CMPEN 454

Computing Dense Disparities with Simple Stereo

Xuannan Su

Qi Zhao

Chen Zhou

Introduction

This Project is about computing dense stereo disparity maps patch-based matching along horizontal epipolar lines. In this project, we will use three different kinds of patch matching function: raw correlation, SSD and NCC to compare patches along the row in the left images with some candidate patches in the right images. Finally, by evaluating all three kinds of patch matching function, we can conclude which one is better than two others under different conditions such as different camera exposure time and different lighting conditions.

Implementation

We four differents main routines that can run individually. All of them will get datasets from the ./imgs folder and the datasets will be put in different folders. NCC_main.m will compute the disparity using normalized cross correlation to find the correspondences, SSD_main.m will use sum-of-square to find the correspondences, and corr_main.m will use the raw cross correlation. We set the patch size to be 5, 7 and 9, therefore for each pair of images there will be 3 output. The output will be saved at ./result folder where each dataset has its own subfolder. The output files have same naming format

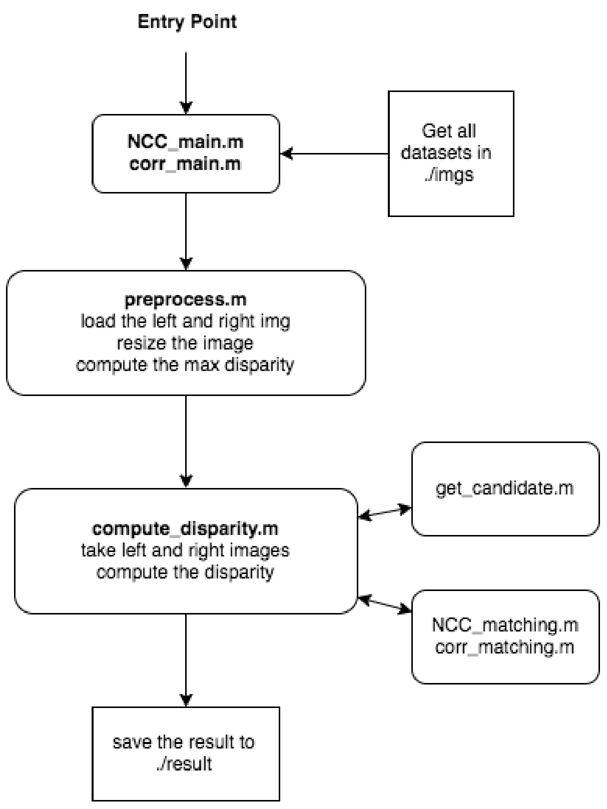
MATCHING_PATCHSIZE_CONIDTION.png

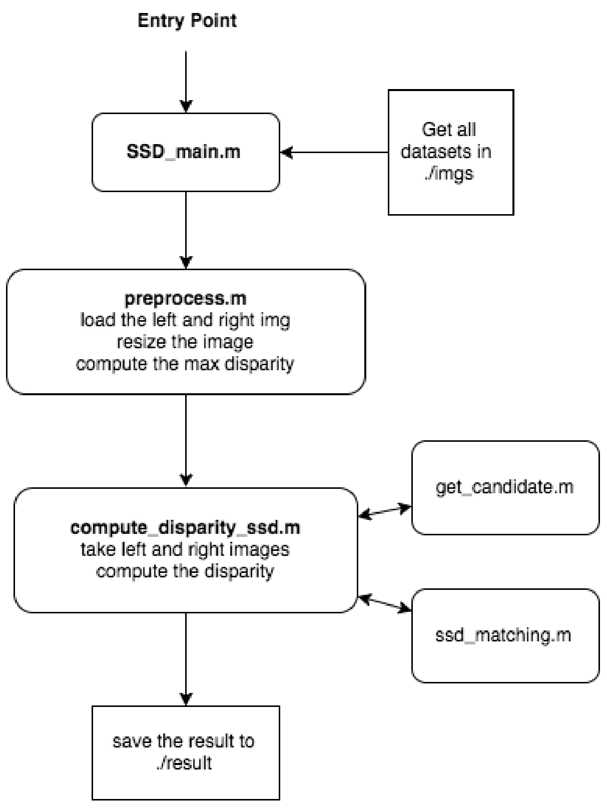
Where

MATCHING can be corr, ncc or ssd depending on the matching function used to find the correspondences.

PATCHSIZE is the patch size use to compute the disparity

CONDITION can be L, which means the right image is taken in different light condition, or E, which means the right image has different exposure. This can be ignore if the right image is taken under the same condition as the left image.


