
(a) Object template image.  
(b) Test image.

(c) SIFT keypoint matches.  
(d) Homography constrained matches (“inliers”).

(e) Recovered 3D pose and augmented views with the Stanford bunny.

Interest point detectors and descriptors are at the heart of many of the more successful applications of computer vision. SIFT in particular has been a boon to vision research since its appearance in Lowe’s paper [2].

The goal of this assignment is to explore the use of interest points in several applications. We will be revisiting some of the camera geometry we used way-back-when, and apply it now to do augmented reality and object pose estimation. Along the process, we will try to make some pretty pictures that you can keep in your wallet and show off at parties and such.
1 Harris corners

A simple (and yet very useful\[1\]) interest point detector is the Harris corner detector. Implementing this is quite straightforward and we’d hate for you to miss out on the opportunity:

For every pixel, we compute the Harris matrix

\[
H = \begin{pmatrix}
\sum I_x I_x & \sum I_x I_y \\
\sum I_y I_x & \sum I_y I_y
\end{pmatrix}
\]

(1)

where the sums are computed over a window in a neighborhood surrounding the pixel, and will be weighted with a Gaussian window. Remember that this matrix will be calculated efficiently using convolutions over the whole image. The “cornerness” of each pixel is then evaluated simply as \(\text{det}(H) - k (\text{trace}(H))^2\) (easily vectorized).

Refer to your notes—the algorithm is described in full detail. I also encourage you to revisit the derivation, it’s really quite beautiful how it all works out in the end.

Figure 1: Harris corners. The importance of scale is apparent in the bottom close-up views. (Zoom in to see.)

(a) First 500 detected corners (red dots). (b) “Cornerness” heatmap (darker is cornier).

(c) Close-up view, one set of parameters. (d) Close-up view, a different set.

Q1.1 Discuss the effect of the parameters \(\sigma_x\) (the Gaussian sigma parameter used when calculating image gradients), \(\sigma_y\), (the sigma for the neighborhood window around

\[1\]A cornerstone method, one could say. *groan*
Q1.2 Implement your own brand of Harris corner detection. (You can choose the parameters as you wish, or refer to the paper [1], with \( k = 0.04 \) a typical value (empirical)). Follow the skeleton given in \texttt{harris.m} and \texttt{pi_script.m}. Save the resulting images as \texttt{q1_2_heatmap.jpg} and \texttt{q1_2_corners.jpg}.

2 SIFT keypoints and descriptor

Because you’d have no time to explore the applications if we ask you to implement SIFT, we have provided you with Andrea Vedaldi’s implementation of SIFT interest point detection, and SIFT descriptor computation. This function and many others can be found in Vedaldi’s (and contributors) excellent \texttt{VLFeat} library [3]. I encourage you to browse the full package, it has implementations for many computer vision algorithms, many at the cutting edge of research (for this assignment though, we’ll ask that you only use the functions provided).

I also suggest—if you’re interested—that you study Lowe’s paper in detail, especially two concepts that you won’t easily find outside of computer vision: (1) the scale-space, and (2) using histograms of gradient orientations as a descriptor of a field.

The function is used like this (after doing \texttt{addpath ./sift}):

\[
\texttt{[keypoints1,descriptors1] = sift(double(rgb2gray(im1))/255, 'Verbosity', 1)};
\]

Verbosity is not necessary but the function takes a while to compute. (You can save the results after running once for each image.) See \texttt{sift_demo.m}

3 Image matching, revisited.

This is similar to the approach we followed in homework 2. Instead of using a bank of filter responses, we will now use the SIFT descriptor as our feature vector, and instead of computing features per image pixel, we will only compute the features at the scale-invariant interest points\(^2\).

This is going to be a very crude object detector—an objet d’art detector, in fact. For each training image (in \texttt{objs/}), we will compute SIFT points and descriptors. Then, for each test image (in \texttt{images_test/}), we will do the same. Because our dataset is so small, we will directly compare all the test SIFT descriptors with all the training descriptors for each training image. We will label the testing image as the image with most.

Q3.1 Write the function \texttt{[matches, dists]=matchsift(D1, D2, th)} that matches the set of descriptors \( D_1 \) to \( D_2 \) (follow the skeleton file given). Use \texttt{pdist2} to compute distances. \( \texttt{matches} \) is \( 2 \times K \) where, \( \texttt{matches}(i, k) \) is the index in the keypoint array of image \( i \), of the k-th match or correspondence, for a total of K matches.

\(^2\)Sometimes SIFT descriptors are computed densely for every pixel, similarly to homework 2.
To make matching a bit more robust, we will implement a commonly heuristic: a descriptor \( i \) in \( D_1 \) will be said to match \( j \) in \( D_2 \) if the ratio of the distance to the closest descriptor (call it \( \text{dist}(i,j) \)) and second-closest descriptor in \( D_2 \) (call it \( \text{dist}_2(i,j) \)) is smaller than \( \alpha \) (i.e., \( \frac{\text{dist}(i,j)}{\text{dist}_2(i,j)} \leq \alpha \)). A common value is \( \alpha = 0.8 \). Note: Vedaldi provides a similar \textit{siftmatch} function, you can use it to test but you are expected to implement your own.

Q3.2 For the 6 test images (in \textit{images_test/}), display the matches side by side with the training image (in \textit{objs/}) with most matches, as below. (Use the function \textit{plotmatches(im1, im2, keypoints1, keypoints2, matches)}, where \textit{matches} is \( 2 \times N \) as above.). Save these images as \texttt{q3_2_match#.jpg}, substituting the testing image number.

Figure 2: SIFT keypoint matches. Image: “Starry night”, from Van Gogh.

