HW2 Write-Up

16-720: Computer Vision

Spring 2016

Cole Gulino

February 17, 2016

1. Keypoint Detector

1.1: Gaussian Pyramid

The Gaussian Pyramid is implemented in the provided function:

```
    GaussianPyramid  = createGaussianPyramid(im, sigma0, k, levels)
```

The provided function to show the pyramid is:

```
displayPyramid(pyramid)
```

An example is given in Figure 1 that uses the following parameters:

$$\sigma_0 = 1;\ k = \sqrt{2};\ \text{levels} = \left[-1, 0, 1, 2, 3, 4\right];\ \theta_c = 0.03;\ \theta_r = 12$$

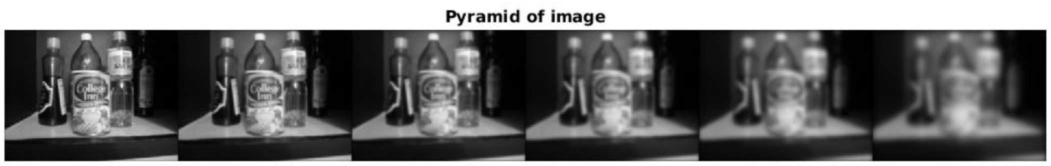


Figure 1) Gaussian Pyramid Example

1.2: The DoG Pyramid

The equation for calculating the Difference of Gaussian Pyramid is shown below:

$$D_l(x, y, \sigma_l) = G(x, y, \sigma_{l-1}) * I(x, y) - G(x, y, \sigma_l) * I(x, y) \tag{2}$$

$$= GP_l - GP_{l-1} \tag{3}$$

Q1.2: The Difference of Gaussian Pyramid was written in the function:

```
    [DoGPyramid, DoGLevels] = createDoGPyramid(GaussianPyramid, levels)
```

An example of the Difference of Gaussian function is shown in Figure 2.

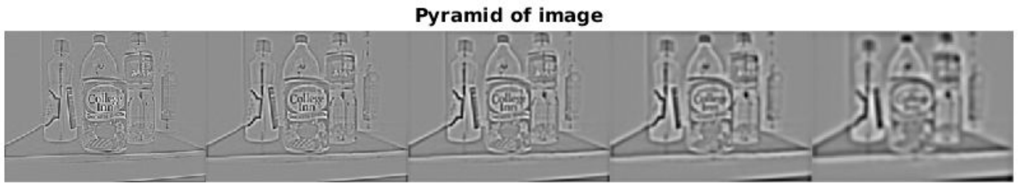


Figure 2) Difference of Gaussian Pyramid Example

1.3: Edge Suppression

When looking for features in the Difference of Gaussian function, edges and blobs are shown strongly. Edges are not good for detection, so we can use edge suppression to remove these features.

Q1.3: Implement the function:

```
      PrincipalCurvature = computePrincipalCurvature(DoGPyramid)
```

The function computes the principal curvature of a pixel using the equation:

$$R = \frac{\mathrm{Tr}(H)^2}{\mathrm{Det}(H)} = \frac{(\lambda_{min} + \lambda_{max})^2}{\lambda_{min}\lambda_{max}}$$

Where the Hessian is described as:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix}$$

I calculated the Hessian in MATLAB commands:

```
[Dx, Dy] = gradient(DoGPyramid)
[Dxx, Dxy] = gradient(Dx)
[Dxy, Dyy] = gradient(Dy)
```

1.4: Detecting Extrema

Q1.4: Here we write a function that detects the local extrema (max and min) in both scale and space:

```
 locsDoG = getLocalExtrema(DoGPyramid, DoGLevels, PrincipalCurvature,
                              th_contrast, th_r)
```

The parameter `th_contrast` is used in order to set a threshold for Difference of Gaussian pixel response and `th_r` thresholds for maximum principal curvature.

This function returns the locations of local extrema in the form:

$$(x, y, \sigma_l)$$

1.5: Putting it all Together

Q1.5: Here we gather the extrema point and plot the keypoints detected.

An example is shown in Figure 3.

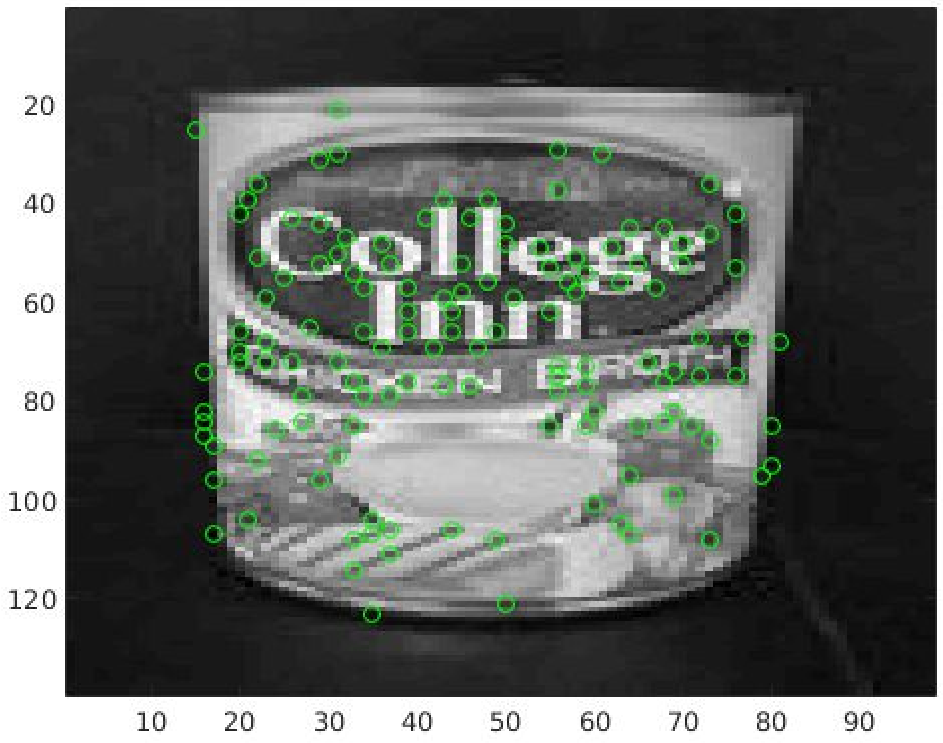


Figure 3) Image with Keypoints Detected

2. BRIEF Descriptor

2.1: Creating a Set of BRIEF Tests

The BRIEF descriptor is an n-bit long vector where each bit is the result of this test:

$$\tau(p; x, y) := \begin{cases} 1, & \text{if } p(x) < p(y). \\ 0, & \text{otherwise.} \end{cases}$$

In order to get the set of BRIEF tests, we must first create two sets of points in the patch-width x patch-width test space around a keypoint.