Entry Point
NCC_main.m
corr_main.m
Get all datasets in ./imgs
preprocess.m
load the left and right img
resize the image
compute the max disparity
compute_disparity.m
take left and right images
compute the disparity
get_candidate.m
NCC_matching.m
corr_matching.m
save the result to ./result


Entry Point
SSD_main.m
Get all datasets in ./imgs
preprocess.m
load the left and right img
resize the image
compute the max disparity
compute_disparity_ssd.m
take left and right images
compute the disparity
get_candidate.m
ssd_matching.m
save the result to ./result

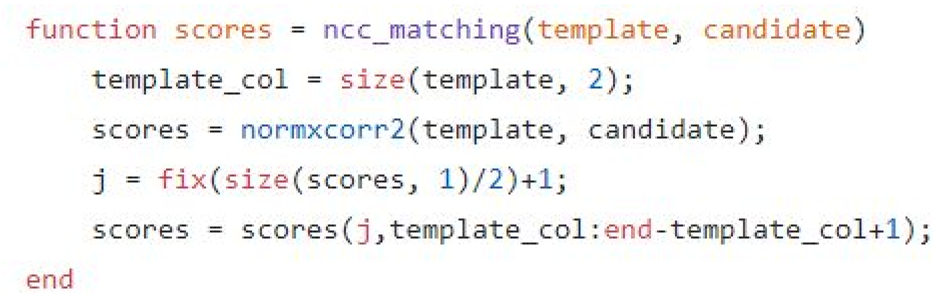
Compute_disparity

There are two versions of compute_disparity due to the different different characteristic of patch matching function. For cross correlation and ncc, we need to find the patch from the right with the highest score to compute the disparity to the path from the right. However, for sum-of-square, we need to find the patch with the lowest score. However, majority of the code has the same structure, which is to compute the disparity pixel by pixel and row by row. For simplicity, we ignore the places where the patch doesn't fit in the image. For example, given a image with size of 10 x 10, and patch with size of 3 x 3, we only compute the disparity of the pixel starting from (3, 3) to (9,9). Then we set the disparities of other pixels to be -1, which indicate no matching. And for each pixel, we check the max/min score against the threshold, if the score is lower/higher than the threshold, we set the disparity of the pixel to be -1, indicating no matching.

The threshold can affect the result. Generally, with a higher threshold for ncc and cross correlation or with a lower threshold for ssd, the result will mark more point to be no matching and the point that pass the threshold will be more accuracy. Therefore, there is a trade off between accuracy and the number of valid disparities. We cannot think of a way to choose the optimal threshold, so we run a couple experiment with different threshold and set our thresholds of ncc and cross correlation to be 0.5 and 0. For ssd, we compute the threshold according to some simple reasoning. The range of the score of ssd is between 0 and 255^2 * patch_size^2. Therefore, we set the threshold to be 0.75 * 255^2 * patch_size^2.

When using ncc to do the matching, we need to make sure that there will be no flat region, which means all the pixel of the patch has the same value. This will cause the ncc to divided by zero. To solve this problem, we simply set the disparity of a flat region to be -1, meaning no matching.

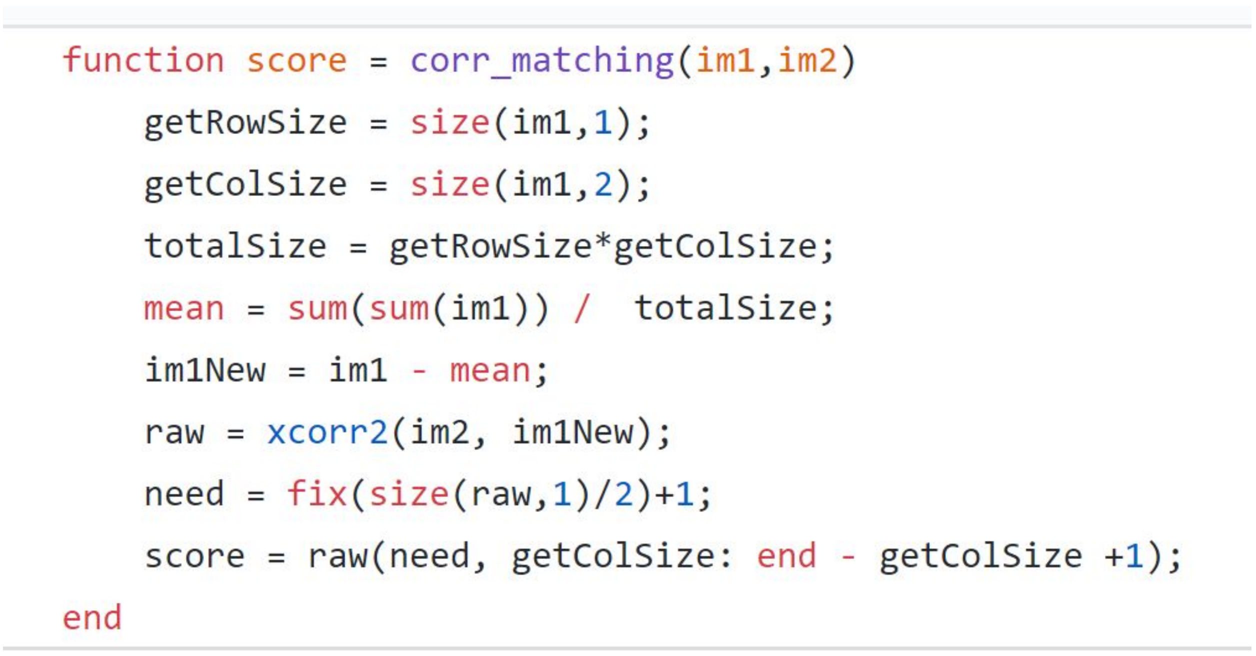
NCC_matching



```
function scores = ncc_matching(template, candidate)
    template_col = size(template, 2);
    scores = normxcorr2(template, candidate);
    j = fix(size(scores, 1)/2)+1;
    scores = scores(j,template_col:end-template_col+1);
end
```

