MIT EECS 6.815/6.865: Assignment 7:

Harris Corners, Features and Automatic Panoramas

Due Wednesday Novemeber 11 at 9pm

1 Summary

- Harris corner detection
- Patch descriptor
- Correspondences using nearest neighbors (NN) and the second NN test
- RANSAC
- Fully automatic panorama stitching of two images
- (part 2) Linear blending
- (part 2) Two-scale blending
- (part 2) Mini Planets
- (part 2) 6.865: Full Panoramas
- (part 2) Make your own panorama!
- (part 2) Halide prep

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 which you can read about throughout this pdf or in `panorama.cpp`. We will make use of homography.cpp from problem set 6.

This problem set is long, but you have two weeks to complete it. Make sure you start early!

2 Previous Problem

This Problem Set uses code from Problem Set 6, homographies. Replace the functions in `homography.cpp` and homography.h with your own code.

3 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.

3.1 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 I_x and I_y along the x and y directions (where subscripts denote derivatives). The structure tensor is

$$M = \sum w \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{1}$$

where w is a weighting function (a Gaussian in our case) and we omit the pixel coordinates for conciseness. For this, we will first compute I_x^2, $I_x I_y$ and I_y^2 at each pixel and convolve them with a Gaussian.

For this, extract the luminance of the image using the `lumiChromi` function from Pset 1.

Using a Gaussian with standard deviation `sigmaG`, blur the luminance to control the scale at which corners are extracted. A little bit of blur helps smooth things out and help extract stable mid-scale corners. More advanced feature extraction uses different scales to add invariance to image scaling.

Next, compute the luminance gradient along the x and y direction. We've added the functions `gradientX` and `gradientY` in `filtering.cpp` for you. Call these functions with the default value of `clamp`.

Then, compute the contribution of each pixel to the structure tensor, using Equation (1) for M. The matrix is symmetric and we only need to store three values per pixels. You can store them in a `Image` with three channels.

Then, compute the local weighted sum by convolving the above per-pixel contributions using a Gaussian blur with standard deviation `sigmaG*factorSigma`. Use the separable gaussian filtering function with default truncation.

1 Write a function `Image computeTensor(const Image &im, float sigmaG=1, float factorSigma=4)` that returns a 3D array (i.e. an `Image`) of the size of the input where the three channels at each location `x, y` store the three values corresponding to the $I_x I_x$, $I_x I_y$, and $I_y I_y$ components of the tensor (in this order).

3.2 Harris corners

Implementing the Harris corner detectors require a few steps, presented here in order. Read the whole subsection before implementing.

Corner response To extract the Harris corners from an image, we need to measure the corner response from the structure tensor. 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 (see class notes). **Only pixels with positive corner responses can be corners**.

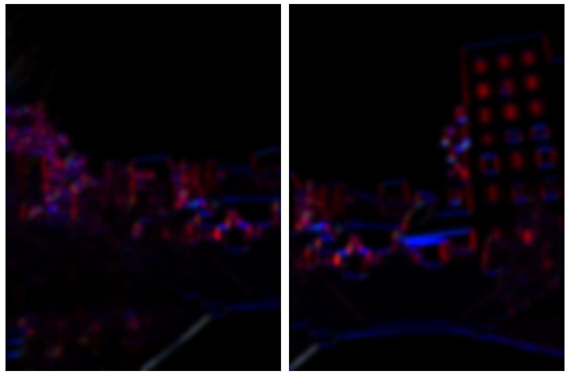


Figure 1: To visualize the results, take a look at the resulting tensor `Image`. These are our results for the Stata pair with RGB channels being xx, xy, yy.

> 2 Implement `Image cornerResponse(const Image &im, float k=0.15, float sigmaG=1, float factorSigma=4)`.

Figure 2: Our stata corner responses, normalized by the maximum values (as done in `testCornerResponse`).

Non-maximum suppression We get a strong corner response in a the neighborhood of each corner. We need to only keep the strongest response in this neighborhood. For this, we need to reject all pixels that are not a local maximum in a window of `maxiDiam`. We've written a function `maximum_filter` in `filtering.cpp` that you might find useful.

