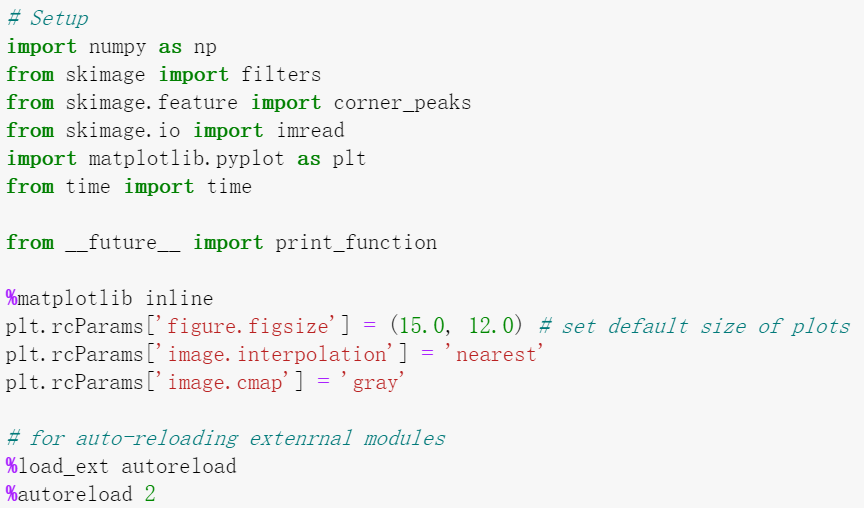
**《Computer Vision Foundations and Applications》Homework 2**

1. **Requirements**

**Attention: Please hand in hw2\_release’s hw2.ipynb file with all the outputs.**

This assignment covers the **Harris corner detector**, **RANSAC** and the **HOG descriptor** for panorama stitching.



## Introduction: Panorama Stitching

Panorama stitching is an early success of computer vision. Matthew Brown and David G. Lowe published a famous [panoramic image stitching paper](http://matthewalunbrown.com/papers/ijcv2007.pdf) in 2007. Since then, automatic panorama stitching technology has been widely adopted in many applications such as Google Street View, panorama photos on smartphones, and stitching software such as Photosynth and AutoStitch.

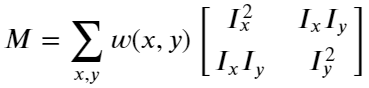
In this assignment, we will detect and match keypoints from multiple images to build a single panoramic image. This will involve several tasks:

1. Use Harris corner detector to find keypoints.
2. Build a descriptor to describe each point in an image.  
   Compare two sets of descriptors coming from two different images and find matching keypoints.
3. Given a list of matching keypoints, use least-squares method to find the affine transformation matrix that maps points in one image to another.
4. Use RANSAC to give a more robust estimate of affine transformation matrix.  
   Given the transformation matrix, use it to transform the second image and overlay it on the first image, forming a panorama.
5. Implement a different descriptor (HOG descriptor) and get another stitching result.

**Part 1: Harris Corner Detector**

In this section, you are going to implement Harris corner detector for keypoint localization. Review the lecture slides on Harris corner detector to understand how it works. The Harris detection algorithm can be divide into the following steps:

1. Compute 𝑥x and 𝑦y derivatives of an image
2. Compute products of derivatives at each pixel
3. Compute matrix  at each pixel, where



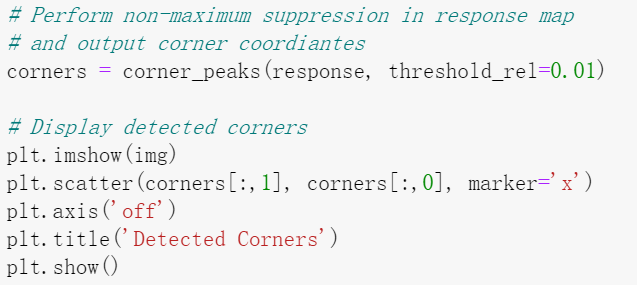
1. Compute corner response  at each pixel
2. Output corner response map

Step 1 is already done for you in the function **harris\_corners** in panorama.py. Complete the function implementation1 and run the code below.

