Problem 1: Short Questions (14 points)

Part A (3 points):

On the domain $-\pi < \omega_x \leq \pi$, $-\pi < \omega_y \leq \pi$, the discrete-space Fourier transform of the input is only non-zero for $(\omega_x, \omega_y) = (0, 0)$ and $(\omega_x, \omega_y) = (\pi, \pi)$, so only the values of the frequency response for these two values of $(\omega_x, \omega_y)$ are needed in order to fully define the output of this linear shift-invariant filter.

Note: Since $F(e^{j\omega_x}, e^{j\omega_y})$ is $2\pi$-periodic in $(\omega_x, \omega_y)$, other answers translated by an integer multiple of $2\pi$ for $\omega_x$ or $\omega_y$ are equally valid.

Part B (2 points):

In the chromaticity diagram, mixing illuminants corresponds to taking a linear combination of the corresponding 2D points, so the user can only match colors that lie on the line between the points corresponding to the monochromatic lights 580 nm (noted A on the diagram) and $\lambda$ (dashed line). In order to be able match the light W, the only possible value of $\lambda$ can be read by taking the intersection between the line that goes through A and W and the border of the gamut. This gives $\lambda = 480$ nm.
Part C (2 points):

For a deuteranope, any line that intersects with the confusion line for W corresponds to a mixture that can be matched with the reference white light, for example the dashed lines shown on the diagram. So, all monochromatic lights with a wavelength lower than the one at the N point (solid line) are valid. In other word, the answer is $\lambda < 500$ nm.

Part D (3 points):

The trick is to decompose the kernel into 2 parts.

$$\begin{bmatrix}-1 & -1 & -1 \\ -1 & [9] & -1 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix}0 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix}1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix}0 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix}1 \\ 1 \\ 1 \end{bmatrix} \times \begin{bmatrix}1 & 1 & 1 \end{bmatrix}$$

(A) required 1 multiplication, (B) requires 2 additions, and (C) requires 2 additions.

Finally, we need 1 subtraction to combine the results. Thus, in total, the filter would require 4 additions, 1 subtraction, and 1 multiplication.
Part E (4 points):

The filter presented in this problem may be familiar to some students as bilateral filter. A standard Gaussian filter blurs the entire image uniformly thereby degrading the edges in an image. A bilateral filter is a modification of Gaussian filter and is designed to blur the input image while preserving prominent edges in the image.

(1) Comparing the output images resulting from bilateral and Gaussian filtering respectively, we notice that while both filters are effective at denoising the input image, the bilateral filter retains the salient information in the input image by preserving the edges. Bilateral filter is thus a better denoiser than a simple Gaussian blur.

![Noisy input image](image1) ![Output bilateral filter](image2) ![Output of Gaussian filter](image3)

The functionality of the bilateral filter can also be deduced by looking at the equation that gives us the filter weights.

\[
 w[i, j, k, l] = \exp \left( -\frac{(i-k)^2 + (j-l)^2}{2\sigma_i^2} - \frac{(I[i,j] - I[k,l])^2}{2\sigma_r^2} \right)
\]

The first term inside the exponent is same as that for a standard Gaussian filter and corresponds to a normal decay of weights as a function of spatial Euclidean distance. The second term is what makes it a bilateral filter, and corresponds to a normal decay of filter weights as a function of pixel intensity differences. As a result, at edges in an image, the drastic change in image intensity values suppresses the weights of the filter for those pixels and prevents the filter from blurring across edges.

(2) Bilateral filter is non-linear. This can be deduced by observing that multiplying the input Image \( I \) by a constant \( \alpha \) does not result in an output image that is multiplied by the same factor. That is,

\[
 f_{\text{bilateral}}(\alpha I) \neq \alpha f_{\text{bilateral}}(I)
\]

(3) The filter is shift-invariant. Shifting the input image by some \([x_0, y_0]\) results in an output that is shifted by the same amount.

\[
 f_{\text{bilateral}}(I[x - x_0][y - y_0]) = f_{\text{bilateral}}(I)[x - x_0][y - y_0]
\]

(4) The filter is not separable since filter weights \( w[i, j, k, l] \) cannot be written as an outer product of a horizontal filter depending only on \( i \) and \( k \), and a vertical filter depending only on \( j \) and \( l \).
The spatial component of the filter is separable

\[ \exp \left( -\frac{(i-k)^2 + (j-l)^2}{2\sigma_s^2} \right) = \exp \left( -\frac{(i-k)^2}{2\sigma_s^2} \right) \exp \left( -\frac{(j-l)^2}{2\sigma_s^2} \right) \]

However, the same cannot be done for the second term in the exponent, making the bilateral filter non-separable.