The vectors `compareX` and `compareY` contain values [1 (patch-width)^2] that correspond to an (x,y) value offset around the image.

Q2.1: Write the function to create the x and y pairs that we will use for comparison to the BRIEF descriptor test.

The method I used for generating the methods was gathered from the reference paper [1] was the first method mentioned by sampling the test pairs from a uniform distribution in the form:

$$(\mathbf{X}, \mathbf{Y}) \sim \text{i.i.d.Uniform}\left(-\frac{S}{2}, \frac{S}{2}\right)$$

The paper claims that random sampling methods including the one above are the preferred methods.

I sampled 256 test values each in `compareX` and `compareY` that I stored in the file: `testPattern.mat`.

2.2: Compute the BRIEF Descriptor

Q2.2: Write the function:

`[locs,desc] = computeBrief(im, GaussianPyramid, locsDoG, k, levels, compareX, compareY)`

This function returns the locations of the keypoints that are valid after running the BRIEF descriptor. Those that are invalid are close to the edge of the image, so that they cannot have the BRIEF test run on them in `locs.`

It also returns the descriptors in `desc`.

2.3: Putting it Together

Q2.3: Write a function:

`[locs, desc] = briefLite(im)`

This function puts together everything in part 2.

2.4: Check Point: Descriptor Matching

Here I will include images that show the Descriptor matching working. In order to accomplish this, I wrote the required script `testMatch.m.`

Figures 4 - show the comparisons.

