EE5731 Visual Computing

Assignment 2: Depth Estimation from Stereo and Video

Student Name: Nie Hongtuo

Student ID: A0224712N

9 Sep 2020

Part 1: Noise Removal

Implementation

This task is basically a bi-label MRF optimization problem. From this part we can know how the graph cut library GCMex is used. Note that the naïve way of constructing the graph structure parameter **`pairwise`** is slow and troublesome. In function **`construct_graph.m`**, I explored several ways to construct the sparse matrix object **`pairwise`**. I finally came up with a way that uses no loops and minimum access over **`pairwise`**. This speed up the process significantly (from 40s to 6s on my laptop), as you can see in future subtasks. Another looping version of this is prepared for finding the best parameter **`lambda`**.

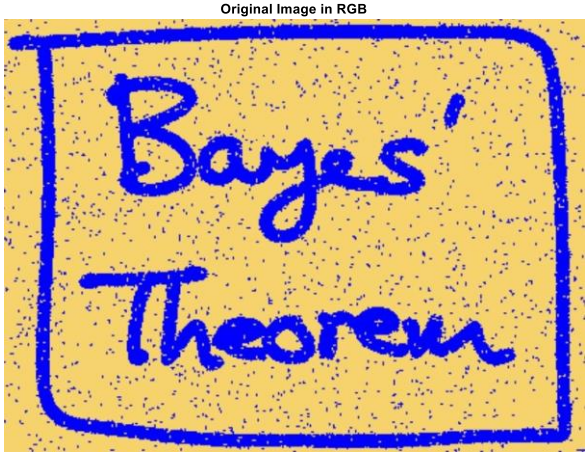
As introduced in *Computer Vision Models, Learning, and Inference*, zero-diagonal pairwise term is used, and it is set to [0 1;1 0]. The **`class`** term of GCMex is not introduced in the lectures. It is the initial state of the MRF nodes. In this part, it is fine to set all '0's or '1's. But in multilabel scenarios like part 2~5, the calculation will take much more time if we just set class randomly or set to the same labels.

This part is quite straight forward, so let us jump to the results.

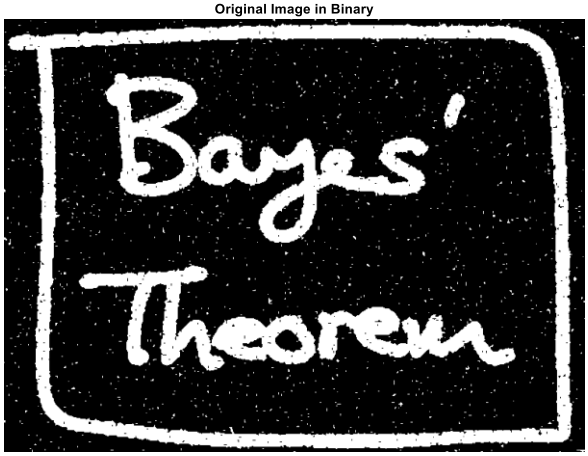
Results

The binary graph shows that this is a bi-label problem. You can also run **`p1_noise_removal_loop.m`** to get more results under different **`lambda`**. The results are place under the gallery folder.

