LAB2: Synergy of Image Processing and Feature Detections

Updated: v1. 2015-10-18; v.2 10-25. v.3 2019-3-3 By Prof. Min Wu (minwu@umd.edu) Edited: Nathaniel Ferlic & Kevin Ho

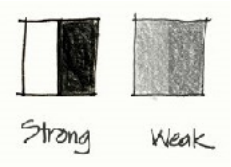
Contents

O. Prelab

Directions: Answer the following questions (1-4) BEFORE the corresponding lab. These are due by the time lab starts. *You may use any resources you find online or in text, but you may not work with your fellow students.*

Resources:

- Strong vs. weak edge:



- Example:

O.1 Prelab Questions

Question 1: Using your phone or laptop, take a photo with many subjects in it. Then, in a photo editing software (like MS Paint or similar), trace the strong edges with RED and trace the weak edges in BLUE. Read the edited image into MATLAB and use `imshow` to display it on a figure.

Question 2: Run the following MATLAB code to convert your **ORIGINAL** photo (i.e. the one that does not have red and blue lines) into a usable file for Lab. Replace the `IMAGE_NAME.FILE_EXTENSION` with the full filename of your photo. Save the outputted .bmp file to an accessible location.

```
I = imread('IMAGE_NAME.FILE_EXTENSION');

I = rgb2gray(I);

I=imresize(I,[640 640],'lanczos3');

imwrite(I,'IMAGE.bmp')
```

Question 3: For an 8-bit grayscale image, what is the range of values each pixel can take? What does the lowest value in this range represent? What does the highest value in this range represent?

Question 4: Navigate to this website to learn about image kernels. Answer the following questions:

- What is an image kernel?

- Describe the process of applying a kernel to an image. This process is called kernel convolution.
- Why do we have issues with applying a kernel to the sides (i.e. the top and bottom rows, and the left and right columns) of an image? How can we deal with these when performing kernel convolution? * Upload the `IMAGE.bmp` file you generated earlier to the website by clicking the 'Browse…' button. To find the IMAGE.bmp file, you will need to switch the search from 'All Supported Types' to 'All Files'
- Apply the `bottom sobel` and `right sobel` kernels to your `IMAGE.bmp` on the website. Save these images, read them into MATLAB, and display them below.
- Why are these `bottom sobel` and `right sobel` kernels useful in detecting edges?

Question 5: A uniform averaging kernel has the value 1/9 in each of its cells given as `uniform_averaging_kernel`. The next example is a weighted average kernel. Which pixel from the original 9 takes the most information of the image? *Note:* When creating your own averging filter make sure the sum of the weights adds up to one

```
format rational
uniform_averaging_kernel = 1/9*ones(3)


weighted_averaging_kernel = ones(3,3)/16; weighted_averaging_kernel(2,2) = 0.5;
weighted_averaging_kernel
```

```
uniform_averaging_kernel =

       1/9            1/9            1/9
       1/9            1/9            1/9
       1/9            1/9            1/9


weighted_averaging_kernel =

       1/16           1/16           1/16
       1/16           1/2            1/16
       1/16           1/16           1/16
```

I. Image Smoothing and Sharpening

```
%clear all   % Use this to clean up the workspace to prepare for this new lab
%close all   % Close all figures if needed
img0 = imread('Lena.bmp');
figure(1), imshow(img0)
```

1) Smoothing: averaging filter

Digital filters can be selected or designed to provide a variety of effect on images, in a similar way as filters would do on other type of signals.

An averaging filter works by setting each pixel of the output image as the average of the original pixel values and its neighborhood; and the average can be a weighted average by adjusting the weight or parameters of the filter. In signal processing terminology, averaging belong to **low-pass filtering**, as it withholds strong variations and passes the low frequency components that form mild or no variations.

The first filter below takes the average value of the original pixel and its 8 neighbors. The second filter takes a weighted average, with more weight given to the original pixel value (i.e. the center of the 3x3 filter matrix is the weight given to the original pixel).

Question 1: What is a low pass filter?

