# CS 231A Computer Vision (Spring 2016)
# Problem Set 3

**Solution Set**

Due: May 4th, 2016 (11:59pm)

## 1 Space Carving (25 points)

Dense 3D reconstruction is a difficult problem, as tackling it from the Structure from Motion framework (as seen in the previous problem set) requires dense correspondences. Another solution to dense 3D reconstruction is space carving[1], which takes the idea of volume intersection and iteratively refines the estimated 3D structure. In this problem, you implement significant portions of the space carving framework.

Download the starter code at `www.stanford.edu/class/cs231a/hw/ps3/space_carving.zip`.

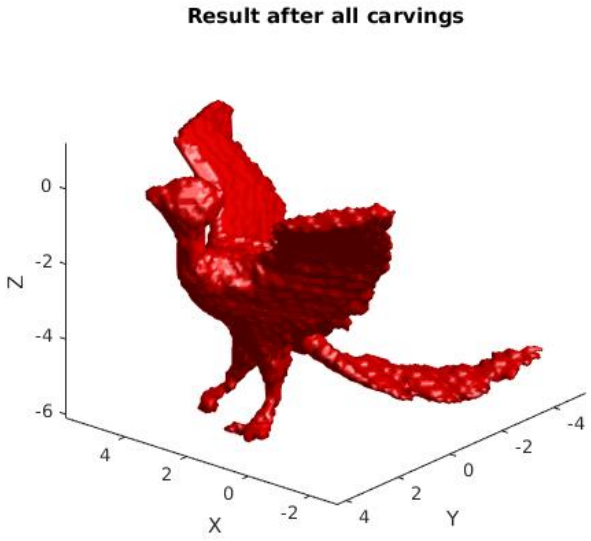



Figure 1: Our final carving using the true silhouette values

(a) The first part of space carving is that we must generate silhouettes for every image. Complete `generateSilhouette.m` and submit your code along with the output from the main

[1] `http://www.cs.toronto.edu/~kyros/pubs/00.ijcv.carve.pdf`

script `spaceCarving.m`. Make sure your silhouettes have the object contained, even if you get many false positives. [**5 points**]

(b) The next part is to generate the initial voxel grid that we will carve into. Complete `formInitialVoxels.m` and submit your code along with a picture of your initial voxels. [**5 points**]

(c) Now, the key step is to implement the carving for one camera. To carve, we need the camera frame and the silhouette associated with that camera. Then, we carve the silhouette from our voxel grid. Do this in `carve.m`. Submit your code and a picture of what it looks like after one iteration of the carving. [**5 points**]

(d) The last step is to simply submit the final output after all carvings have been completed. Why does your carving look (hopefully) good if your silhouettes weren't perfect? [**5 points**]

(e) Finally, let's use the true silhouettes to carve, which is given in the code. Give the final output after all carvings have been completed. [**2 points**]

(f) Recall that in part a, your silhouettes are conservative estimates of the true silhouettes - that is the object is contained, but there are also a lot of extraneous points. However, in part d, you still should get a decently good estimate of the 3D reconstruction. Why is this the case? What happens if you reduce the number of views? What if your silhouettes weren't conservative and one or a few views had parts of the object missing? [**3 points**]

**Solution:**

1. `generateSilhouette.m` and its output

```
function s = generateSilhouette( im )
%GENERATESILHOUETTE - find the silhouette of an object centered in the ...
    image
% Arguments:
%          im - an image matrix read in by im2read (size H X W X C)
%
% Returns:
%          s - the silhouette matrix (size HxW) of 0's and 1's, where 0
%          means that it is not part of the object, and 1 is part of the
%          object (the parts labeled 0 will be carved away).


% Initial segmentation based on more red than blue and green
s = im(:,:,1) > (im(:,:,3)-2) & im(:,:,1) > (im(:,:,2)-2);
```

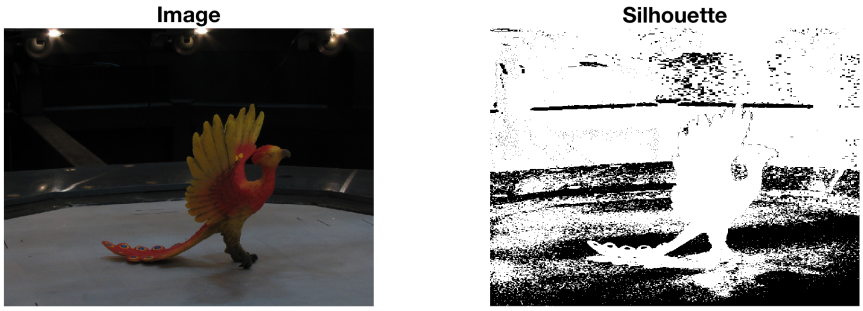



Figure 2: Output of the function 'generateSilhouette' on one of the images

2. `formInitialVoxels.m` and its output

```
function [voxels, voxel_size] = formInitialVoxels(xlim,ylim,zlim,N)
%FORMINITIALVOXELS  create a basic grid of voxels ready for carving
% Arguments:
%          xlim - The limits of the x dimension given as [xmin xmax]
%          ylim - The limits of the y dimension given as [ymin ymax]
%          zlim - The limits of the z dimension given as [zmin zmax]
%          N - The approximate number of voxels we desire in our grid
%
% Returns:
%          voxels - the matrix of N'x3 (where N' approximately equals ...
    N) of
%               voxel locations
%          voxel_size - the distance between the locations of adjacent ...
    voxel
%               (a voxel is a cube)
%

% We need to create cube-shaped voxels, so choose a voxel_size to give
% roughly N voxels
volume = diff( xlim ) * diff( ylim ) * diff( zlim );
voxel_size = power( volume/N, 1/3 );
x = xlim(1) : voxel_size : xlim(2);
y = ylim(1) : voxel_size : ylim(2);
z = zlim(1) : voxel_size : zlim(2);

[X,Y,Z] = meshgrid( x, y, z );
XData = X(:);
YData = Y(:);
ZData = Z(:);
voxels = [XData, YData, ZData];
```