***-Hint:*** *You may use the function scipy.ndimage.filters.convolve, which is already imported in panoramy.py*



Once you implement the Harris detector correctly, you will be able to see small bright blobs around the corners of the sudoku grids and letters in the output corner response image. The function **corner\_peaks** from skimage.feature performs non-maximum suppression to take local maxima of the response map and localize keypoints.



**Part2: Describing and Matching Keypoints**

We are now able to localize keypoints in two images by running the Harris corner detector independently on them. Next question is, how do we determine which pair of keypoints come from corresponding locations in those two images? In order to match the detected keypoints, we must come up with a way to describe the keypoints based on their local appearance. Generally, each region around detected keypoint locations is converted into a fixed-size vectors called descriptors.

**2.1 Creating Descriptors**

In this section, you are going to implement2 the **simple\_descriptor** function, where each keypoint is described by the normalized intensity of a small patch around it.



**2.2 Matching Descriptors**

Next, implement3 the **match\_descriptors** function to find good matches in two sets of descriptors. First, calculate Euclidean distance between all pairs of descriptors from image 1 and image 2. Then use this to determine if there is a good match: if the distance to the closest vector is significantly (by a given factor) smaller than the distance to the second-closest, we call it a match. The output of the function is an array where each row holds the indices of one pair of matching descriptors.



**Part3: Transformation Estimation**

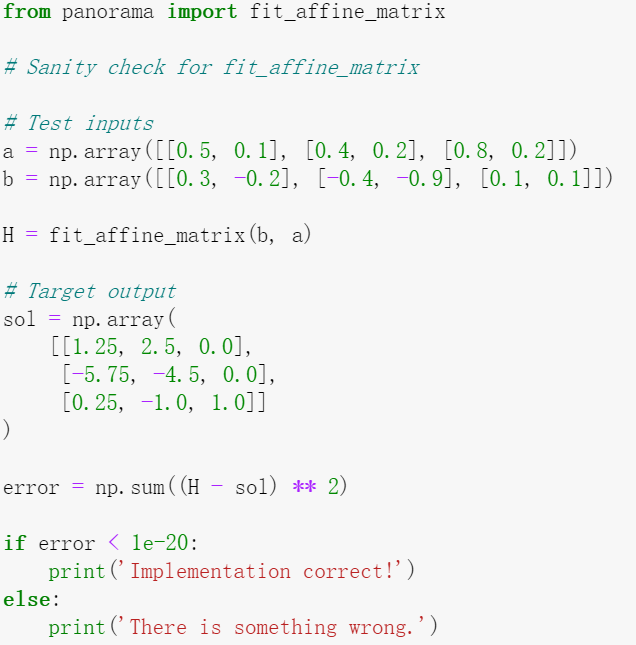
We now have a list of matched keypoints across the two images. We will use this to find a transformation matrix that maps points in the second image to the corresponding coordinates in the first image. In other words, if the point in image 1 matches with in image 2, we need to find an affine transformation matrix such that

where and are homogenous coordinates ofand .