Useful functions: `filter2( )`, `imfilter( )`, `fspecial( )`.

```

```

```
filt_avg1 =

       1/9            1/9            1/9
       1/9            1/9            1/9
       1/9            1/9            1/9
```

Question 2: Create a 3x3 uniform averge kernel as done in the pre-lab and name it `filt_avg1` and feed it into the `filter2()` function as shown in the next line of code.

```
img1_LP = uint8(filter2( filt_avg1, img1 ));
```

Question 3: Create a 3x3 weighted average kernel. The center pixel will have a weight of 0.5 and the surrounding pixels will have a weight of their mean. Call your variable `filt_avg2` and feed it to the `filter2()` function as shown in the next line of code.

```
img1_LP2 = uint8(filter2( filt_avg2, img1 ));
```

```
filt_avg2 =

       1/16          1/16          1/16
       1/16          1/2           1/16
       1/16          1/16          1/16
```

Question 4: Produce the subplot as shown below with both averged images along with the original.

Notice the smoothened skin, and the different degrees of smoothening by the two filters.

A more convenient way is to use the newer function imfilter( ) that takes care of variable type and padding issues on the border. Learn more by running this to see documentation in the Help browser: web(fullfile(docroot, 'images/filter-images-using-imfilter.html'))

```
img1_LP2 = imfilter( img1, filt_avg2 );
```

2) Locating Smooth vs. "Busy" Areas:

The difference between the original image and its smoothened version reveals where the original image is busy (in terms of having different values in nearby pixels, which are around edges and texture areas).

In signal processing terminology, this difference belong to **high-pass filtering**, as it withholds smooth area by outputing zero or small values, and passes the high frequency components that form sharp variations.

Tips:

- The images should be converted from unsigned integers to signed datatypes in order to perform difference operation (in MATLAB, this can be done by converting the data to "double".
- Due to the difference operation, the resulting "image" can have both positive and negative values, and the values are very small in areas where nearby pixels have similar shades. To facilitate the display, we specify an argument when using `imshow` (learn more details through the Help documentations in MATLAB).
- Learn to use the built-in tools from the tool bar and menus in the figure window to zoom in/out, read pixel values or data points, etc.

Question 5: In-terms of image processing what does high-pass filtering mean doing to an image?

```
img1_HP = double(img1) - double(img1_LP);

figure(3), subplot(1, 2, 1), imshow(img1), title('Original'); axis equal
subplot(1, 2, 2), imshow(img1_HP, []), title('High-pass Filter1'), colorbar
%h3 = subplot(1, 3, 3); cla(h3), title('')  % clear the previous display if any
```

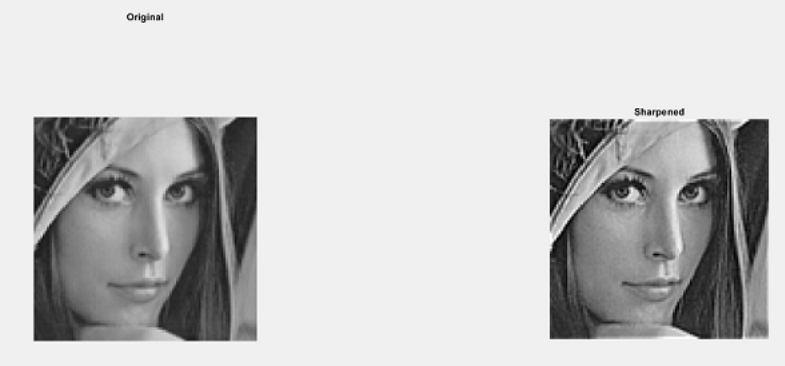
3) Sharpening: Making transitions more dramatic

Once we compute the degree of transitions using the high-pass filtering, we can add it back to the original image to strengthen the degree of transition, thus **sharpen** the image. We can apply a scaling factor to the transition to control the degree of sharpening.

Question 5: What is going on in the third line of code below? How is the image being sharpend?

Question 6: Change the SP_factor and see what happens to the sharpened image and plot it on the subplot below.

```
SP_factor = 1;  % parameter to control the degree of sharpening
temp = double(img1) + double(img1_HP) * SP_factor;
img1_SP = uint8( (temp > 255) * 255 + (temp >= 0 & temp <= 255) .* temp );
            % clip the values outside [0, 255] range and convert to uint8

subplot(1, 3, 3), imshow(img1_SP), title('Sharpened')
```



Consider an area in an image that is from the same object and supposed to be of similar values, but turns out to have a lot of variations in pixel values due to noise or imperfections in camera capture.