Thus, while bilateral filtering results in better quality denoising output, the non-linear and non-separable nature of the filter also makes it computationally more taxing than Gaussian filtering.
Problem 2: Reverse Engineering (16 points)

Part A (8 points):
In the processed zoneplate image (zoneplate_IPA1.tif), we observe that the central bull’s eye pattern from the original zoneplate has been replicated to have 4 replicas horizontally and 2 vertically. The replicas signal that the zoneplate was downsampled by a factor of 4 horizontally and by a factor of 2 vertically.

Additionally, we observe that both the zoneplate_IPA1.tif and the comic_noisy_IPA1.tif have the same size as the respective original, unprocessed images. This indicates that the images were subsequently upsampled to match the original size. We also notice that comic_noisy_IPA1.tif looks pixelated. So the upampling was probably achieved using nearest neighbor interpolation, also known “pixel-repeat”.

Part B (8 points):
The most noticeable difference in the processed images is perhaps that zoneplate_IPA2.tif looks largely white. Additionally, we notice that some of the pepper noise or the black specs from comic_noisy.tif have vanished after the image was processed with IPA2. Both of these indicate that IPA2 involves morphological dilation. We also notice that some of the pepper noisy remains even after processing. This probably corresponds to the bigger black specs in the original comic image that did not get removed after performing dilation. We can use this information to decide the size of the structuring element used for dilation. Finally, zoneplate_IPA2 looks symmetric horizontally and vertically. So the structuring element was quite likely a square.

Let’s try a 3x3 and a 5x5 dilation. The results on comic_noisy.tif are shown below.

![3x3 dilation](image1)

![5x5 dilation](image2)

We see that 5x5 dilation removes all the pepper noise. So the correct structuring element is a square of size 3x3.
%% Part A %%
zp = imread('zoneplate.tif');
im = imread('comic_noisy.tif');

zp_A = zp(1:2:end, 1:4:end);
zp_A = imresize(zp_A, [res, res], 'nearest');
im_A = im(1:2:end, 1:4:end);
im_A = imresize(im_A, [h, w], 'nearest');

imshow(zp_A);
imwrite(im_A);

%% Part B %%
zp = imread('zoneplate.tif');
im = imread('comic_noisy.tif');

im_B = imdilate(im, ones(3));
zp_B = imdilate(zp, ones(3));

imwrite(zp_B);
imwrite(im_B);
Problem 3: Morphological Image Processing (20 points)

In all figures, the red circle marks the origin of the structuring element.

Part A (5 points):

We apply a binary dilation with the following structuring element.

![Structuring Element](image)

The structuring element is a lower-left quarter-disk of radius $s$ centered at the point $(s, 0)$.

The set-theoretic definition of dilation on Slide 14 of Lecture 7 can be used to verify the result:

$$ G = \bigcup_{(p_x,p_y) \in \Pi_{xy}} F_{+}(p_x,p_y) $$

where $F$ is the original shape, $G$ is the morphologically processed shaped, $\Pi_{xy}$ is the structuring element, and $F_{+}(p_x,p_y)$ means $F$ is shifted by $(p_x, p_y)$.

Part B (5 points):

We apply a binary erosion with the following structuring element.

![Structuring Element](image)
The structuring element is made of two impulses, one at \((0, 0)\) and another one at \((s, s)\).

The result can be inferred by using the set-theoretic definition of erosion given on Slide 15 of Lecture 7:

\[
G = \bigcap_{(p_x, p_y) \in \Pi_{xy}} F_{+}(p_x, p_y)
\]

In the case of this structuring element, there are only two elements in this intersection. The “translated” versions of \(F\) that we need to intersect are the two following shapes. (We show the outline of \(F\) in red as usual).

\[
F_{+}(p_x, p_y), \text{ where } (p_x, p_y) = (0, 0)
\]

\[
F_{+}(p_x, p_y), \text{ where } (p_x, p_y) = (-s, -s)
\]

Taking the intersection of these shapes gives the result we want.

**Part C (5 points):**

We apply a binary erosion with the following structuring element.

The structuring element is a line segment that joins the points \((0, 0)\) and at \((s, s)\).

Using the set-theoretic definition of erosion is again helpful here. If we translate our original shape \(F\) from position \((0, 0)\) to \((-s, -s)\) and take the intersection of all shapes, we get the correct result.
Part D (5 points):

We apply a binary erosion with the following structuring element.