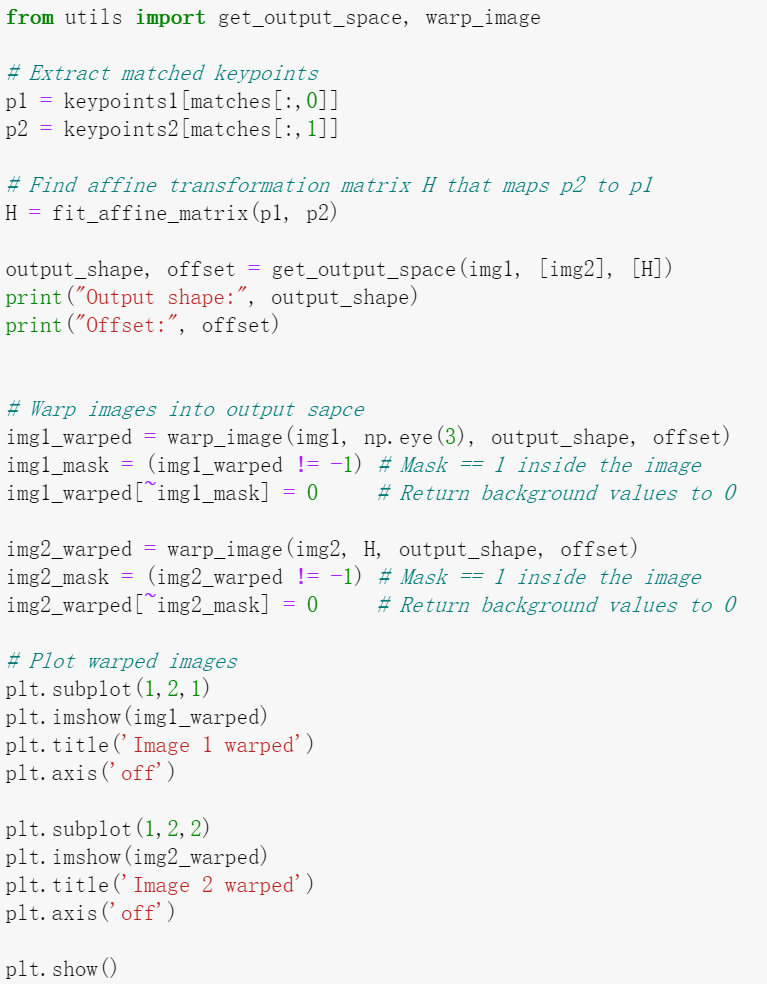
Note that it may be impossible to find the transformation that maps every point in image 2 exactly to the corresponding point in image 1. However, we can estimate the transformation matrix with least squares. Given matched keypoint pairs, let and be matrices whose rows are homogenous coordinates of corresponding keypoints in image 1 and image 2 respectively. Then, we can estimate by solving the least squares problem,

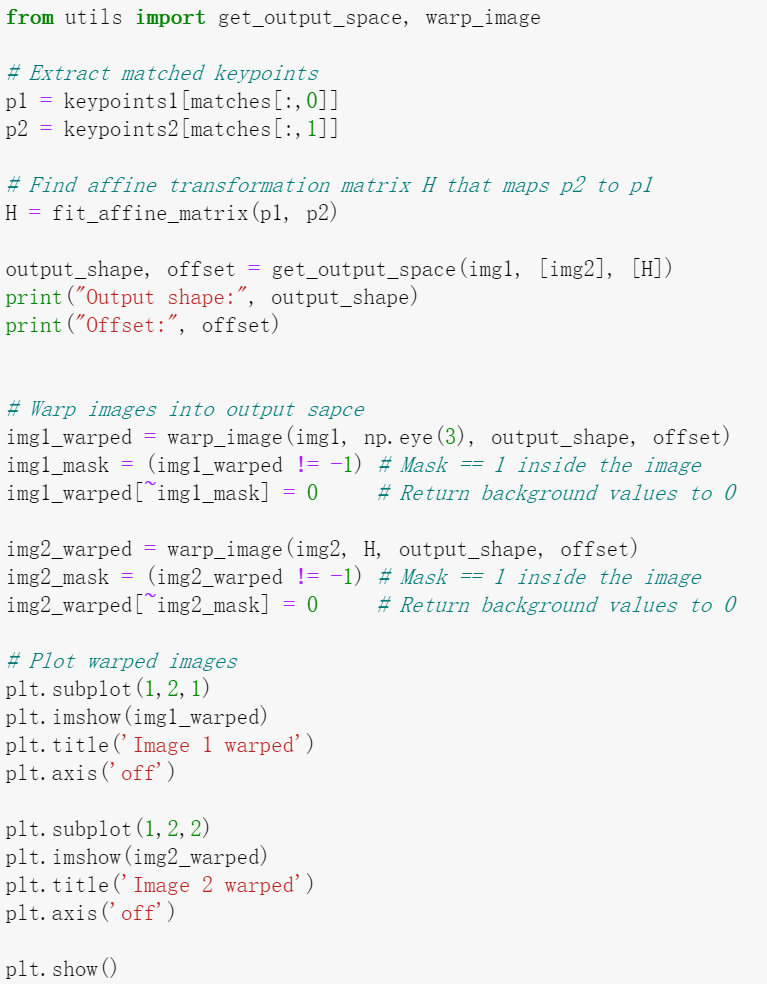
Implement4 **fit\_affine\_matrix** in panorama.py

