# 3D Perception and Sensor Fusion – Lab 1: Stereo Reconstruction

*Ryan Crowell & Enis Hidri*

*3D Perception and Sensor Fusion*

## Objective

The objective of this lab is to understand and implement the stereo reconstruction equations and usage with two cameras. A stereo vision system allows for the recovery of 3D information from 2D images by exploiting geometric constraints between a calibrated stereo pair. This lab walks through all stages of stereo vision:

1. Monocular calibration
2. Stereo calibration
3. Rectification
4. Sparse reconstruction
5. Dense reconstruction
6. Comparison between block matching and semi global matching.

## 1. Monocular Calibration

The first step involves calibrating each individual camera to recover intrinsic parameters and distortion coefficients. This is done by observing a known calibration pattern (chessboard) from multiple perspectives. With the known distances between points and the distance between the two cameras the calibration process can easily estimate the camera's intrinsic parameter matrix (K) and distortion coefficients using OpenCV's cv2.calibrateCamera() function.

```
# Compute the camera parameters given all the samples collected
print('- Found pattern in %d/%d images' % (len(images_used), len(files)))
print('- Calibrating...')
error, K, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points, img_points, (w, h), None, None)
```

*Figure 1: cv2 library calibration function*

To run the code the following  has to be typed into the terminal with the usage of the ArgumentParser class within the *argparse* library. These parameters are defined and set up within the desired executed function.

```
## Calibation monocular (intrinsic)
python 3dpsf_lab_1_stereo/monocular_calibration.py \
--row_corners 8 \
--col_corners 12 \
--squares_size 0.02 \
--images_dir ./sample_data/calibration/left/ \
--out_calib ./sample_data/left_calib.yaml
```

```
python ./monocular_calibration.py \
--row_corners 8 \
--col_corners 12 \
--squares_size 0.02 \
--images_dir ./sample_data/calibration/right/ \
--out_calib ./sample_data/right_calib.yaml
```

*Figure 2: Terminal input using arparse library parameters for left and right cameras*

After running this line for each camera a .yaml file that contains all of the intrinsic parameters of the camera along with the distortion coefficients and the mean reprojection error. These different bits of data describe how the camera converts 3D points on a 2D image plane with the error associated along with how light gets distorted through the lens.

```yaml
%YAML:1.0
---
camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1007.4544869794573, 0., 942.05085125440689, 0.,
       1005.604205583193, 560.50371017508269, 0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
   rows: 1
   cols: 5
   dt: d
   data: [ -0.12575445402717594, 0.084823757684138495,
       0.00432355646537462, -0.00054770853928695, -0.039309763357041907 ]
mean_error: 0.28506118965290589
```

*Figure 3: .yaml file with cameras intrinsics, distortions, and error for left camera*

right_calib.yaml

```yaml
%YAML:1.0
---
camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1059.9103413671926, 0., 983.0895804946615, 0.,
       1059.9346091034308, 554.03892306895182, 0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
   rows: 1
   cols: 5
   dt: d
   data: [ -0.10903015198797893, 0.12913704392588662,
       -0.00044556503623092216, 0.0031641419837797661,
       -0.068606211912031645 ]
mean_error: 0.34886972322754217
```

*Figure 4: .yaml file with cameras intrinsics, distortions, and error for right camera*

The mean error we see here is within a good range of what we should expect from a well calibrated camera. When running the debug parameter the chessboard is detected properly, this is known because the detected chessboard images show points at each inside corner and sharing colors with all other corners in the same column with a diagonal line leading to the next column in sequence. The detected versus reprojected images correctly place two different crosses one for each at first glance they look as if they overlap correctly but on a few they are off by a few pixels (Figure 6). Observe the figures 5-7 for the left camera and 8-10 for the right camera:

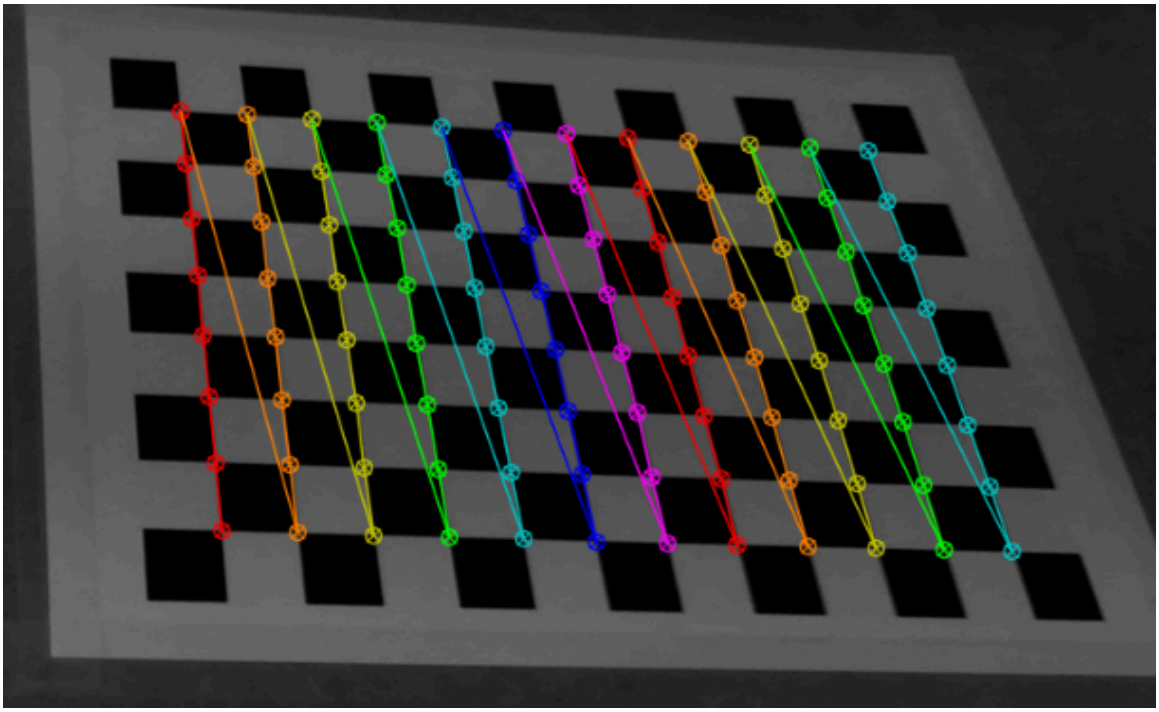


*Figure 5: Chessboard detection left camera*

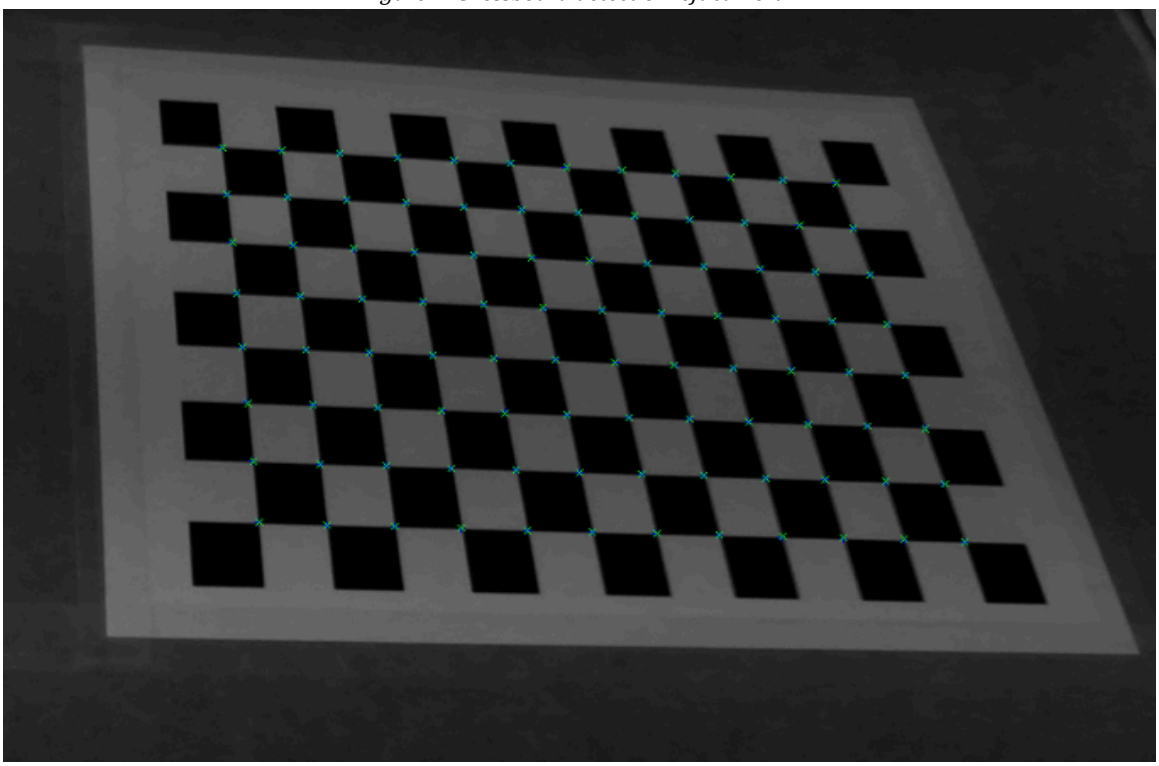


*Figure 6: Chessboard detection vs. Reprojection left camera*

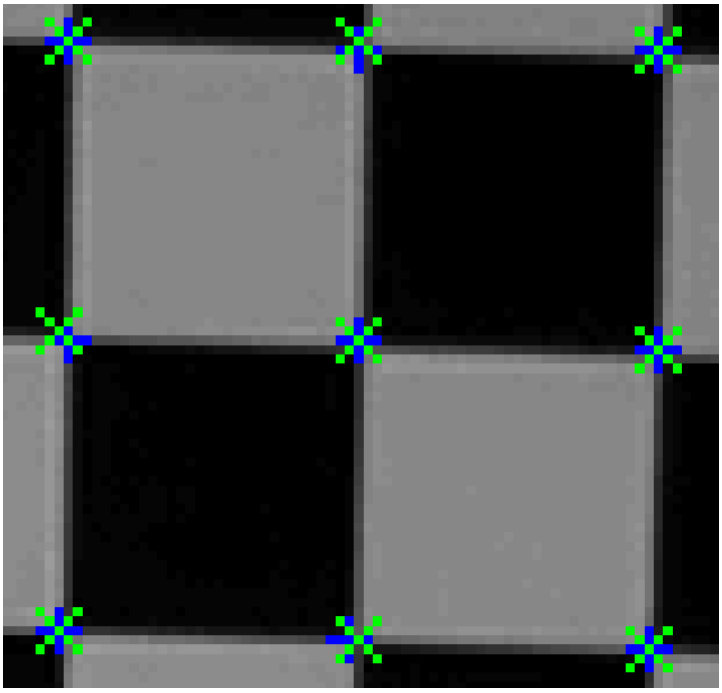


*Figure 7: Zoomed in photo of Figure 6 left camera*

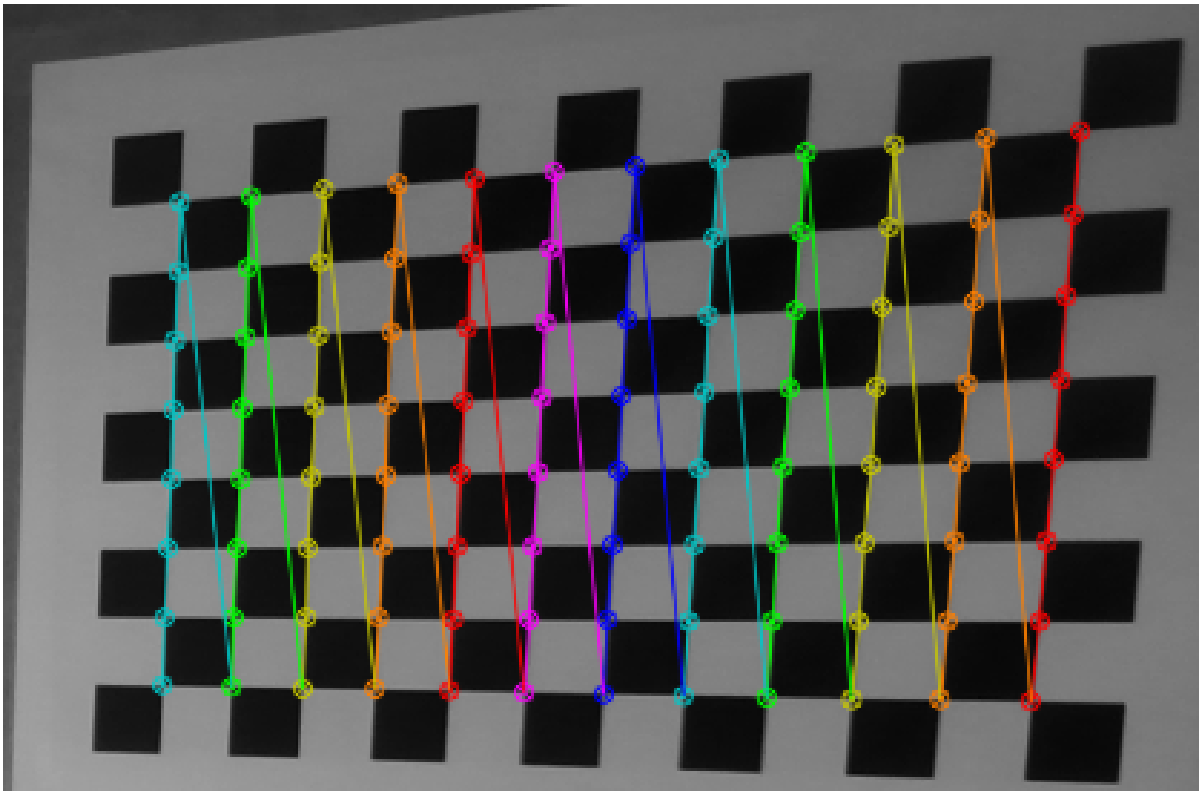


*Figure 8: Chessboard detection right camera*

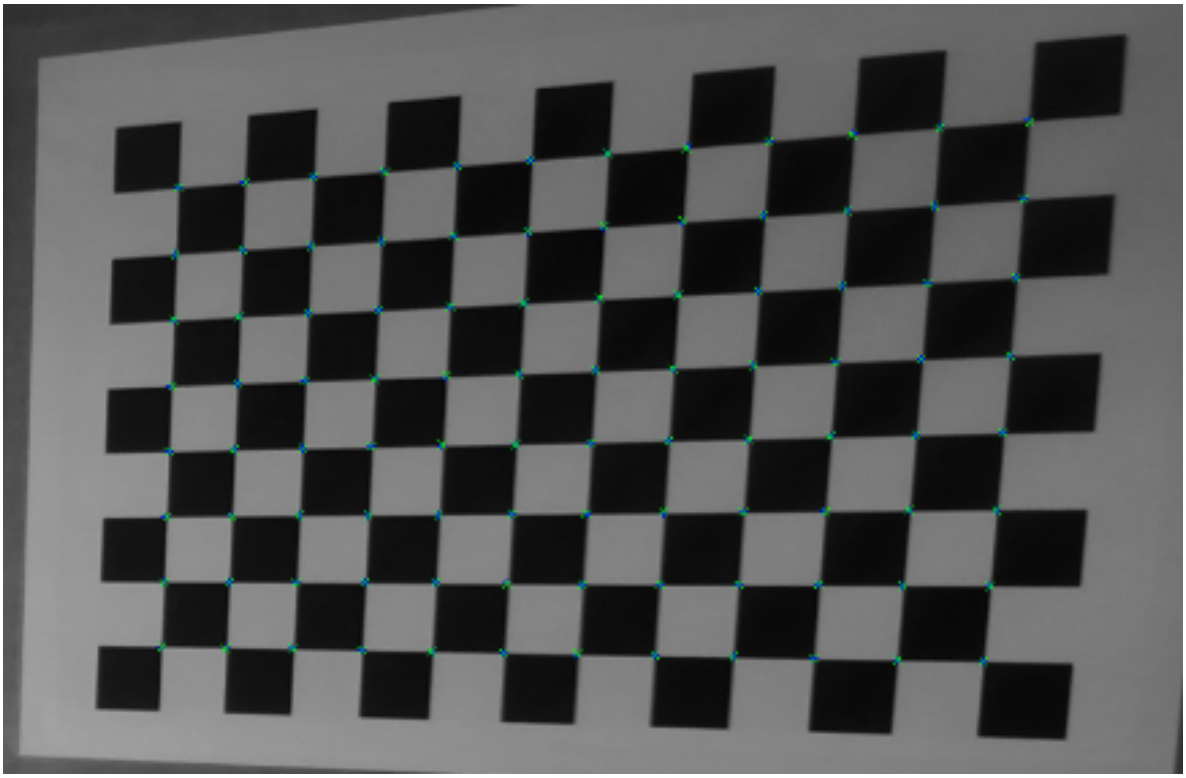


*Figure 9: Chessboard detection vs. Reprojection right camera*

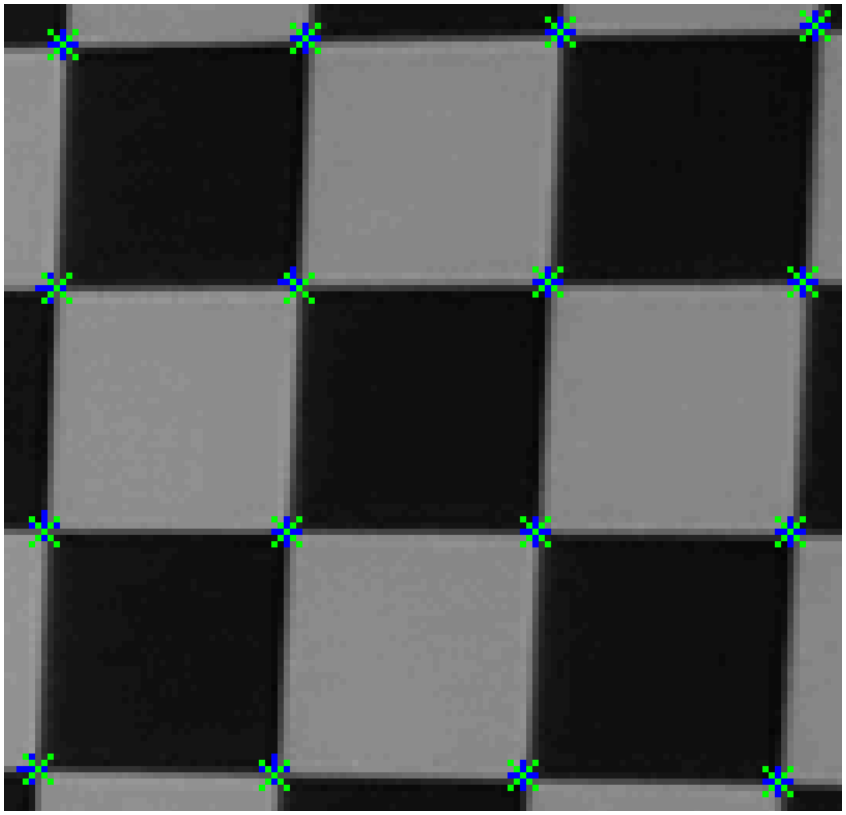


*Figure 10: Zoomed in photo of Figure 9 right camera*

## Exercise 1

As observed in Figure 6 the blue and green crosses don't all exactly align which indicates an imperfect calibration. However for this lab it is within reasonable to perform reconstruction. The mean reprojection error is less than 0.4 for the two different cameras which as described before is great calibration.

## 2. Stereo Calibration

After calibrating the individual cameras, the next step is to determine the extrinsic parameters – the rotation (R) and translation (T) between both cameras. This process is performed using the cv2.stereoCalibrate() function. It is run with the two individual .yaml files found in the previous exercise and is executed by the command prompt found below.

```
16  ## Calibration stereo (extrinsic)
17  python ./stereo_calibration.py \
18  --row_corners 8 \
19  --col_corners 12 \
20  --squares_size 0.02 \
21  --left_images_dir ./sample_data/calibration/stereo/left \
22  --right_images_dir ./sample_data/calibration/stereo/right \
23  --left_calib_file ./sample_data/left_calib.yaml \
24  --right_calib_file ./sample_data/right_calib.yaml \
25  --out_file ./sample_data/stereo_calib.yaml
```

*Figure 11: Command prompt for stereo calibration*

These parameters describe the relative pose of one camera with respect to the other, allowing for 3D triangulation of matched points. For the debug photos we can tell that it is working because each column on the board refers to a new color and the points are correctly placed for the photos taken by each camera.

