# Comp 558 – Assignment 3

Manoosh Samiei
McGill University

November 23, 2019

**Question 1:**

**Technical differences between the provided implementation and my approach in assignment 2:**

In the provided SIFT implementation, each key point **detector** is described by four parameters: the key point center coordinates *x* and *y*, its *scale*, and its *orientation* (in radians). For the key point's orientation, the same approach is taken as what we did in previous assignment: a neighborhood is taken around the key point location. Then depending on the scale (in Gaussian Pyramid), the gradient magnitude direction is calculated in that region. An orientation histogram with 36 bins covering 360 degrees is created. (It is weighted by gradient magnitude and gaussian-weighted circular window with $\sigma$) Then the highest peak in the histogram is taken. Here one difference with the previous approach is that any peak above 80% of the highest peak is also considered to calculate the orientation. Hence, when the orientation is ambiguous, the SIFT detectors returns a list of up to four possible orientations, constructing up to four frames for each detected image blob. This contributes to the stability of matching.

Another difference of provided implementation is that extrema are detected across octaves. An octave is like a Laplacian pyramid (or DoG) except that for each level of pyramid (for each size of image) we blur image with all blurring scales. Like the below image:

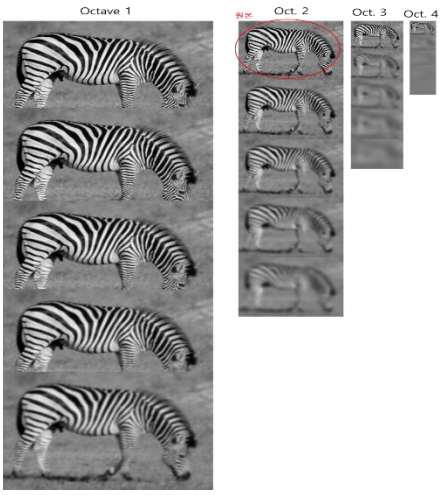


*Figure 1.1[1]- Octaves in SIFT*

Furthermore, in this implementation, key points are further refined by eliminating those that are likely to be unstable, either because they are selected nearby an image edge, or are found on image structures with low contrast. In my implementation, I only had a threshold for eliminating low-contrast (weak) extrema, I did not have edge rejection threshold.

For building the key point **descriptor**, a 16x16 neighborhood around the key point is taken. It is divided into 16 sub-blocks of 4x4 size. For each sub-block, an 8-bin orientation histogram is created. So, a total of (16x8)128 bin values are available. This 128-element vector is then normalized to unit length in order to enhance invariance to affine changes in illumination. This method leads to a more rotation-invariant feature

[1] Picture is taken from https://bskyvision.com/21

descriptor compared to my implementation that only had 36 orientation bins for all 16x16 pixels around each key point.

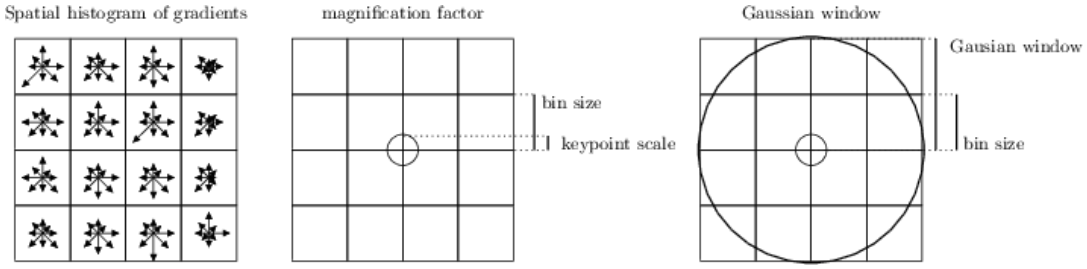


*Figure 1.2[2]- 128 orientation bins in key point descriptor*

**Comparing methods by observation:**

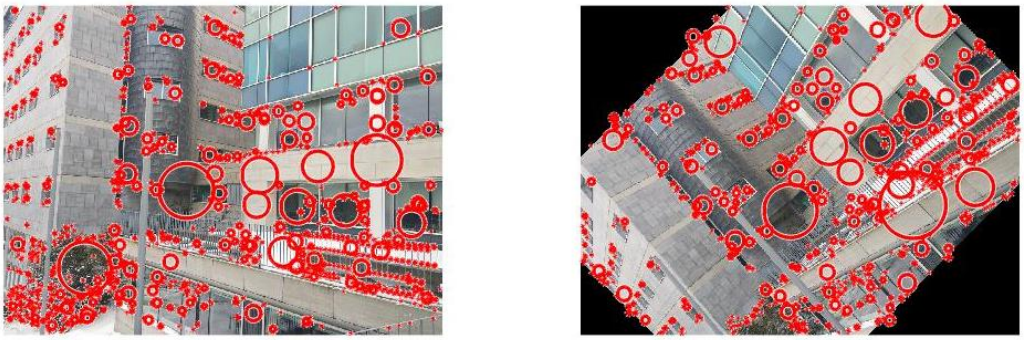

*Figure 1.3- The detected SIFT key points in the original image and a rotated version (around the center of image by 45 degrees) using the provided implementation.*

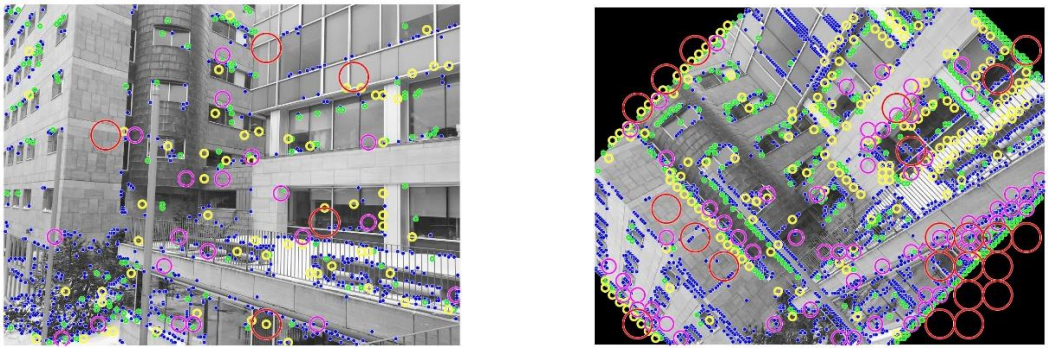

*Figure 1.4- The detected SIFT key points using my implementation in assignment two (threshold =3)*

It can be observed that in my implementation, there are so many false key points in the rotated image detected at the edges, especially in the boundaries of the image frame. However, in the sample implementation, these edge-like regions are not detected as SIFT points due to an edge rejection threshold being used.

[2] Picture is taken from http://www.vlfeat.org/api/sift.html

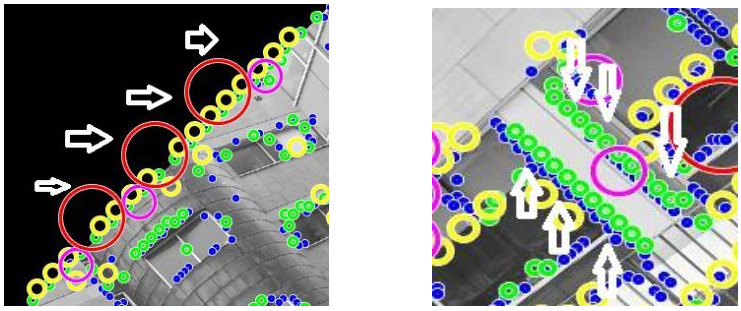

*Figure 1.5- Edge-like features detected as SIFT points (false detections)*

*Figure 1.6- Not detecting edge-like features in sample implementation*

Comparing SIFT descriptor (features) orientation histogram in the provided implementation vs. my implementation:

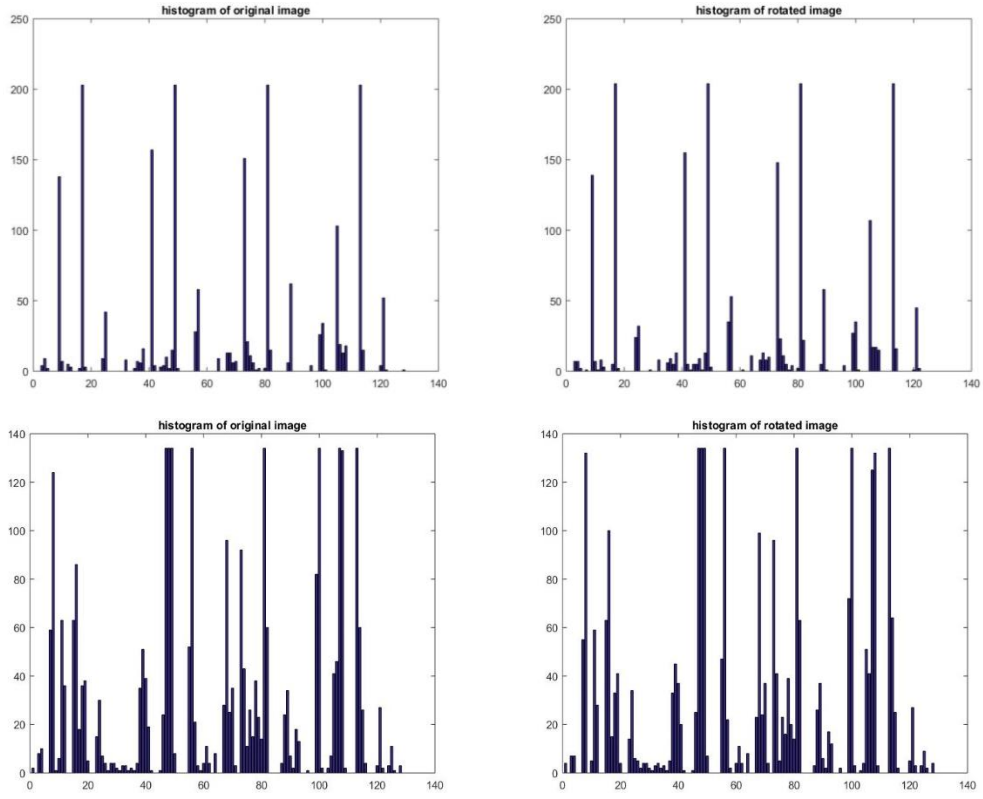


*Figure 1.7- Feature vectors of two sets of matched key points in the original and rotated image using* ***provided*** *implementation*

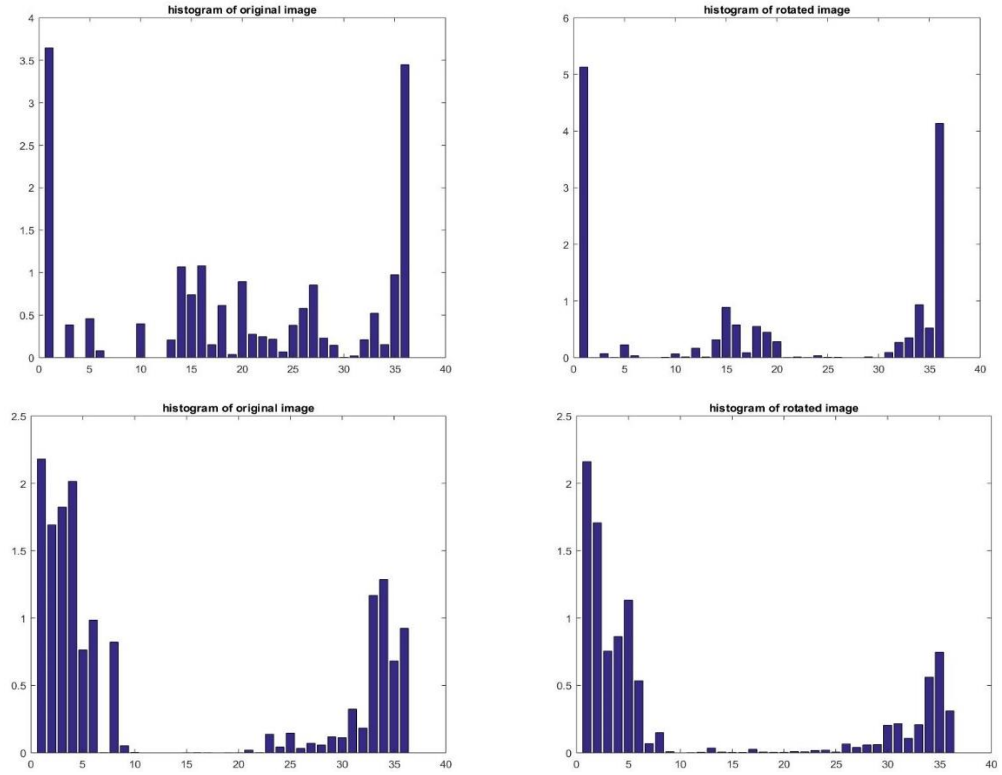


*Figure 1.7- Feature vectors of two sets of matched key points in the original and rotated image using **my** implementation*

From the above images, we can observe that in the provided implementation there is a great similarity between feature vectors of the matched key points in the original and rotated image, which means that key points (features) are rotation invariant. While in my implementation, this similarity is much lower. Additionally, the number of bins in the feature vector of the provided implementation is 128 while in my implementation is 36. Having greater number of bins considers more spatial details around the key point and helps in a more accurate detection of those key points in the rotated/scaled image.

**Question2:**

*(the results and pictures of this question were computed using the original size of the image (before resizing it to half of its size for panorama purposes.)*

After finding SIFT feature key points in two consecutive images, I used *matchFeatures* MATLAB function to find matching key points between the two images.

In *matchFeatures* function, two feature vectors match when the distance between them is less than the threshold set by *MatchThreshold*. *MatchThreshold* is a scalar percent value in the range (0,100]. The default Matching threshold of this function for non-binary features is 1.0. This value is too low for the images in our case and does not output any matched features. I changed the threshold to 3 and above to increase the acceptable distance between matching points. The funny thing is that even with a threshold of 100! We get all matchings correct!

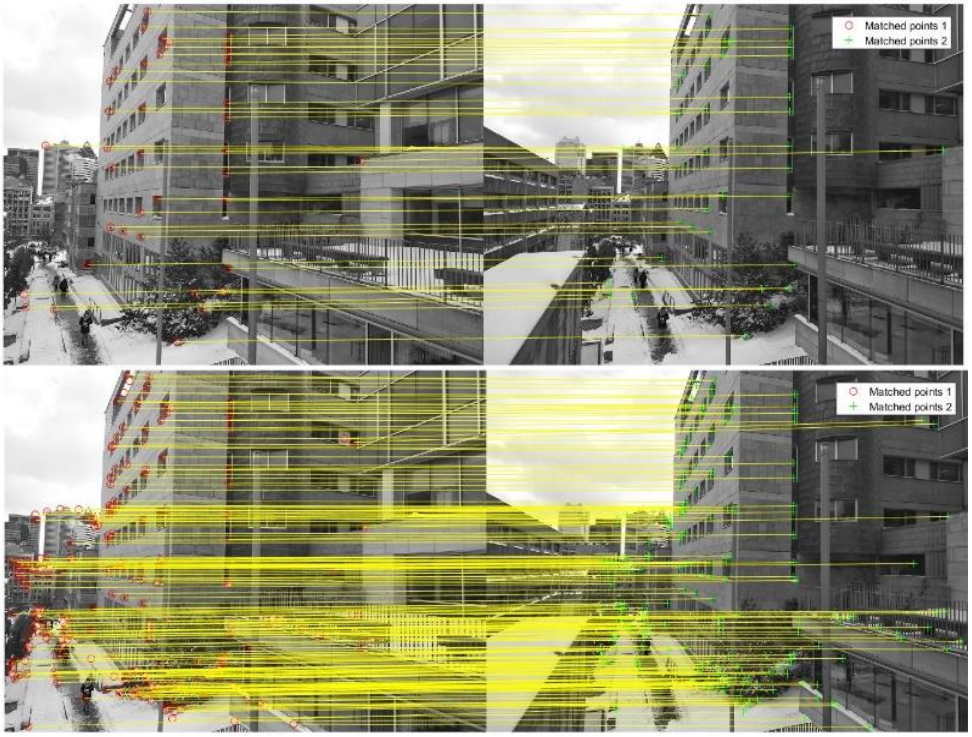


*Figure 2.1- The same parameters are used in the top and bottom images except the matching threshold. Threshold=3 for the top image (43 matchings) and Threshold=100 for the bottom image (261 matchings). Here the distance metric (which I will elaborate on, later) is the sum of absolute distances.*