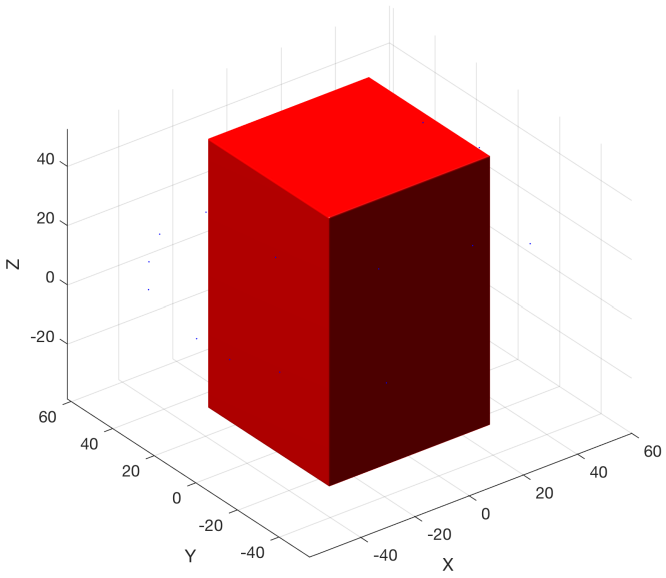



Figure 3: Output of the function 'formInitialVoxels' on one of the images

3. `carve.m` and its output

```
function [voxels] = carve( voxels, camera)
% CARVE: carves away voxels that are not inside the silhouette ...
    contained in
%   the view of the camera. The resulting voxel array is returned.
% Arguments:
%    voxels - an Nx3 matrix where each row is the location of a cubic voxel
%    camera - The camera we are using to carve the voxels with. Useful data
%       stored in here are the "Silhouette" matrix and the
%       projection matrix "P". (you can use but don't need the "Image"
%       matrix)
% Returns:
%    voxels - a subset of the argument passed that are inside the
%       silhouette

% Project into image
v = camera.P * [voxels ones(size(voxels,1),1)]';
v = v';
x = v(:,1) ./ v(:,3);
y = v(:,2) ./ v(:,3);

% Clear any that are out of the image
[h,w,d] = size(camera.image);
keep = find( (x>=1) & (x<=w) & (y>=1) & (y<=h) );
x = x(keep);
y = y(keep);

% Now clear any that are not inside the silhouette
ind = sub2ind( [h,w], round(y), round(x) );
keep = keep(camera.silhouette(ind) >= 1);

voxels = voxels(keep,:);
```

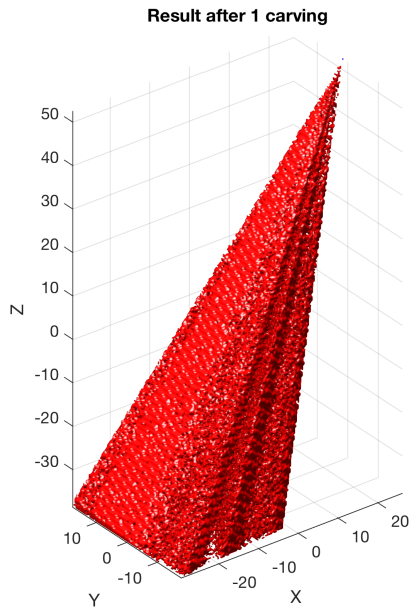



Figure 4: Output of the function ‘carve’ on one of the images

4. Final carving using coarse, self-generated silhouettes

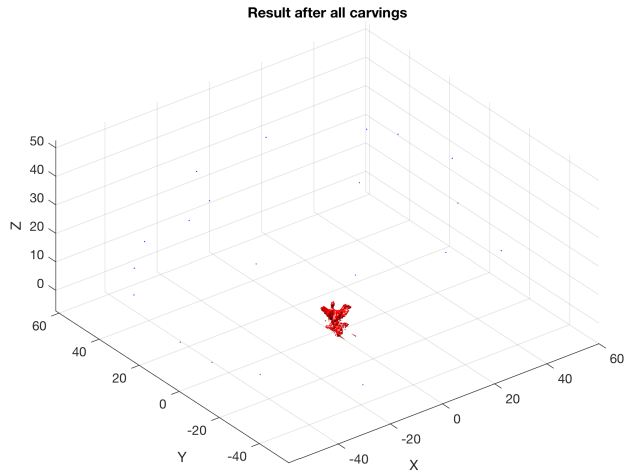



Figure 5: Final carving output of the whole scene using coarse silhouettes

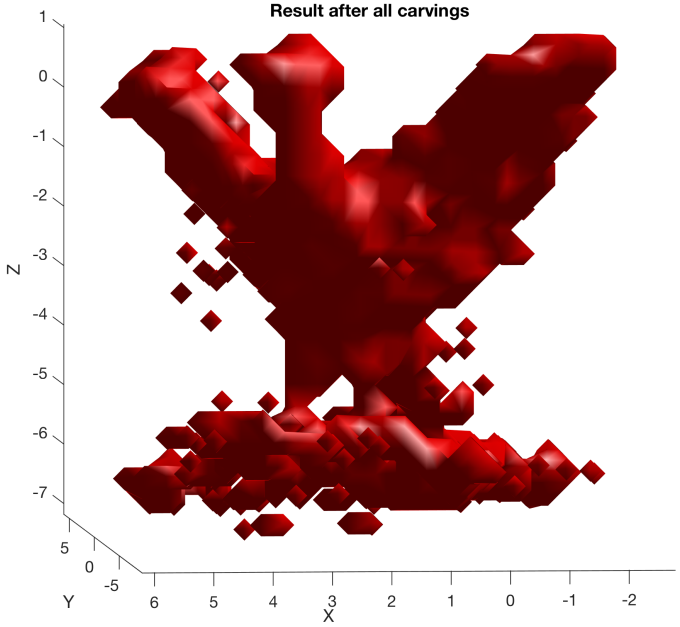



Figure 6: Final carving output of the 3D model using coarse silhouettes

Note that the carving looks "good" even though the silhouettes were not precise as we are relying on multiple images of the 3D model from a distribution of viewpoints. Even though the silhouette from each of the viewpoints consists of numerous outliers, they all agree upon a majority of inliers.

