Question 1: Homography Theory

Suppose we have two cameras $\mathbf{C}_1$ and $\mathbf{C}_2$ looking at a common plane Π in 3D space. Any 3D point $\mathbf{P}$ on Π generates a projected 2D point located at $\mathbf{p} \equiv (u_1, v_1, 1)^T$ on the first camera $\mathbf{C}_1$ and $\mathbf{q} \equiv (u_2, v_2, 1)^T$ on the second camera $\mathbf{C}_2$. Since $\mathbf{P}$ is confined to the plane Π, we expect that there is a relationship between $\mathbf{p}$ and $\mathbf{q}$. In particular, there exists a common 3×3 matrix $\mathbf{H}$, so that for any $\mathbf{p}$ and $\mathbf{q}$, the following conditions holds:

$$\mathbf{p} \equiv \mathbf{H}\mathbf{q}$$

We call this relationship **planar homography**. Recall that both $\mathbf{p}$ and $\mathbf{q}$ are in homogeneous coordinates and the equality $\equiv$ means $\mathbf{p}$ is proportional to $\mathbf{Hq}$ (recall homogeneous coordinates). It turns out this relationship is also true for cameras that are related by pure rotation without the planar constraint.

1.1 Homography (5 points)

Prove that there exists an $\mathbf{H}$ that satisfies $\mathbf{p} \equiv \mathbf{Hq}$ given two 3×4 camera projection matrices $\mathbf{M_1}$ and $\mathbf{M_2}$ corresponding to cameras $\mathbf{C_1}$, $\mathbf{C_2}$ and a plane Π. Do not produce an actual algebraic expression for $\mathbf{H}$. All we are asking for is a proof of the existence of $\mathbf{H}$.

Note: A degenerate case may happen when the plane Π contains both cameras' centers, in which case there are infinite choices of $\mathbf{H}$ satisfying $\mathbf{p} \equiv \mathbf{Hq}$. You can ignore this case in your answer.

Consider the projection of 3D homogeneous point $\mathbf{P}$ on plane $\mathbf{\Pi}$ to the two camera planes as $\mathbf{p}$ and $\mathbf{q}$

$$\therefore \mathbf{p} \equiv \mathbf{M_1}\mathbf{P}$$
$$\therefore \mathbf{q} \equiv \mathbf{M_2}\mathbf{P}$$

Since $\mathbf{p}$ and $\mathbf{q}$ are related to the same planar point $\mathbf{P}$ by a system of linear equations, there exists a system of linear equations that relates $\mathbf{p}$ and $\mathbf{q}$ as linear transformations are closed under composition.

Therefore, there exists an $\mathbf{H}$ that satisfies $\mathbf{p} \equiv \mathbf{Hq}$

1.2 Homography under rotation (5 points)

Prove that there exists a homography $\mathbf{H}$ that satisfies $\mathbf{p_1} \equiv \mathbf{Hp_2}$, given two cameras separated by a pure rotation. That is, for camera 1, $\mathbf{p_1} = \mathbf{K_1}\begin{bmatrix}\mathbf{I} & \mathbf{0}\end{bmatrix}\mathbf{P}$ and for camera 2, $\mathbf{p_2} = \mathbf{K_2}\begin{bmatrix}\mathbf{R} & \mathbf{0}\end{bmatrix}\mathbf{P}$. Note that $\mathbf{K_1}$ and $\mathbf{K_2}$ are the 3×3 intrinsic matrices of the two cameras and are different. $\mathbf{I}$ is 3×3 identity matrix, $\mathbf{0}$ is a 3×1 zero vector and $\mathbf{P}$ is the homogeneous coordinate of a point in 3D space. $\mathbf{R}$ is the 3×3 rotation matrix of the camera.

Writing homogeneous coordinate, $\mathbf{P} = \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix}$ where $\mathbf{X}$ are the 3D world coordinates.

$$\therefore \mathbf{p_1} \equiv \mathbf{K_1} \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} \equiv \mathbf{K_1 X}$$

$$\therefore \mathbf{p_2} \equiv \mathbf{K_2} \begin{bmatrix} \mathbf{R} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} \equiv \mathbf{K_2 R X}$$

From the second equation,

$$\mathbf{X} \equiv (\mathbf{K_2 R})^{-1} \mathbf{p_2}$$

Substituting this in the first equation,

$$\therefore \mathbf{p_1} \equiv \mathbf{K_1 X} = \mathbf{K_1} (\mathbf{K_2 R})^{-1} \mathbf{p_2}$$
$$\therefore \mathbf{p_1} \equiv \mathbf{K_1 R}^{-1} \mathbf{K_2}^{-1} \mathbf{p_2}$$

where $\mathbf{K_1 R}^{-1} \mathbf{K_2}^{-1}$ is a 3x3 matrix.

