Stereo Dense Reconstruction

Contents

The goal of this laboratory session is to get you familiarized with dense epipolar matching and 3D reconstruction.

1 Preliminaries

1.1 Outline of the exercise

In this exercise, you will reconstruct a 3D scene using dense epipolar matching. As in the previous exercise, we are making use of the public KITTI dataset.

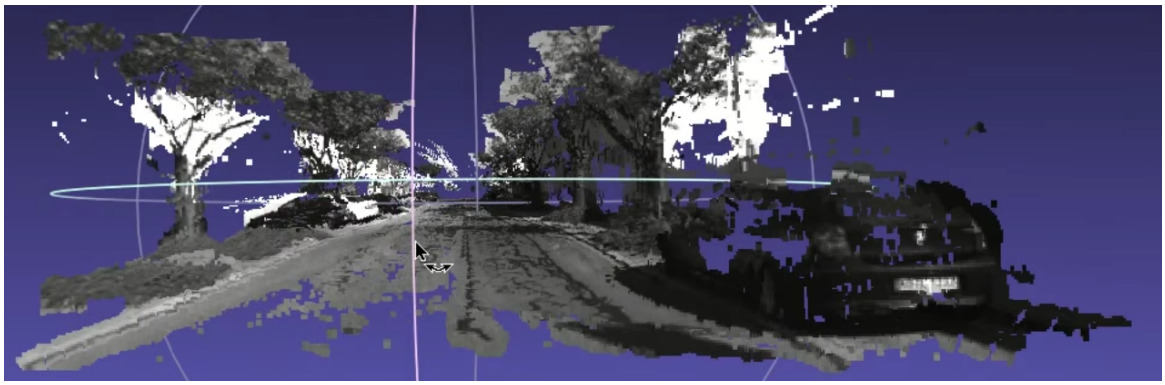

Figure 1: The final product of this exercise is a dense point cloud accumulated over a subsequence of the KITTI dataset.

You will achieve this with the following steps: First, you will try to determine pixel disparity between a left and a right stereo frame using SSD matching on a disparity range. You will parallelize this to save computation time. Second, you will apply some very simple heuristics to remove outliers. Third, you will backproject the matched pixels and triangulate the corresponding 3D point. Finally, you will use pose information of the frames to accumulate a global point cloud and visualize it in Meshlab. A video of the reference final product can be found at https://youtu.be/cyPFR61uuHA.

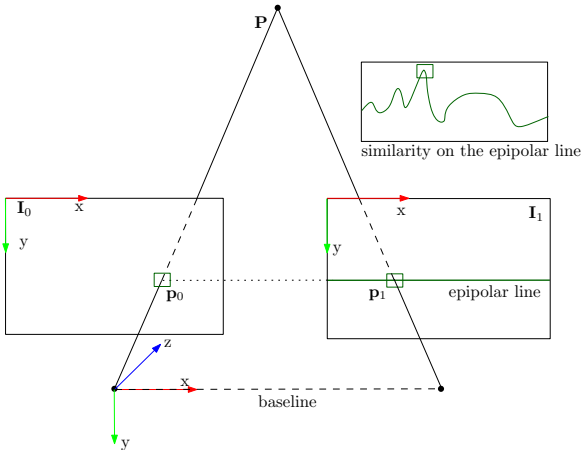

Figure 2: The geometry of the problem and notation. I_0 and I_1 are the rectified images of the left and right frame, respectively.

1.2 Provided code

As last time, we provide you with skeletal Matlab code (`main.m`) with a section for each part of the exercise. Your job will be to implement the code that does the actual logic. We also provide the functions stubs with some comments about the input and output formats, so if these are not clear from this PDF, they should be clear from the function stubs. *Again, you do not need to reproduce the reference outputs exactly.*

1.3 Conventions

Because all (square) patches need to be odd-sized, i.e. have a center pixel, we specify their size with a `patch_radius`, such that the patch has dimensions $(\texttt{patch\_radius} \cdot 2 + 1)^2$. The geometry of the problem and names of physical sizes are depicted in Fig. 2.