***-Hint:*** *read the*[*documentation*](https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html)*about np.linalg.lstsq*

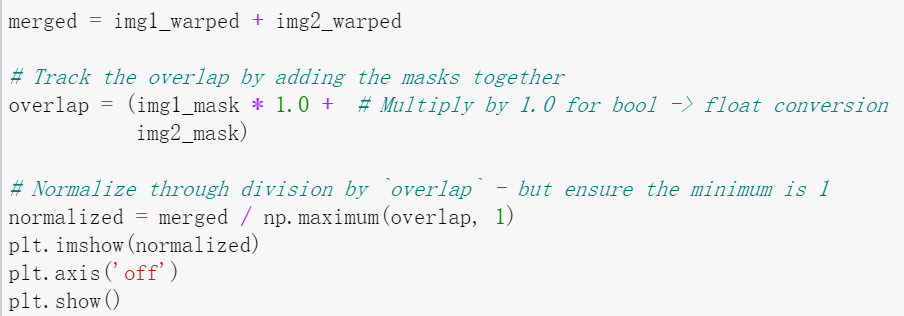


After checking that your **fit\_affine\_matrix** function is running correctly, run the following code to apply it to images. Images will be warped and image 2 will be mapped to image 1.





Next, the two warped images are merged to get a panorama. Your panorama may not look good at this point, but we will later use other techniques to get a better result.