The structuring element is made up of a line segment that joins the points \((0, 0)\) and at \((s, s)\) and a lower-left quarter-circle of radius \(s\) centered at the point \((s, 0)\). We show the shape as filled in our figure, but it actually does not matter if it is hollow.

Once more, we use the set-theoretic definition of erosion. We notice that the shape is included in the shape from part C, so as a starting point we can use the structuring element from C to get that shape. We then need to erode away the circular lower-left edge of that shape. This is done by adding additional points to the structuring element that form a quarter-circle as shown.
Problem 4: QR code reader (30 points)

Part A (6 points):

One efficient way to coarsely localize the QR code is to apply a 2D Gaussian smoothing kernel with a large standard deviation so that we only keep the coarse details of the image. A good choice is a standard deviation of 100. This transforms the original image as follows:

We then take the minimum across the image and this gives us a good estimate of the approximate location of the QR code on the image. Other valid methods are accepted, as long as the QR code is fully included in the cropped region.

Applying the 350x350 cropping to the first test image gives:

Part B (6 points):

Looking closely at the image shows that there is three types of noise: some salt and pepper noise, some Gaussian noise and the sparse larger dark specks. It is hard to find a way to remove the dark specks without losing information about the pixels in the QR code since they are roughly the same size, so for now we do not attempt to remove them. However, the salt and pepper noise can be removed easily with a median filter. It is recommended to use a small window (e.g. 3x3) so as to minimize the loss of detail in the QR code. Finally, binarizing the image will get rid of most of the Gaussian noise.
Applying the 3x3 median filter removes most of the salt and pepper noise:

After binarization (we use Otsu’s method), the image is much cleaner.

This gives the following edge map. The dark specks in the background add spurious pixels, which justifies why it was necessary to crop the image in part A.

Part C (6 points):

Since there is no shearing or perspective transformation, the grid of the QR code is made up of lines with two orthogonal orientations. Reading from left to right, we call them ascending lines (red lines in the figure below) and descending lines (green lines). Without loss of generality, we can always define $\alpha$ the orientation angle of the grid as the angle between the $x$ axis and the descending lines. By definition, this angle is always in the interval $[0^\circ, 90^\circ]$. 
In the Hough space, using the default parameterization, the ascending lines correspond to the angle $\theta = \alpha$ (see figure) and the descending lines correspond to the angle $\theta = \alpha - 90^\circ$. So in the ideal case, the Hough transform should contain sharp peaks at $\alpha - 90^\circ$ and $\alpha$.

For the implementation, we extract the top 40 peaks in the transform and set the threshold parameter to 0. This will ensure that we extract the 4 outer lines of the code, which will be useful for the next part. Plotting the histogram of the angles of the peaks for the first test image gives the following.
From this histogram, one easy way to get both ascending and descending lines to contribute to the same grid orientation is to add the $\theta < 0$ half of the histogram with the $\theta \geq 0$ half. This way, the two peaks for ascending and descending lines will add up, and we can then extract the maximum to get the grid orientation. From that we deduct the $\theta$ angles for the ascending and descending lines.

An alternative could be to look at the top two peak values in the histogram but this is a less robust method since noise may cause secondary peaks so the two peak values may return values of $\theta$ that are both positive or both negative. It is also possible to look at the peak values in each half of the histogram but this does not guarantee that the angles will be separated by 90°.

Using our convention, the orientation that we detect for image 1 is $\alpha = 83°$.

**Part D (6 points):**

The Hough peaks for $\theta = \alpha$ correspond to the ascending lines, and we go from line A to line B by increasing the value of $\rho$. We can thus get the $\rho$ values for lines A and B by respectively taking the minimum and maximum values of $\rho$ for these peaks. Similarly, the Hough peaks for $\theta = \alpha - 90°$ correspond to the descending lines, and we can get the $\rho$ values for lines C and D by respectively taking the minimum and maximum values of $\rho$ for these peaks (note that in our figure, the values of $\rho$ are negative for the descending lines).

It is recommended to look at the peaks for values of $\theta$ within 1° of the expected value, to allow for more robustness, in case the peaks for the lines are detected with a slightly different $\theta$ value.