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 Points containing the coordinates of each corner. For this assignment, we provide a class `Point`. Each point corresponds to a Harris corner.

You are now ready to implement `HarrisCorners`.

3 Implement a function `vector<Point> HarrisCorners(const Image &im, float k=0.15, float sigmaG=1, float factor=4, float maxiDiam=7, float boundarySize=5)` that returns a list of 2D `Points`.

Implement the algorithm following the steps previously described.

Figure 3: Use the provided function `visualizeCorners` to verify your results `HarrisCorners`. These are our 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.

4 Descriptor and correspondences

Descriptors characterize the local neighborhood of an interest point such as a Harris corner so that we can match it with the same point in 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's coordinates and its descriptor a `Feature`, which is implemented as a class in `panorama.cpp`.

Our descriptors will be all the pixels in a `radiusDescriptor*2+1` by `radiusDescriptor*2+1` window around the associated interest point. That is, they will be a small single-channel Image of size 9×9 when `radiusDescriptor=4`, whose center pixel has the coordinates of the point.

We also want to address potential brightness and contrast variation between images. For this, we subtract the mean of each patch, and divide the resulting patch by its standard deviation. Note that, as a result of the offset and scale, our descriptors will have negative numbers and might be greater than 1. We have added the `mean` and `var` methods to the `Image` class, in `Image.cpp`.

4.1 Descriptors

> 4 Write a subroutine `Image descriptor(const Image &blurredIm, Point p, float radiusDescriptor=4)` that extracts a single descriptor around interest `Point P`. Here, `im` is a **single-channel** Image (since we will be computing descriptors based on the luminance alone).

4.2 Features

We define a `Feature` as a pair (p, d) where p is a `Point` and d is the corresponding `Descriptor` encoded as a single-channel 9x9 `Image`. See our `Feature` class in `panorama.h` for more information.

> 5 Write a function `vector <Feature> computeFeatures(const Image &im, vector<Point> cornersL, float sigmaBlurDescriptor=0.5, float radiusDescriptor=4)` that takes as input a list `cornerL` of the above Harris corners associate each of them with a descriptor. The function should return a vector of features.
>
> We compute the descriptor from a blurred single channel image. Specifically, first, extract the luminance of the input image. To avoid aliasing issues, blur the image with a Gaussian blur of standard deviation `sigmaBlurDescriptor`. Then, for each corner, extract the patch descriptor around it.

We provided you with a function `visualizeFeatures` 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.

4.3 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 L_2 norm to compare pairs of features. The

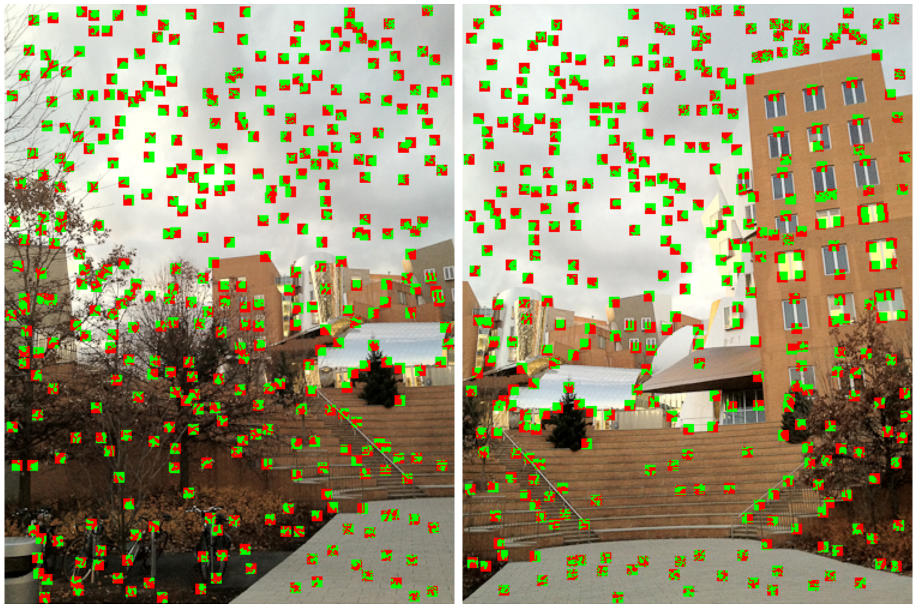


Figure 4: Visualizing features.

procedure is not symmetric (we match from the first to the second image). We will use the `FeatureCorrespondence` class, which you can also see in `panorama.cpp`.

