# Computer Vision Task 3

---------Stereo Vision

131180022 姜志健

1. **Assignment**
2. [This step is optional] Setup your own stereo vision system with two cameras. Take a pair of stereo images and perform rectification. Please refer to the 3-D vision toolbox of Matlab (rectifyStereoImages, stereoParams, etc.)
3. Calculate disparity of a pair of stereo images. Use window based searching algorithm and try different window size. If you choose to setup your own stereo vision system, calculate the actual depth according to the length of the baseline.

**2.Process**

**2.0 Download pictures and test them**

Download sample pictures from <http://vision.middlebury.edu/stereo/data/>. There some notes below:

* Download the 2005 datasets if you want to figure out the depth from disparity because the 2005 dataset provide the baseline length and focal length. For the 2005 dataset, the focal length is 3740 pixels, and the baseline is 160mm.
* There are three kinds of resolutions, full size, half size and third size. Notably, in the half-size and third-size versions, the intensity values of the disparity maps need to be divided by 2 and 3, respectively. To map the disparities into 3D coordinates, add the value in "dmin.txt" to each disparity value, since the images and disparity maps were cropped.

Just to be sure, match two pictures by Harris corner to ensure they have be calibrated.

In this experiment, one of the results is shown below,



Obviously, they have been calibrated.

**2.1 Match each pixels with SSD**

There are two ways available:

* Use *for* loop to traverse each pixel to figure out SSD.

Advantage: Simple

Disadvantage: It takes too much time.

* Use *parfor* to traverse each pixel to figure out SSD.

Advantage: Save some time

Disadvantage: It still takes too much time and the use of *parfor* has many limitations.

* Use the function *imfilter* to figure out SSD.

Advantage: It will saves a lot of time because *Matlab* is good at matrice calculation.

Disadvantage: It brings some difficulty in programming.

In order to save time and practice my programming ability, I decide to adopt the third method in this experiment.

**2.1 Write function to extend images**

If I directly use the function imfilter to figure out the sum of pixel’s neighbourhood’s intensity, the target pixel is at the upper-left corner of the nieghbourhood actually, which is not consistent with the definition of the neighbourhood.

In order to solve this problem, we must extend the pictures a little( exactly around the half size of neighbourhood). And then the use of imfilter would obtain results while the target pixel is in the centre of the neighbourhood.

Sample codes:

function [ imgExtended ] = ImgExtend( img, offset[[1]](#footnote-1) )

%UNTITLED Summary of this function goes here

% Detailed explanation goes here

[heightImg, lengthImg, dimImg] = size(img);

if dimImg > 1

error('Inputed Image exceeded dimension limit');

end

imgExtended = zeros(heightImg + offset, lengthImg + offset);

imgExtended(offset + 1: heightImg + offset, offset + 1: lengthImg + offset) = img(:, :);

end

**2.2 Match pixels with SSD**

I write this part into a function:

[ matchedPointImg1, matchedPointImg2 ] = ssdMatch( img1, img2, sizeNeighbourhood )

After extending images, *imfilter* can figure out the sum of each pixel’s neighbourhood’s intensity with the target pixel in the centre of the nieghbourhood. Assume two extended images is called *img1Extended* and *img2Extended* perspectively. The all pixels’ SSD can be figured out in the way below:

mask = ones(sizeOfNeighbourhood);

SSD = imfilter( (img1Extended – img2Extended) .^ 2, mask);

Because of the baseline is parallel and the baseline limitation, I can just shift one of the image horizontally to figure out all possible SSD.

The detailed process is shown below.

###############################################################################

LOOP2:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LOOP1:

2.2.1 Figure out all pixels’ SSD in the way above.

Attention:

* Extend two images first.
* Crop the upper-left part of the result with the size same as the image.

2.2.2 Shift one of the image.

Assume the shift image is *img2*.

After shifting, some part of the *img2*’s intensity is 0, which is meaningless to figure out SSD while some part of the *img2* would be out of the range, which would not participate in calculating SSD and make me shift the image in the other direction again.

So, to save time and improve productivity, the shift must be circular.