Hence, can be seen that there exists a 3x3 homography ($\mathbf{H} = \mathbf{K_1 R}^{-1} \mathbf{K_2}^{-1}$) from $\mathbf{p_2}$ to $\mathbf{p_1}$

1.3 Correspondences (10 points)

Let $\mathbf{x_1}$ be a set of points in an image and $\mathbf{x_2}$ be the set of corresponding points in an image taken by another camera. Suppose there exists a homography $\mathbf{H}$ such that:
$\mathbf{x_1^i} \equiv \mathbf{H x_2^i} \quad (i \in \{1 \dots N\})$ where $\mathbf{x_1^i} = \begin{bmatrix} x_1^i & y_1^i & 1 \end{bmatrix}^T$ are in homogenous coordinates, $\mathbf{x_1^i} \in \mathbf{x_1}$ and $\mathbf{H}$ is a 3×3 matrix. For each point pair, this relation can be rewritten as

$$\mathbf{A_i h} = 0$$

where $\mathbf{h}$ is a column vector reshaped from $\mathbf{H}$, and $\mathbf{A_i}$ is a matrix with elements derived from the points $\mathbf{x_1^i}$ and $\mathbf{x_2^i}$. This can help calculate **H** from the given point correspondences.

- How many degrees of freedom does $\mathbf{h}$ have? (3 points)
- How many point pairs are required to solve $\mathbf{h}$? (2 points)
- Derive $\mathbf{A_i}$. (5 points)

h has 8 degrees of freedom. So, the number of point pairs required to solve for **h** is 4.

Given, $\mathbf{x_1^i} \equiv \mathbf{H x_2^i}$

$$\begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} \equiv \begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & h33 \end{bmatrix} \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix}$$

Simplifying RHS

$$\begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & h33 \end{bmatrix}\begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix} = \begin{bmatrix} h11 * x_2^i + h12 * y_2^i + h13 \\ h21 * x_2^i + h22 * y_2^i + h23 \\ h31 * x_2^i + h32 * y_2^i + h33 \end{bmatrix}$$

In order for this to match this to $\mathbf{x_1^i}$, need to make the last element 1

$$\begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{h11*x_2^i+h12*y_2^i+h13}{h31*x_2^i+h32*y_2^i+h33} \\ \frac{h21*x_2^i+h22*y_2^i+h23}{h31*x_2^i+h32*y_2^i+h33} \\ 1 \end{bmatrix}$$

$$\therefore x_1^i = \frac{h11 * x_2^i + h12 * y_2^i + h13}{h31 * x_2^i + h32 * y_2^i + h33}$$
$$\therefore y_1^i = \frac{h21 * x_2^i + h22 * y_2^i + h23}{h31 * x_2^i + h32 * y_2^i + h33}$$

$$\therefore (h31 * x_2^i + h32 * y_2^i + h33)x_1^i = h11 * x_2^i + h12 * y_2^i + h13$$
$$\therefore (h31 * x_2^i + h32 * y_2^i + h33)y_1^i = h21 * x_2^i + h22 * y_2^i + h23$$

$$\therefore -h11 * x_2^i - h12 * y_2^i - h13 + x_1^i x_2^i * h31 + x_1^i y_2^i * h32 + x_1^i * h33 = 0$$
$$\therefore -h21 * x_2^i - h22 * y_2^i - h23 + y_1^i x_2^i * h31 + y_1^i y_2^i * h32 + y_1^i * h33 = 0$$

Writing this in matrix form,

$$\begin{bmatrix} -x_2^i & -y_2^i & -1 & 0 & 0 & 0x_1^i x_2^i & x_1^i y_2^i & x_1^i \\ 0 & 0 & 0 & -x_2^i & -y_2^i & -1 & y_1^i x_2^i & y_1^i y_2^i & y_1^i \end{bmatrix}\begin{bmatrix} h11 \\ h12 \\ h13 \\ h21 \\ h22 \\ h23 \\ h31 \\ h32 \\ h33 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Hence, the matrix,

$$A_i = \begin{bmatrix} -x_2^i & -y_2^i & -1 & 0 & 0 & 0x_1^i x_2^i & x_1^i y_2^i & x_1^i \\ 0 & 0 & 0 & -x_2^i & -y_2^i & -1 & y_1^i x_2^i & y_1^i y_2^i & y_1^i \end{bmatrix}$$

