Problem Set 3 Solution

CS231A: Computer Vision
Stanford University
Spring 2017

Chi Zhang
SUID: 06116342
Date: June 2, 2017

1 Space Carving

The following code includes implementations of `form_initial_voxels()`, `get_voxel_bounds()` and `carve()`:

```
'''
FORM_INITIAL_VOXELS  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]

    num_voxels - The approximate number of voxels we desire in our grid

Returns:
    voxels - An ndarray of size (N, 3) where N is approximately equal the
        num_voxels of voxel locations.

    voxel_size - The distance between the locations of adjacent voxels
        (a voxel is a cube)

Our initial voxels will create a rectangular prism defined by the x,y,z
limits. Each voxel will be a cube, so you'll have to compute the
approximate side-length (voxel_size) of these cubes, as well as how many
cubes you need to place in each dimension to get around the desired
number of voxel. This can be accomplished by first finding the total volume of
the voxel grid and dividing by the number of desired voxels. This will give an
approximate volume for each cubic voxel, which you can then use to find the
side-length. The final "voxels" output should be a ndarray where every row is
the location of a voxel in 3D space.
'''
def form_initial_voxels(xlim, ylim, zlim, num_voxels):
    # TODO: Implement this method!
    x_dim = xlim[-1] - xlim[0]
    y_dim = ylim[-1] - ylim[0]
    z_dim = zlim[-1] - zlim[0]
    total_volume = x_dim * y_dim * z_dim
```

```
    voxel_volume = float(total_volume / num_voxels)
    voxel_size = np.cbrt(voxel_volume)

    x_voxel_num = np.round(x_dim / voxel_size)
    y_voxel_num = np.round(y_dim / voxel_size)
    z_voxel_num = np.round(z_dim / voxel_size)

    x_coor = np.linspace(xlim[0]+0.5*voxel_size, xlim[0]+(0.5+x_voxel_num-1)*voxel_size,
                         x_voxel_num)
    y_coor = np.linspace(ylim[0]+0.5*voxel_size, ylim[0]+(0.5+y_voxel_num-1)*voxel_size,
                         y_voxel_num)
    z_coor = np.linspace(zlim[0]+0.5*voxel_size, zlim[0]+(0.5+z_voxel_num-1)*voxel_size,
                         z_voxel_num)

    XX, YY, ZZ = np.meshgrid(x_coor, y_coor, z_coor)
    voxels = np.vstack((XX.reshape(-1), YY.reshape(-1), ZZ.reshape(-1))).reshape(3, -1).T

    return voxels, voxel_size

'''
GET_VOXEL_BOUNDS: Gives a nice bounding box in which the object will be carved
from. We feed these x/y/z limits into the construction of the inital voxel
cuboid.

Arguments:
    cameras - The given data, which stores all the information
        associated with each camera (P, image, silhouettes, etc.)

    estimate_better_bounds - a flag that simply tells us whether to set tighter
        bounds. We can carve based on the silhouette we use.

    num_voxels - If estimating a better bound, the number of voxels needed for
        a quick carving.

Returns:
    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]

The current method is to simply use the camera locations as the bounds. In the
section underneath the TODO, please implement a method to find tigther bounds:
One such approach would be to do a quick carving of the object on a grid with
very few voxels. From this coarse carving, we can determine tighter bounds. Of
```

```
course, these bounds may be too strict, so we should have a buffer of one
voxel_size around the carved object.
'''
def get_voxel_bounds(cameras, estimate_better_bounds = False, num_voxels = 4000):
    camera_positions = np.vstack([c.T for c in cameras])
    xlim = [camera_positions[:,0].min(), camera_positions[:,0].max()]
    ylim = [camera_positions[:,1].min(), camera_positions[:,1].max()]
    zlim = [camera_positions[:,2].min(), camera_positions[:,2].max()]

    # For the zlim we need to see where each camera is looking.
    camera_range = 0.6 * np.sqrt(diff( xlim )**2 + diff( ylim )**2)
    for c in cameras:
        viewpoint = c.T - camera_range * c.get_camera_direction()
        zlim[0] = min( zlim[0], viewpoint[2] )
        zlim[1] = max( zlim[1], viewpoint[2] )

    # Move the limits in a bit since the object must be inside the circle
    xlim = xlim + diff(xlim) / 4 * np.array([1, -1])
    ylim = ylim + diff(ylim) / 4 * np.array([1, -1])

    if estimate_better_bounds:
        # TODO: Implement this method!
        voxels, voxel_size = form_initial_voxels(xlim, ylim, zlim, num_voxels)
        for c in cameras:
            voxels = carve(voxels, c)

        xlim = [voxels[0][0]-1.5*voxel_size, voxels[0][0]+1.5*voxel_size]
        ylim = [voxels[0][1]-1.5*voxel_size, voxels[0][1]+1.5*voxel_size]
        zlim = [voxels[0][2]-1.5*voxel_size, voxels[0][2]+1.5*voxel_size]
    return xlim, ylim, zlim


'''
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, "image", and the
        projection matrix "P".

Returns:
    voxels - a subset of the argument passed that are inside the silhouette
```

```
'''
def carve(voxels, camera):
    # TODO: Implement this method!
    # find all corresponding image points of voxels
    homo_voxels = np.hstack((voxels, np.ones((voxels.shape[0], 1)))).T
    # keep track of voxels index
    N = voxels.shape[0]
    voxel_index = np.arange(0, N)

    # project from 3D to 2D, projection matrix: (3, 4)
    P = camera.P
    img_voxels = P.dot(homo_voxels)
    # normalize
    img_voxels /= img_voxels[2, :]
    # drop out z
    img_voxels = img_voxels[0:2, :].T

    # check whether the voxel points are in range of image
    img_y_max, img_x_max = camera.silhouette.shape
    img_y_min = 0; img_x_min = 0

    voxelX = img_voxels[:, 0]
    x_range_filter = np.all([voxelX > img_x_min, voxelX < img_x_max], axis=0)
    img_voxels = img_voxels[x_range_filter, :]
    voxel_index = voxel_index[x_range_filter]

    voxelY = img_voxels[:, 1]
    y_range_filter = np.all([voxelY > img_y_min, voxelY < img_y_max], axis=0)
    img_voxels = img_voxels[y_range_filter, :]
    voxel_index = voxel_index[y_range_filter]

    # check whether the point is in the silhouette
    img_voxels = img_voxels.astype(int)
    silhouette_filter = (camera.silhouette[img_voxels[:, 1], img_voxels[:, 0]] == 1)
    voxel_index = voxel_index[silhouette_filter]

    return voxels[voxel_index, :]
```