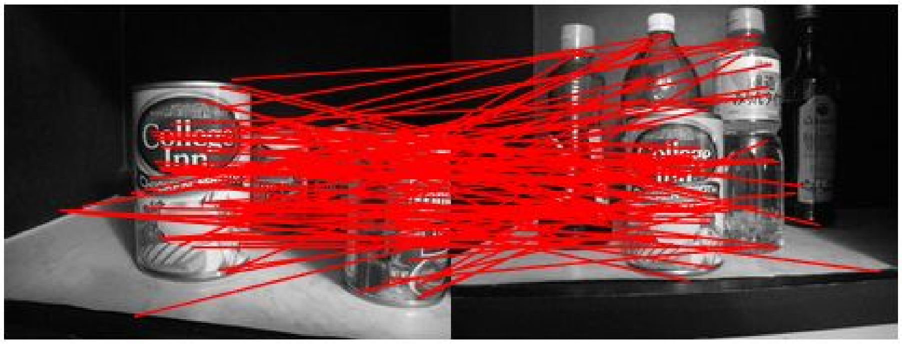


Figure 4) Comparison of Two Can Images

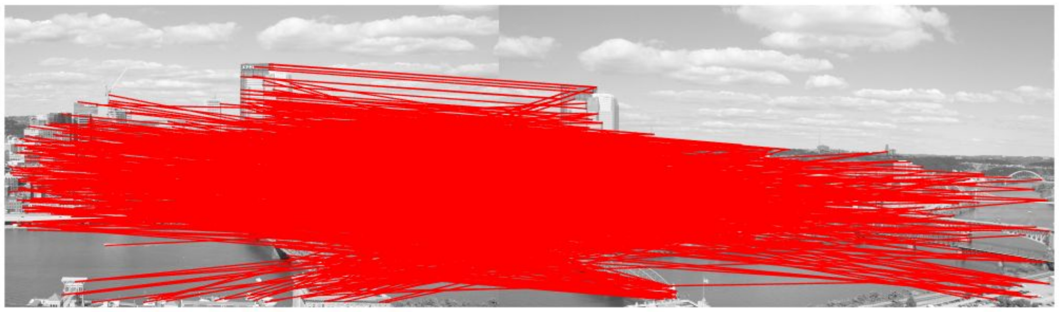


Figure 5) Comparison of Incline Images

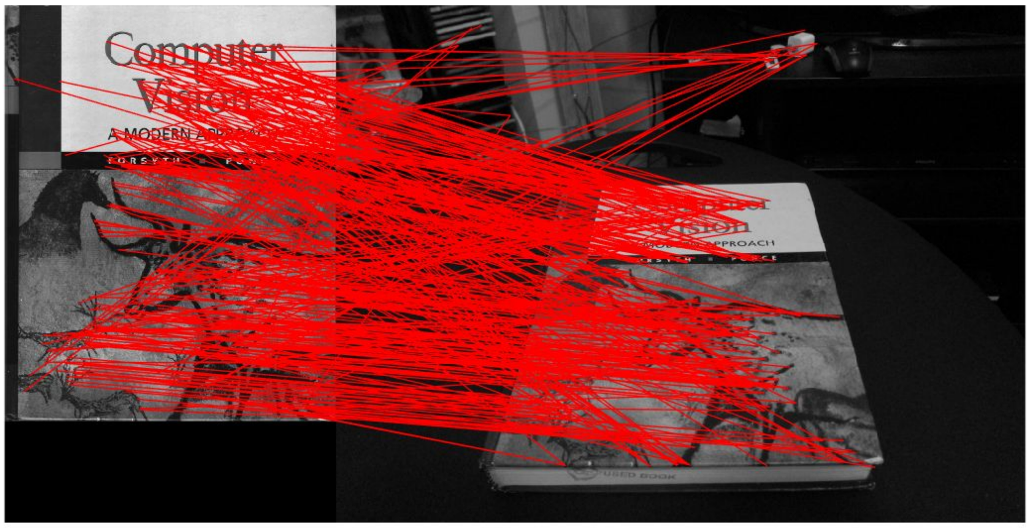


Figure 6a) Book Comparison for pf_desk.jpg

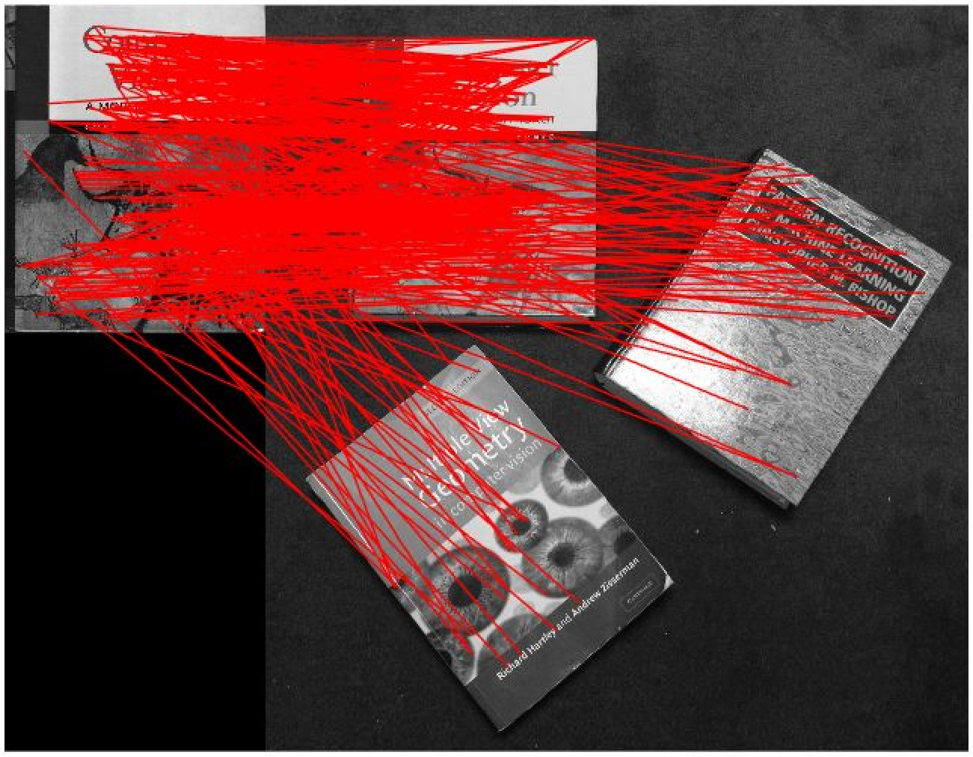


Figure 6b) Book Comparison for pf_floor.jpg

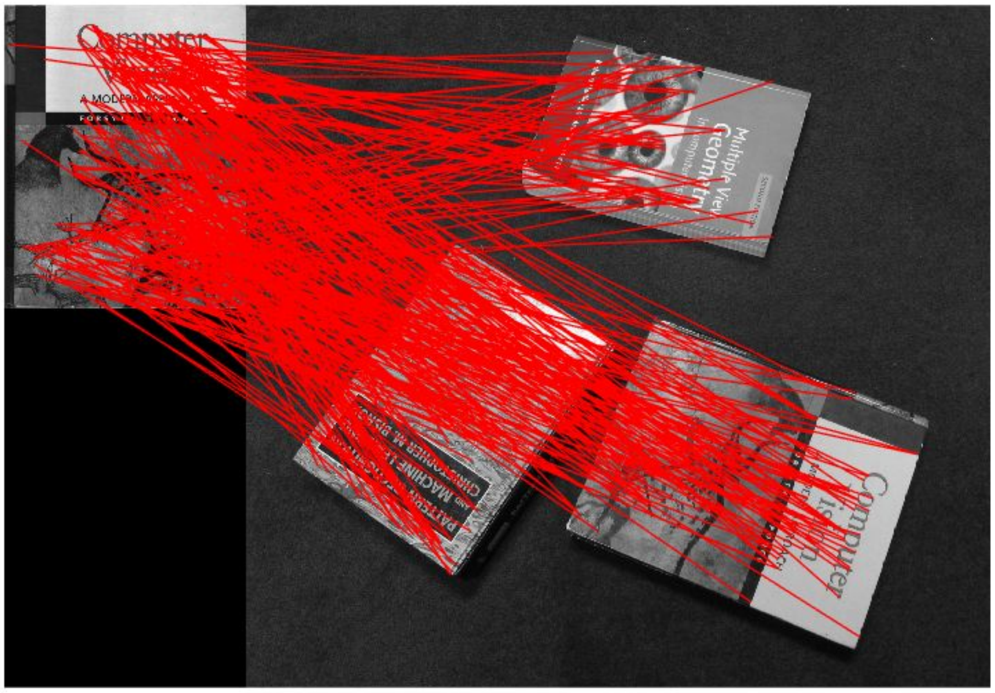


Figure 6c) Book Comparison for pf_floor_rot.jpg

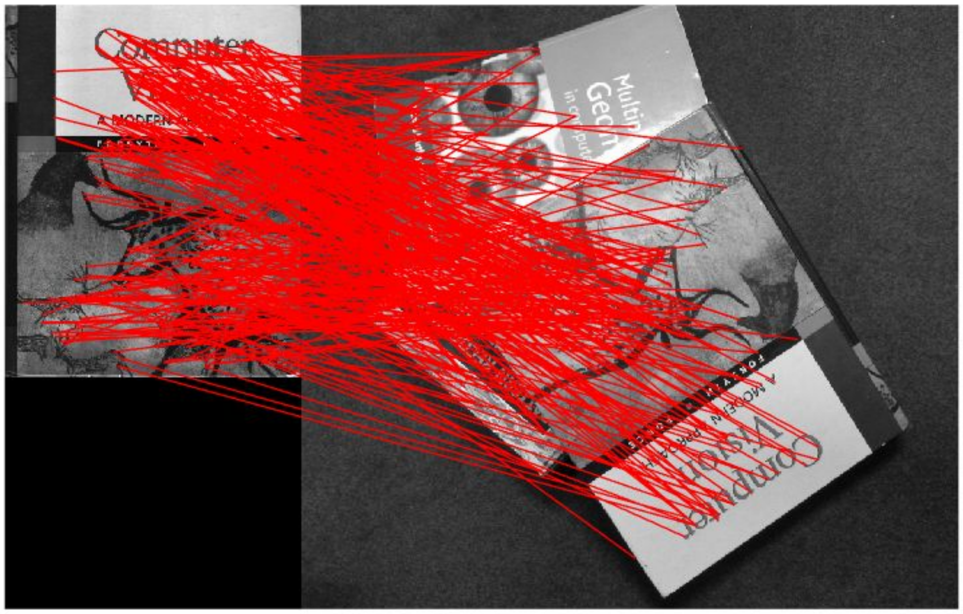


Figure 6d) Book Comparisons for pf_pile.jpg

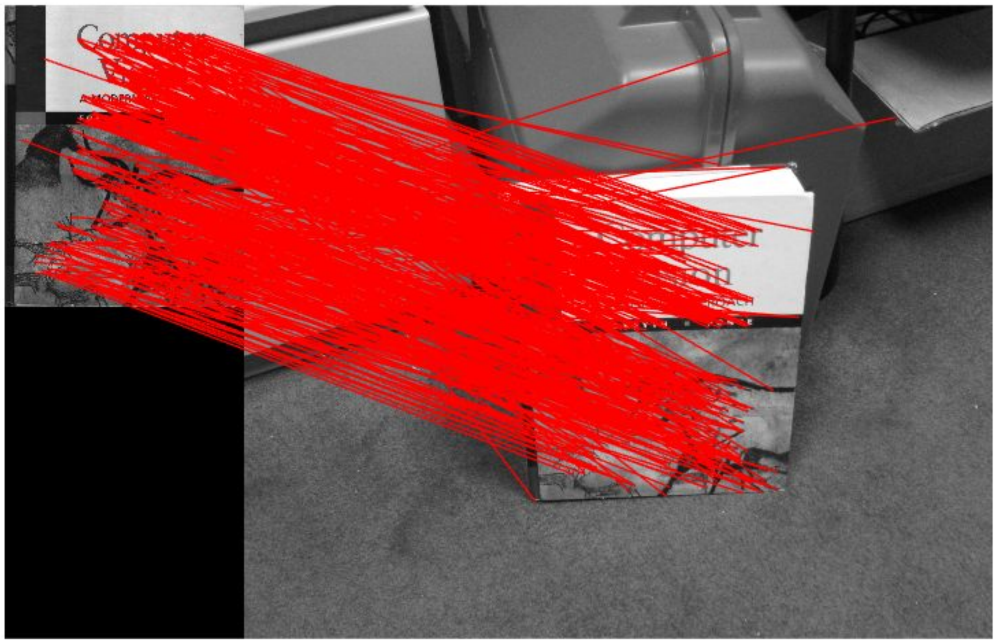


Figure 6e) Book Comparisons for pf_stand.jpg

The best performance of the book comparisons is shown in Figure 6e. This is whenever the book is in the same general orientation and without any other objects that are similar to the book.

The worst performing comparison is Figure 6c. This corresponds to an image with multiple books and the image is rotated. It seems from this