1.4 Understanding homographies under rotation (5 points)

Suppose that a camera is rotating about its center $\mathbf{C}$, keeping the intrinsic parameters $\mathbf{K}$ constant. Let $\mathbf{H}$ be the homography that maps the view from one camera orientation to the view at a second orientation. Let θ be the angle of rotation between the two. Show that $\mathbf{H^2}$ is the homography corresponding to a rotation of 2θ. Please limit your answer within a couple of lines. A lengthy proof indicates that you're doing something too complicated (or wrong).

Considering that the camera's only being rotated around its center, $\mathbf{H}$ would be of the form

$$\mathbf{H} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Computing, $\mathbf{H}^2$

$$\mathbf{H}^2 = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos^2(\theta) - \sin^2(\theta) & -(2\sin(\\ 2\sin(\theta)\cos(\theta)) & \cos^2(\theta) \\ 0 \end{bmatrix}$$

Therefore, $\mathbf{H}^2$ is the homography corresponding to a rotation of 2θ.

1.5 Limitations of the planar homography (2 points)

Why is the planar homography not completely sufficient to map any arbitrary scene image to another viewpoint? State your answer concisely in one or two sentences.

This expects scene points to lie on a plane which might not always be possible when there's a significant depth involved. Besides that, a translation could lead to an occluded viewpoint from which the object of interest/scene is no longer completely visible.

1.6 Behavior of lines under perspective projections (3 points)

We stated in class that perspective projection preserves lines (a line in 3D is projected to a line in 2D). Verify algebraically that this is the case, i.e., verify that the projection $\mathbf{P}$ in $\mathbf{x} = \mathbf{PX}$ preserves lines.

Consider the 3D line to be given by $(x_0, y_0, z_0) + t(a, b, c)$ where (a, b, c) represent the direction vector, (x_0, y_0, z_0) represents one point lying on the line and $t \in \mathbb{R}$ is the stepping variable that helps satisfy all points lying on this line.

$$\therefore \mathbf{X} = \begin{bmatrix} t * a + x_0 \\ t * b + y_0 \\ t * c + z_0 \end{bmatrix}$$

Also, the projection matrix can be broken down into intrinsic and extrinsic parameter matrices. Assuming that there's no rotation or translation,

$$\mathbf{x} \equiv \alpha \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t * a + x_0 \\ t * b + y_0 \\ t * c + z_0 \end{bmatrix}$$

where α is the scaling factor

$$\mathbf{x} \equiv \alpha \begin{bmatrix} f_x * (t * a + x_0) + o_x * (t * c + z_0) \\ f_y * (t * b + y_0) + o_y * (t * c + z_0) \\ t * c + z_0 \end{bmatrix}$$

Making these homogeneous

$$\mathbf{x} = \alpha \begin{bmatrix} \frac{f_x * (t * a + x_0) + o_x * (t * c + z_0)}{t * c + z_0} \\ \frac{f_y * (t * b + y_0) + o_y * (t * c + z_0)}{t * c + z_0} \\ 1 \end{bmatrix}$$

$$\therefore x = \alpha * f_x * \frac{t * a + x_0}{t * c + z_0} + \alpha * o_x$$
$$\therefore y = \alpha * f_y * \frac{t * b + y_0}{t * c + z_0} + \alpha * o_y$$

Now, consider 2 3-D points lying on the 3-D line - $(x_0, y_0, z_0) + t_1(a, b, c)$ and $(x_0, y_0, z_0) + t_2(a, b, c)$

The corresponding 2D coordinates would be given by

$$\therefore x_1 = \alpha * f_x * \frac{t_1 * a + x_0}{t_1 * c + z_0} + \alpha * o_x$$
$$\therefore y_1 = \alpha * f_y * \frac{t_1 * b + y_0}{t_1 * c + z_0} + \alpha * o_y$$

and

$$\therefore x_2 = \alpha * f_x * \frac{t_2 * a + x_0}{t_2 * c + z_0} + \alpha * o_x$$
$$\therefore y_2 = \alpha * f_y * \frac{t_2 * b + y_0}{t_2 * c + z_0} + \alpha * o_y$$