The coordinates of I must verify the equation for the two lines:

$$\begin{align*}
\rho_1 &= x_1 \cos \theta_1 + y_1 \sin \theta_1 \\
\rho_2 &= x_1 \cos \theta_2 + y_1 \sin \theta_2
\end{align*}$$

We can rewrite these as a matrix operation:

$$\begin{bmatrix} \rho_1 \\ \rho_2 \end{bmatrix} = \begin{bmatrix} \cos \theta_1 & \sin \theta_1 \\ \cos \theta_2 & \sin \theta_2 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

Finally, the coordinates of I can be deduced from the line parameters with the formula:

$$\begin{bmatrix} x_I \\ y_I \end{bmatrix} = \begin{bmatrix} \cos \theta_1 & \sin \theta_1 \\ \cos \theta_2 & \sin \theta_2 \end{bmatrix}^{-1} \begin{bmatrix} \rho_1 \\ \rho_2 \end{bmatrix}$$

This formula allows us to find the four corners, shown in the image below.
Finally, in order to rectify the QR code, we should perform a counter-clockwise rotation of angle $\alpha$. This is the output we get for our first test image. Note that depending on the conventions you use, you may end up with a different orientation of the code, which is fine.

Part E (6 points):

In order to get the binary value corresponding to each 10x10 tile in the rectified image, we compute the mean of the tile and compare it with the threshold value 0.5. Now that we have a 21x21 binary matrix, we may still need to rotate it. In order to do that, we consider each 7x7 corner of the code, apply a XOR with the known fixed template and we count the number of pixels that differ (simple sum). The corner that has the highest value is the one that we put at the bottom right.

This gives the following matrix for the first test image:
Note that due to the noise, a few values are actually incorrect, as is clearly visible on the marker at the bottom left. This is not a problem: the QR codes are built-in with error correction to account for these errors.

Finally, we can decode the message from all codes and discover the secret message by concatenating all the strings. The secret message is the following quote:

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools -Douglas Adams

MATLAB code:

```matlab
% EE368/CS232 WIN17-18
% Midterm
% Problem: QR code reader
% Script by: Jean-Baptiste Boin, Jayant Thatte

clear all
javaaddpath('qr_code.jar');

qr_code_size = 21;
hidden_message = '';

for test_idx = 1:9
    img_noisy = im2double(imread(sprintf('qr_test_%d.png', test_idx)));
    
    % Find center and crop image
    crop_size = 340;
    img_gauss = imgaussfilt(img_noisy, 100);
    [~,min_idx] = min(img_gauss(:));
    [cent_y, cent_x] = ind2sub(size(img_gauss), min_idx);
    img_cent = img_noisy(cent_y-crop_size/2+1:cent_y+crop_size/2, ...%
                            cent_x-crop_size/2+1:cent_x+crop_size/2);

    % Apply median filtering, binarize and extract edges
    img_filt = medfilt2(img_cent, [3 3], 'symmetric');
    level = graythresh(img_filt);
    img_filt_bin = imbinarize(img_filt, level);
    edges = edge(img_filt_bin, 'Canny');

    % Compute Hough transform and get histogram of peaks
    [H,theta,rho] = hough(edges, 'Theta', -90:89);
    num_peaks = 40;
    thresh = 0;
    peaks = houghpeaks(H, num_peaks, 'Threshold', thresh);
    theta_hist = hist(theta(peaks(:,2)), theta);

    % Select the angle by adding contributions to angles from orthogonal lines
    theta_hist_wrap_around = theta_hist(1:90) + theta_hist(91:180);
    [~,idx] = max(theta_hist_wrap_around);
    alpha = theta(idx)+90;
```
theta_asc = alpha;
theta_desc = alpha - 90;

% Get the rho values for the outer lines
rho_peaks = rho(peaks(:,1));
theta_peaks = theta(peaks(:,2));
rho_asc = rho_peaks(abs(theta_peaks - theta_asc) <= 1);
rho_desc = rho_peaks(abs(theta_peaks - theta_desc) <= 1);
rho_a = min(rho_asc);
rho_b = max(rho_asc);
rho_c = min(rho_desc);
rho_d = max(rho_desc);

% Find coordinates of the corners of the QR code
rho_a = min(rho_asc);
rho_b = max(rho_asc);
rho_c = min(rho_desc);
rho_d = max(rho_desc);

% Rectify image using two opposite corners
img_rectified = rectify_image(img_filt, alpha, ...
    corners(:,1), corners(:,4));

% Get binary values
qr_code_values = zeros(qr_code_size);
tile_size = 10;
for i = 1:qr_code_size
    for j = 1:qr_code_size
        tile = img_rectified((i-1)*tile_size+1:i*tile_size, ...
            (j-1)*tile_size+1:j*tile_size);
        qr_code_values(i,j) = mean(tile(:)) > 0.5;
    end
end

