Object recognition has three basic parts: feature extraction, feature matching, and fitting a transformation. In this lab, you’ll learn about SIFT feature extraction and feature matching—the transformation fitting part will be done for you. To test your functions, we provide two test images, template.png and search.jpg, as well as our own reference implementation. The function stubs and images are in the directory:

/courses/cs1114/section/sift/

1 Feature detection

![Feature detection](image)

*Figure 1: SIFT features detected in an image.*

In this section you will be using SIFT (Scale-Invariant Feature Transform) to detect features in a “template” image, and match it to a “search image.” First, you can try this out in the provided GUI, in the directory above:

```bash
>> sift_gui
```

You should see an interface like this pop up:
In this interface, you load in a template image, which contains the thing you are searching for, and the search image, the image you are searching for the object in. To try this out, click “load template,” and choose `template.png`, then click “load search,” and choose `search.png`. Next, click “look for template in search.” After a moment, a red box should outline the book in the search image. What happened behind the scenes? Our code computed SIFT features for both images, matched them, then found a transformation the maps the template image to the search image, drawing a red box accordingly. Next, try this with a slightly harder example by loading in “template2.png” (the Blackadder DVD cover) as the template image. SIFT is still able to do a good job on this example (SIFT is incredibly powerful).

We provide you with a function in Matlab called `sift` (courtesy of Andreas Veldaldi). 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`:

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

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. 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, as we talked about in class. Figure ?? shows an example of detected SIFT features in an image.

Try out the two lines above to compute SIFT features for the `template.png` image.
Next, you can plot the positions of the detected features using the plotsiftframes function. This will just show the SIFT features, so you want to use it as follows:

```matlab
>> imshow(template);
>> hold on;
>> plotsiftframe(frames);
```

You should now see a bunch of circles showing where SIFT features were detected.

## 2 Feature matching

![Template image](image1.png)

![Search image](image2.png)

**Figure 2: A pair of images we might wish to match.**

Your real work begins with the second step in our object recognition pipeline: feature matching. 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 is shown in Figure 2. 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 usual Euclidean distance:

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

Assuming $a$ and $b$ are column vectors, this formula can be written using matrix operator as:

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

where $^T$ denotes the transpose operator.

Your first task is to write a function `match_nn_student` to find nearest neighbors. This function will take in two sets of descriptors, `descs1` and `descs2`, and will return a $3 \times \text{number of columns in descs1}$ matrix. 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. To test, you can use our provided function `match_nn`.

You’ve seen the `max` function before, but you may find it useful to use some of its more advanced features. For example, `max(A,[],2)` will find the `max` of each row of a 2-D matrix `A`, and `[C, I] = max(...)` will return the indices of the maximum values as well as the values themselves. See the documentation for `max` for details.

Write the function `match_nn_student`.

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`:

```matlab
>> template = imread('template.png');
>> [frames1, descriptors1] = sift(im2double(template));
>> search = imread('search.png');
>> [frames2, descriptors2] = sift(im2double(search));
>> matches = match_nn_student(descriptors1, descriptors2);
>> plotmatches(template, search, frames1, frames2, matches, 'interactive', 1);
```

This interface allows you to click on a feature in one image and see the corresponding feature in the other image. As you can see, this matching procedure will undoubtedly return many false matches; SIFT is not perfect. 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_student`, which takes in a set of matches (the output of the function `match_nn_student` or `match_nn`) and a threshold, and returns a new matrix of matches whose distances are less than the threshold.

→ Write the function `threshold_matches_student`. This function can be written in one line of Matlab code (not including comments).

![Figure 3: Example of images with repetitive features.](image)

Try visualizing the resulting remaining matches using `plotmatches`.

It turns out that even this thresholding doesn't work that well. Consider the example image pair shown in Figure ???. There are a lot of repetitive features in these 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 \(\text{distance}(a, b)\) and \(\text{distance}(a, c)\). If \(\text{distance}(a, b)\) is much smaller than \(\text{distance}(a, c)\), 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.

\[
\frac{\text{distance}(a, b)}{\text{distance}(a, c)}.
\]

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

→ Write the function `match_nn_ratio_student`. 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_student` 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.

Try visualizing the resulting remaining matches using plotmatches.