Let us first implemented the Euclidean distance between features:

> 6 The squared distance between two descriptors is the sum of squared differences between individual values. Implement `float l2Features( Feature &f1, Feature &f2)` that returns this sum of squared distances.

Second-best test If you remember the discussion in class, for our matching procedure, not only do we consider the most similar descriptor, but also the second best. If the ratio of distances of the second best 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. Be careful between the squared distance and the distance itself. You can compute everything with just the squared distance (it's faster, no need for `sqrt`) but then you need to use the square of the threshold.

Your function `findCorrespondences` should return a vector of `FeatureCorrespondence` (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.

7 Write a function `vector <FeatureCorrespondence> findCorrespondences( vector<Feature> listFeatures1, vector<Feature> listFeatures2, float 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 helper functions might help. The search for the minimum (squared) distance can be brute force.

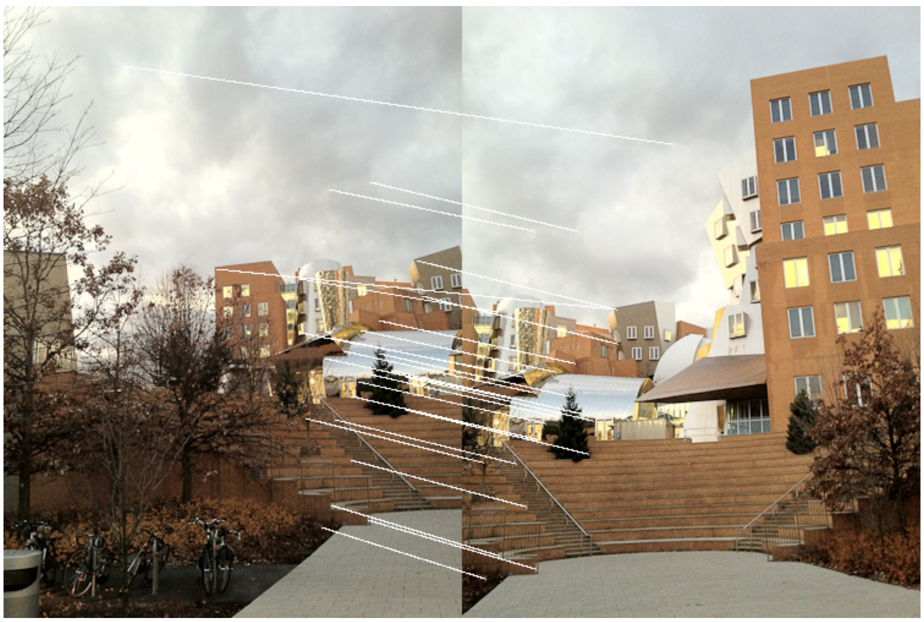


Figure 5: Use the provided `visualizePairs` to debug your matches. Note that not all correspondences are going to be perfect. We will reject outliers in the next section using RANSAC. But a decent fraction should be coherent, as shown here.

5 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. Our RANSAC function will estimate the best homography from a listOfCorrespondences.

Random correspondences For each RANSAC iteration, pick four random pairs in listOfCorrespondences. The function vector<FeatureCorrespondence> sampleCorrespondances(vector <FeatureCorrespondence> listOfCorrespondences) can help you randomly shuffle the vector, and you can then use the first four entries as the four random pairs.

Converting correspondences For each RANSAC iteration, pick four random pairs in Given four pairs of points, you should have a function from problem set 6 that computes a homography. In that problem set you used a listOfPairs as the list of pairs of points. We supply you with a function vector<CorrespondencePair> getListOfPairs(vector <FeatureCorrespondence> listOfCorrespondences).