% Compare each corner with the following template and rotate the
% matrix so that the most dissimilar corner is located at the
% bottom right
template = [
    0 0 0 0 0 0 0;
    0 1 1 1 1 1 0;
    0 1 0 0 0 1 0;
    0 1 0 0 0 1 0;
    0 1 0 0 0 1 0;
    0 1 1 1 1 0;
    0 0 0 0 0 0
];
distances = [
    sum(sum(xor(template, qr_code_values(end-6:end,1:7))));
    sum(sum(xor(template, qr_code_values(1:7,1:7))));
    sum(sum(xor(template, qr_code_values(1:7,end-6:end))));
    sum(sum(xor(template, qr_code_values(end-6:end,end-6:end))));
];
[~, amax] = max(distances);
qr_code_values_rot = rot90(qr_code_values, amax);

% Decode the QR code and concatenate to result string
result = decode_qr_code(qr_code_values_rot);
hidden_message = [hidden_message result];

% Save/print various outputs for the first image
if test_idx == 1
mkdir('output');
imwrite(img_cent, 'output/part_a.png');
imwrite(img_filt, 'output/part_b1.png');
imwrite(img_filt_bin, 'output/part_b2.png');
imwrite(edges, 'output/part_b3.png');
fprintf('Orientation of image 1: %d degrees\n', alpha)
figure(1); clf;
imsho(img_filt)
hold on;
plot(corners(1,:),corners(2,:),'ro','MarkerSize',15,
     'LineWidth',3)
print('output/part_d1','-dpng','-r0')
imwrite(img_rectified, 'output/part_d2.png');
imwrite(kron(qr_code_values_rot, ones(10)),
     'output/part_e.png');
end

disp(hidden_message)
Problem 5: Image Stitching (30 points)

We will tackle this problem using the following steps: (A) SIFT feature extraction, (B) feature matching, (C) compute horizontal and vertical shifts to align the input images, (D) color correction, (E) linear alpha blending to synthesize final panorama, (F) Postprocessing

Part A: SIFT Feature Extraction

We use the VLFeat library (vlfeat.org) to extract SIFT features using \texttt{vl\_sift} function with a PeakThresh of 0.03 and an EdgeThresh of 10. Note that we read the input images to have a range of $[0, 1]$. If you choose to have a range of $[0, 255]$, you will need a PeakThresh of about 7.5

![Extracted SIFT Features]

Part B: Feature Matching

For each image pair, we match features by finding nearest neighbors in descriptor space using L2 distance. Consider a feature and the corresponding descriptor for image 1. We denote these using $f_1$ and $d_1$ respectively. Then the matching feature in image 2 [$f_2^*, d_2^*$] is obtained by the equation,

$$d_2^* = \arg \min ||d_1 - d_2||^2$$

Once we find all the matches, we then rank the matches using ratio of distance robustness metric.
robustness(d₁, d₂*) = || d₁ - d₂** || / || d₁ - d₂* ||
where d₂* is the best match for d₁ among the features of image 2 and d₂** is the second best match. Notice that robustness defined as above returns a number in [1, ∞). We say a match is robust if the L2 distance for the matching feature is much smaller than the L2 distance to the nearest non-matching feature. Intuitively, this means that unique matches have a high robustness according to our metric.

Top 100 matching features sorted by robustness, for each image pair

Part C: Estimate Shifts

For more complicated homography, the 3x3 homography matrix is typically estimated using RANSAC. However, for our simple model which only has horizontal and vertical shifts, we can simply use the median of horizontal and vertical shifts computed across the top 100 feature matches sorted by robustness.

\[
\text{shift}_x(I, I') = \text{median}([x_1' - x_1, x_2' - x_2, \ldots, x_{100}' - x_{100}])
\]
\[
\text{shift}_y(I, I') = \text{median}([y_1' - y_1, y_2' - y_2, \ldots, y_{100}' - y_{100}])
\]
where $x_i$ and $y_i$ are the x and y coordinates respectively of the $i^{th}$ matching feature in image $I$ and $x'_i, y'_i$ are those of the corresponding feature in $I'$. We use median as opposed to mean, since the median value is minimally affected by outliers and is therefore robust against noisy matches. We obtained the following shifts for the 4 image pairs.

Horizontal shifts: 591.1775, 601.0782, 608.2762, 593.6235 pixels  
Vertical shifts: -18.3676, 50.6672, -47.3860, -24.8107 pixels

Finally, we use the computed shifts to align the images. An image showing all of the shifted images overlaid is shown below.