2 Part 1: Calculate pixel disparity

As shown in Fig. 2, the 3D point $\mathbf{P}$ projects onto locations $\mathbf{p_0}$ and $\mathbf{p_1}$ of the rectified images I_0 and I_1 respectively. Take note of the following properties:

- If the right camera has the same orientation as the left camera and is offset only in the x axis of the left camera frame, $\mathbf{p_1}$ lies on the epipolar line of $\mathbf{p_0}$ on I_1, and this epipolar line is horizontal, with the same y coordinate as $\mathbf{p_0}$.
- In particular, $\mathbf{p_1} = \mathbf{p_0} - \begin{bmatrix} d \\ 0 \end{bmatrix}$, where d is the pixel disparity of $\mathbf{p_0}$. $d \geq 0$ and $d \to 0$ for $P_z \to \infty$. We will for now assume that d is discrete.

For practical reasons, we will further assume $d \in \{d_{min}, d_{min} + 1, ..., d_{max}\}$. This has two advantages: first, putting a lower bound on d allows us to avoid noisy triangulations: as $d \to 0$, noise and rounding errors in the estimation of d have an increasing effect on the triangulated position of $\mathbf{P}$. Second, putting an upper bound on d reduces the search space and thus the computational effort. It is a valid thing to do if we know that there are no objects immediately in front of the camera, and provided that d_{max} corresponds to the minimum distance between camera and objects.

Similarly to the keypoint matching of the last exercise, the correct disparity is estimated by minimizing the SSD between image patches around $\mathbf{p_0}$ and $\mathbf{p_1}$. Formally, we need to find the optimal $d^\star$ which satisfies:

$$d^\star(\boldsymbol{p_0}) = \arg\min_d SSD_{\text{patch},\mathbf{p_0}}(x) = \arg\min \sum_{\mathbf{i} \in \text{patch}} (I_0(\mathbf{p_0} + \mathbf{i}) - I_1(\mathbf{p_0} - \begin{bmatrix} d \\ 0 \end{bmatrix} + \mathbf{i}))^2. \quad (1)$$

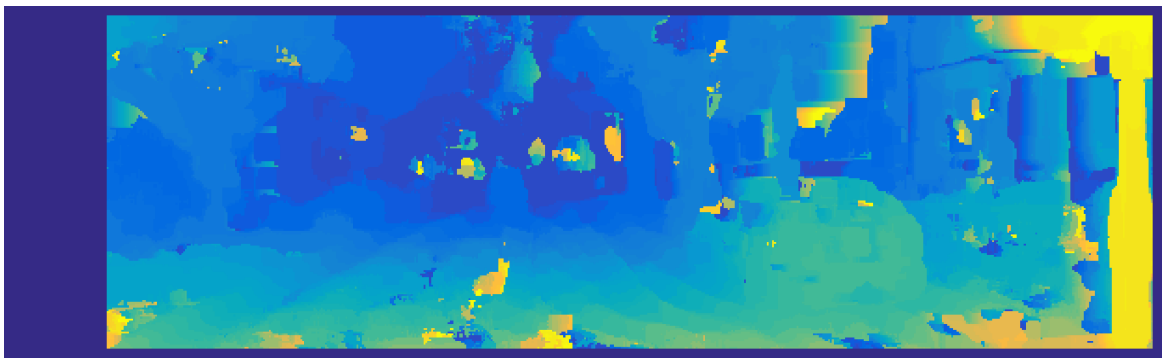

Figure 3: Unfiltered disparity image for the first stereo pair.

For this exercise, you need to implement a function that returns $d^\star$ for each pixel of I_0, given a patch radius and the bounds on disparity. Note that $SSD_{\text{patch},\mathbf{p_0}}(x)$ and thus $d^\star$ is not defined for border pixels for which the patch would not fully overlap I. It is furthermore not defined for d_{min} additional columns on the left. Simply use 0 to indicate undefined $d^\star$. To avoid the hassle of special cases, you may also set the `patch_radius`$+d_{max}$ leftmost columns to 0. Then, you should get a disparity image similar to the one shown in Fig. 3. Some hints:

- You do not need to worry about undistortion, since the KITTI images are already rectified. Also, note that we define d on the rectified (original) image, so no need to use the projection matrix K yet.
- There is a lot of potential for bugs when writing this code, and at the same time it's computationally expensive. We recommend to attack it step by step and make sure that you get what you expect in the intermediate steps. See Fig. 4 for a recommended intermediate result. This result has been obtained using among others Matlab commands `imagesc` and `pause` .
- As in the previous exercise, we recommend stacking patches to compare into a matrix and feeding them to `pdist2` for efficiency. Note that since Matlab 2016b you can use the `'squaredeuclidean'` option to actually calculate the SSD (cheaper than the default `'euclidean'`).
- As you will see, efficiency matters. We have implemented this function with three for loops: One each to iterate over rows and columns of $\mathbf{p_0}$ and one to form the matrix representing the candidate I_1 patches we feed to `pdist2`. `pdist2` does not seem to accept integer arguments and will convert integers to doubles all while printing a warning. To avoid this, while staying efficient, we recommend converting the `pdist2` inputs to singles using the `single` command.
- To squeeze out even more performance, you can try replacing the *outermost* for-loop with `parfor` (problematic with debug output, do this only as the final step). Monitor the CPU usage to ensure that you squeeze out as much as possible. You might need to increase the default maximum of parallel workers. We recommend leaving one thread for the operating system to prevent crashes. We achieve $\sim 1.6s$ with fifteen 4GHz threads, which should correspond to $\sim 6s$ for seven 2.5GHz threads.

