Darius Rückert
Prathmesh Madhu
2.07.2020

The deadline for this exercise is on Sunday 16.07.2020 at 23:59

Stereo Vision

1 Patch Match Disparity Maps

In this task, you will implement a simple patch match algorithm to compute disparity maps from rectified gray-scale images. The program runs the code on two images from a Middelbury stereo dataset. The algorithm to implement is a simplified version of *Patch Match Stereo* which is based on the famous *Patch Match* algorithm. The algorithm itself is implemented in the function `computeDisparity` and consists of the following steps:

Algorithm

1. Initialize disparity map with random values.
2. Propagate disparity to nearby pixels.
3. Optimize disparity by searching in a small neighborhood.
4. Go to 2 until #maxIterations is reached.

Resources

- Patch Match [Wiki]
- Patch Match [Paper]
- Patch Match [Video]
- Patch Match Stereo [Paper]
- Patch Match Stereo [Video]

Figure 1: The two input images of the *Tsukuba* dataset and the disparity map computed in this task.

1.1 Matching Cost

The first thing to do in any stereo matching algorithm is to define a cost function $C(p, d)$ that describes how good a disparity value d is for a give pixel p. In this task we will use the sum of absolute differences (SAD), which directly compares the image intensities:

$$C(p,d) = \frac{1}{N} \sum_{(x,y)\in\Omega_p} |I_L(x,y) - I_R(x-d,y)|,$$

where Ω_p is a small patch around p and N is the number of pixels in Ω_p. Since the disparity d can take non-integer values, it is important to use linear interpolation when sampling the right image.

Implement the function `cost` in `disparity.py`. Make sure that the test case succeeds before continuing with the next task.

1.2 Random Initialization

In Step 1 of the algorithm, the disparity image is initialized with random values. For each pixel, a fixed number of disparity samples are drawn in the range $[d_{min}, d_{max}]$. The disparity with the lowest cost according to (Task 2.1) is saved and stored in the disparity image. Figure 2 shows the disparity image after a few iterations. Feel free to change the iteration count `initIterations` in `disparity.py`.

Implement the function `initRandom` in `disparity.py`. Random values in the correct range can be sampled with `np.random.uniform()`.

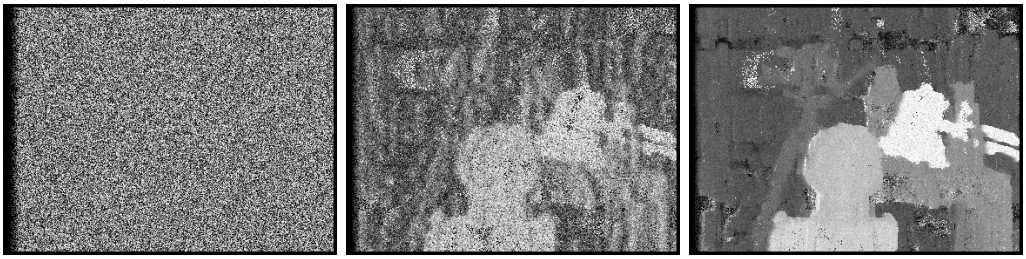

Figure 2: The random disparity map after 1, 3, and 15 iterations.

1.3 Spatial Propagation

After the random initialization, we propagate disparity values to nearby pixels, because neighboring elements are likely to have similar depth values. This propagation is computed in scan-line order from top left to bottom right. The image is iterated row-wise and each pixel reads the disparity of the left and upper neighbor. The cost is computed for these two disparities and compared to the current cost of the pixel. The disparity producing the lowest cost is then written back to the current pixel.

Implement the function `propagate` in `disparity.py` that iterates over the disparity image and propagates good values to nearby pixels. Note that for each pixel the disparity of the left and upper neighbor must be sampled.

After you have implemented the forward propagation, this step is also done in reverse order. Modify the `propagate` function so that it can handle backward propagation. Use the flag `forward` to decide what direction is used. Note that in backward order for each pixel the disparity of the lower and right neighbor has to be sampled.

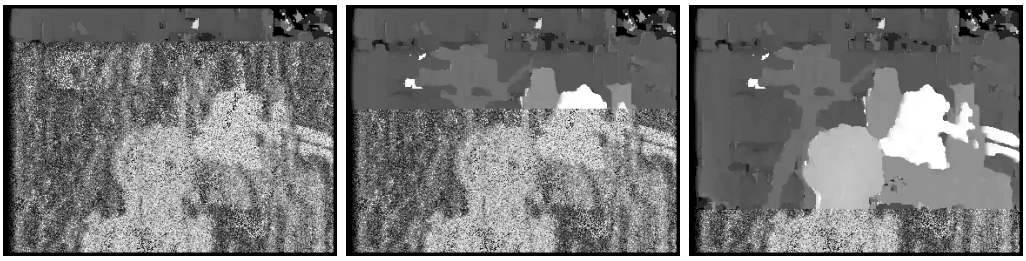

Figure 3: The down sweep of the spatial propagation on a noisy input image.

1.4 Local Search

After each propagation step we do a local search to optimize the current disparity. For that, a fixed number of new disparity values with decreasing distance is sampled around the current estimate. The best disparity (including the old value) survives and is stored in the pixel. Given the initial search radius R_0, the disparity offset Δd at iteration i is

$$\Delta d = \frac{\alpha R_0}{2^i},$$

where α is a uniform random value in the range $[-1, 1]$.

Implement the function `randomSearch` in `disparity.py`.