(a) Refer to `form_initial_voxels()` and see the figure below:

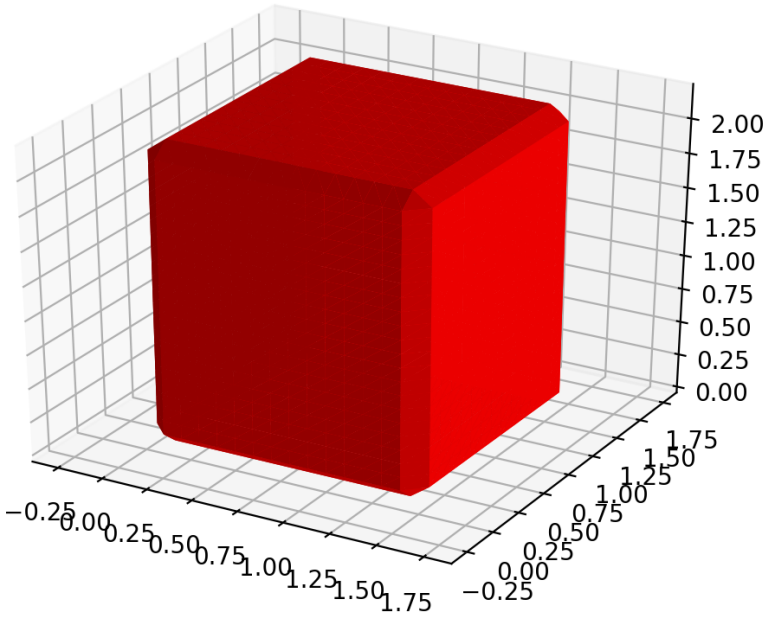


Figure 1: Initial voxel grid

(b) The figure shows what looks like after one iteration of carving:

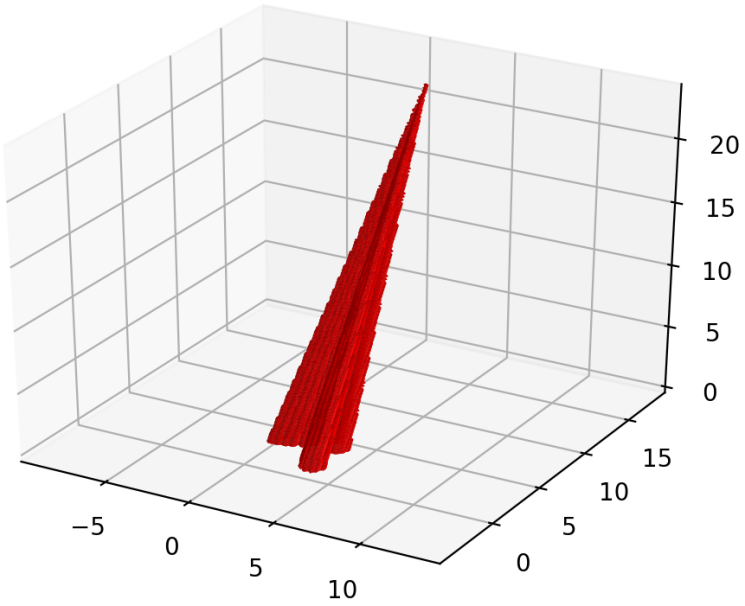


Figure 2: One iteration of carving

(c) The final output after all carvings is:

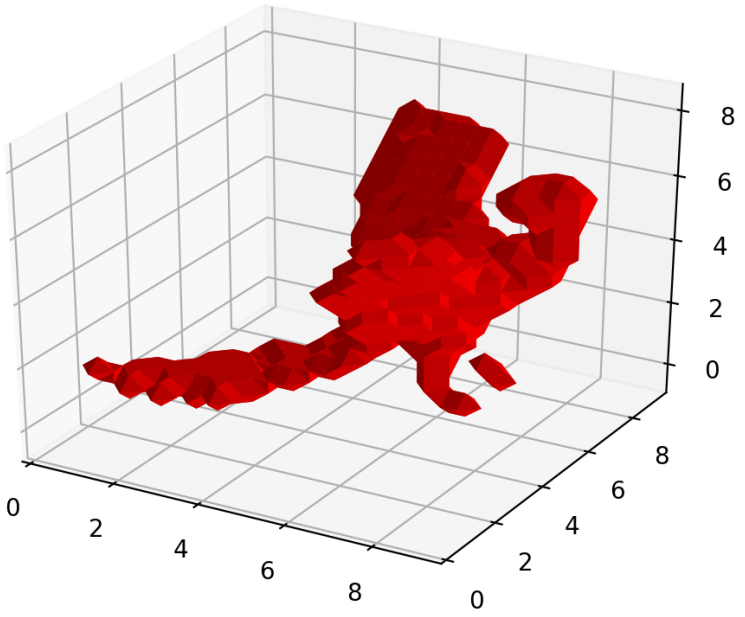


Figure 3: Non-ideal carving

(d) The finer carving is shown as below:

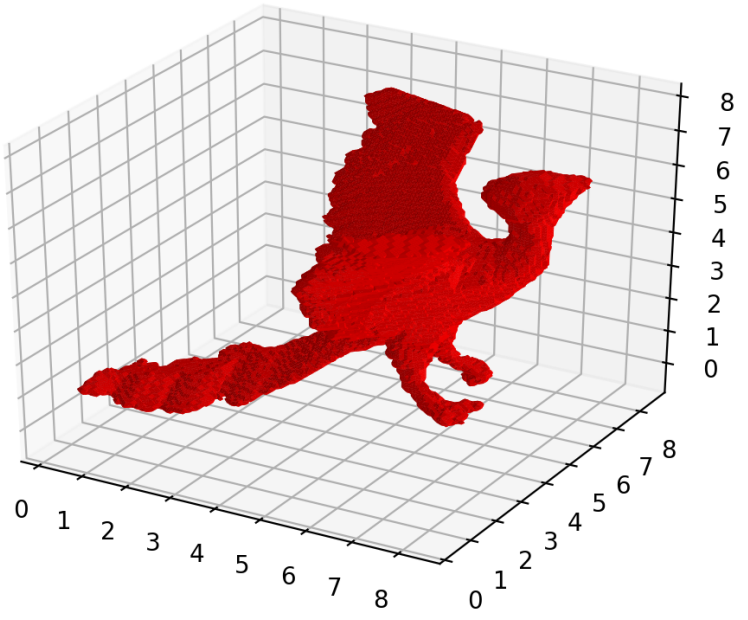


Figure 4: Finer carving

(e) My answers are:

(i) The reason lies in `estimate_silhouette()` function. This function uses a very naive and color-specific heuristic to generate the silhouette of an object, which compares the R channel of the image with both G and B channels of it. Given that the object is red-colored, the silhouette consists of all pixels which satisfy

$$(image[R] > image[B]) \wedge (image[R] > image[G])$$

In fact, this constraint depicts the general shape of the object pretty well although there are still noises in silhouette. And the noises are removed from other camera views.

(ii) If the number of camera views are reduced, then some of noises have a lower chance of being eliminated by other camera views, which leads to a coarser carving result.

(iii) These parts of the object will be missing in the final output. The reason is that if these parts are not in one view, the corresponding 3D points will be removed from voxels, even though these parts may appear in other camera views. Thus, it is better that we estimate silhouettes in a more conservative way (means noises in most cases) and turn to more camera views for removing noises. Otherwise, the carving result will end up being really coarse.

2 Single Object Recognition Via SIFT

The following code includes implementations of `compute_epipole()` and `compute_matching_homographies()`:

```
'''
MATCH_KEYPOINTS: Given two sets of descriptors corresponding to SIFT keypoints,
find pairs of matching keypoints.

Note: Read Lowe's Keypoint matching, finding the closest keypoint is not
sufficient to find a match. thresh is the threshold for a valid match.

Arguments:
    descriptors1 - Descriptors corresponding to the first image. Each row
        corresponds to a descriptor. This is a ndarray of size (M_1, 128).

    descriptors2 - Descriptors corresponding to the second image. Each row
        corresponds to a descriptor. This is a ndarray of size (M_2, 128).

    threshold - The threshold which to accept from Lowe's Keypoint Matching
        algorithm

Returns:
    matches - An int ndarray of size (N, 2) of indices that for keypoints in
        descriptors1 match which keypoints in descriptors2. For example, [7 5]
        would mean that the keypoint at index 7 of descriptors1 matches the
        keypoint at index 5 of descriptors2. Not every keypoint will necessarily
        have a match, so N is not the same as the number of rows in descriptors1
        or descriptors2.
```

```
'''
def match_keypoints(descriptors1, descriptors2, threshold = 0.7):
    # TODO: Implement this method!
    matches_list = []
    # for each row of descriptors1, find Euclidean distances to each row of descriptor2
    for i in xrange(descriptors1.shape[0]):
        descpt1 = descriptors1[i]
        euc_dist = np.sqrt(np.sum((descriptors2 - descpt1)**2, axis=1))
        index_sort = np.argsort(euc_dist, axis=None)

        closest = index_sort[0]
        second_closest = index_sort[1]
        if (euc_dist[closest] < threshold * euc_dist[second_closest]):
            matches_list.append(i)
            matches_list.append(closest)

    matches = np.array(matches_list).reshape(-1, 2)
    return matches


'''
REFINE_MATCH: Filter out spurious matches between two images by using RANSAC
to find a projection matrix.

Arguments:
    keypoints1 - Keypoints in the first image. Each row is a SIFT keypoint
        consisting of (u, v, scale, theta). Overall, this variable is a ndarray
        of size (M_1, 4).

    keypoints2 - Keypoints in the second image. Each row is a SIFT keypoint
        consisting of (u, v, scale, theta). Overall, this variable is a ndarray
        of size (M_2, 4).

    matches - An int ndarray of size (N, 2) of indices that indicate what
        keypoints from the first image (keypoints1)  match with the second
        image (keypoints2). For example, [7 5] would mean that the keypoint at
        index 7 of keypoints1 matches the keypoint at index 5 of keypoints2).
        Not every keypoint will necessarily have a  match, so N is not the same
        as the number of rows in keypoints1 or keypoints2.

    reprojection_threshold - If the reprojection error is below this threshold,
        then we will count it as an inlier during the RANSAC process.

    num_iterations - The number of iterations we will run RANSAC for.
```

```
    Returns:
        inliers - A vector of integer indices that correspond to the inliers of the
            final model found by RANSAC.

        model - The projection matrix H found by RANSAC that has the most number of
            inliers.
    '''
    def refine_match(keypoints1, keypoints2, matches, reprojection_threshold = 10,
            num_iterations = 1000):
        # TODO: Implement this method!
        # best parameters to return
        best_model = None
        best_inliers = []
        best_count = 0

        # every sample provides two constraints and
        # H is a 3x3 matrix known up to scale
        sample_size = 4
        P = np.zeros((2 * sample_size, 9)) # refer to Note 1
        for i in xrange(num_iterations):
            sample_indexes = random.sample(range(0, matches.shape[0]), sample_size)
            sample = matches[sample_indexes, :] # (4x2)

            for index, elem in enumerate(sample):
                # two keypoints indexes
                point1_index = elem[0]
                point2_index = elem[1]

                # xi' = H(xi), xi: keypoints1(ui, vi), xi': keypoints2(ui', vi')
                point1 = keypoints1[point1_index, 0:2] # (ui, vi)
                point1 = np.append(point1, 1)          # (ui, vi, 1)
                point2 = keypoints2[point2_index, 0:2] # (ui', vi')
                ui_prime = point2[0]
                vi_prime = point2[1]

                # construct P matrix
                P[2*index, :] = np.reshape(np.array([point1, np.zeros(3), -ui_prime*point1]), -1)
                P[2*index+1, :] = np.reshape(np.array([np.zeros(3), point1, -vi_prime*point1]), -1
            # solve
            U, s,VT = np.linalg.svd(P)