4 Homographies, revisited.

In homework 1, we used homographies to create panoramas given two images, but we had to (tediously) mark the corresponding points manually. Now, with SIFT and RANSAC in our bag of tricks, we can find these correspondences automatically. Our target application will be augmented reality—a fancy way of saying that we will embed synthetically generated elements from a virtual world into real-world images.

The RANSAC algorithm can be applied quite generally to fit any model to data. We will implement it in particular for (planar) homographies between images:

1. Select a random sample of 4 (tentative) correspondences from the total \( K \) (use the function \textit{randsample}).

2. Fit a homography to these. (A tentative model.) Use \texttt{computeH_norm}.

3. Evaluate the model on all correspondences. All matches that the model fits with an error smaller than \( t \) make up the “consensus set”. Elements in this set are called \textit{inliers}. To determine whether a match fits the model, evaluate the error in pixels of the projective relation: \( p_1 \equiv H_{2 \rightarrow 1} p_2 \).

4. If \( \text{numInliers} \geq d \), fit a homography to the consensus set.
5. Evaluate the average fitting error on the consensus set. We will use mean distance between actual and predicted matches, in pixels. If the error is smaller than the best homography found up till now, update the current best.

Q4.1 Implement RANSAC. The function should have the following fingerprint

\[
\text{[bestH2to1, bestError, inliers]} = \text{ransacH2to1(keypoints1, keypoints2, matches)}.\]

bestH2to1 should be the homography with least error found during RANSAC, along with this error bestError. H2to1 will be a homography such that if \( p_2 \) is a point in keypoints2 and \( p_1 \) is a corresponding point in keypoints1, then \( p_1 \equiv H p_2 \). inliers will be a vector of length \( \text{size(keypoints1,:)} \) with a 1 at those positions that are part of the consensus set, and 0 elsewhere. We have provided \( \text{computeH_norm} \) to compute the homography, but feel free to use your own implementation from homework 1.

Image matching with geometric constraints

In homework 2, you used the spatial pyramid match kernel as a way to use (very coarse) spatial relations between image features. If a more accurate model of the object is known (say, it is planar), the geometric constraint that this imposes on the matched features can be used to reduce false positives.

Q4.2 Create the 6 testing images showing the image with most matches after determining the set of inliers. Count only inliers as matches this time, and save these images as q4_1_ransac#.jpg

Figure 3: Keypoint matches after RANSAC with a homography (inliers only).

Note that neither matching nor homography estimation are symmetric. You will get different results (and different problems) depending on which order you use. Most common is to match the testing image against the training data, but you can try and see which one works best.

Augmented reality

Assume that the template image points lie on a 3D plane with \( Z = 0 \), and that the \( X \) and \( Y \) axes of this 3D plane are aligned with the \( x \) and \( y \) axes of the image plane. Thus, the 3D position of our interest points is then \( \mathbf{P}_i = (x_i - \mu_x, y_i - \mu_y, 0, 1)^T \) (in homogeneous coordinates) where \( x_i \) and \( y_i \) are the coordinates of the keypoints in the template image.
The vector \( \mu \) (or \( \text{mu} \)) is a centering vector given in \texttt{intrinsics.m}, it merely centers the points such that the origin of the object’s coordinate system lies at the center of the image. The camera matrix \( \mathbf{K} \) is also given in \texttt{intrinsics.m}.

Note that you should now recompute your matrix \( \mathbf{H} \) with the centered points, but you can use the same inlier set you computed above.

**Q4.3** Write the relation between \( \mathbf{P}_i \) and the imaged points in our target image, \( (u_i, v_i, 1)^T \), in terms of the camera matrix \( \mathbf{K} \), and the camera rotation \( \mathbf{R} \) and translation \( \mathbf{t} \). Use homogeneous coordinates and projective equivalence.

**Q4.4** Write an expression for the columns of a matrix \( \mathbf{A} \) such that \( \mathbf{H} \equiv \mathbf{KA} \). This should be in terms of the rows and/or columns of \( \mathbf{R} \) and \( \mathbf{t} \).

**Q4.5** Devise a way to (approximately) recover \( \mathbf{R} \) and \( \mathbf{t} \) from \( \mathbf{H} \) given \( \mathbf{K} \). Explain it. Hint: use the properties of a rotation matrix.

**Q4.6** Implement the method you devised in Q4.5 as \( [\mathbf{R}, \mathbf{t}] = \text{approxRtFromH}(\mathbf{H}) \) where \( \mathbf{R} \) is \( 3 \times 3 \), \( \mathbf{t} \) is \( 3 \times 1 \), and \( \mathbf{H} \) is \( 3 \times 3 \).

Remember that there are a number of ambiguities that you may need to resolve (e.g., the object should face the camera and be in front of it). This can get tricky! You may want to ignore this issue at first.

**Q4.7** Overlay the image axes and the outline of the image onto the picture. (Project the corresponding object points with your \( \mathbf{K}, \mathbf{R}, \) and \( \mathbf{t} \) matrices, and plot them.) Save at least 3 of these images as \texttt{q4_7_axes#.jpg}.

**Q4.8** Overlay the image axes and the outline of the image onto the picture. Draw the Stanford bunny landing on the image at the estimated 3D position. (Project the corresponding object points with your \( \mathbf{K}, \mathbf{R}, \) and \( \mathbf{t} \) matrices, and plot them, using the (rotated, translated) \( Z \) coordinate now as well. See \texttt{displaybunny.m}). Save at least 3 of these images as \texttt{q4_7_bunny#.jpg}.

Figure 4: Estimated pose and bunny augmented image.

Also included is a teapot model, feel free to use that instead, or any other augmentation that shows that it is working. There are also some extra example images, in case you want to experiment with more difficult cases, or the creation of other mixed reality images.
Figure 5: Estimated pose and teapot augmented image. Image: Pablo Picasso, *La Lectrice*.

References