There are two methods to match key points in this function: First Computing the pairwise distance between feature vectors which is named *'Exhaustive'*; Second, using an efficient approximate nearest neighbor search which is named *'Approximate'*. The second method is better for large feature sets. In our case both methods have the same result.

Another parameter is *'Unique'*. I set this value to *true* to return only unique matches between the two feature vectors.

Another parameter that I changed is *'MaxRatio'* which is a scalar ratio value in the range (0,1]. We can use the max ratio for rejecting ambiguous matches and increasing this value will return more matches. In case of having a high matching threshold (~100), increasing *'MaxRatio'* will lead to wrong matches. However, when we have a low matching threshold (~3) increasing *'MaxRatio'* to 1 will not cause mismatch.

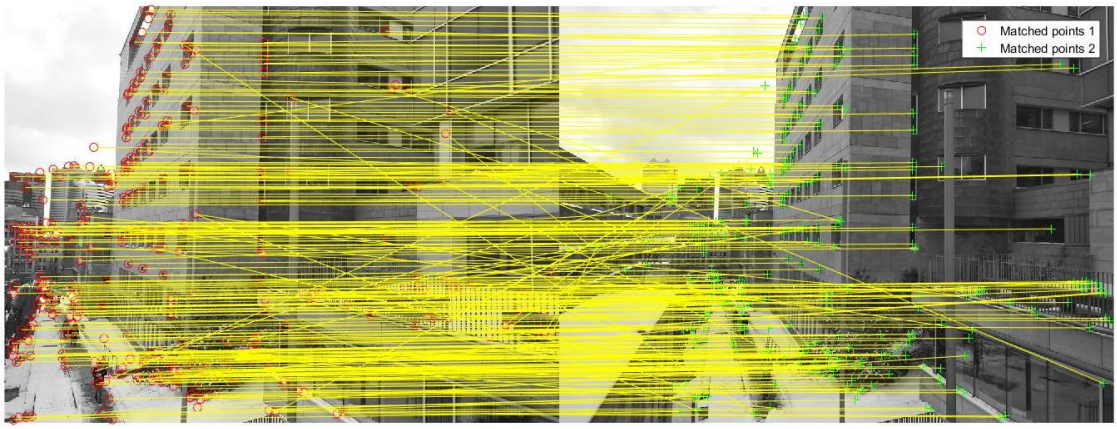


*Figure 2.2- The sloping lines are showing wrong matches between the two images. MatchingThreshold=100, Maxratio=1, Distance metric= sum of absolute distances.*

The Feature matching metric for non-binary features, can be defined as either 'SAD' which is the sum of absolute differences or 'SSD' which is the sum of squared differences. Changing the metric to 'SSD' will give us much more matchings (5 times the 'SAD'), as we are relaxing the condition on the matching distance. However, a wrong match might also occur as seen in below picture.

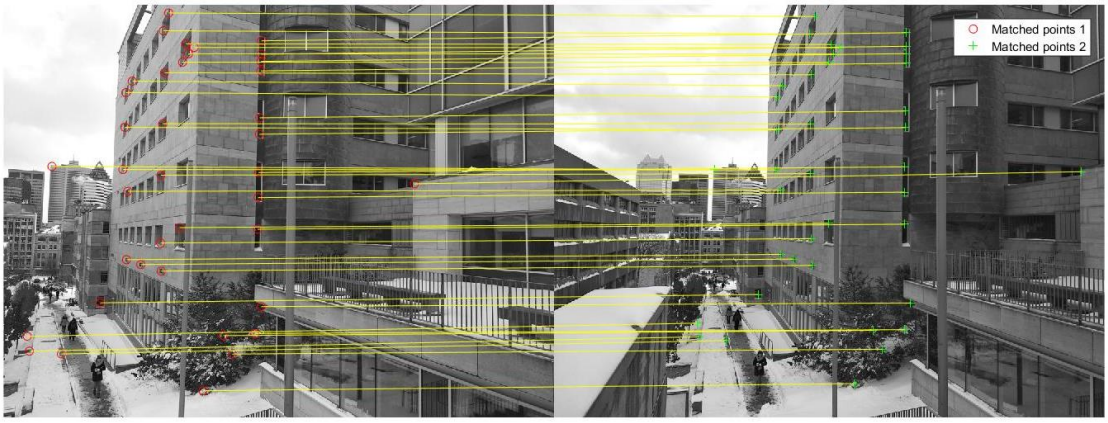


*Figure 2.3 -Matched feature points suing sum of absolute distance metric; threshold=3; 43 matches*


*Figure 2.4-Matched feature points suing sum of squared distance metric; threshold=3; 259 matches; the black circle shows the only mismatch.*

For the above experiments, horizontal images 1.png and 2.png were used. Now we will match features between the rest of images.

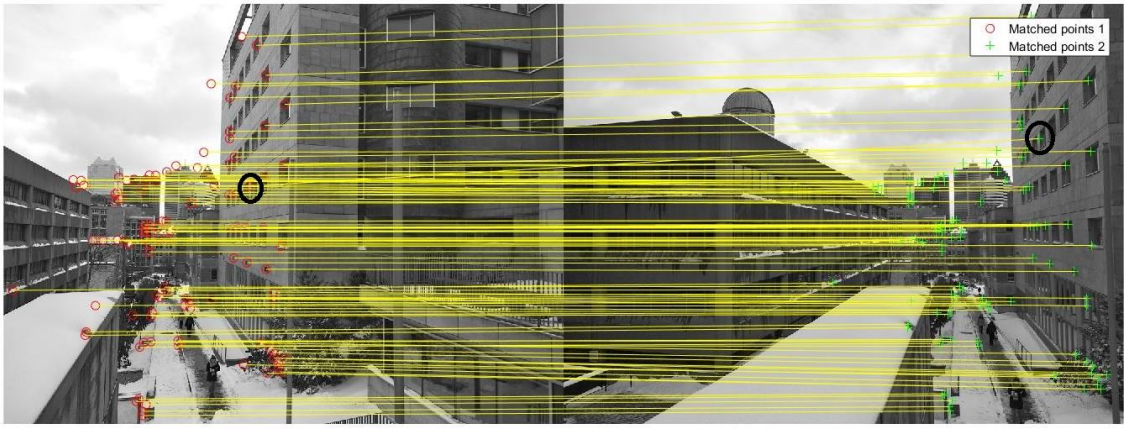


*Figure 2.5- One wrong matching occurred between horizontal images 2 and 3, which is shown by black circles. 124 matchings, MatchingThreshold=100, Distance metric='SAD', MaxRatio=0.6*