3 Part 2: Simple outlier removal

While the results of a naive implementation as seen in Fig. 3 provide a d estimate for every pixel for which the SSD is defined, they contain many outliers (e.g. sky, yellow blobs). Implement two simple outlier filters (set the corresponding $d^\star$ to 0):

1. Reject all ambiguous matches, i.e. matches where several d candidates exhibit a score similar to the smallest score. In the reference implementation, we reject a match if more than two d candidates have an SSD less than or equal $1.5\times$ the minimum (manually tuned to this dataset).

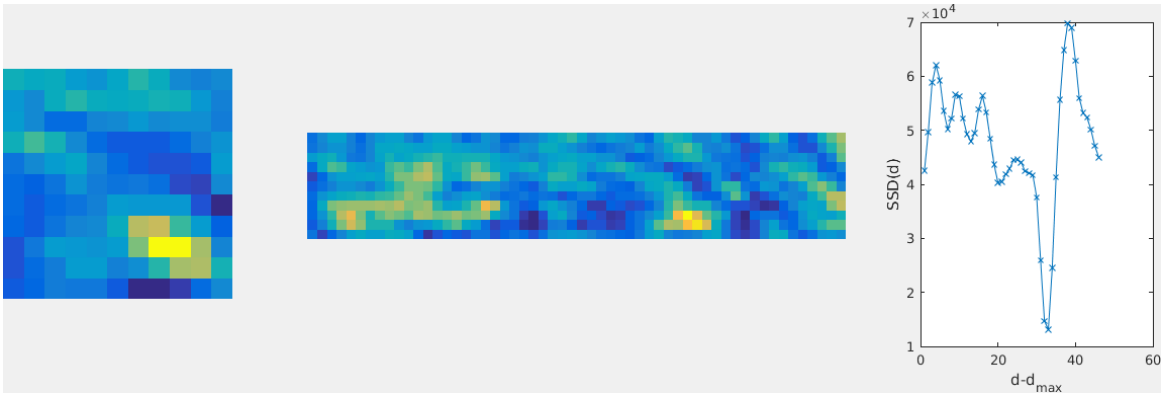

Figure 4: Debugging disparity matching. From left to right: patch around the first valid $\mathbf{p_0}$, excerpt from I_1 against which the left-hand patch can be matched, and SSDs for different values of d.

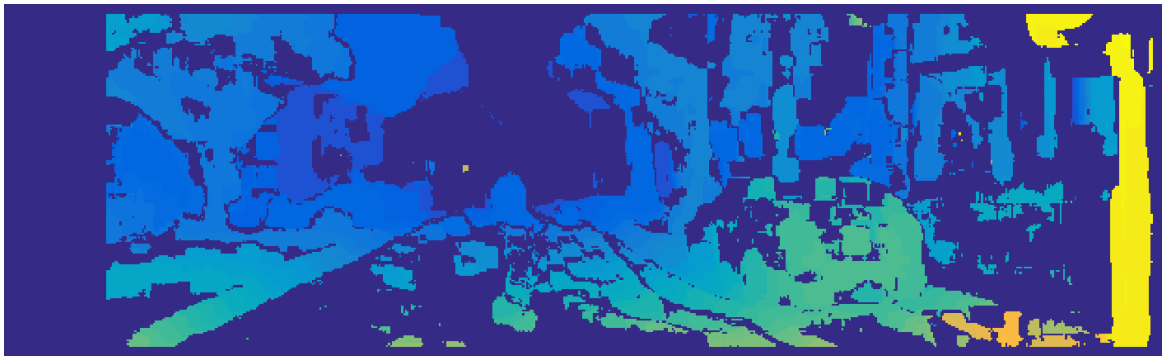

Figure 5: Filtered stereo disparity for the first stereo pair. Might look less appealing, but has far less outliers.

