

Problem Set 6

CMSC 426

Assigned Thursday, November 10, Due Tuesday, November 22

1. **Stereo Correspondence.** For this problem set you will solve the stereo correspondence problem using a shortest path algorithm, as described in class. The goal of this algorithm is to find the lowest cost matching between the left and right images, so that the matching obeys the epipolar, ordering, non-negative disparity and uniqueness constraints. First, let's define these:
 - a. The epipolar constraint tells us that we can match the images one row at a time. So we have to solve a matching problem with 1D images, matching pixels in a row in the left image to pixels in the same row of the right image. Then we combine the results for every row.
 - b. The ordering constraint means that if pixel i in the left image matches pixel j in the right image, then no pixel to the right of pixel i is allowed to match a pixel to the left of pixel j .
 - c. The uniqueness constraint means that every pixel can match at most one pixel. However, a pixel might be occluded, and match nothing.
 - d. Non-negative disparity means that no point should have negative disparity, because all points are in front of the camera, and have positive depth.
 - e. Subject to these constraints we use a cost function to measure how good a match is. If we match pixel i in image I to pixel j in image J , the cost of this match will be $(I(i)-J(j))^2$. If any pixel is not matched, the cost of this is OC, which is some constant occlusion cost. For the experiments below, I suggest an occlusion cost of $OC = 625$, which is about the same as the cost of matching two pixels with an intensity that differs by 25. However, feel free to experiment with other values to try to improve the results.

We will define a graph with start and end nodes, so that a path from the start to the end will encode a complete matching of the two images. At every step of the way, we can do two possible things. One is to match two pixels, the second is to allow a pixel to go unmatched. We will create a graph in which nodes represent which pixels have been accounted for, edges represent possible matches or occlusions, and edge weights encode the cost of matching. We will name a node in a way that indicates which pixels have been matched so far. For example, if we reach node $N(5,3)$ this will mean that the first five pixels in image 1, and the first three pixels in image 2, have all been taken care of. From $N(5,3)$ we can go to $N(6,4)$. This must mean that we take care of both nodes 6 and 4 in one step, by matching them together. Or we can go to node $N(6,3)$. This means that node 6 in the first image is taken care of by not matching it to anything. That is, node 6 in the first image is occluded. Likewise, we could go to $N(5,4)$. We also need a special start node, S . This is connected to $N(0,1)$, $N(1,0)$ and $N(1,1)$. We need a special end node, E . For example, if there are 9 pixels in each image, E will be $N(9,9)$, indicating that all pixels are accounted for. It will be connected to $N(8,8)$, $N(8,9)$, $N(9,8)$.

Finally, we use edge weights to encode the cost of these choices.

$$E(N(i-1,j-1), N(i,j)) = (I(i)-I_2(j))^2.$$

$$E(N(i-1,j),N(i,j)) = 0C$$

$$E(N(i,j-1),N(i,j)) = 0C.$$

Now when we take a path from S to E we are going through edges that represent a matching of the images. The cost of the path is the cost of the matching.

We will give you code that finds a shortest path in a graph. Your task will be to use this code to do stereo correspondence. The code, with comments that explain how to call it, is in the file *shortest_path.m*. This code is a bit hacky and inefficient (ie., I wrote it). So, two things to keep in mind. First, this code will be kind of slow for big graphs; on my laptop it takes a couple of seconds for a graph with 10,000 nodes, but dies pretty badly with 100,000 nodes. Second, although I've been able to use this code to solve the problems in this problem set, I won't guarantee that there are no bugs. If there are, it is your responsibility to fix them, or rewrite the code, and get the whole program to work.

