CS 231A Computer Vision (Spring 2017)
Problem Set 3

Due: May 17th, 2017 (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. In the starter code, you will be modifying the `main.py` file inside the `space_carving` directory.

Note: You will need the scikit-image package. Please make sure that you have this installed.

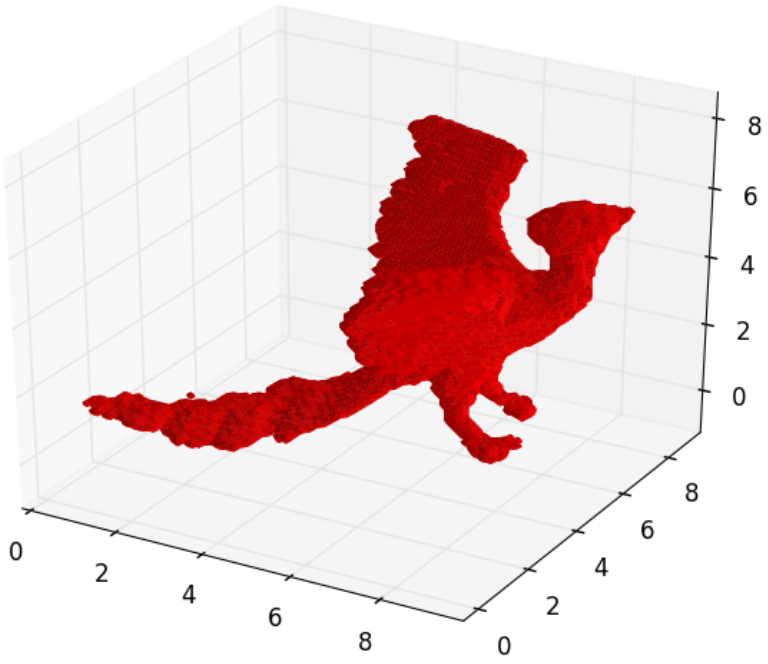


Figure 1: Our final carving using the true silhouette values

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

(a) The first step in space carving is to generate the initial voxel grid that we will carve into. Complete the function `form_initial_voxels()`. Submit an image of the generated voxel grid and a copy of your code. **[5 points]**

(b) 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. Implement this carving process in `carve()`. Submit your code and a picture of what it looks like after one iteration of the carving. **[5 points]**

(c) The last step in the pipeline is to carve out multiple views. Submit the final output after all carvings have been completed, using `num_voxels` $\approx 6,000,000$. **[5 points]**

(d) Notice that the reconstruction is not really that exceptional. This is because a lot of space is wasted when we set the initial bounds of where we carve. Currently, we initialize the bounds of the voxel grid to be the locations of the cameras. However, we can do better than this by completing a quick carve on a much lower resolution voxel grid (we use `num_voxels` $= 4000$) to estimate how big the object is and retrieve tighter bounds. Complete the method `get_voxel_bounds()` and change the variable `estimate_better_bounds` in the `main()` function to `True`. Submit your new carving, which is hopefully more detailed, and your code. **[5 points]**

(e) Finally, let's have a fun experiment. Notice that in the first three steps, we used perfect silhouettes to carve our object. Look at the `estimate_silhouette()` function implemented and its output. Notice that this simple method does not produce a really accurate silhouette. However, when we run space carving on it, the result still looks decent!

(i) Why is this the case? **[2 points]**

(ii) What happens if you reduce the number of views? **[1 points]**

(iii) What if the estimated silhouettes weren't conservative, meaning that one or a few views had parts of the object missing? **[2 points]**

2 Single Object Recognition Via SIFT (40 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]. In the starter code, you will be modifying the `main.py` file inside the `single_object_recognition` directory.

(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 `match_keypoints()` and test its performance using the given 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.

