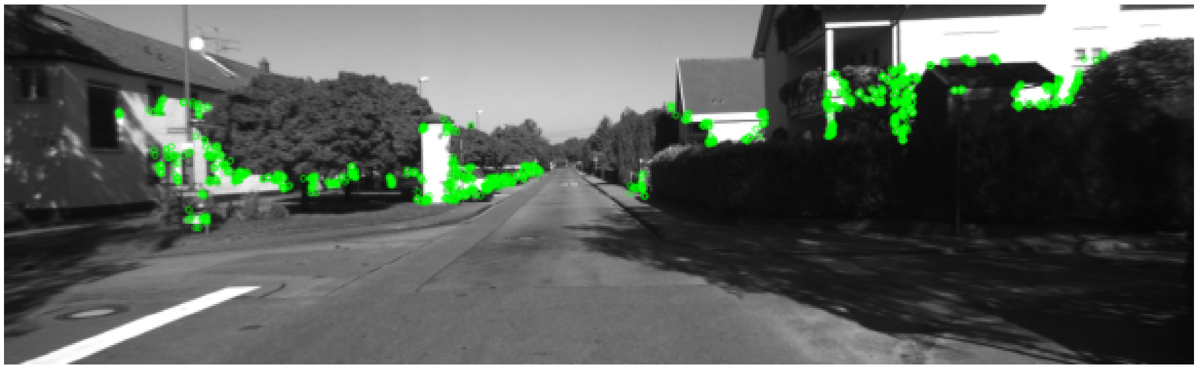
Ehud Gordon 203861422

NAV Stage 1

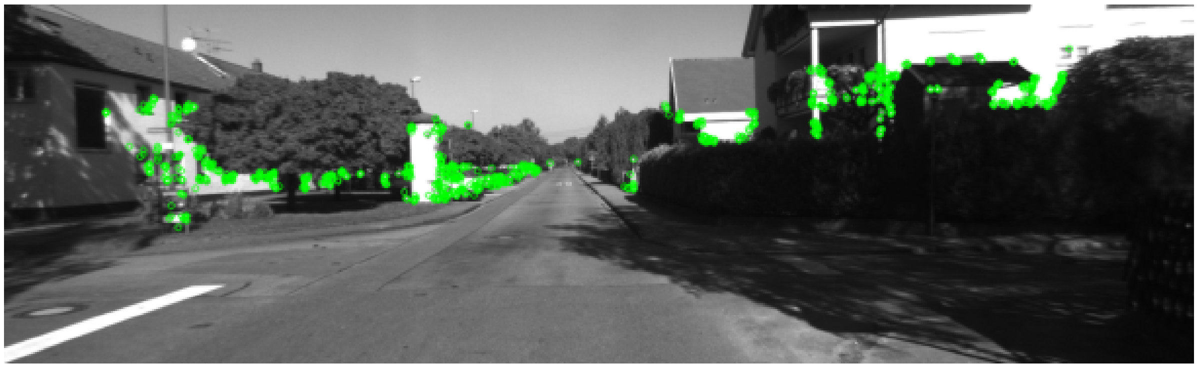
https://github.com/ehud-gordon/VAN_ex

Q1.1

Feature2D.ORB_keypoints



Feature2D.ORB_keypoints



I've used ORB (which uses FAST detector and BRIEF descriptor), because it's multi-scale, rotation/orientation invariant, fast, and simple to use,