Why two? This happens to be nicely illustrated in Fig. 4. Because a continuous d (see section 5) would lie pretty much in the middle between two pixels, both of these pixels exhibit a low score, so we can still accept one of them as inlier. When implementing this filter, pay attention to the case where several SSDs are 0, which is the case in over-exposed regions of the image.

2. If the lowest SSD occurs at d_{min} or d_{max}, this might indicate that there is a local minimum outside of the provided disparity range. Having rather fewer false positives than many true positives, we also reject these d estimates.

After applying these filters in the same function as Part 1, you should get a disparity image similar to Fig.5. At this point, if your code is fast enough, you can enjoy running the disparity estimation on the entire sequence. Compare to https://www.youtube.com/watch?v=czEo6XEtwAQ.

4 Part 3: Point cloud triangulation

Now that we have determined $\mathbf{p_1}$ for each $\mathbf{p_0}$ (or that there is none), we can triangulate $\mathbf{P}$. As illustrated in Fig. 2, $\mathbf{P}$ projects into $\mathbf{p_0}$ and $\mathbf{p_1}$, which can be expressed as:

$$\lambda_0 \begin{bmatrix} \mathbf{p_0} \\ 1 \end{bmatrix} = \mathbf{KP}, \quad \lambda_1 \begin{bmatrix} \mathbf{p_1} \\ 1 \end{bmatrix} = \mathbf{K}(\mathbf{P} - \begin{bmatrix} b \\ 0 \\ 0 \end{bmatrix}) \tag{2}$$

where b stands for the baseline between the stereo frames. By applying K^{-1} on the left and re-arranging terms, we get

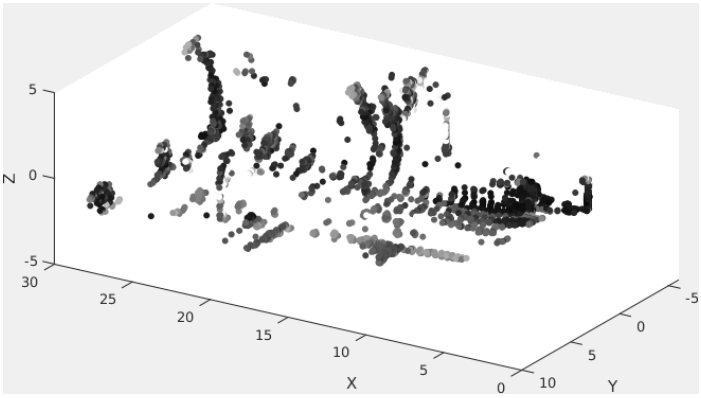

Figure 6: Point cloud of first stereo match, rotated into the world frame, downsampled by a factor of 10 for rendering speed, and limited to a $30 \times 16 \times 10$ bounding box to reject outliers.

$$\mathbf{P} = \lambda_0 K^{-1} \begin{bmatrix} \mathbf{p_0} \\ 1 \end{bmatrix} = \lambda_1 K^{-1} \begin{bmatrix} \mathbf{p_1} \\ 1 \end{bmatrix} + \begin{bmatrix} b \\ 0 \\ 0 \end{bmatrix} \tag{3}$$

We may tranform the above equation into a single matrix equation by rearranging

$$\lambda_0 K^{-1} \begin{bmatrix} \mathbf{p_0} \\ 1 \end{bmatrix} - \lambda_1 K^{-1} \begin{bmatrix} \mathbf{p_1} \\ 1 \end{bmatrix} = \begin{bmatrix} b \\ 0 \\ 0 \end{bmatrix}$$

$$\lambda_0 \begin{bmatrix} \hat{\mathbf{p}}_\mathbf{0} \\ 1 \end{bmatrix} - \lambda_1 \begin{bmatrix} \hat{\mathbf{p}}_\mathbf{1} \\ 1 \end{bmatrix} = \begin{bmatrix} b \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \hat{\boldsymbol{p}}_\mathbf{0} & -\hat{\boldsymbol{p}}_\mathbf{1} \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \end{bmatrix} =: A\lambda = \begin{bmatrix} b \\ 0 \\ 0 \end{bmatrix}$$

You should notice that this system is overconstrained, i.e. A is 3×2 and cannot be inverted (the last row simply ensures that $\lambda_0 = \lambda_1$, which is implied in the geometry of the problem). So in order to obtain the most fitting λ, apply the least squares approximation: $A^T A\lambda = A^T \mathbf{b}$. Once you have solved this system, you can recover $\mathbf{P}$ using the left equation in (3). Do this for every $\mathbf{p_0}$ in I_0 with a valid $d^\star$. In the code, we also ask you to associate the correct image intensity to each point. Simply pick it from I_0.