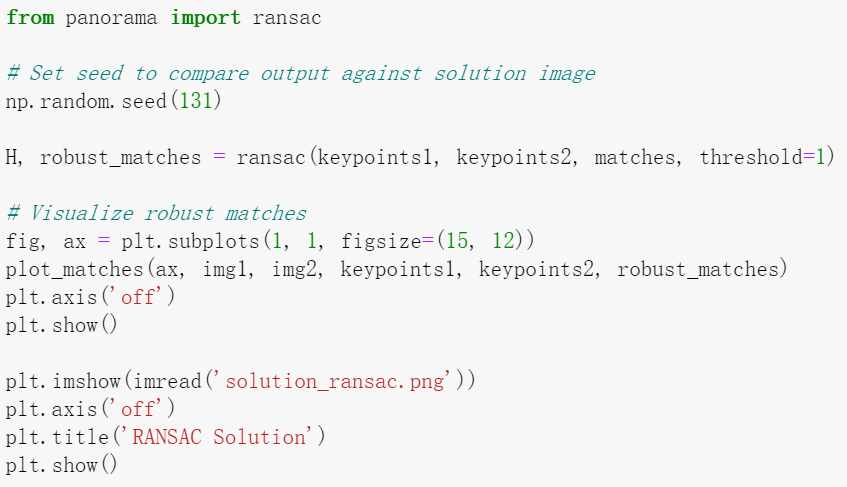


**Part4: RANSAC**

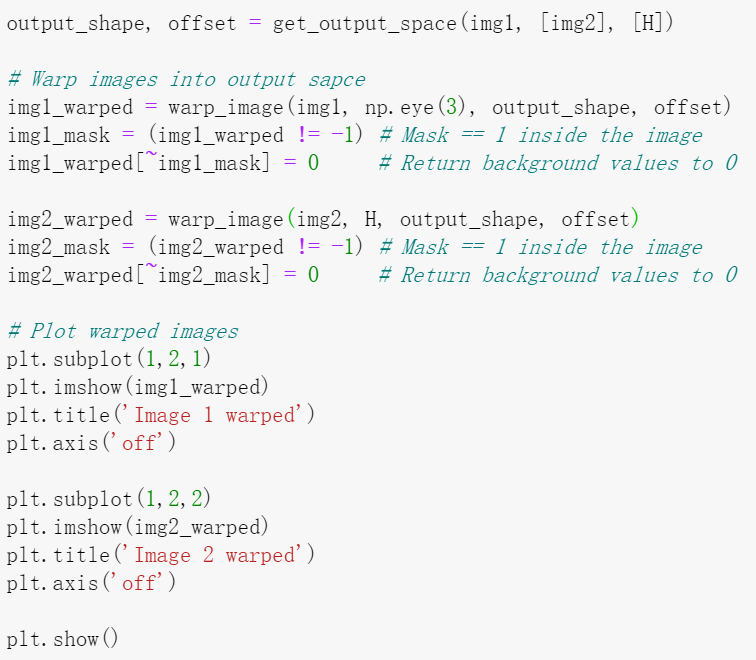
Rather than directly feeding all our keypoint matches into **fit\_affine\_matrix** function, we can instead use RANSAC ("RANdom SAmple Consensus") to select only "inliers" to use for computing the transformation matrix.

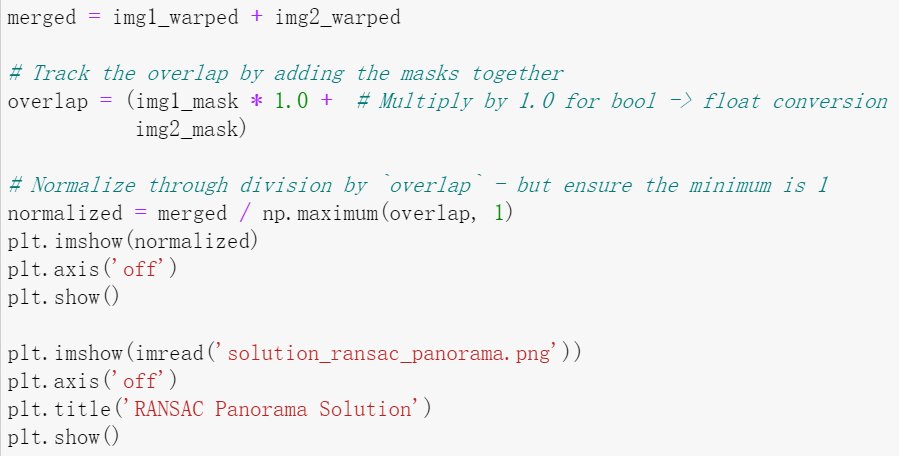
The steps of RANSAC are: 1. Select random set of matches 2. Compute affine transformation matrix 3. Find inliers using the given threshold 4. Repeat and keep the largest set of inliers 5. Re-compute least-squares estimate on all of the inliers

Implement5 **ransac** in panorama.py, run through the following code to get a panorama. You can see the difference from the result we get without RANSAC.



We can now use the tranformation matrix computed using the robust matches to warp our images and create a better-looking panorama.