**Part D: Color Correction**

We can now use the overlapping regions in the aligned images to perform color matching. We start by taking the middle image and we color balance it using Shades of Gray algorithm with $p=6$.

![Input image 3 before and after color balancing](image)

Now that the center image is color balanced, we each neighboring image (image 2 and image 4), we compute a 3x4 linear transform matrix $A$ that will transform those images to have the same color balance as image 3.

Consider the process of color matching image 2 to have the same balance as image 3. We use the pixels in the overlapping region in the aligned images 2 and 3 to perform this operation. Let $Y$ denote a 3xN matrix where each column denotes the RGB value of a pixel in image 3 in the overlapping region, and N is the total number of pixels in the overlapping region. Let $X$ be the corresponding matrix for image 2.

Thus, the $i^{th}$ column of $X$ and $Y$ give the RGB value of the same pixel in images 2 and 3 respectively. We first append the a 4$^{th}$ row to $X$ containing all 1’s so that $X$ has a shape of 4xN. Then we find a 3x4 linear transform matrix $A$ such that

$$A = \text{arg min } \| Y - AX \|$$
We then transform the color of all pixels in image 2 using the computed matrix A.

We first use image 3 as reference to balance images 2 and 4. We then use image 2 to balance image 1 and image 4 to balance image 5.
**Part E: Alpha Blending**

*Without Alpha Blending:*

The final task is to merge all these images. The most naïve way of doing this would be to simply add the 5 images together and divide by 5.

\[
\text{No alpha blending: output} = \frac{I_1 + I_2 + I_3 + I_4 + I_5}{5}
\]

This is clearly incorrect. The problem is that since the images are shifted relative to each other, each output pixel receives contributions from varying number of inputs and therefore dividing by 5 is not the right way to compute the average.

*Constant Alpha Blending:*

This brings us to alpha blending. We create 5 alpha masks, each of which has the same height and width as the output image. The alpha masks are computed as follows.

\[
A_i[x, y] = \begin{cases} 
1 & \text{if output}[x, y] \text{ receives a contribution from } I_i \\
0 & \text{otherwise}
\end{cases}
\]

The output is computed as

\[
\text{output} = \frac{\text{sum}(I_i \cdot A_i)}{\text{sum}(A_i)}
\]

where \(\cdot\) and \(\div\) denote elementwise multiplication and division, respectively.

This image looks much better and actually even flawless at first glance. But if we look closer, we can observe 2 types of artifacts at boundaries of original images – (1) seams due to color mismatch and (2) ghosting where the scene objects do not fully align.
Left: Visible seams at boundaries of input images due to color mismatch
Right: Ghosting at image boundaries due to misaligned scene objects

Such misalignment could result from incorrect homography estimation, parallax from small viewpoint changes between successive camera poses, or movement of scene objects from one shot to the next.

Linear Alpha Blending:

Linear alpha blending aims to mitigate the above artifacts. Instead of assigning a constant alpha value to all pixels of an input image, this algorithm gives more weight to image center and the weight (or alpha) linearly drops off to 0 at the periphery of the image. An example of this is shown below.

Image 3 aligned according to the computed horizontal and vertical shifts

Alpha 3 corresponding to linear alpha blending

Alpha 3 corresponding to constant alpha blending
This solves both ghosting and seams

\[ \text{Constant alpha blending} \quad \text{Linear Alpha Blending} \]

\[ \text{Constant alpha blending} \quad \text{Linear Alpha Blending} \]

The output panorama synthesized after all of the above steps is shown below.

\[ \text{Panorama} \]

**Part F: Postprocessing**

This brings us to the last part of this problem – process the generated panorama to make it ready for the end-user. We will perform two types of postprocessing – (1) crop the panorama to a nice rectangle, (2) adaptive histogram equalization.

*Crop to Rectangle:*

There are several ways to achieve this. We solve this problem by finding the largest rectangle that can fit inside the current irregularly shaped panorama. The resulting panorama after cropping appropriate borders is shown below.
Adaptive Histogram Equalization:

Finally, we equalize the histograms of the output panorama to enhance its appearance. We use a clip limit of 0.002 to generate the final result shown below.