Finding the slope of the line between these 2 points

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\left[\alpha * f_y * \frac{t_2 * b + y_0}{t_2 * c + z_0} + \alpha * o_y\right] - \left[\alpha * f_y * \frac{t_1 * b + y_0}{t_1 * c + z_0} + \alpha * o_y\right]}{\left[\alpha * f_x * \frac{t_2 * a + x_0}{t_2 * c + z_0} + \alpha * o_x\right] - \left[\alpha * f_x * \frac{t_1 * a + x_0}{t_1 * c + z_0} + \alpha * o_x\right]}$$

Simplifying,

$$m = \frac{\left[f_y * (t_2 * b + y_0)(t_1 * c + z_0)\right] - \left[f_y * (t_1 * b + y_0)(t_2 * c +\right.}{[f_x * (t_2 * a + x_0)(t_1 * c + z_0)] - [f_x * (t_1 * a + x_0)(t_2 * c +}$$

$$= \frac{f_y}{f_x}\left[\frac{t_1 t_2 bc + t_1 cy_0 + t_2 bz_0 + y_0 z_0 - t_1 t_2 bc - t_2 cy_0 - t_1 bz_0 - y_0 z_0}{t_1 t_2 ac + t_1 cx_0 + t_2 az_0 + x_0 z_0 - t_1 t_2 ac - t_2 cx_0 - t_1 az_0 - x_0 z_0}\right] = \frac{f_y}{f_x}\left[\frac{t_1 cy_0 +}{t_1 cx_0 +}\right.$$

$$m = \frac{f_y}{f_x}\left[\frac{cy_0(t_1 - t_2) - bz_0(t_1 - t_2)}{cx_0(t_1 - t_2) - az_0(t_1 - t_2)}\right] = \frac{f_y}{f_x}\frac{(t1 - t2)}{(t1 - t2)}\left[\frac{cy_0 - bz_0}{cx_0 - az_0}\right]$$

$$\therefore m = \frac{f_y}{f_x}\left[\frac{cy_0 - bz_0}{cx_0 - az_0}\right]$$

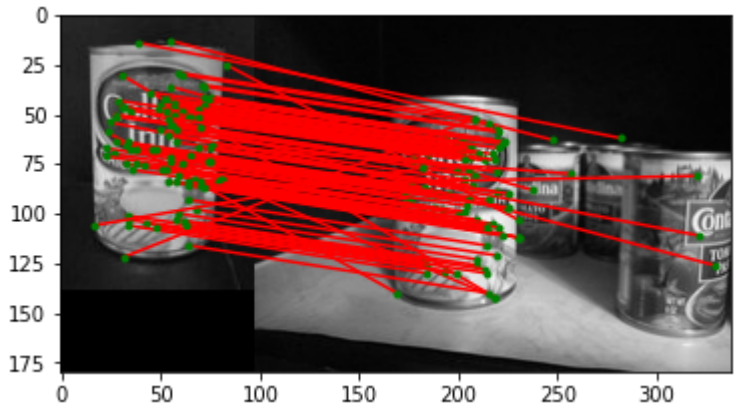
As can be seen, the slope, m, is independent of the variable t. Therefore, for any point lying on the 3D line, it would be projected to this 2D line with slope m using the projection matrix, $\mathbf{P}$.

2.4 Check Point: Descriptor Matching (5 pts)

Save the resulting figure and submit it in your PDF. Briefly discuss any cases that perform worse or better.