The function takes a template(patch), and a candidate, which is an image with size that is greater or equal to the size of the patch. And it contain a possible matching patch to the template. We use the normxcorr2 function to compute the ncc scores. However, it will automatically do a zero padding, while we only want the value at which the no zero padding applied. Therefore, we only extract and return the valid scores at the middle row of the result of normxcorr2.

Corr_matching



```
function score = corr_matching(im1,im2)
    getRowSize = size(im1,1);
    getColSize = size(im1,2);
    totalSize = getRowSize*getColSize;
    mean = sum(sum(im1)) /  totalSize;
    im1New = im1 - mean;
    raw = xcorr2(im2, im1New);
    need = fix(size(raw,1)/2)+1;
    score = raw(need, getColSize: end - getColSize +1);
end
```

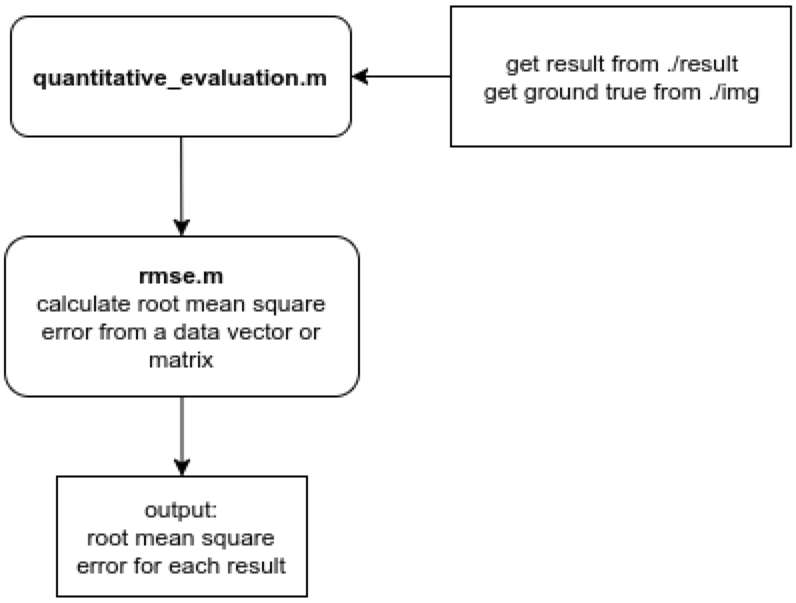
The code above is the implementation of raw correlation. Before raw correlation, we first need to subtract the mean value of the left matrix. Then, we perform the correlation between two patches. Since, we need the match score without the influence of padding, we choose the part when two patches are perfectly matched.

SSD_matching

```
function result = ssd_matching(im1,im2)
   getRowSize = size(im1,1);
   getColSize = size(im1,2);
   getColSize2 = size(im2, 2);
   temp = zeros(1,getColSize2 - getColSize + 1);
   range = 1;
   max = getColSize2;
   count = 1;
   while (range + getColSize - 1 <= max)
           g = double(im2(1:getRowSize, range:range+getColSize - 1));
           fmg = im1 - g;
           fmgsquare = fmg.*fmg;
           sum_temp = sum(sum(fmgsquare));
           temp(1,count) = sum_temp;
           count = count + 1;
           range = range + 1;
   end
   result = temp;
end
```

The code above is the implementation of SSD. Since the right patch has more columns than left patch. The result of the scores is a 1×N matrix, where N = the column size of right patch - the column size of left patch + 1. The SSD is computed by sum the square of difference between two patches. By using the while loop, we can put the score result in the matrix one by one.