The shift process is shown below.

|  |  |  |
| --- | --- | --- |
| E:\Document Files\Learning\DIP\Homework8\Project\Result\img2Move0.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\img2Move400.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\img2Move800.png |

Sample codes:

img2Move = zeros( heightImg2, lengthImg2 + img2MoveDistance );

img2Move(:, img2MoveDistance + 1: lengthImg2) = img2(:, 1: lengthImg2 - img2MoveDistance);

if img2MoveDistance >= 1

img2Move(:, 1: img2MoveDistance) = img2(:, lengthImg2 - img2MoveDistance + 1: lengthImg2);

2.2.3 Go to 2.2.1

LOOP1 END

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2.2.4 Remove the larger SSD

Compare the new SSD with the old SSD to remove the larger ones.

Details:

* Of course, using *for* loop to traverse each element is feasible but it would cost a lot of time. So, I use the *Mask* to accept or reject element.
  + Step1: *Errors = SSDOld(:, :, 1) - SSDNew(:, :, 1);*
  + Step2: create *newMask* by turning the elements over 0 in *Errors* into 1.
  + Step3: create *oldMask* by turning the elements over 0 in *Errors* under 1 into 1.
  + *Step4: SSD (:, :, 1) = newMask .\* SSDNew(:, :, 1) + oldMask .\* SSDOld(:, :, 1);*
  + Step5: Save the move distance for each element in *SSD* for the future use.

Sample codes:

OldNewErrors = SSDimg1Img2Old(:, :, 1) - SSDimg1Img2New(:, :, 1);

newMask = zeros(heightImg2, lengthImg2, 1);

oldMask = zeros(heightImg2, lengthImg2, 1);

newMask(OldNewErrors > 0 ) = 1;[[2]](#footnote-2)

oldMask(OldNewErrors <= 0) = 1;

SSD (:, :, 1) = newMask .\* SSDNew(:, :, 1) + oldMask .\* SSDOld(:, :, 1);

SSD (:, :, 2)[[3]](#footnote-3) = newMask .\* img2MoveDistance + oldMask .\* SSDimg1Img2Old(:, :, 2);

2.2.5 Go to LOOP1.

LOOP2 END

###############################################################################

2.2.6 Figure out the matched pairs.

Since the move distance has been saved, we can figure out the matched pairs easily.

Sample codes:

matchedPointImg1 = zeros(heightImg1, lengthImg1, 2);

matchedPointImg2 = zeros(heightImg2, lengthImg2, 2);

for x = 1: lengthImg1

matchedPointImg1(:, x, 1) = [1: 1: heightImg1]; % y is 1st

end

matchedPointImg2(:, :, 1) = matchedPointImg1(:, :, 1);

for y = 1: heightImg1

matchedPointImg1(y, :, 2) = [1: 1: lengthImg1]; % x is 2nd

end

matchedPointImg2(:, :, 2) = matchedPointImg1(:, :, 2);

matchedPointImg2(:, :, 2) = mod(matchedPointImg2(:, : ,2) + SSDimg1Img2(:, :, 2), lengthImg2);

**2.3 Figure disparity and depth**

2.3.1 Figure out disparity and depth according to the equation above

Sample codes

[matchedPointView1, matchedPointView2] = ssdMatch( view1Gray, view2Gray, size );

disparity = (matchedPointView1(:, :, 2) - matchedPointView2(:, :, 2));

Z = (B \* f) ./ disparity;

2.3.2 Show the result

* Obviously, not all pixels have matched pixels. So there must be errors in the result of *ssdMatch*. Since the view2 is in the left of view1:



the right *disparity* is under 0, which means we can remove all positive result and must turn the right results to their absolute value before showing them.

Sample code:

D (D > 0) = 0;

D = abs(D);

Z (Z > 0) = 0;

Z = abs(Z);

* Then, normalize the results.

D = 1 \* D / 200;

Z = 1 \* Z / 10000;

Because it is hard to figure out the max value of *disparity* and *Z* due to noise and edge, *200* and *10000* are all estimated.

**2.4 Figure out disparity and depth with different sizes of window**

**3. Experiment Result**

The picture used in this experiment has the resolution of 1110 \* 1342.

|  |  |
| --- | --- |
| E:\Document Files\Learning\DIP\Homework8\Project\view1.png | E:\Document Files\Learning\DIP\Homework8\Project\view5.png |

**3.1 Disparity**

|  |  |  |
| --- | --- | --- |
| Size | Gray | Jet |
| 5 | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize5Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize5Jet.png |
| 9 | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize9Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize9Jet.png |
| 17 | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize17Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize17Jet.png |
| 25 | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize25Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize25Jet.png |
| 33 | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize33Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\disparityOfSize33Jet.png |

**3.2 Depth**

|  |  |  |
| --- | --- | --- |
| Size | Gray | Jet |
| 5 | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize5Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize5Jet.png |
| 9 | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize9Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize9Jet.png |
| 17 | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize17Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize17Jet.png |
| 25 | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize25Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize25Jet.png |
| 33 | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize33Gray.png | E:\Document Files\Learning\DIP\Homework8\Project\Result\ZOfSize33Jet.png |

The data is [here](depth.xlsx).

