EE106A: Lab 7 - Multiview Geometry and Feature Tracking [*]

Fall 2020

Goals

By the end of this lab you should be able to:

- Use OpenCV to undistort imges from calibrated cameras and extract keypoint features from images.
- Match features between two images of the same scene to find corresponding points.
- Reject outlier matches using the epipolar constraint.
- Triangulate feature matches to find their location in 3D space.

Relevant Tutorials and Documentation:

- OpenCV Python Tutorial
- Feature Matching Python Tutorial

If you get stuck at any point in the lab you may submit a help request during your lab section at `https://tinyurl.com/106alabs20`. You can check you position on the queue at `https://tinyurl.com/106Fall20LabQueue`.

Note: For all labs this semester you may collaborate with a lab partner but we expect everyone to do every part of the labs themselves. You should work closely with your partner to overcome obstacles in the labs but each member of the team must do the lab themselves.

Contents

[*]Developed for Fall 2020 by Ritika Shrivastava, Jay Monga, and Amay Saxena

1 Introduction

In this lab you will work with data collected from stereo RGB camera pair mounted on a drone. This data comes from the EuRoC MAV dataset. You will write a node that subscribes to two image streams coming from the left and right cameras of the stereo pair respectively. Your node will then extract relevant visual features from the images and triangulate them to find the 3D coordinates of the feature points. Your node will then publish the resulting pointcloud of visual features, which you will visualize in RViz.

2 Starter Code

Create a new workspace in your `~/ros_workspaces` directory called `lab7` and clone the starter code into the `src` subdirectory. After creating your workspace and copying over the contents of `lab7_starter` into `lab7/src`, use `catkin_make` to build your workspace.

```
git clone https://github.com/ucb-ee106/lab7_starter.git
```

After creating and building your workspace, navigate to the `ros_numpy` directory and run

```
python setup.py install --user
```

The starter code includes the following packages:

1. `ros_numpy`: A very useful ROS package that can be used to convert between ROS message datatypes and numpy datatypes. Check out the documentation at `http://wiki.ros.org/ros_numpy`.

2. `stereo_pointcloud`: This is the package that contains all the skeleton code. All files that you will need to edit are in this package.

3 Overview

In this lab you will build a ROS node that subscribes to two image topics, one from the left camera and one from the right camera of a stereo camera pair mounted on a drone. Then, with each pair of `(left, right)` images, the node will:

1. Extract visual features from the left and right image.

2. Match visual features between the two images in a nearest-neighbours fashion to come up with a preliminary set of corresponding points in the two images.

3. Reject spurious matches by enforcing the epipolar constraint between feature matches in the two images.

4. Triangulate to find the 3D coordinates of each feature point in the robot's reference frame.

5. Publish a pointcloud consisting of the locations of all interesting features as seen from the robot's reference frame.

In order to do this, you will write code to perform feature extraction and matching, to perform outlier rejection using the known epipolar geometry between the two cameras, and to triangulate a feature match to compute its spatial location given its image coordinates as seen from the left and right cameras.

3.1 The Data

Unfortunately, we are not able to acquire a drone for every student given the current situation. Luckily, the EuRoC MAV Dataset gives us a nice collection of data from a drone flying around an indoor room that we can use. On the linked page, you can see that there is data taken from several scenes. We will only be focusing on the "Machine Hall 01" dataset, although you are free to play around with the other data on your own. There are links to download bag files and other data, however we have combined all our relevant data into one bag file with a shortened trajectory to reduce file size.

Since bag files store raw information from a wide variety of topics, including those carrying Image messages, they can be quite large in size. Because of this, we are distributing the bag file via Goolge Drive instead of through the git repo. It should be about 1.4GB in size, so make sure you have space for it before downloading the file. You can get the bag file by navigating to the `/bagfiles` folder of the `lab7_starter` package.

```
cd ~/ros_workspaces/lab7/src/stereo_pointcloud/bagfiles
```

and running

```
chmod +x download_bag.sh
./download_bag.sh
```

This will take a few minutes.

You should see a newly created `drone_data.bag` file. Recall that bag files let us record all messages published over select ROS topics to be played back later. We can see what information our bag file has recorded by running

```
rosbag info drone_data.bag
```

You should be able to see a lot of useful information about the bag file here, including its size, its duration, the topics recorded and their message types, etc. We can play back the recorded information by running

```
rosbag play drone_data.bag
```

and you should be able to see similar information about the published topics through a simple `rostopic list` (don't forget to start a ROS master node first!).