5. Final carving using true silhouettes

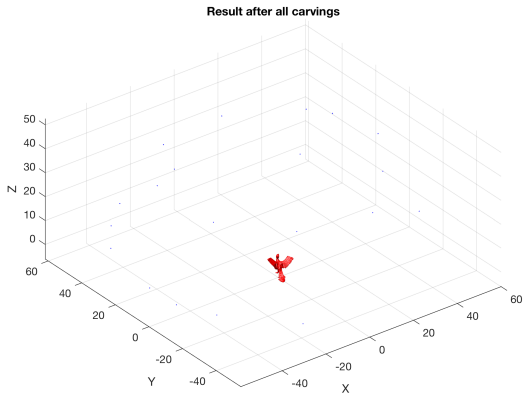



Figure 7: Final carving output of the whole scene using true silhouettes

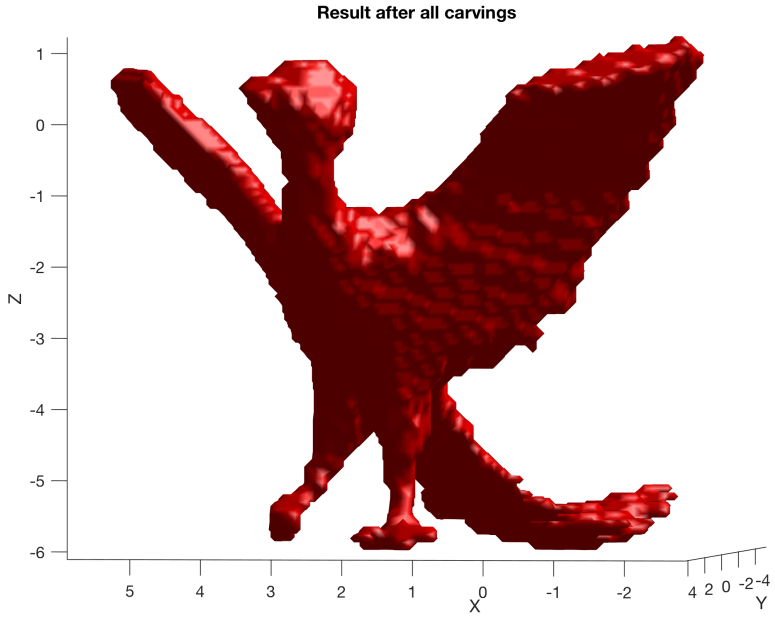



Figure 8: Final carving output of the 3D model using true silhouettes

6. A good estimate of the 3D model is possible as multiple images from various viewpoints have been used to carve. If the number of the views are reduced, the reconstruction will not be precise, in particular, there will be a lot of outlier voxels which will be part of the 3D model. If the silhouettes weren't conservative or if few of the views had parts of the object missing then the 3D reconstruction will have fewer inlier voxels, i.e., the 3D model would be deficient.

## 2 Single Object Recognition Via SIFT (35 points)

In his 2004 SIFT paper, David Lowe demonstrates impressive object recognition results even in situations of affine variance and occlusion. In this problem, we will explore a similar approach for recognizing and locating a given object from a set of test images. It might be useful to familiarize yourself with sections 7.1 and 7.2 of the paper[2]. The code and data necessary to solve this problem can be found at `www.stanford.edu/class/cs231a/hw/ps3/sift_matching.zip`.


Figure 9: Sample Output, showing training image and keypoint correspondences.

(a) Given the descriptor $g$ of a keypoint in an image and a set of keypoint descriptors from another image $f_1 ... f_n$, write the algorithm and equations to determine which keypoint in $f_1 ... f_n$ (if any) matches $g$. Implement this matching algorithm in the given function `matchKeypoints.m` and test its performance using the `matchObject.m` skeleton code. The matching algorithm should be based on the one used by Lowe in his paper, using the ratio of the two closest matches to determine if a keypoint has a match. Please read the section on object recognition for more details. Load the data in `PS3_Prob3.mat` and run the system with the following line:

```
>>matchObject(stopim{1}, sift_desc{1}, keypt{1}, obj_bbox, stopim{3}, ...
 sift_desc{3}, keypt{3});
```

Note that the SIFT keypoints and descriptors are given to you in `PS3_Prob3.mat` file. Your result should match the sample output in Fig. 9. Turn in your code and a sample image similar to Fig. 9. **[7 points]**

(b) From Figure 9, we can see that there are several spurious matches that match different parts of the stop sign with each other. To remove these matches, we can use a technique called RANSAC to find matches that are consistent with a homography between the locations of the matches in the two images.

The RANSAC algorithm generates a model from a random subset of data and then calculates how much of the data agrees with the model. In the context of this problem, a random subset of matches and their respective key point locations in each image is used to generate a homography in the form of $x_i^{'} = Hx_i$ where $x_i$ and $x_i^{'}$ are the homogeneous coordinates of matching key points of the first and second images respectively. We then calculate the per-keypoint reprojection error by applying $H$ to $x_i$:

[2] `http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf`


Figure 10: Sample Output, showing training image and keypoint correspondences with RANSAC filtering.

$$error_i = ||x_i' - Hx_i||_2,$$

where $x_i'$ and $Hx_i$ should be converted back to nonhomogeneous coordinates. The inliers of the model are those with an error smaller than a given threshold.

We then iterate by choosing another set of random matches to find a new $H$ and repeat the process, keeping track of the model with the most inliers. This is the model and inliers returned by `refineMatch.m`. The result of using RANSAC to filter the matches can be seen in Figure 10.