that performance is diminished whenever the object to detect features from has a high angle of rotation and the frame contains other objects that have similar features.

2.5: BRIEF and rotations

Q2.5: Here I wrote a script that takes the model_chickenbroth.jpg and rotates it in angles from 0 to 350 degrees and then finds the matching features. The number of matching features is stored and then shown in a bar chart detailing Angle of Rotation vs. Number of Correct Matches.

Figure 7 shows the bar chart with the parameter `ratio = 0.95`:

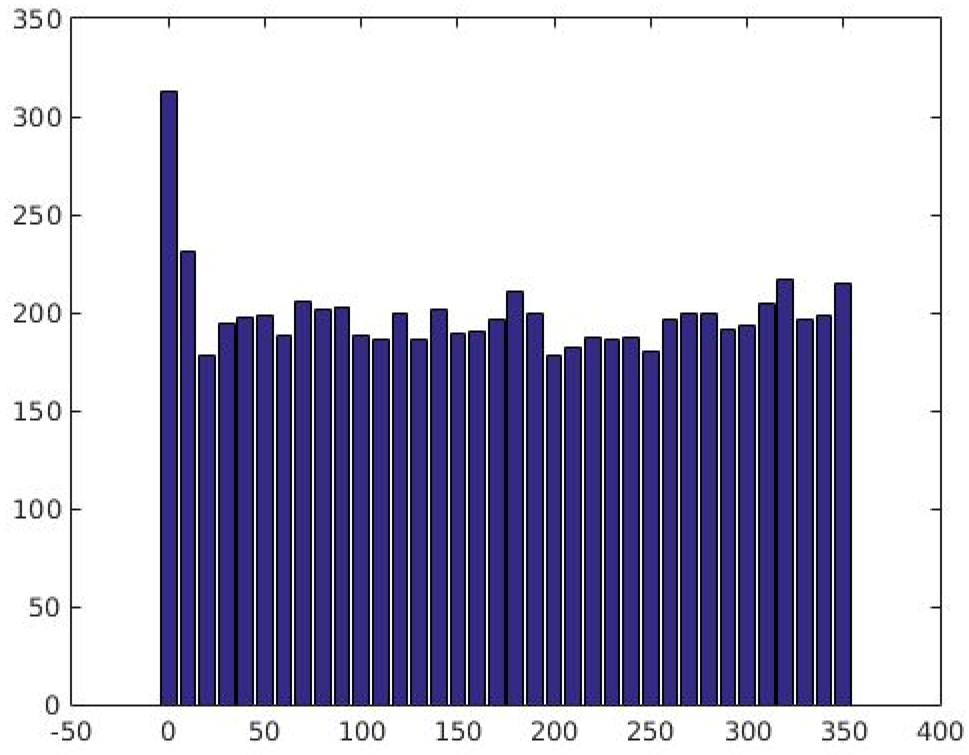


Figure 7) Angle of Rotation vs Number of Correct Matches

The descriptor works very well for zero rotation, and then works fairly well after it. There is a steep drop off after a zero rotation, but after that it remains fairly even. There are small bumps around rotation = 180 and 350. This is when the rotated image resembled the original image the most.