Since our bagged data is now being published, we can view it in Rviz to get a feel for what information we have to work with. Open up Rviz with

```
rosrun rviz rviz
```

and then start playing the bag file. You will want to set Global Options > Fixed Frame to be `world`, and then you can use Add > Display Type > TF to see the different frames of the drone moving over time as the bag file plays out. To see the actual view of the drone's cameras, you need to do Add > Display Type > Image twice and the topic of one Image display to `/left/image_raw` and the the other Image display to `/right/image_raw`. You should be able to see a stream of image data from each camera. You may notice that the images appear to have some curvature near their edges, due to distortion from the camera. We will explain how to deal with this later.

Armed with knowledge of the information we can publish from our bag file, we can now setup our subscribers for the main node of the project, located in the `__init__` function of `stereo_pointcloud.py`.

3.2 Task 0

By now, you should be able to visualize messages stored in the bag file through Rviz and properly instantiat subscribers in `stereo_pointcloud.py`.

4 Camera Calibration Review: Pinhole Model

Like in Lab 6, we will be using calibrated cameras in this lab, which means that we know their camera intrinsic matrices K. Recall from lecture and Lab 6 that if $p = (X, Y, Z)$ is a point written in the camera's reference frame, then the homogeneous image coordinates $x = (u, v, 1)$ of the image of p are given by

$$x = \frac{1}{Z} K p$$

further recall that we rearrange the above equation for convenience, and instead state that there exists a scalar "depth" λ such that

$$\lambda x = K p \tag{1}$$

4.1 Normalized image coordinates

For convenience, we will "normalize" our image coordinates. In particular, if x is the pixel coordinates of a point p, then we call $\bar{x} = K^{-1} x$ the *normalized image coordinates* of p. These normalized coordinates have the property that

$$\lambda \bar{x} = p \tag{2}$$

i.e. we can ignore the multiplication by K on the right hand side. When the camera matrix K is known (i.e. the camera is calibrated), we can always freely switch between the normalized and regular coordinates simply by multiplying by K^{-1}. In most of this lab, we will work with normalized image coordinates for simplicity, so your implementations should always start by normalizing the coordinates of an image point before using them.

4.2 Image Distortion

One of the ways in which our camera differs from a true pinhole camera is that it adds nonlinear distortion to the image in addition to the standard pinhole projection. We model this in the form of a "radial-tangential" distortion. When you visualized the raw images in RViz in the previous section, you may have noticed that straight lines do not always show up as straight in the image. Instead, they show up curved, with the curvature being stronger towards the periphery of the image. Luckily, we have a good model for the distortion of this image. As part of the camera calibration parameters, we also have the *distortion coefficients* of the camera, which parameterize this distortion. We can use these coefficients to undistort the image using the `opencv` function `cv2.undistort`. This has already been done for you.

Make sure to have the bagfile playing, and then run the main node using

```
rosrun stereo_pointcloud stereo_pointcloud.py
```

The undistorted images are now being published to the topics `/drone/left/undistorted_image` and `/drone/right/undistorted_image`. Visualize one of these topics in RViz and compare the image to the undistorted version.

5 Stereo Vision

A *calibrated stereo pair* is a pair of cameras attached rigidly to the robot such that (1) both cameras are calibrated, so the camera matrices K_1, K_2 are known and (2) the static transform $g_{21} = (R, T)$ is known. We then generally speak of the "left" and "right" cameras of a given stereo pair when referring to them. In this lab, we will be working with data collected from a drone that is equipped with such a calibrated stereo pair of cameras.

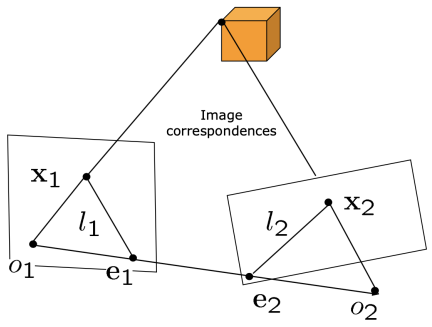

Figure 1: A pair of cameras looking at the same scene. o_1 and o_2 are the optical centers of the cameras. x_1 and x_2 are the image coordinates of the corner of the cube in image 1 and image 2 respectively. Since x_1 and x_2 are the coordinates of the same point in the two image frames, they are called *image correspondences* or *corresponding points* between the images. l_1 and l_2 are the epilines (see section 7.3).