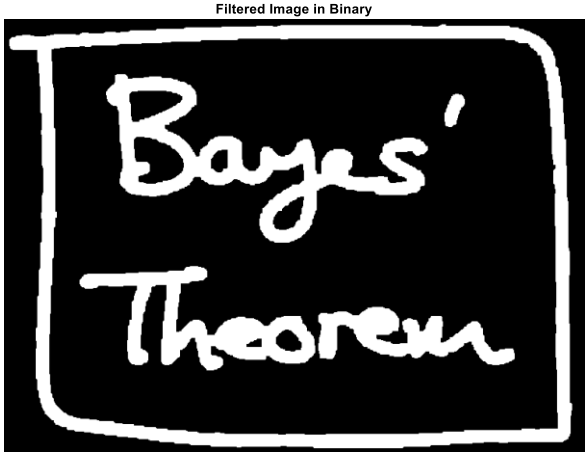
Original Image in RGB



Original Image in Binary

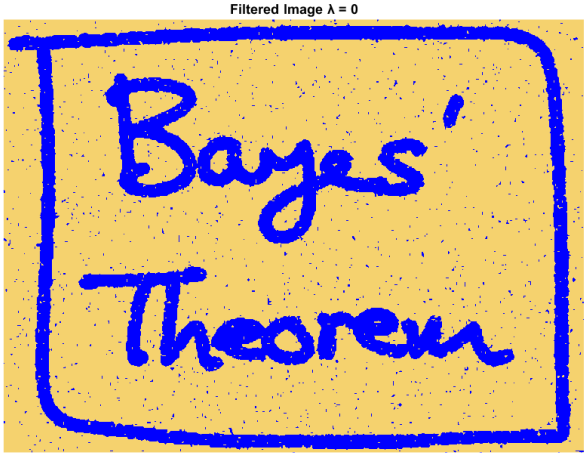


Filtered Image in Binary

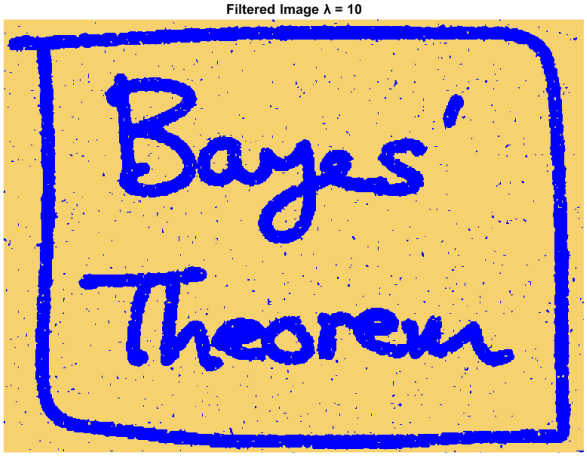


Filtered Image in RGB

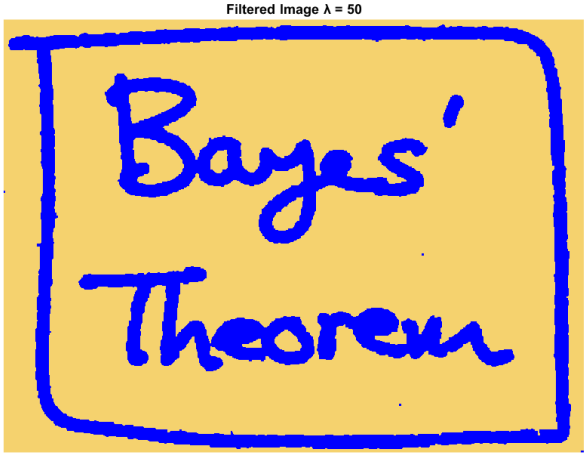
Filtered Image λ = 0



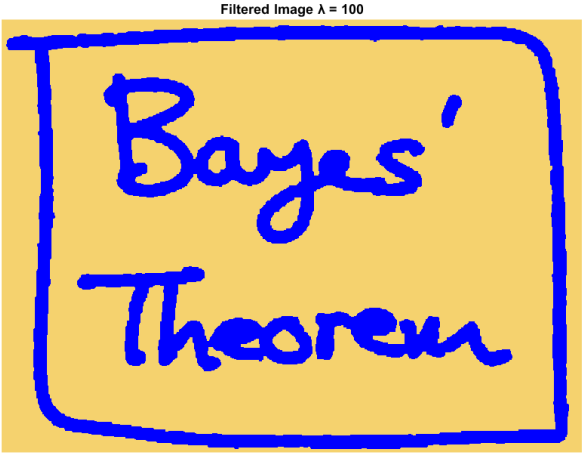
Filtered Image λ = 10



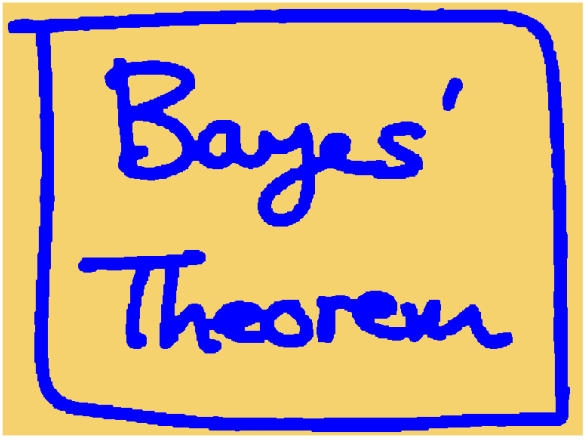
Filtered Image λ = 50



Filtered Image λ = 100



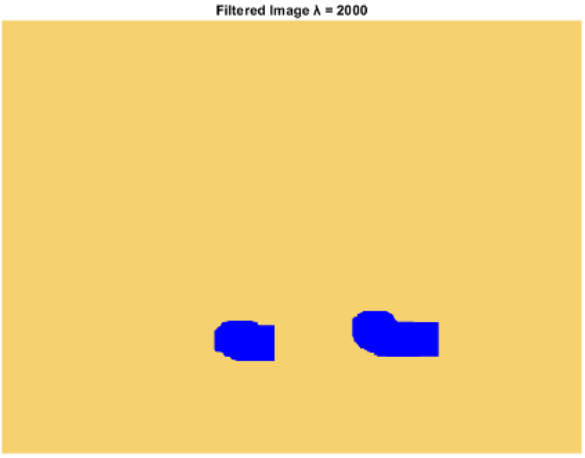
Filtered Image λ = 200



Filtered Image λ = 500

Filtered Image λ = 1000

Filtered Image λ = 2000



Part 2: Depth from Rectified Stereo Images

Implementation

In this part, we are asked to generate disparity map and then run MRF optimization on a pair of rectified images. You can run **`p2_depth_rectified_reverse.m`** to see what will happen if the disparity searching orientation is reversed.

Also, instead of setting all the weights in the directed graph as 1, I used a graph weighted by the distance between the color of the source node and target node. This weighted graph is actually used in the coming parts. Run **`p2_depth_rectified_weighted.m`** to see the difference.

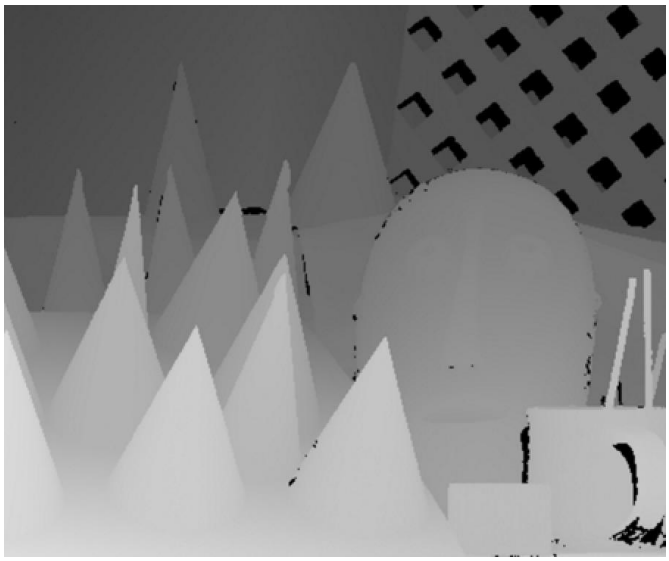
Results

The original image pair and the ground truth:

Original Image (L)

Original Image (R)

Ground Truth

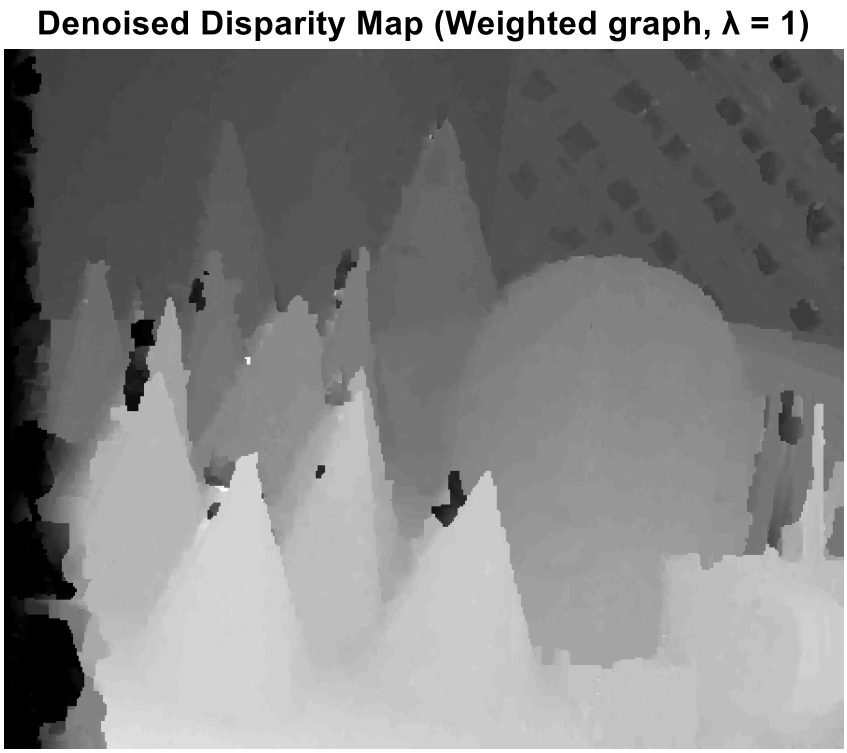


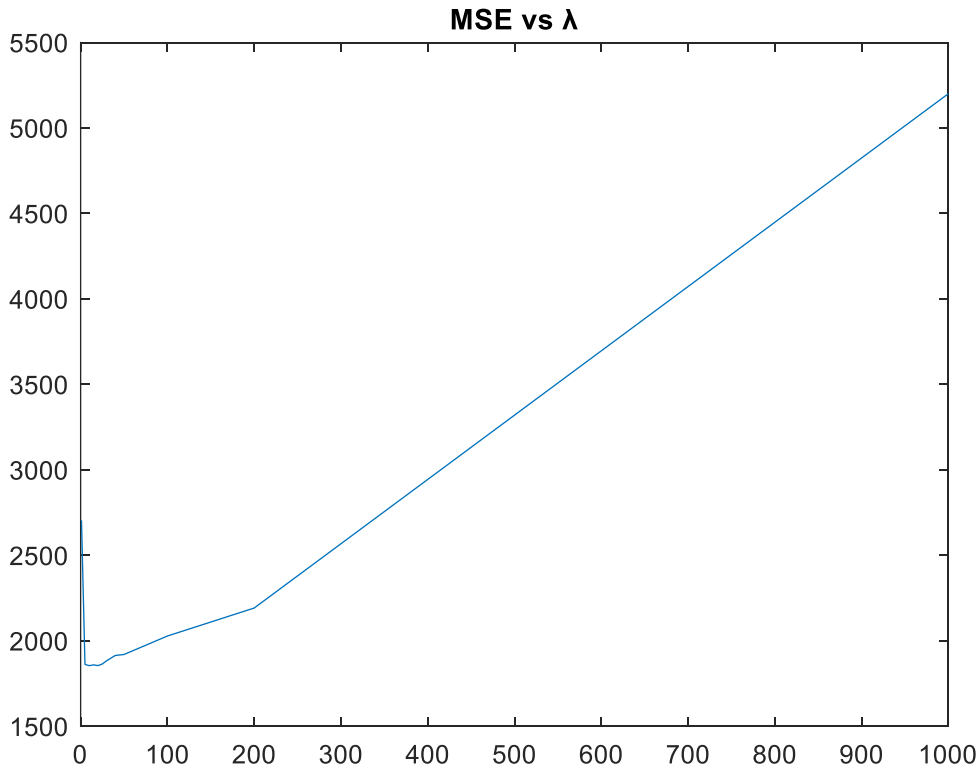
Original Disparity Map (RIGHT-LEFT)

Denoised Disparity Map (Unweighted graph, λ = 10)

Original Disparity Map (LEFT-RIGHT)