The drop off in performance comes from the fact that BRIEF is not rotationally invariant. Because the test cases are relative to the keypoints, a rotated image will have a different set of test points sampled from it. Depending on the rotation, the performance will drop due to the sampled pixels around keypoints being different for the non-rotated and the rotated image.

3. Planar Homographies: Theory

Assume that you have two cameras which have points in a 2D plane so that:

$$\mathbf{C}_1 \text{ with point } p \equiv (u_1, v_1, 1)^T$$
$$\mathbf{C}_2 \text{ with point } q \equiv (u_2, v_2, 1)^T$$

Both are confined to the II plane, so we assume some relationship between them, so that there exists a relationship such that a common 3x3 matrix H, so that for any p and q:

$$\mathbf{p} \equiv \mathbf{Hq}$$

This is called planar homography.

Q3.1: Assume a set of points where:

$$\mathbf{C}_1 \text{ with points } \mathbf{p} = \{p^1, p^2, ..., p^N\}$$
$$\mathbf{C}_2 \text{ with points } \mathbf{q} = \{q^1, q^2, ..., q^N\}$$

Assume that we know there exists an unknown homography between corresponding points for all i such that:

$$\forall i, \exists \mathbf{H} \text{ such that } p^i \equiv \mathbf{H}q^i$$

(a). Given N correspondences in p and q and using the above equation, derive a set of 2N independent linear equations in the form:

$$\mathbf{Ah} = \mathbf{0}$$

Where h is a vector of the elements of H and A is a matrix composed of elements derived from the point coordinates. Write out an expression for A.

Let: $\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$

$$\begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix}$$

Then if we solve for $u_1 \text{ and } v_1$:

$$u_1 = h_{11}u_2 + h_{12}v_2 + h_{13}$$
$$v_1 = h_{21}u_2 + h_{22}v_2 + h_{23}$$

If we want to satisfy the third equation that:

$$1 = h_{31}u_2 + h_{32}v_2 + h_{33}$$

We need to divide the whole matrix by the right side of the equation so that:

$$u_1 = \frac{h_{11}u_2 + h_{12}v_2 + h_{13}}{h_{31}u_2 + h_{32}v_2 + h_{33}}$$
$$v_1 = \frac{h_{21}u_2 + h_{22}v_2 + h_{23}}{h_{31}u_2 + h_{32}v_2 + h_{33}}$$

Now we can set the equation to 0:

$$u_1\left(h_{31}u_2 + h_{32}v_2 + h_{33}\right) - \left(h_{11}u_2 + h_{12}v_2 + h_{13}\right) = 0$$
$$v_1\left(h_{31}u_2 + h_{32}v_2 + h_{33}\right) - \left(h_{21}u_2 + h_{22}v_2 + h_{23}\right) = 0$$

I can solve out this equation to get:

$$h_{31}u_1u_2 + h_{32}u_1v_2 + h_{33}u_1 - h_{11}u_2 - h_{12}v_2 - h_{13} = 0$$
$$h_{31}v_1u_2 + h_{32}v_1v_2 + h_{33}v_1 - h_{21}u_2 - h_{22}v_2 - h_{23} = 0$$

We can then formulate this equation as a matrix:

$$\begin{bmatrix} -u_2 & -v_2 & -1 & 0 & 0 & 0 & u_1u_2 & u_1v_2 & u_1 \\ 0 & 0 & 0 & -u_2 & -v_2 & -1 & v_1u_2 & v_1v_2 & v_1 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \mathbf{0}$$

So then A for one instance of i:

$$\mathbf{A} = \begin{bmatrix} -u_2 & -v_2 & -1 & 0 & 0 & 0 & u_1u_2 & u_1v_2 & u_1 \\ 0 & 0 & 0 & -u_2 & -v_2 & -1 & v_1u_2 & v_1v_2 & v_1 \end{bmatrix}$$

So for each instance of q and p, there are two separate equations. Since there are N instances of q and p there are 2N equations.

(b.) How many elements are there in h

As shown in part (a.), h is in the form:

$$\mathbf{h} = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix}$$

Thus there are **9 elements in h**.

(c.) How many point pairs (correspondences) are required to solve the system?

Since there are 9 values in **h**, **h** has 9 Degrees of Freedom. If you normalize the **h** by dividing by h33 then you have only 8 Degrees of Freedom. As seen in part A, each point correspondence encodes two of the 2N rows of the A matrix.

So, in order to solve the 2N independent equations, we need **N point correspondences.**

(d.) Show how to estimate the elements in **h** to find a solution to minimize this homogeneous linear least squares system. Step us through this procedure.

The citation for this section is found here [2].

We know that the optimal **h** corresponds to the eigenvector of the matrix A that corresponds to the smallest eigenvalue.

One solution is to solve:

$$\mathbf{A}\mathbf{h} = \mathbf{0}$$
$$\mathbf{A}^T\mathbf{A}\mathbf{h} = \mathbf{A}^T\mathbf{0}$$
$$\mathbf{A}^T\mathbf{A}\mathbf{h} = \mathbf{0}$$

Now we can use SVD so that $\mathbf{A}^T\mathbf{A}$ represents the singular values of A. The right-most row of the matrix $\mathbf{A}^T\mathbf{A}$ corresponds to the smallest eigenvalue of A. This is the vector that we want **h** to be in order to minimize the homogenous linear least squares system.
We can solve this then by finding the SVD of A:

$$\mathbf{A} = UDV^T$$

Where V holds the eigenvectors. So then the right-most value of V is the vector we should set to **h**.