5.1 Corresponding points

A pair of image coordinates x_1 in the frame of the first camera and x_2 in the frame of the second camera are called *corresponding points* if they are both the image of the same point in 3D space. A set of such pairs are then called a set of *point correspondences*. In order to infer the structure of the scene from our stereo pair of cameras, we will need to first find a set of point correspondences between the two images. Recall that given only the image coordinates of a point in one image, we cannot extract its location in 3D space, since all points lying along some ray starting at the optical center of the camera get projected to the same point in the image plane. However, given the coordinates of the same point in two *distinct* images (i.e. when there is a nonzero shift between the two cameras), we can in fact solve for the 3D coordinates of the point they represent. We will see in more detail how to do this in section 8.

In this lab, our goal is to extract visually interesting features from our image stream and track their location in 3D space. In order to do this, we will need to find the corresponding images of the feature point in both cameras of our stereo pair. Then, we will be able to infer the location of that point in the robot's reference frame.

6 Feature Extraction

In dealing with image data we often wish to abstract away some information from the image that allows us to compare the scenes represented in different images in some quantitative way. This is often done by extracting visual features from the image, and then computing a description vector for each feature. There is no concrete definition for what constitutes a "visual feature", and different applications require different choices of features and descriptors. For real-time computer vision applications, a popular choice is to use visual primitives like corners and edges. The most popular feature extraction and description algorithms used in real-time applications such as structure-from-motion and visual SLAM usually extract corners or "interest points". i.e. they extract point-like features from the image. Such features are known as *keypoints*. Point-like features are convenient because the location of such a feature in an image can be described by a single set of 2D image coordinates, and the feature corresponds to objects in the real world in a straightforward way: it is the image of some *point* in 3D space.

Once we have extracted a keypoint, we also need to extract a *feature description* which is computed by featurizing a patch of pixels centered at the keypoint. Popular feature description algorithms include SURF, SIFT, BRIEF, ORB, BRISK etc, which you are encouraged to look up on your own. These algorithms largely differ from each other in how they choose to describe the neighbourhood of the keypoint.

6.1 Feature description

We wish to use the visual features we extract to compare different images of the same scene to detect when two keypoints correspond to the same object in the scene. As such, we would like the feature description to have some desirable properties. An ideal feature descriptor would be one that always gave us a unique description vector for each point in 3D space, so that we would get perfect matches whenever we compared feature descriptions of the same point in two different images. This is, of course, impossible, so we seek feature descriptions with some desirable set of approximate guarantees. Different feature extraction algorithms claim different invariance properties, but in general

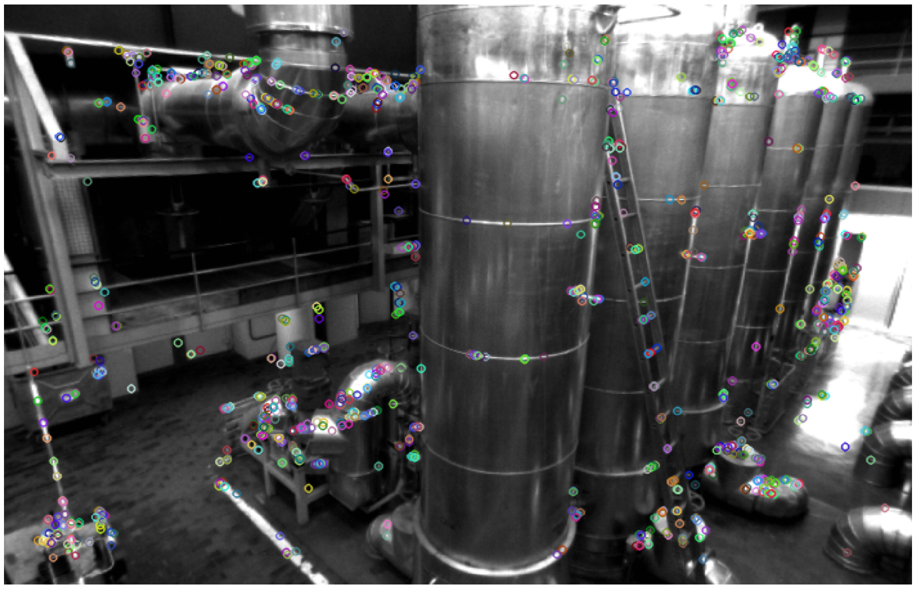

Figure 2: BRISK Features extracted from one frame of the trajectory. This is an image from the left camera of the stereo pair. Notice that only visually distinctive points in the image get extracted, such as corners or points of sudden intensity variation.

