CS1114 Assignment 5, Part 2

out: Wednesday, April 10, 2013.
due: **Friday, April 19, 2013, 5PM.**

This assignment covers two topics in two parts: interpolation (Part 1), and feature-based image recognition (Part 2). This document contains Part 2. Stub functions for the code you need to write can be found in `/courses/cs1114/student_files/A5P2`. Please copy these to your working directory. You'll also need to add library directories to your Matlab path for this assignment to get access to the `sift` functions and `sift_gui`. Please add `/courses/cs1114/lib/lab5` and `/courses/cs1114/lib/lab5/sift`.

In this part, you'll be using local features for object recognition. The ultimate goal is to have a robot be able to find an object in the environment using feature matching. This approach to object recognition has three basic parts: feature extraction, feature matching, and fitting a transformation. To test your functions, you will use the provided `sift_gui` interface; we also provide several test images, including `template.png`, `template2.png`, and `search.jpg` (another set of test images, `video_*`, is derived from a "find a DVD title in a large stack" application). The function stubs and images are in the directory `/courses/cs1114/student_files/A5P2/`. Figure 1 shows an example object we might want to detect, and an image we might want to detect it in. Note that the object appears at a different location, orientation, and scale in the search image; we'll have to deal with this, and part of the strategy will be to search for an affine transformation that relates the two images.

(a) Template image

(b) Search image

Figure 1: A pair of images we might wish to match. Note that the object in the template image appears at a different location, scale, and orientation in the search image, which complicates our lives.

The `sift_gui` function has two modes: one ("Our code") uses the instructors' implementation of the assignment, and one ("Your code") uses the code you are writing. You can use our version as a benchmark for testing your version.

1 Feature detection

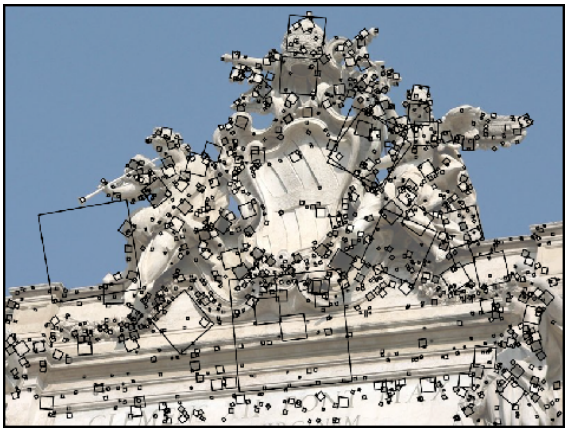

Figure 2: SIFT features detected in an image.

In this assignment you will be using SIFT (Scale-Invariant Feature Transform) to detect features in an image. We provide you with a function in Matlab called `sift` (courtesy of Andreas Veldaldi).[1] The `sift` function takes in a grayscale image (in double format), and returns two matrices, a set of feature *coordinate frames* and a set of feature *descriptors*:

```
>> img = im2double(imread('template.png'));
>> [frames, descriptors] = sift(img);
```

If SIFT detects n features, then `frames` is a $4 \times n$ matrix, and `descriptors` is a $128 \times n$ matrix. Each coordinate frame (column of the `frames` matrix) describes the 2D position, scale, and orientation of a feature (note that we will only be using the 2D position for this assignment). Each feature descriptor (column of the `descriptors` matrix) is a 128-dimensional vector describing the local appearance of the corresponding feature. Local features corresponding to the same *scene point* in different images should have similar (though not identical) feature descriptors. Figure 2 shows an example of detected SIFT features in an image.

You can use the provided function `plotsiftframes` to visualize a set of extracted SIFT features. You'll want to first display the original image before plotting the frames, e.g.:

```
>> imshow(img); plotsiftframe(frames);
```

[1] If you try running SIFT on your own machine, you will likely get an error—you'll first need to "mex" some functions. Try running `sift_compile`, and ask the TAs if you need help.