Singular linear system In some cases, the four pairs might result in a singular system for the homography. Our first 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? Use the determinant method from Eigen's Matrix class.

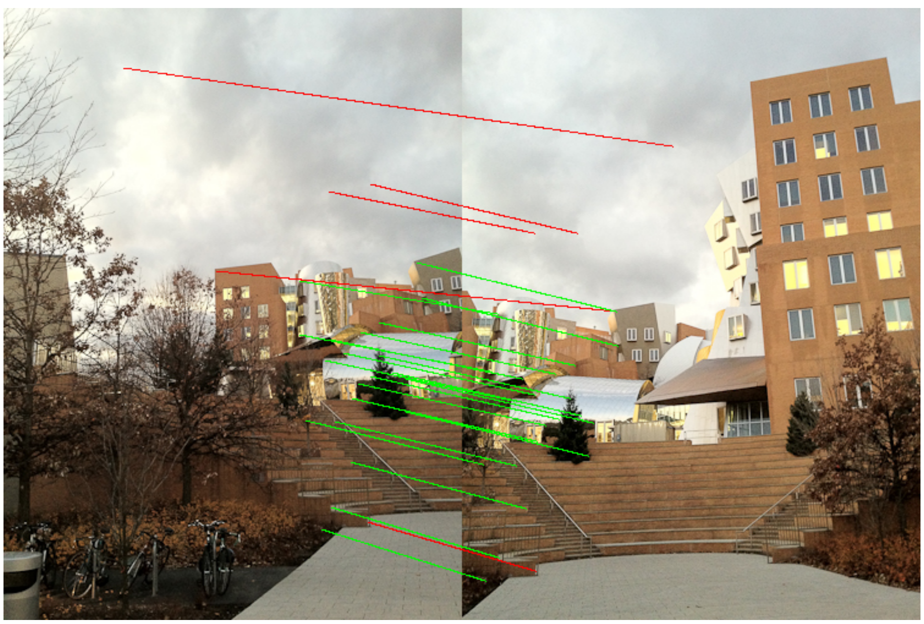
Scoring the fit We need to evaluate how good a solution this homography might be. This is done by counting the number of inliers. See the function inliers described below.

8.a Implement vector<bool> inliers(Matrix H, vector <FeatureCorrespondence> listOfCorrespondences, float epsilon=4).

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. For this, use the test $||p' - Hp|| <$epsilon. Use the output boolean vector to count the number of inliers. 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.

8.b Write a function `Matrix RANSAC(vector <FeatureCorrespondence> listOfCorrespondences, int Niter=200, float 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. `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 if $||p' - Hp||_2 <$`epsilon` ($|| \cdot ||_2$ indicates L2 norm).

You can use the provided function `visualizePairsWithInliers` to see which correspondences are considered inliers. It outputs an image 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, RANSAC uses randomized trials.



You can also use the provided `visualizeReprojection`, which 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. Our reproductions for Stata are below. The result below further emphasizes that RANSAC is probabilistic: the set of inliers is not exactly the same as above.

6 Automatic panorama stitching

> 9 Write a function `Image autostitch(Image &im1, Image &im2, float blurDescriptor, float radiusDescriptor)`
> that takes two images as input and automatically outputs a panorama image where the first image is warped into the domain of the second one. You should get a similar-ish result to the last assignment, but automatically.

Try it on the Stata, Boston-skyline and at least another pair of images.

7 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.

7.1 Linear blending

We will first implement a simple smooth transition between images. For this, the final output will be a weighted average of the reprojected inputs, where the weights decrease from 1 at the center of an image to 0 at the edges.

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

We will use piecewise linear weights in the source domain. The weights will be given by a separable function, which means that it is the product of a function only in x and another function only in y. Each of these two functions will be piece-wise linear with a value of 1.0 in the center and 0.0 at the edges. For images with even width or height, the center doesn't have to be at a pixel (i.e. if the image has width 2, the center should be at 0.5). If you need a function to compute absolute value, make sure you use `fabs`, which is for floats.

10.a Implement `Image blendingweight(int imwidth, int imheight)`, which returns an Image of weights as described.