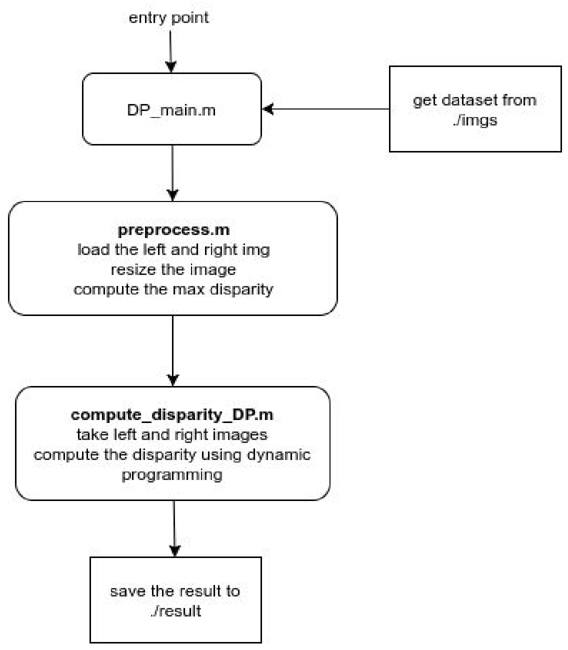
Quantitative evaluation (Extra 2)


quantitative_evaluation.m
get result from ./result
get ground true from ./img
rmse.m
calculate root mean square error from a data vector or matrix
output:
root mean square error for each result

Here we compare the disparity results we derive from various methods with the ground truth disparity, utilizing the root mean square error as the testbench. Ground truth value is extracted from the website in the form of .pfm file, which include the accurate measured disparity value for each pixel.

To pre-process the ground truth value, we make all the NaN and infinite value to -1 and then we add 1 in the next step to differentiate with the 0s that exist originally. And we use the root mean square error to evaluate and quantify the difference between the disparity we calculate before. Note that because we previously scale down the picture size ⅛ time, here we process the ground truth value to ⅛ and divide the value by 8 so that we can have a fair comparison.

Dynamic Programming (Extra 3)

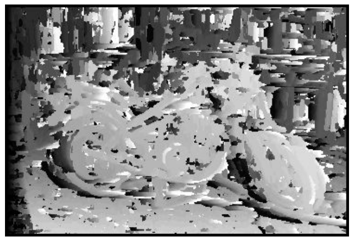
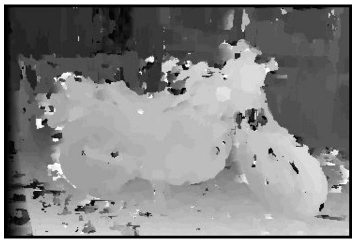
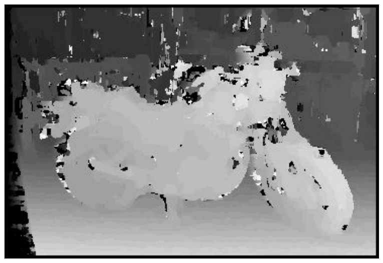
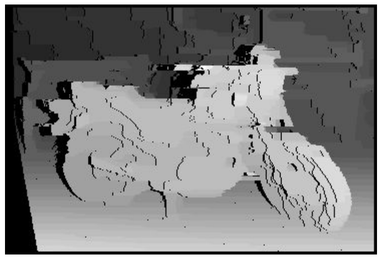

entry point
DP_main.m
get dataset from ./imgs
preprocess.m
load the left and right img
resize the image
compute the max disparity
compute_disparity_DP.m
take left and right images
compute the disparity using dynamic programming
save the result to ./result

DP_main is the main routine to compute the disparity using dynamic programming method. It will load dataset from ./imgs folder and compute the disparity using compute_disparity_DP.

compute_disparity_DP is a function that takes two images (left image and right image) and compute the disparity of each valid pixel using dynamic programing. We first compute the disparity space image DSI. Let m be the number of column in left image, DSI has a dimension of m x m, where DSI[i, j] is 1 minus the matching score of the patch at i column at the left image and the patch of j column at the right image. Then we start from the DSI[0,0] and find the lowest cost path to DSI[m, m]. There are three possible movement at any given point, moving right(occluded from the left), moving down(occluded from the right) or moving bottom right(matched). Then we can using dynamic programing to get the minimal cost. And we use backtracking to construct the optimal path and compute the disparity at each pixel.

Experimental Observation

The program take a patch from the left image and find the correspondence at the right image at the same row, then compute the disparity. It will do the same computation at each pixel and output a image of disparity. The pixel with higher disparity has higher intensity. Therefore, the brighter region of the disparity image is closer to the viewer and the dimmer region is further to the viewer.

Left Image	Right Image
Raw Correlation	SSD
	
NCC	Dynamic Programming
	

From the result shown above, we can see that the program work as expected. The closer region is brighter and the further region is dimmer, even though different algorithm has different performance.