% EE368/CS232 WIN17-18
% Midterm
% Problem: Image Stitching
% Script by: Jean-Baptiste Boin, Jayant Thatte

function result = pano_stitching(userConfig)
    I = readImages;
    [f, d] = extractFeatures(I);
    topMatches = matchFeatures(d);
    shift = computeShifts(f, topMatches);
    [out, alpha] = alignImages(I, shift);
    out = colorCorrection(out, alpha);
    result = sum(out.*alpha, 4);
    result = cropToRectangle(result);
    result = histogramEqualization(result);
end

function userConfig
    global imageTemplate n numMatchesToUse;
    imageTemplate = 'im_%d.png';  % Filename template
    n = 5;  % No. of input images
    numMatchesToUse = 100;
end

function I = readImages
    global imageTemplate n h w c;
    [h, w, c] = size(imread(sprintf(imageTemplate, 1)));
    I = zeros([h, w, c, n]);
    for i=1:n
        I(:,:, :, i) = im2double(imread(sprintf(imageTemplate, i)));
    end
end

function [f, d] = extractFeatures(I)
    global n;
    f = cell(1, n);  % Features
end
d = cell(1, n); % Descriptors
for i=1:n
    [f{i}, d{i}] = vl_sift(single(rgb2gray(I(:, :, :, i))),
    'PeakThresh', 0.03);
end
return

function topMatches = matchFeatures(d)
% This function extracts top <numMatchesToUse> robust matches %
% using ratio of L2 distances robustness metric %
% This is done for each image pair, or <n-1> times in total %
global n numMatchesToUse;
% For image pair i, <topMatches> saves the indices of matching features
% in the first and the second image in <topMatches(:, 1, i)> and %
% <topMatches(:, 2, i)> respectively
for i=1:n-1
    [maxNumCommon, idx] = min([size(d{i}, 2), size(d{i+1}, 2)]);
    ref = idx + i - 1;
    other = 2 - idx + i;
    scores = zeros(1, maxNumCommon);
    matchIdx = zeros(1, maxNumCommon);
    for j = 1:maxNumCommon
        % For each feature
        dist = sqrt(sum((d{ref}(:, j) - d{other}).^2, 1));
        % Get distance to the best match
        [minDist, matchIdx(j)] = min(dist);
        % Ratio of distances robustness metric
        % = (L2 distance for best match)
        % / (L2 distance for second best match)
        scores(j) = minDist / min([dist(1:matchIdx(j)-1), ...
                                  dist(matchIdx(j)+1:end)]);
    end
    [~, sortedIdx] = sort(scores); % Sort features by robustness
    % Use top <numMatchesToUse> features %
    topMatches(:, idx, i) = sortedIdx(1:numMatchesToUse);
    topMatches(:, 3 - idx, i) = matchIdx(sortedIdx(1:numMatchesToUse));
end
return

function shift = computeShifts(f, topMatches)
global n;
shift = zeros(n, 2);
for i=1:n-1
    % Use median to compute shifts to be robust to outliers %
    shift(i+1,:) = median(f{i}(1:2, topMatches(:, 1, i)) - f{i+1}(1:2, topMatches(:, 1, i)), 2);
end
shift = cumsum(shift, 1);
shift = shift - min(shift, [], 1);
return

function [out, alpha] = alignImages(I, shift)
global h w c n H W;
% Calculate the size of output panorama %
H = ceil(max(shift(:, 2))) + h;
W = ceil(max(shift(:, 1))) + w;
out = zeros(H, W, c, n);
alpha = zeros(H, W, 1, n);
% <alphaTemplate> has a value of 1 at the center and drops off to 0 at
% the periphery
alphaTemplate = min(linspace(0.5/h, 1-0.5/h, h),
    linspace(1-0.5/h, 0.5/h, h))' * min(linspace(0.5/w, 1-0.5/w,
    w),
    linspace(1-0.5/w, 0.5/w, w));
alphaTemplate = alphaTemplate / max(alphaTemplate(:));
for i=1:n
    out(round(shift(i, 2))+(1:h), round(shift(i, 1))+(1:w), :, i)
    = I(:, :, :, i);
    alpha(round(shift(i, 2))+(1:h), round(shift(i, 1))+(1:w), 1,
    i)
    = alphaTemplate;
end
% Normalize alpha so that the 5 alpha values for each pixel sum
to 1
alpha = alpha ./ sum(alpha, 4);
return

function out = colorCorrection(out, alpha)
% Color balance the center image
out = colorBalance(out, alpha, 3);
% Match remaining images to have the same color balance
out = matchColors(out, alpha, 3, 2);
out = matchColors(out, alpha, 2, 1);
out = matchColors(out, alpha, 3, 4);
out = matchColors(out, alpha, 4, 5);
return