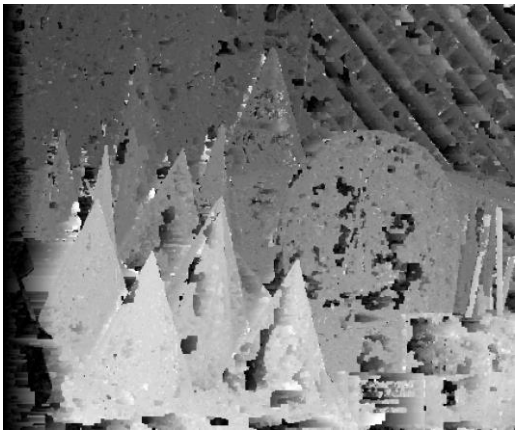
Denoised Disparity Map (Unweighted graph, λ = 10)





The MSE between result and ground truth is shown above. The results are shown below:

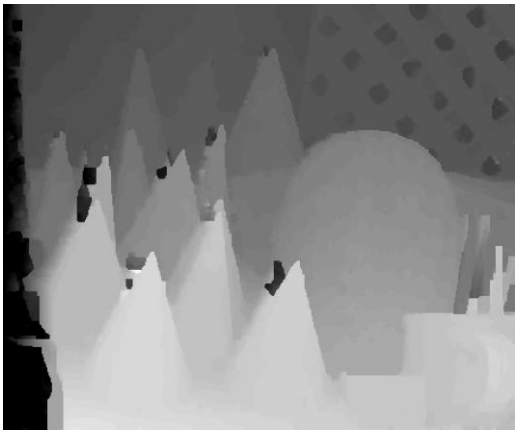
Denoised Disparity Map (Unweighted graph, λ = 1)



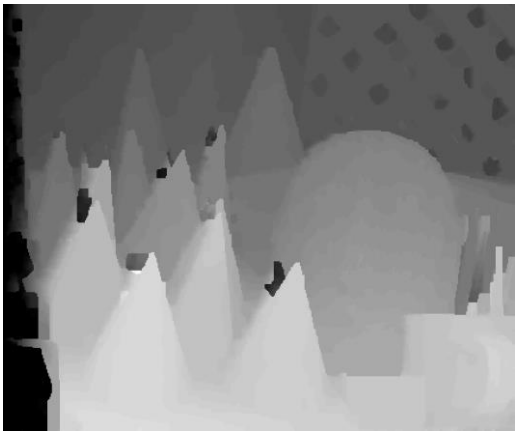
Denoised Disparity Map (Unweighted graph, λ = 5)

Denoised Disparity Map (Unweighted graph, λ = 10)

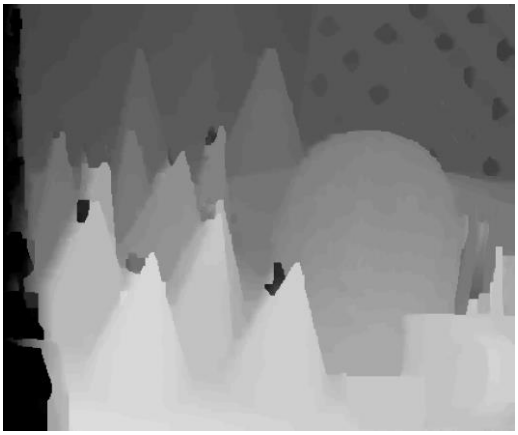
Denoised Disparity Map (Unweighted graph, λ = 15)



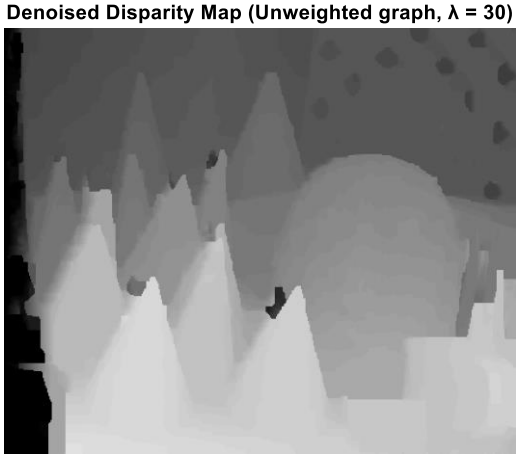
Denoised Disparity Map (Unweighted graph, λ = 20)



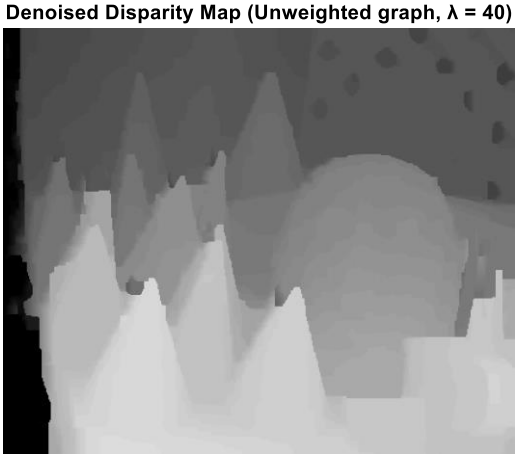
Denoised Disparity Map (Unweighted graph, λ = 25)



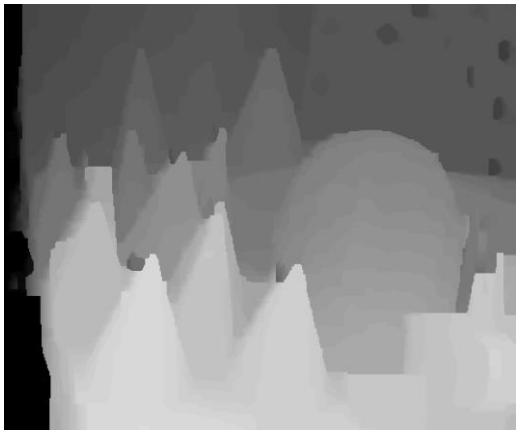
Denoised Disparity Map (Unweighted graph, λ = 30)



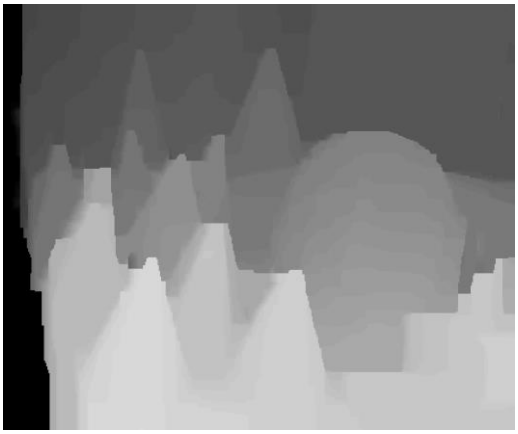
Denoised Disparity Map (Unweighted graph, λ = 40)



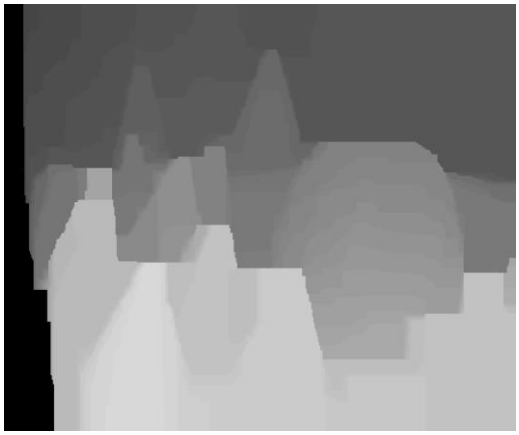
Denoised Disparity Map (Unweighted graph, λ = 50)



Denoised Disparity Map (Unweighted graph, λ = 100)



Denoised Disparity Map (Unweighted graph, λ = 200)



Denoised Disparity Map (Unweighted graph, λ = 1000)