we would like our features to be robust to translations and rotations. i.e. a feature corresponding to a point in the scene should get approximately the same descriptor even after the camera rotates or translates by some amount. Additionally, algorithms vary in terms of the kinds of feature description vectors they extract. For instance, feature descriptions extracted by the SURF and SIFT algorithms are vectors of 128 floating point numbers, while those extracted by BRIEF, ORB or BRISK are 512-bit long binary vectors.

6.2 BRISK - Binary Robust Invariant Scalable Keypoints

In this lab, we will be using BRISK, which stands for *Binary Robust Invariant Scalable Keypoints.* The BRISK extractor is optimized for computational speed, while maintaining the same repeatability and invariance guarantees in practice as other state of the art algorithms. It is a keypoint descriptor, and uses the AGAST corner detection algorithm as its underlying feature extractor. It produces a 512-bit binary description vector for each keypoint. In the next section, we will discuss how using binary descriptors makes feature matching very computationally efficient.

7 Feature Matching

Recall that our goal is to come up with corresponding points between the left and right image of the stereo camera pair. As described above, we will begin by extracting BRISK features from both images. We will then compare the description vectors of the features in both images to find the closest matches. We will do this in a nearest neighbours fashion.

7.1 Nearest Neighbours Search

We have extracted feature descriptors $\{f_i\}_{i=1}^{n} \subset \mathbb{R}^d$ from image 1 and $\{g_i\}_{i=1}^{m} \subset \mathbb{R}^d$ from image 2. For each feature f_i, we compute the distance of this feature to every feature in image 2 (in the $d-$dimensional feature space). We then do the same for each feature in image 2, finding its distance to each feature in image 1. A pair (f_i, g_j) is taken to be a *match* if g_j is the closest neighbour to f_i in image 2, *and* f_i is the closest feature to g_j in image 1.

Note that in order to perform this matching, we need to define the *distance* between two feature descriptors. When the feature descriptors f_i and g_j are just floating point vectors, we can use the standard euclidean norm $d(f_i, g_j) = \|f_i - g_j\|$. Recall, however, that BRISK, the feature extractor we are using, outputs descriptors of *binary* vectors. So to compare two such features, we will be using a different distance metric called the *Hamming distance.*