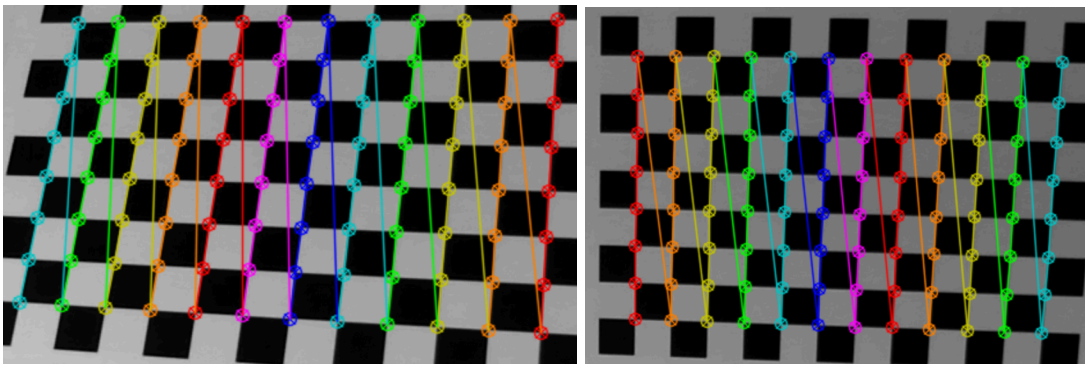
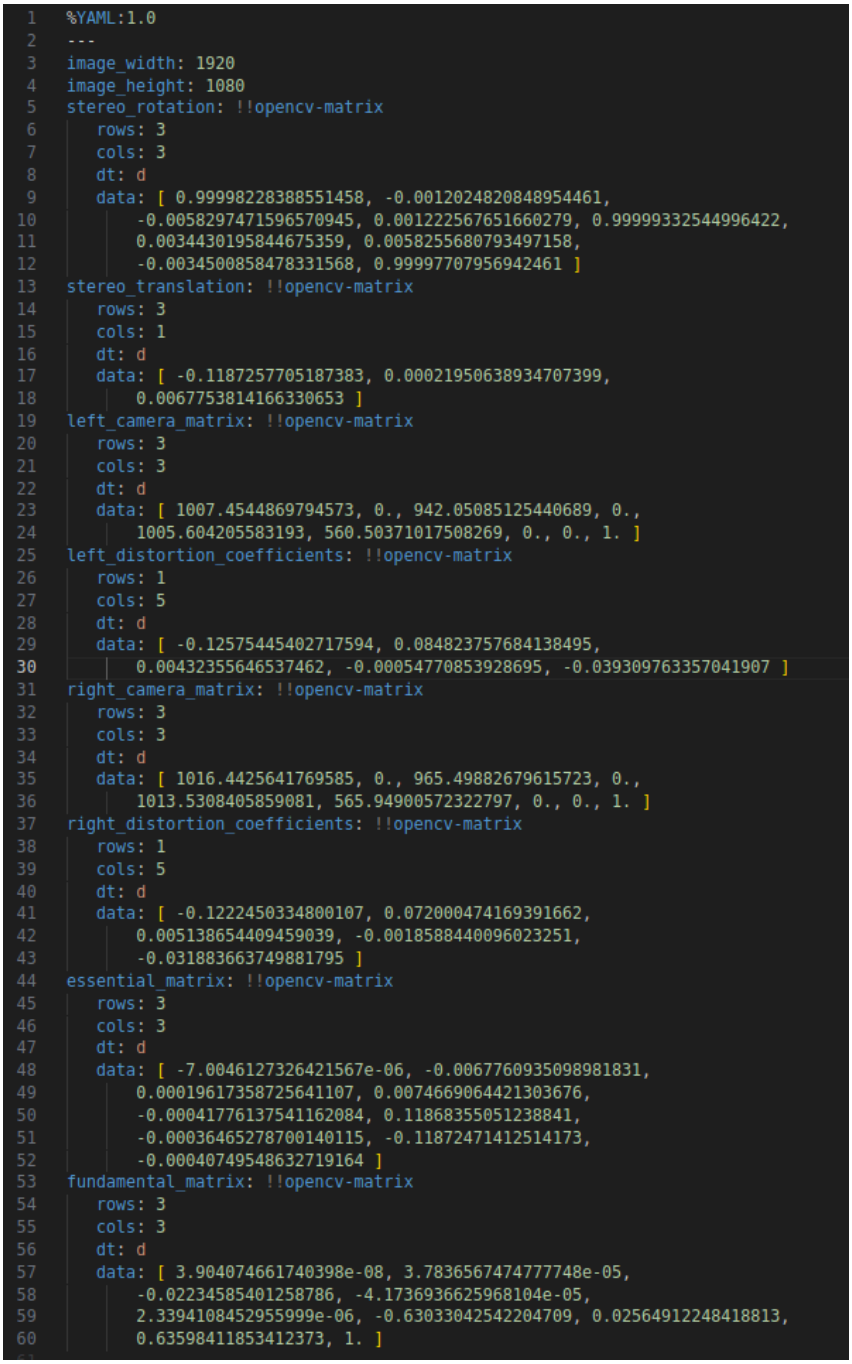


*Figure 12: Left and Right Debug Stereo Calibration*

The function outputs a .yaml file similar to the one seen in the monocular calibration with some added matrices that assist in describing a camera with respect to the other. We can also see the reprojection error here and it is again less than 0.4 signifying a good calibration.



```yaml
%YAML:1.0
---
image_width: 1920
image_height: 1080
stereo_rotation: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 0.99998228388551458, -0.0012024820848954461,
       -0.0058297471596570945, 0.0012225676516602 79, 0.99999332544996422,
       0.0034430195844675359, 0.0058255680793497158,
       -0.0034500858478331568, 0.99997707956942461 ]
stereo_translation: !!opencv-matrix
   rows: 3
   cols: 1
   dt: d
   data: [ -0.1187257705187383, 0.00021950638934707399,
       0.0067753814166330653 ]
left_camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1007.4544869794573, 0., 942.05085125440689, 0.,
       1005.604205583193, 560.50371017508269, 0., 0., 1. ]
left_distortion_coefficients: !!opencv-matrix
   rows: 1
   cols: 5
   dt: d
   data: [ -0.12575445402717594, 0.084823757684138495,
       0.00432355646537462, -0.00054770853928695, -0.039309763357041907 ]
right_camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1016.4425641769585, 0., 965.49882679615723, 0.,
       1013.5308405859081, 565.94900572322797, 0., 0., 1. ]
right_distortion_coefficients: !!opencv-matrix
   rows: 1
   cols: 5
   dt: d
   data: [ -0.1222450334800107, 0.072000474169391662,
       0.0051386544094590 39, -0.0018588440096023251,
       -0.031883663749881795 ]
essential_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ -7.0046127326421567e-06, -0.0067760935098981831,
       0.00019617358725641107, 0.0074669064421303676,
       -0.00041776137541162084, 0.11868355051238841,
       -0.00036465278700140115, -0.11872471412514173,
       -0.00040749548632719164 ]
fundamental_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 3.9040746617403 98e-08, 3.7836567474777748e-05,
       -0.02234585401258786, -4.1736936625968104e-05,
       2.3394108452955999e-06, -0.63033042542204709, 0.02564912248418813,
       0.63598411853412373, 1. ]
```

*Figure 7: Stereo .yaml file*

## Exercise 2

Even though this also outputs all the info described with the monocular calibration it is better to run the previous one first to provide a stable starting point for the stereo calibrate such that it has fewer unknowns leading to a more accurate and stable result of the extrinsic parameters.

## 4. Rectification

Rectification simplifies the matching process by aligning epipolar lines horizontally. This process creates new virtual camera parameters where corresponding points in both images lie on the same image row (same Y-coordinate). The rectification is computed using cv2.stereoRectify().

```
R1, R2, P1, P2, Q, roi1, roi2 = cv2.stereoRectify(stereo_calib["left_camera_matrix"],
                                                  stereo_calib["left_distortion_coefficients"],
                                                  stereo_calib["right_camera_matrix"],
                                                  stereo_calib["right_distortion_coefficients"], im_size,
                                                  stereo_calib["stereo_rotation"],
                                                  stereo_calib["stereo_translation"], alpha=param.alpha)
left_maps = cv2.initUndistortRectifyMap(stereo_calib["left_camera_matrix"],
                                        stereo_calib["left_distortion_coefficients"],
                                        R1, P1, im_size, cv2.CV_16SC2)
right_maps = cv2.initUndistortRectifyMap(stereo_calib["right_camera_matrix"],
                                         stereo_calib["right_distortion_coefficients"],
                                         R2, P2, im_size, cv2.CV_16SC2)
```

*Figure 8: cv2 stereo rectification and undistort functions*

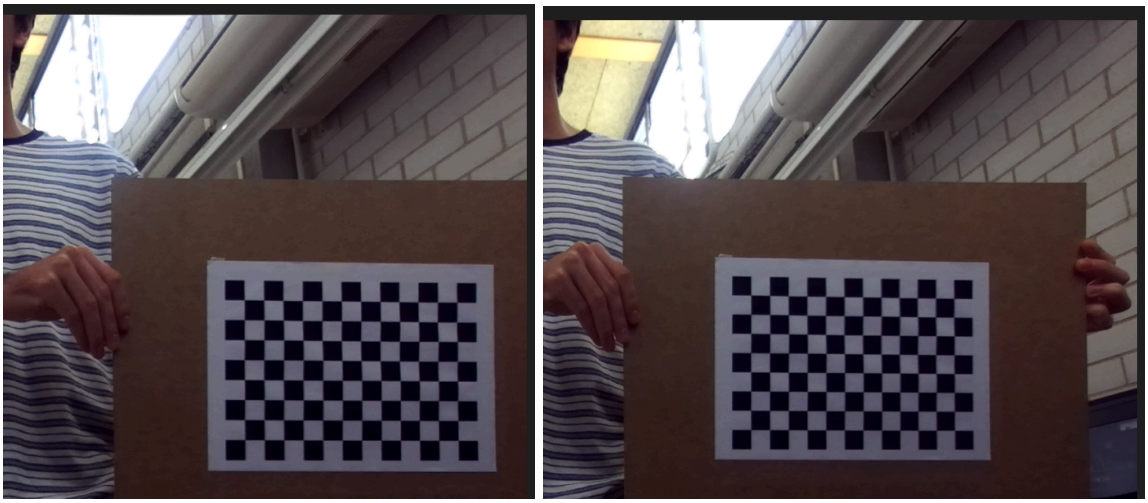


Figure 9: Alpha 0 rectified left and right images

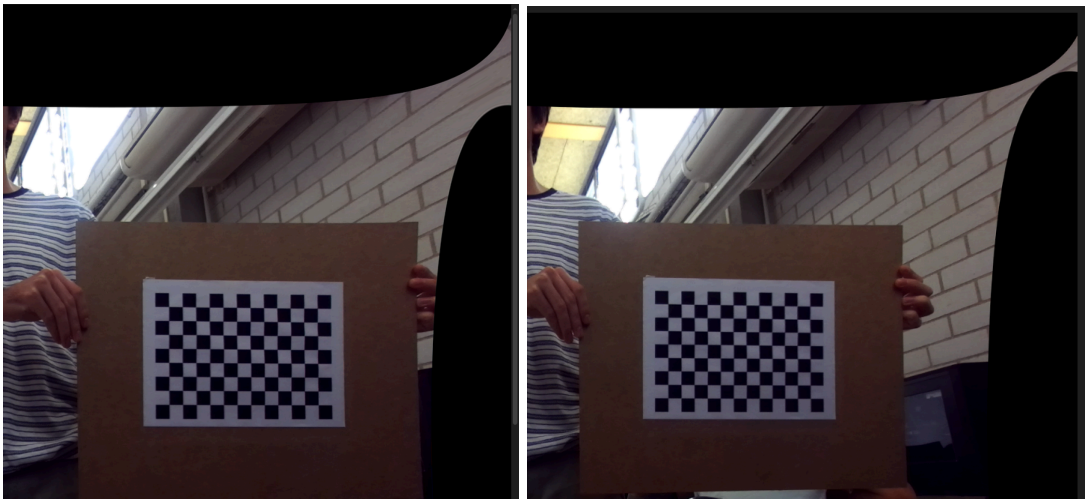


*Figure 10: Alpha 1 rectified left and right images*

## Exercise 3

The alpha parameter controls how much of the original image you want preserved. When rectifying the images the function warps the images such that epipolar lines become horizontal which helps with matching features across the two different images. Because of the warping as well as the distortion of the lens some areas of the image get compressed or extended causing some invalid pixels. Alpha helps with determine how much of the original image you want, alpha 1 with leave all of the original image in view and have no data for the other pixels needed to make it square while alpha 0 will zoom into the photo until only valid pixels remain but removing some valid pixels as well (such as the pixels in the top right of the images in Figure 10).