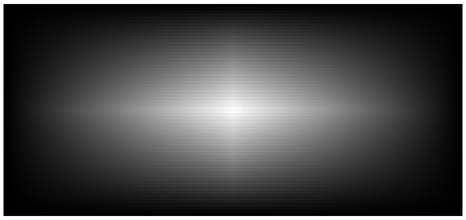
10.b Implement `void applyhomographyBlend(const Image &source, const Image &weight, Image &out, Matrix &H, bool bilinear)`, which is similar to `applyHomography` (or `applyHomographyFast`). But instead of writing the pixels of the input image to the output image, this function should **add** the pixels of the input image (`source`) times the `weight` to the output image.

10.c Implement `Image stitchLinearBlending(const Image &im1, const Image &im2, const Image &we1, const Image &we2, Matrix H)`, which stitches two images using the given weights. Note that stitch should not do any normalization – if the two weights don't add up to 1 at some pixel, that's ok. Use `applyhomographyBlend` to help you with this.

10.d Implement `stitchBlending(Image &im1, Image &im2, Matrix H, int blend)`, where blend can be 0, 1 or 2. If blend is 0, then this should behave the same as the normal stitching (**using im2 as reference**). If blend is 1, stitch the two images using linear blending. Make sure you figure out some way of keeping track of the sum of the weights at each pixel. If blend is 2, use 2-scale blending (which we will implement in the next section).

10.e Implement `Image autostitch(Image &im1, Image &im2, int blend, float blurDescriptor=0.5, float radiusDescriptor=4)`, which is the same as the first part of the pset's autostitch, except it calls the stitch implementation above, with the fourth parameter `blend`. In other words, it automatically finds the homography (via RANSAC) before doing the stitch with blending 0, 1 or 2.

Here is our weight map for the poster image, and our applyhomographyBlend applied to the green/poster combo with a constant weight of 0.5 for every pixel of the poster:



7.2 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.

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.

11 Finish implementing `stitchBlending(Image &im1, Image &im2, Matrix H, int blend)` for the case when `blend` is 2.

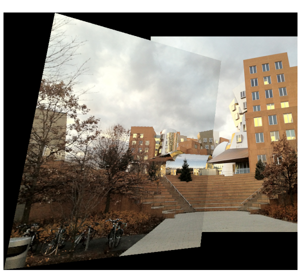
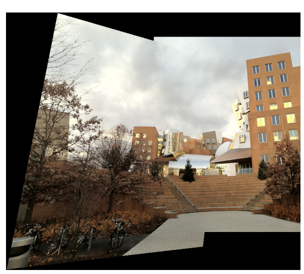
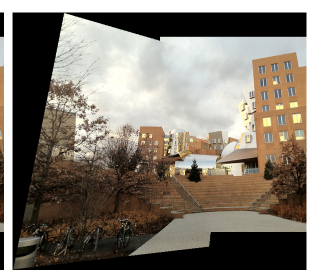
First, decompose each source image into low frequencies and high frequencies using a Gaussian blur. Use a spatial sigma of 2 pixels.

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.

Our results for blend = 0, 1, and 2 with a pre-determined homography for Stata:

zoom in on the tree on your pdf viewer to see the difference between the second and third image. And the Boston autostitch (since we all know Boston skyline examples are cooler):

You can see more problems in the linear blending, as expected from lecture notes. Note, interestingly, that the projections are slightly different. Why? We're running RANSAC each time, and we're getting slightly different Homographies!

8 Mini planet

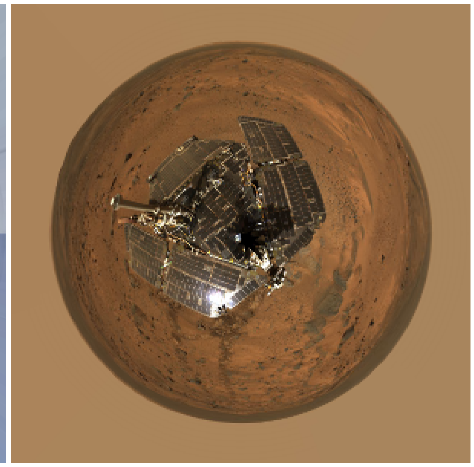
Assume you've been given a panorama image. Use the stereographic projection to yield the popular mini planet view. See e.g. http://en.wikipedia.org/wiki/Stereographic_projection, http://www.miniplanets.co.uk/ .