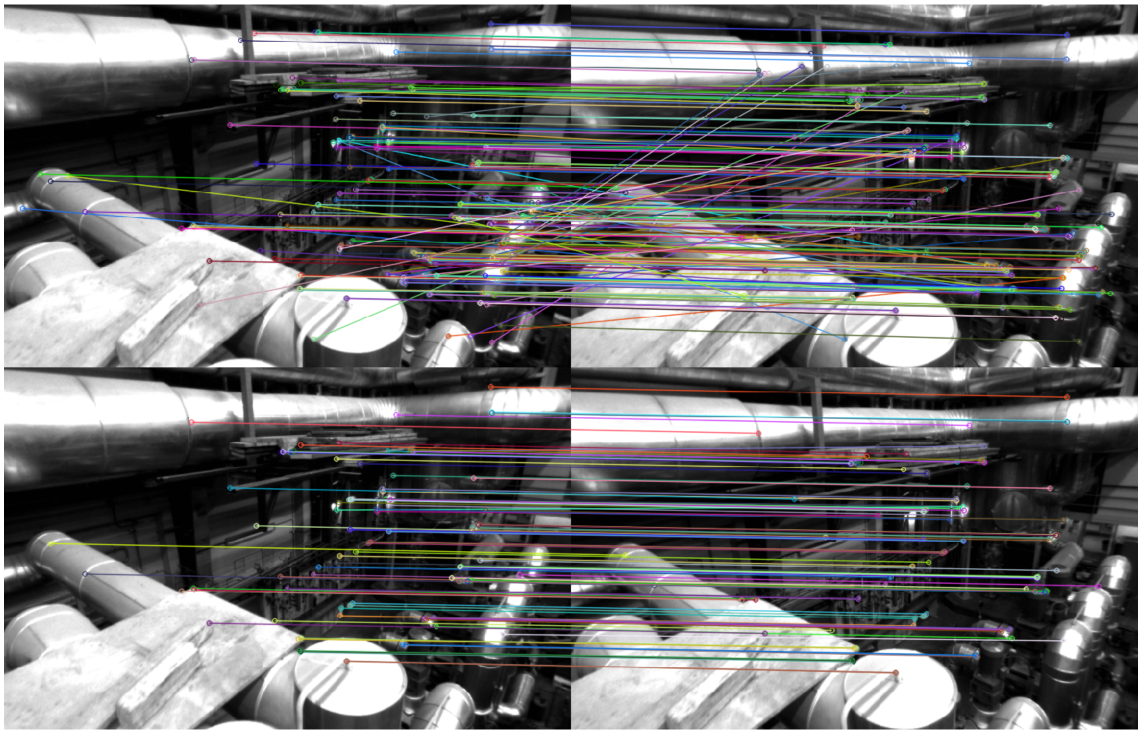

Figure 3: Shows the result of feature matching between a pair of images from the left and right cameras respectively. Top: set of feature matches after simple nearest-neighbours matching. Bottom: filtered matches after rejecting all matches that violate the epipolar constraint.

7.2 Hamming Distance

The *Hamming distance* is a distance metric defined on the space of binary sequences. Given two binary sequences $a = (a_1, \cdots, a_d)$ and $b = (b_1, \cdots, b_d)$ with $a_i, b_i \in \{0, 1\}$, the Hamming distance $d(a, b)$ between them is the number of bits in which a and b *differ*. In other words, it is the minimum number of entries that would need to be changed to make a and b match. Note that unlike euclidean distance between floating point vectors, the Hamming distance between two binary vectors is always an integer.

Using binary description vectors allows us to compare them using the Hamming distance, which is generally faster to compute than the Euclidean distance between comparably-sized floating point descriptors. This is because the hamming distance can be compared using an XOR operation, which is much more efficient than floating point multiplications.

$$d(a, b) = \sum_{i=1}^{d} a_i \boxplus b_i$$

where $\boxplus$ is XOR. We can see that the hamming distance between two bitstrings is exactly equal to the number of active bits in their bitwise XOR.

7.2.1 Task 1

First, uncomment the call to `self.process_images` in the function `camera_callback` in `stereo_pointcloud.py`. Then complete the `extract_and_match_features` function in `stereo_pointcloud.py` by creating a `cv2.BFMatcher` object and then completing the call to `matcher.match`.

- Hint: You will need to pass in an argument to specify the distance function OpenCV should use.
- Hint: This link will be helpful.

To test your implementation, make sure the bag file is playing and first run the main node using

```
rosrun stereo_pointcloud stereo_pointcloud.py
```

and then visualize your feature matches by running

```
rosrun image_view image_view image:=/drone/matches
```

Note: You will get a print statement from stereo_pointcloud.py saying that the number of keypoints in the pointcloud is 0. This is expected, as we have not implemented triangulation yet and so are not publishing any pointclouds.

This will show you an array of four images. The top two images are the left and right images with the feature matches marked on them. The bottom two are the same left and right images. Currently, both the top and bottom will show the same feature matches. In the next section, we will implement functionality to reject bad matches. Once we do that, we will be able to run this visualization again and we will see the filtered matches on the bottom.

7.3 Outlier Rejection: Enforcing the Epipolar Constraint

Despite our best efforts at using nearest neighbours to match features, we will inevitably end up with false matches. This is simply due to the fact that our features are a very local description of the keypoints and hence spurious matches are possible between different points that locally look similar. We want some way to detect and eliminate such spurious matches. We will do this by enforcing the *Epipolar constraint.*

Let (x_1, x_2) be a feature match with *normalized* image coordinates x_1 in image 1 and x_2 in image 2. We know the transform (R, T) between the two cameras, and hence we know the essential matrix $E = \hat{T}R$. Recall that if these two keypoints do in fact correspond to the same point in 3D space, then they must satisfy the Epipolar constraint, i.e. it must be the case that

$$x_2^T E x_1 = 0$$

Of course, since we are dealing with real world noisy data, this quantity will never be exactly zero even for correct matches. So, we will define some geometrically meaningful error for when this constraint is violated.

