::now().toSec();
      cmd++;
    }

  }
  else if (cmd==2)
  {
    hold_1p2m();
    if(t > 10.0)
    {
      //ROS_INFO("Land!");
      cmd_data.z_vel_des = 0.3;
    }

  }
    if (cmd_data.z_vel_des>0.3)     { cmd_data.z_vel_des=0.3; }
      if (cmd_data.z_vel_des<-0.3)    { cmd_data.z_vel_des=-0.3; }
    if (cmd_data.x_vel_des>1.0)     { cmd_data.x_vel_des=1.0; }
      if (cmd_data.x_vel_des<-1.0)    { cmd_data.x_vel_des=-1.0; }
    if (cmd_data.y_vel_des>1.0)     { cmd_data.y_vel_des=1.0; }
      if (cmd_data.y_vel_des<-1.0)    { cmd_data.y_vel_des=-1.0; }
  //cmd_data.yaw_des = yaw_home;
      cmd_data.mode =      50.0f;
      //ROS_INFO("vz desired: %0.3f", cmd_data.z_vel_des);
      cmd_data.header.stamp =  ros::Time::now();
  cmd_pub.publish(cmd_data);

  ros::spinOnce();
  pub_rate.sleep();
}
```

```
}
```

Lawnmower:

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <fstream>
#include <vector>
#include <string>
#include <thread>
#include <mutex>
#include <exception>

#include <ros/ros.h>

#include <GPU_2_Quad_ROS/to_px4.h>
#include <GPU_2_Quad_ROS/from_px4.h>

#define PI 3.14159f

using namespace std;
using namespace ros;




float xic, yic, zic;
float xbc, ybc, zbc;
bool data_recieved;
float yaw_home;
float xvel_sp_speed;

ros::Publisher cmd_pub;

GPU_2_Quad_ROS::from_px4   quad_data;
GPU_2_Quad_ROS::to_px4    cmd_data;



void quadCallback(const GPU_2_Quad_ROS::from_px4 quad_data_msg)
{
  quad_data = quad_data_msg;
  if (!data_recieved)
  {
    yaw_home = quad_data.yaw;
    data_recieved = true;
  }

}
```

```
void turnr(double turn_time, double t, float x_set, float y_set, float start_yaw)
{
    float yaw_set =  start_yaw + (PI/2)*(t/turn_time);
  float xyp = 1.0f;
  float zp =  1.0f;

    ROS_INFO("yaw: %f", yaw_set);

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(y_set));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode     =  7.5f;
  cmd_data.yaw_des   =  yaw_set;
}

void turnl(double turn_time, double t, float x_set, float y_set, float start_yaw)
{
    float yaw_set =  start_yaw - (PI/2)*(t/turn_time);
  float xyp = 1.0f;
  float zp =  1.0f;

    ROS_INFO("yaw: %f", yaw_set);

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(y_set));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode     =  7.5f;
```

```
  cmd_data.yaw_des    =  yaw_set;
}

void fwd(double t, int side)
{
    float speed = 0.1f;
  float xyp = 1.0f;
  float zp =  1.0f;

    float yaw_set;

    if (side==1) {
        float y_set = -t*speed;

      xic =  -xyp*(quad_data.x_pos-(-1.0f));
      yic =  -xyp*(quad_data.y_pos-(y_set));

        yaw_set = (PI*3)/2;
    }

    else if (side==2) {
        float x_set = -1.0f+t*speed;

      xic =  -xyp*(quad_data.x_pos-(x_set));
      yic =  -xyp*(quad_data.y_pos-(-0.5));

        yaw_set = 0;
    }


    else if (side==3) {
        float y_set = -0.5+t*speed;

      xic =  -xyp*(quad_data.x_pos-(-0.5f));
      yic =  -xyp*(quad_data.y_pos-(y_set));

        yaw_set = PI/2;
    }

    else if (side==4) {
        float x_set = -0.5f+t*speed;

      xic =  -xyp*(quad_data.x_pos-(x_set));
      yic =  -xyp*(quad_data.y_pos-(0.5f));

        yaw_set = 0;
    }


    else if (side==5) {
        float y_set = 0.5-t*speed;

      xic =  -xyp*(quad_data.x_pos-(0.0f));
```

```
  yic =  -xyp*(quad_data.y_pos-(y_set));

    yaw_set = (3*PI)/2;
}

else if (side==6) {
    float x_set = 0.0f+t*speed;

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(-0.5));