Your result should match the sample output in Figure 2. Turn in your code and a sample image similar to Figure 2. **[5 points]**

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

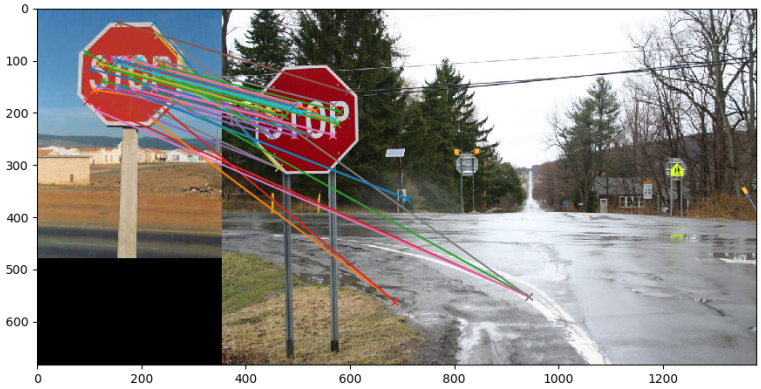


Figure 2: Sample Output, showing training image and keypoint correspondences.

(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 `refine_match()`.

Implement this RANSAC algorithm in the given function `refine_match()` and test its performance. Submit your code and the image showing the inlier matches. **[10 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.

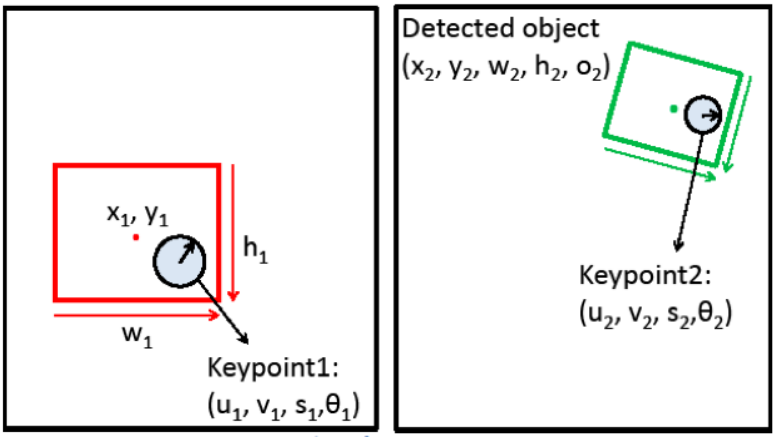
$$e = \frac{\text{\# outliers}}{\text{\#total correspondences}}$$

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? **[5 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.



(a) Image 1 (b) Image 2

Figure 3: Two sample images for part (c)

(i) A keypoint is specified by its coordinates, scale and orientation (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 3. 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. **[5 points]**

(e) Implement the function `get_object_region()` 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 (the default parameter). Your Hough space should be four dimensional.

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 code and matching result images. **[15 points]**

3 Histogram of Oriented Gradients(35 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. In the starter code, you will be modifying the `main.py` file inside the `hog` directory.

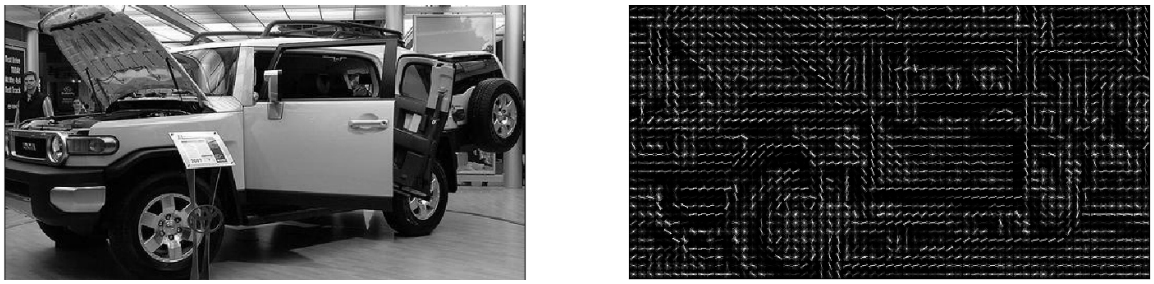


Figure 4: Extracted HOG features from an image of a car.

(a) The first part of HoG is to compute the gradient across the image. Complete the function `compute_gradient()` and check it on a simple image to verify correctness. Submit the angle and magnitude of the gradient of the center pixel of the second test, as well as the code you wrote. [**10 points**]

(b) The next step of HoG is to compute the histograms for a given grid of gradients. Fill out the `generate_histogram()` method and verify that it works on our unit test. Submit your code and the histogram you get on the test case. [**10 points**].

(c) Complete `compute_hog_features()`, which computes the final HoG features. Submit your code and the final output, which displays a visual representation of the features. [**15 points**].

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