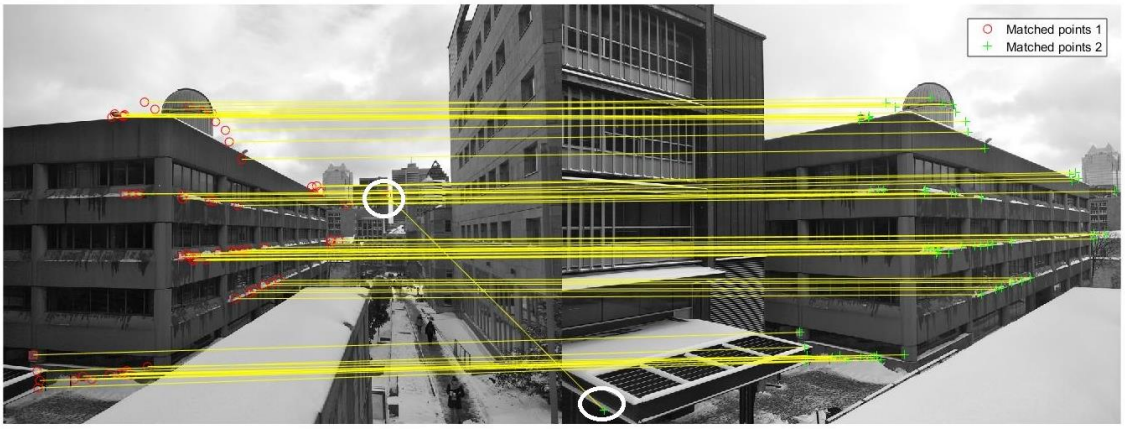


*Figure 2.6- One wrong matching occurred between horizontal images 3 and 4, which is shown by white circles. 68 matchings, MatchingThreshold=100, Distance metric='SAD', MaxRatio=0.6*

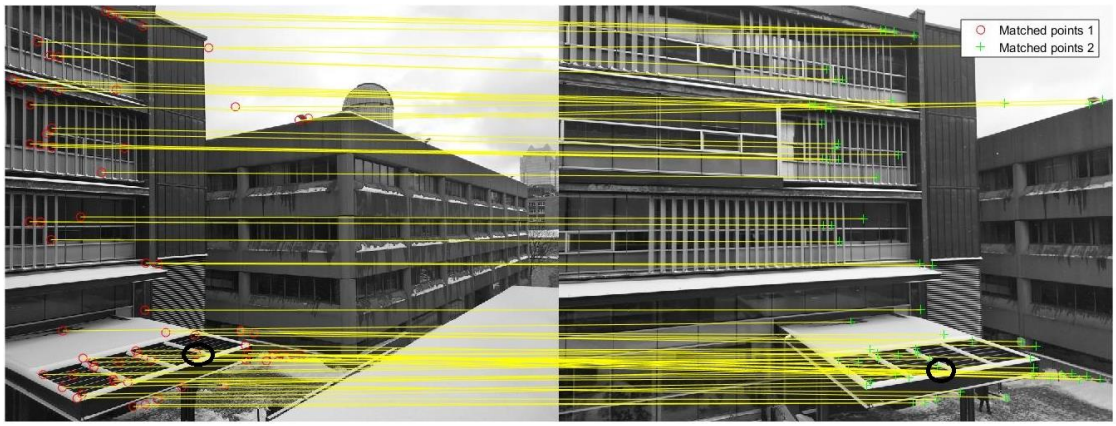


*Figure 2.7- One wrong matching occurred between horizontal images 4 and 5, which is shown by black circles. 75 matchings, MatchingThreshold=100, Distance metric='SAD', MaxRatio=0.6*

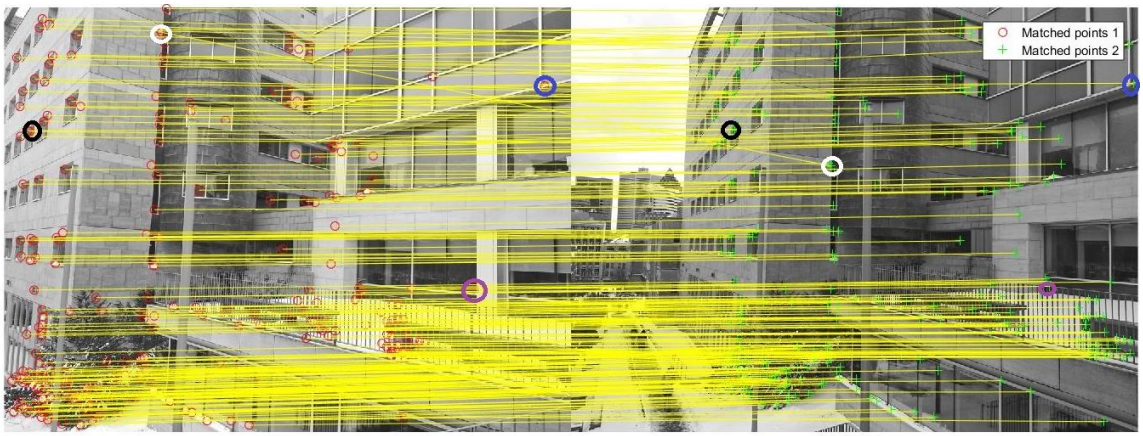


*Figure 2.8- Multiple wrong matching occurred between horizontal images 0 and 1, which is shown by bigger circles. 290 matchings, MatchingThreshold=100, Distance metric='SAD', MaxRatio=0.6*

In the horizontal sequence, more mismatch occurred between images 0 and 1, as image 0 includes some distortion and intensity changes. Most of these mismatches occur between similar structures in the image that have different locations, such as the corner of the windows that are repeated in other parts in the image.

For the sake of report volume, the images for vertical sequence matchings are included in the picture folder.

**Question 3:**
*(the results and pictures of this question were computed using the original size of the image (before resizing it to half of its size for panorama purposes.)*

To consider an underlying model for matching features between the two images, we can use RANSAC algorithm. This algorithm works properly when we have few outliers (incorrect matches), as in our case. According to Lecture 20, RANSAC algorithm steps are as follows:

1) We need to find some (SIFT) feature points in each of the two images. (we did this for Question 1)

2) For each feature point in one image, we should find a candidate corresponding feature point in the other image whose intensity neighborhood is similar. This step is the same as what we did for Question 2. Also, we can allow each key point to have several candidate matching points. However, as most of my matching points were correctly matched between the two images, I only considered one candidate for each key point.

3) We should randomly select 4 sets of matching points and fit a Homography that maps those points to each other. Then, we should calculate the consensus set for this homography, meaning that we should find how many of the other matching points can fit into this homography. Hence, we first calculate the matching points' coordinates for all remaining key points in one of the images, using the below formula:

$$\begin{bmatrix} w\tilde{x} \\ w\tilde{y} \\ w \end{bmatrix} = \mathbf{H}_i \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

And then, we calculate the distance between these homography-generated matching point $(\tilde{x}, \tilde{y})$ with real candidate matching points $(\acute{x}, \acute{y})$ in the other image. We can use Euclidean norm as below formula:

$$d = \sqrt{(\acute{x} - \tilde{x})^2 + (\acute{y} - \tilde{y})^2}$$

If this distance is lower than a specified threshold (=10 in my code) for a set of matching points, those points fit into the homography and are considered as consensus set of that homography. (These are our inliers)

4) We repeat step 3 until we find a homography whose consensus set is above a determined threshold. (in my code 95% of matching points) After finding this homography, we will use its consensus set to re-estimate the homography H using least squares. We do this final step, because H was previously calculated using four matching points between the two images, and if one of those points was noisy then the fit would be biased with that noise.

After doing the above steps, we have a homography whose consensus set is our inliers. Therefore, we can only consider inliers in our feature matching and remove those points which do not fit into this matrix and are considered as outliers.

