MIT EECS 6.815/6.865: Assignment 7:

Harris Corners, Features and Automatic Panoramas

Due Monday April 2 at 9pm

1 Summary

- Harris corner detection
- patch descriptor
- correspondences using nearest neighbors and the second nearest neighbor test
- RANSAC
- 6.865 only: probabilistic termination for RANSAC
- 6.815: fully automatic panorama stitching of two images
- 6.865: fully automatic panorama stitching of N images
- Capture and stitch your own panorama

This is not an easy assignment. There are many steps that all depend on the previous ones and it's not always trivial to debug intermediate results. We provided you with visualization helpers in `a7help.py`.

The use of list operators such as `zip`, `map`, and `filter` as well as the `lambda` operator can greatly simplify this assignment.

2 Harris Corner Detection

The Harris corner detector is founded on solid mathematical principles, but its implementation looks like following a long cookbook recipe. Make sure you get a good sense of where you're going and debug/check intermediate values. If needed, we provide a helper function `imageFrom1Channel` that turns 2D arrays into black-and-white images, and `writeGrey` that takes 1-channel images and writes them to disk with unique names.

2.1 Structure tensor

The Harris Corner detector is based on the structure tensor, which characterizes local image variations. We will focus on greyscale corners and forget color

variations. We start from the gradient of the luminance L_x and L_y along the x and y directions (where subscripts denote derivatives). The structure tensor is

$$M = \sum w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

where w is a weighting function, a Gaussian in our case. For this, we will first compute $\begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$ at each pixel and will convolve it with a Gaussian.

Write a function `computeTensor(im, sigmaG=0.5, factorSigma=4)` that returns a 3D array of the size of the image where each location `y, x` stores 3 values corresponding to the xx, yx, and yy components of the tensor.

For this, you should first extract the luminance of the image. I used my favorite weights $[0.3, 0.6, 0.1]$ for the RGB channels, and numpy's `dot`.

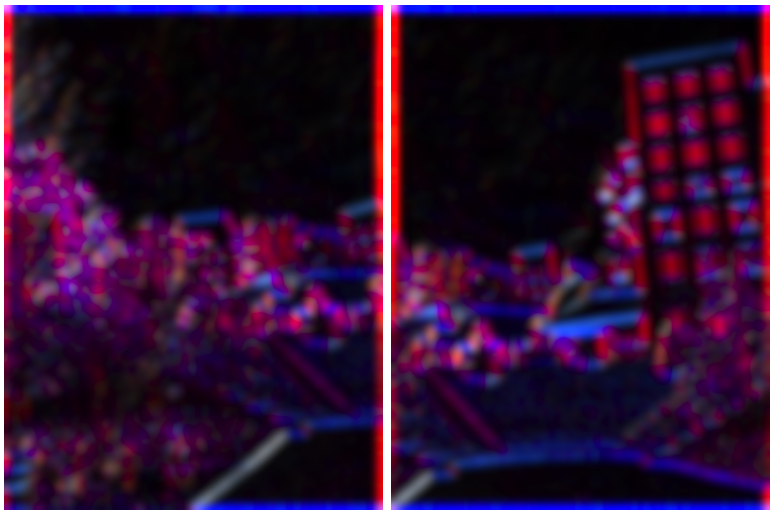
We then blur luminance to control the scale at which corners are extracted. A little bit of blur helps smooth things out and make sure we extract stable mid-scale corners. For more advanced feature extraction, different scales can be used to make the whole process invariant to image scaling. Use a Gaussian of standard deviation `sigmaG`. We recommend you use scipy's `ndimage.filters.gaussian_filter`.

Next, compute the image gradient along the y and x direction. We'll reuse our Sobel kernel, but for faster computation will rely on scipy and run:
`signal.convolve(L, Sobel, mode='same')`.
The last argument forces the output to be the same size as the input. You then should remember how to get the orthogonal component of the gradient. Note that each of the input luminance `L`, and the output gradients along x and y are 2D arrays because they only have one value per pixel.

Now is time to compute the contribution of each pixel to the structure tensor, using the simple formula above. Note that the matrix is symmetric and we need to store only three values per pixels. You can store them in a 3D array or in 3 2D arrays.

Then, compute the local weighted sum by convolving the above per-pixel contributions using a Gaussian blur of size `sigmaG*factorSigma`

For visualization purposes, you can combine the three components and write the result as an image. Here are my results for the Stata pair where red and blue are xx and yy and green is xy.



