Computer Vision

Exercise 6: Stereo Matching

Viviane Yang 16-944-530

13th November 2019

Today I am going to implement the winner-takes-all and graph-cut stereo matching algorithms to create a disparity map between two images. Since the camera configuration is known, we can make a depth estimation using simple triangulation and therefore recover a §D model of the object, which will then be displayed using a tool called MeshLab.

1 Disparity computation

We compute the sum of squared distances (SSD) and sum of absolute distances (SAD) using a bos filter of size $k \times k$ for every disparity in the disparity range $[-40, 40]$ and use the best one for each pixel, that minimizes the sum for each pixel. This is how we get a disparity map of the image. k determines how sensitive one is to depth discontinuities. The larger k, the more neighboring distance differences are taking into account. It therefore smooths discontinuities as seen in the following:

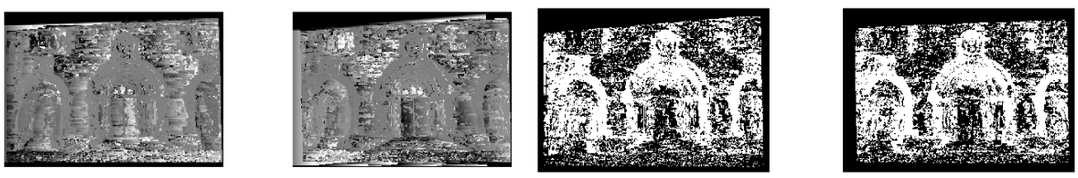

Figure 1: $k = 3$

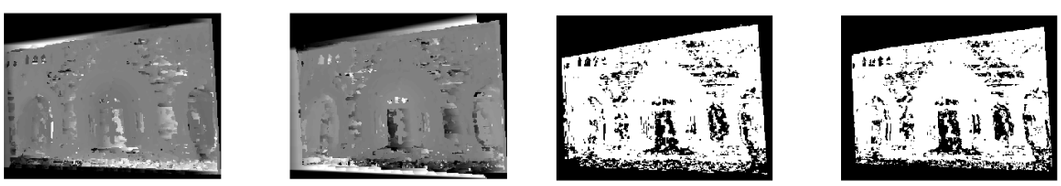

Figure 2: $k = 7$

Figure 3: $k = 20$

With a box filter size of $k = 20$, we get smooth results and not too many discontinuities. However, the border around the image that is due to the use of a convolution is getting thicker with increasing k.

1.1 Comparing SSD and SAD

Now, I want to examine the difference between SAD and SSD as the metric. SSD would penalties the sum of distances more than SAD, since they are squared. The results are better of more stable in a way, since large discontinuities in difference get penalized more as seen in Figure 4

Figure 4: 3D models of winner-takes-all for $k = 20$ with SAD(left) and SSD(right)

2 Graph-Cut

We now compute the disparity map using graph-cut. Graph-cut formulates the problem as a graph problem using the disparity map as a cost function. It has the SSD values as a third dimension at different disparity value and then finds the cut with the lowest cost.

2.1 Tuning parameter k

We can again plot the disparity maps for $k = 3, 7, 20$:

Figure 5: Graph cut with $k = 3$

Figure 6: Graph cut with $k = 7$

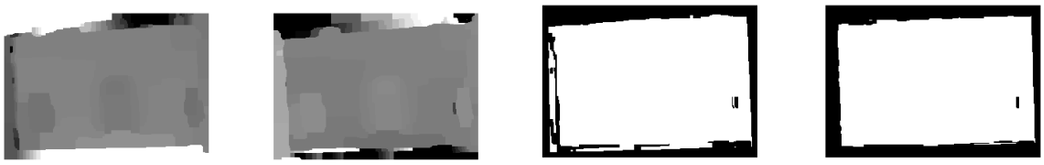

Figure 7: Graph cut with $k = 20$

k, the box filter size, does not seem to have a huge impact on the performance of graph cut. All of the results are quite smooth in terms of depth discontinuities. With $k = 7$ you can already hardy see any discontinuities. On the left two pictures, we can see the door wells, since they have a different color than the wall. On the right two pictures we see that most of the area got matched.

2.2 Comparison to winner-takes-all

Graph cut favors neighboring pixels and thus has not as many discontinuities as winner-takes-all. However, it is computationally more expensive and therefore needs more time to run. The results are a lot better and give a good 3D reconstruction of the entrance image:

Graph cut really outperforms winner-takes-all in terms of accuracy. In the graph cut model, you can even see the stairs and the archway that have different depths to it. We also observe less gaps in the model as well as no distortions as in the winner-takes-all model.

Figure 8: 3D reconstruction using graph cut with $k = 20$

Figure 9: 3D reconstruction using winner-takes-all with $k = 20$

3 Automate disparity range

The Default disparity range $[-40, 40]$ is too large for the given image, thus sometimes especially on the edges, we get disparity values that have a large absolute value, which do not make sense for the image. This is why I have come up with a method to automatically compute the disparity range for any pair of rectified stereo images.

3.1 Getting matched points

Using the given `getClickedPoints.m` function, we can manually click on the corresponding points on both rectified images and then compute the maximum distance between two matched points $maxR$. This is then a indicator for the disparity range. Using the maximum disparity, our range is then $[-maxR, maxR]$.

3.2 Automation of matching points

However, if you want to automate the whole process, you can also use a feature extraction technique to find interesting points and then match them using RANSAC algorithm as discussed in Exercise 4. I found out that since the images are already rectified and the camera positions are close together, the sampson error is very small. This is why I have set the threshold for inlier detection to 0.3. From those matched points I have gotten through RANSAC, I then compute the distance of the points. The maximum distance is then $maxR$. Since the feature extraction parts selects points randomly, we get different points each time. To filter out noises, I ignore points that have a distance larger than 40 and run RANSAC several times to average over the maximum distance. With this method, I get values roughly between 7 and 13 at a iteration of 10. If you want to find a even exact value, you can increase the number of iterations. However, the trade of is a increase in computational time. Using a iteration of 100, the average maximum difference is 8. The implementation of the algorithm can be found in `getDispPoints.m` and `getMatchedPointsRANSAC.m`.

```
function maxR = getDispRange(imgRectL, imgRectR, upper)
    iter = 10;
    maxRList = zeros(1, iter);
    for i=1:iter
        [x1s, x2s] = getMatchedPointsRANSAC(imgRectL, imgRectR);
        dispMatched = vecnorm((x1s - x2s));
        dispMatched = dispMatched(dispMatched < upper);
        maxRList(i) = ceil(max(dispMatched));
    end

    maxR = ceil(mean(maxRList));
end
```

```
function [x1s, x2s] = getMatchedPointsRANSAC(img1, img2)
    img1 = single(rgb2gray(img1));
    img2 = single(rgb2gray(img2));

    %extract SIFT features and match
    [fa, da] = vl_sift(img1);
    [fb, db] = vl_sift(img2);
    [matches, scores] = vl_ubcmatch(da, db);

    x1s = [fa(1:2, matches(1,:)); ones(1,size(matches,2))];
    x2s = [fb(1:2, matches(2,:)); ones(1,size(matches,2))];

    threshold = 0.3;

    [M, inliers, F] = ransac8pF(x1s, x2s, threshold);

    x1s = x1s(1:2, inliers);
    x2s = x2s(1:2, inliers);

end
```