            H = VT[-1, :].reshape(3, 3)
            H /= H[2, 2]

            # evaluate H
            inliers = []
```

```
        count = 0
        for index, match in enumerate(matches):
            point1 = keypoints1[match[0], 0:2] # (ui, vi)
            point1 = np.append(point1, 1) # (ui, vi, 1)
            # project (ui, vi, 1) and get (ui', vi')
            # (ui, vi, 1) -H-> (u, v, w) -norm-> (ui', vi', 1) -drop-> (ui, vi)
            point2_pred = H.dot(point1)
            point2_pred /= point2_pred[2]
            point2_pred = point2_pred[0:2]
            # compare prediction and ground truth
            point2 = keypoints2[match[1], 0:2]
            err = np.sqrt(np.sum(np.square(point2 - point2_pred)))
            if err < reprojection_threshold:
                count += 1
                inliers.append(index)

        # update the best model
        if count > best_count:
            best_model = H
            best_inliers = inliers
            best_count = count

    return best_inliers, best_model

'''
GET_OBJECT_REGION: Get the parameters for each of the predicted object
bounding box in the image

Arguments:
    keypoints1 - Keypoints in the first image. Each row is a SIFT keypoint
        consisting of (u, v, scale, theta). Overall, this variable is a ndarray
        of size (M_1, 4).

    keypoints2 - Keypoints in the second image. Each row is a SIFT keypoint
        consisting of (u, v, scale, theta). Overall, this variable is a ndarray
        of size (M_2, 4).

    matches - An int ndarray of size (N, 2) of indices that indicate what
        keypoints from the first image (keypoints1)  match with the second
        image (keypoints2). For example, [7 5] would mean that the keypoint at
        index 7 of keypoints1 matches the keypoint at index 5 of keypoints2).
        Not every keypoint will necessarily have a  match, so N is not the same
        as the number of rows in keypoints1 or keypoints2.

    obj_bbox - An ndarray of size (4,) that contains [xmin, ymin, xmax, ymax]
```

```
        of the bounding box. Note that the point (xmin, ymin) is one corner of
        the box and (xmax, ymax) is the opposite corner of the box.

    thresh - The threshold we use in Hough voting to state that we have found
        a valid object region.

Returns:
    cx - A list of the x location of the center of the bounding boxes

    cy - A list of the y location of the center of the bounding boxes

    w - A list of the width of the bounding boxes

    h - A list of the height of the bounding boxes

    orient - A list f the orientation of the bounding box. Note that the
        theta provided by the SIFT keypoint is inverted. You will need to
        re-invert it.
'''
def get_object_region(keypoints1, keypoints2, matches, obj_bbox, thresh = 5,
        nbins = 4):
    # TODO: Implement this method!
    cx, cy, w, h, orient = [], [], [], [], []

    for match in matches:
        # find keypoint1 & keypoint2
        key_index1 = match[0]
        key_index2 = match[1]
        key_point1 = keypoints1[key_index1]
        key_point2 = keypoints2[key_index2]
        # keypoint 1 vars
        u1 = key_point1[0]
        v1 = key_point1[1]
        s1 = key_point1[2]
        theta1 = key_point1[3]
        # keypoint 2 vars
        u2 = key_point2[0]
        v2 = key_point2[1]
        s2 = key_point2[2]
        theta2 = key_point2[3]

        # find center, width and height of bounding box in img1
        xmin, ymin, xmax, ymax = obj_bbox
        xc1 = (xmin + xmax) / 2.0
        yc1 = (ymin + ymax) / 2.0
```

```
    w1 = (xmax - xmin) * 1.0
    h1 = (ymax - ymin) * 1.0
    # find bounding box in img2
    o2 = theta2 - theta1
    xc2 = (s2/s1)*np.cos(o2)*(xc1-u1) - (s2/s1)*np.sin(o2)*(yc1-v1) + u2
    yc2 = (s2/s1)*np.sin(o2)*(xc1-u1) + (s2/s1)*np.cos(o2)*(yc1-v1) + v2
    w2 = (s2/s1) * w1
    h2 = (s2/s1) * h1
    # add to list
    cx.append(xc2)
    cy.append(yc2)
    w.append(w2)
    h.append(h2)
    orient.append(o2)

# find min and max of vars
cx_min = min(cx)
cx_max = max(cx)
cy_min = min(cy)
cy_max = max(cy)
w_min = min(w)
w_max = max(w)
h_min = min(h)
h_max = max(h)
orient_min = min(orient)
orient_max = max(orient)
# bin sizes
cx_bin_size = (cx_max - cx_min) / float(nbins)
cy_bin_size = (cy_max - cy_min) / float(nbins)
w_bin_size = (w_max - w_min) / float(nbins)
h_bin_size = (h_max - h_min) / float(nbins)
orient_bin_size = (orient_max - orient_min) / float(nbins)

# add elements into bins
bins = defaultdict(list)
for n in xrange(matches.shape[0]):
    cx_point = cx[n]
    cy_point = cy[n]
    w_point = w[n]
    orient_point = orient[n]

    for i in xrange(nbins):
        for j in xrange(nbins):
            for k in xrange(nbins):
                for l in xrange(nbins):
```

```
                    if (cx_min+i*cx_bin_size <= cx_point
                            <= cx_min+(i+1)*cx_bin_size):
                        if (cy_min+j*cy_bin_size <= cy_point
                                <= cy_min+(j+1)*cy_bin_size):
                            if (w_min+k*w_bin_size <= w_point
                                    <= w_min+(k+1)*w_bin_size):
                                if (orient_min+l*orient_bin_size
                                        <= orient_point
                                        <= orient_min+(l+1)*orient_bin_size):
                                    bins[(i,j,k,l)].append(n)

    # parameters to return
    cx0, cy0, w0, h0, orient0 = [], [], [], [], []
    for bin_index in bins:
        indices = bins[bin_index]
        votes = len(indices)
        print votes

        if votes >= thresh:
            cx0.append(np.sum(np.array(cx)[indices]) / votes)
            cy0.append(np.sum(np.array(cy)[indices]) / votes)
            w0.append(np.sum(np.array(w)[indices]) / votes)
            h0.append(np.sum(np.array(h)[indices]) / votes)
            orient0.append(np.sum(np.array(orient)[indices]) / votes)
    print ""
    return cx0, cy0, w0, h0, orient0
```

(a) The sample image with keypoints matching is shown as below:

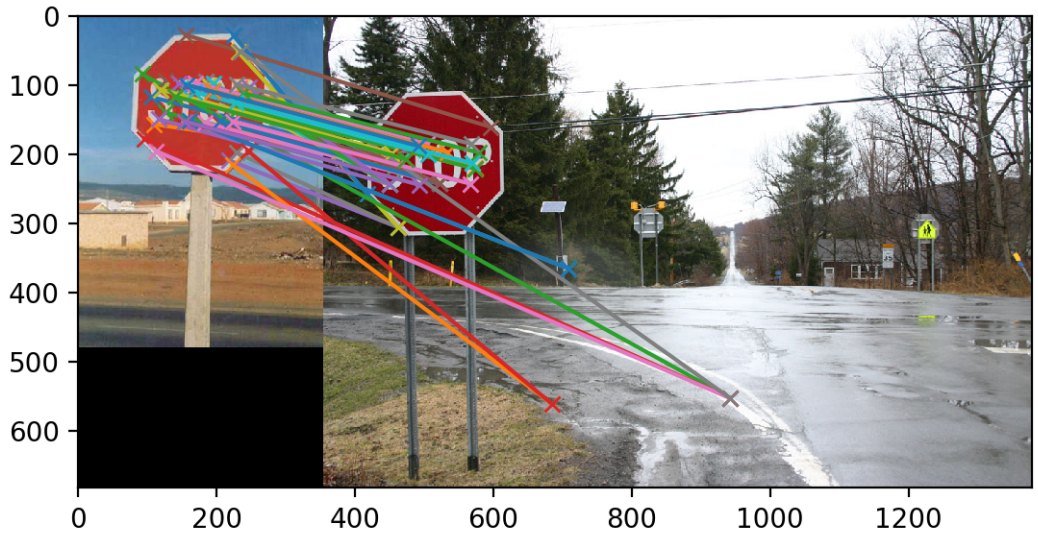


Figure 5: Raw keypoints matching without RANSAC

(b) The refined matching is like:

Figure 6: Finer keypoints matching with RANSAC

(c) Let's explore some theoretical properties of RANSAC:

(i) Given the homography H is a 3×3 matrix and known up to scale (i.e. 8 DOFs), at least 4 correspondences (every correspondence gives two constraints) are needed for solving all unknowns. Also, we assume one outlier will lead to incorrect homography, thus we need to select correspondences free of outliers 4 times with replacement:

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

(ii) This event is inverse to all of the samples will produce incorrect homography, thus

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

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

(iii) Based on conditions, we can simply find that

$$p_s = (1 - 0.15)^4 = 0.522$$

Meanwhile,

$$p = 1 - (1 - 0.522)^N \geqslant 0.99$$

$$0.478^N \leqslant 0.01$$

$$N \geqslant \frac{\log(0.01)}{\log(0.478)} = 6.239$$

Thus, we need at least 7 samples.

(d) Localize the same object in another image given the object in one image:

(i) It is intuitive to find that

$$w_2 = \frac{s_2}{s_1} w_1$$
$$h_2 = \frac{s_2}{s_1} h_1$$
$$o_2 = \theta_2 - \theta_1$$

To find x_2 and y_2, the transform between the two images is neeed:

$$\begin{bmatrix} x_2 - u_2 \\ y_2 - v_2 \end{bmatrix} = \boldsymbol{SR} \begin{bmatrix} x_1 - u_1 \\ y_1 - v_1 \end{bmatrix} (\boldsymbol{S}\text{: scale, } \boldsymbol{R}\text{: rotation})$$

$$\begin{bmatrix} x_2 - u_2 \\ y_2 - v_2 \end{bmatrix} = \begin{bmatrix} s_2/s_1 & 0 \\ 0 & s_2/s_1 \end{bmatrix} \begin{bmatrix} \cos(\theta_2 - \theta_1) & -\sin(\theta_2 - \theta_1) \\ \sin(\theta_2 - \theta_1) & \cos(\theta_2 - \theta_1) \end{bmatrix} \begin{bmatrix} x_1 - u_1 \\ y_1 - v_1 \end{bmatrix}$$

Since keypoints $(u_i, v_i, s_i, \theta_i)(i = 1, 2)$ and the bounding box in Image 1 (x_1, y_1, w_1, h_1) are known, then

$$x_2 = \frac{s_2}{s_1} \cos(\theta_2 - \theta_1)(x_1 - u_1) - \frac{s_2}{s_1} \sin(\theta_2 - \theta_1)(y_1 - v_1) + u_2$$

$$y_2 = \frac{s_2}{s_1} \sin(\theta_2 - \theta_1)(x_1 - u_1) + \frac{s_2}{s_1} \cos(\theta_2 - \theta_1)(y_1 - v_1) + v_2$$

(ii) To determine the bounding box in Image 2, we need to find six parameters. However, we can only obtain ranges of these parameters because of noise. By using Hough transform, n correspondences will vote for bins (discretized ranges of parameters) and only parameter values in bins with votes beyond the threshold will be adapted, otherwise discarded. Detailed implementation steps are:

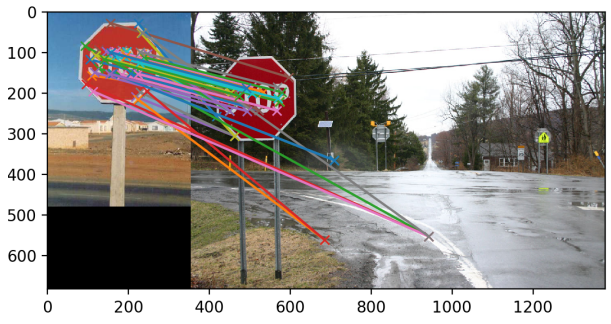
- Calculate all values of six parameters with n correspondences.
- Find min and max values of them and discretize the value ranges into four bins

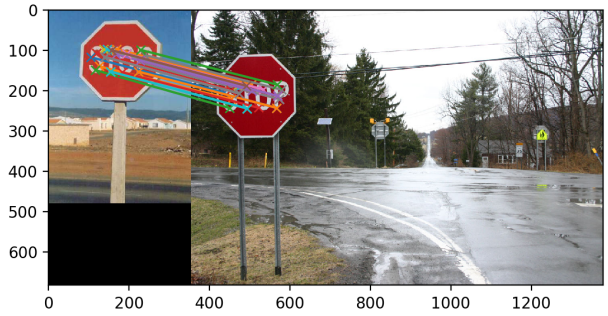
$$\text{bin size} = \frac{max - min}{4}$$

and there are four-dimensional bins we need to keep track of: x_c, y_c, w and θ of the bounding box (h is dismissed here because w/h is a constant in images). In total, there are $(4 \times 4 \times 4 \times 4 =)$ 256 bins.