- a. **Paper and pencil exercise:** Using the cost function above *but with an occlusion cost of .01*, compute all minimum cost matches that obey all the constraints listed above for the 1D images, $v1 = [1\ 0\ 1\ 1]$; $v2 = [1\ 1\ 0\ 1]$. Write your answers as a disparity map for $v1$. Since disparity is always non-negative, we can use -1 to indicate an occlusion. That is, a map of $[0\ -1\ 1\ 1]$ means the disparity for the first point in $v1$ is 0 ($v1(1) = 1$ matches $v2(1) = 1$), the second point is occluded and unmatched, the third point has a disparity of 1, ($v1(3) = 1$ matches $v2(2) = 1$), and the fourth point has a disparity of 1 ($v1(4) = 1$ matches $v2(3) = 0$). Of course, this example is not necessarily a minimum cost matching.
- a. **5 points** There may be one or more matches with the same minimum cost. List all of them, along with their cost.

There is no way to match the two 0s to each other without violating the constraint that all disparity be non-negative. There is exactly one way to match all the 1s to each other obeying the ordering constraint, so this is the best we can do. The disparity map is: $[0\ -1\ 1\ 0]$. There are two occlusions (both 0s are occluded) so the cost is .02.

- b. **5 points** List all minimum cost matches if we are allowed to ignore the non-negative disparity constraint.

Now we can match 0 to 0. However, to do this, something still has to be occluded. If we match without occlusions we can only have all disparities 0, which would have a high cost of 1. The new matches we can get with only one occlusion are:

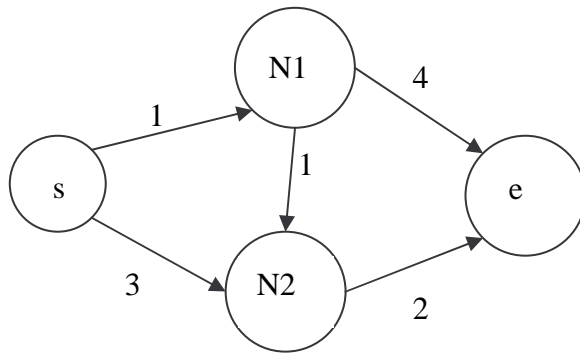
$[0\ -1\ -1\ X]$, $[0\ -1\ X\ 0]$, and $[-1\ -1\ -1\ X]$. Plus, we can still have $[0\ X\ 1\ 0]$. All these matches have a cost of .02. Note that for these matches we use X to mean occlusion, since -1 means a disparity of -1.

- c. **5 points** List all minimum cost matches if we ignore the ordering constraint and the non-negative disparity constraint.

Now we can match any 1 to any other 1. However, we still want to match 0 to 0 to get the best matches. This gives us:

$[0 -1 1 0]$, $[0 -1 -1 2]$, $[-1 -1 2 0]$, $[-1 -1 -1 3]$, $[-3 -1 1 0]$, $[-3 -1 -1 2]$. These all have a cost of 0.

- b. **10 points** Now, check out the documentation for *shortest_path*. Verify that you know how to use it by setting up a matrix, C , that encodes the graph shown below. Show the value you use for C along with the output of your call to *shortest_path*.



We can encode this graph with the matrix:

$$C = \begin{matrix} & 2 & 1 & 3 & 3 \\ 3 & 1 & 4 & 4 \\ 4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

Notice that we have to make s the first node, and e the last (4^{th} node). We make $N1$ the second node, and $N2$ the third node.

Then, running the *shortest path* code we get:

$$p = \begin{matrix} 1 & 2 & 3 & 4 \\ c = & 4 \end{matrix}$$

- c. **40 points** Now, write a function `stereo1D(I1, I2, oc)`. This function will take as input two 1D images, along with an occlusion cost, `oc`. For our experiments, `oc` will typically be 625. The function will return two values, the disparities for the best match, and the cost of this match. Test this with the following call:

Here is our code, with some comments.

```
function [D c] = stereo1D(I1, I2, oc)
% We will compute the disparity for the best match between
% two 1D images, I1 and I2. oc gives the occlusion cost;
% this is a parameter.
% d will be a vector of the same length as I1, indicating
% the disparity for
% each pixel in I1. The disparity is always positive, so a
% value of -1
% will indicate an occluded pixel.
%
% We need to come up with a way of encoding states of the
% matching into
% nodes. Then we can run shortest path code.
MAXDISP = 16;
% We can speed things up by limiting disparity to be 16,
% which is fine for these problems.
n = length(I1);
m = length(I2);
N = (n+1)*(m+1);
% This is the total number of nodes.
C = zeros(N,6);
% We'll initialize C with zeros, which denote a lack of
% edges. There are
% six columns, because every matching state can only lead
% to two other matching states.
D = ((I1'*ones(1,m)) - (ones(n,1)*I2)).^2;
% D contains the cost of each matching. D(i,j) is the cost
% of matching I1(i) to I2(j). It is a bit quicker if we
% compute this all at once, with matrix operations.
for i = 0:n
    for j = max(0,i-MAXDISP):i
        % We don't allow j to be bigger than i, or too much
        % less, because we want disparity to be
        % non-negative and not too big. If we allow
        % arbitrary disparity, j just goes from 0 to i.
        % We will no work on the edges leaving node N(i,j).
        no = node_number(i,j,n,m);
        % This function converts (i,j) into a single index.
        if i < n & j < m
```

```

        % This edge describes a match between pixels in
        % each image, which we can have as long as
        % neither pixel we've accounted for is the last
        % one.
        C(no,1) = node_number(i+1,j+1,n,m);
        C(no,2) = D(i+1,j+1);
    end
    % In the next two cases, we add occlusions. We
    % just check to make sure that the occluded pixel
    % isn't after the last one.
    if i < n
        C(no,3) = node_number(i+1,j,n,m);
        C(no,4) = oc;
    end
    if j < m
        C(no,5) = node_number(i,j+1,n,m);
        C(no,6) = oc;
    end
    % Note that if we don't need an edge, we just leave
    % it 0,0.
end
end
[P, c] = shortest_path(C);
D = [];
% D will be a vector giving the disparity map.
[i,j] = decode_node(P(1),n,m);
for node = P(2:length(P))
    [nexti, nextj] = decode_node(node, n, m);
    if nexti == i + 1 & nextj == j + 1 & i >= j
        % This means i+1 in I1 was matched to j+1 in I2.
        % So disparity is i-j. Disparity must be positive.
        D = [D, i - j];
    elseif nexti == i & nextj == j + 1
        % This means that j+1 in I2 was occluded and not
        % matched.
        % So we don't actually need to do anything here.
        % The next line is a dummy command.
        t1 = 0;
    elseif nexti == i+1 & nextj == j
        % This means that i+1 in I1 was occluded.
        D = [D, -1];
    else
        error('It seems like an illegal move was taken.');
```

```

function nn = node_number(i, j, n, m)
% nn will be the number of the node that corresponds to
% pixel i in I1 and pixel
% j in I2 having both been accounted for. We also need the
% lengths of I1(n) and I2(m).
nn = 1 + i*(n+1)+j;

function [i j] = decode_node(nn, n, m)
j = mod(nn-1, n+1);
i = (nn - 1 - j)/(n+1);

```

```
[d, c] = stereo1D([100 0 200 220 40 30 230], [0 200 220 40 30 230 90], 625)
```

The result should be:

```

d =
  -1  1  1  1  1  1  1
c =
  1250

```

Also test with the following call:

```
[d, c] = stereo1D([50 70 90 110], [30 50 70 90], 625)
```

The result should be:

```

d =
  0  0  0  0
c =
  1600

```

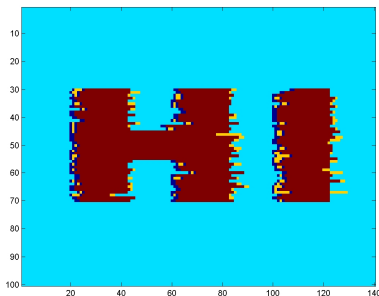
Turn in your code, and the output on these runs.