2.2 Harris corners

Implement a function `HarrisCorners(im, k=0.15, sigmaG=0.5, factor=4, maxiDiam=7, boundarySize=5)` that returns a list of 2D points (encoded as 1D 2-valued arrays) corresponding to Harris corners. Implement the following step after you have computed the per-pixel structure tensor for `im`. Read the whole section before implementing.

Corner response The measure of corner response is $R = det(M) - k(trace(M))^2$, which compares whether the matrix has two strong eigenvalues, indicative of strong variation in all directions. Only pixels with positive corner responses can be corners.

My corner response looks like

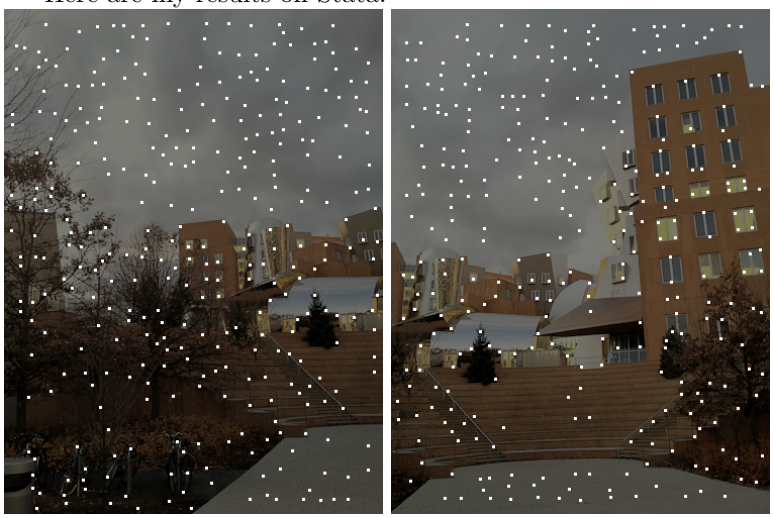
Non-maximum suppression We get a strong corner response in a neighborhood around each corner, and we need to only keep the strongest response. For this, you need to reject all pixels that are not a local maximum in a window of `maxiDiam`.

I recommend you hire the help of scipy's `ndimage.filters.maximum_filter`, which will give you, for each pixel, the maximum value of an image in a window.

Removing boundary corners Because we will eventually need to extract a local patch around each corner, we can't use corners that are too close to the boundary of the image. Exclude all corners that are less than `boundarySize` pixels away from any of the four image edges.

Putting it all together In the end, your function should return a list of arrays containing the coordinates of each corner. I personally scanned to find all pixels that satisfied my criteria, but smarter people might be able to use some numpy magic. Use the provided function `visualizeCorners` to verify your results.

Here are my results on Stata:



More bells and whistles such as adaptive maximum suppression or different luminance encoding might help, but this will be good enough for us.

3 Descriptor and correspondences

Descriptors describe the local neighborhood of an interest point such as a Harris corner in order to match the same point in a different images. Points in two images are put in correspondence when their descriptors are similar enough. We will call the combination of an interest point coordinates and a descriptor a feature.

3.1 Descriptors

In this section, you will write a function
`computeFeatures(im, cornerL, sigmaBlurDescriptor=0.5, radiusDescriptor=4)`
that uses the above Harris corner detector to compute interest points and then associate each of them with a descriptor to output a list of features, where each feature is a pair (corner, descriptor).

I found it convenient to write a subroutine `descriptor(blurredIm, P, radiusDescriptor)` that extracts a single descriptor around interest point `P`.