- Return the medians of parameters in chosen bins whose votes exceed the threshold

(e) The matching result images are





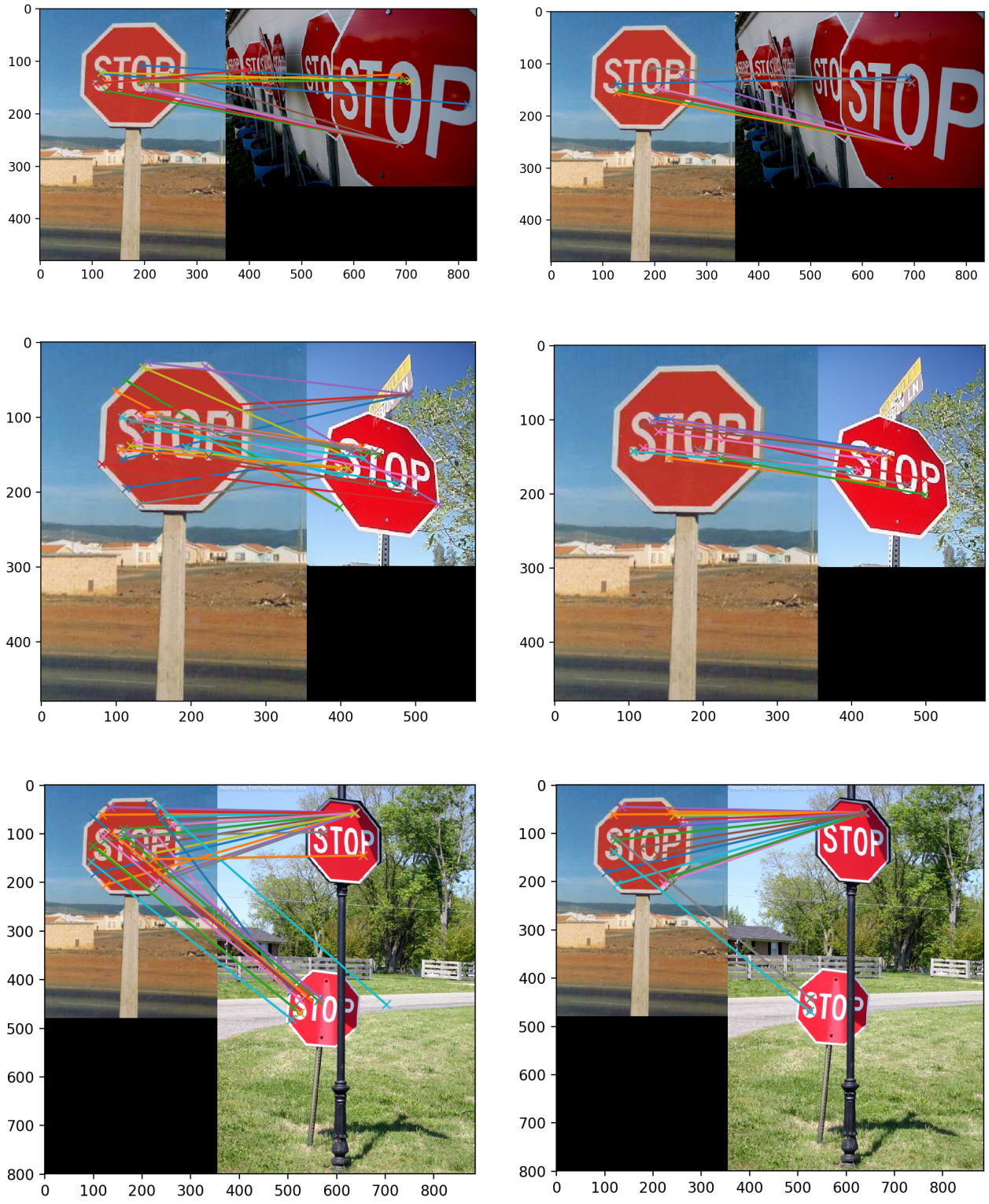


Figure 7: Coarse(left) versus finer(right) keypoint matching of the test images

For localization, it turns out that only the bounding box in the first image can be found if we use default parameters `thresh = 5, nbins = 4`. If we tweak `thresh` to be 4, then the stop signs in both the first and last images will be localized:

Figure 8: Localization results of the test images with threshold as 5

Figure 9: Localization results of the test images with threshold as 4

Discussion: as Figure 8 and 9 shown above, the threshold is one of the most important reasons in failure cases, which means none of the votes in bins exceeds the threshold.

If you print all votes of bins in the last image, you will see a 4 and several 3s. So when the threshold is set to 5, all of these parameters will be discarded but parameters with vote as 4 will be kept if the threshold is 4 instead. We can tweak `thresh` and `nbins` to get the best

result and intuitive moves are

- setting `thresh` either too low or too high will both cause you problems: too low means Hough transform does not have good ability of rejecting outliers while too high entails large number of correspondences.
- generally, the tighter bins are (smaller `nbins`), the better the solution will be.

Another possible reason is that there exist too many outliers. Even after removing some outliers by running RANSAC, there are insufficient corresponding matches to enable any vote of bins after Hough transform to reach the threshold.