- d. **10 points** Now, write a function *stereo2D(I1, I2, oc)*. This function will take as input two 2D images, along with an occlusion cost, *oc*. For our experiments, *oc* will typically be 625. The function will return two values, a 2D disparity map for the best match, and the cost of this match. In order to make this code more efficient, modify your *stereo1D* function so that it does not allow disparities that are greater than 16. You can use comments to show the changes in your code, and just hand in one file with this function. Test your function on the images *I1L.jpg* and *I1R.jpg*, which contain left and right images respectively, from a random dot stereogram. Turn in your code, along with an image of the disparity map that you compute. You will have to scale the disparities or perhaps show them in color to make them visible, because

they will be small numbers. I suggest using *imagesc*. I am not including an image of the result, but you will know when you have the right answer.

```
function D = stereo2D_SP(I1, I2, oc)
% The input consists of two images (they should have the
% number of rows) and an occlusion cost.
% We will return a matrix that indicates the disparity for
% every pixel.
I1 = double(I1);
I2 = double(I2);
% We want to make these double, so we can perform
% arithmetic operations.
D = [];
for i = 1:size(I1)
    d = stereo1D_SP(I1(i,:), I2(i,:), oc);
    D = [D; d];
    i % print i so we can see how far we're getting.
end
```

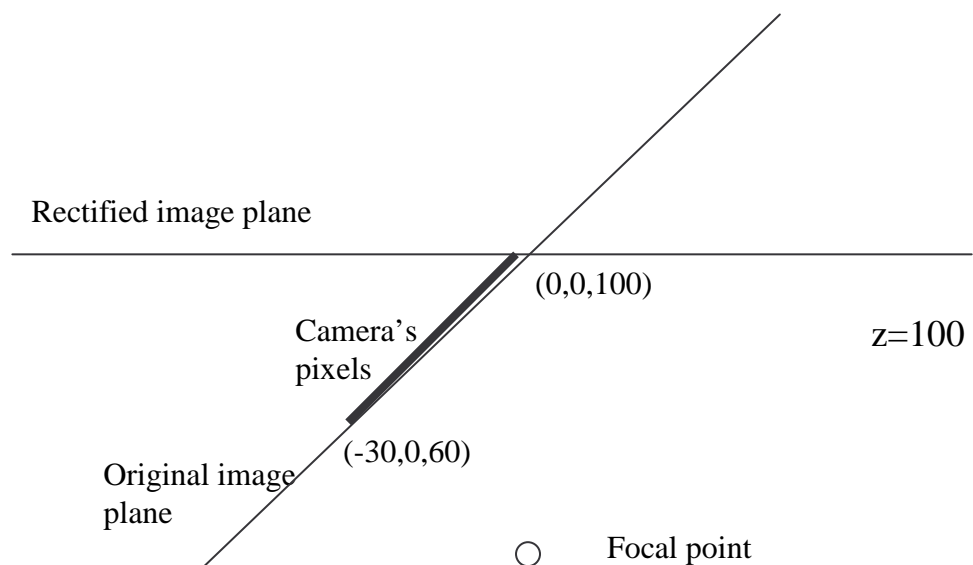
Here is a picture of the result:



Note that things are basically right, but the result is a bit jagged. When a pixel is occluded, it still might have a good match in the other image, leading to this effect.

- e. **5 points.** Now you are to run this algorithm on a part of the Tsukuba images. My shortest path code is too poorly written to run on the entire images, so just use columns 125 through 250 of the images, which are on the class web page. My results are also on the class web page.
 - f. **Challenge Problem 10 points:** Run shortest path stereo matching on the Tsukuba images provided on the class web page. To do this, you will probably have to rewrite my shortest path code, or find some better code. Hand in all your code and the results.
2. **20 points** Now you're going to write a program to perform rectification for a 1D image in the $y=0$ plane. This will have the form:
[O,s] = rectify1D(I,xs,zs,xe,ze). We will assume that the focal point is always at (0,0,0). We will also assume that all coordinates are written in units of pixels.