    yaw_set = 0;
}

else if (side==7) {
    float y_set = -0.5+t*speed;

  xic =  -xyp*(quad_data.x_pos-(0.5f));
  yic =  -xyp*(quad_data.y_pos-(y_set));

    yaw_set = PI/2;
}

else if (side==8) {
    float x_set = 0.5f+t*speed;

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(0.5f));

    yaw_set = 0;
}

else if (side==9) {
    float y_set = 0.5-t*speed;

  xic =  -xyp*(quad_data.x_pos-(1.0f));
  yic =  -xyp*(quad_data.y_pos-(y_set));

    yaw_set = (3*PI/2);
}

else if (side==10) {
    float x_set = 1.0f-t*speed;

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(-0.5f));

    yaw_set = PI;
}

else if (side==11) {
    float y_set = -0.5+t*speed;
```

```
  xic =  -xyp*(quad_data.x_pos-(0.5f));
  yic =  -xyp*(quad_data.y_pos-(y_set));

    yaw_set = PI/2;
}

else if (side==12) {
    float x_set = 0.5f-t*speed;

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(0.5f));

    yaw_set = PI;
}

else if (side==13) {
    float y_set = 0.5-t*speed;

  xic =  -xyp*(quad_data.x_pos-(0.0f));
  yic =  -xyp*(quad_data.y_pos-(y_set));

    yaw_set = (3*PI)/2;
}

else if (side==14) {
    float x_set = 0.0f-t*speed;

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(-0.5f));

    yaw_set = PI;
}

else if (side==15) {
    float y_set = -0.5+t*speed;

  xic =  -xyp*(quad_data.x_pos-(-0.5f));
  yic =  -xyp*(quad_data.y_pos-(y_set));

    yaw_set = PI/2;
}

else if (side==16) {
    float x_set = -0.5f-t*speed;

  xic =  -xyp*(quad_data.x_pos-(x_set));
  yic =  -xyp*(quad_data.y_pos-(0.5f));

    yaw_set = PI;
}

else if (side==17) {
    float y_set = 0.5-t*speed;
```

```
      xic =  -xyp*(quad_data.x_pos-(-1.0f));
      yic =  -xyp*(quad_data.y_pos-(y_set));

        yaw_set = (3*PI)/2;
    }


    else { //shouldn't happen
        xic = 0.0f;
        yic = 0.0f;
        yaw_set = quad_data.yaw;
    }

  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;


  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
    cmd_data.yaw_des   =  yaw_set;

}




void hold_1p2m()
{
  float xyp =   0.7f;
  float zp =  0.5f;

  xic =  -xyp*(quad_data.x_pos-(-1.0f));
  yic =  -xyp*(quad_data.y_pos-(0.0f));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;
```

```
  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
  cmd_data.yaw_des   =  0;
}

void circle(double t)
{
  float xyp =   1.0f;
  float zp =  1.0f;

  float circle_ang =   t/40.0f*2*PI;
  float x_des =     1.0f*cosf(circle_ang+PI);
  float y_des =     1.0f*sinf(circle_ang+PI);

  float yaw_set =    circle_ang;


  xic =  -xyp*(quad_data.x_pos-(x_des));
  yic =  -xyp*(quad_data.y_pos-(y_des));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode     =  7.5f;
  cmd_data.yaw_des   =  yaw_set;
}

void hold_origin()
{
  float xyp =   1.0f;
  float zp =  1.0f;

  xic =  -xyp*(quad_data.x_pos-(-0.0f));
  yic =  -xyp*(quad_data.y_pos-(-0.0f));
  zic =  -zp*(quad_data.z_pos-(-1.5f));

  float yaw =     quad_data.yaw;
  float ang_diff =  yaw-atan2f(yic,xic);
  float vxy_mag =    sqrtf(xic*xic+yic*yic);

  xbc =  vxy_mag*cosf(-ang_diff);
  ybc =  vxy_mag*sinf(-ang_diff);
```

```
  zbc =  zic;

  cmd_data.x_vel_des  =  xbc;
  cmd_data.y_vel_des  =  ybc;
  cmd_data.z_vel_des  =  zbc;
  cmd_data.mode =      7.5f;
}



int main(int argc, char **argv)
{
  ros::init(argc, argv, "vicon_test");
  ros::NodeHandle nm;
  ros::Rate pub_rate(60);


  data_recieved = false;
  yaw_home = 0;

  cmd_pub = nm.advertise<GPU_2_Quad_ROS::to_px4>("/px4_to", 2);
