MIT EECS 6.815/6.865: Assignment 7:
Harris Corners, Features and Automatic Panoramas

Due Wed Oct 30 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
- Linear blending
- Two scale blending

This is not an easy assignment, so start this early. There are many steps that all depend on the previous ones and it's not always trivial to debug intermediate results. In the test script, we provide you different visualization tools. Look at them, and figure how to use them to debug. You are encouraged to write your own debugging tools. In `a7_readme.py` , you will need to explain the way you debug and the input image used for debugging. Debugging will be essential in this assignment.

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.

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=1, factorSigma=4)` that returns a 3D array of the size of the image where each location `y, x` stores 3 values corresponding to the I_x^2, $I_x I_y$, and I_y^2 components of the tensor, in the order from the first channel to the third channel.

For this, you should first extract the luminance of the image. Use the `BW` function in the assignment one for the conversion. We read the image directly from image value. This is equivalent to compressing the luminance by raising it to the power 1/2.2, in order to compress contrast in the bright areas and get a more uniform set of corners across different luminance levels.

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 in `A3`, but for faster computation will rely on scipy and run: `signal.convolve(L, Sobel, mode='reflect')`.
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 need to store them in a 3D array in the order as described above.

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-1 where red and blue are I_x^2 and I_y^2 and green is $I_x I_y$. Use the visualization tool in the test script to see if your result makes sense.

2.2 Harris corners

Implement a function
`HarrisCorners(im, k=0.15, sigmaG=1, factor=4, maxiDiam=9, boundarySize=5)`
that returns a list of 2D points. For this assignment, we provide a class `point` to avoid the confusion of x and y. Each point corresponds to a Harris corner. Implement the following step after you have computed the per-pixel structure tensor for `im`. Read the whole section before implementing.

Corner response Implement a function
`cornerResponse(im, k=0.15, sigmaG=1, factorSigma=4)` that returns corner response $R = det(M) - k(trace(M))^2$, 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.

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[2] `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.

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 takes as input a list `cornerL` of the above Harris corners to compute interest points and then associate each of them with a descriptor to output a list of `feature` objects with the provided class. Each feature contains a corner and a descriptor (one-dim array).

You have to write a function `descriptor(blurredIm, P, radiusDescriptor)` that extracts a single descriptor around interest point `P`.

Our descriptors will be a version of the pixel luminance (single channel) 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 has an interest point location (`point` object) and a 9x9 descriptor encoded as a 1D array.

We provided a function 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 second best and to the 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. Careful between the squared distance and the distance itself. You can compute everything with just the squared distance (it's faster) but then you need to use the square of the threshold.

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.

We provided some visualization code in the test script. 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 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 $||p' - Hp||_2 <$`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. We provide the interface for `point` class and the point representation in A6. 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 $||p' - Hp||_2 <$`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 `H`, represented by a 2-D array, and the list of inlier Booleans.

Our test script will mark the inlier pairs green, while outlier pairs red.

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. For 6.815 You can ignore this parameter.

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. For this

you will write `computeNHomographies(L, refIndex, blurDescriptior=0.5, radiusDescriptor=4)` that returns a list of homographs relative to L[refIndex]. You should get the same result as the last assignment, but automatically.

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. Pay attention: things are different before and after the reference image. 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 Blending

So far, we have reprojected input images into a common domain without paying much attention to problems at the boundaries. Our goal is now to mask the transition between images. For all the problems below, 6.815 only needs to handle two image case, but 6.865 needs to handle N images.

6.1 Linear blending

We will first implement a smooth transition `linear_blending(L, refIndex, blurDescriptor=0.5, radiusDescriptor=4)` function between the images. For this, the output image will be a weighted average of the reprojected input ones, where the weights go from 1 at the center of an image to 0 at the edges. Write a function `weight_map(h,w)` to compute the weight for each pixel.

Weights are not easy to compute in the output image because of the reprojection, which makes it hard to tell how far a pixel is from an image's boundary. Instead, we will compute weights in the source domain where it is trivial to tell where a pixel is compared to the image boundary.

We will use simple base weights in the source domain. Weights will be separable in x and y, meaning that they are the product of two functions, one in x only and one in y only. We will use simple piecewise-linear function going from 1.0 at the center down to 0.0 at the edges of the images.

Make sure you figure out some way of keeping track of the sum of the weights at each pixel.

Modify your automatic stitching code to perform smooth blending.

Turn in your own panorama from the previous problem.

6.2 Two-scale blending

The problem with smooth transitions like the ones above is that the resulting image can be blurry at the transition or exhibit ghosts when features are not exactly matched.