2 Feature matching

Your work begins with the second step in our object recognition pipeline: feature matching. (Note that you may have already work on part of this in a previous section, specifically the nearest neighbor and ratio distances. We will revisit those steps here.) Suppose we are given a reference template image of an object, and a second image in which we want to find the same object; an example pair of images was shown in Figure 1. To do so, we need to find pairs of features in the two images that correspond to the same point on the object; these should be features with similar descriptors. Thus, feature matching involves finding features with similar descriptors. We'll use *nearest neighbor* matching for this task: for each feature in image 1 (the template image), we'll find the most similar feature in image 2 (the search image), that is, the feature with the most similar descriptor. We'll define the distance between two descriptors a and b (two 128-dimensional vectors) using the standard Euclidean distance:

$$\texttt{distance}(a, b) = \sqrt{\sum_{i=1}^{128} (a_i - b_i)^2}$$

Assuming a and b are column vectors, this formula can be written using vector operations as:

$$\texttt{distance}(a, b) = \sqrt{(a-b)^T (a-b)}$$

where ^{T} denotes the transpose operator (' in MATLAB).

Your first task is to write a function `match_nn` to find nearest neighbors. This function will take in two sets of descriptors, `descs1` and `descs2` (with n and m descriptors, respectively), and will return a $3 \times n$ matrix, i.e., a column for each descriptor in `descs1`. The first two rows of this matrix will be *pairs of indices* of matching features, and the third row will be the distances between the matching feature descriptors. For instance, the first column of this matrix might be `[ 1; 27; 1.5 ]`; this means that feature 1 in image 1 is closest (in terms of its descriptor) to feature 27 in image 2, and the distance between the descriptors is 1.5.

There are many ways to write this function; we'll reserve a bit of extra credit for implementations that are much faster than the simplest approach.

$\Longrightarrow$ Write the function `match_nn`.

This matching procedure will undoubtedly return many false matches. One way to reduce the number of false matches is to threshold the matches by distance between descriptors. You will do this by writing a function `threshold_matches`, which takes in a set of matches (the output of the function `match_nn`) and a threshold, and returns a new matrix of matches whose distances are less than the threshold.

$\Longrightarrow$ Write the function `threshold_matches`. This function can be written in one line of Matlab code (not including comments), though this is not a requirement.

It turns out that the even this thresholding doesn't work that well. Consider the example image pair shown in Figure 3. There are a lot of repetitive features in these

Figure 3: Example of images with repetitive features.

images, and all of their descriptors will look similar. To find *unique*, distinctive feature matches, consider the following strategy: for each descriptor a in image 1, find the *two* nearest neighbors in image 2. Call these b and c, and let their distances from a be `distance`(a, b) and `distance`(a, c). If `distance`(a, b) is much smaller than `distance`(a, c), then b is a much better match than even the second closest feature. Thresholding using this test will tend to get rid of features with multiple possible matches. To make this concrete, we'll define our new distance function between a and b as the *ratio* of the distances to the two nearest neighbors.

$$\frac{\texttt{distance}(a, b)}{\texttt{distance}(a, c)}.$$

We'll call this the ratio distance. You'll now write a function `match_nn_ratio` that does the exact same thing as `match_nn`, except that it returns the ratio distance in the third row of the output matrix.

$\Longrightarrow$ Write the function `match_nn_ratio`. You'll need to find the top two nearest neighbors in this function. Your function must spend at most $O(n)$ time finding the top two neighbors for each feature in the first image, where n is the number of features in the second image.

You can now use your `threshold_matches` function on the output of this function. Note that the distances are now ratios rather than pure distances, so the "units" are different (for instance, the ratio distance will be always be between 0 and 1). For the ratio distance, a threshold of 0.6 usually works pretty well.