Also, it should be noticed that RANSAC may have slightly different outputs sometimes in multiple runs:

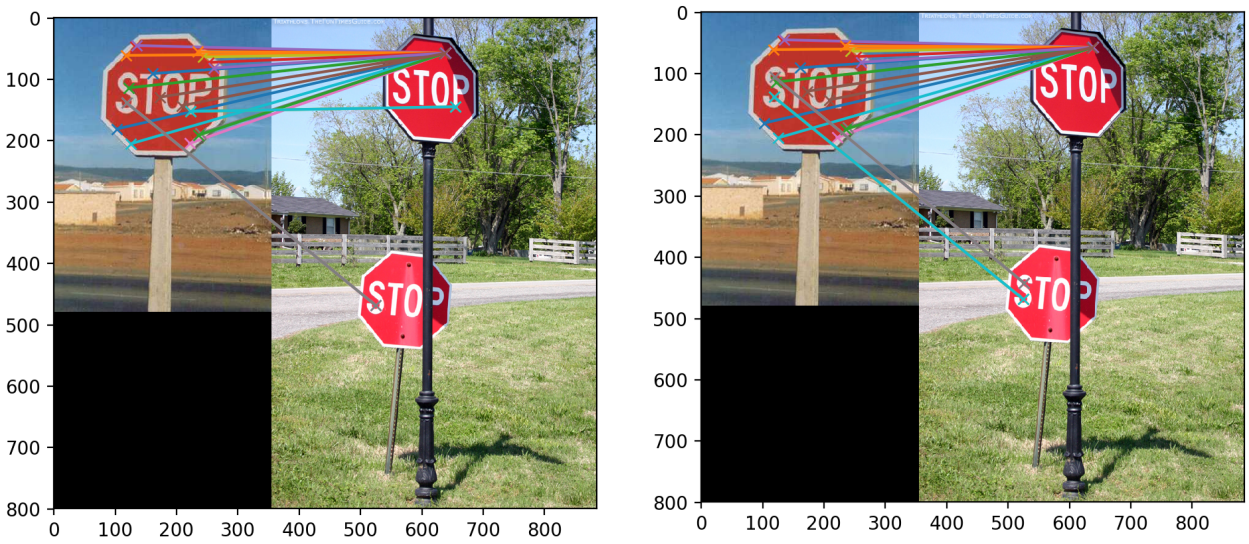


Figure 10: Minor differences between different RANSAC runs (rarely happens)

3 Histogram of Oriented Gradients

The following code includes the implementation of `compute_gradients()`, `generate_histogram()` and `compute_hog_features()`:

```
'''
COMPUTE_GRADIENT Given an image, computes the pixel gradients

Arguments:
    im - a grayscale image, represented as an ndarray of size (H, W) containing
        the pixel values

Returns:
    angles - ndarray of size (H-2, W-2) containing gradient angles in degrees
```

```
    magnitudes - ndarray of size (H-2, W-2) containing gradient magnitudes

The way that the angles and magnitude per pixel are computed as follows:
Given the following pixel grid

    P1 P2 P3
    P4 P5 P6
    P7 P8 P9

We compute the angle on P5 as arctan(dy/dx) = arctan(P2-P8 / P4-P6).
Note that we should be using np.arctan2, which is more numerically stable.
However, this puts us in the range [-180, 180] degrees. To be in the range
[0,180], we need to simply add 180 degrees to the negative angles.

The magnitude is simply sqrt((P4-P6)^2 + (P2-P8)^2)
'''
def compute_gradient(im):
    # TODO: Implement this method!
    H, W = im.shape
    angles = np.zeros((H-2, W-2))
    magnitudes = np.zeros((H-2, W-2))

    for i in range(1, H-1):
        for j in range(1, W-1):
            top = im[i-1, j]
            bottom = im[i+1, j]
            left = im[i, j-1]
            right = im[i, j+1]
            angle = np.arctan2(top-bottom, left-right) * (180/math.pi)
            magnitude = np.sqrt((left-right)**2 + (top-bottom)**2)

            if angle < 0:
                angle += 180
            angles[i-1, j-1] = angle
            magnitudes[i-1, j-1] = magnitude

    return angles, magnitudes
'''
GENERATE_HISTOGRAM Given matrices of angles and magnitudes of the image
gradient, generate the histogram of angles

Arguments:
    angles - ndarray of size (M, N) containing gradient angles in degrees

    magnitudes - ndarray of size (M, N) containing gradient magnitudes
```

```
    nbins - the number of bins that you want to bin the histogram into

Returns:
    histogram - an ndarray of size (nbins,) containing the distribution
        of gradient angles.

This method should be implemented as follows:

1)Each histogram will bin the angle from 0 to 180 degrees. The number of bins
will dictate what angles fall into what bin (i.e. if nbins=9, then first bin
will contain the votes of angles close to 10, the second bin will contain those
close to 30, etc).

2) To create these histogram, iterate over the gradient grids, putting each
gradient into its respective bins. To do this properly, we interpolate and
weight the voting by both its magnitude and how close it is to the average
angle of the two bins closest to the angle of the gradient. For example, if we
have nbins = 9 and receive angle of 20 degrees with magnitude 1, then we the
vote contribution to the histogram weights equally with the first and second bins
(since its closest to both 10 and 30 degrees). If instead, we recieve angle of
25 degrees with magnitude 2, then it is weighted 25% in the first bin and 75%
in second bin, but with twice the voting power.

Mathematically, if you have an angle, magnitude, the center_angle1 of the lower
bin center_angle2 of the higher bin, then:

histogram[bin1] += magnitude * |angle - center_angle2| / (180 / nbins)
histogram[bin2] += magnitude * |angle - center_angle1| / (180 / nbins)

Notice how that we're weighting by the distance to the opposite center. The
further the angle is from one center means it is closer to the opposite center
(and should be weighted more).

One special case you will have to take care of is when the angle is near
180 degrees or 0 degrees. It may be that the two nearest bins are the first and
last bins respectively.OA
'''
def generate_histogram(angles, magnitudes, nbins = 9):
    # TODO: Implement this method!
    histogram = np.zeros(nbins)
    bin_interval = 180 / nbins
    center_angles = np.zeros_like(histogram)
    for i in xrange(nbins):
        center_angles[i] = (0.5+i) * bin_interval
```

```
    M, N = angles.shape
    for m in xrange(M):
        for n in xrange(N):
            angle = angles[m, n]
            mag = magnitudes[m, n]
            bin_diff = np.abs(center_angles - angle)
            # when the angle is near 0 degrees
            if (180 - center_angles[-1] + angle) < bin_diff[-1]:
                bin_diff[-1] = 180 - center_angles[-1] + angle
            # when the angle is near 180 degrees
            if (180 - angle + center_angles[0]) < bin_diff[0]:
                bin_diff[0] = 180 - angle + center_angles[0]
            # find two closet bins
            bin1, bin2 = np.argsort(bin_diff)[0:2]
            histogram[bin1] += mag * bin_diff[bin2] / (180.0/nbins)
            histogram[bin2] += mag * bin_diff[bin1] / (180.0/nbins)

    return histogram

'''
COMPUTE_HOG_FEATURES Computes the histogram of gradients features
Arguments:
    im - the image matrix

    pixels_in_cell - each cell will be of size (pixels_in_cell, pixels_in_cell)
        pixels

    cells_in_block - each block will be of size (cells_in_block, cells_in_block)
        cells

    nbins - number of histogram bins

Returns:
    features - the hog features of the image represented as an ndarray of size
        (H_blocks, W_blocks, cells_in_block * cells_in_block * nbins), where
            H_blocks is the number of blocks that fit height-wise
            W_blocks is the number of blocks that fit width-wise

Generating the HoG features can be done as follows:

1) Compute the gradient for the image, generating angles and magnitudes

2) Define a cell, which is a grid of (pixels_in_cell, pixels_in_cell) pixels.
Also, define a block, which is a grid of (cells_in_block, cells_in_block) cells.
```

```
This means each block is a grid of side length pixels_in_cell * cell_in_block
pixels.

3) Pass a sliding window over the image, with the window size being the size of
a block. The stride of the sliding window should be half the block size, (50%
overlap). Each cell in each block will store a histogram of the gradients in
that cell. Consequently, there will be cells_in_block * cells_in_block
histograms in each block. This means that each block feature will initially
represented as a (cells_in_block, cells_in_block, nbins) ndarray, that can
reshaped into a (cells_in_block * cells_in_block *nbins,) ndarray. Make sure to
normalize such that the norm of this flattened block feature is 1.

4) The overall hog feature that you return will be a grid of all these flattened
block features.

Note: The final overall feature ndarray can be flattened if you want to use to
train a classifier or use it as a feature vector.
'''
def compute_hog_features(im, pixels_in_cell, cells_in_block, nbins):
    # TODO: Implement this method!
    # compute gradients for the image
    angles, magnitudes = compute_gradient(im)

    # define cells and blocks
    cell_size = pixels_in_cell
    block_size = pixels_in_cell * cells_in_block
    # compute H_blocks, W_blocks, here block serves as filters in CNN
    H, W = angles.shape
    stride = block_size / 2
    H_blocks = (H - block_size) / stride + 1
    W_blocks = (W - block_size) / stride + 1

    # construct hog feature
    hog_feature = np.zeros((H_blocks, W_blocks, cells_in_block * cells_in_block * nbins))
    for h in xrange(H_blocks):
        for w in xrange(W_blocks):
            block_angles = angles[h*stride : h*stride+block_size,
                                  w*stride : w*stride+block_size]
            block_magnitudes = magnitudes[h*stride : h*stride+block_size,
                                          w*stride : w*stride+block_size]
            # find histogram of a single block, loop over all cells in this block
            block_hog_feature = np.zeros((cells_in_block, cells_in_block, nbins))
            for i in xrange(cells_in_block):
                for j in xrange(cells_in_block):
                    cell_angles = block_angles[i*pixels_in_cell:(i+1)*pixels_in_cell,
```

```
                            j*pixels_in_cell:(j+1)*pixels_in_cell]
                cell_magnitudes = block_magnitudes[i*pixels_in_cell:(i+1)*pixels_in_cell,
                            j*pixels_in_cell:(j+1)*pixels_in_cell]
                cell_hist = generate_histogram(cell_angles, cell_magnitudes, nbins)
                block_hog_feature[i, j, :] = cell_hist

        # flattened to a vector
        block_hog_feature = np.reshape(block_hog_feature, -1)
        # normalize
        block_hog_feature /= np.linalg.norm(block_hog_feature)
        hog_feature[h, w, :] = block_hog_feature

    return hog_feature
```