The pixels in the original image are contained in the vector I . In world coordinates, this image appears on the image plane beginning at the point $(x_s, 0, z_s)$, and extends to the point $(x_e, 0, z_e)$. This means that we must have $\text{length}(I) \geq \sqrt{(x_s - x_e)^2 + (z_s - z_e)^2}$, so that the region in the image plane that contains the image is no longer than the number of pixels we have in the image. If I contains extra pixels, they don't need to be used in the rectified image.



Our goal is to rectify the image so that it appears as if it were taken by a camera with a focal point at $(0,0,0)$, and an image plane at $z=100$. So to rectify an image according to the geometry shown in the picture above, we would need I to be a vector 50 long containing 50 intensities, and we would call: $[O,s] = \text{rectify1D}(I, -30, 60, 0, 100)$.

`rectify1D` will produce two outputs. O is a vector of pixel intensities for the rectified image. s indicates the x coordinate in the image plane where the rectified image starts.

Test your function by executing the calls:

```
--- [O,s] = rectify1D(ones(1,50), -30, 60, 0, 100).
```

This should return for O a vector containing 50 1s.

```
--- [O,s] = rectify1D(1:50, -30, 60, 0, 100).
```

My code returns the following answer for O :

$O =$

Columns 1 through 10

1 1 2 3 3 4 5 5 6 7

Columns 11 through 20

7 8 9 10 10 11 12 13 14 14

Columns 21 through 30

15 16 17 18 19 20 21 22 23 24

Columns 31 through 40

25 26 27 28 29 30 31 33 34 35

Columns 41 through 50

36 38 39 41 42 43 45 46 48 50

Your code might produce a slightly different answer depending on how you choose to sample pixels in the image plane and match them to pixels in the original image. For example, in my code, for the center of each pixel in the rectified image I compute the intensity value that was present in the original image.

Turn in your code and the results you get for these two test images.

```
function [O,start] = rectify1D(I,xs,zs,xz,ze)
% I is a 1D image that lies in the y = 0 plane.
% Assume this image is taken with a focal point at
% (0,0,0). The image plane is oriented so that the
% image starts at (xs,0,zs) and ends at (xz,0,ze).
% We write everything in units of pixels so that,
% for example, (0,0,0) and (0,0,1) are one pixel apart.
% This means that the length of I should be equal to the
% distance between (xs,0,zs) and (xz,0,ze).
% This function outputs a new, rectified image, O.
% This is the image that we would have gotten had we
% taken the picture with a focal point still at (0,0,0),
% but with the image plane equal to the z = 100.
% s will be the starting point of this image.
% So the image will extend from (s,0,100)
% to (s+length(O), 0, 100).
f = 100;
% This is the focal length of the camera we want.
s = .5 + ceil((f*xs/zs)-.5);
% We find the starting point of the new image by taking
% the starting point of the original image and projecting
```

```

% it into the image. We assume that a pixel is the point
% in the middle between two integer values.
start = floor(s);
e = floor((f*xe/ze)+.5) - .5;
% We compute intensity using the middle of all
% pixels that see the original image.
O = [];
m = (ze-zs)/(xe-xs);
b = zs - m*xs;
% The line containing the image is z = mx+b
for i = s:e
    % A line through the focal point and (i,0,100) is zi =
    % 100x.
    % Combining the two equations we get mxi + bi = 100x
    if i == 0
        x = 0;
        z = b;
    else
        x = (b*i)/(100 - m*i);
        z = 100*x/i;
    end
    % (x,0,z) is the point where these lines intersect.
    d = sqrt( (x-xs)^2 + (z-zs)^2 );
    % d is the distance from the point on the line of the
    % original image that we want, and the starting point
    % of the original image.
    O = [O, I(1+floor(d))];
end

```