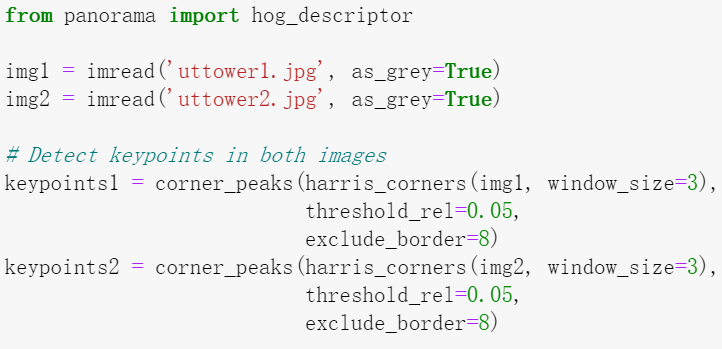
**Part5: Histogram of Oriented Gradients (HOG)**

In the above code, you are using the **simple\_descriptor**, and in this section, you are going to implement a simplified version of HOG descriptor.

HOG stands for Histogram of Oriented Gradients. In HOG descriptor, the distribution ( histograms ) of the directions of gradients ( oriented gradients ) are used as features. Gradients ( and derivatives ) of an image are useful because the magnitude of a gradient is large around edges and corners ( regions of abrupt intensity changes ) and we know that edges and corners pack in a lot more information about object shape than flat regions.

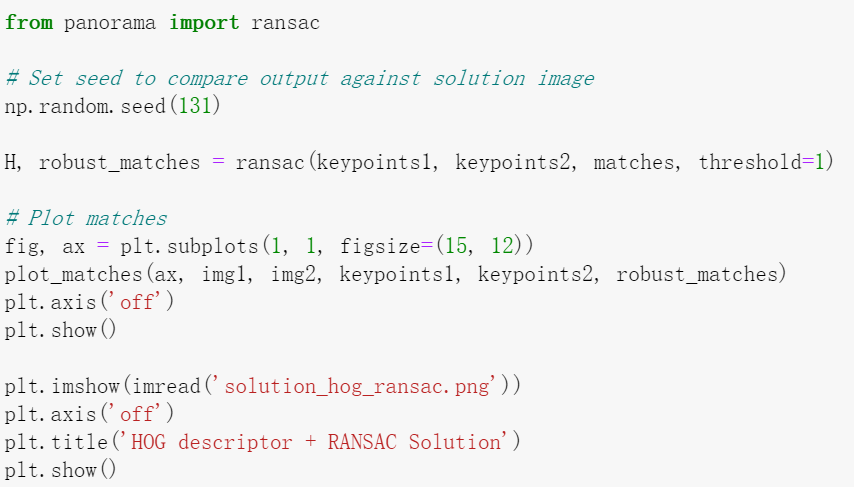
The steps of HOG are: 1. Compute the gradient image in x and y directions Use the sobel filter provided by skimage.filters 2. Compute gradient histograms Divide image into cells, and calculate histogram of gradients in each cell 3. Flatten block of histograms into feature vector 4. Normalize flattened block

Implement6 **hog\_descriptor** in panorama.py and run through the following code to get a panorama image.