Implement this RANSAC algorithm in the given function `refineMatch.m` and test its performance using `matchObject.m`. A skeleton for the code as well as parameters such as the number of iterations and allowable error for inlier detection have been included. **[7 points]**

(c) We will now explore some theoretical properties of RANSAC.

(i) Suppose that e is the fraction of outliers in your dataset, i.e.

$$e = \frac{\#\text{ outliers}}{\#\text{total correspondences}}$$

If we choose a single random set of matches to compute a homography, as we did above, what is the probability that this set of matches will produce the correct homography?

**Solution:**

We need 4 correspondences to estimate a homography. For any randomly sampled point, the probability of being an inlier is $1 - e$. We need all 4 samples to be inliers in order to produce the correct homography, so the probability of sampling points that produce the correct homography is

$$p(\text{success}) = (1 - e)^4$$

(ii) Let $p_s$ be your answer from above, i.e. $p_s$ is the probability of sampling a set of points that produce the correct homography. Suppose we sample N times. In terms of $p_s$, what is the probability $p$ that at least one of the samples will produce the correct

homography? Remember, sets of points are sampled with replacement, so models are independent of one another.

**Solution:**

If $p$ is the probability that AT LEAST one of the samples will produce the correct homography, then $1-p$ is the probability that NONE of the N samples will produce the correct homography. Also, $1-p_s$ is the probability that each sample will not produce the correct homography, so we have

$$(1-p_s)^N = 1-p$$

or

$$p = 1-(1-p_s)^N$$

(iii) Combining your answers for the above, if 15% of the samples are outliers and we want at least a 99% guarantee that at least one of the samples will give the correct homography, then how many samples do we need? **[7 points]**

**Solution:**

Combining the above equations, we get

$$(1-p_s)^N = 1-p$$
$$(1-(1-e)^4)^N = 1-p$$

or

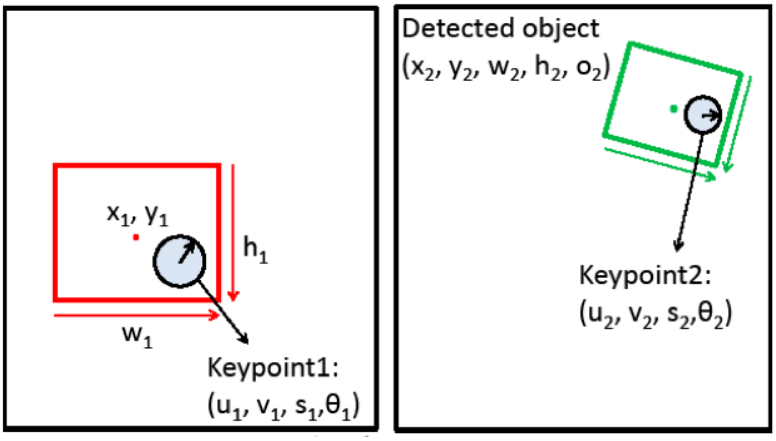
$$N = \log(1-p)/\log(1-(1-e)^4)$$

Plugging in p = 0.99 and e = 0.15, we get N = 6.2, which rounds up to 7 samples to get at least a 99% guarantee.

(d) Now given an object in an image, we want to explore how to find the same object in another image by matching the keypoints across these two images.

(i) A keypoint is specified by its coordinates, scale and orientation $(u, v, s, \theta)$. Suppose that you have matched a keypoint in the bounding box of an object in the first image to a keypoint in a second image, as shown in Figure 11. Given the keypoint pair and the red bounding box in Image 1, which is specified by its center coordinates, width and height $(x_1, y_1, w_1, h_1)$, find the predicted green bounding box of the same object in Image 2. Define the center position, width, height and relative orientation $(x_2, y_2, w_2, h_2, o_2)$ of the predicted bounding box. Assume that the relation between a bounding box and a keypoint in it holds across rotation, translation and scale.

(ii) Once you have defined the five features of the new bounding box in terms of the two keypoint features and the original bounding box, briefly describe how you would utilize the Hough transform to determine the best bounding box in Image 2 given $n$ correspondences. **[7 points]**




(a) Image 1 (b) Image 2

Figure 11: Two sample images for part (c)

(e) Implement the function `getObjectRegion.m` to recover the position, scale, and orientation of the objects (via calculating the bounding boxes) in the test images. You can use a coarse Hough transform by setting the number of bins for each dimension equal to 4. Your Hough space should be four dimensional.

Use the line in (a) to test your code and change all the 3's to 2, 4, 5 to test on different images. If you are not able to localize the objects (this could happen in two of the test images), explain what makes these cases difficult. Turn in your `getObjectRegion.m` and matching result images. **[7 points]**

## Solution:

(a) Find the two nearest neighbors in the second image using Euclidean distance. If the distance to the nearest neighbor is less than t times the distance to the second nearest neighbor (e.g, for $t = 0.8$), then the points are considered to match.