To calculate the homography matrix in my code, I used the below formula (from lecture 20):

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -\tilde{x}_1 x_1 & -\tilde{x}_1 y_1 & -\tilde{x}_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -\tilde{y}_1 x_1 & -\tilde{y}_1 y_1 & -\tilde{y}_1 \\ : & : & : & : & : & : & : & : & : \\ x_N & y_N & 1 & 0 & 0 & 0 & -\tilde{x}_N x_N & -\tilde{x}_N y_N & -\tilde{x}_N \\ 0 & 0 & 0 & x_N & y_N & 1 & -\tilde{y}_N x_N & -\tilde{y}_N y_N & -\tilde{y}_N \end{bmatrix} \begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{21} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \\ H_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ : \\ 0 \\ 0 \end{bmatrix}$$

From the above equation, the least squares solution for H is obtained by taking the 2Nx9 data matrix A (the left matrix above) and find the eigenvector of $A^T$ A which has the smallest eigenvalue. In my code, homography matrix is calculated in function *Q3_Homography*.

Finally, I used the consensus set of the selected homography as my new matching points. After showing new matched features between the two images, we can observe that almost all of the incorrect matches which were outliers, are removed from our matching features.

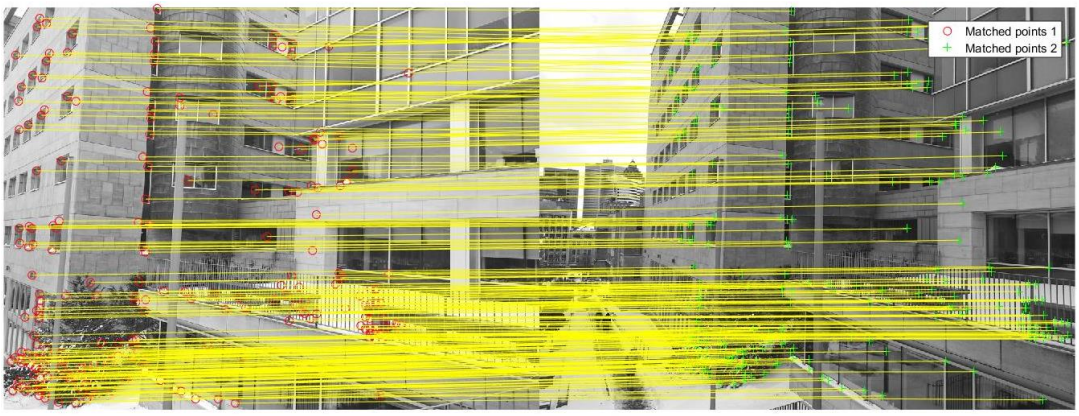


*Figure 3.1- Matched features between horizontal images 0 and 1 after applying RANSAC algorithm.*

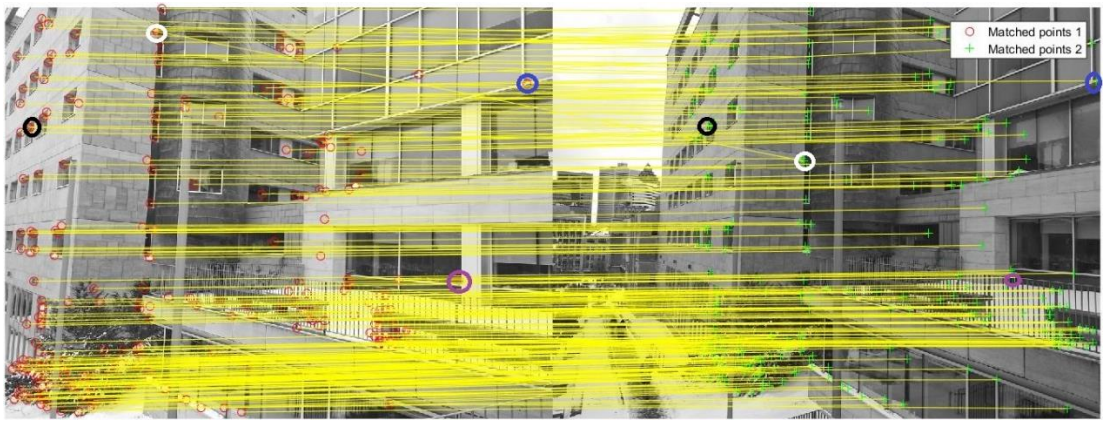


*Figure 3.2- Incorrect matches that were generated before using RANSAC algorithm are shown with larger circles. These points were outliers and were removed after using RANSAC. (compare this picture with Figure 3.1)*

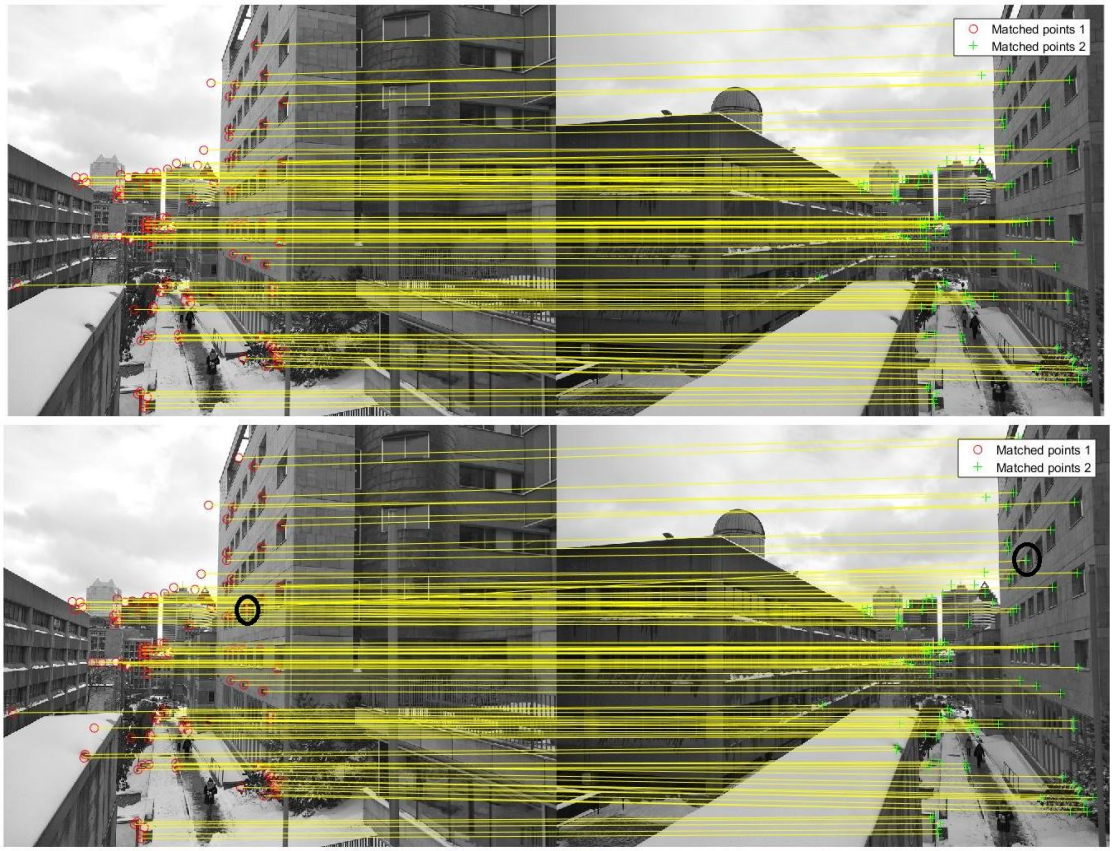


*Figure 3.3- Comparing Matched features between horizontal images 2 and 3 before (lower image) and after (upper image) applying RANSAC algorithm. Outlier is removed.*