Part 3: Depth from Stereo

Implementation

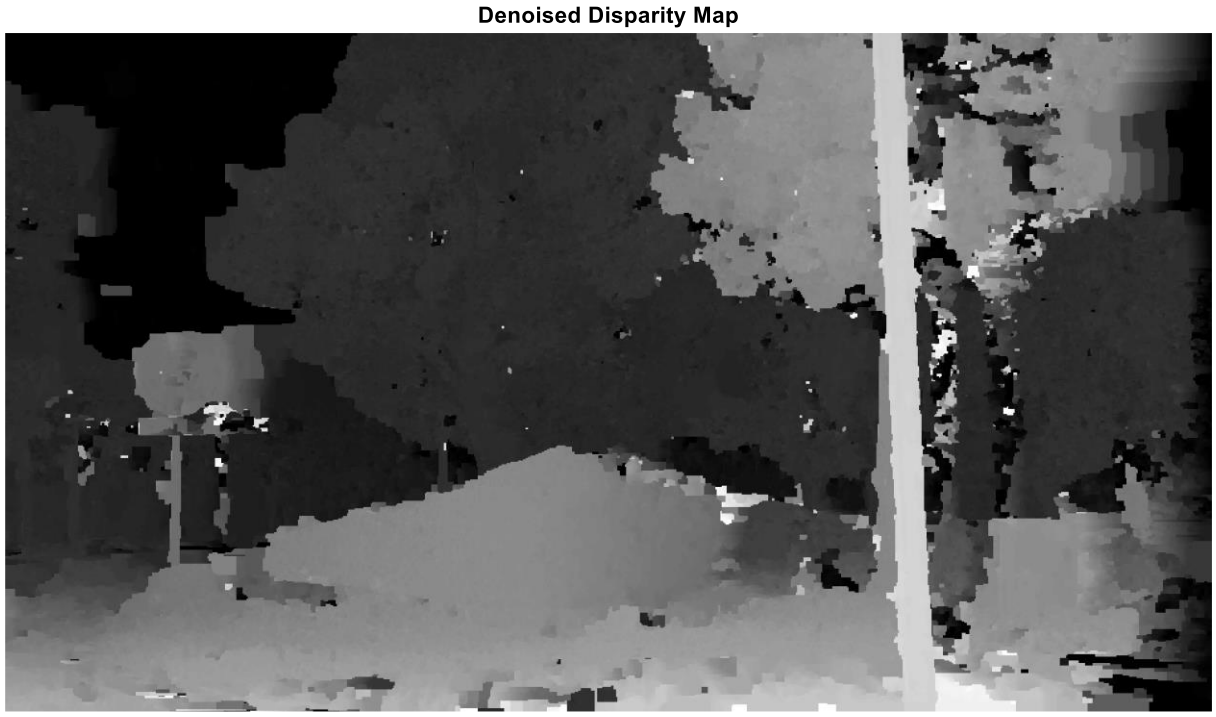
In this part, the parameters in the scripts are **exactly the same** as the in the thesis *Computer Vision Models, Learning, and Inference.* The details of the algorithm are finely commented in the scripts.
Also, the disparity map construction is performed in both orders to test the robustness of the algorithm.

Results

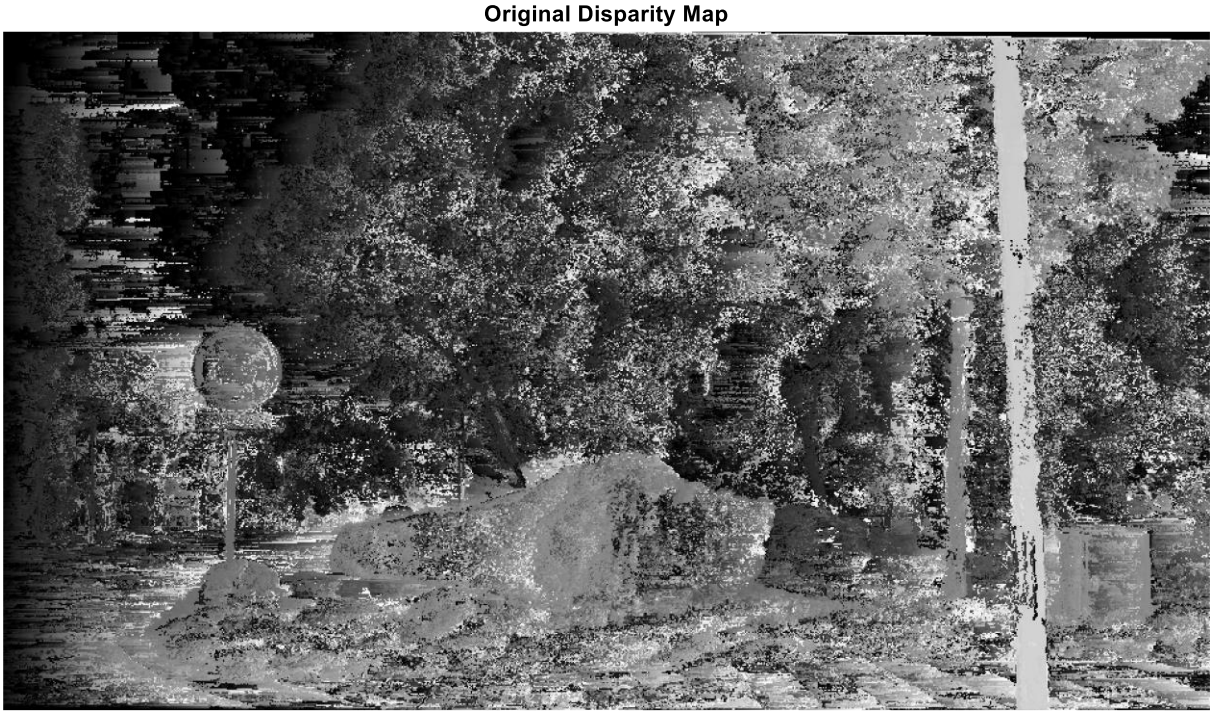
Original Image (L)

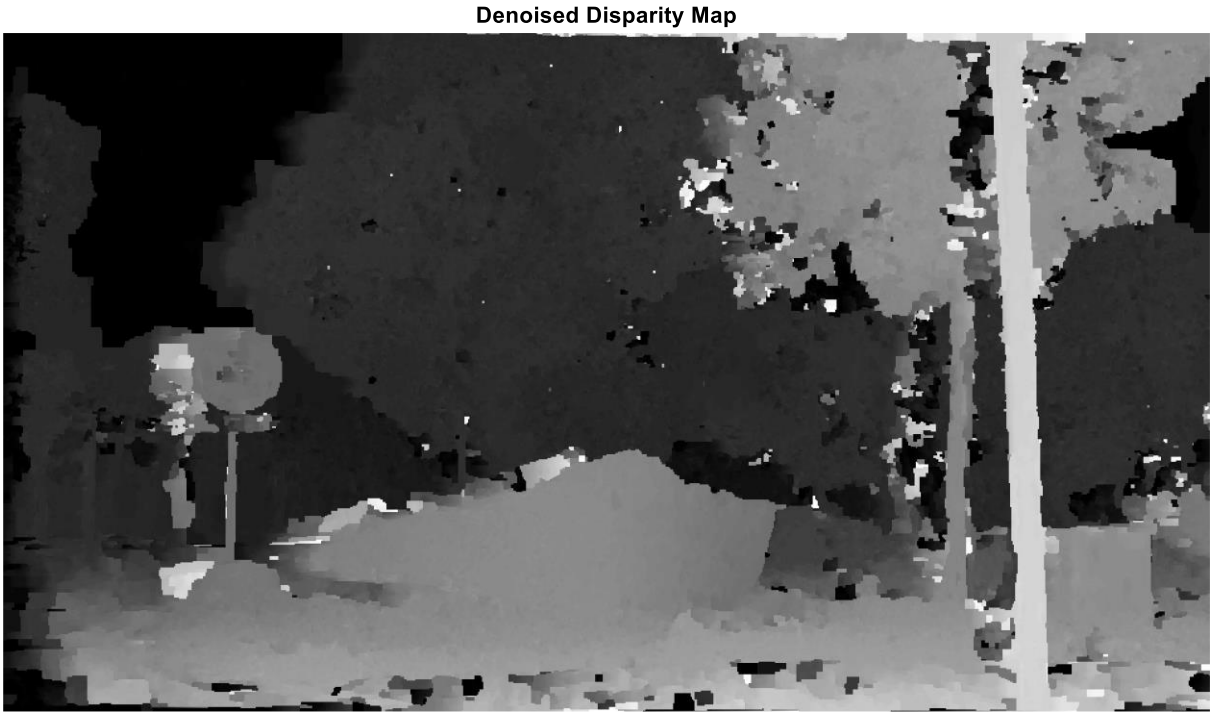
Original Image (R)

Search from left to right:



Search from right to left:





Part 4: Depth from Video -- Basic

Implementation

In this part, rather than just constructing disparity map from an image pair and then run graph cut like in part 3, we need to implement disparity initialization and bundle optimization.