## 4. Sparse Reconstruction and Exercise 4

Using the rectified stereo images, sparse feature-based reconstruction can be performed. This step involves detecting feature points, matching them between images, and triangulating their 3D positions using cv2.triangulatePoints(). SIFT is used as the default feature detector and descriptor.

Below in Figure 11 the output for the debug of the sparse matching is displayed showing first the left and right images with the key features shown, second the matches between those key features in the reconstructed image and lastly the 3D location of all the matched features using a heatmap to display points in and out of the page.

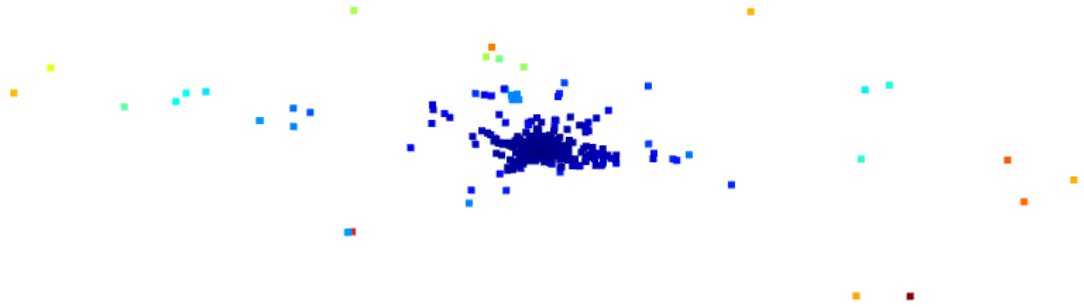
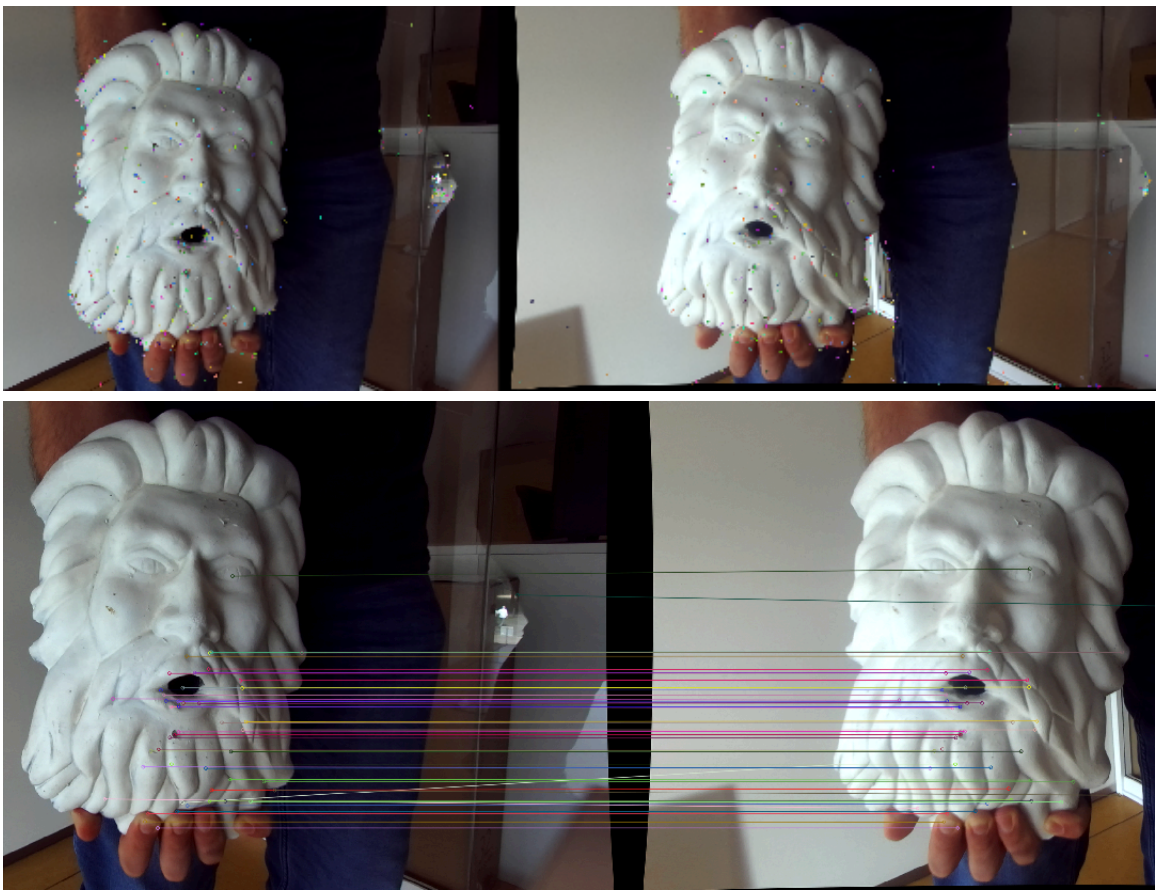


*Figure 11: Sparse matching Debug for boxes*



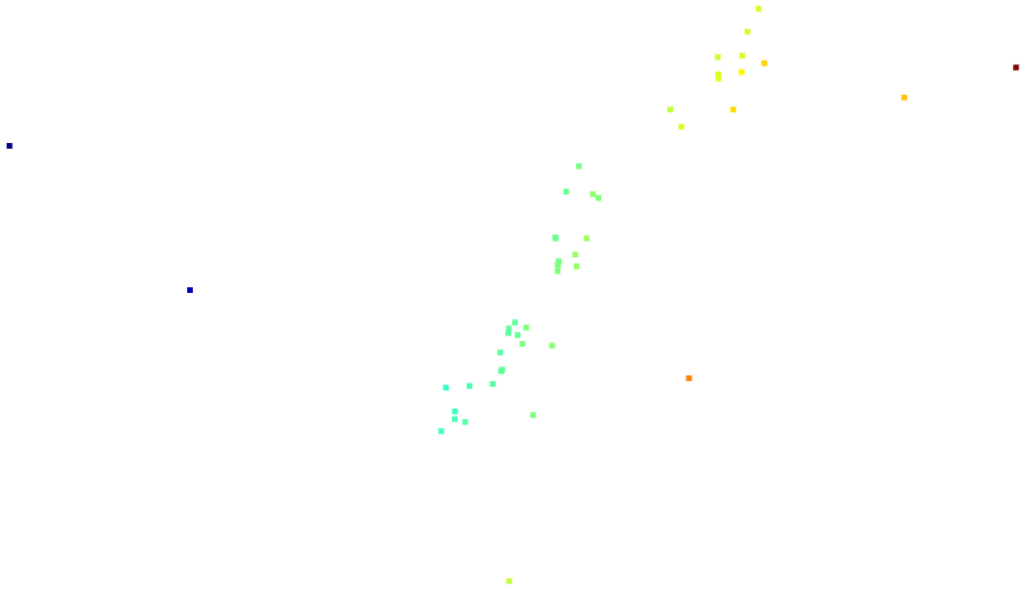


*Figure 12: Sparse matching Debug for statue*

In this figure the important features are the ones in blue that represent matches found in the two images, the points that are further spread out and straying toward the color red are outliers in the matches as most pixels in the image are relatively close to one another in the 3D space. This could be due to unsatisfactory matches because the function is not taking advantage of the known camera geometry. Between the two data sets we can notice much more matches with the boxes this is due to the statue having a lot of smooth features where the boxes have hard edges and corners.

## 5. Epipolar Constraint Filtering

In the previous matching step, some matches may not satisfy the epipolar constraint. This exercise filters out matches that do not have approximately the same Y-coordinate in both images, ensuring they lie along the same epipolar line. The code added for exercises 6 below adds this constraint utilizing the camera's known geometry.

```
### Ex. 5: Use the epipolar constraint on rectified images to further constrain the matches

if param.rect_stereo_calib_file:

    good_matches = []
    vertical_threshold = 6.0  # pixels

    for m in matches:
        ptL = kpL[m.queryIdx].pt
        ptR = kpR[m.trainIdx].pt

        if abs(ptL[1] - ptR[1]) < vertical_threshold:
            good_matches.append(m)

    matches = good_matches
###
```

*Figure 13: Code for applying epipolar constraint*

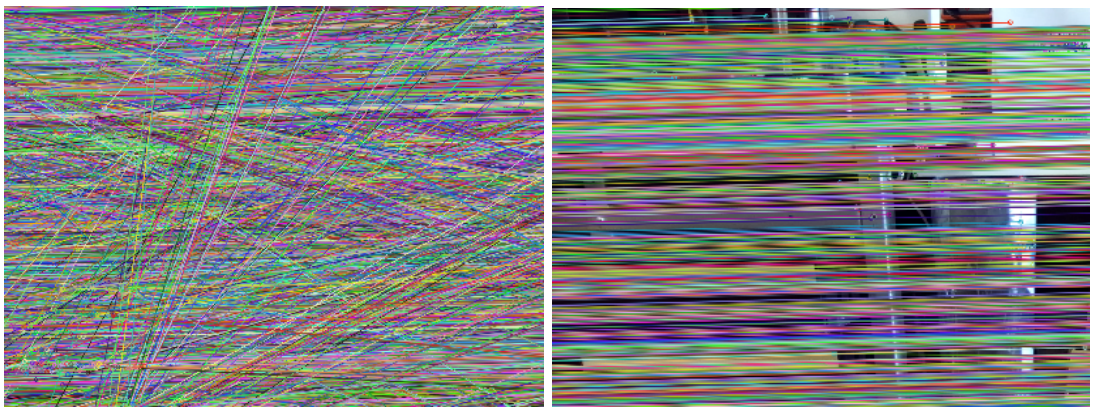


*Figure 14: Results from using epipolar constraints*

As it can be seen from the two images of the results the outlier rejection and restriction of points now shows only straight horizontal lines to matches between points which leads to more correct matches. Th

## 6. Exercise 6 (Feature Detector Comparison)

Alternative feature detectors/descriptors such as ORB, AKAZE, or BRISK can be tested to compare their performance against SIFT in terms of computation time and accuracy. To do this simply change the line “sift = cv2.SIFT_create()” to “sift = cv2.ORB_create()” this uses the detector ORB to find matches between the points and uses binary encoding! Although some other parameters can be set with orb it works just fine with the default by just changing that line.

*Figure 15: Matches using ORB*

It is obvious through visual inspection that the ORB descriptor finds fewer matches than SIFT. ORB seemed to run much faster however. Even still the lack of features is a major downside of ORB. There is no wonder why SIFT is the typical descriptor used and preferred in many applications.