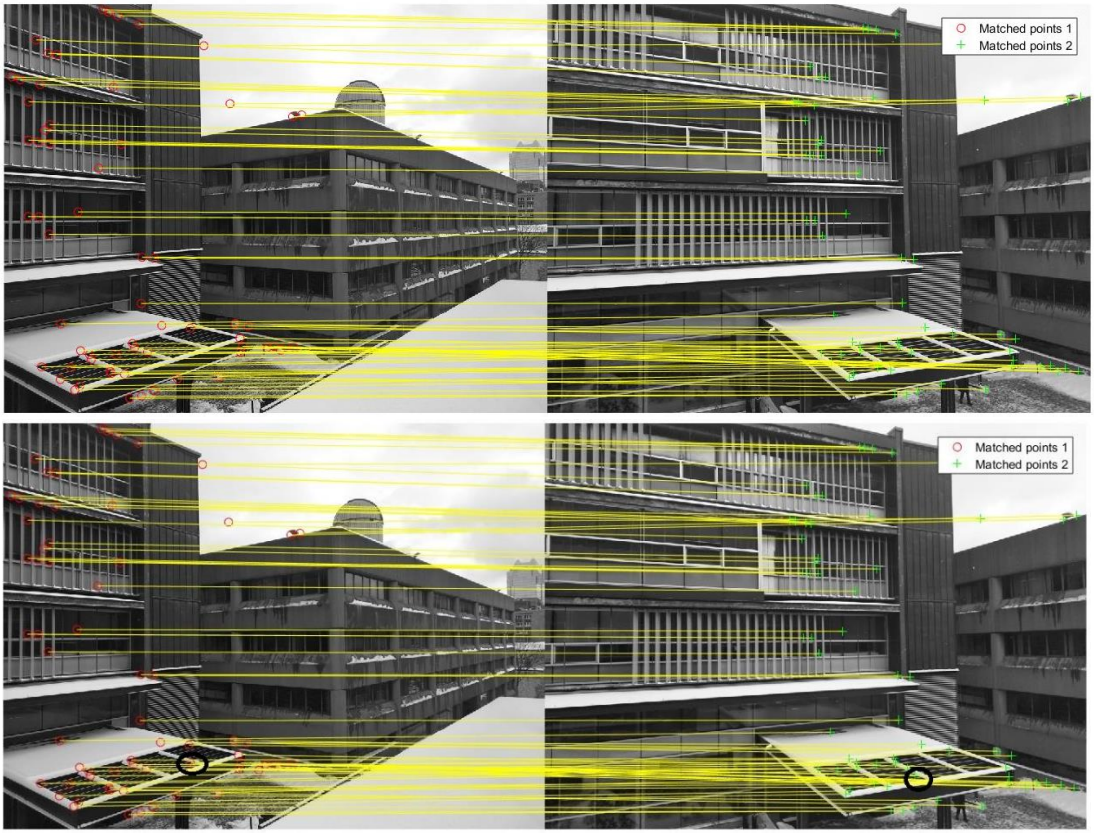


*Figure 3.4- Comparing Matched features between horizontal images 4 and 5 before (lower image) and after (upper image) applying RANSAC algorithm. Outlier is removed.*

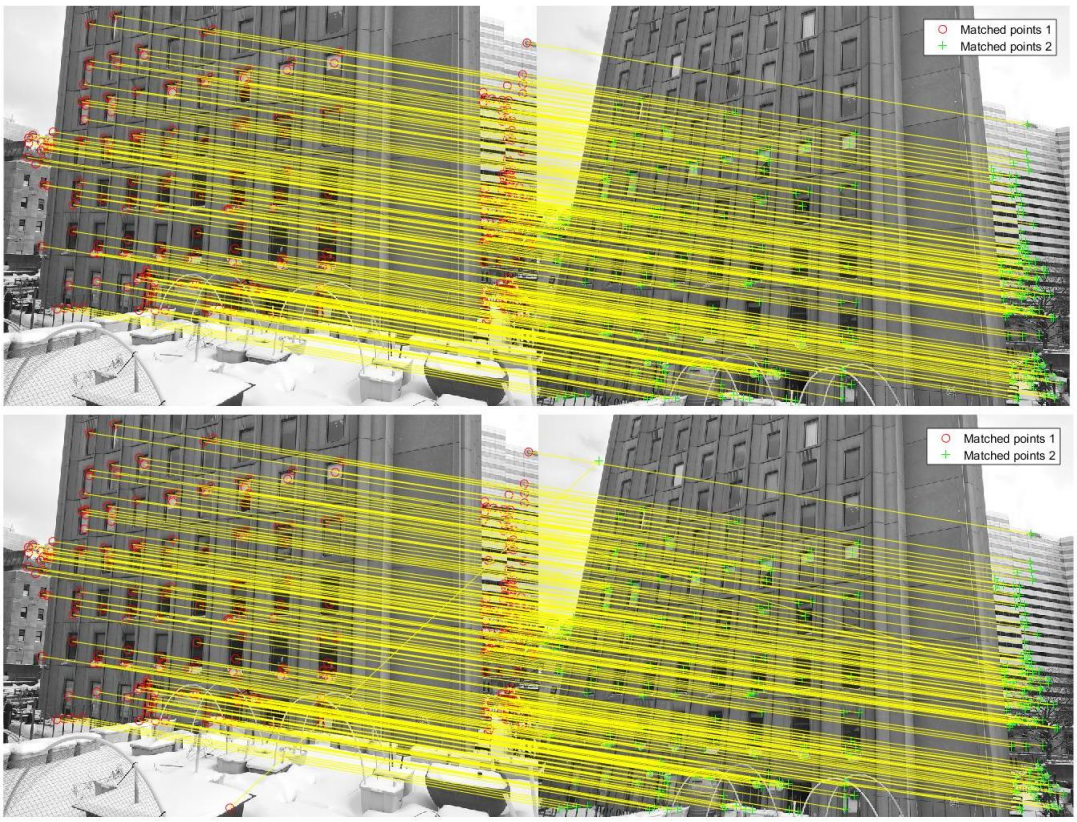


*Figure 3.5- Comparing Matched features between vertical images 1 and 2 before (lower image) and after (upper image) applying RANSAC algorithm. Outlier is removed.*

For the comparisons between other images in horizontal and vertical sequence, refer to the *pics* folder.

**Question 4:**

To stitch two consecutive images based on their matched features, I used their homography matrix. First, homography matrices should be changed to 2-D projective geometric transformations using `projective2d` Matlab built-in function. This conversion is necessary if we want to warp the image using `imwarp` Matlab function. To convert homography to projective transformations, I used this line of code: `tforms{m} = projective2d(inv(H'))`. By experiment, I realized the homography matrix should be transposed and inverted before being converted to a transformation matrix, to output a correct result. Then I warped the image with respect to its transformation using `imwarp`. The output size of `imwarp` was set to the reference image (which in case of 2 by 2 overlays is the first image). To blend the transformed image with its previous (reference) image, we can use `AlphaBlender` system object or `imfuse` function in matlab.

*Figure 4.1- Overlaying horizontal sequence images two-by-two using imfuse function in matlab. The overlays are accurate. (even between images 0 and 1)*

*Figure 4.2- Overlaying horizontal sequence images two-by-two using AlphaBlender function with 'binary mask' to overwrite the pixel values of one image with another in the shared parts. The overlays are accurate; however due to overwriting, some artifacts are generated as the one shown in black circle in above. (If we don't use masking, the results are not fully blended)*

*Figure 4.3- overlaying two horizontal images using AlphaBlender, without binary masking. The images' brightness is not naturally blended. (there might be some solutions for this, but I did not find that!)*

Now, to build a panorama we should stitch multiple images in a sequence. When we stitch the second image to the first image, the second image is warped (transformed) to overlay the first image. Now, when we want to stitch the third image to the blend of the first and second image, we can not only consider the transformation of third image with respect to the second image, since the second image is now transformed. Therefore, we should transform the third image according to the multiplication of the second image's transformation and its own transformation. In this regard, we are considering the first image as our reference, and we transform all other stitches according to the amount of needed transformation from the reference image. We do this process consecutively to stitch other images to the blend of stitched images. This line of code, computers the transformations for each image, considering the first image as the reference:

```
tforms{m}.T = tforms{m}.T * tforms{m-1}.T;
```

*Figure 4.4- overlaying four horizontal images using AlphaBlender, with binary masking. The images are blended according to their transformation from the first image, as their reference.*

Now to build the panorama, we need to change our reference image, to be the image in the center of the sequence. This way we can minimize the propagated transformation errors in the panoramas. In each transformation minor errors occur, and as we go further away from the reference image, errors get bigger and bigger and cause mismatch in the panorama. This also results in a better display in the final panorama. Another solution is to consider the transformed image, with the middle width and length to be in the center of panorama. Meaning that we sort the width and length ranges of all transformed images, and we sort them, and choose the image with the middle (mean) values as our reference image. (sometimes this image is the same as the middle image in sequence, but not always.) This solution leads to a better size adjustment of other transformed images. For the vertical sequence, I used this method to choose the reference image. In horizontal sequence, by experiment, I chose the fourth image as the reference.