(b) See code. The difficulties are mainly to formulate and solve for the homograph given the initial random 4 matches

```
function [inliers, model]=refineMatch(P1,P2,matches)


N=length(matches);
X  = [ P1(1,matches(1,:)) ; P1(2,matches(1,:)); ones(size(P2(2,matches(2,:))))];
Xp = [ P2(1,matches(2,:)) ; P2(2,matches(2,:)); ones(size(P2(2,matches(2,:))))];


iter=1000;
pixelThresh=5;
```

```
N=4;
inliers=zeros(1,length(matches(1,:)));
model=eye(3);

method=1;

%Main loop, picks a set of 4 random points, calculates homography,
%calculates inliers.
for interation=1:iter

    %pick 4 matches at random
    pts=ceil(rand(1,N)*length(matches(1,:)));

    %make sure points are unique
    while(sum(diff(sort(pts))==0)>0)
        pts=ceil(rand(1,N)*length(matches(1,:)));
    end

    %Use SVD approach
    if(method==1)
        A=zeros(2*N,9);
        for i=1:N

            Ai=zeros(2,9);
            xi=X(:,pts(i));
            xp=Xp(:,pts(i));
            Ai(1,4:6)=-xi';
            Ai(1,7:9)=xp(2)*xi';
            Ai(2,1:3)=xi';
            Ai(2,7:9)=-xp(1)*xi';

            A((2*i-1):(2*i),:)=Ai;

        end
        [U,S,V] = svd(A);

        h=V(:,end);
        H=[h(1:3)';h(4:6)';h(7:9)'];

        %Use Least squares approach
    else
        A=zeros(2*N,8);
        b=zeros(2*N,1);
        for i=1:N
            Ai=zeros(2,8);
            xi=X(:,pts(i));
            xp=Xp(:,pts(i));
            Ai(1,:)=[xi(1) xi(2) 1 0 0 0 -xi(1)*xp(1) -xi(2)*xp(1)];
```

```
            Ai(2,:)=[0 0 0 xi(1) xi(2) 1 -xi(1)*xp(2) -xi(2)*xp(2)];

            A((2*i-1):(2*i),:)=Ai;
            b((2*i-1):(2*i))=xp(1:2);
        end
        h = (A'*A)^(-1)*A'*b;
        H=[h(1:3)';h(4:6)'; [h(7:8)' 1]];

    end

    %Calculate mapping from first image to second
    test=H*X;

    %Convert to image points
    test(1,:)=test(1,:)./test(3,:);
    test(2,:)=test(2,:)./test(3,:);
    test(3,:)=test(3,:)./test(3,:);

    %Calculate reprojection error
    error=((test(1,:)-Xp(1,:)).^2+(test(2,:)-Xp(2,:)).^2).^(.5);

    %Calculates inliers
    inlier=error<pixelThresh;

    %Keep track of best model
    if(sum(inlier)>sum(inliers))
        inliers=inlier;
        model=H;
    end

end
end
```

(c) (i) For a single correspondence we can use the following geometric relations to calculate the bounding box in the second orientation.

$$
\begin{aligned}
o_2 &= \theta_2 - \theta_1 \\
w_2 &= w_1 * (s_2/s_1) \\
h_2 &= h_1 * (s_2/s_1) \\
x_2 &= u_2 + (s_2/s_1) * [\cos(o_2) * (x_1 - u_1) - \sin(o_2) * (y_1 - v_1)] \\
y_2 &= v_2 + (s_2/s_1) * [\sin(o_2) * (x_1 - u_1) + \cos(o_2) * (y_1 - v_1)]
\end{aligned}
$$

Note: it is also possible to define the center by computing the distance and relative

orientation of the descriptor to the object center.

(ii) To extend this approach to $n$ correspondences we would need to use a hough transform (or some voting scheme) in 5 dimensional bins to determine a good fit for the 5 parameter of the bounding box in the second image.

(d) See code. The difficulties are the strong slant of the stop signs (SIFT is not invariant to out of plane rotation) and the multiple signs (Lowe's idea of using relative distances of first and second nearest neighbors as threshold assumes that there is only one valid match in the scene). In image 5, there are two signs, potentially causing the second problem mentioned above. To get full credit, there needed to be mechanisms to handle outliers (incorrect matches), such as Hough voting with a threshold on the number of votes required.