Definition. The *epiline* corresponding to point x_2 is a line in the image plane of frame 1 which is the set of all points in image 1 that satisfy the epipolar constraint with x_2. If we define $l_1 = E^T x_2$, then the epiline is the set of all points x_1 in normalized homoegeneous image 1 coordinates that satisfy $l_1^T x_1 = 0$.

Likewise, the *epiline* corresponding to point x_1 is a line in the image plane of frame 2 which is the set of all points in image 2 that satisfy the epipolar constraint with x_1. If we define $l_2 = Ex_1$, then the epiline is the set of all points x_2 in normalized homoegeneous image 2 coordinates that satisfy $l_2^T x_2 = 0$.

Let $l_1 = E^T x_2 = (a_1, b_1, c_1)$. Then the first epiline is the set of all points $x_1 = (u_1, v_1, 1)$ that satisfy the equation $a_1u_1 + b_1v_1 + c_1 = 0$. Likewise, if $l_2 = Ex_1 = (a_2, b_2, c_2)$ then the second epiline is the set of all points $x_2 = (u_2, v_2, 1)$ that satisfy the equation $a_2u_2 + b_2v_2 + c_2 = 0$.

If x_1 does not lie on the epiline corresponding to x_2 then we should reject the match. In turn, if x_2 does not lie on the epiline corresponding to x_1, then we should reject the match. So we will define an error function that will measure the sum of distance between x_1 and the first epiline and the distance between x_2 and the second epiline. We can use well-known formulas for the distance between a point and a line to do this computation. Our error function can then be written as

$$e(x_1, x_2) = \frac{|a_1u_1 + b_1v_1 + c_1|}{\sqrt{a_1^2 + b_1^2}} + \frac{|a_2u_2 + b_2v_2 + c_2|}{\sqrt{a_2^2 + b_2^2}}$$

where $(a_1, b_1, c_1)^T = E^T x_2$, $(a_2, b_2, c_2)^T = Ex_1$ and $x_i = (u_i, v_i, 1)$. Finally, to reject outliers, we will loop through the set of matches $\{(x_1^{(i)}, x_2^{(i)})\}_{i=1}^n$, and keep a match $(x_1^{(j)}, x_2^{(j)})$ if $e(x_1^{(j)}, x_2^{(j)}) < \delta$, for some threshold δ that we pick. Note that the units of δ are pixels, and it is the sum of the distances between each point to its respective epiline.

7.3.1 Task 2

Implement the functions `FilterByEpipolarConstraints` and `epipolar_error` in `epipolar.py`. To do this you will need to:

1. Enforce epipolar constrains on the image by computing the essential matrix, normalizing the image coordinates, and computing the epilines for the matches. (*Hint*: Go back to Lab 6 for a reminder on what the Camera Intrinsic Matrix contains.)

2. Calculate the distances between epilines and their corresponding points to retrieve the epipolar error.

3. Tune the threshold `epipolar_threshold` being passed into the constructor for `Stereo_Pointcloud`. This is the threshold being used by your epipolar constraint filtering function. Recall that this threshold is a distance in units of pixels. This will be `0.07` by default. It gets passed in in the `__main__` section of `stereo_pointcloud.py` (at the bottom of the file). You should tune it until you get about 20-250 total matches per frame.

Once you have completed this, run the visualization again. This time, the result should be a stream of images with visualizations of all matched features as the top pair of images and only matches that satisfy epipolar constraints marked in the bottom pair. Take note of the number of filtered matches you are getting on average per image. This information is printed by the node `stereo_pointcloud` as "Total matches". You should tune the threshold for epipolar filtering until you get a decent number of matches (around 20-250).

Note that you may get a large variability in number of good feature matches per image along the trajectory (not all frames will be feature rich), so you should sample a good chunk of the trajectory before deciding on a threshold.

Run the main node using:

```
rosrun stereo_pointcloud stereo_pointcloud.py
```

and visualize the feature matches using:

```
rosrun image_view image_view image:=/drone/matches
```

Tip: Make sure the bag file is running.