## 7. Dense Reconstruction

Dense reconstruction aims to estimate depth for all image pixels using stereo matching algorithms. The Block Matching (BM) algorithm computes disparities for each pixel based on local intensity differences. Parameters such as numDisparities and blockSize influence accuracy and smoothness of the disparity map. Below find the block matching function used

from the cv2 library giving the results found in Figure 16 and 17.

```
73      stereo = cv2.StereoBM_create(numDisparities=num_disparities,
74                                   blockSize=block_size)
75      stereo.setMinDisparity(min_disparity)
76      stereo.setPreFilterType(prefilter_type)
77      stereo.setPreFilterSize(prefilter_size)
78      stereo.setPreFilterCap(prefilter_cap)
79      stereo.setDisp12MaxDiff(disp12_max_diff)
80      stereo.setTextureThreshold(texture_threshold)
81      stereo.setUniquenessRatio(uniqueness_ratio)
82      stereo.setSpeckleWindowSize(speckle_size)
83      stereo.setSpeckleRange(speckle_range)
```

*Figure 16: Function for Block Matching Algorithm*

*Figure 17: Original photos*

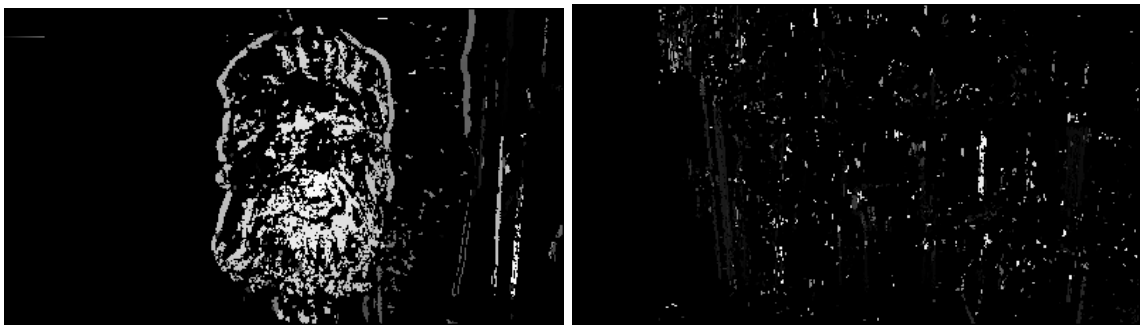


*Figure 18: Disparity Maps*

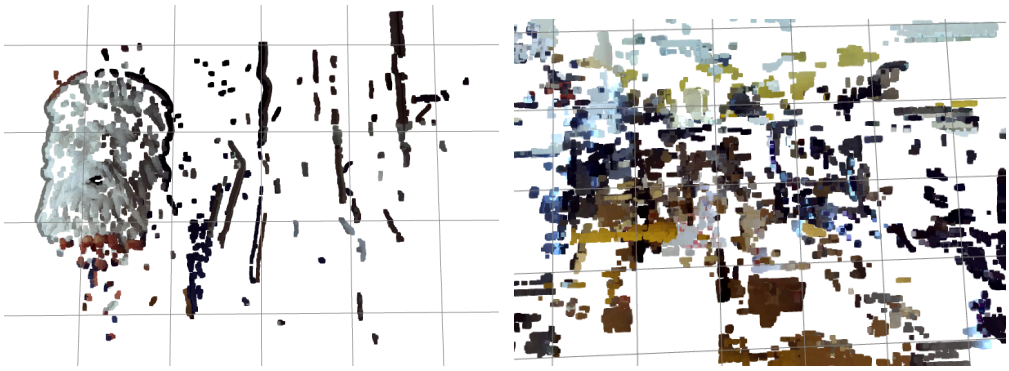


*Figure 19: 3D Point clouds of matches*

The above results are using the best parameters found through tuning with the GUI. The

most ideal parameters are below. As you can see the boxes yielded worse results despite having more feature matches and this is because finding the flat surfaces of the box and the wall and the table objects can be quite challenging.

```
# Current parameters: STATUE ALPHA 0
block_size= 2 * 10 + 5
num_disparities=16 * 25
min_disparity=134
prefilter_type=0
prefilter_size=15
prefilter_cap=62
texture_threshold=100
uniqueness_ratio=15
speckle_size=60
speckle_range=25
disp12_max_diff =25
```

```
block_size= 2 * 1 + 5
num_disparities=16 * 13
min_disparity=77
prefilter_type=1
prefilter_size=2*8+5
prefilter_cap=57
texture_threshold=31
uniqueness_ratio=6
speckle_size=2*20
speckle_range=30
disp12_max_diff =12
```

Figure 20: Ideal Parameters (statue-left, boxes-right)

It is worth to mention the most important parameters that have been tuned:

- **Block_size**: size of the comparison window; larger values smooth the disparity map but reduce detail.
- **numDisparities**: range of disparity search; defines how far features are shifted between images and affects depth range.
- **minDisparity**: starting disparity value; aligns the search range to the actual image offset.
- **uniquenessRatio**: filters ambiguous matches; higher values reduce mismatches but can leave more empty areas.
- **textureThreshold**: removes regions with low texture that produce unreliable matches.
- **speckleWindowSize / speckleRange**: eliminate small noise regions in the disparity map for smoother results.

## 8. Semi-Global Block Matching (SGBM)

The Semi-Global Block Matching algorithm refines the dense reconstruction by enforcing consistency between neighboring pixels while maintaining efficiency. This section involves replacing the BM algorithm with cv2.StereoSGBM_create() and comparing results.

```
    block_size= 11
    num_disparities=16 * 20
    min_disparity=153
    uniqueness_ratio=9
    speckle_size=100
    speckle_range=31
    disp12_max_diff =8
    prefilter_cap=63

    stereo = cv2.StereoSGBM_create(minDisparity=min_disparity,
                                   numDisparities=num_disparities,
                                   blockSize=block_size,
                                   P1=8 * 3 * block_size ** 2,
                                   P2=32 * 3 * block_size ** 2,
                                   disp12MaxDiff=disp12_max_diff,
                                   preFilterCap=prefilter_cap,
                                   uniquenessRatio=uniqueness_ratio,
                                   speckleWindowSize=speckle_size,
                                   speckleRange=speckle_range,
                                   mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY
                                   )
```

*Figure 21: Semiglobal cv2 function and ideal parameters for statue*

*Figure 22: Original and Disparity map of semiglobal statue*

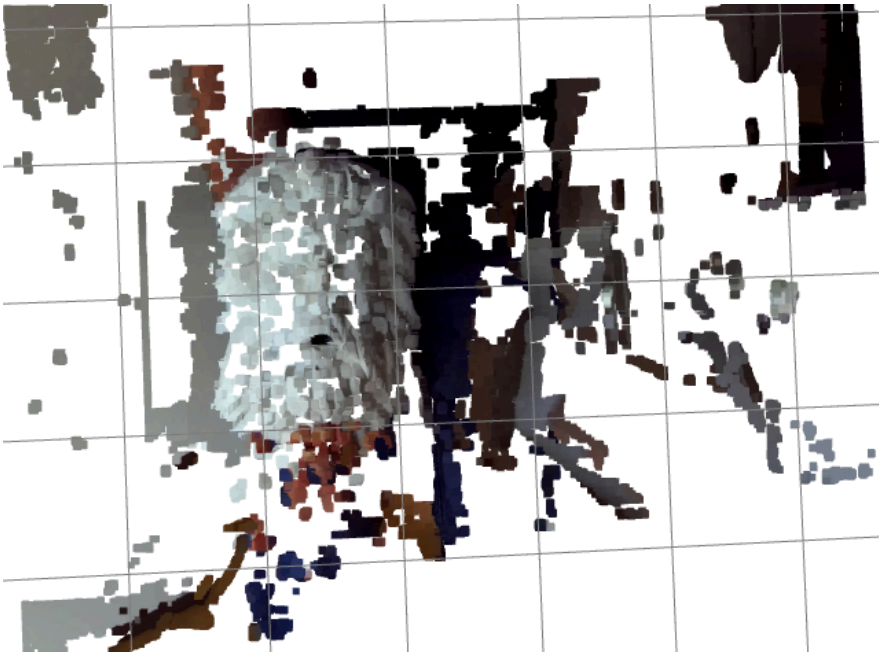


*Figure 23: 3D point cloud using semiglobal statue*

With semiglobal the results yielded a worse disparity map but a much better defined 3D point cloud when comparing it to the typical block matching. This is because it is not as strict with what it considers a feature match with less accuracy of the z axis.

The same methods can be done with the other data set of the boxes. After tuning the best parameters found are displayed below along with the yielded results.

```
block_size= 8
num_disparities=14 * 20
min_disparity=108
uniqueness_ratio=5
speckle_size=2*20
speckle_range=25
disp12_max_diff =10
prefilter_cap=63

stereo = cv2.StereoSGBM_create(minDisparity=min_disparity,
                               numDisparities=num_disparities,
                               blockSize=block_size,
                               P1=8 * 3 * block_size ** 2,
                               P2=32 * 3 * block_size ** 2,
                               disp12MaxDiff=disp12_max_diff,
                               preFilterCap=prefilter_cap,
                               uniquenessRatio=uniqueness_ratio,
                               speckleWindowSize=speckle_size,
                               speckleRange=speckle_range,
                               mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY
                               )
```