Question 7: What would be the effect of the averaging filter on this area?

Question 8: What would be the effect of sharpening on this area?

II. Detecting and Visualizing Edges

An edge in an image is a line or curve that shows a path of rapid change in image intensity. Edges are often associated with the object boundaries and are crucial in many image processing and computer vision tasks.

- The high-pass filter that we have explored above can be used to detect edge. Edge detection essentially involves high-pass operations. However, a simple high-pass filter may not always capture all desirable boundary points (depending on how a threshold is set, it can be tricky to catch mild boundaries and leave out the variations that are due to noise and not on object boundaries). The high-pass filter you have seen also does not tell along which direction the edge is.
- A common approach to determine the direction of edge is to use two high-pass filters tuned to orthogonal directions (90-degree apart of each other). For example, the first filter is tuned to take the difference between surrounding left and right pixels; this captures the degree of horizontal changes of image intensity, and a vertical-looking edge would give a high response of this filter (in terms of the absolute value of the differences. Similarly, the second filter is tuned to take the difference between surrounding pixels above and below each image pixel being processed; this captures the degree of vertital changes of image intensity, and a horizontal-looking edge would give a high response of this filter.
- The results from the two filters can be combined to describe the overall strength of the edge and its orientation.

Question 9: Which kind of edge do each of the filters, `edge_filt1_1` and edge_filt1_2, below find?

Question 10: In simple terms what represents an edge? *Hint: Think of the difference between the two backgrounds in greyscale*

Question 11: Based on your answer from question 10, what does this represent in-terms of a mathematical operation?

```
edge_filt1_1 = [-1, 0, 1; -1, 0, 1; -1, 0, 1],  % "Prewitt" edge filters
edge_filt1_2 = [-1, -1, -1; 0, 0, 0; 1, 1, 1],
```

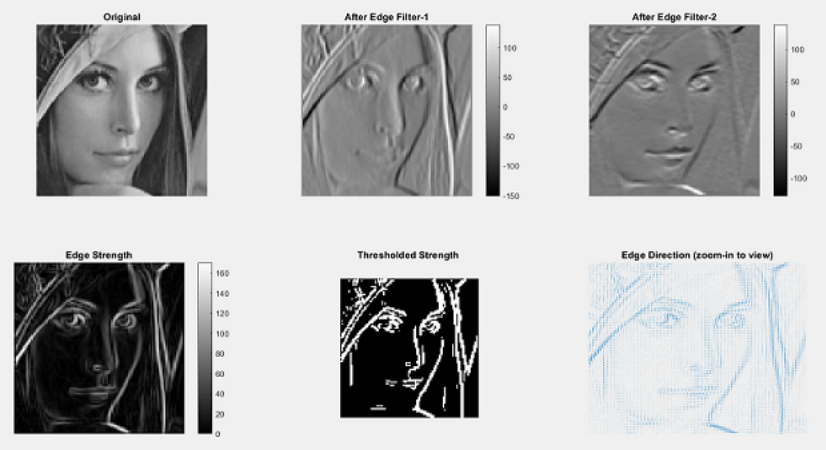
```
img1_edge1_1 = filter2( edge_filt1_1 / 3, img1 );  % apply edge filters
img1_edge1_2 = filter2( edge_filt1_2 / 3, img1 );
```

```
edge_filt1_1 =

     -1     0     1
     -1     0     1
     -1     0     1


edge_filt1_2 =

     -1    -1    -1
      0     0     0
      1     1     1
```

```
img1_edge1_2(1,:) = 0;    % handle cases on image border
img1_edge1_1(:, 1) = 0;
img1_edge1_2(size(img1,1),:) = 0;
img1_edge1_1(:, size(img1,2)) = 0;
```