Parameters:
Number of neighboring images: 30 (left + right)
σ_d: 2.5
w_s: 2 / (d_{max}-d_{min})

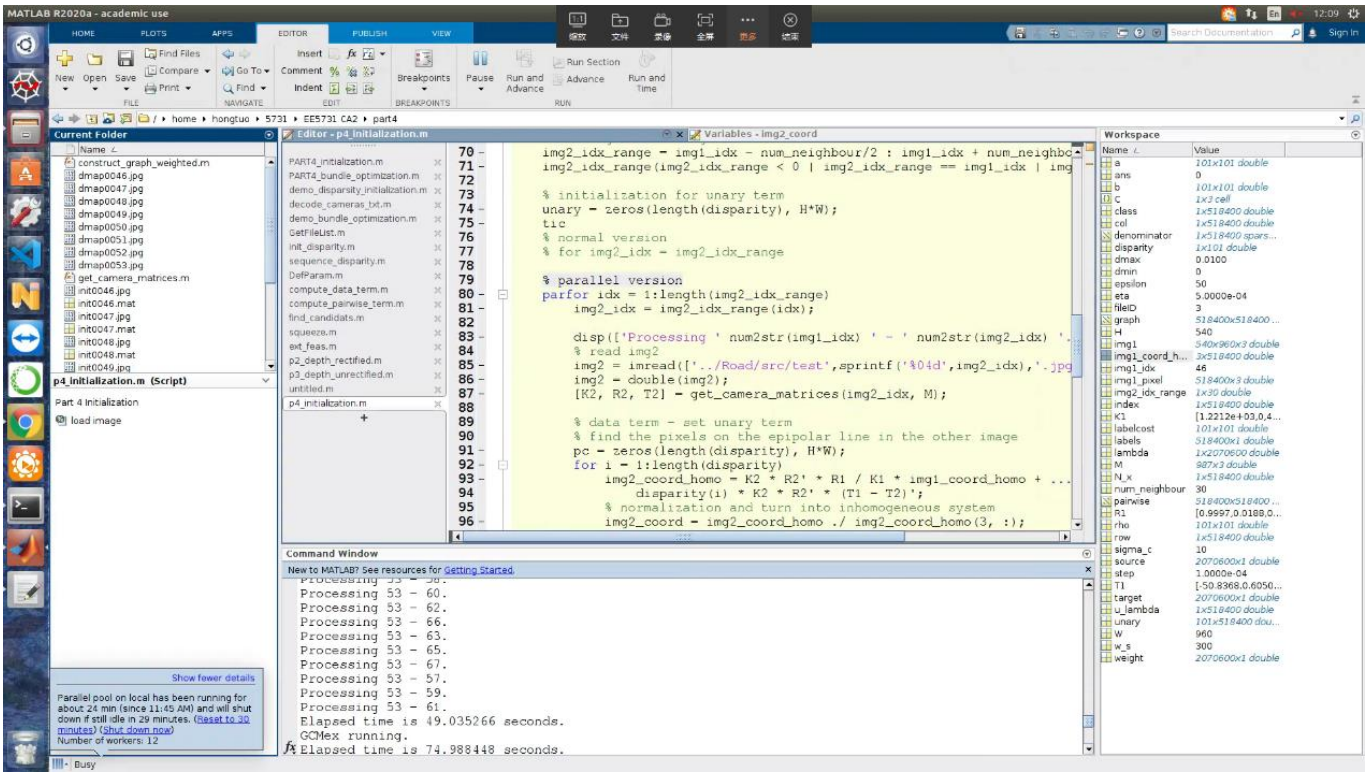
Results

1. Final Results

I would love to give some further illustration and analyzation on each process of this part, including disparity initialization, bundle optimization, and the outcome of extra iterations of bundle optimization.

2. Parallel Computation

MATLAB parallel pool is used to speed up computation. On a PC that has Intel Xeon E5 2697 v4 installed, with 18 cores working at 2.30GHz, parallel iteration can be much faster. Below is a screenshot showing the parallel version of the script is running on 12 workers assigned by MATLAB, when doing disparity initialization. The time to calculate the disparity of 30 pairs of images is less than 50s and the total time used on 1 image is about 2 minutes.



The parallel version of the code is provided with comments. If you need to output each step like I did in the analyzations below, please use for instead of parfor, or follow the instructions of MATLAB to modify. If not, then I recommend you use parfor and comment out the displaying part like in the second image below.

```
% parallel version
% to use parallel pool please comment out the preview output pa
parfor idx = 1:length(img2_idx_range)

% serial version
for idx = 1:length(img2_idx_range)

    img2_idx = img2_idx_range(idx);
    disp(['Processing ' num2str(img1_idx) ' - ' num2str(img2_id
    % read img2
    img2 = imread(sprintf('../Road/src/test%04d.jpg',img2_idx))
    img2 = double(img2);
    [K2, R2, T2] = get_camera_matrices(img2_idx, M);

    % data term - set unary term
```

```
        unary = unary + pc;

%         % view pc - for analyzation
%         unary_tmp = 1 - pc ./ max(pc);
%         [~, index] = min(unary_tmp);
%         class = index - 1;
%         dmap = reshape(class, [W, H])';
%         imwrite(mat2gray(dmap), sprintf('%04d%04d.jpg', img1_idx,
%         figure
%         imshow(uint8(dmap), [min(class), max(class)])
%         title(['pc' sprintf('%04d%04d.jpg', img1_idx, img2_idx)])

%         % view unary - for analyzation
%         unary_tmp = 1 - unary ./ max(unary);
%         [~, index] = min(unary_tmp);
%         class = index - 1;
%         dmap = reshape(class, [W, H])';
%         imwrite(mat2gray(dmap), sprintf('unary%04d%04d.jpg', img1
%         figure
%         imshow(uint8(dmap), [min(class), max(class)])
%         title(['unary' sprintf('%04d%04d.jpg', img1_idx, img2_idx
    end
    toc
```

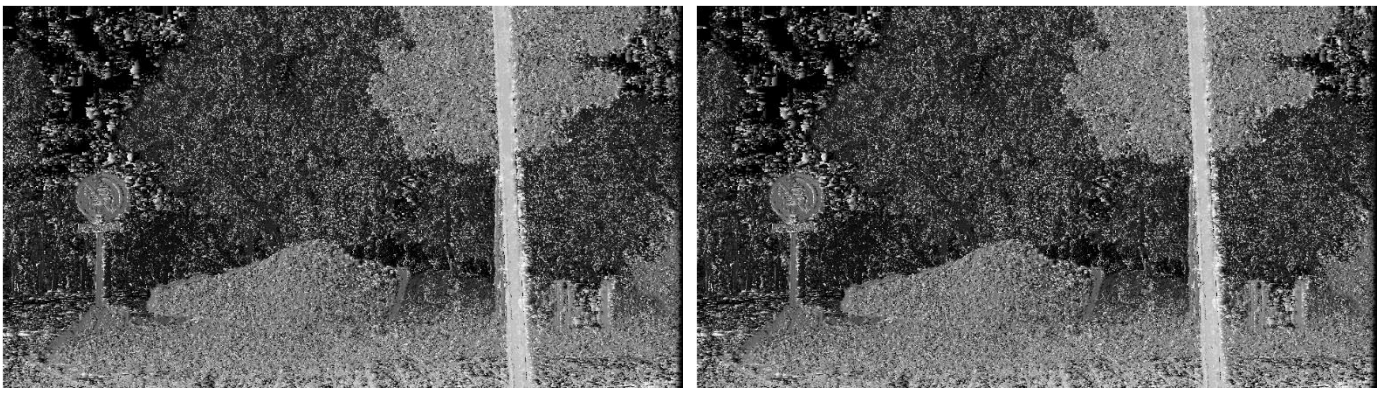
3. Disparity Initialization

Here I am showing the process of summing up each photo-consistency constraint p_c to form the final likelihood term L_{init}.