An alternative method is to use Rayleigh quotient theorem.

This allows you to find the Rayleigh quotient:

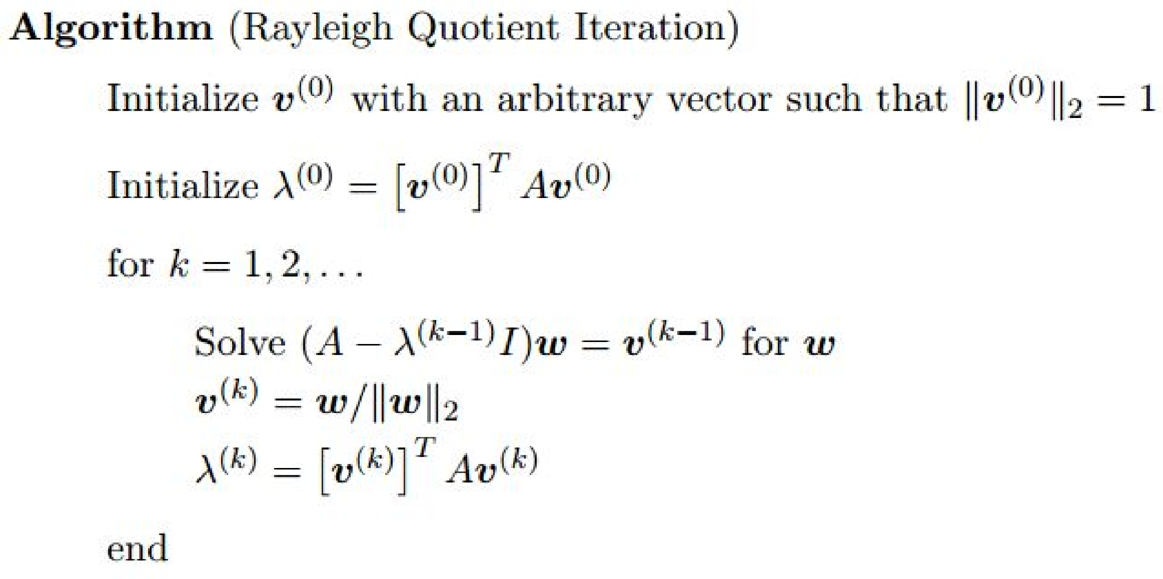
$$r(\mathbf{x}) = \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}}$$

If we say that x is an eigenvector of A, which is what we want **h** to be, we can say:

$$r(\mathbf{h}) = \frac{\lambda\mathbf{h}^T\mathbf{h}}{\mathbf{h}^T\mathbf{h}} = \lambda$$

Where λ is an eigenvalue. The linear least squares system is minimized if $\lambda = \lambda_{min}$.

Figure 8 from [2] shows an algorithm to find the value of h that solves this:



Algorithm (Rayleigh Quotient Iteration)

```
    Initialize $\boldsymbol{v}^{(0)}$ with an arbitrary vector such that $\|\boldsymbol{v}^{(0)}\|_2 = 1$
    Initialize $\lambda^{(0)} = \left[\boldsymbol{v}^{(0)}\right]^T A\boldsymbol{v}^{(0)}$
    for $k = 1, 2, \ldots$
        Solve $(A - \lambda^{(k-1)}I)\boldsymbol{w} = \boldsymbol{v}^{(k-1)}$ for $\boldsymbol{w}$
        $\boldsymbol{v}^{(k)} = \boldsymbol{w}/\|\boldsymbol{w}\|_2$
        $\lambda^{(k)} = \left[\boldsymbol{v}^{(k)}\right]^T A\boldsymbol{v}^{(k)}$
    end
```

Figure 8) Rayleigh Quotient Algorithm from [2]

4. Planar Homographies: Implementation

Q4.1: Implement the function:

```
H2to1 = computeH(p1,p2)
```

To do this, we use the theoretical knowledge from section 3. I used the intuition from section 3 that we can compute the vector **h** by computing the SVD of the matrix **A**:

$$\mathbf{A} = \begin{bmatrix} -u_2 & -v_2 & -1 & 0 & 0 & 0 & u_1 u_2 & u_1 v_2 & u_1 \\ 0 & 0 & 0 & -u_2 & -v_2 & -1 & v_1 u_2 & v_1 v_2 & v_1 \end{bmatrix}$$

Where the **A** shown above is for one set of pairs. The matrix is thus 2Nx9 where N is the number of pairs in p1 and p2. I then generated the A matrix for each of the points and ended up with a 2Nx9 matrix. I used the matlab function `svd()` in order to get:

$$\mathbf{A} = UDV^T$$

From section 3, I determined that the **h** that minimizes the equation $\mathbf{Ah} = \mathbf{0}$ is the one that corresponds to the eigenvector of **A** that corresponds to the smallest eigenvalue of **A**. We can get this by setting **h** to be the last column of V.

I then converted the 9x1 vector **h** to the 3x3 matrix **H** through the following relation:

$$\mathbf{h} = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

5. Stitching it Together: Panoramas

Q5.1: Implement the function:

```
[panoImg] = imageStitching(img1, img2, H2to1)
```

This function takes in two images and the Homography matrix that corresponds to the the transformation between the matched points.

I wrote a script called: `getHScript.m` that puts everything into the workspace that is needed to run this function.

It also runs the `imageStitching` function. As specified in the prompt, I stored the warped image and the H matrix in the results folder.