To visualize a set of matches, you can use a function we provide to you called `plotmatches`. This function takes in two images, two sets of frames, and your matches, and draws a figure where you can see the matches. If you provide `'interactive', 1` as the last two arguments to this function, you will get an interactive window where you can click on features to see their matches. Here's an example call to `plotmatches`, assuming that our images are stored in variables `template` and `search`, and our frames in `frames1` and `frames2`:

```
>> plotmatches(template, search, frames1, frames2, matches, 'interactive', 1)
```

This will probably look very messy unless you threshold the matches first, and pass in that thresholded set. An example of what we get from our (thresholded matches) on some of the test images is shown in Figure 4.

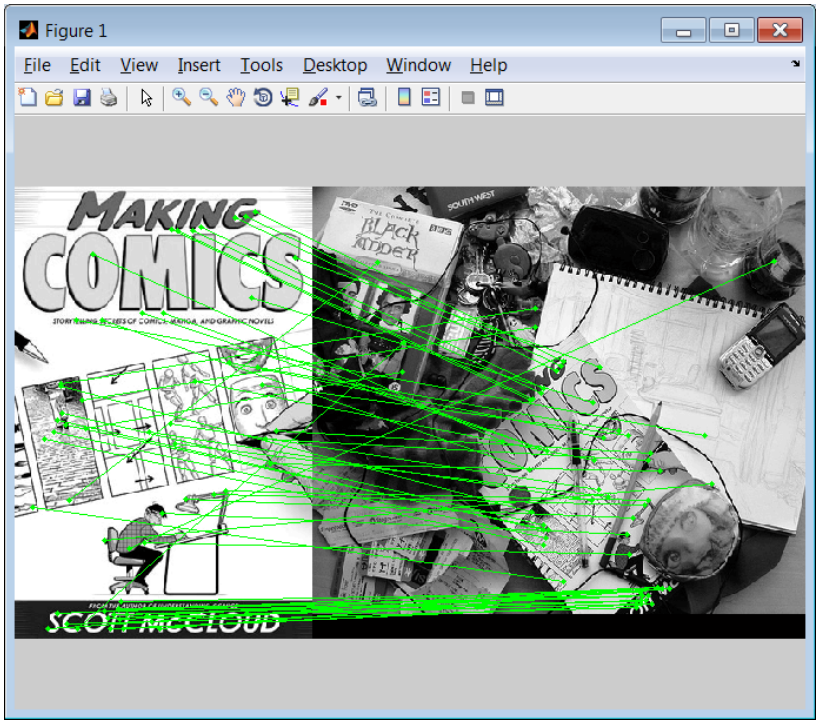

Figure 4: Matches visualized with `plot_matches`.

Speeding up SIFT. Using it's default parameters, SIFT can be fairly slow. One reason is that SIFT builds an image pyramid and searches every different level of the pyramid for features (incidentally, this is one way SIFT achieves scale invariance). To speed things up, we can tell SIFT to skip the first couple levels. To do this, provide `'FirstOctave', 1` as the last two arguments to the `sift` function, e.g:

```
>> [frames, descs] = sift(im2double(img), 'FirstOctave', 1);
```

The default value of `'FirstOctave'` is -1, so this tells SIFT to skip the first two levels of the pyramid (also known as octaves, like in music—the fact that the name is the same is not coincidental, but this is outside of the scope of this assignment). Adding this argument will result in many fewer features, however, which might also be a problem if too few are detected.

3 Finding the transformation between two images

The final step is to find a transformation mapping the features in image 1 to the corresponding features in image 2. This is the most complicated of the steps. We will be using the RANSAC algorithm to find a transformation despite the existence of (potentially many) false matches. The first step is to write a function that takes three matches (that you will sample at random) and fits an affine transformation T, where

$$T = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}.$$

Recall from class that this problem is equivalent to solving two linear systems with three equations each, and that a linear system can be represented by a matrix equation

$$Ax = b.$$

Where x is the unknown variables, Ax represents the left-hand-side of the linear system, and b represents the right-hand-side. The solution to the linear system is then:

$$x = A^{-1}b.$$

You will do this fitting inside of a function called `fit_affine_transform`. This function will take in two 2×3 matrices, `P` and `Q`, where the columns of `P` are 2D coordinates in the first image and the columns of `Q` are the corresponding 2D coordinates in the second image. This function will return a 3×3 affine transformation matrix `T`:

```
function T = fit_affine_transform(P, Q)
```

$\Longrightarrow$ Write the function `fit_affine_transform`.