After choosing the reference image, we should change the transformations in a way to make the transformation of the reference equal to identity matrix. This is done by multiplying all other transformations by inverse of reference's transformation matrix. Further, we should define an empty matrix (canvas) to attach our blended images to and display the panorama. Therefore, we compute the output limits on x and y (width and length of image) and build an empty panorama made of zeros. Then, we need to blend our images. We can use `imwarp` function to transform our image, we also build a mask for our transformation to be able to overwrite shared pixels of the stitched images on to one another. Also, we should create a 2-D spatial reference object defining the size of the panorama using imref2d function. Finally, we use the step function in `AlphaBlender` to Overlay the warped images onto the canvas (panorama). (Besides, to deal with low memory issues, I resized the images to half of their original size: 350x500)

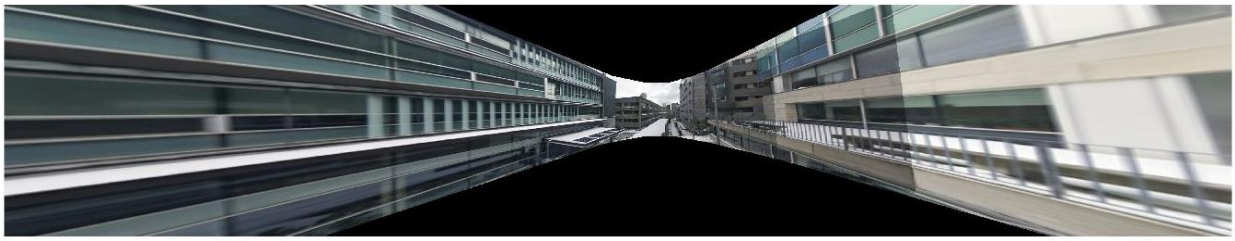

*Figure 4.5- The horizontal panorama. The inconsistency between the size of center images (around the reference) with further images (far from the reference image), has caused a deformed shape. (due to a big size of further stitches a memory error occurs; I limited the boundary of panorama to be able to plot it.) (output limits x =[-4000 4000], y = [-500 1000])*

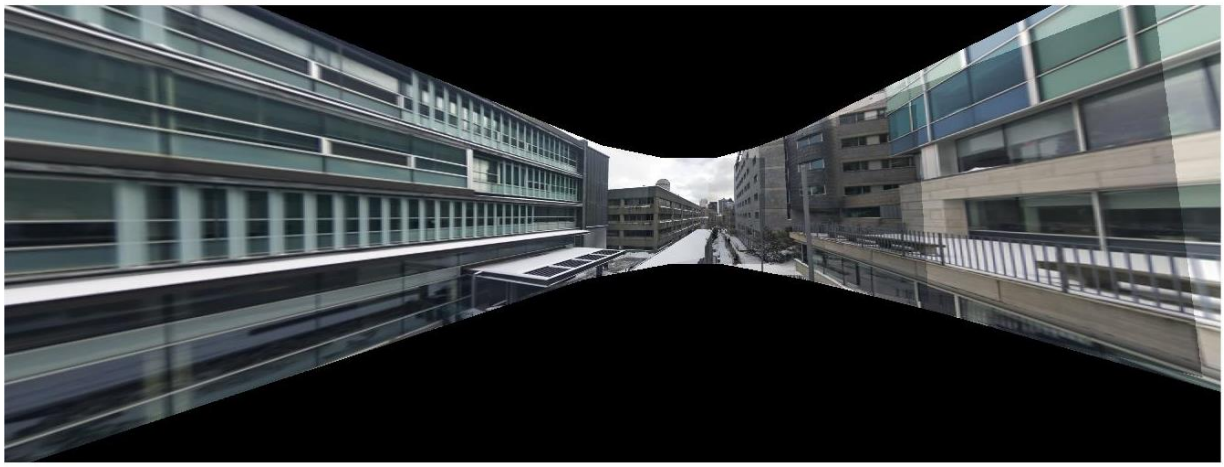

*Figure 4.6- The horizontal panorama. A closer view. (smaller output limits x =[-2000 2000])*

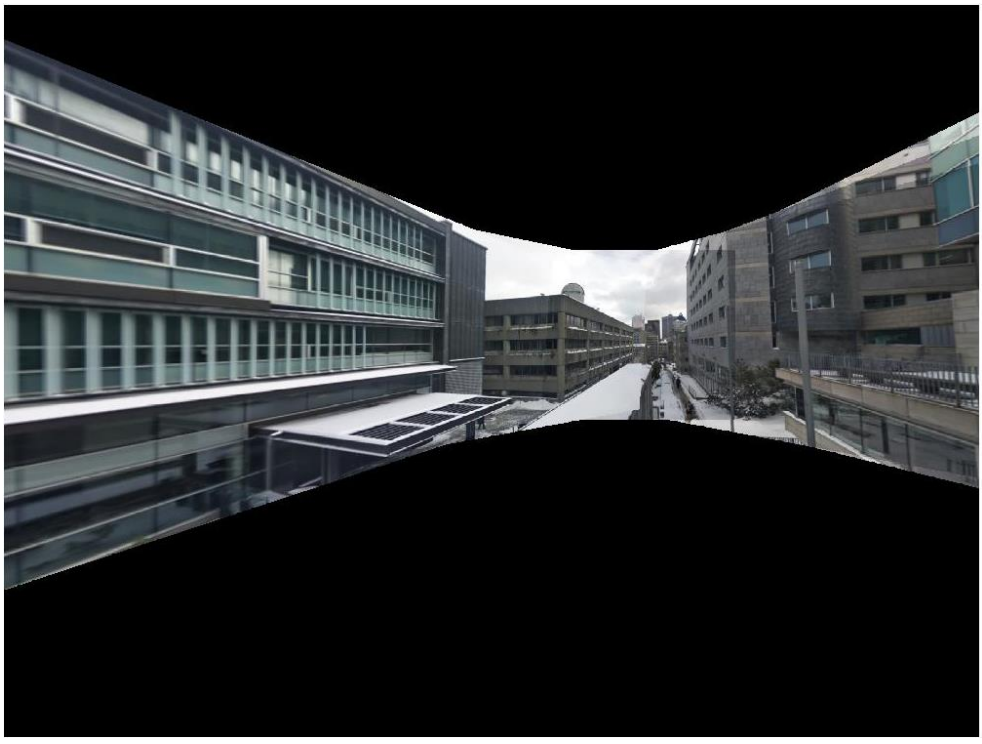

*Figure 4.7- The horizontal panorama. A closer view. (smaller output limits x=[-1000 1000])*

*Figure 4.8- The vertical panorama. Choosing the image whose transformed version has the mean width (x) as the reference image. (the third image)*

*Figure 4.9- The vertical panorama. Choosing the image whose transformed version has the mean height (y) as the reference image. (the fourth image)*

**Question 5:** For this question I used below sequence of images:

*Figure 5.1- A sequence of images taken by me*

*Figure 5.2- Panorama view of the sequence of images.*

Some mismatches have occurred in left side of the panorama. These mismatches might be due to the translation of camera that is not considered in our calculations. Also, the number of extracted sift feature key points is less in my images compared to the provided ones, as there might be less corner-like features (extrema) in my images. Also, here the rotation seems to be greater between each two consecutive images (compared to the provided images) which leads to greater scene change and less matching features between them.