Q1.2

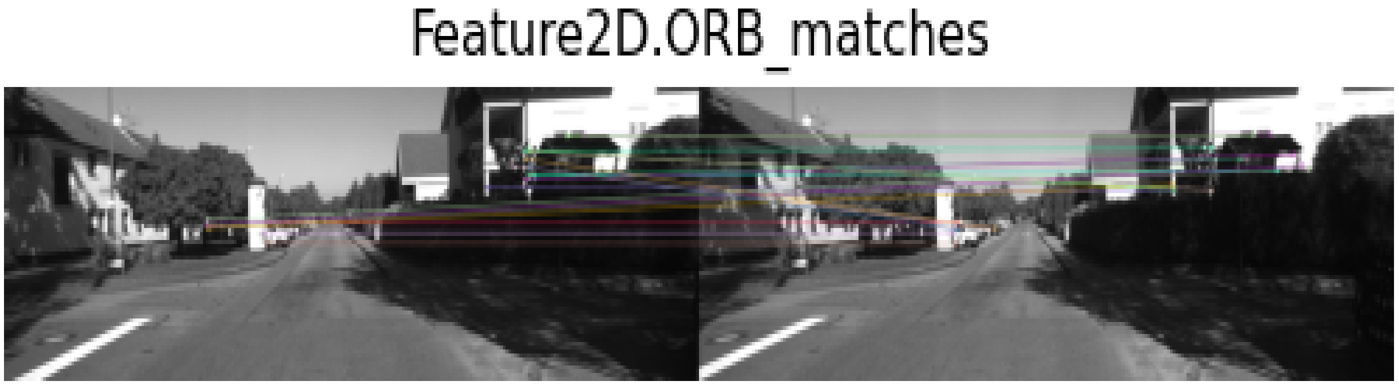
I've used ORB (which uses FAST detector and BRIEF descriptor). Here are the features of the two best matches - as you can see, they're similar.

```
---- desc 0 img 0 -----
[ 41 240 244  97 109  31 240 216 127 249 246  40 203 157 222  61 143 156
 255  93 142 247 235 144 223 217  49 246  71 141  82 215]
---- desc 0 img 1-----
[ 41 240 244  97 109  31 240 216 127 249 246  40 203 157 222  61 143 156
 255  93 142 247 235 144 223 217  49 246  71 141  82 215]

---- desc 1 img 0-----
[210 116 151 161 195  38  60  79  57 179 130  82  12  26  64 219 227 215
 123 177  49 223 145 194 189 240 222  70 173  81 169   2]
```

```
---- desc 1 img 1 -----
[210 116 151 161 195  38  60  79  57 179 130  82  12  26  64 219 227 215
 123 177  49 223 145 194 189 240 222  70 173  81 169   2]
```

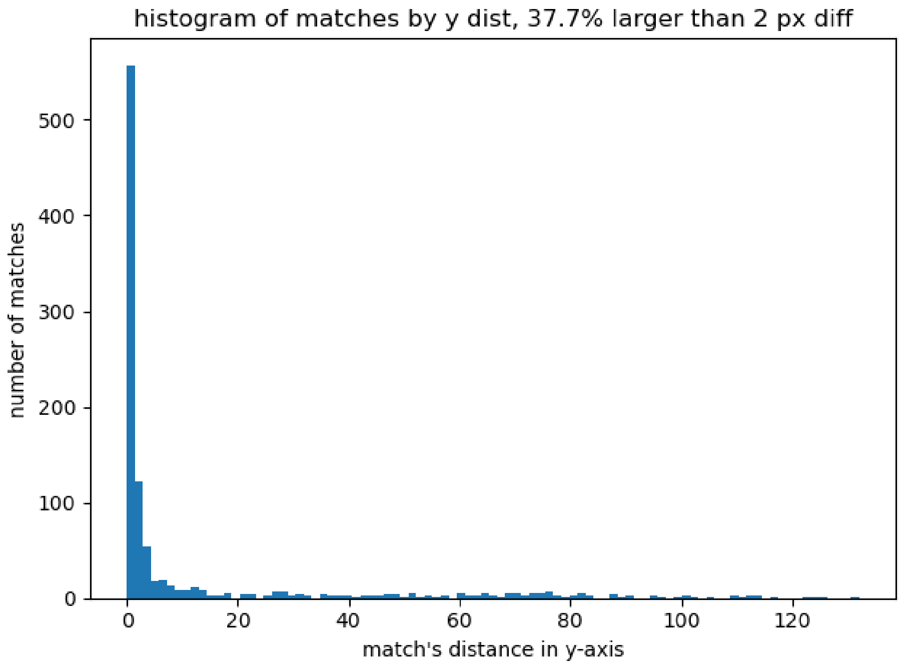
Q1.3



I've used BruteForce matcher, because I'm not in a hurry, and can afford the extra computation in order to get the best results. From opencv documentation: "Brute-Force takes the descriptor of one feature in first set and matches with all other features in second set using some distance calculation, and the closest one is returned."