*Figure 24: Semiglobal cv2 function and ideal parameters for boxes*

*Figure 25: Original and Disparity map of semiglobal boxes*

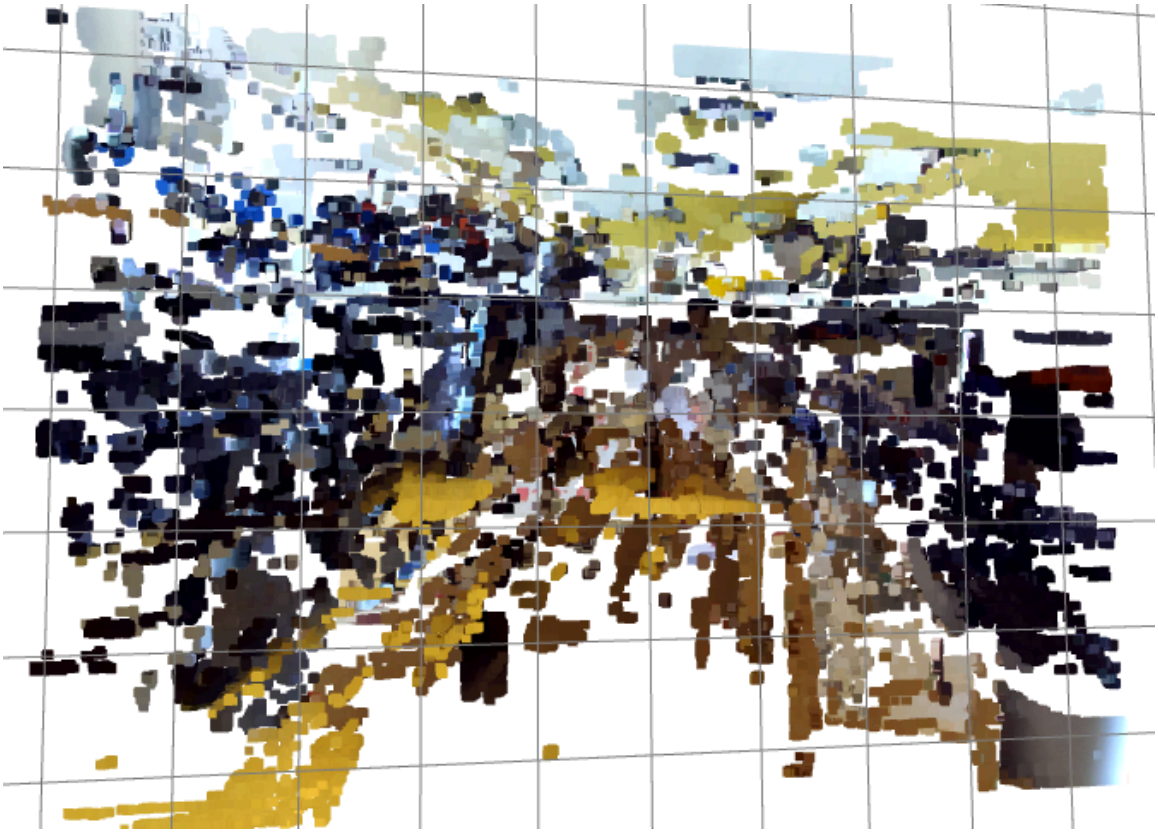


*Figure 26: 3D point cloud using semiglobal statue*

While semiglobal seems to have yielded slightly worse results for the statue for the boxes it seemed to help. This could be due to the parameter tuning being better with one than the others.

## 9. Capturing New Data

Finally, new data was captured using the ZED2 stereo camera available in the lab. The data collection process included capturing calibration and reconstruction images. The same pipeline was applied to this new dataset to evaluate robustness and generalization. Going through the same steps as above we can calibrate the cameras individually then together with stereo. The debug and errors are shown below in the following figures.

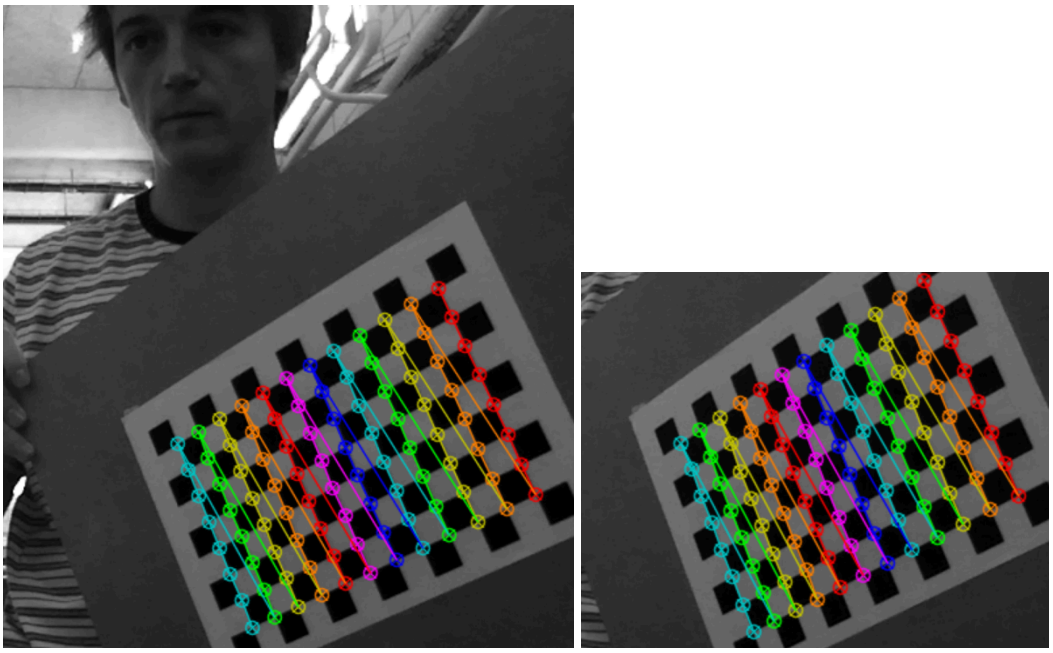


*Figure 27: Left and Right Cameras monocular calibration debug 1*

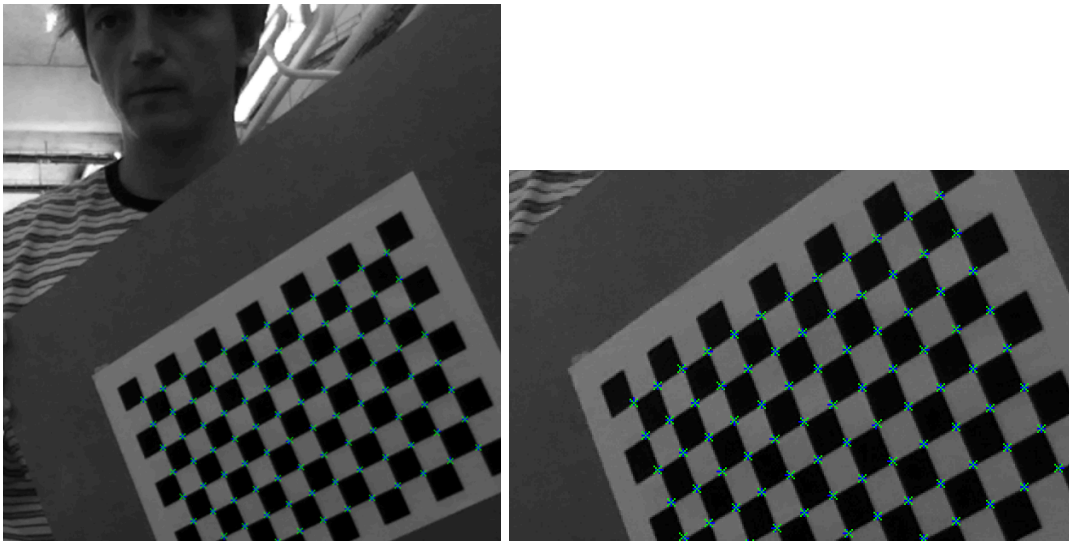


*Figure 28: Left and Right Cameras monocular calibration debug 2*

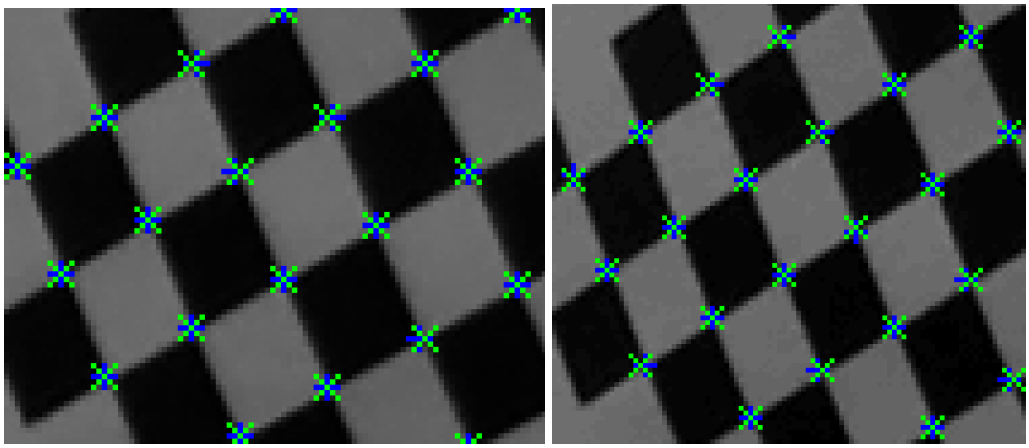


*Figure 29: Left and Right Cameras monocular calibration debug 2 (zoomed in)*

```
    - Mean reprojection error: 0.28506118965290844
    - Camera matrix:
[[1.00745449e+03 0.00000000e+00 9.42050851e+02]
[0.00000000e+00 1.00560421e+03 5.60503710e+02]
[0.00000000e+00 0.00000000e+00 1.00000000e+00]]
    - Distortion parameters:  [-0.12575445  0.08482376  0.00432356 -0.00054771
-0.03930976]
```

*Figure 30: Left Camera monocular calibration .yaml file*

```
    - Mean reprojection error: 0.27651567587152476
    - Camera matrix:
[[1.01644256e+03 0.00000000e+00 9.65498827e+02]
[0.00000000e+00 1.01353084e+03 5.65949006e+02]
[0.00000000e+00 0.00000000e+00 1.00000000e+00]]
    - Distortion parameters:  [-0.12224503  0.07200047  0.00513865 -0.00185884
-0.03188366]
- Generating reprojection images...
```

*Figure 31: Right Camera monocular calibration .yaml file*

The individual cameras seem to get an error much like the sample data or maybe even slightly better.

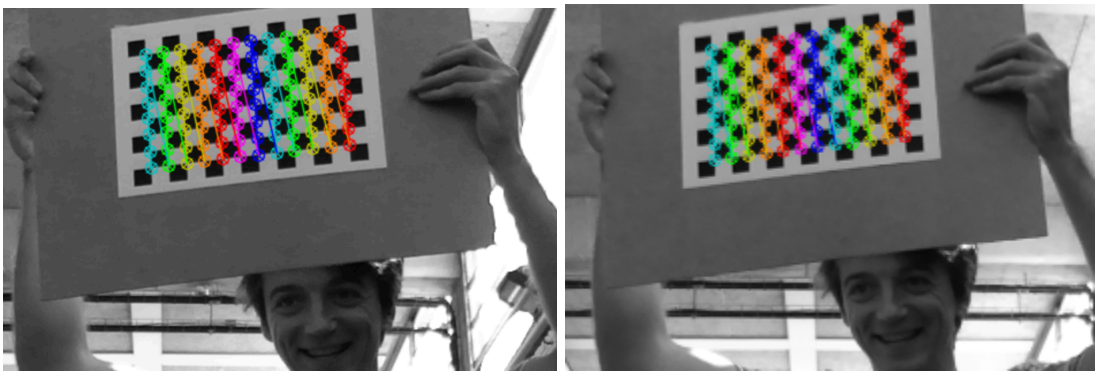


*Figure 32: Stereo calibration debug left and right cameras*

```
    - Error:  1.018113524718737
    - R:
[[ 0.99998228 -0.00120248 -0.00582975]
 [ 0.00122257  0.99999333  0.00344302]
 [ 0.00582557 -0.00345009  0.99997708]]
    - T:
[[-0.11872577]
 [ 0.00021951]
 [ 0.00677538]]
```

*Figure 33: Stereo calibration results*

The error here is much larger and it is most likely due to the chess board being a bit too far away. If error is not swaying more than a pixel i think that it is still good enough for a decent reconstruction.