  ros::Subscriber quad_data_sub = nm.subscribe("/px4_from", 2, quadCallback);

  xic = 0; yic = 0; zic = 0.2f;
  xbc = 0; ybc = 0; zbc = 0.2f;


  // sleep for 1s to init subscriber
  usleep(1000000);

  xvel_sp_speed = 0.5f;

  int cmd = 0;
    int cmd_sqr = 0;
  double time_cmd = (double)ros::Time::now().toSec();

    ROS_INFO("Change 2");
  ROS_INFO("Started - launch!");
  double t;
    int num_laps = 0;
    double turn_time = 5.0f;


  while(ros::ok())
  {
    t = ((double)ros::Time::now().toSec())-time_cmd;
        cmd_data.yaw_des = yaw_home;


    if (cmd==0)
    {
      hold_1p2m();
      if(t > 20.0)
```

```
    {
      ROS_INFO("Lawn mower mode!");
              time_cmd = (double)ros::Time::now().toSec();
      cmd++;
    }

  }
  else if (cmd==1)
  {
          if (cmd_sqr==0){
              fwd(t, 1);
              if(t > 5.0) {
      ROS_INFO("Initiating turn");
              time_cmd = (double)ros::Time::now().toSec();
      cmd_sqr++;
                  }
          }
          else if (cmd_sqr==1){
              turnr(turn_time, t, -1.0f, -0.5f, 0.0f);
              ROS_INFO("t: %f", t);
              if(t >turn_time) {
      ROS_INFO("Turn finished");
              time_cmd = (double)ros::Time::now().toSec();
      cmd_sqr++;
                  }
          }
          else if (cmd_sqr==2){
              fwd(t, 2);
              if(t > 5.0) {
      ROS_INFO("Initiating turn");
              time_cmd = (double)ros::Time::now().toSec();
      cmd_sqr++;
                  }
          }
          else if (cmd_sqr==3){
              turnr(turn_time, t, 0.5f, -0.5f, PI/2);
              if(t >turn_time) {
      ROS_INFO("Turn finished");
              time_cmd = (double)ros::Time::now().toSec();
      cmd_sqr++;
                  }
          }
          else if (cmd_sqr==4){
              fwd(t, 3);
              if(t > 10.0) {
      ROS_INFO("Initiating turn");
              time_cmd = (double)ros::Time::now().toSec();
      cmd_sqr++;
                  }
          }
          else if (cmd_sqr==5){
              turnl(turn_time, t, 0.5f, 0.5f, PI);
              if(t >turn_time) {
```

```
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==6){
        fwd(t, 4);
        if(t > 5.0) {
ROS_INFO("Initiating turn");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==7){
        turnl(turn_time, t, 0.0f, 0.5f, (PI*3)/2);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==8){
        fwd(t, 5);
        if(t > 10.0) {
  ROS_INFO("Initialing turn");
        time_cmd = (double)ros::Time::now().toSec();
        }
        }
    }
    else if (cmd_sqr==9){
        turnr(turn_time, t, 0.0f, -0.5f, 0.0f);
        ROS_INFO("t: %f", t);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==10){
        fwd(t, 6);
        if(t > 5.0) {
ROS_INFO("Initiating turn");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==11){
        turnr(turn_time, t, -0.5f, -0.5f, PI/2);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
```

```
    }
    else if (cmd_sqr==12){
        fwd(t, 7);
        if(t > 10.0) {
ROS_INFO("Initiating turn");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==13){
        turnl(turn_time, t, -0.5f, 0.5f, PI);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==14){
        fwd(t, 8);
        if(t > 5.0) {
ROS_INFO("Initiating turn");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==15){
        turnl(turn_time, t, -1.0f, 0.5f, (PI*3)/2);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==16){