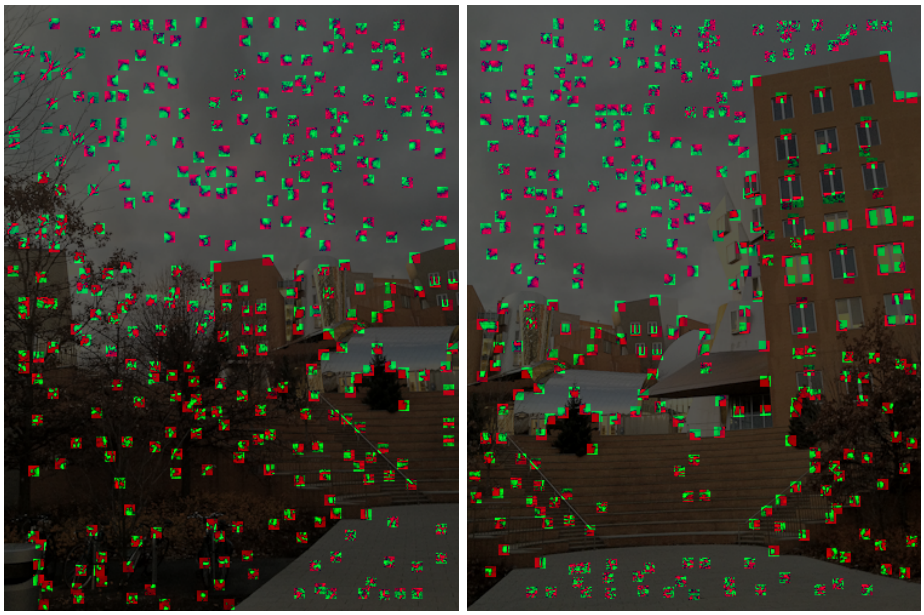
Our descriptors will be a version of the pixel luminance in a `radiusDescriptor*2+1` by `radiusDescriptor*2+1` window around their interest point. That is, they will be numpy arrays of size 9×9. To avoid aliasing issues, blur the image by a Gaussian blur of standard deviation `sigmaBlurDescriptor` before extracting the pixel values.

Then, for each corner, extract the patch around it. I highly recommend the use of smart indexing with numpy and the "`:`" operator.

Finally, we want to address potential brightness and contrast variation between images. For this, we subtract the mean of each patch and scale the resulting patch by the 1 over its standard deviation. You can use numpy's `mean` and `std` functions. Note that, as a result of the offset and scale, our features will have negative numbers and might be greater than 1.

To recap, your function `computeFeatures` should return a list of features, where each feature is a pair of an interest point location (1D size-2 array) and a 9x9 descriptor encoded as a 2D array.

We provided you with a function `visualizeFeatures(LF, radiusDescriptor, im)` that overlays the descriptors at the location of their interest points, with positive values in green and negative ones in red. The normalization by the standard deviation makes low-contrast patches harder to recognize, but high-contrast ones should be easy to debug, e.g. around the tree or other strong corners.



3.2 Best match and 2nd best match test

Now that we have code that can compute a list of features for each image, we want to find correspondence from features in one image to those in a second one. We will use our descriptors and the L2 norm to compare pairs of features. The procedure is not symmetric (we match from the first to the second image) but it doesn't matter.

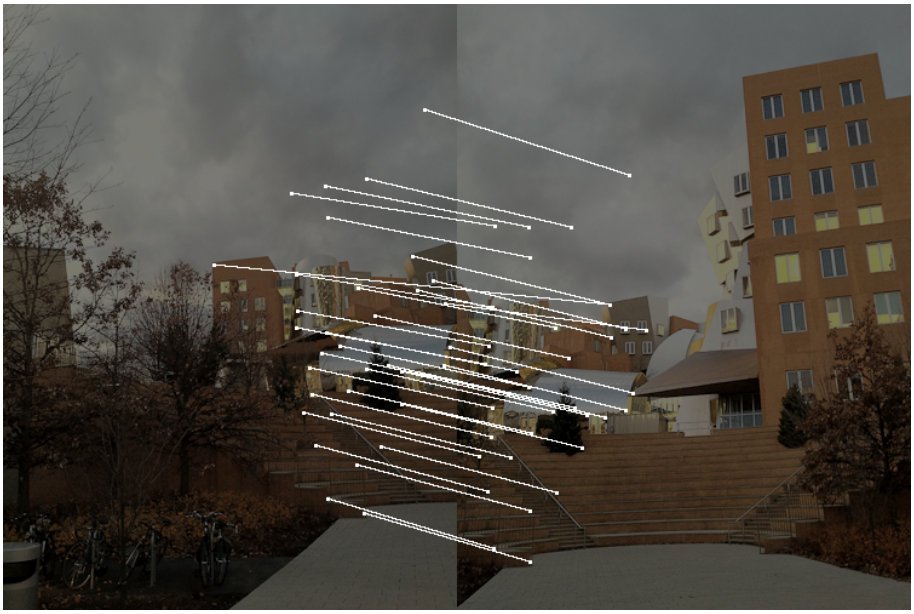
Write a function `findCorrespondences(listFeatures1, listFeatures2, threshold=1.7)` that computes, for each feature in `listFeatures1`, the best match in `listFeatures2`, but rejects matches when they fail the second-best comparison studied in class. As usual, writing a helper function that handles a single feature in `listFeatures1` might help. Also, I have found `map` and `filter` very useful.