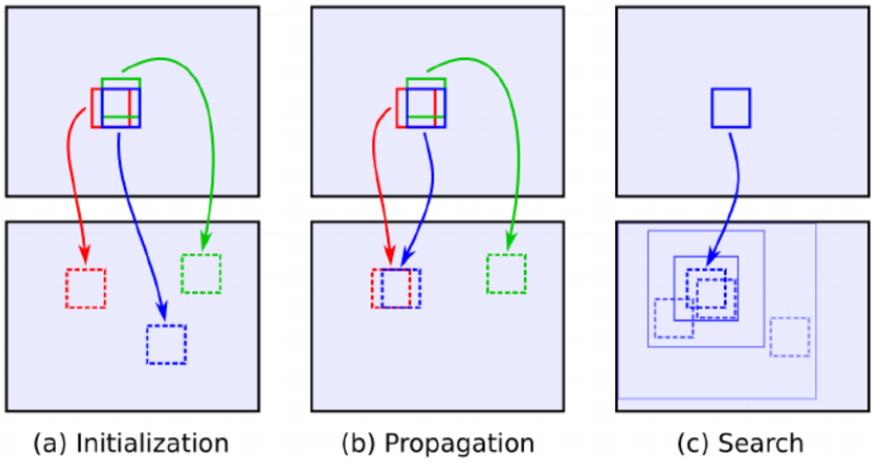

Figure 4: The three steps of the patch match algorithm. Note that the search radius is halved in each step.

2 Outlier Removal

After an independent disparity map is computed for each image, outliers are removed in three stages. First, a simple thresholding removes all values which cost is too large. Then, connected regions are detected and removed if the region is too small. Finally, we test for disparity consistency between the left and right image.

2.1 Thresholding

Implement the function `thresholding` in `disparity.py`. Invalidate the disparity measurement (by setting it to 0) if the cost is above the threshold t.

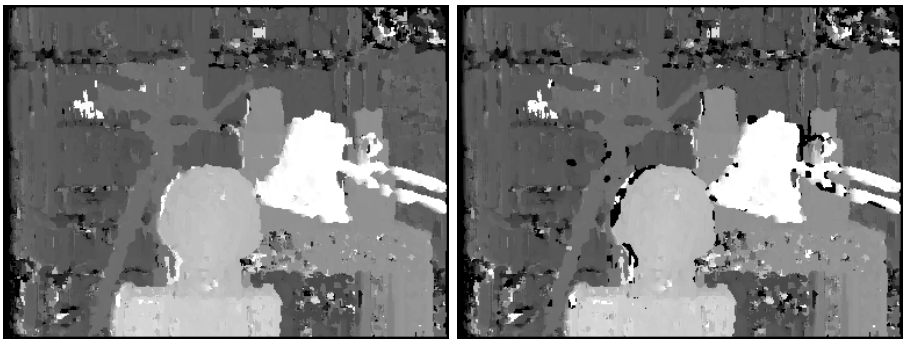

Figure 5: Before (left) and after (right) thresholding.

2.2 Segmentation

The most reliable outlier detection technique for patch match based stereo algorithms is a connected region segmentation. The idea is that, outliers most often appear in small connected patches. Inliers, on the other hand, are part of much larger regions. By counting the number of pixels in each connected patch, we can filter out small patches and therefore remove ouliers.

Implement the function `segmentation` in `disparity.py`. All disparity values which belong to a region smaller than `self.params.minRegionSize` must be removed by setting them to 0. Two neighbouring pixels belong to the same patch, if the disparity difference is smaller than `self.params.segmentationThreshold`.

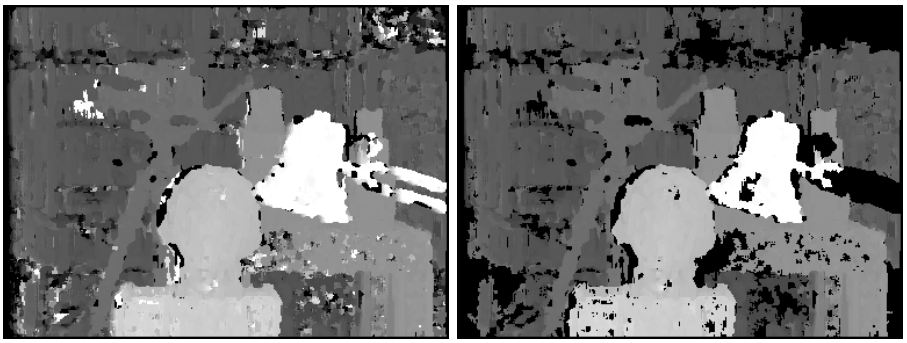

Figure 6: Before (left) and after (right) segmentation based outlier removal.

2.3 Left-Right Consistency Check

Due to occlusion, for some pixels it is impossible to find a correct disparity value. The best way to find such regions is a left-right consistency check. Each disparity value of the left image is projected to the right and compared if these values match. Note: Remember that the right image contains negative disparities.

Implement the function `consistencyCheck` in `disparity.py`.

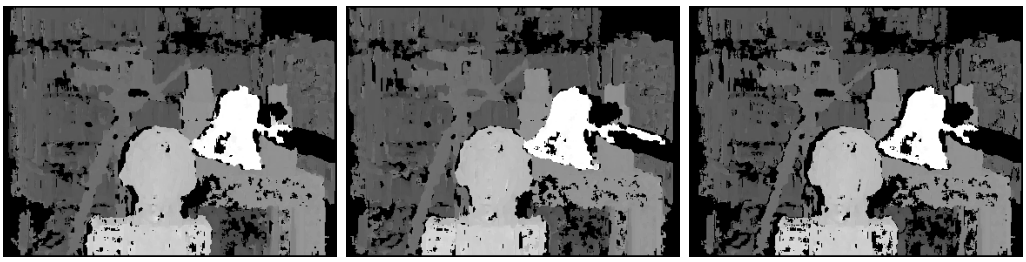

Figure 7: Disparity of the left and right image. Right: Consistent values.