OUTPUT

```
--------------------------------------------------------------------------------
Part A: Image gradient
--------------------------------------------------------------------------------
Expected angle: 126.339396329
Expected magnitude: 0.423547566786
Checking gradient test case 1: True
Expected angles:
[[ 100.30484647   63.43494882  167.47119229]
 [  68.19859051    0.           45.        ]
 [  53.13010235   64.53665494  180.        ]]
Expected magnitudes:
[[ 11.18033989  11.18033989   9.21954446]
 [  5.38516481  11.           7.07106781]
 [ 15.          11.62970335   2.        ]]
Checking gradient test case 2: True
--------------------------------------------------------------------------------
Part B: Histogram generation
--------------------------------------------------------------------------------
Checking histogram test case 1: True
Checking histogram test case 2: True
Submit these results: [ 3.435  1.915  0.95   0.8    0.45   0.9    0.     0.     0.55 ]
```

(a) The angle and magnitude of the center pixel of second test are

$$\theta = 0^\circ,\ \|\nabla pixel\| = 11$$

For details of implementation, please refer to `compute_gradient()` and **Part A** of output.

(b) Refer to `generate_histogram()` and **Part B** of output.

(c) See the figure below

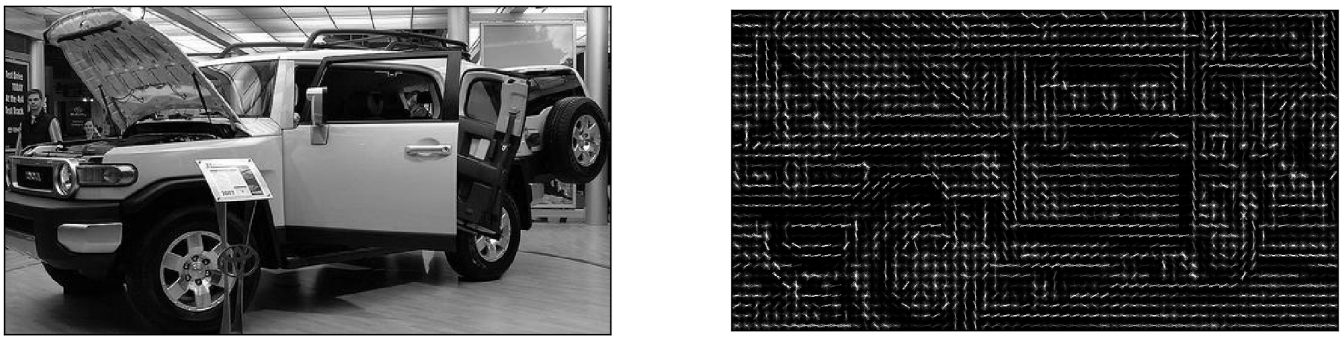


Figure 11: Extracted HOG features from an image of a car