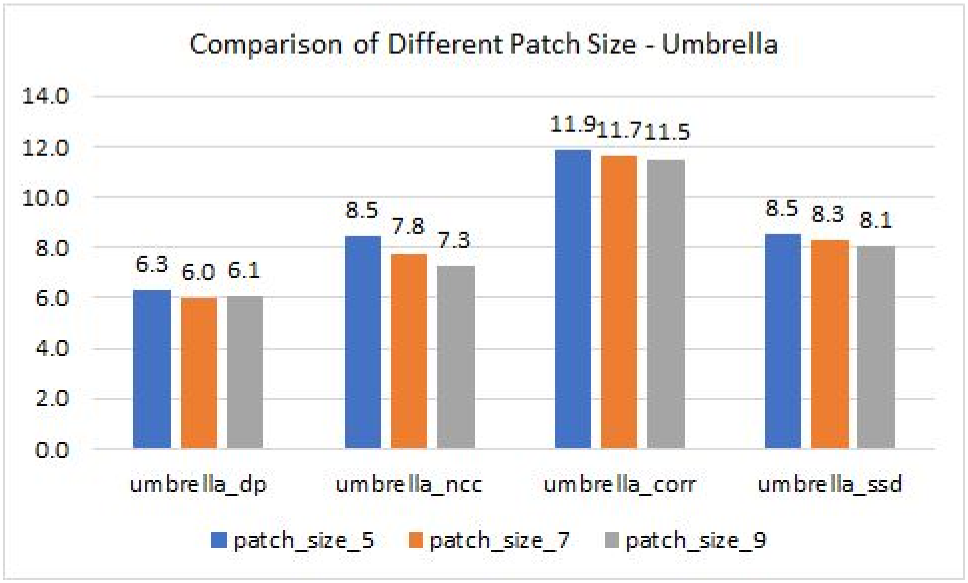
And we can also see that there are many obvious errors in the output. With raw cross correlation, even though we can see a motorcycle in the image, there are so much noise at the background, because the raw cross correlation does not do a good job at matching patch. SSD and NCC have much better outputs compared to raw cross correlation, because SSD and NCC do a better job at patch matching. However, we can see that there are noise at the flat region. The reason is that the flat region is not good for patch matching, so it will has a higher probability to produce unaccurate result. To solve this problem, we can use dynamic programming. Dynamic programming utilize the fact that most of the surface in the image is continuous, so it can handle the flat region. Indeed, the result of dynamic programming has no white dots at the background. However, we can see that there are some black region at the edge of the object in the image. This is because of the left occlusion and right occlusion.

We take motorcycle as an example to demonstrate the execution time.

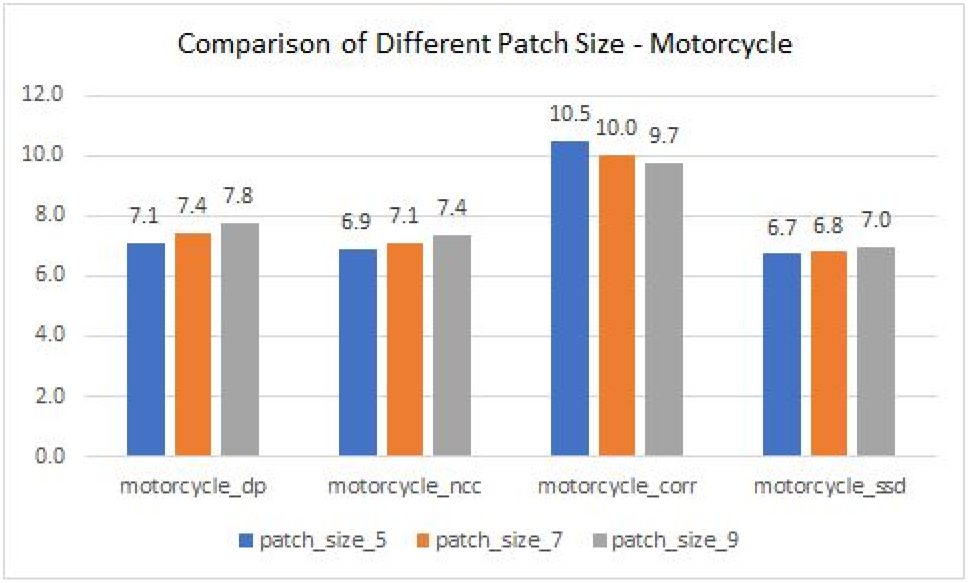
	Patch Size = 5	Patch Size = 7	Patch Size = 9
NCC	20.0s	21.4s	22.8s
DP	62.4s	76.2s	77.6s
Correlation	2.4s	3.0s	3.5s
SSD	6.3s	6.6s	6.9s

As we can see in the table, dynamic programming takes the longest time among all the methods, while it works best on the umbrella and it works similar to ssd and ncc when dealing with motorcycle. Correlation takes least time but the performance is not that great, which is a trade off between time and accuracy.

The larger the image the slower the longer it has to run. And as shown in the table above, a larger patch size also requires a longer running time.



Generally, we can conclude that smaller size has better advantages when compared to larger ones. However, the differences are not that large. And correlation method has disadvantages in performance while it is the fastest way to implement.