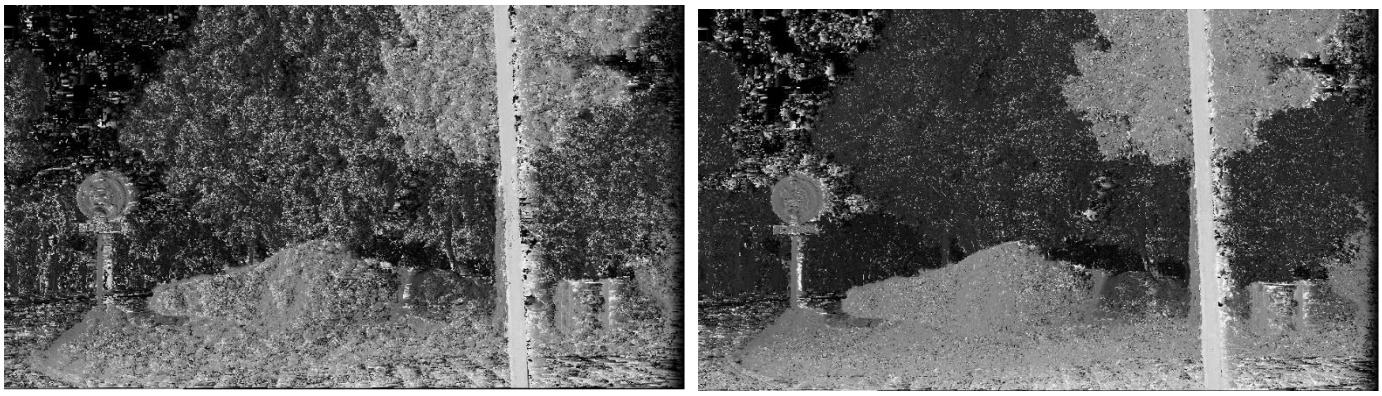
How I get the samples below:

The first column is just the disparity map of image pairs derived from unary term using [~, index] = min(unary).

The second column is using the formula $Linit = \sum_t p_c$ instead of just using p_c term. In this case, the whole process will be summing up the p_c term using the disparity from 30 neighboring images. The more it adds, the better resolution we will get.



Disparity of image pair 00-01 (left) Sum of p_c with disparity from 00-01 to 00-01 (right)



Disparity of image pair 00-03 (left) Sum of p_c with disparity from 00-01 to 00-03 (right)

Disparity of image pair 00-10 (left) Sum of p_c with disparity from 00-01 to 00-10 (right)

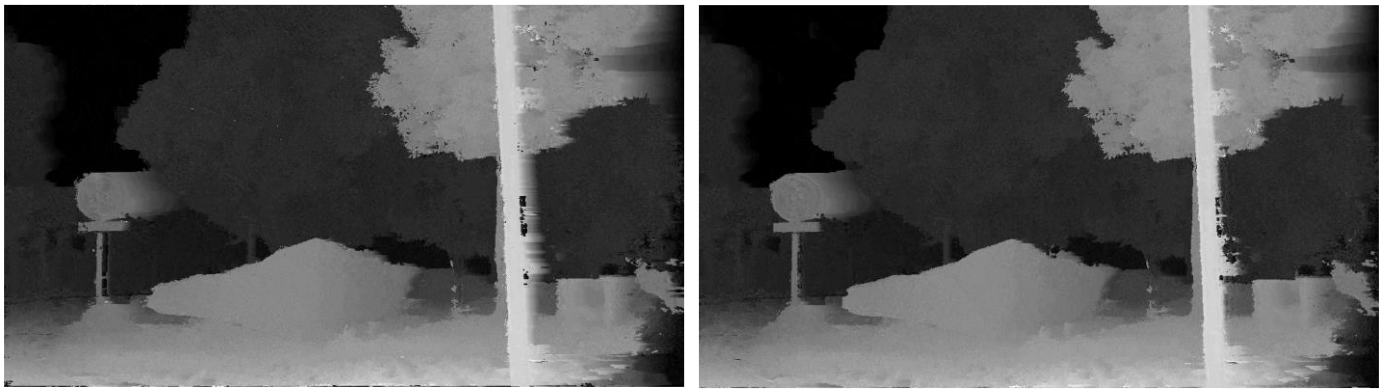
From the result we can see, although image pairs that far from each other like 00 and 10 will create a more chaotic disparity map, the sum of the photo-consistency term will be clearer and more precise. Note that the images above are just for a general view. The final disparity map is placed in folder **part4/img_original_disparity**.

For image 00 to 45, in the folder **part4/img_sum_up_process**, I saved all the disparity maps as the summing process goes by. For example, 00000005.jpg is the disparity map using the sum of p_c from image pairs 00-01, 00-02, …, 00-05. That is where the right part of the display above is from. From the process we can learn how the summing process benefits the quality of the disparity maps.

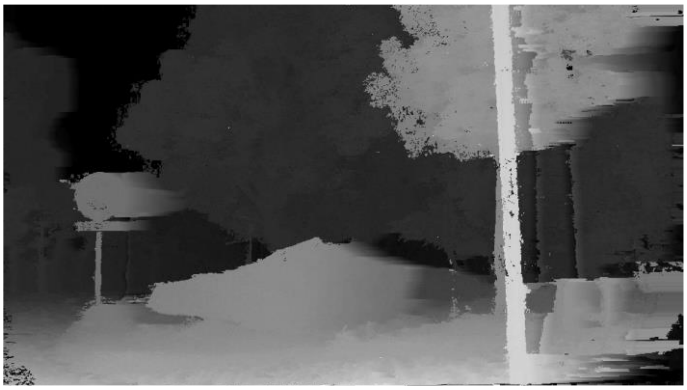
In folder **part4/init** there are all the initialized depth map after graph cut. Only part of the full result is uploaded.

4. Bundle Optimization

Below are the disparity maps derived from p_c*p_v, that is, the photo-consistency constraint weighted by geometric coherence term. Compared to the result above (without p_v term), there are less noise, and the details are clearer after bundle optimization.



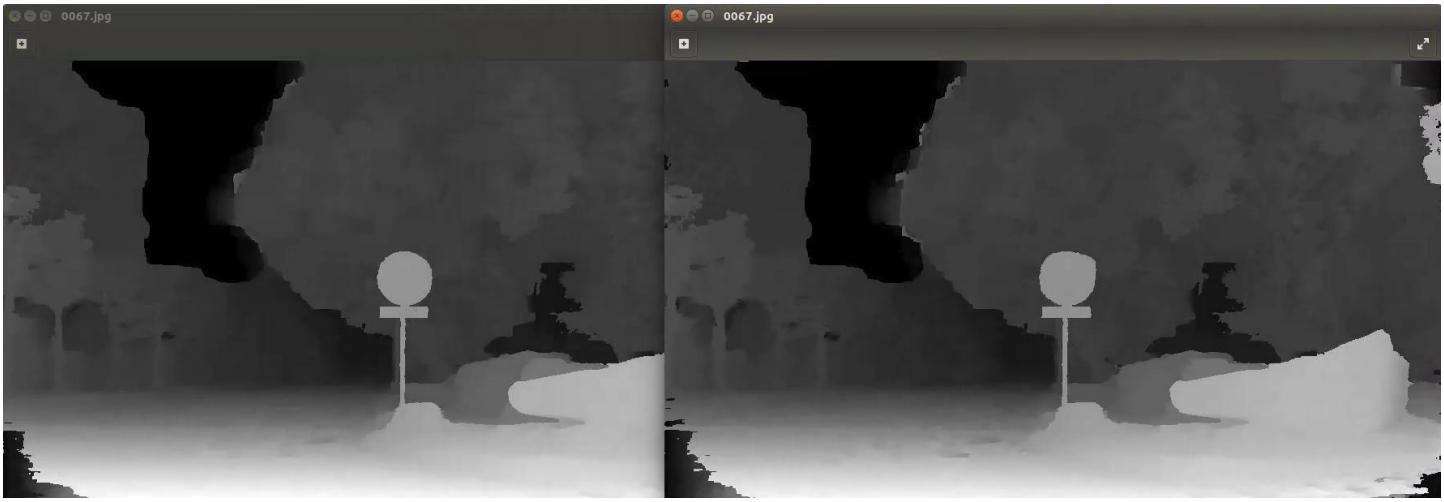
The p_c*p_v term of image pair 00-03 (left) Sum of p_c*p_v with disparity from 00-01 to 00-03 (right)