Origin images is [here](Image).

**Acoording to 3.1 and 3.2,**

**smaller size of window saves more details but more noise, larger size of window saves less details but less noise, which looks better.**

**3.3 Calculate time**

|  |  |
| --- | --- |
| Size of window | Time/ seconds |
| 5 | 280.917543 |
| 9 | 293.424274 |
| 17 | 292.826251 |
| 25 | 299.189220 |
| 33 | 305.760168 |

* **Obviously, for 1110 \* 1342 image, the time is not long.**
* **Perhaps the calculating time has positive correlation with size of window.**

**4. Conclusion**

**4.1 Experiment Conclusion**

According to the results above, my program is able to calculate disparity of a pair of stereo images.. So, I believe I complete the assignment successfully.

During this experiment, faced with the problems like how to use the *Matlab* advantage of matrice calculation to save time, I was trained extremely. In my opinion, patience, carefulness and courage to explore and use new methods are the key quality to this homework. Moreover, I experience the function of many useful functions.

**4.2 Problems and Weaknesses**

* The testing pictures are not taken by myself;
* Optional assignment is not finished;
* Many details are missed in the disparity view even though the window size is small.
  + Possible causes: image resolution, SSD’s weakness, illumination.

**Appendix A. Code**

%ssdMatch.m

function [ matchedPointImg1, matchedPointImg2 ] = ssdMatch( img1, img2, sizeNeighbourhood )

%UNTITLED2 Summary of this function goes here

% Detailed explanation goes here

[heightImg1, lengthImg1] = size(img1);

[heightImg2, lengthImg2] = size(img2);

mask = ones(sizeNeighbourhood);

if mod(sizeNeighbourhood, 2) == 0

error('sizeNeighbour must be odd');

else if mod(sizeNeighbourhood, 2) == 1

leftUpOffset = (sizeNeighbourhood - 1) / 2;

end

end

if (heightImg1 ~= heightImg2) && (lengthImg1 ~= lengthImg2)

error(' The input images have different sizes!');

end

img2MoveDistance = 0;

img2Move = zeros( heightImg2, lengthImg2 + img2MoveDistance );

img2Move(:, img2MoveDistance + 1: lengthImg2) = img2(:, 1: lengthImg2 - img2MoveDistance);

if img2MoveDistance >= 1

img2Move(:, 1: img2MoveDistance) = img2(:, lengthImg2 - img2MoveDistance + 1: lengthImg2);

end

tmp = imfilter((ImgExtend(img1, leftUpOffset) - ImgExtend(img2Move(:, 1: lengthImg2), leftUpOffset)) .^ 2, mask);

SSDimg1Img2 = zeros(heightImg1, lengthImg1, 2);

SSDimg1Img2(:, :, 1) = tmp(1: heightImg1, 1: lengthImg1);

clear tmp;

SSDimg1Img2Old = zeros(heightImg2, lengthImg2, 2);

SSDimg1Img2New = zeros(heightImg2, lengthImg2, 2);

for img2MoveDistance = 1: lengthImg2 - 1

SSDimg1Img2Old = SSDimg1Img2;

img2Move = zeros( heightImg2, lengthImg2 + img2MoveDistance );

img2Move(:, img2MoveDistance + 1: lengthImg2) = img2(:, 1: lengthImg2 - img2MoveDistance);

if img2MoveDistance >= 1

img2Move(:, 1: img2MoveDistance) = img2(:, lengthImg2 - img2MoveDistance + 1: lengthImg2);

end

tmp = imfilter((ImgExtend(img1, leftUpOffset) - ImgExtend(img2Move(:, 1: lengthImg2), leftUpOffset)) .^ 2, mask);

SSDimg1Img2New(:, :, 1) = tmp(1: heightImg1, 1: lengthImg1);

clear tmp;

OldNewErrors = SSDimg1Img2Old(:, :, 1) - SSDimg1Img2New(:, :, 1);

newMask = zeros(heightImg2, lengthImg2, 1);

oldMask = zeros(heightImg2, lengthImg2, 1);

newMask(OldNewErrors > 0 ) = 1;