function out = colorBalance(out, alpha, ref)
im = getImage(out, alpha, ref); % Extract image
% Color balance using Shades of Gray with p=6
gains = squeeze((mean(mean(im.^6, 1), 2)).^(1/6));
for i=1:3
    im(:, :, i) = im(:, :, i)/gains(i);
end
% Scale to occupy full [0, 1] range
im = im - min(im(:));
im = im / max(im(:));
% Place image back in <out> matrix
out = putImage(out, alpha, im, ref);
return

function out = matchColors(out, alpha, ref, tgt)
global H W c
% Detect overlapping region between reference and target
mask = (alpha(:, :, :, ref) .* alpha(:, :, :, tgt)) > 0;
% Estimate a linear transform that maps target to reference
x = reshape(out(:, :, :, tgt), [H*W, c]); % Reshape target
y = reshape(out(:, :, :, ref), [H*W, c]); % Reshape reference
x = x(:, mask==1); % Retain only the matching region
y = y(:, mask==1); % Retain only the matching region
x = cat(1, x, ones(1, size(x, 2))); % Append 1's
A = ((y' * x') ^ -1); % Estimate linear transform
% Apply the estimated transform to the target image
x = cat(1, reshape(out(:, :, :, tgt), [H*W, c]),
    ones(1, H*W));
x = reshape(x, [H, W, c]);
temp = out(:, :, :, tgt);
mask = repmat(alpha(:, :, :, tgt)>0, [1, 1, c]);
temp(mask) = x * mask; out(:, :, :, tgt) = temp;
return

function result = histogramEqualization(result)
% Enhance contrast by equalizing histograms
clipLimit = 0.002;
result = cat(3,...
    adapthisteq(result(:, :, 1), 'ClipLimit', clipLimit),...
    adapthisteq(result(:, :, 2), 'ClipLimit', clipLimit),...
    adapthisteq(result(:, :, 3), 'ClipLimit', clipLimit));
return

function result = cropToRectangle(result)
    while 1
        % Estimate how much border should be cropped from each side
        [missing, margins] = getMarginsToCrop(result);
        % When the no more cropping is needed, exit
        if all(margins==0)
            break
        end
        % Pick one of top, bottom, left, right and crop that margin
        result = performCrop(missing, margins, result);
    end
return

function [missing, margins] = getMarginsToCrop(result)
    global H W;
    missing = 1*isnan(sum(result, 3));
    [u, v] = find(missing==1);
    distFromEdge = [u, v, H-u+1, W-v+1]; % Compute for each missing pixel
    % For each missing pixel, pick the distance to the closest edge
    % <MarginType> indicates [1: top, 2: left, 3: bottom, 4: right]
    [distFromEdge, marginType] = min(distFromEdge, [], 2);
    margins = zeros(1, 4);
    for i=1:4
        % If an edge has no missing pixels close to it, crop margin is 0
        if isempty(find(marginType==i, 1))
            margins(i) = 0;
        else
            % For each side, find the border width that must be cropped
            margins(i) = max(distFromEdge(marginType==i));
        end
    end
return

function result = performCrop(missing, margins, result)
% This function picks on of top, bottom, left, right margins based on
% whichever border has the largest fraction of missing pixels.
The % width of the border is dictated by <margins>
    global H W;
    [x, y] = meshgrid(1:W, 1:H);
    percentageMissing = [mean(missing(y<=margins(1))),...,
                         mean(missing(x<=margins(2))),...
                         mean(missing(y>H-margins(3))),...
                         mean(missing(x>W-margins(4)))];
    [~, idx] = max(percentageMissing);
    if idx == 1
        result = result(margins(1)+1:end, :, :); % Crop top border
elseif idx == 2
    result = result(:, margins(2)+1:end, :);  % Crop left border
elseif idx == 3
    result = result(1:end-margins(3), :, :);  % Crop bottom border
else
    result = result(:, 1:end-margins(4), :);  % Crop right border
end
[H, W, ~] = size(result);
return

function im = getImage(out, alpha, ref)
    global h w c;
    im = out(:, :, :, ref);
    im = im(repmat(alpha(:, :, :, ref)>0, [1, 1, c]));
    im = reshape(im, [h, w, c]);
return

function out = putImage(out, alpha, im, ref)
    global c;
    temp = out(:, :, :, ref);
    temp(repmat(alpha(:, :, :, ref)>0, [1, 1, c])) = im;
    out(:, :, :, ref) = temp;
return