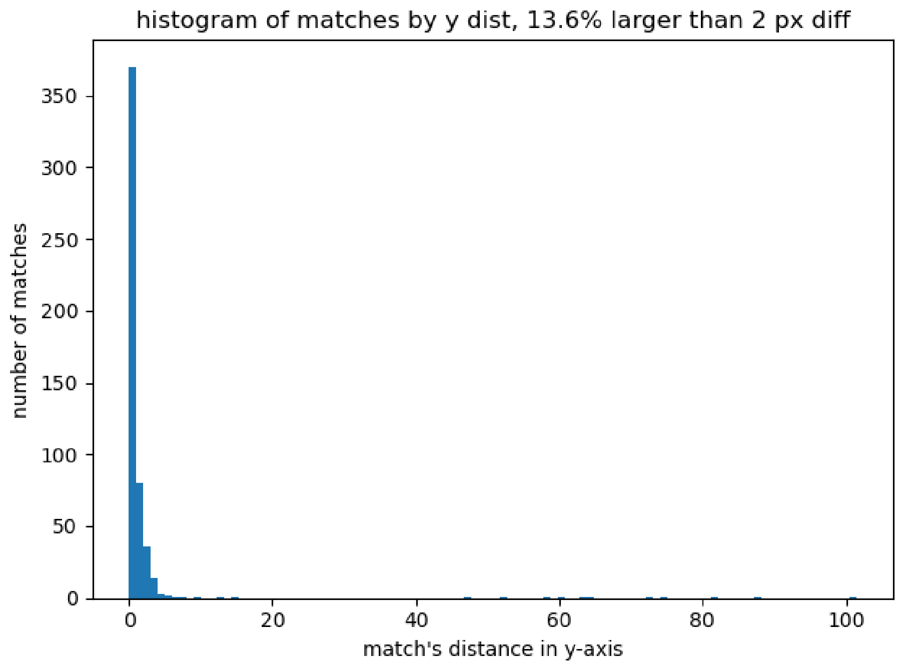
Q1.4

All of the correct matches will be a straight line. this is because the rectified stereo images are rectified as if the two cameras were on the same plane when the images were taken (and also at the same time).



percentage of matches larger than 2 px diff: 37.7%

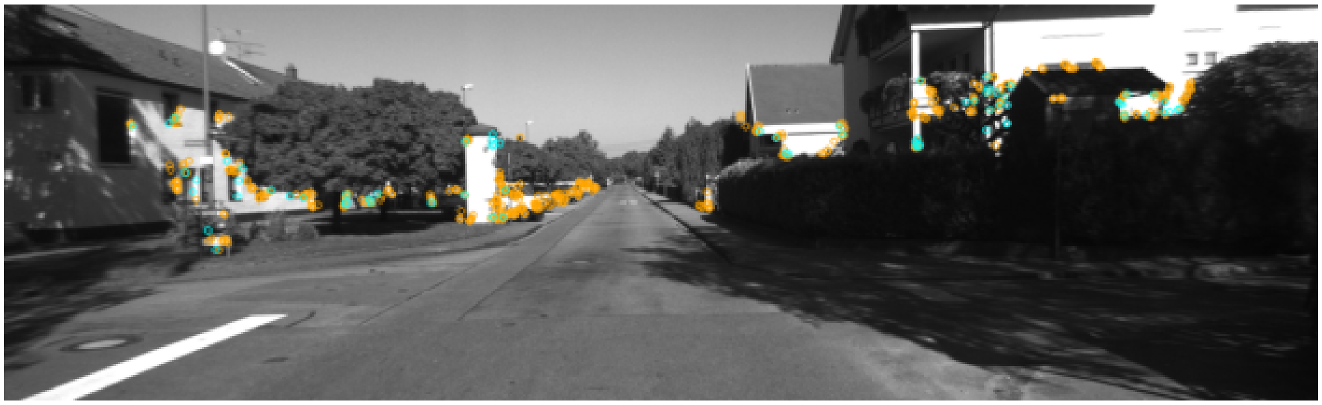
Q1.5



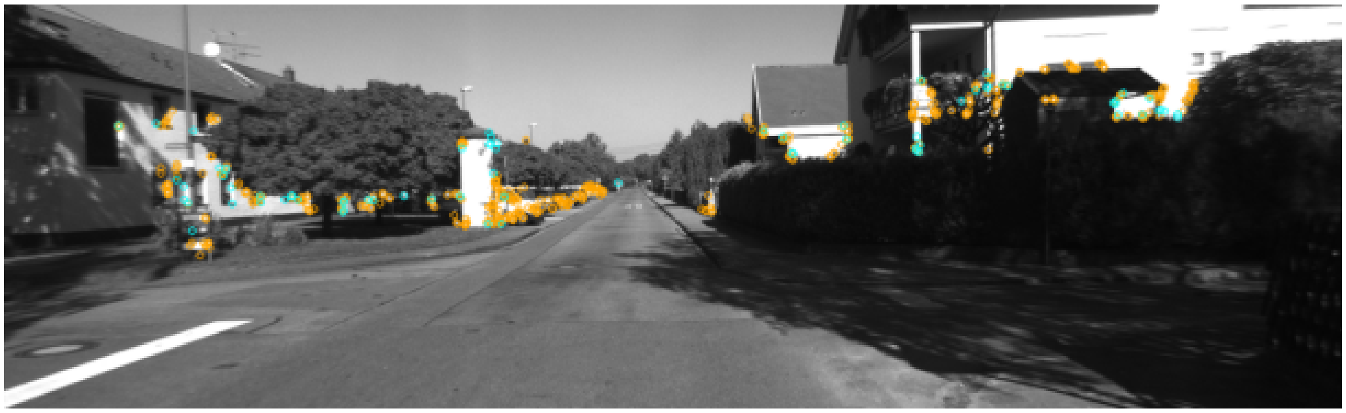
percentage of matches larger than 2 px diff: 13.6%