Figure 8 shows my warped image for Q5.1:

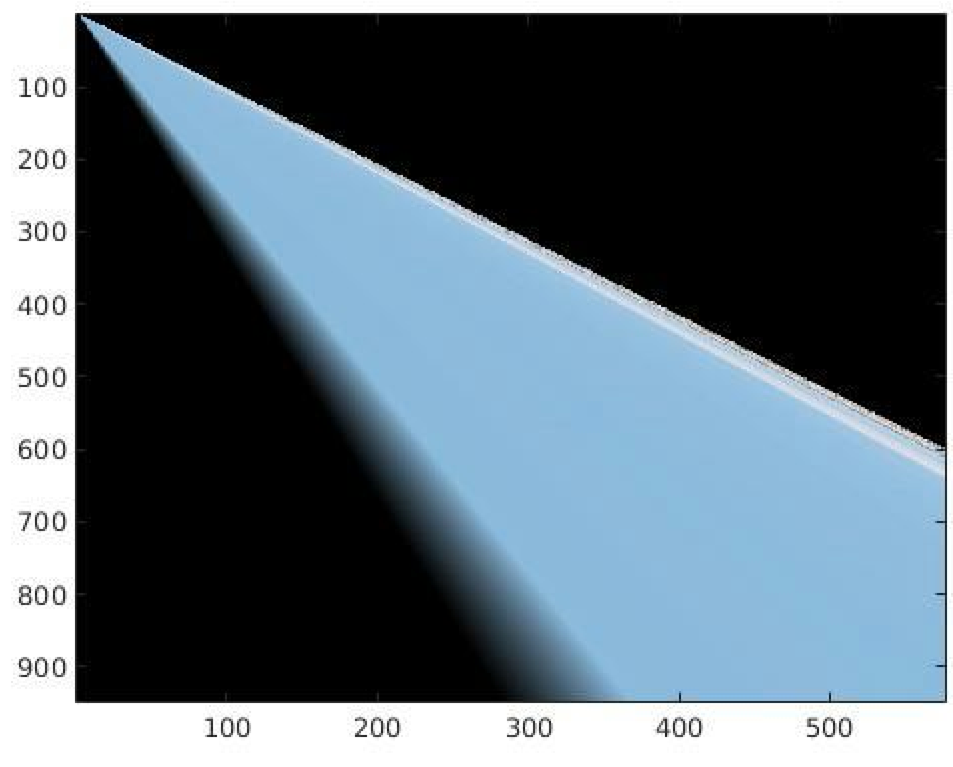


Figure 8) Warped Image of Q5.1

The warped image does not seem like we want it to be, this is because the warping technique that we used is very sensitive to outliers. Increasing the `ratio` parameter to the [matches] function returns fewer but better matches.Increasing the `ratio` parameter can increase the effectiveness of the stitching. I set the ratio parameter to 0.99 for the image in Figure 8.

Q5.2: Implement function:

```
          [panoImg] = imageStitching_noClip(img1, img2, H2to1)
```

This function is similar to the function created in Q5.1 except that it aims to prevent the image from being clipped.

As seen in Figure 8, the bottom of the image is clipped out of the frame. This function aims to fix that.

In this function, we are finding a matrix M that we can use to warp both the original image and the previously warped image in order to create a panorama without any deformation of the images.The M matrix should only scale and translate.

Figure 9 shows the panorama:

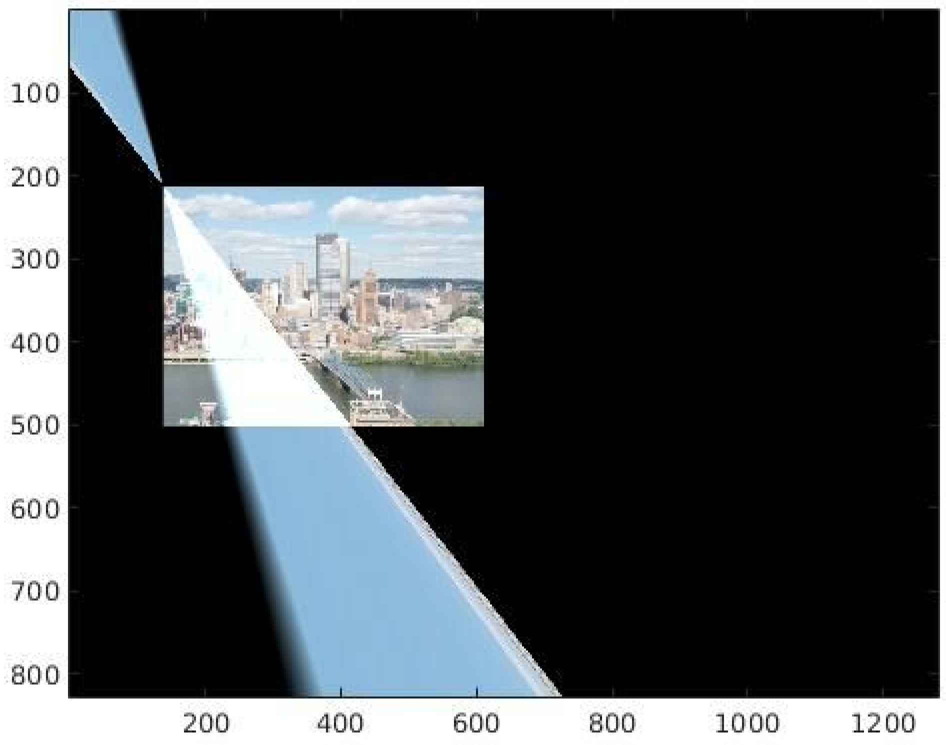


Figure 9) Panorama for Q5.2

Figure 9 shows that there is still some problems with the outlier, but there is clearly a framework to be able to show both images in one figure.

6. RANSAC

Q6.1: Implement the function:

```
        [bestH] = ransacH(matches, locs1, locs2, nIter, tol)
```

This function is designed to find the best Homography matrix that corresponds to the most number of inliers. It thus minimizes the outliers found and will provide much better results than in part 5.

Figure 10 shows an example of the panorama allowing for clipping using RANSAC.