Be careful with indices in this exercise! As specified in Fig. (2), x corresponds to the image column and y to the image row! The point cloud you get should roughly look like the one in Fig. (6) (note that there is a rotate button in the `scatter3` plot). Can you identify the different parts of the scene?

5 Part 4: Sub-pixel refinement

As you can see in Fig. (6), the resulting pointcloud exhibits distinct layers. This is a consequence of our choice to make d discrete. We can, however, look for a continuous disparity by apllying a very simple trick to (1): We can interpolate $SSD_{\text{patch},\mathbf{p_0}}(x)$ at $x = \{d^\star - 1, d^\star, d^\star + 1\}$ using a second-order polynomial fit, then replace $d^\star$ with the arg min of the polynomial. Modify the function `getDisparity` to do this using the Matlab function `polyfit`, then re-run the disparity matching (the disparity image should look similar, but with smoother color transitions) and point cloud generation parts of main.m . You should now get a point cloud similar to the one in Fig. 7.

Once you have the point cloud for the first stereo pair triangulated properly, you are ready to create the point cloud of the full scene, as shown in Fig. (1.1) and the preview video. No need for

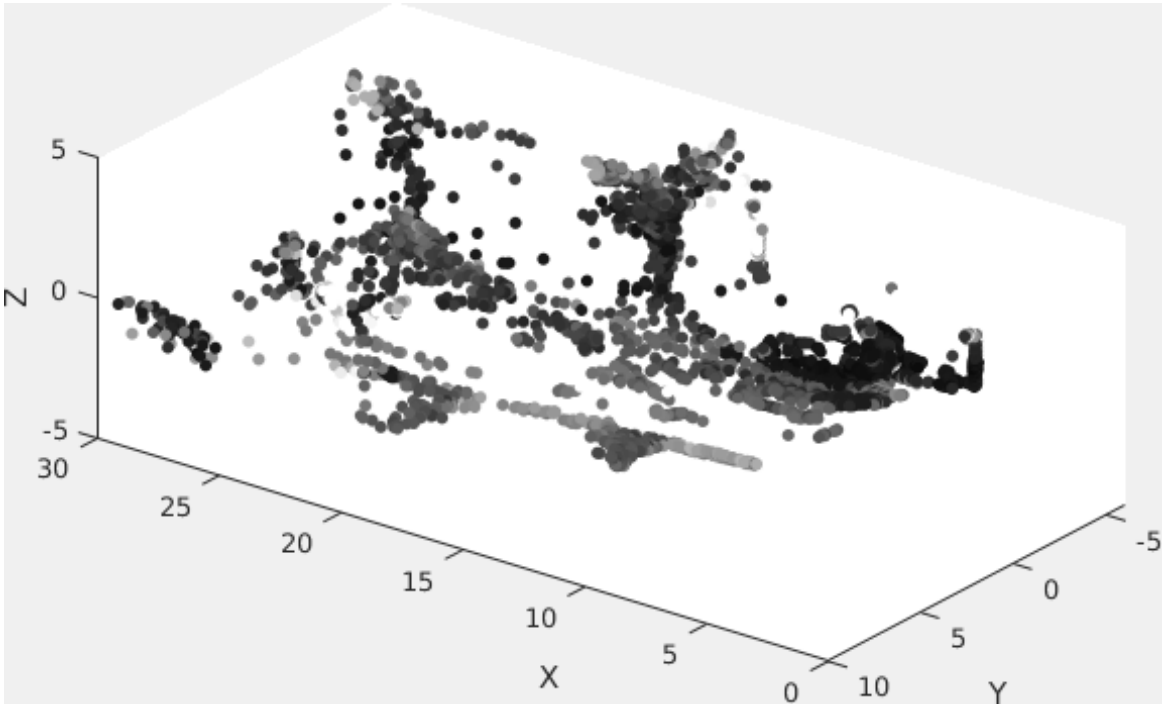

Figure 7: Same as Fig. 6, but with sub-pixel refinement.

additional code, but you will need time (probably an hour or more) and Meshlab (`sudo apt-get install meshlab` on Ubuntu). Once the corresponding part in `main.m` has finished, a PLY file `points.ply` is created that you can view in Meshlab.