        fwd(t, 9);
        if(t > 10.0) {
  ROS_INFO("Initialing turn");
        time_cmd = (double)ros::Time::now().toSec();
        }
        }
    }


    else if (cmd_sqr==17){
        turnl(turn_time, t, 1.0f, -0.5f, PI);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==18){
        fwd(t, 10);
```

```
        if(t > 5.0) {
ROS_INFO("Initiating turn");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==19){
        turnl(turn_time, t, 0.5f, -0.5f, (PI*3)/2);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==20){
        fwd(t, 11);
        if(t > 10.0) {
  ROS_INFO("Initialing turn");
        time_cmd = (double)ros::Time::now().toSec();
        }
        }
    }
    else if (cmd_sqr==21){
        turnr(turn_time, t, 0.5f, 0.5f, PI);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==22){
        fwd(t, 12);
        if(t > 5.0) {
ROS_INFO("Initiating turn");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==23){
        turnr(turn_time, t, 0.0f, 0.5f, (PI*3)/2);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==24){
        fwd(t, 13);
        if(t > 10.0) {
  ROS_INFO("Initialing turn");
        time_cmd = (double)ros::Time::now().toSec();
        }
        }
```

```
    }
    else if (cmd_sqr==25){
        turnl(turn_time, t, 0.0f, -0.5f, PI);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==26){
        fwd(t, 14);
        if(t > 5.0) {
ROS_INFO("Initiating turn");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==27){
        turnl(turn_time, t, -0.5f, -0.5f, (PI*3)/2);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==28){
        fwd(t, 15);
        if(t > 10.0) {
  ROS_INFO("Initialing turn");
        time_cmd = (double)ros::Time::now().toSec();
        }
        }
    }
    else if (cmd_sqr==29){
        turnr(turn_time, t, -0.5f, 0.5f, PI);
        if(t >turn_time) {
ROS_INFO("Turn finished");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==30){
        fwd(t, 16);
        if(t > 5.0) {
ROS_INFO("Initiating turn");
        time_cmd = (double)ros::Time::now().toSec();
cmd_sqr++;
            }
    }
    else if (cmd_sqr==31){
        turnr(turn_time, t, -1.0f, 0.5f, (PI*3)/2);
        if(t >turn_time) {
ROS_INFO("Turn finished");
```

```
                time_cmd = (double)ros::Time::now().toSec();
        cmd_sqr++;
                    }
            }
            else if (cmd_sqr==32){
                fwd(t, 17);
                if(t > 5.0) {
            ROS_INFO("Hold mode at x=1.2m!  Will then land after 10s!");
            time_cmd = (double)ros::Time::now().toSec();
            cmd++;
      }

    }
    else if (cmd==2)
    {
      hold_1p2m();
      if(t > 10.0)
      {
        //ROS_INFO("Land!");
        cmd_data.z_vel_des = 0.3;
      }

    }
      if (cmd_data.z_vel_des>0.3)     { cmd_data.z_vel_des=0.3; }
        if (cmd_data.z_vel_des<-0.3)    { cmd_data.z_vel_des=-0.3; }
      if (cmd_data.x_vel_des>1.0)     { cmd_data.x_vel_des=1.0; }
        if (cmd_data.x_vel_des<-1.0)    { cmd_data.x_vel_des=-1.0; }
      if (cmd_data.y_vel_des>1.0)     { cmd_data.y_vel_des=1.0; }
        if (cmd_data.y_vel_des<-1.0)    { cmd_data.y_vel_des=-1.0; }
    //cmd_data.yaw_des = yaw_home;
        cmd_data.mode =      50.0f;
        //ROS_INFO("vz desired: %0.3f", cmd_data.z_vel_des);
        cmd_data.header.stamp =  ros::Time::now();
    cmd_pub.publish(cmd_data);

    ros::spinOnce();
    pub_rate.sleep();
  }



}
```