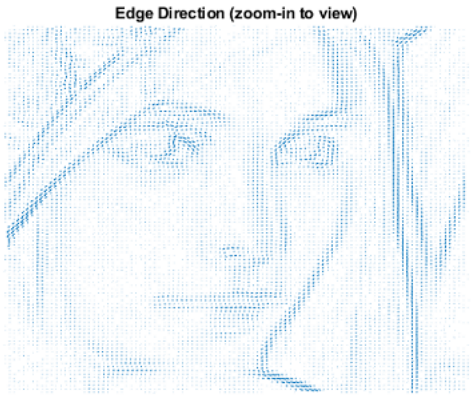
```
img1_edge1 = sqrt( double(img1_edge1_1) .^2 + double(img1_edge1_2) .^2 ); % edge strength
edge_th = 40; % threshold of edge strength
```

```
figure(4), subplot(2, 3, 1), imshow(img1), title('Original')
subplot(2, 3, 2), imshow(img1_edge1_1, []), title('After Edge Filter-1'), colorbar
subplot(2, 3, 3), imshow(img1_edge1_2, []), title('After Edge Filter-2'), colorbar
subplot(2, 3, 4), imshow(img1_edge1, []), title('Edge Strength'), colorbar
subplot(2, 3, 5), imshow(img1_edge1 > edge_th, []), title('Thresholded Strength ')
axis equal
subplot(2, 3, 6), st=1;    %% visualize edge directions at a selected resolution
quiver( img1_edge1_2( size(img1,1):(-st):1, 1:st:size(img1,2) ), ...
        img1_edge1_1( size(img1,1):(-st):1, 1:st:size(img1,2) ) ),
axis([1, size(img1,2), 1, size(img1,1) ]), axis off,
title('Edge Direction (zoom-in to view)')
```



Question 12: Explain the difference between figure Edge_Filter 1 and Edge_Filter 2.

```
figure(6); st = 1;
quiver( img1_edge1_2( size(img1,1):(-st):1, 1:st:size(img1,2) ), ...
        img1_edge1_1( size(img1,1):(-st):1, 1:st:size(img1,2) ) ),
axis([1, size(img1,2), 1, size(img1,1) ]), axis off,
title('Edge Direction (zoom-in to view)')
```



Question 13: Use the code above to produce a quiver plot and zoom in very close on the blue image. What way are the arrows pointing in different places of the figure?

An alternative set of edge filter is known as the **Sobel** filters. They give slightly different weight from the above set of simple filters (the **Prewitt** filters)

```
edge_filt2_1 = [-1, 0, 1; -2, 0, 2; -1, 0, 1],  % "Sobel" edge filters
edge_filt2_2 = [-1, -2, -1; 0, 0, 0; 1, 2, 1],

img1_edge2_1 = filter2( edge_filt2_1 / 4, img1 );  % apply edge filters
img1_edge2_2 = filter2( edge_filt2_2 / 4, img1 );

   %... continue with the visualization of edge detection results
   %    as shown above
```

```
edge_filt2_1 =

     -1     0     1
     -2     0     2
     -1     0     1


edge_filt2_2 =

     -1    -2    -1
      0     0     0
      1     2     1
```

MATLAB also offers a built-in function `edge` that implements several popular edge detection algorithms. Among them, the **Canny** detector is the most widely used ones. It incorporates several important pre- and post- processing to help reliably identify edges. This includes preprocessing to remove variations due to noise, employing more than a single threshold to detect strong and weak edges, and linking occasional broken pieces of a continuous edge. Specific algorithm and parameter settings can be set as input to the function. Usage examples can be found at:
http://www.mathworks.com/help/images/edge-detection.html

```
img1_edge3 = edge(img1,'sobel');
img1_edge4 = edge(img1,'canny');
figure(7),  subplot(1, 3, 1), imshow(img1), title('Original')
subplot(1, 3, 2), imshow(img1_edge3), title('Sobel Edge Detector')
subplot(1, 3, 3), imshow(img1_edge4), title('Canny Edge Detector')
```

III. Detecting Circles

Explore step-by-step this built-in demo that combine RGB-to-gray, edge detection at varying sensitivity parameters with feature detection based on strategic mathematical properties to hunt for circles in an image.

http://www.mathworks.com/help/images/examples/detect-and-measure-circular-objects-in-an-image.html

Click the top-right box on the page to download and step-by-step run the demo.