The p_c*p_v term of image pair 00-10 (left) Sum of p_c*p_v with disparity from 00-01 to 00-10 (right)

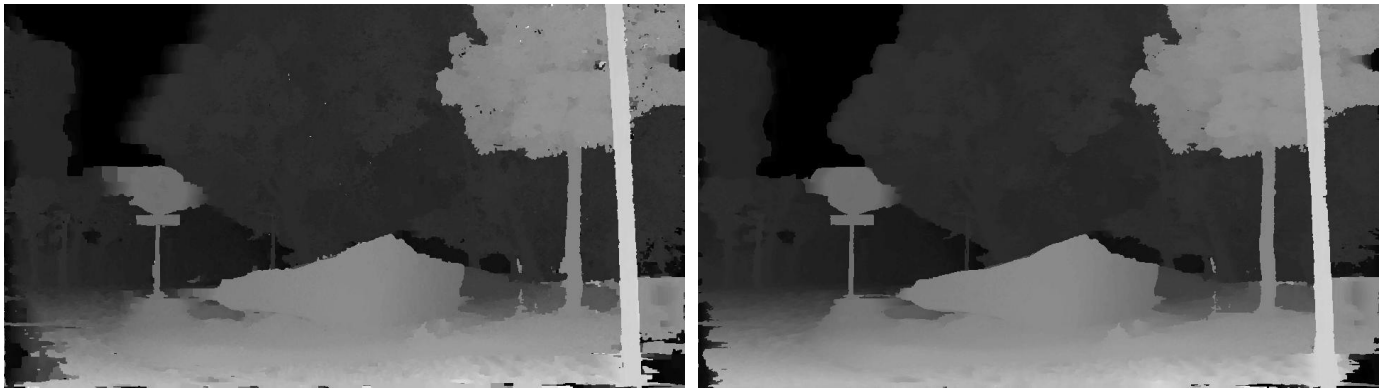
5. Difference of Number of Neighbors

Here I will show the power of extra members. In the right side, the frame 67 used 30 neighbors. In the left, I assigned 50 neighbors for it.



6. More Iterations of Bundle Optimization

In below they are the 1st and 2nd iteration result. More optimization rounds eliminate the error in the depth maps, including the noise (in the trees), the “streaking” (near to the traffic sign), and the missed patches in the corners.



Part 5: Depth from Video -- Advanced

Discussion

Issue 1: Error Recovery

In the original implementation of *Consistent Depth Maps Recovery from a Video Sequence*, the algorithm picked 30-40 neighbors for each image in every bundle optimization iteration. We already know this can lighten the flickering effect, which is basically inconsistent depths. The inconsistency comes from errors during pixel-wise disparity estimation, errors in camera matrices and so on. The small inconsistent patches can be easily rectified in only 1 iteration. But if the inconsistent areas **A. exist in many continuous frames** or **B. are very large**, or **both A and B**, then we need more iterations, leading to time-consuming computation. What's worse, most of the correct images will hardly improve in extra iterations, while the "wrong" frames are refined in a very slow pace. Thus, we need better strategies when picking neighbors. I proposed 2 ways to select neighbors when we need to handle initialization errors.

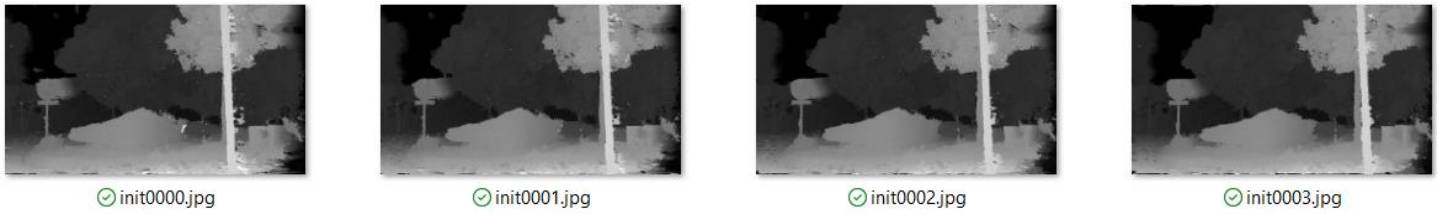
Plan 1: Random Neighbor Selection
Plan 2: Reliable Neighbor Voting

Issue 2: Feature-based Matching vs. Pixel-wise Matching

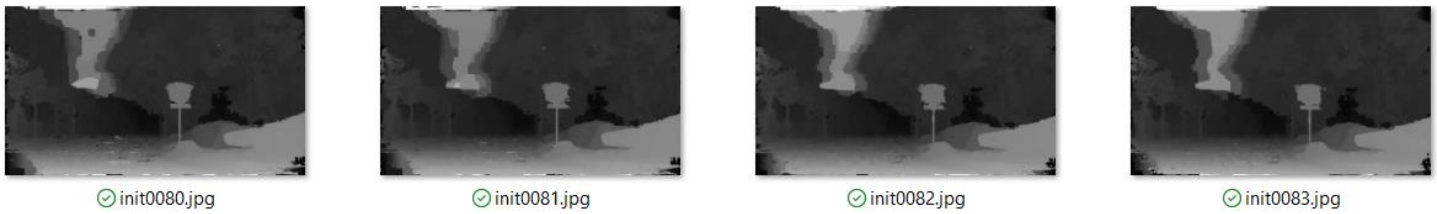
(still working on it)

Error Recovery - Plan 1: Random Neighbor Selection

Here I prepared an image sequence that have some frames wrongly initialized. The other images are the same as before, but the image **No. 80 ~ 90** have large inconsistent patch in the sky. Here shows a part of the initialized images:



"The Good"

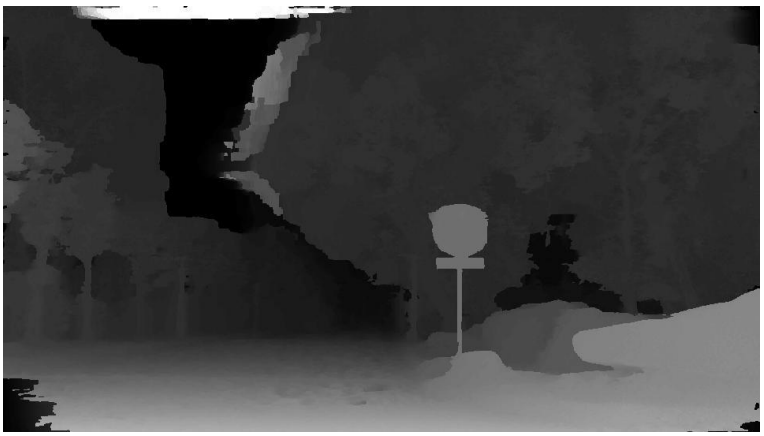


"The Bad"

↓

Run bundle optimization for 2 iterations.

↓



"The Ugly"

From the result above, after 2 iterations, the large gray patch in the sky is getting smaller and smaller. But since the images are optimized mainly using the information from neighboring images, the mistakes are hard to get totally removed.

By using Random Neighbor Selection, after only 1 iteration with 30 neighbors, the image No. 84 look like this:

Error recovery by Random Neighbor Selection

Let us look at who is selected as the neighbor of image No. 84 this time:

```
>> img2_idx_range

img2_idx_range =

  Columns 1 through 15

    11    34    18    26    33    57     7   121   125    65    64    44   116    47    15

  Columns 16 through 30

    98    49    30    50    12    16   113   119    68   134    27    41    93     2     5
```

Error Recovery - Plan 2: Reliable Neighbor Voting

Although Random Neighbor Selection can alleviate continuous initialization error, it has several drawbacks:

1. It is based on random selection, so the stability is not guaranteed.
2. We hope the number of neighbors on left and right are balanced.
3. We still need to select as many neighbors as before, making the computation time-consuming.
4. Last but the most important: we might still select "bad" images as our neighbors.