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Explain the different topics being published by the bag file.
- Show the moving body frame of the drone relative to the world frame in Rviz.
- Display a stream of distorted images against their un-distorted images in Rviz.
- Display stream of matched features before and after enforcing epipolar constraints using `image_view`. Was your node successful at rejecting bad matches?

---

8 Triangulation

After matching features and rejecting outlier matches, we are left with a set of corresponding points between the left and right image. For each such pair (x_1, x_2), we wish to find the coordinates of the point p they represent in 3D space in the robot's body frame. We have the transform (R, T) between the two camera frames. Let X_1 and X_2 be the 3D coordinates of the point p in the reference frames of cameras 1 and 2 respectively. We will once again assume that the image coordinates (x_1, x_2) are given in normalized form, so that there exist positive depth scalars λ_1, λ_2 such that $\lambda_1 x_1 = X_1$ and $\lambda_2 x_2 = X_2$. Now we can write

$$\lambda_2 x_2 = \lambda_1 R x_1 + T \qquad (3)$$

where the only unknowns are λ_1 and λ_2. We can then recast this into a matrix equation

$$\begin{bmatrix} -Rx_1 & x_2 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = T \tag{4}$$

Now if we define $A = [-Rx_1 \ x_1] \in \mathbb{R}^{3\times 2}$, $\lambda = (\lambda_1, \lambda_2)^T \in \mathbb{R}^2$, then the above can be expressed as solving for λ in the equation $A\lambda = T$.

However, there are two additional considerations when we deal with real data. Firstly, our measurements for x_1 and x_2 are not perfect, so in general equation (4) may have no exact solutions. Hence, we should find the least squares solution. i.e. we should solve

$$\min_{\lambda \in \mathbb{R}^2} \|A\lambda - T\|_2^2 \tag{5}$$

which has the closed form solution $\lambda^* = (A^\top A)^{-1} A^\top T$. Secondly, it may occasionally be the case that our image measurements are noisy enough that the solution to equation (5) gives us negative values for λ_1 and λ_2. In this case, we should simply throw away that point, as we do not have enough information to locate it.

After solving the above, we will end up with two depths λ_1 and λ_2, which will give us two possible values for the coordinates of p in camera frame 2. These are

$$\begin{aligned} p_2' &= \lambda_2 x_2 \\ p_2'' &= \lambda_1 R x_1 + T \end{aligned}$$

We should pick the average of these solutions:

$$p_2 = (p_2' + p_2'')/2 \tag{6}$$

This will give us our estimate of p in the second camera frame. Finally, we can find p by transforming p_2 to be in the robot's body frame instead.

8.0.1 Task 3

Complete the function `least_squares_triangulate` in `stereo_pointcloud.py` so that we publish a pointcloud constructed from matched points in the two images. This function should return your estimate of the input point's 3D location in the reference frame of the right camera. This will involve

1. Finding the least squares solution to 4 for each set of matched image points. You will need to discard solutions where λ_1 and λ_2 is negative.
2. Using the least squares solution to find a 3D point for each image, and taking the average of the point, as shown in equation (6).

Once you are done implementing this function, you can test the whole pipeline by running the main node.

```
rosrun stereo_pointcloud stereo_pointcloud.py
```

You will now be able to visualize the 3D points you found in Rviz. You can do this by using Add > By Display Type > PointCloud2 and then changing its topic to `/drone/pointcloud`. Make sure that you have Global Options > Fixed Frame set to `world`. Finally, set the Decay Time parameter of the PointCloud2 display to some big number greater than 100. By default, Rviz will only display the last published message to a specified topic, whereas we want our published points to persist in order to create a visualization of the environment as the drone flies. We can fix this by changing the "Decay Time" parameter of the display. It is 0 by default, so the pointlcoud messages get cleared as soon as a new message is published. Setting this value to some positive x will cause points to persist for x seconds before being erased. The bag file plays for about 100 seconds so some value $x \geq 100$ will suffice.

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Triangulate corresponding world points for a set of matched image points.
- Display point cloud in Rviz generated from corresponding image points.
- Comment on the quality of the pointcloud. Is it noisy? Are there a lot of stray points?
- Answer the following question: Even though the floor is visible in most of our images, it is not included in the pointcloud at all. Why is this the case?