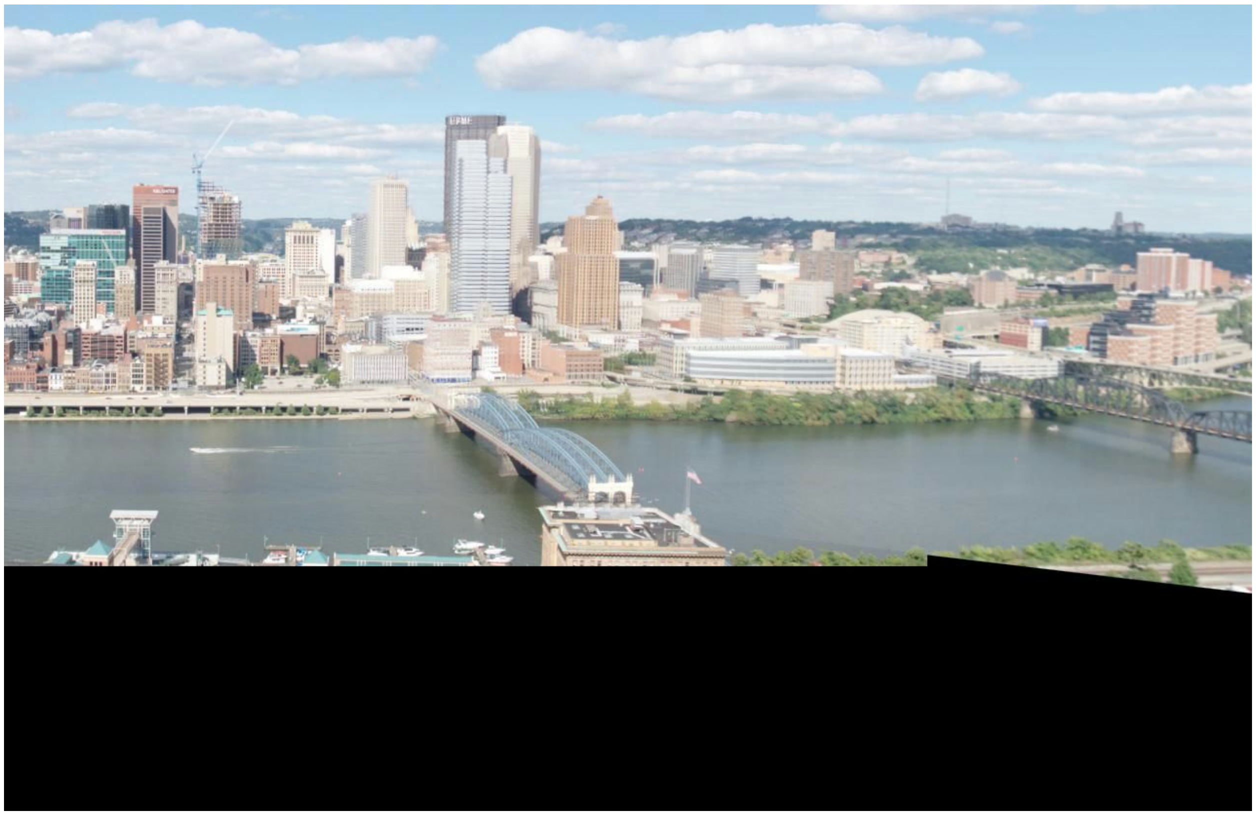


Figure 10) Panorama with Clipping Using RANSAC

Figure 11 shows the image of the panorama without Clipping using RANSAC

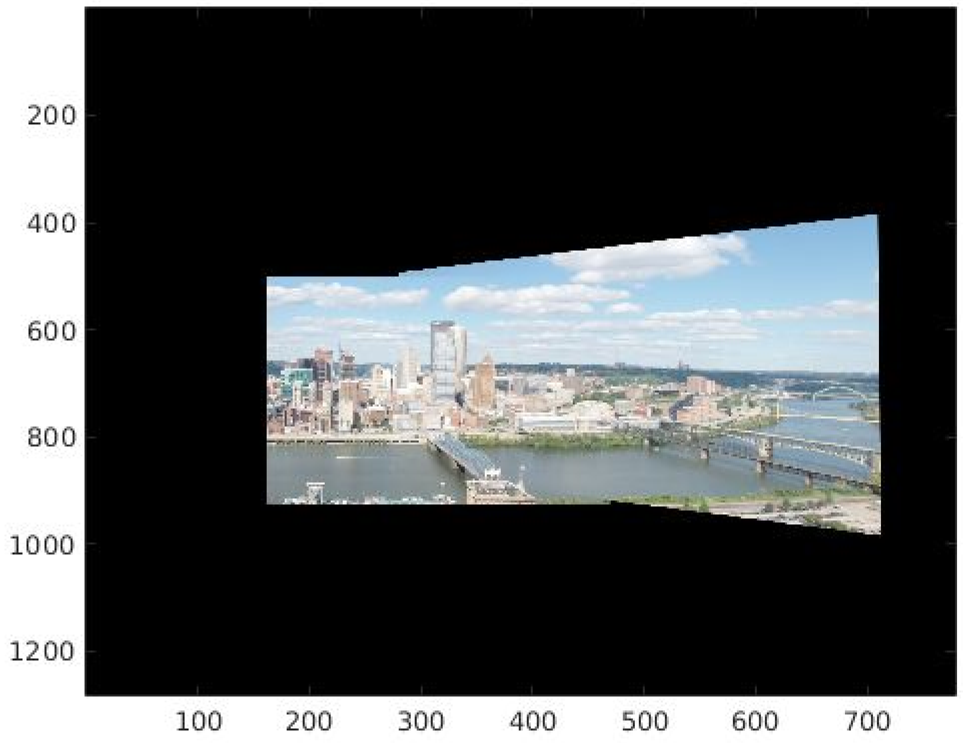


Figure 11) Panorama without Clipping Using RANSAC

Q6.2: Write the function:

```
im3 = generatePanorama(im1, im2)
```

This function takes in two images and returns a panorama image. A sample output for the panorama in Figure 11.

References

[1] BRIEF: Binary Robust Independent Elementary Features
[2] The Rayleigh Quotient and Inverse Iteration