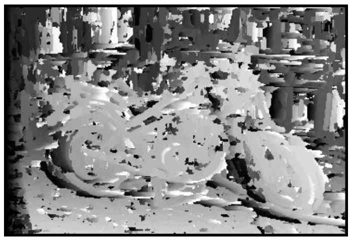
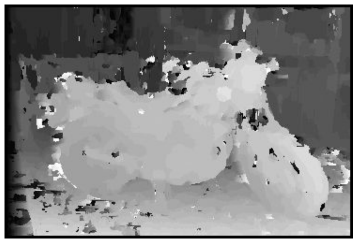
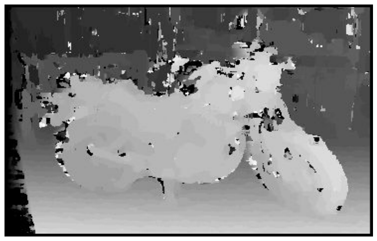
For Motorcycle, effects of different patch size on the performance varies with different analysis method.

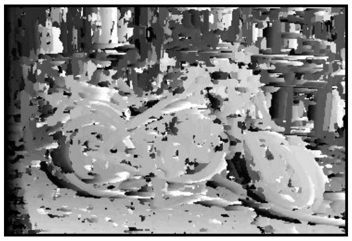
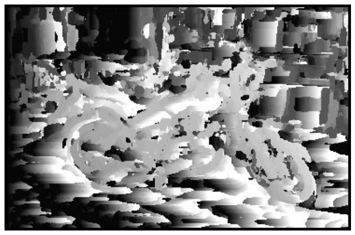
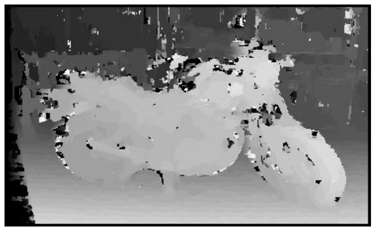
Performance under Different Condition (Extra 1)

The following image is motorcycle E image.

E:

The following table shows the result of patch size of 7:

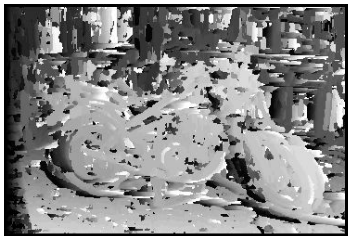
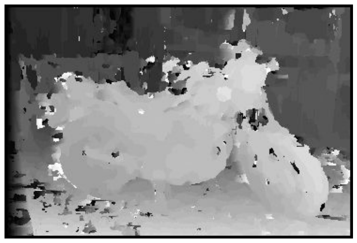
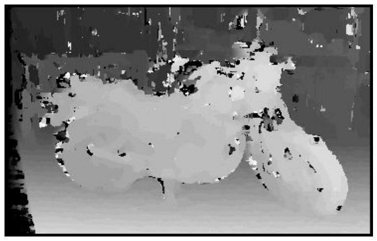
	Raw Correlation	SSD	NCC
Original			

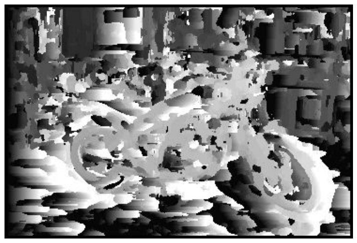
E			

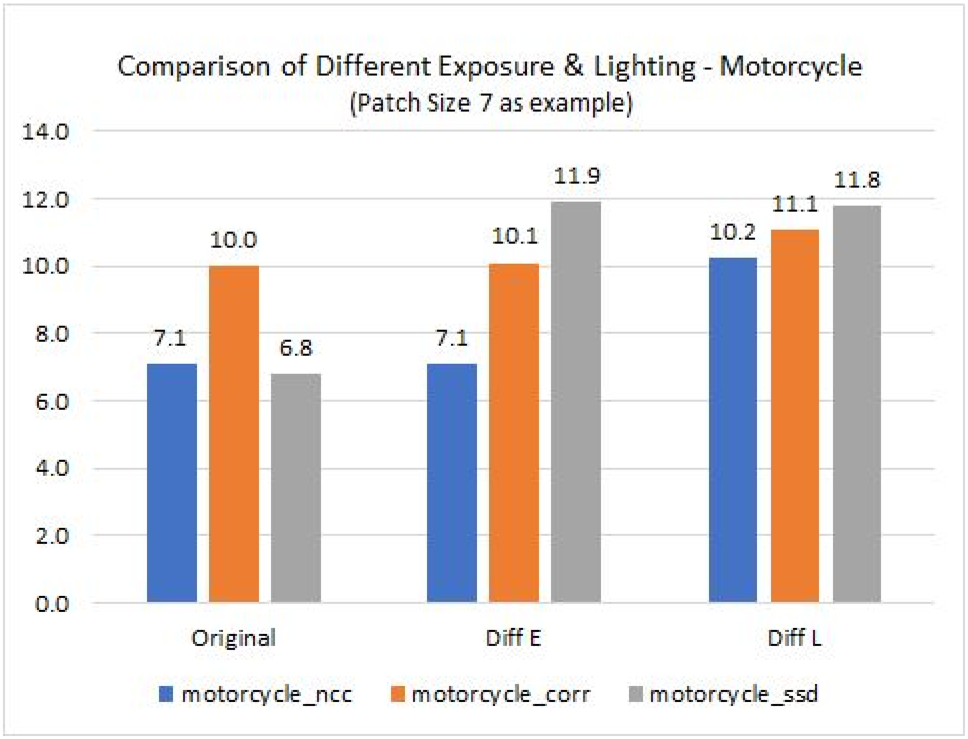
The following image is motorcycle L image.