In [ ]:
```
def computeBrief(gaussPyramid, locsDoG, compareX, compareY, patch_width=

    left_limit = patch_width // 2
    right_limit_h = gaussPyramid.shape[0] - patch_width // 2
    right_limit_w = gaussPyramid.shape[1] - patch_width // 2

    valid_locsDoG = locsDoG[np.logical_and(np.logical_and(locsDoG[:, 0]
                                                          locsDoG[:, 0]
                                                 np.logical_and(locsDoG[:, 1]
                                                          locsDoG[:, 1]


    desc = []

    for i in range(valid_locsDoG.shape[0]):
        patch = gaussPyramid[valid_locsDoG[i, 0] - patch_width // 2: val
                             valid_locsDoG[i, 1] - patch_width // 2: val
                             valid_locsDoG[i, 2]]
        patch = patch.flatten()
        tau = patch[compareX] < patch[compareY]
        desc.append(list(map(lambda j: str(int(tau[j])), range(len(tau))

    newlocs = valid_locsDoG[:, -2: -4: -1]
    desc = np.stack(desc)

    return newlocs, desc

def briefLite(im):
    locsDoG, gauss_pyramid = DoGdetector(im)
    [compareX, compareY] = np.load('data/testPattern.npy')

    locs, desc = computeBrief(gauss_pyramid, locsDoG, compareX, compareY

    return locs, desc

def briefMatch(desc1, desc2, ratio=0.8):
    D = cdist(np.float32(desc1), np.float32(desc2), metric='hamming')
    # find the smallest distance
    ix2 = np.argmin(D, axis=1)
    d1 = D.min(1)
    # find the second smallest distance
    d12 = np.partition(D, 2, axis=1)[:,0:2]
    d2 = d12.max(1)
    r = d1/(d2+1e-10)
    is_discr = r<ratio
    ix2 = ix2[is_discr]
    ix1 = np.arange(D.shape[0])[is_discr]
    matches = np.stack((ix1,ix2), axis=-1)
    return matches

def plotMatches(im1, im2, matches, locs1, locs2):
    fig = plt.figure()
    # draw two images side by side
    imH = max(im1.shape[0], im2.shape[0])
    im = np.zeros((imH, im1.shape[1]+im2.shape[1]), dtype='uint8')
    im[0:im1.shape[0], 0:im1.shape[1]] = cv2.cvtColor(im1, cv2.COLOR_BGR
    im[0:im2.shape[0], im1.shape[1]:] = cv2.cvtColor(im2, cv2.COLOR_BGR2
```

```
    plt.imshow(im, cmap='gray')
    for i in range(matches.shape[0]):
        pt1 = locs1[matches[i,0], 0:2]
        pt2 = locs2[matches[i,1], 0:2].copy()
        pt2[0] += im1.shape[1]
        x = np.asarray([pt1[0], pt2[0]])
        y = np.asarray([pt1[1], pt2[1]])
        plt.plot(x,y,'r')
        plt.plot(x,y,'g.')
    plt.show()

img1 = cv2.imread('data/model_chickenbroth.jpg')
locs1, desc1 = briefLite(img1)

img2 = cv2.imread('data/chickenbroth_01.jpg')
locs2, desc2 = briefLite(img2)

matches = briefMatch(desc1, desc2)
plotMatches(img1, img2, matches, locs1, locs2)
```

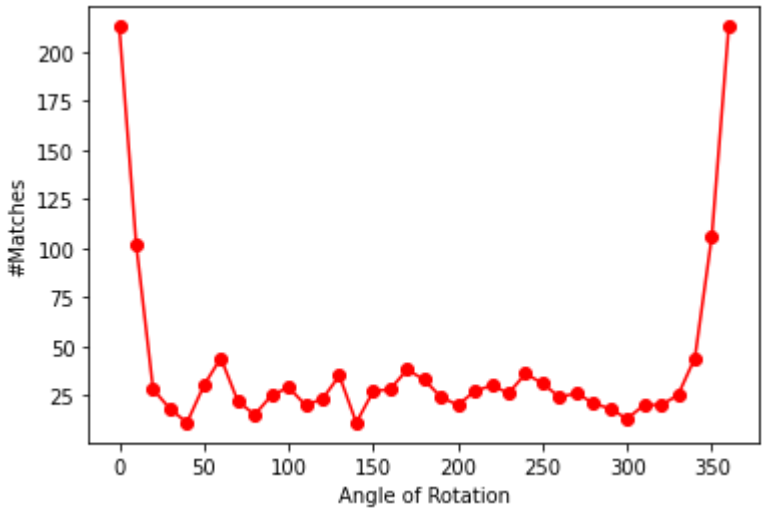


BRIEF descriptors are not invariant to scale and rotational changes. So, in case of same image being rotated or upscaled/downscaled, the descriptors generated between these two images wouldn't match.

2.5 BRIEF and rotations (5 pts)

Include your code and the historgram figure in your PDF, and explain why you think the descriptor behaves this way.