Question 14: In the MATLAB demo how are "lighter" circles dealt with?

Question 15: Would the way we implemented edge detection, in part 2, work for all cases of the circles? Why or why not?

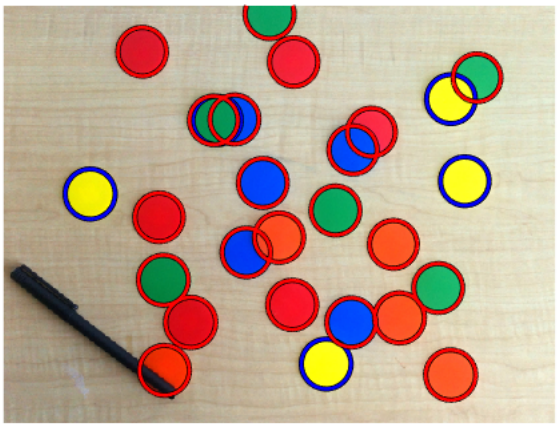
Question 16: What are gradient values?

Question 17: Run the demo code in MATLAB and produce the final image and attatch the code and final plot to your lab report. Do not submit all the intermediate images.

Question 18: Change the Sensitivity in the following code line below from 0.92 to different values and plot what happens. Place the figures in a subplot along with the original image at 0.92 sensitivity. do at least 2 different sensitivities. You can get the rest of the code from the demo to plot the circles

edit DetectCirclesExample % Run this line for the demo

```
[centers,radii] = imfindcircles(rgb,[20 25],'ObjectPolarity','dark', ...
          'Sensitivity',0.95);

[centersBright,radiiBright,metricBright] = imfindcircles(rgb,[20 25], ...
    'ObjectPolarity','bright','Sensitivity',0.92,'EdgeThreshold',0.1);
```



IV. Putting Together: Stitching Images into a Panorama

Demo on panoramic image stitching: http://www.mathworks.com/help/vision/examples/feature-based-panoramic-image-stitching.html

Question 18: Use the Tahoe images to create a panorama using the following code from the above MATLAB demo. Is the stitching done well? Is there any problem and what may be the cause? Include your stitched image and discuss your findings.

Question 19: After loading the pictures in create a new folder and crop the images using MS Paint resize the images in MATLAB using `im1 = imread()` and `im2 = imresize(im1 , [length width])` and `imwrite(im2,'filename.jpg')` _Note: In `buildingDir = fullfile('.');` the `'.\'` represents the local directory and after the backslash place the folder name with the Tahoe images to be loaded into MATLAB. This can be done for other panorama photos such as the dragons. All pictures are downloadable on ELMS.

You may need to put your images in a folder, make a copy of the MATLAB's original FeatureBasedPanoramicImageStitchingExample.m code, and revise it accordingly, such as using `imageSet( )` with proper folder path settings.

```
% edit FeatureBasedPanoramicImageStitchingExample  % Run this line for the demo
% *Question:*
```

Step 1 - Load Images

The image set used in this example contains pictures of a building. These were taken with an uncalibrated smart phone camera by sweeping the camera from left to right along the horizon, capturing all parts of the building.

As seen below, the images are relatively unaffected by any lens distortion so camera calibration was not required. However, if lens distortion is present, the camera should be calibrated and the images undistorted prior to creating the panorama. You can use the `cameraCalibrator` App to calibrate a camera if needed.

```
% Load images.
buildingDir = fullfile(toolboxdir('vision'), 'visiondata', 'building');
%buildingDir = fullfile('.\Tahoe');
buildingScene = imageDatastore(buildingDir);

% Display images to be stitched
figure(10);montage(buildingScene.Files)
```

Step 2 - Register Image Pairs

To create the panorama, start by registering successive image pairs using the following procedure:

1. Detect and match features between $I(n)$ and $I(n-1)$.
2. Estimate the geometric transformation, $T(n)$, that maps $I(n)$ to $I(n-1)$.
3. Compute the transformation that maps $I(n)$ into the panorama image as $T(n) * T(n-1) * \ldots * T(1)$.

```
% Read the first image from the image set.
I = readimage(buildingScene, 1);

% Initialize features for I(1)
grayImage = rgb2gray(I);
points = detectSURFFeatures(grayImage);
[features, points] = extractFeatures(grayImage, points);

% Initialize all the transforms to the identity matrix. Note that the
% projective transform is used here because the building images are fairly
% close to the camera. Had the scene been captured from a further distance,
% an affine transform would suffice.
numImages = numel(buildingScene.Files);
tforms(numImages) = projective2d(eye(3));

% Initialize variable to hold image sizes.
imageSize = zeros(numImages,2);

% Iterate over remaining image pairs
for n = 2:numImages

    % Store points and features for I(n-1).
    pointsPrevious = points;
    featuresPrevious = features;

    % Read I(n).
    I = readimage(buildingScene, n);

    % Convert image to grayscale.
    grayImage = rgb2gray(I);
```

```
    % Save image size.
    imageSize(n,:) = size(grayImage);

    % Detect and extract SURF features for I(n).
    points = detectSURFFeatures(grayImage);
    [features, points] = extractFeatures(grayImage, points);

    % Find correspondences between I(n) and I(n-1).
    indexPairs = matchFeatures(features, featuresPrevious, 'Unique', true);

    matchedPoints = points(indexPairs(:,1), :);
    matchedPointsPrev = pointsPrevious(indexPairs(:,2), :);

    % Estimate the transformation between I(n) and I(n-1).
    tforms(n) = estimateGeometricTransform(matchedPoints, matchedPointsPrev,...
        'projective', 'Confidence', 99.9, 'MaxNumTrials', 2000);

    % Compute T(n) * T(n-1) * ... * T(1)
    tforms(n).T = tforms(n).T * tforms(n-1).T;
end
```

At this point, all the transformations in `tforms` are relative to the first image. This was a convenient way to code the image registration procedure because it allowed sequential processing of all the images. However, using the first image as the start of the panorama does not produce the most aesthetically pleasing panorama because it tends to distort most of the images that form the panorama. A nicer panorama can be created by modifying the transformations such that the center of the scene is the least distorted. This is accomplished by inverting the transform for the center image and applying that transform to all the others.

Start by using the `projective2d` `outputLimits` method to find the output limits for each transform. The output limits are then used to automatically find the image that is roughly in the center of the scene.

```
% Compute the output limits  for each transform
for i = 1:numel(tforms)
    [xlim(i,:), ylim(i,:)] = outputLimits(tforms(i), [1 imageSize(i,2)], [1 imageSize(i,1)]);
end
```

Next, compute the average X limits for each transforms and find the image that is in the center. Only the X limits are used here because the scene is known to be horizontal. If another set of images are used, both the X and Y limits may need to be used to find the center image.

```
avgXLim = mean(xlim, 2);

[~, idx] = sort(avgXLim);

centerIdx = floor((numel(tforms)+1)/2);

centerImageIdx = idx(centerIdx);
```

Finally, apply the center image's inverse transform to all the others.

```
Tinv = invert(tforms(centerImageIdx));

for i = 1:numel(tforms)
    tforms(i).T = tforms(i).T * Tinv.T;
end
```

Step 3 - Initialize the Panorama

Now, create an initial, empty, panorama into which all the images are mapped.

Use the `outputLimits` method to compute the minimum and maximum output limits over all transformations. These values are used to automatically compute the size of the panorama.