The squared distance between two descriptors is the sum of squared differences between individual values. It can easily be computed using numpy's `dot`. The search for the minimum (squared) distance can be brute force, and python even has functions `argmin` and `min`.

The second-best test considers not only the most similar descriptor, but also the second best. If the ratio of distances to the best and to the second best is less than `threshold`, we reject the match because it is too ambiguous: the second best match is almost as good as the best one.

Your function `findCorrespondences` should return a list of pairs of 2D points corresponding to the matching interest points that passed the test. The size of this list should be at most that of `listFeatures1`, but is typically much smaller.

Use the provided `visualizePairs` to debug your matches. Note that not all correspondences are going to be perfect. We will reject outliers in the next section using RANSAC. But a decent fraction should be coherent, as shown below.



4 RANSAC

So far, we've dealt with the tedious engineering of feature matching. Now comes the apotheosis of automatic panorama stitching, the elegant yet brute force RANSAC algorithm (RANdom Sample Consensus). It is a powerful algorithm to fit low-order models in the presence of outliers. Read the whole section and check the slides to make sure you understand the algorithm before starting your implementation. If you have digested its essence, RANSAC is a trivial algorithm to implement. But start on the wrong foot and it might be a path of pain and misery.

In our case, we want to fit a homography that maps the list of feature points from one image to the corresponding ones in a second image, where correspondences are provided by the above `findCorrespondences` function. Unfortunately, a number of these correspondences might be utterly wrong, and we need to be robust to such so-called *outliers.* For this, RANSAC uses a probabilistic strategy and tries many possibilities based on a small number of correspondences, hoping that none of them is an outlier. By trying enough, we can increase the probability of getting an attempt that is free of outliers. Success is estimated by counting how many pairs of corresponding points are explained by an attempt.

You will write a function `RANSAC(listOfCorrespondences, Niter=1000, epsilon=4)` that takes a list of correspondences (list of pairs of 2D points represented by 2-valued 1D arrays) and returns a homography that best transforms the first member of each pair into the second one. In addition, for visualization purposes, the function should return a list of Booleans of the same length as `listOfCorrespondences` that indicates whether each correspondence pair is an inlier, i.e., is well modeled by the homography. `Niter` is the maximum number of RANSAC iterations (random attempts) and `epsilon` is the precision, in pixel, for the definition of an outlier. vs. inlier. That is, the pair p, p' is said to be an inlier with respect to a homography H is $||p' - Hp|| <$`epsilon`.