% oldMask = abs((OldNewErrorsBin - ones(heightImg2, lengthImg2)) / 2);

oldMask(OldNewErrors <= 0) = 1;

SSDimg1Img2(:, :, 1) = newMask .\* SSDimg1Img2New(:, :, 1) + oldMask .\* SSDimg1Img2Old(:, :, 1);

SSDimg1Img2(:, :, 2) = newMask .\* img2MoveDistance + oldMask .\* SSDimg1Img2Old(:, :, 2);

end

A = SSDimg1Img2(:, :, 1);

B = SSDimg1Img2(:, :, 2);

matchedPointImg1 = zeros(heightImg1, lengthImg1, 2);

matchedPointImg2 = zeros(heightImg2, lengthImg2, 2);

for x = 1: lengthImg1

matchedPointImg1(:, x, 1) = [1: 1: heightImg1]; % y is 1st

end

matchedPointImg2(:, :, 1) = matchedPointImg1(:, :, 1);

for y = 1: heightImg1

matchedPointImg1(y, :, 2) = [1: 1: lengthImg1]; % x is 2nd

end

matchedPointImg2(:, :, 2) = matchedPointImg1(:, :, 2);

matchedPointImg2(:, :, 2) = mod(matchedPointImg2(:, : ,2) + SSDimg1Img2(:, :, 2), lengthImg2);

end

%main.m

clear;

clc;

view1 = imread('E:\Document Files\Learning\DIP\Homework8\Project\view1.png');

view2 = imread('E:\Document Files\Learning\DIP\Homework8\Project\view5.png');

view1 = im2double(view1);

view2 = im2double(view2);

view1Gray = rgb2gray(view1); % must im2double before rgb2gray

view2Gray = rgb2gray(view2);

%%%%%%%%%%%%%%%%%%% Harris Corner %%%%%%%%%%%%%%%%%%%%%%%

cornersImg1 = detectHarrisFeatures(view1Gray);

cornersImg2 = detectHarrisFeatures(view2Gray);

[f1, vpts1] = extractFeatures(view1Gray, cornersImg1);

[f2, vpts2] = extractFeatures(view2Gray, cornersImg2);

matchedPairs = matchFeatures(f1, f2);

matchedPoints1 = vpts1(matchedPairs(:, 1));

matchedPoints2 = vpts2(matchedPairs(:, 2));

matchedPoints1Coordinate = matchedPoints1.Location;

matchedPoints2Coordinate = matchedPoints2.Location;

figure,

cornerTest = showMatchedFeatures(view2, view1, [matchedPoints2Coordinate(:, 1), matchedPoints2Coordinate(:, 2)], ...

[matchedPoints1Coordinate(:, 1), ...

matchedPoints1Coordinate(:, 2)], 'montage');

close all;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B = 160;

f = 3740;

dmin = 230; % added to the halfsize and thirdsize

for size = [5, 9, 17, 25, 33]

%%%%%%%%%%%%%%%%%%%%%% calculation %%%%%%%%%%%%%%%%%%%%%%%%%%%

tic;

[matchedPointView1, matchedPointView2] = ssdMatch( view1Gray, view2Gray, size );

disparity = (matchedPointView1(:, :, 2) - matchedPointView2(:, :, 2));

Z = (B \* f) ./ disparity;

toc;

%%%%%%%%%%%%%%%%%%%%% data output %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

D = disparity;

% edgeRange = size(find(disparity(1, :) > 0), 0)

D (D > 0) = 0;

D = abs(D);

D = 1 \* D / 200;

% figure, imshow(D), colormap jet;

% imwrite(D, ['./Result/disparityOfSize', num2str(size), 'Gray.png']);

Z (Z > 0) = 0;

Z = abs(Z);

xlswrite(['./Result/size', num2str(size), '.xlsx'], Z);

% Z = 1 \* Z / 10000;

% figure, imshow(Z), colormap jet;

% imwrite(Z, ['./Result/ZOfSize', num2str(size), 'Gray.png']);

end

1. As the size of neighbourhood is odd in thid expeirmnt ,the value of offset is (the size of neighbourhood - 1) / 2. [↑](#footnote-ref-1)
2. This kind of use instead of *for* loop is very useful. [↑](#footnote-ref-2)
3. Move distance of each element is saved in SSD (:, :, 2). [↑](#footnote-ref-3)