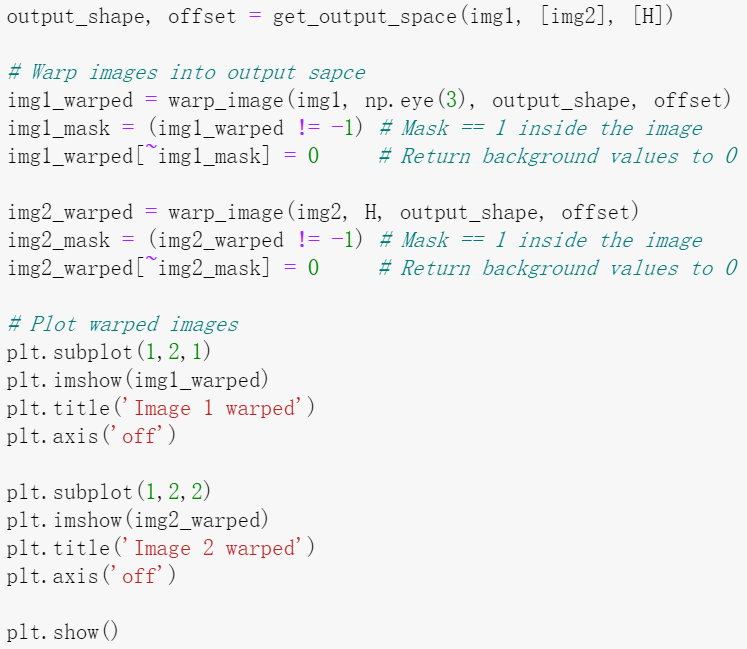


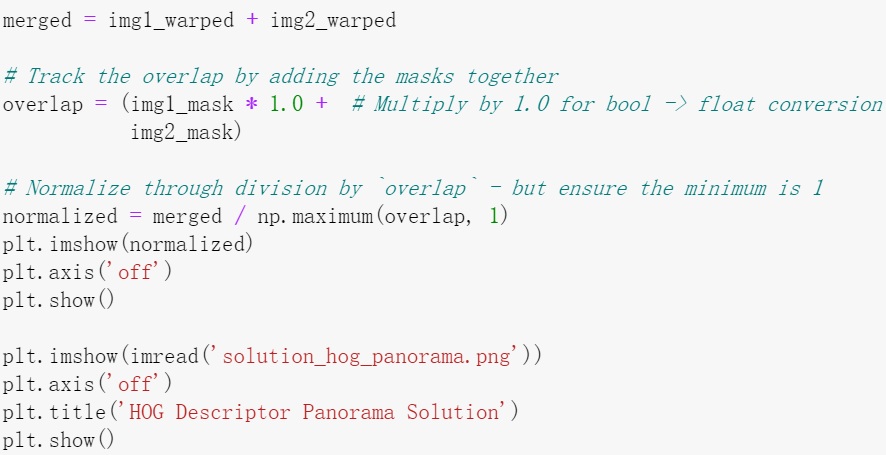


Once we've described our keypoints with the HOG descriptor and have found matches between these keypoints, we can use RANSAC to select robust matches for computing the tranasformtion matrix.



Now we use the computed transformation matrix to warp our images and produce our panorama.





**Extra Credit: Better Image Merging(5 Points)**

You will notice the blurry region and unpleasant lines in the middle of the final panoramic image. Using a very simple technique called linear blending, we can smooth out a lot of these artifacts from the panorama.

Currently, all the pixels in the overlapping region are weighted equally. However, since the pixels at the left and right ends of the overlap are very well complemented by the pixels in the other image, they can be made to contribute less to the final panorama.

Linear blending can be done with the following steps: 1. Define left and right margins for blending to occur between 2. Define a weight matrix for image 1 such that: - From the left of the output space to the left margin the weight is 1 - From the left margin to the right margin, the weight linearly decrements from 1 to 0 3. Define a weight matrix for image 2 such that: - From the right of the output space to the right margin the weight is 1 - From the left margin to the right margin, the weight linearly increments from 0 to 1 4. Apply the weight matrices to their corresponding images 5. Combine the images

In **linear\_blend** in panorama.py implemente1 the linear blending scheme to make the panorama look more natural. This extra credit can be worth up to 1% of your final grade.



**Extra Credit: Stitching Multiple Images(5 Points)**

Implemente2**stitch\_multiple\_images** in panorama.py to stitch together an ordered chain of images. This extra credit can be worth up to 1% of your final grade.

Given a sequence of 𝑚m images take every neighboring pair of images and compute the transformation matrix which converts points from the coordinate frame of  to the frame of . Then, select a reference image , which is in the middle of the chain. We want our final panorama image to be in the coordinate frame of .

*-Hint:*

* If you are confused, you may want to review the Linear Algebra slides on how to combine the effects of multiple transformation matrices.
* The inverse of transformation matrix has the reverse effect. Please use [numpy.linalg.inv](https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.inv.html" \t "_blank) function whenever you want to compute matrix inverse.

***Attention:* *The total score will not exceed 100 with Extra Credit.***