Figure 12: Issues with high affine variation


Figure 13: Clean Result

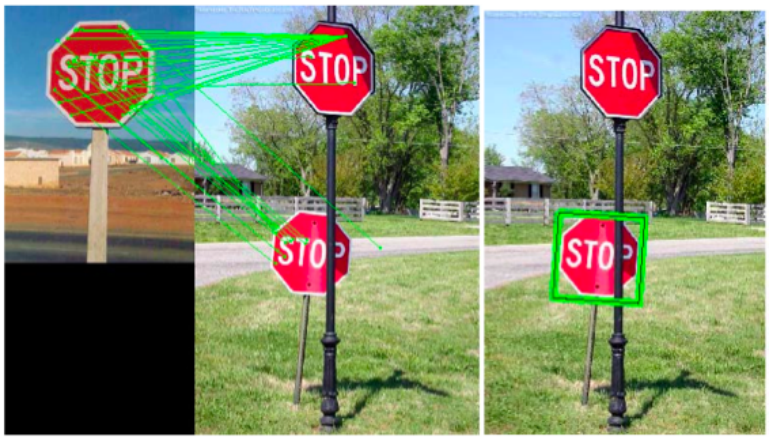



Figure 14: Difficulties with multiple/spurious matchest

```
function matches = matchKeypoints(desc1, desc2, thresh)

n1 = size(desc1,2);
matches = zeros(n1, 2);
n2 = size(desc2,2);
for k1 = 1:n1
    dist = zeros(n2,1);
    for k2 = 1:n2
        dist(k2) = sqrt(sum((desc1(:, k1)-desc2(:,k2)).^2));
    end
    [sval, sind] = sort(dist);
    if sval(1)/sval(2)<thresh
        matches(k1,1) = k1;
        matches(k1,2) = sind(1);
    end

end

matches = matches(matches(:,1)>0, :);
```

```
function [cx, cy, w, h, orient, count] = getObjectRegion(keypt1, keypt2, ...
 matches, objbox, thresh)

% Find parameters for object bounding box
objx = mean(objbox([1 3])); % x-center
objy = mean(objbox([2 4])); % y-center
objw = objbox(3)-objbox(1);
objh = objbox(4)-objbox(2);

% Find parameters for keypoints in image 1
s1 = keypt1(3, matches(1, :));
o1 = keypt1(4, matches(1, :));
x1 = keypt1(1, matches(1, :));
y1 = keypt1(2, matches(1, :));

% Find parameters for keypoints in image 2
s2 = keypt2(3, matches(2, :));
o2 = keypt2(4, matches(2, :));
x2 = keypt2(1, matches(2, :));
y2 = keypt2(2, matches(2, :));

% vote from each keypoint
vote_w = s2./s1*objw;
vote_x = x2 + sum([(objx-x1) ; (objy-y1)] .* [cos(o2-o1) ; -sin(o2-o1)],1).* ...
```

```
(s2./s1);%(objx-x1)./s1.*s2;
vote_y = y2 + sum([(objx-x1) ; (objy-y1)] .* [sin(o2-o1) ; cos(o2-o1)],1).*...
(s2./s1); %(objy-y1)./s1.*s2;
vote_o = mod(o2-o1+pi/4, 2*pi)-pi/4; % pi/4 shift is so that 0 rotation...
 doesn't get split bins

% Use four uniform bins for each dimension within range of x.  This
% certainly isn't optimal, but it gets the job done for this problem.
nbins = 4;
bs = assign2bins(vote_w, nbins);
bo = assign2bins(vote_o, nbins);
bx = assign2bins(vote_x, nbins);
by = assign2bins(vote_y, nbins);

% note: having many nested for loops and dynamic array extensions (end+1)
% is very slow, but it makes the code easier to read
cx = []; cy = []; w = []; h = []; orient = []; count=[];
for ks = 1:nbins
  for ko = 1:nbins
    for kx = 1:nbins
      for ky = 1:nbins
        ind = bs==ks & bo==ko & bx==kx & by==ky;
        if sum(ind)>thresh
          cx(end+1) = median(vote_x(ind));
          cy(end+1) = median(vote_y(ind));
          w(end+1) =  median(vote_w(ind));
          h(end+1) = median(vote_w(ind))/objw*objh;
          % "orient" for SIFT seems to be defined differently than...
          % "orient" in display code, probably because of flipped vertical axis
          orient(end+1) = -median(vote_o(ind));
          count(end+1) = sum(ind);
        end
      end
    end
  end
end

% creates nb uniform bins within range of x and assigns each x to a bin
function b = assign2bins(x, nb)
b = min(max(ceil((x-min(x))/(max(x)-min(x))*nb), 1), nb);
```

## 3 Histogram of Oriented Gradients(40 points)

One of the pivotal ideas in computer vision was the histogram of oriented gradients (HoG), which was introduced by Dalal and Triggs to detect pedestrians [3]. In this problem, you will implement HoG and see a simple case in which it can be applied.

Download the starter code at `www.stanford.edu/class/cs231a/hw/ps3/HOG.zip`. Also, download VLFeat from `http://www.vlfeat.org/download/vlfeat-0.9.20-bin.tar.gz` and place the unzipped folder as a `vlfeat` directory inside the starter code root directory.

(a) The first part of HoG is to compute the gradient across the image. Complete `computeGradient.m` and check it on a simple image `simple.jpg` using our script in `hog.m` to verify correctness. Submit the angle and magnitude of the gradient of the center pixel, as well as the code you wrote. [**5 points**]

(b) Complete `computeHOGFeat.m`, which computes the final HoG features. Submit your code for it and the final output from `hog.m` which shows the hog features for the training images. [**15 points**]

(c) Given HoG features, we can train an SVM to recognize whatever we want. We have supplied the code to train an SVM on a large number of positive face examples of 36x36 px images and an even larger number of negative examples. Use this SVM to run a sliding window detector to find bounding boxes in `runDetector.m`. Specfically, you will be implementing the sliding window in order to find bounding boxes around positive areas on the test images. Submit code and output from `hog.m`. [**10 points**]

(d) Now that we have a large number of bounding boxes, we will notice that these boxes will cluster around the same area. Therefore, we need to implement nonmaximal suppression in `nonmaxSuppress.m`. After doing so, run the last part of `hog.m` to see the final results! Submit code and output from `hog.m`. [**10 points**]

[3] `https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf`

**Solution:**

1. `grad_angle` = 126.34 and `grad_magnitude` = 0.42355. `computeGradient.m`

```
function [ angles, magnitudes ] = computeGradient( im )
%COMPUTEGRADIENT Given an image, computes the pixel gradients
% Arguments:
%          im - an image matrix read in by im2read (size H X W X C)
%
% Returns:
%          angles - (H-2) x (W-2) matrix of gradient angles
%          magnitudes - (H-2) x (W-2) matrix of gradient magnitudes
%

height=size(im,1);
width=size(im,2);

% compute gradient in x and y direction
gradx = zeros(height-2, width-2);
grady = zeros(height-2, width-2);

for i=1:height-2
    A = im(i,2:end-1,:)-im(i+2,2:end-1,:);
    [~,index] = max(abs(A),[],3);
    absmax = zeros(size(A,1), size(A,2));
    for j =1:numel(index)
        absmax(:,j) = A(:,j,index(j));
    end
    grady(i,:)=absmax;
end
for i=1:width-2
    A = im(2:end-1,i,:)-im(2:end-1,i+2,:);
    [~,index] = max(abs(A),[],3);
    absmax = zeros(size(A,1), size(A,2));
    for j =1:numel(index)
        absmax(j,:) = A(j,:,index(j));
    end
    gradx(:,i)=absmax;
end

% compute angle and magnitude
angles=atand(gradx./grady);
angles=90 - angles;
magnitudes=sqrt(gradx.^2 + grady.^2);

% Remove redundant pixels in an image.
angles(isnan(angles))=0;
magnitudes(isnan(magnitudes))=0;
end
```