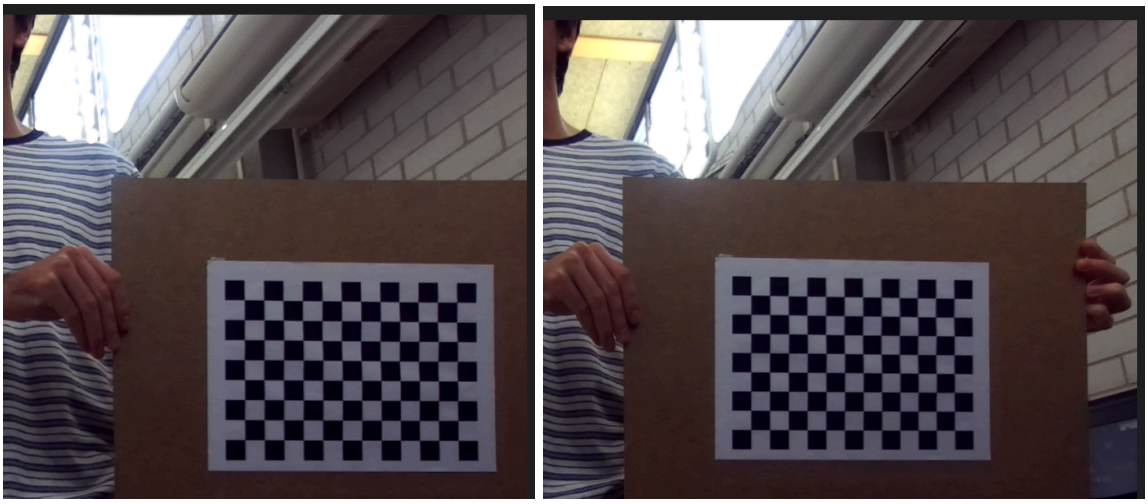


Figure 34: Alpha 0 rectified left and right images

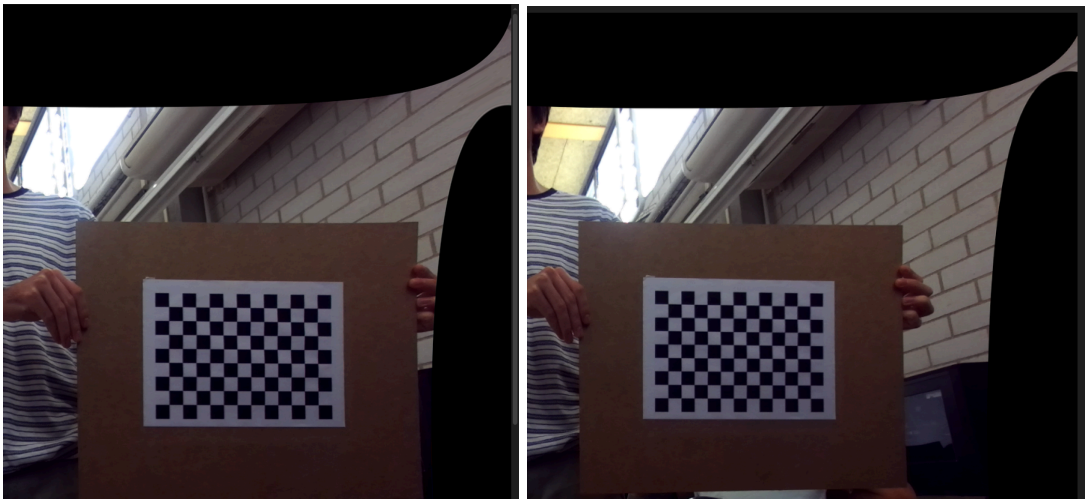


*Figure 35: Alpha 1 rectified left and right images*

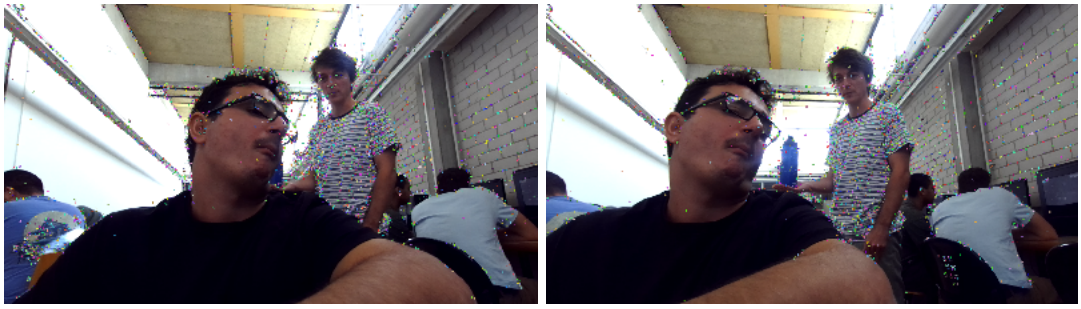


*Figure 36: Sparse feature matching with SIFT left and right images*

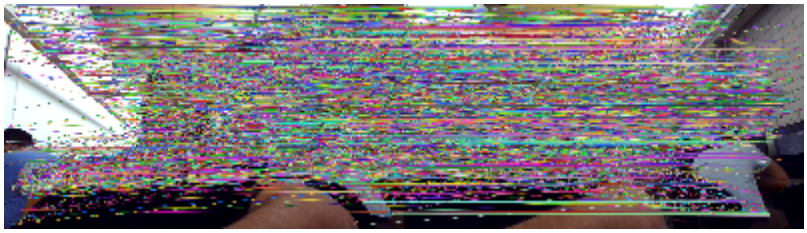


*Figure 37: Sparse feature matches with SIFT*

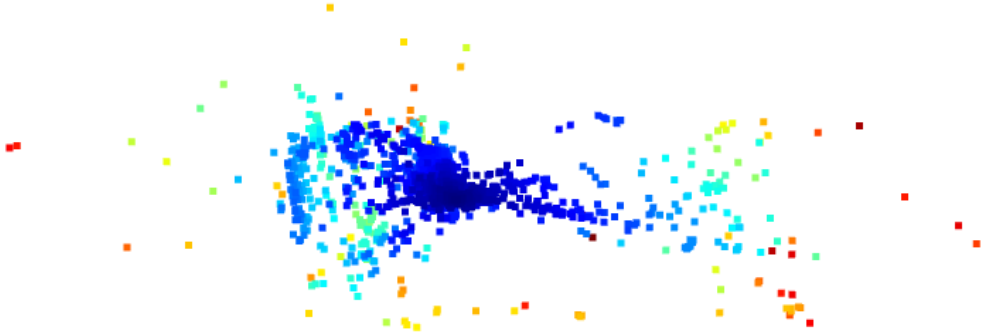


*Figure 38: Sparse feature matches with SIFT pointcloud*

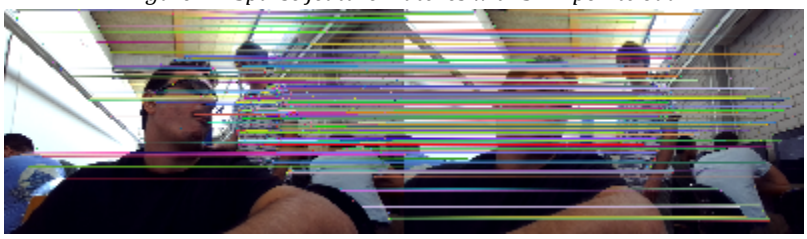


*Figure 39: Sparse feature matches with epipolar constraint (SIFT)*

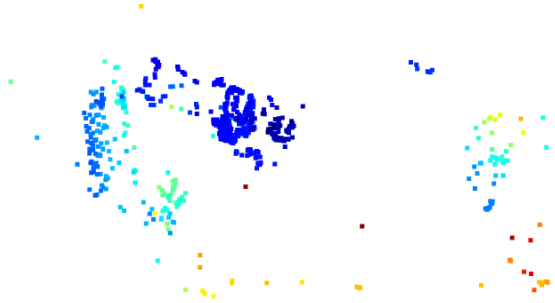


*Figure 40: Sparse feature matches 3D point cloud (SIFT)*

As observed with the sample data before applying the epipolar constraint we have a lot more matches however a lot more outliers. This indicates everything is working as before. With ORB we arrived at the same findings.