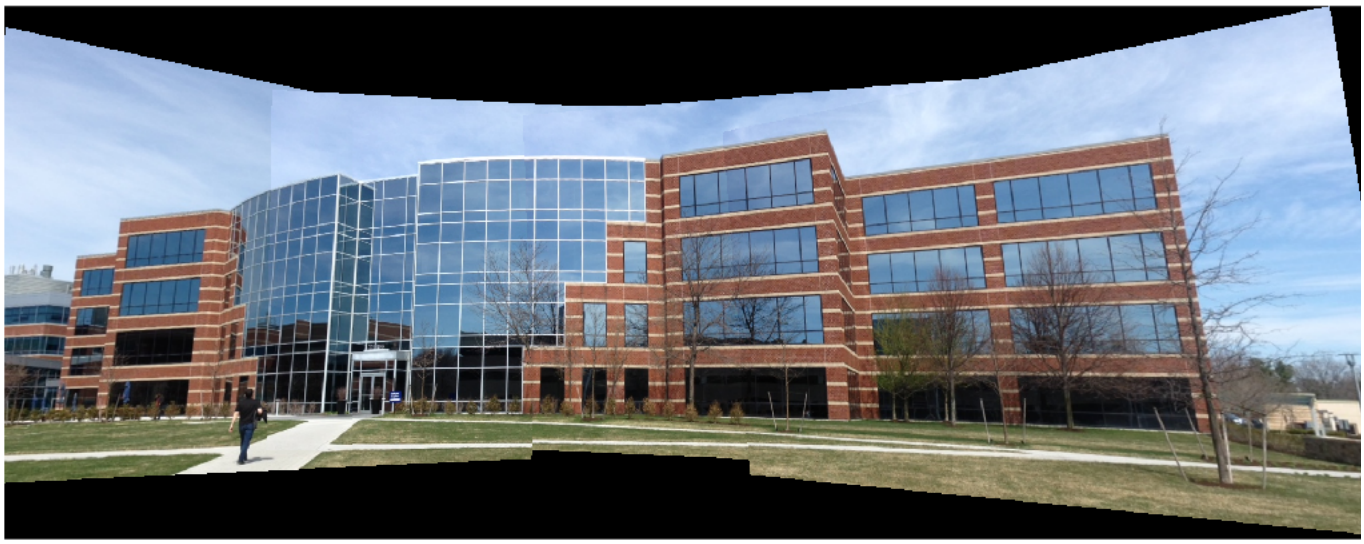
```
for i = 1:numel(tforms)
    [xlim(i,:), ylim(i,:)] = outputLimits(tforms(i), [1 imageSize(i,2)], [1 imageSize(i,1)]);
end

maxImageSize = max(imageSize);

% Find the minimum and maximum output limits
xMin = min([1; xlim(:)]);
xMax = max([maxImageSize(2); xlim(:)]);

yMin = min([1; ylim(:)]);
yMax = max([maxImageSize(1); ylim(:)]);

% Width and height of panorama.
width  = round(xMax - xMin);
height = round(yMax - yMin);

% Initialize the "empty" panorama.
panorama = zeros([height width 3], 'like', I);
```

Step 4 - Create the Panorama

Use `imwarp` to map images into the panorama and use `vision.AlphaBlender` to overlay the images together.

```
blender = vision.AlphaBlender('Operation', 'Binary mask', ...
    'MaskSource', 'Input port');

% Create a 2-D spatial reference object defining the size of the panorama.
xLimits = [xMin xMax];
yLimits = [yMin yMax];
panoramaView = imref2d([height width], xLimits, yLimits);

% Create the panorama.
for i = 1:numImages

    I = readimage(buildingScene, i);

    % Transform I into the panorama.
    warpedImage = imwarp(I, tforms(i), 'OutputView', panoramaView);

    % Generate a binary mask.
    mask = imwarp(true(size(I,1),size(I,2)), tforms(i), 'OutputView', panoramaView);

    % Overlay the warpedImage onto the panorama.
    panorama = step(blender, panorama, warpedImage, mask);
end

figure(11);
imshow(panorama)
```

```
Warning: Image is too big to fit on screen; displaying at 67%
```



Conclusion

This example showed you how to automatically create a panorama using feature based image registration techniques. Additional techniques can be incorporated into the example to improve the blending and alignment of the panorama images[1].

References

[1] Matthew Brown and David G. Lowe. 2007. Automatic Panoramic Image Stitching using Invariant Features. Int. J. Comput. Vision 74, 1 (August 2007), 59-73.

References

- MATLAB demos and documentation pages listed above
- For further explorations on image features and demos:
- http://www.mathworks.com/help/vision/feature-detection-and-extraction.html
- http://www.mathworks.com/help/vision/ug/local-feature-detection-and-extraction.html

---

Published with MATLAB® R2018a