2. `computeHOGFeatures.m` and its output:

```
function [ features ] = computeHOGFeatures( im, cell_size, block_size, ...
    nbins )
%COMPUTEHOGFEATURES Computes the histogram of gradients features
```

```
3  % Arguments:
4  %          im - the image matrix
5  %          cell_size - each cell will be of size (cell_size, cell_size)
6  %                      pixels
7  %          block_size - each block will be of size (block_size, block_size)
8  %                      cells
9  %          nbins - number of histogram bins
10 % Returns:
11 %          features - the hog features of the image (H_blocks x ...
       W_blocks x nbins)
12 %
13
14 height=size(im,1)-2;
15 width=size(im,2)-2;
16
17 [angles, magnitudes] = computeGradient(im);
18
19 total_block_size = block_size * cell_size;
20 features = zeros(floor(height/(total_block_size/2)) - 1, ...
       floor(width/(total_block_size/2)) -1, nbins*block_size*block_size);
21
22 %iterate over the blocks, 50% overlap
23 for w = 1:(total_block_size/2):width-total_block_size+1
24     for h = 1:(total_block_size/2):height-total_block_size+1
25         block_features = [];
26         block_magnitude = magnitudes(h:h+total_block_size-1, ...
               w:w+total_block_size-1);
27         block_angle = angles(h:h+total_block_size-1, ...
               w:w+total_block_size-1);
28
29         % iterate over the cells
30         for i=0:block_size-1
31             for j=0:block_size-1
32
33                 cell_magnitude = ...
                       block_magnitude(i*cell_size+1:(i+1)*cell_size, ...
                       j*cell_size+1:(j+1)*cell_size);
34                 cell_angle = block_angle(i*cell_size+1:(i+1)*cell_size, ...
                       j*cell_size+1:(j+1)*cell_size);
35
36                 histograms = zeros(1, nbins);
37
38                 % iterate over the pixels
39                 for p = 1:cell_size
40                     for q = 1:cell_size
41                         ang = cell_angle(p,q);
42                         if ang >= 180
43                             ang = ang - 180;
44                         end
45                         % interpolate the votes
46                         lower_idx = round(ang / (180/nbins));
47                         upper_idx = lower_idx + 1;
48
49                         lower_ang = (lower_idx-1) * (180/nbins) + 90/nbins;
50                         upper_ang = (upper_idx-1) * (180/nbins) + 90/nbins;
51                         if upper_idx > nbins
52                             upper_idx = 1;
53                         end
54                         if lower_idx == 0
```

```
                                lower_idx = nbins;
                            end
                            lower_ang = abs(ang - lower_ang);
                            upper_ang = abs(ang - upper_ang);
                            lower_percent = upper_ang / (lower_ang+upper_ang);
                            upper_percent = lower_ang / (lower_ang+upper_ang);
                            histograms(lower_idx) = histograms(lower_idx) + ...
                                lower_percent * cell_magnitude(p,q);
                            histograms(upper_idx) = histograms(upper_idx) + ...
                                upper_percent * cell_magnitude(p,q);
                        end
                    end
                    block_features = [block_features histograms];

                end
            end
            block_features=block_features/sqrt(norm(block_features)^2+.01);
            features(ceil(h/(total_block_size/2)),ceil(w/(total_block_size/2)),:) ...
                = block_features;
        end
    end

    features(isnan(features))=0; %Removing Infinitiy values
    end
```

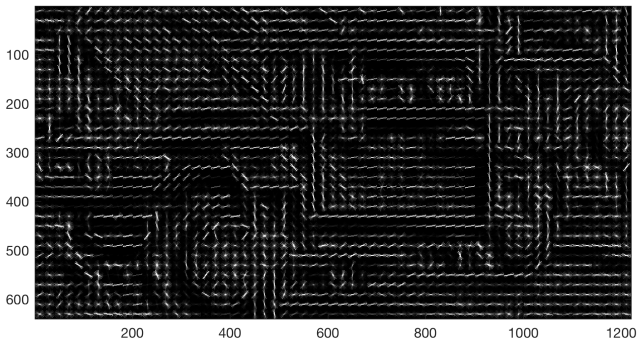



Figure 15: Extracted HOG Features from one of the training images

3. `runDetector.m` and output of `hog.m` using it

```
function [ bboxes, scores ] = runDetector( im, weight, bias, ...
    window_size, cell_size, block_size, nbins )
%RUNDETECTOR Given an image, runs the SVM detector and outputs bounding
% boxes and scores
% Arguments:
%          im - the image matrix
%          weight - weight matrix trained by SVM
%          bias - bias vector trained by SVM
%          window_size - an array which contains the height and width
%                        of the sliding window
%          cell_size - each cell will be of size (cell_size, cell_size)
%                        pixels
%          block_size - each block will be of size (block_size, block_size)
%                        cells
%          nbins - number of histogram bins
% Returns:
%          bboxes - D x 4 bounding boxes that tell [xmin ymin xmax ymax]
%               per bounding box
%          scores - the SVM scores associated with each bounding box in ...
    bboxes
%
width = window_size(2); height = window_size(1);
total_block_size = block_size * cell_size;
num_blocks = [floor((window_size(1)-2)/(total_block_size/2)) - 1, ...
    floor((window_size(2)-2)/(total_block_size/2)) - 1];

top_bboxes = [];
top_scores = [];

test_features = computeHOGFeatures(im,cell_size,block_size,nbins);
scores = zeros(size(test_features,1)-num_blocks(1), ...
    size(test_features,2)-num_blocks(2));
for h=1:size(test_features,1)-num_blocks(1)
    for w=1:size(test_features,2)-num_blocks(2)
        features = test_features(h:h+num_blocks(2)-1, ...
            w:w+num_blocks(2)-1,:);
        features = reshape(features, 1,[]);
        scores(h,w) = features*weight + bias;
    end
end

%Create Nx4 bbox and scores matrices only if positive
scores2 = reshape(scores, [],1);
a = repmat((1:size(scores,1))', size(scores,2),1)-1;
a = a * cell_size + 1;
b = floor((0:size(scores2)-1)'/size(scores,1))+1;
b = b*cell_size;
bbox = [b,a,b+width,a+height];
ind = scores2 > 1;
bbox = bbox(ind,:);
scores3 = scores2(ind,:);
top_bboxes = [top_bboxes; bbox];
top_scores = [top_scores; scores3];

% only for part d)
nmax = nonmaxSuppress(top_bboxes, top_scores, size(im));
```

```
52  top_bboxes = top_bboxes(nmax,:);
53  top_scores = top_scores(nmax,:);
54
55  bboxes = top_bboxes;
56  scores = top_scores;
57  end
```