In this part, I did some interesting research. My goal is to construct **an initialized image quality indicator**, so that we can know which image is more "correct" and "reliable", which means this can be set as a good reference for others; and which image is more "wrong", which means it should be fixed.

Here comes the idea of **Reliable Neighbor Voting**:
When the initialization is done, using the initialized disparity and the camera matrices, we can match each pair of images and see how similar they are. By projecting one image to another, we can find the corresponding coordinates.

Of course, there will be inliers and some outliers that don't belong to the other image. I did an experiment on the adjacent images. For example, image No. N and N+1 has 99.99% inliers (in this case, since the camera hardly moves between each frame), and image No. N and N+10 has 96.5% inliers.

We just focus on the disparity values in the covered part. By computing the sum of the **squared Euclidian distance** of the corresponding **disparity labels** in the 2 images, we can get the **inconsistency value** I_{ij}. This is what I called "votes".

Let's first discuss what we expect from the calculation of "votes". A good image i will give smaller inconsistency value I_{ij} to its neighbor j if they match each other in the covered part. If j has big error patch, then j will get larger vote I_{ij} from i. But also, a bad image i will also give high votes to those that are different from i. Since there are more correct images than the corrupted ones, finally bad images will **receive** more inconsistency votes. Also, they will **send** more votes to their neighbors since they themselves are the abnormal ones. Combining these two metrics, we hope we can identify the "good" frames against "bad" frames.

$$\text{Sent inconsistency votes of frame } i \ = \sum_{j \in N_i} I_{ij}$$

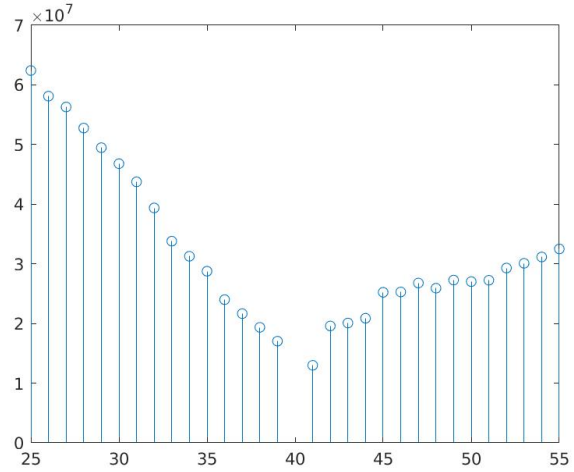
$$\text{Received inconsistency votes of frame } j \ = \sum_{i \in N_j} I_{ij}$$

$$I_{ij} = \sum_{x,y \in inliers} [disparity\_label\_of\ \_i(x,y) - disparity\_label\_of\ \_j(x,y)]^2 * \frac{inliers}{H * W}$$

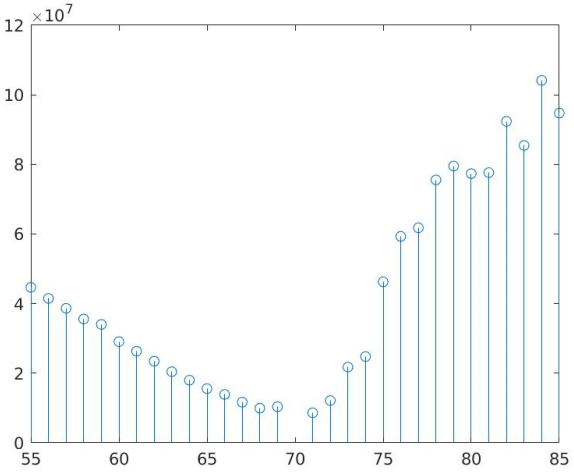
$$N_i = \#neighbors\_of\_i$$

$$inliers = \#coordinates\_that\_lie\_inside\_j\_when\_projecting\_from\_i$$

This algorithm is very tractable and fast. It only takes less than 15s for 1 image to vote for its 30 neighbors, even without using parallelism. For each image, the sent votes may look like this:



This is the votes that given by image No. 40. From the initialized images (placed in folder init/initialized jpg mat), we know that 40~50 is a series of good image. No. 20~30 are good but they have little blurry parts around the stop sign is the image. This kind of errors can also be spotted by this algorithm.



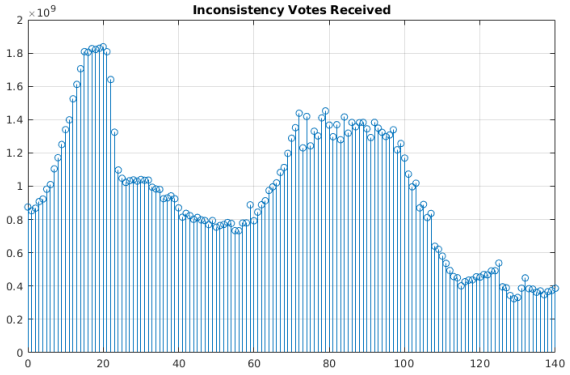
This is the votes given by image No. 70. From the initialized images (placed in folder init/initialized jpg mat), we know that 80~90 is a series of bad image. That's why No. 70 has given out high inconsistency votes to them.

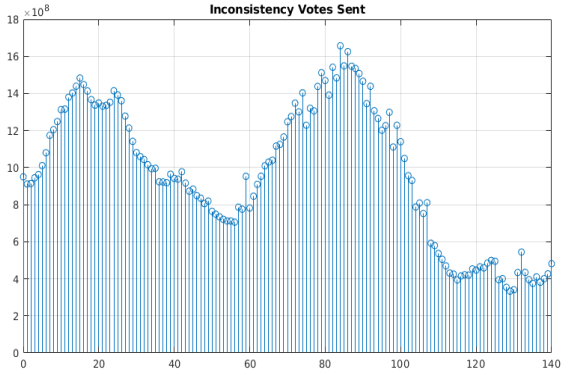
Where are the votes from? We can visualize this by showing the inconsistency map, or inconsistency image.
For each image, the inconsistency image (generated by pixel wise depth label distance) will look like this:


Left: inconsistency map of image 70 – 60. Right: inconsistency map of image 70 – 80.

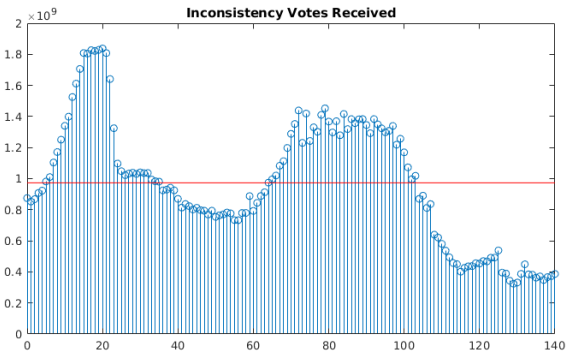
For good images 60 and 70, the inconsistency values are small. For good image 70 and bad image 80, the inconsistent patch can be seen from the map.

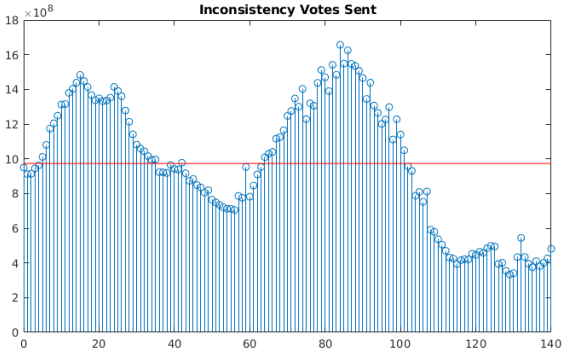
The final votes are shown below: (these can be found inside gallery folder)





We can see for bad images around No. 20 and 85, both the votes sent and received are very high. To be more clear, we can compare them with the average line:





We can see the good images are lower than the average line. This can then be used for constructing a dictionary of good frame references and bad images that need to be fixed. For good images, we can use less neighbors for bundle adjustment. For bad images, we can use more neighbors and avoid the bad ones. So we can automatically get refined as much as possible in one iteration. Goal achieved!

Rotation Invariance Feature Based Matching

Results

[1]