L:

The following table shows the result of patch size of 7:

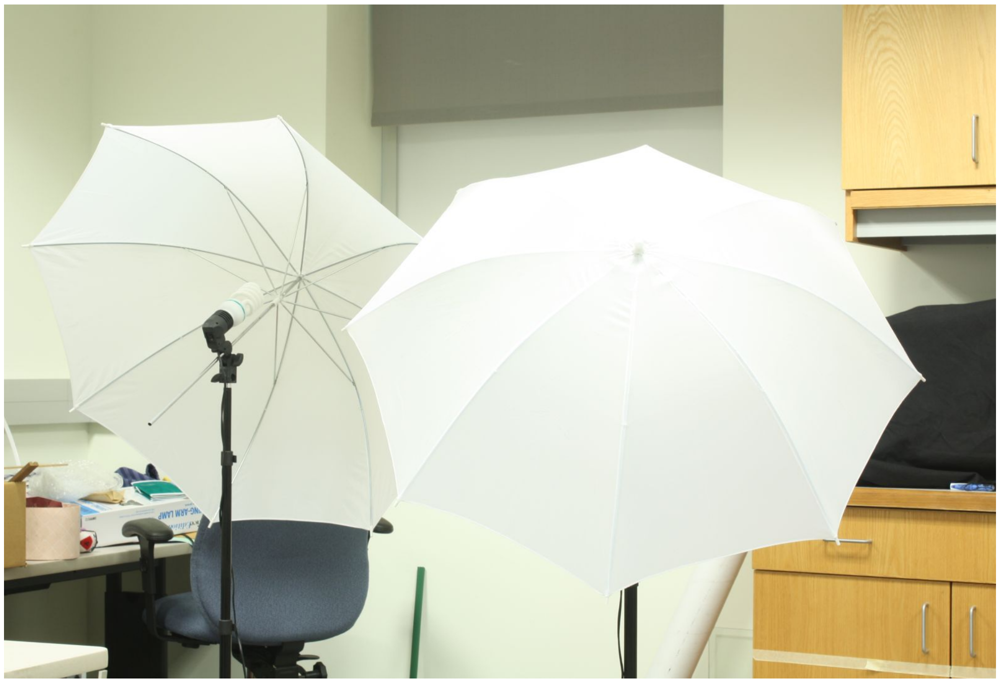
	Raw Correlation	SSD	NCC
Original			

L			

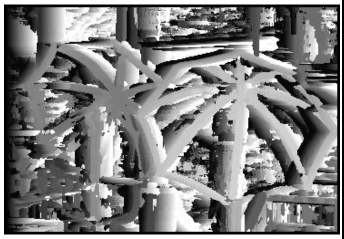
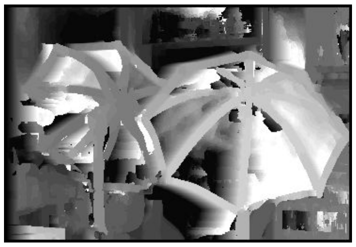
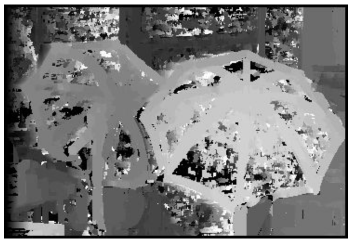
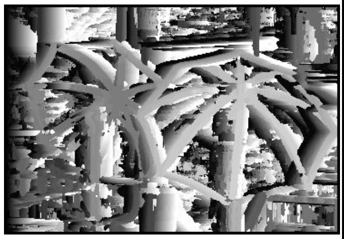
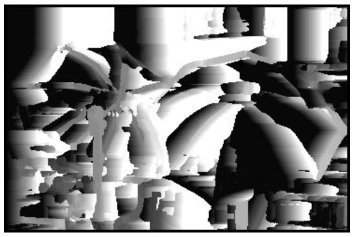
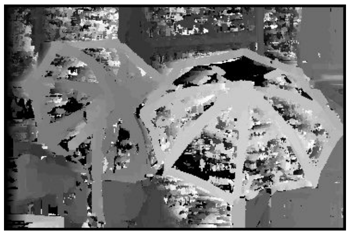

Comparison of Different Exposure & Lighting - Motorcycle
(Patch Size 7 as example)
14.0
12.0
10.0
8.0
6.0
4.0
2.0
0.0
7.1
10.0
6.8
7.1
10.1
11.9
10.2
11.1
11.8
Original
Diff E
Diff L
motorcycle_ncc
motorcycle_corr
motorcycle_ssd

The following images are umbrella E image.