Q1.6

left image outliers + inliers

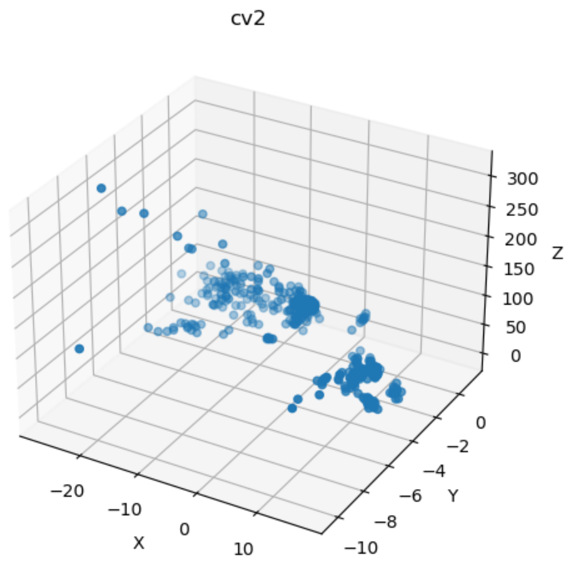


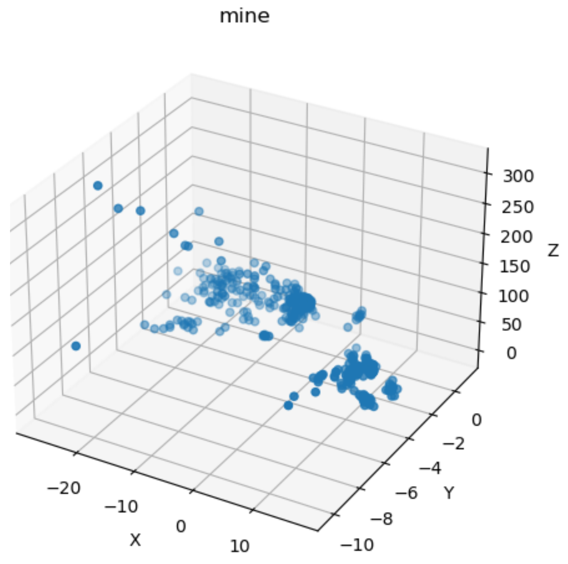
right image outliers(CYAN) + inliers(ORANGE)



Using the stereo rejection policy, erroneous matches will be only due to keypoints on the same line that have similar descriptors. Thus the rate of erroneous matches will decrease. With the assumption that the Y-coordinate of the matches is distributed uniformly, let x denote the probability of making a mistake in any row, then the expected number of erroneous matches will be: height * x.

Q1.7





They're the same. If you look at my code, I've written a function `vis_triangulation()` that prints circles on the original image, corresponding to matched points, along with their triangulated 3D location. I get the same results for both cv2 and my triangulation. In fact, the iterative part of LS wasn't really necessary.

Q1.8

I've noticed some incorrect locations. Sometimes, at the edges of trees, it's hard to tell whether it refers to the point on the top of the tree, or to the house behind it (which could be much farther away). Since we know our projection matrix is very accurate, then we can assume that incorrect location is due to incorrect matches.

I've several ideas on how to eliminate incorrect locations:

1) There must be a way to reconstruct the original camera matrix from these 3D points and the matches. For any group of points, we can use something like RANSAC to estimate camera matrix, and then compare it to the true camera matrix - and thuse remove outliers.
2) Some points lie on a plane (a building side for example) that is parallel to us, and thus should have the same z-axis. When one of these points have a vastly different z-axis than the others, it's probable to say that this is an incorrect location.