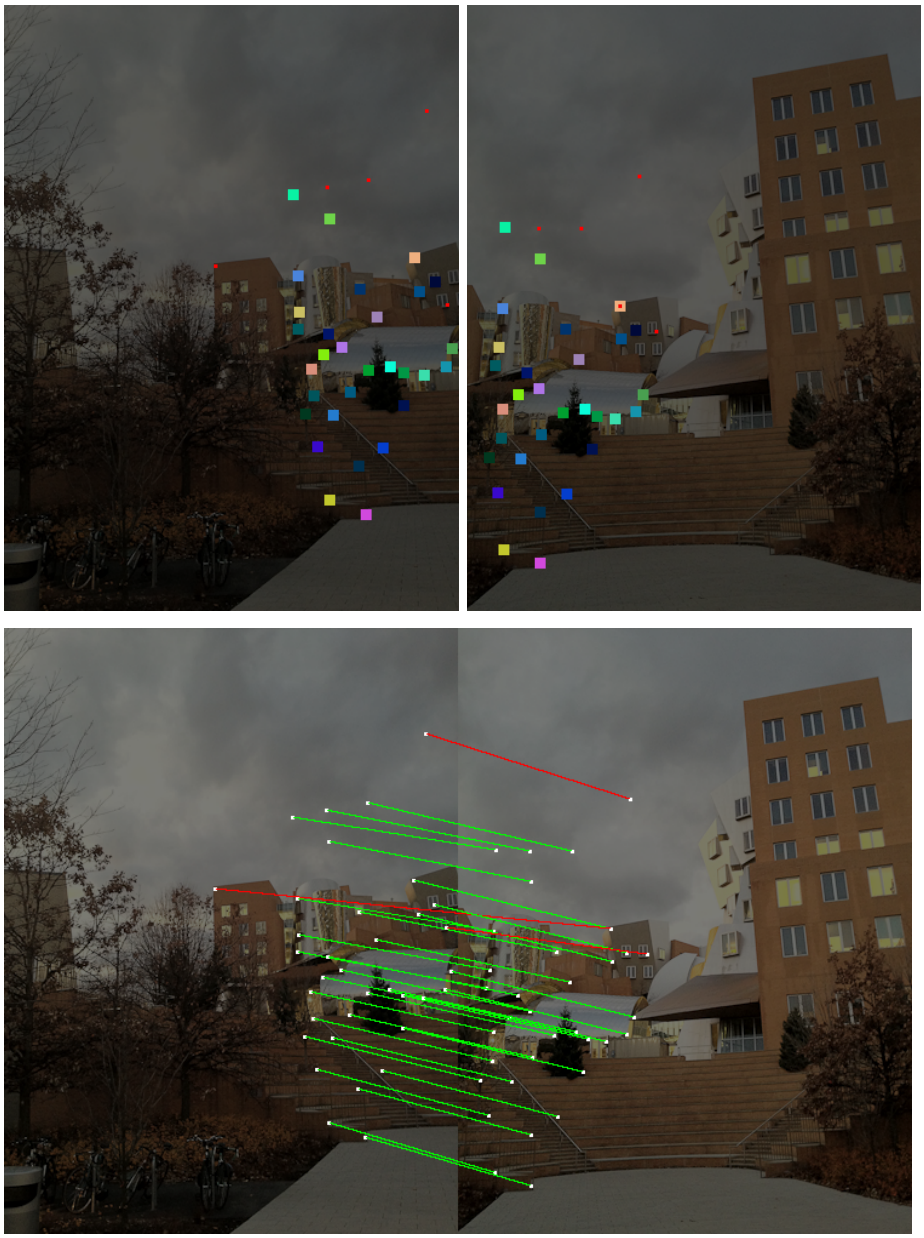
For each RANSAC iteration, pick four random pairs in listOfCorrespondences. Pyhton's module `random` has the good taste of having a function `random.sample(List, N)` that does just this. Just be careful that numpy's random functions don't interfere with it if you did `from numpy import *`, for example by importing the random module under a different name using e.g. `import random as rnd`

Given four pairs of points, you should have a function from problem set 6 that computes a homography. In some cases, the four pairs might result in a singular system for the homography. My original solution was to test the determinant of the system and return the identity matrix when things go wrong. It's not the cleanest solution in general, but RANSAC will have no problem dealing with it and rejecting this homography, so why not? A better solution is to replace `linalg.inv` by `linalg.pinv`

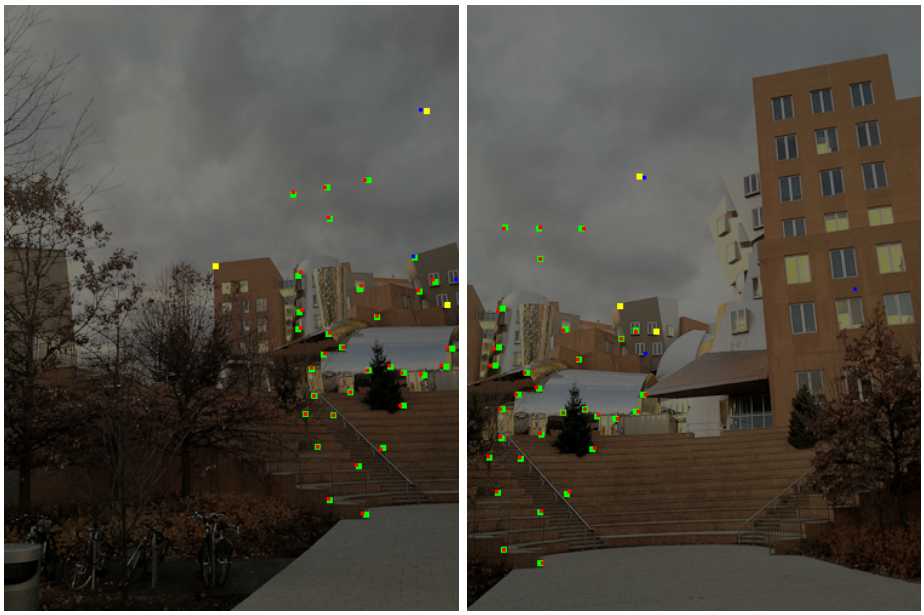
We now need to evaluate how good a solution this homography might be. For this, just compute the number of inliers, as defined above by the test $||p' - Hp|| <$`epsilon`. This is another case where list operators such as `map` and `filter` and `len` can help. Or write explicit loops if you don't feel comfortable with functional programming.