*Figure 5.3- fewer key points are found in images taken by me.*

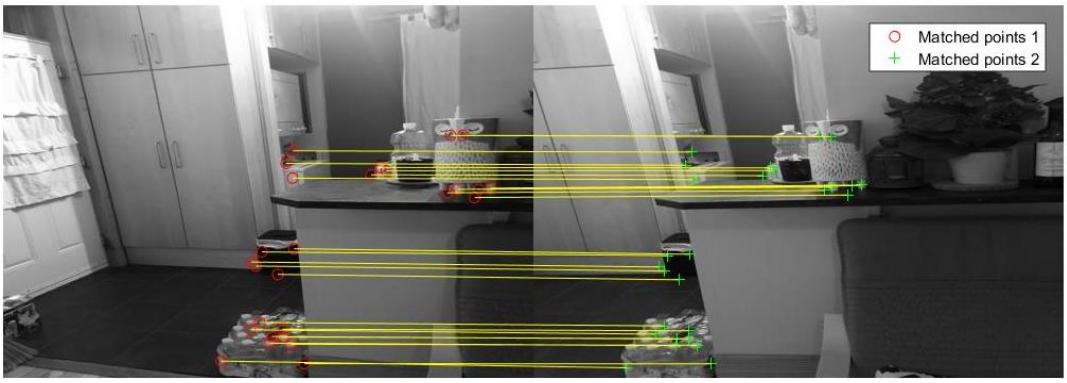


*Figure 5.4- fewer matched key points are found.*

**Question 6:**

When we overlay images in panoramas, moving objects or people can cause “ghosting”, mis-registration can cause blurring, and exposure differences can cause visible seams. There are several ways to reduce the visible seams in mosaics:

– Feathering (weighted average)

– Optimal seam selection

– Laplacian Blending

– Gradient-Domain Image Stitching

– Regions of Differences (ROD)

Feathering (weighted average):
The simplest way to create a final composite is to simply take an average value at each pixel:

$$C(x) = \sum_k w_k(x)\tilde{I}_k(x) \Big/ \sum_k w_k(x)$$

where $\tilde{I}_k(x)$ are the warped (re-sampled) images and $w_k(x)$ is 1 at valid pixels and 0 elsewhere. Simple averaging usually does not work very well, since exposure differences, misregistrations, and scene movement are all very visible. If rapidly moving objects are the only problem, taking a median filter (which is a kind of pixel selection operator) can often be used to remove them (using the function `medfilt2` in MATLAB). Conversely, center-weighting and minimum likelihood selection can sometimes be used to retain multiple copies of a moving object. A better approach to averaging is to weight pixels near the center of the image more heavily and to down-weight pixels near the edges. When an image has some cutout regions, down-weighting pixels near the edges of both cutouts and the image is preferable. This can be done by computing a distance map (or grassfire transform). Weighted averaging with a distance map is often called feathering. and does a reasonable job of blending over exposure differences. However, blurring and ghosting can still be problems.

*Figure 6.1- The people shown, are in different locations in one frame of the picture compared to the other, resulting in "ghosting".*

Optimal seam selection:

Another approach is to place the seams in regions where the images agree, so that transitions from one source to another are not visible. In this way, the algorithm avoids "cutting through" moving objects where a seam would look unnatural. For a pair of images, this process can be formulated as a simple dynamic program starting from one edge of the overlap region and ending at the other. When multiple images are being composited, the dynamic program idea does not readily generalize.

Regions of Differences (ROD):

To overcome this problem, Uyttendaele, Eden, and Szeliski (2001) observed that for well-registered images, moving objects produce the most visible artifacts, namely translucent looking ghosts. Their system therefore decides which objects to keep and which ones to erase. First, the algorithm compares all overlapping input image pairs to determine regions of difference (RODs) where the images disagree. Next, a graph is constructed with the RODs as vertices and edges representing ROD pairs that overlap in the final composite. Since the presence of an edge indicates an area of disagreement, vertices (regions) must be removed from the final composite until no edge spans a pair of remaining vertices. The smallest such set can be computed using a vertex cover algorithm. Since several such covers may exist, a weighted vertex cover is used instead, where the vertex weights are computed by summing the feather weights in the ROD (Uyttendaele, Eden, and Szeliski 2001). The algorithm therefore prefers removing regions that are near the edge of the image, which reduces the likelihood that partially visible objects will appear in the final composite. Once the desired excess regions of difference have been removed, the final composite can be created by feathering.

Blending:

Once the seams between images have been determined and unwanted objects removed, we still need to blend the images to compensate for exposure differences and other misalignments. The spatially varying weighting (feathering) previously discussed can often be used to accomplish this. However, it is difficult in practice to achieve a pleasing balance between smoothing out low-frequency exposure variations and retaining sharp enough transitions to prevent blurring (although using a high exponent in feathering can help).

Laplacian pyramid blending:

An attractive solution to this problem is the Laplacian pyramid blending technique developed by Burt and Adelson (1983b). Instead of using a single transition width, a frequency-adaptive width is used by creating a band-pass (Laplacian) pyramid and making the transition widths within each level a function of the level, i.e., the same width in pixels. In practice, a small number of levels, i.e., as few as two (Brown and Lowe 2007), may be adequate to compensate for differences in exposure.

Gradient domain blending:

An alternative approach to multi-band image blending is to perform the operations in the gradient domain. Reconstructing images from their gradient fields has a long history in computer vision, starting originally with work in brightness constancy (Horn 1974), shape from shading (Horn and Brooks 1989), and photometric stereo (Woodham 1981). More recently, related ideas have been used for reconstructing images from their edges (Elder and Goldberg 2001), removing shadows from images (Weiss 2001), separating reflections from a single image (Levin, Zomet, and Weiss 2004; Levin and Weiss 2007), and tone mapping high dynamic range images by reducing the magnitude of image edges (gradients) (Fattal, Lischinski, and Werman 2002). P´erez, Gangnet, and Blake (2003) show how gradient domain reconstruction can be used to do seamless object insertion in image editing applications. Rather than copying pixels, the gradients of the new image fragment are copied instead. The actual pixel values for the copied area are then computed by solving a Poisson equation that locally matches the gradients while obeying the fixed Dirichlet (exact matching) conditions at the seam boundary. P´erez, Gangnet, and Blake (2003) show that this is equivalent to computing an additive membrane interpolant of the mismatch between the source and destination images along the boundary. In earlier work, Peleg (1981) also proposed adding a smooth function to enforce consistency along the seam curve.

Exposure compensation:

Pyramid and gradient domain blending can do a good job of compensating for moderate amounts of exposure differences between images. However, when the exposure differences become large, alternative approaches may be necessary. Uyttendaele, Eden, and Szeliski (2001) iteratively estimate a local correction between each source image and a blended composite. First, a block-based quadratic transfer function is fit between each source image and an initial feathered composite. Next, transfer functions are averaged with their neighbors to get a smoother mapping and per-pixel transfer functions are computed by splining (interpolating) between neighboring block values. Once each source image has been smoothly adjusted, a new feathered composite is computed, and the process is repeated (typically three times). The results shown by Uyttendaele, Eden, and Szeliski (2001) demonstrate that this does a better job of exposure compensation than simple feathering and can handle local variations in exposure due to effects such as lens vignetting. Ultimately, however, the most principled way to deal with exposure differences is to stitch images in the radiance domain, i.e., to convert each image into a radiance image using its exposure value and then create a stitched, high dynamic range image, as discussed in Section 10.2 (Eden, Uyttendaele, and Szeliski 2006).[3]

[3] Computer Vision: Algorithms and Applications, Richard Szeliski