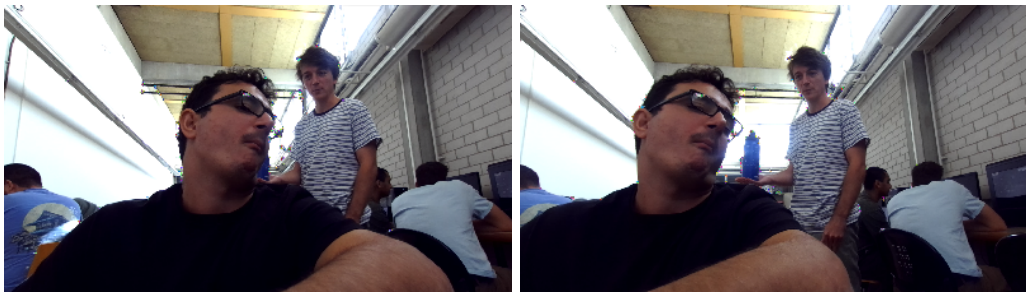


*Figure 36: Sparse matching features left and right images (ORB)*

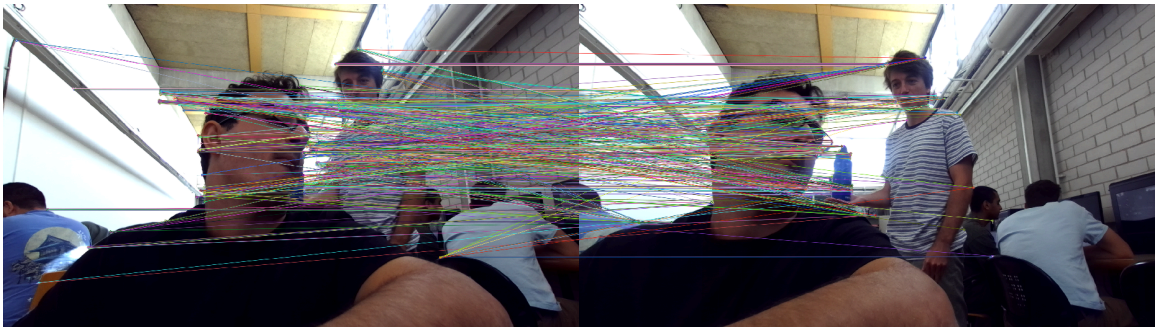


*Figure 37: Sparse matching features (ORB)*

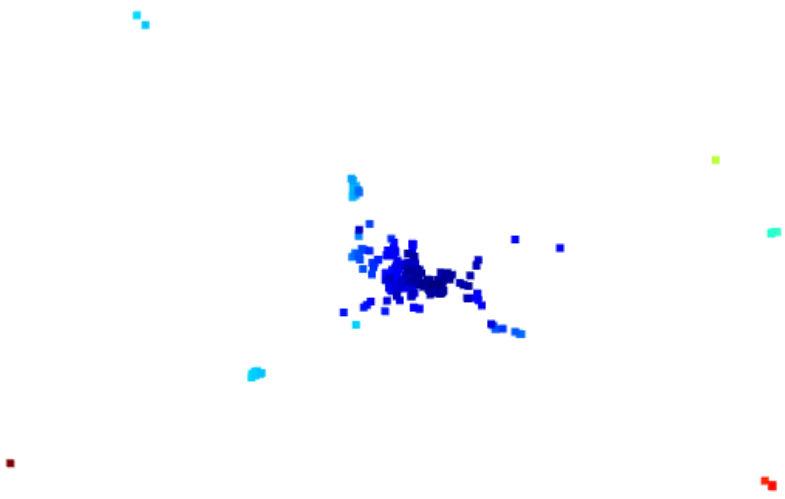


*Figure 38: Sparse matching point cloud (ORB)*

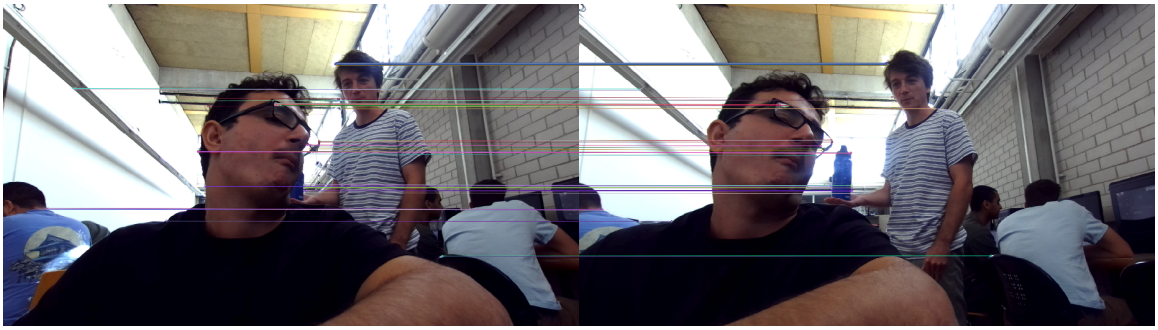


*Figure 39: Sparse matching with epipolar constraint (ORB)*

*Figure 40: Sparse matching with epipolar constraint point cloud (ORB)*

With orb again we can notice a lot less matches. Yet again displaying the importance of SIFT.

```
# Aquired image
block_size= 2 * 1 + 5
num_disparities=16 * 13
min_disparity=77
prefilter_type=1
prefilter_size=2*8+5
prefilter_cap=57
texture_threshold=31
uniqueness_ratio=6
speckle_size=2*20
speckle_range=30
disp12_max_diff =12
stereo = cv2.StereoBM_create(numDisparities=num_disparities,
                             blockSize=block_size)
stereo.setMinDisparity(min_disparity)
stereo.setPreFilterType(prefilter_type)
stereo.setPreFilterSize(prefilter_size)
stereo.setPreFilterCap(prefilter_cap)
stereo.setDisp12MaxDiff(disp12_max_diff)
stereo.setTextureThreshold(texture_threshold)
stereo.setUniquenessRatio(uniqueness_ratio)
stereo.setSpeckleWindowSize(speckle_size)
stereo.setSpeckleRange(speckle_range)
```

*Figure 40: Block Matching ideal parameters*

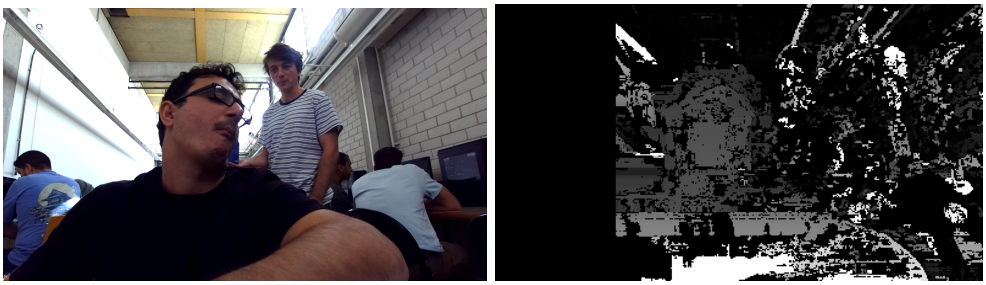


*Figure 41: Personally Acquired Photo with block matching*

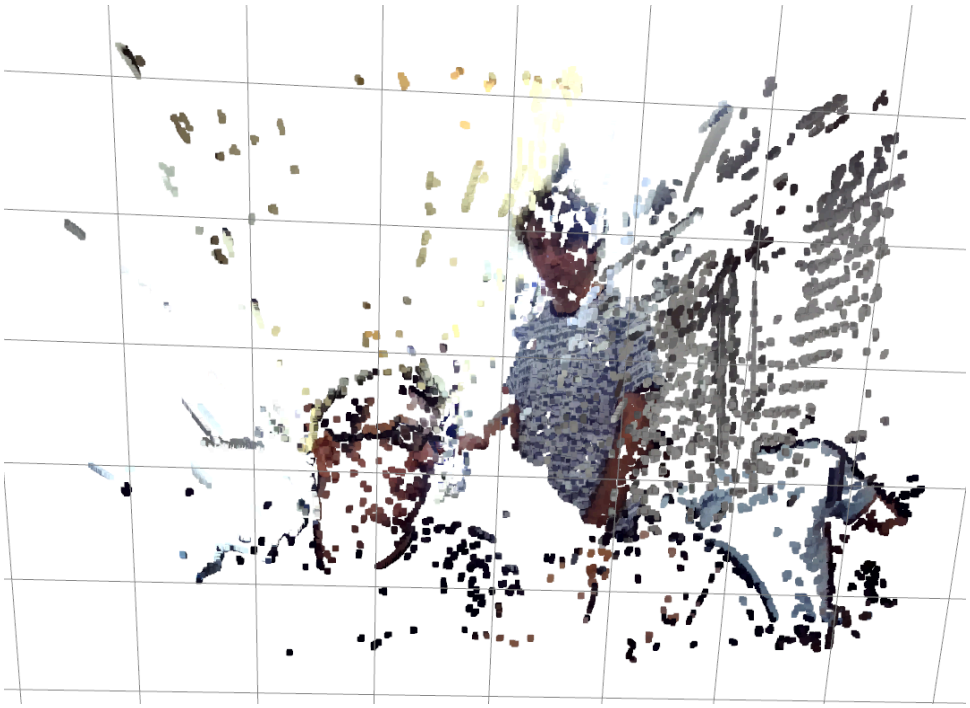


*Figure 42: Returned 3D Point Cloud with block matching*

```
block_size= 2 * 5 + 5
num_disparities= 16 * 9
min_disparity= 0
prefilter_cap= 25
disp12_max_diff= 1
uniqueness_ratio= 0
speckle_size= 2 * 1
speckle_range= 1

stereo = cv2.StereoSGBM_create(minDisparity=min_disparity,
                               numDisparities=num_disparities,
                               blockSize=block_size,
                               P1=8 * 3 * block_size ** 2,
                               P2=32 * 3 * block_size ** 2,
                               disp12MaxDiff=disp12_max_diff,
                               preFilterCap=prefilter_cap,
                               uniquenessRatio=uniqueness_ratio,
                               speckleWindowSize=speckle_size,
                               speckleRange=speckle_range,
                               mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY,
                               )
# ###
```

*Figure 43: Semi-global ideal tuning parameters found*

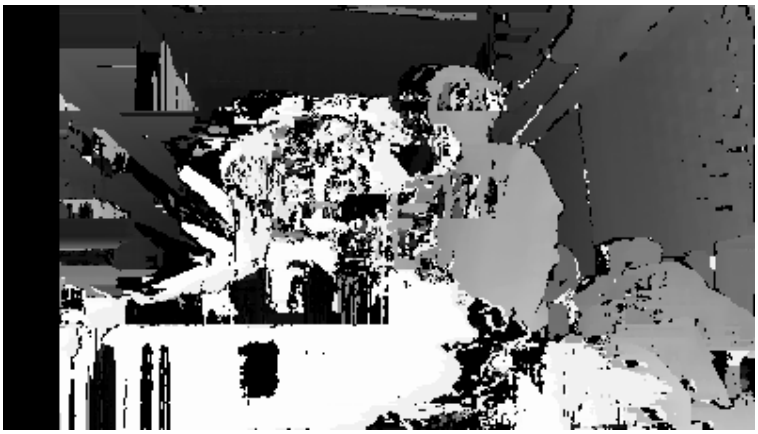


*Figure 43: Disparity map using Semi-global*

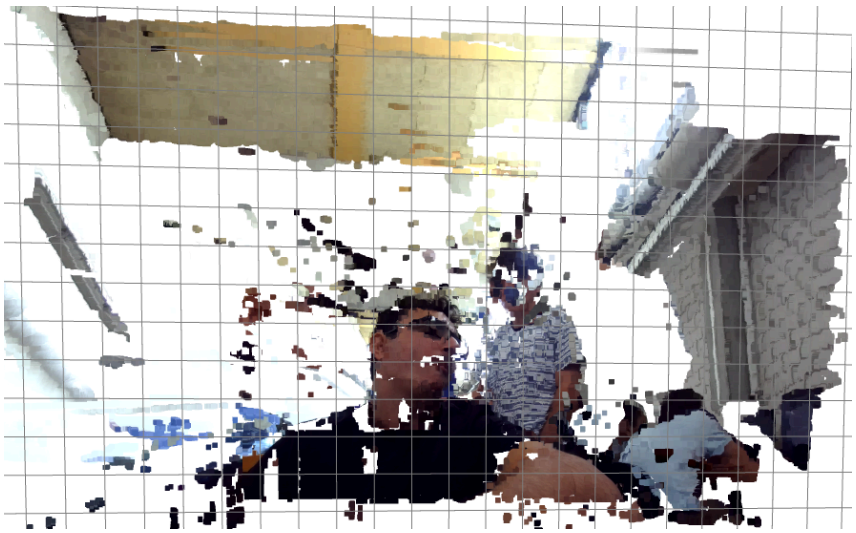


*Figure 44: Received 3D point cloud with Semi-global*

As observed in the 3D reconstruction, the process successfully recovered most of the main scene features. The first person in the foreground is not well defined, likely due to a lack of distinctive texture or recognizable features. In contrast, the person in the background and the wall were reconstructed almost completely. However, in this implementation it was difficult to achieve an optimal trade-off between accurate calibration for nearby and distant objects simultaneously. With semi-global we achieved a very strong result which is different from what we observed with the sample data. Likely this is due to a better parameter tuning that was found.

## Conclusion

In this lab, we implemented a complete stereo vision pipeline. Through successive steps of calibration, rectification, and reconstruction, we demonstrated the geometric principles behind stereo vision. Sparse reconstruction provided feature-level depth, while dense reconstruction extended this to full-image depth estimation. The results highlight the importance of accurate calibration and rectification for reliable 3D perception along with comparison of different detectors and descriptors as well as different matching methods.