Next, we'll use this function inside of the RANSAC algorithm. Recall that RANSAC works as follows:

RANSAC(input: set of matches):

1. Randomly select three matches.
2. Solve for the corresponding transformation T.
3. Compute the number of inliers to T.
4. If T has the most inliers so far, save it.
5. Repeat steps 1-4 N times, then return the best T.

You'll be implementing RANSAC in a function called `ransac_affine`. This function will take in five parameters: the frames of the two images, the matches resulting

from your matching and thresholding code, the inlier threshold, and the number of rounds. RANSAC will return two outputs: the final transformation `T`, and a matrix `inliers` containing the matches that are inliers to this transformation:

```
function [T, inliers] = ransac_affine(frames1, frames2, matches, threshold, rounds)
```

The builtin function `randint` might be helpful here.

We will give a small amount of extra credit if your solution uses least squares to refit a better affine transformation to all of the inlier matches.

$\Longrightarrow$ Write the function `ransac_affine`.

Next, you should put all of these steps (SIFT, matching, RANSAC) together in a function called `detect_object`. This function takes two images, `search` and `template`, and returns two outputs: a logical value `detected` (which is 1 if the object is detected and 0 otherwise), and a transformation `T` (if the object is not detected, then `T` could contain anything, e.g., all zeros). It's up to you to decide how to tell if the object is detected.

$\Longrightarrow$ Write the function `detect_object`.

We can now finally write a function to draw the outline of a template object image in a search image. To do so, we'll transform the four corners of the template image using the affine transform you solved for, then draw a polygon connected the four transformed points. You'll be writing this in a function called `draw_transformed_object`. This function will be called from an interface we provide called `sift_gui`, which you can use to test your entire pipeline. The GUI will call your `draw_transformed_object` function (in fact, it will also call your `detect_object` function).

To draw lines in Matlab, you can use the regular `plot` function. For instance:

```
plot(canvas, [x1; x2], [y1; y2], 'r', 'LineWidth', 3);
```

will draw a red line from `(x1, y1)` to `(x2, y2)` on the provided canvas (which is just a normal image in Matlab). The interface to `draw_transformed_object` is:

```
function draw_transformed_object(canvas, template, T)
```

where `canvas` is the image to draw on, `template` is the template image (so you know which points to transform), and `T` is the transformation matrix.

$\Longrightarrow$ Write the function `draw_transformed_object`.

4 Object tracking on robots

Now comes the trickiest part, and one that we will count as extra credit. Once you have used `sift_gui` to test your code, your next task is to get this working on the

robots. Your goal is to be able to give the robot an image of an object, and have it roam around the room looking in search of that object. To do so, it will patrol the room and use your `detect_object` function. This part is fairly open-ended; it is up to you how you implement the patrolling.

Once the robot has found the object (if it ever does), it should announce that fact by playing a song using `robotPlaySong`.

$\Longrightarrow$ Write the function `robot_patrol_for_object`. This function takes a two parameters, a handle to a robot and an image that the robot should seek out.

5 What to turn in

To recap, you will turn in the following functions for Part 2:

1. `match_nn` (Section 2)
2. `threshold_matches` (Section 2)
3. `match_nn_ratio` (Section 2)
4. `fit_affine_transform` (Section 3)
5. `ransac_affine` (Section 3)
6. `detect_object` (Section 3)
7. `draw_transformed_object` (Section 3)
8. `robot_patrol_for_object` (Section 4) (extra credit)