If the number of inliers of the current homography is greater than the best one so far, keep the homography and update the best number of inliers.

At the end of the process, make sure you somehow return both the homography and the list of inlier Booleans. You can use the provided function `visualizePairsWithInliers(im1, im2, pairs, isInlier)` to see which correspondences are considered inliers. It outputs three images. In the first pair, feature points are displayed on top of the two images, with color codes to show corresponding inliers, while outliers are displayed as small red points. The second image is similar the output of `visualizePairs` except that inliers are in green and outliers are red. But, your mileage will vary because RANSAC is a probabilistic algorithm. Don't freak out if you don't get exactly the same inliers as me. Plus you know how likely I am to have bugs.



You can also use the provided `visualizeReprojection`, wich shows where the homography reprojects features points. For inlier, detected corners are in green, while those reprojected from the other image are in red. For outliers, the local corners are yellow and the reprojected ones are blue. My reproductions for Stata are below. The result below further emphasizes that RANSAC is probabilistic: my set of inliers is not exactly the same as above.



4.1 6.865 only: probabilistic termination

Compute the probability of having had only sets of 4 pairs that always contain at least one outlier and use for early termination in RANSAC when this probability falls below a threshold. Add one parameter `acceptableProbFailure=1e-9` to RANSAC for this threshold. The equation requires the probability that a pair be an outlier. You can estimate this using your best number of inliers so far.

Print the probability at each iteration.

5 Putting it all together

5.1 6.815: Automatic panorama for a pair of images

Write a function
`autostitch(im1, im2, blurDescriptor=0.5, radiusDescriptor=4)`
that takes two images as input and automatically outputs a panorama image where the second image is warped into the domain of the first one. You should get the same result as the last assignment, but automatically.

Try it on the Stata sequence and on at least another pair of images provided with pset 6. Turn in your result.

5.2 6.865: Automatic panorama for N images

Write a function
`autostitch(L, refIndex, blurDescriptor=0.5, radiusDescriptor=4)`

that takes as input a list of images `L` and stitches them automatically to create a panorama in the coordinate system of image `refIndex`. You need to compute pairwise homographies and chain them backward and forward to deduce global homographies. Don't forget the bounding box business.

Run your function on one of a sequence of at least three images from pset 6. Turn in your result.

5.3 Both: Make your own panorama

Capture your own sequence and run it through your automatic algorithm. Two images for 6.815, at least three for 6.865.

Make sure you keep the camera horizontal enough because our naive descriptors are not invariant to rotation. Similarly, don't use a lens that is too wide angle (Don't push below a 35mm equivalent of 24mm). Your total panorama shouldn't be too wide angle (don't go too close to 180 degrees yet) because the distortions on the periphery would lead to a very distorted and ginormous output. Some of the provided sequences are already pushing it. Finally, recall that you should rotate around the center of projection as much as possible in order to avoid parallax errors. This is especially important when your scene has nearby objects.

Turn in both your source images and your results.

If you need to convert images, one online tool that appears to work is `http://www.coolutils.com/online/image-converter/`

6 Extra credits

Adaptive non-maximum suppression.

Wavelet descriptor.

Rotation or Scale invariance

Full SIFT.

Evaluation of repeatability.

Least square refinement of homography at the end of RANSAC

Reweighted least square.

Bundle adjustment

7 Submission

Turn in your images, python files, and make sure all your functions can be called from a module `a7.py`. Put everything into a zip file and upload to Stellar.

Include a `README.txt` containing the answer to the following questions:

- How long did the assignment take?
- Potential issues with your solution and explanation of partial completion (for partial credit)

- Any extra credit you may have implemented
- Collaboration acknowledgement (but again, you must write your own code)
- What was most unclear/difficult?
- What was most exciting?

Images:

- result on the provided images
- source and result for your own pano