12 Implement `Image pano2planet(const Image &pano, int newImSize, bool clamp=true)`. Make a new image of square size (`newImSize`), and for each pixel `(x, y)` in the new image, compute the polar coordinates `(angle, radius)` assuming that the center is the floating point center as in `blendingweights`. Map the bottom of your input panorama to the center of the the new image and the top to a radius corresponding to the distance between the center and the right edge (in the square output).

The left and right sides of the input panorama should be mapped to an angle of 0, along the right horizontal axis in the new image with increasing (counter-clockwise) angle in the output corresponding to sweeping from left to right of the input panorama. Assume standard polar coordinate conventions (angle is 0 along right horizontal axis and $\frac{\pi}{2}$ is along the top vertical axis). Use `interpolateLin` to copy pixels from panorama to planet image. Hint: see C++'s `atan2`.

Here's a Boston winter panorama, and the resulting planet. Note that there is a rough line in the middle of the sky. This is because the panorama is not really 360 degrees. By contrast, the Mars panorama on the right is 360, and we have a much smoother planet result (Mars pano credit: http://mars.nasa.gov/mer/gallery/panoramas/spirit/2005.html):

9 6.865: Stitch N Images (6.815: Extra Credit 10%)

In this section you're going to compose a larger panorama using N images! Finally, some **really** fun stuff!

13.a Implement vector<Matrix> sequenceHs(vector<Image> ims, float blurDescriptor=0.5, float radiusDescriptor=4); which computes a sequence of N-1 homographies for N images. H[i] should take ims[i] to ims[i+1].

13.b Implement vector <Matrix> stackHomographies(vector <Matrix> Hs, int refIndex);. This takes the N-1 homographies from the previous function, and translates them into N homographies for the N images. H[i] takes ims[i] to image ims[refIndex]. Therefore, H[refIndex] should be identity. Note that this requires some chaining of pairwise homographies to get the global homographies. Pay attention: things are different before and after the reference image.

13.c Implement BoundingBox bboxN(const vector<Matrix> &Hs, const vector<Image> &ims);, which takes in N-1 homographies and N images, and computes the overall bounding box.

13.d Implement Image autostitchN(vector<Image> ims, int refIndex, float blurDescriptor=0.5, float radiusDescriptor=4);, which computes the sequence of homographies using sequenceHs, then propagates those homographies using stackHomographies, then computes the overall bounding box, then the translation of the bounding box to (0,0), and finally applies the homographies to all images to get the panorama. **Use linear blending**.

Here's our answers with refIndex=1 for Boston and Guédélon:

10 Make your own panorama

Capture your own sequence of images and run it through your automatic algorithm. Two images for 6.815, at least three images (and use the N stitching) 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, via *stellar*, both your source images and your results.

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

11 Extra credits (15% max)

For any extra credit you attempt (5% each), please write a new test function in your main file, and include the name of the test function in the submission questionnaire. This is a requirement for getting the extra credit.

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

Automatically crop margins (5%) It's not trivial, but not too hard. Find a reasonable way to crop out the black margins automatically. Be careful not to crop too much from the Image.

Cylindrical reprojection (5%) 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..width, 0..height]. Define `size=max(height, width)`, then the normalized coordinates

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

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 (5%) 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.

12 Submission

Turn in your files to the online submission system (link is on Stellar) and make sure all your files are in the `asst` directory under the root of the zip file. If your code compiles on the submission system, it is organized correctly. The submission system will run code in your main function, but we will not use this code for grading. The submission system should also show you the image your code writes to the `./Output` directory

In the submission system, there will be a form in which you should answer the following questions:

- How long did the assignment take? (in minutes)
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented and their function signatures if applicable
- Collaboration acknowledgment (you must write your own code)
- What was most unclear/difficult?
- What was most exciting?