In order to fix this, we will use a two-scale approach that uses a smooth transition for the low frequencies and an abrupt transition for high frequencies. Write a function `two_scale_blending(L, refIndex, blurDescriptor=0.5, radiusDescriptor=4)`

First, decompose each source image into low frequencies and high frequencies using a Gaussian blur. We recommend a spatial sigma of 2 pixels.

For the low frequencies, use the same transition as above.

For the high frequencies, use an abrupt transition that only keeps the high frequency of the image with the highest weight.

Compute the final image by adding the resulting low and high frequencies.

Turn in the output for your own panorama from the previous problem.

7 6.865 Only: Review

This problem is for 6.865 only, and unlike the previous problems, it's going to be all reading and writing. Your job is to review a recent publication in computational photography. This will hopefully give you some more insight into the review process that these papers go through, and perhaps help you come up with ideas about your final assignment (and in case you were concerned, we don't expect your final pset to be publication-quality, but it's still helpful to keep the review criteria in mind).

You should select one paper from the provided papers.html file. If there's another relevant paper that you're particularly interested in reading, ask us about it and we'll probably be okay with it (unless it's something that you're already supposed to read for one of the problem sets).

You should use the SIGGRAPH review form, available at the following URL: http://s2012.siggraph.org/submitters/technical-papers/review-form. You may omit the numerical ratings from questions 5 and 6. For question 8, try to come up with at least one outstanding question that you think the authors didn't address in the paper. You can skip question 9. You should put your review in an ascii text file. Try to be thorough! Probably a paragraph or two for each question, and significantly more for question 7.

One final note: don't be compelled to like a paper because it was already published. Every paper has some shortcomings. Maybe the method is very elegant and general but doesn't end up producing very impressive results. Maybe the results are spectacular, but the method itself has some mathematical holes. Maybe everything seems great, but there just aren't enough details to reproduce the results. It's up to you as a reviewer to weigh the strengths and weaknesses of each paper

Submit your review along with the file: review.txt.

8 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

Write a function `extra_credits` that returns all the function names you implemented for extra credit as a list.

Cylindrical reprojection We can reproject our panorama onto a virtual cylinder. This is particularly useful when the field of view becomes larger. This is not a difficult task per say, but it requires you to keep track of a number of coordinate systems and to perform the appropriate conversions. For this, it is best to think of the problem in terms of 3D projection onto planes vs. cylinders.

At the end of the day, we will start with cylindrical coordinates, turn them into 3D points/rays, and reproject them onto planar coordinate systems to lookup pixel values in the original images.

The projection matrix for a planar image when the optical axis is along the z coordinates is

$$K = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

where f is the normalized focal length, corresponding to a sensor of width 1.0.

This projects 3D points into 2D homogenous coordinates, which need to be divided by the 3rd component to yield Euclidean coordinates. The coordinates in the sensor plane are assumed to go from -0.5 to 0.5 for the longer dimension.

We then need to convert these normalized coordinates into [0..height, 0..width]. Define `size=max(height, width)`, then the normalized coordinates

$$S = \begin{pmatrix} size & 0 & height/2 \\ 0 & size & width/2 \\ 0 & 0 & 1 \end{pmatrix}$$

In the end, for the reference image, we have

$$P_{2D} = SKP_{3D}$$

We also know that for another image

$$P^i_{2D} = H^{ref \to i} P^{ref}_{2D}$$

Now that we have equations for planar projections, we compute the cylindrical projection of one image. We interpret the output pixel coordinates as cylindrical coordinates y, θ (after potential scaling and translation). y is the vertical dimension of the cylinder and θ the angle in radian. We convert these into a 3D point P_{3D}, which we reproject into the source image where we perform a bilinear reconstruction.

I encourage to debug this using manually-set bounding boxes (e.g. $-\pi/2..\pi/2$ in θ, and a scaling factor that maps preserves the height of the reference image).

You can then, if you want adapt your bounding box computation. Note that cylindrical projections are not convex, and taking the projection of the 4 corners does not bound the projection. You can ignore this and accept some cropping or sample the image boundary more finely.

Horizon correction for cylindrical reprojection The y axis of the reference image is not necessarily the vertical axis of the world. This might result in some distorted reproduction where the horizon is not horizontal.

You can address this by fitting a plane onto the centers of the panorama source images in the 3D coordinate system of the reference image.

Mini planet Use the stereographic projection to yield the popular mini planet view. See e.g. http://en.wikipedia.org/wiki/Stereographic_projection
http://www.miniplanets.co.uk/

9 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 the submission site. Also, this time, you write `a7_readme.py` instead of README.txt. Follow the instruction in the .py file. Upload everything except the data we provided.