E:

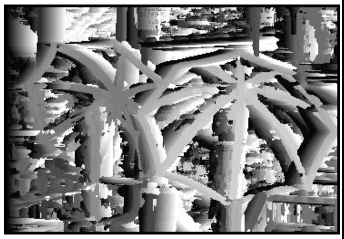
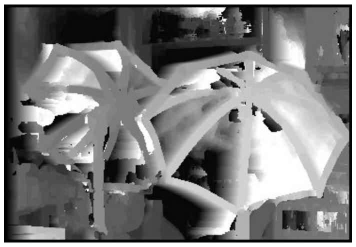
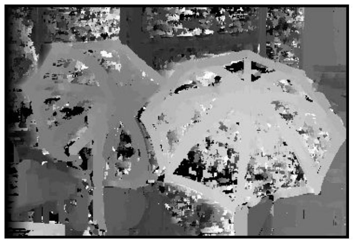
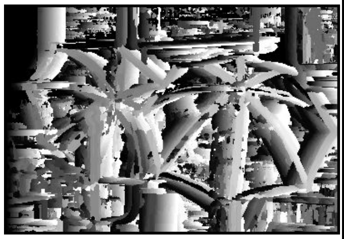
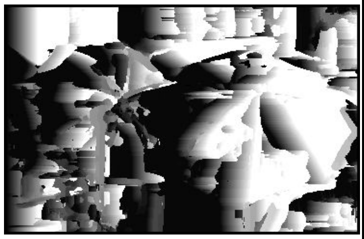
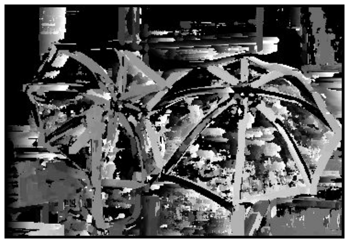


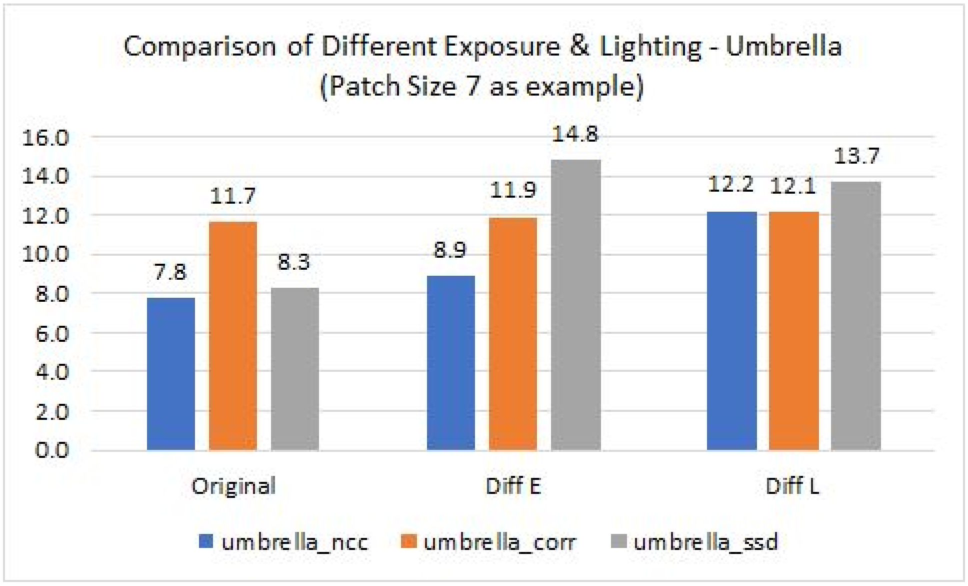
The following table shows the result of patch size of 7:

	Raw Correlation	SSD	NCC
Original			
E			

The following images are umbrella E image.

The following table shows the result of patch size of 7:

	Raw Correlation	SSD	NCC
Original			
L			



From the evaluating results, we can find that the matching scores of NCC and raw correlation under different camera exposure condition are close but the score of SSD is different. So, we can conclude that under different camera exposure condition, the matching score of SSD will changes a lot while the scores of raw correlation and NCC almost keep the same. Raw correlation and NCC are better than SSD and NCC has the best performance.

When lighting is strengthened, the matching scores of SSD, NCC and raw correlation increase. Although NCC has the lowest score, the change rate of raw correlation is the lowest . So we can conclude that under different lighting condition, raw correlation may have better performance than NCC and the change of raw correlation is most stable.

Member Contribution

Xuannan Su: pre-processing, NCC, dp

Qi Zhao: Correlation, SSD, extra problems 1,images

Chen Zhou: Extra questions 2, charts