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

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`.

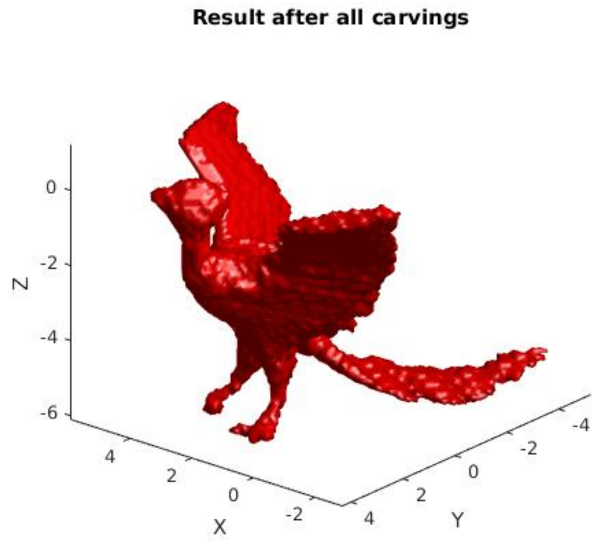


Figure 1: Our final carving using the true silhouette values

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

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

(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**]

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`.

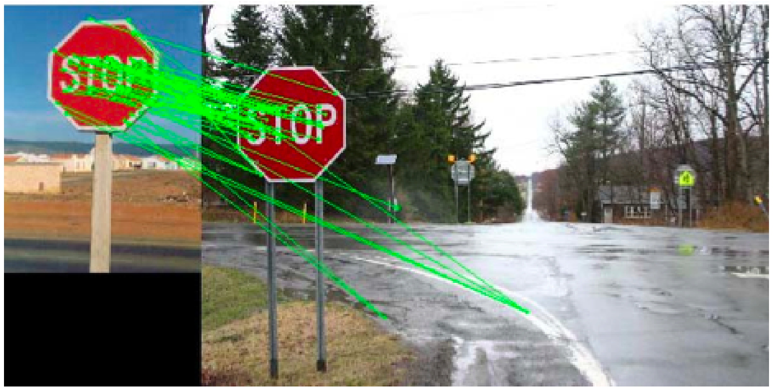


Figure 2: Sample Output, showing training image and keypoint correspondences.

(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

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

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. 2. Turn in your code and a sample image similar to Fig. 2. [**7 points**]

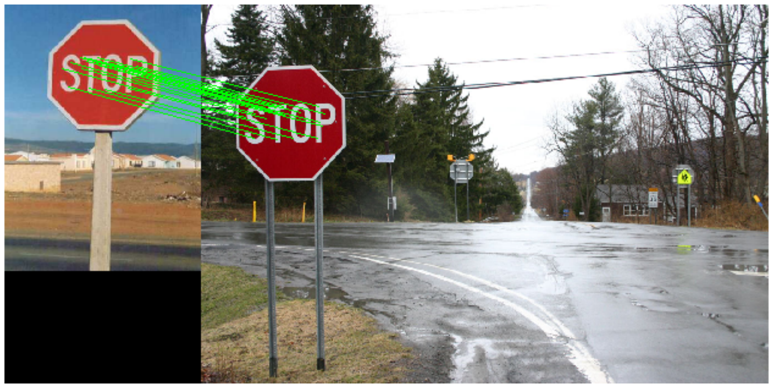


Figure 3: Sample Output, showing training image and keypoint correspondences with RANSAC filtering.

(b) From Figure 2, 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:

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

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 3.

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.

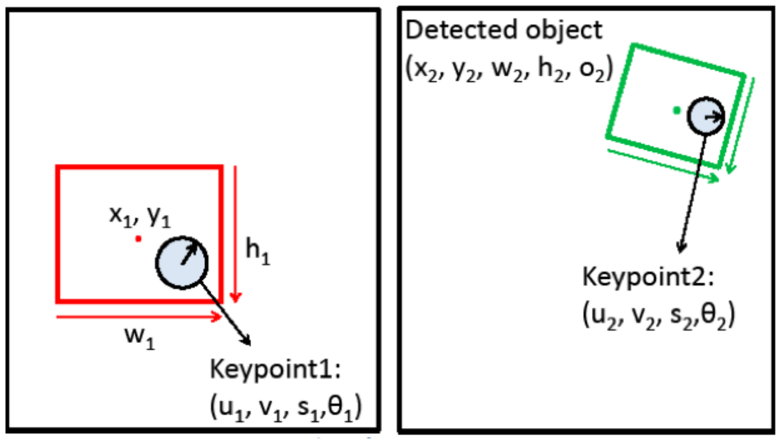
$$e = \frac{\#\text{ outliers}}{\#\text{total correspondences}}$$

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?

(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.

(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**]

(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.



x_1, y_1
h_1
w_1
Keypoint1:
$(u_1, v_1, s_1, \theta_1)$
Detected object
$(x_2, y_2, w_2, h_2, o_2)$
Keypoint2:
$(u_2, v_2, s_2, \theta_2)$

(a) Image 1 (b) Image 2

Figure 4: Two sample images for part (c)

(i) A keypoint is specified by its coordinates, scale and orientaion (u, v, s, θ). 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 4. 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**]

(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**]

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`