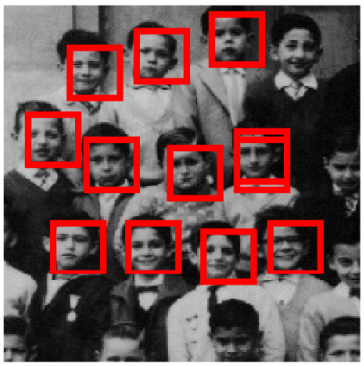


Figure 16: Bounding box output for part c)

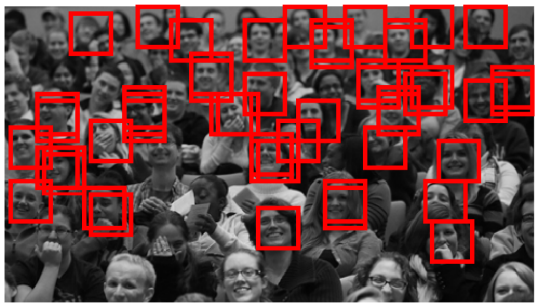


Figure 17: Bounding box output for part c)

4. `nonmaxSuppress.m` and output of `hog.m` using it in `runDetector.m`

```
function [is_valid_bbox] = nonmaxSuppress(bboxes, confidences, img_size)
% NONMAXSUPPRESS Given a list of bounding boxes, returns a subset that
% uses high confidence detections to suppresses other overlapping
% detections. Detections can partially overlap, but the
% center of one detection can not be within another detection.
% Arguments:
%          bboxes - Nx4, N is the number of non-overlapping detections, ...
    and each
%               row is [x_min, y_min, x_max, y_max]
%          confidences - Nx1 (final cascade node) confidence of each
%               bounding box.
%          img_size' - [height,width] dimensions of the image.
% Returns:
%          is_valid_bbox - a logical array of Nx1 that tells whether the
%               corresponding index of bboxes is suppressed (0) or not (1)
%


%Truncate bounding boxes to image dimensions
x_out_of_bounds = bboxes(:,3) > img_size(2); %xmax is greater than x ...
    dimension
y_out_of_bounds = bboxes(:,4) > img_size(1); %ymax is greater than y ...
    dimension

bboxes(x_out_of_bounds,3) = img_size(2);
bboxes(y_out_of_bounds,4) = img_size(1);


num_detections = size(confidences,1);

%higher confidence detections get priority.
[confidences, ind] = sort(confidences, 'descend');
bboxes = bboxes(ind,:);

% indicator for whether each bbox will be accepted or suppressed
is_valid_bbox = logical(zeros(1,num_detections));

for i = 1:num_detections
    cur_bb = bboxes(i,:);
    cur_bb_is_valid = true;

    for j = find(is_valid_bbox)
        %compute overlap with each previously confirmed bbox.

        prev_bb=bboxes(j,:);
        bi=[max(cur_bb(1),prev_bb(1)) ; ...
            max(cur_bb(2),prev_bb(2)) ; ...
            min(cur_bb(3),prev_bb(3)) ; ...
            min(cur_bb(4),prev_bb(4))];
        iw=bi(3)-bi(1)+1;
        ih=bi(4)-bi(2)+1;
        if iw>0 && ih>0
%             % compute overlap as area of intersection / area of union
%             ua=(cur_bb(3)-cur_bb(1)+1)*(cur_bb(4)-cur_bb(2)+1)+...
%                (prev_bb(3)-prev_bb(1)+1)*(prev_bb(4)-prev_bb(2)+1)-...
%                iw*ih;
```

```
54 %             ov=iw*ih/ua;
55 %             if ov > 0.3 %If the less confident detection overlaps too ...
       much with the previous detection
56 %                 cur_bb_is_valid = false;
57 %             end
58
59             %special case-- the center coordinate of the current bbox is
60             %inside the previous bbox.
61             center_coord = [(cur_bb(1) + cur_bb(3))/2, (cur_bb(2) + ...
                   cur_bb(4))/2];
62             if( center_coord(1) > prev_bb(1) && center_coord(1) < ...
                   prev_bb(3) && ...
63                 center_coord(2) > prev_bb(2) && center_coord(2) < ...
                       prev_bb(4))
64
65                 cur_bb_is_valid = false;
66             end
67
68             center_coord = [(prev_bb(1) + prev_bb(3))/2, (prev_bb(2) + ...
                   prev_bb(4))/2];
69             if( center_coord(1) > cur_bb(1) && center_coord(1) < ...
                   cur_bb(3) && ...
70                 center_coord(2) > cur_bb(2) && center_coord(2) < cur_bb(4))
71
72                 cur_bb_is_valid = false;
73             end
74
75         end
76     end
77
78     is_valid_bbox(i) = cur_bb_is_valid;
79
80 end
81
82 %This statement returns the logical array 'is_valid_bbox' back to the order
83 %of the input bboxes and confidences
84 reverse_map(ind) = 1:num_detections;
85 is_valid_bbox = is_valid_bbox(reverse_map);
```

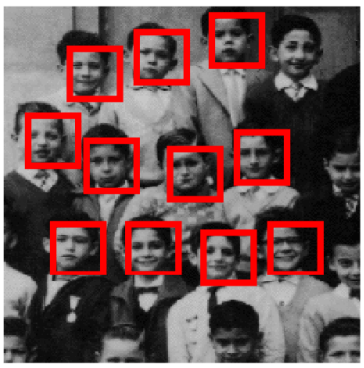


Figure 18: Bounding box output for part d)

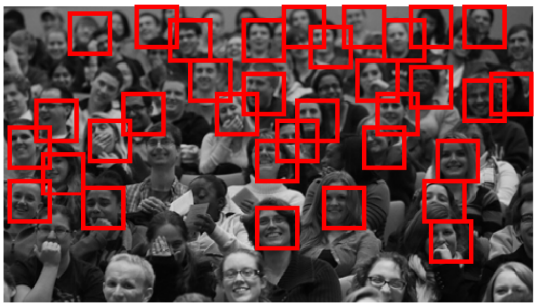


Figure 19: Bounding box output for part d)