In [ ]:
```
img1 = cv2.imread('data/model_chickenbroth.jpg')
locs1, desc1 = briefLite(img1)

img2 = cv2.imread('data/model_chickenbroth.jpg')
h, w, _ = img2.shape

counts = []

for i in range(0, 370, 10):

    # could've just used this matrix again and again. However, the img2
    # so, matches are reducing.
    M = cv2.getRotationMatrix2D((w / 2, h / 2), i, 1.0)
    rotated = cv2.warpAffine(img2, M, (w, h))
    locs2, desc2 = briefLite(rotated)
    matches = briefMatch(desc1, desc2)

    counts.append(len(matches))

plt.plot(list(range(0, 370, 10)), counts, 'ro-')
plt.xlabel('Angle of Rotation')
plt.ylabel('#Matches')
plt.show()
```



The descriptor works by looking at n pairs of pixel in a patch around a keypoint. However, because of rotation, the pixel locations have locally changed within the patch. Thus, the n pairs are no longer referring to the same pixels as the ones in the unrotated image.

2.6 Improving Performance - (Extra Credit, 10 pts)

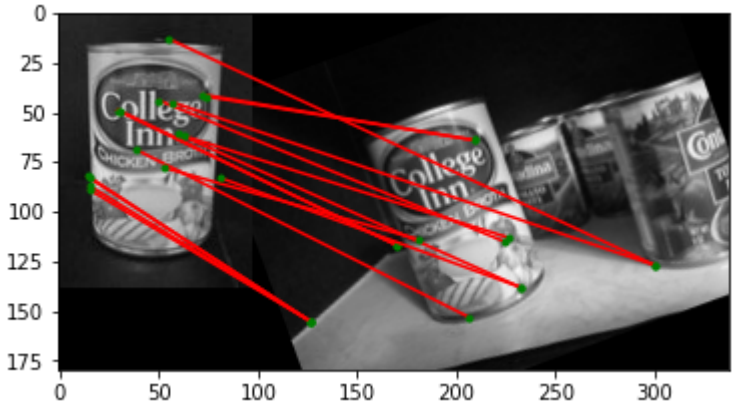
The extra credit opportunities described below are optional and provide an avenue to explore computer vision and improve the performance of the techniques developed above.

1. (**5 pts**) As we have seen, BRIEF is not rotation invariant. Design a simple fix to solve this problem using the tools you have developed so far (think back to edge detection and/or Harris corner's covariance matrix). Include yout code in your PDF, and explain your design decisions and how you selected any parameters that you use. Demonstrate the effectiveness of your algorithm on image pairs related by large rotation.
2. (**5 pts**) This implementation of BRIEF has some scale invariance, but there are limits. What happens when you match a picture to the same picture at half the size? Look to section 3 of Lowe2004 (https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf) for a technique that will make your detector more robust to changes in scale. Implement it and demonstrate it in action with several test images. Include yout code and the test images in your PDF. You may simply rescale some of the test images we have given you.

In [ ]:
```
def computeRBrief(gaussPyramid, locsDoG, compareX, compareY, patch_width

    h, w, _ = gaussPyramid.shape
    left_limit = patch_width // 2
    right_limit_h = gaussPyramid.shape[0] - patch_width // 2
    right_limit_w = gaussPyramid.shape[1] - patch_width // 2

    valid_locsDoG = locsDoG[np.logical_and(np.logical_and(locsDoG[:, 0]
                                                          locsDoG[:, 0]
                                           np.logical_and(locsDoG[:, 1]
                                                          locsDoG[:, 1]


    desc = []

    for i in range(valid_locsDoG.shape[0]):
        patch = gaussPyramid[valid_locsDoG[i, 0] - patch_width // 2: val
                             valid_locsDoG[i, 1] - patch_width // 2: val
                             valid_locsDoG[i, 2]]

#         patch = patch.flatten()
        m10 = np.sum(np.repeat(np.arange(valid_locsDoG[i, 0] - patch_wid
                                         valid_locsDoG[i, 0] + patch_wid
                               patch_width, axis=0) * patch)
        m01 = np.sum(np.repeat(np.arange(valid_locsDoG[i, 1] - patch_wid
                                         valid_locsDoG[i, 1] + patch_wid
                               patch_width, axis=0) * patch)
        m00 = np.sum(patch)
        theta = - np.arctan2(m01, m10)
        R = np.array([[np.cos(theta), - np.sin(theta)],
                      [np.sin(theta), np.cos(theta)]])

#         print(R.shape)
#         print(np.stack((compareX.flatten(), compareY.flatten())).shape

        compareMat = np.int32(R @ np.stack((compareX.flatten(), compareY

        tau = gaussPyramid[(int(valid_locsDoG[i, 0] - patch_width // 2)
                           (int(valid_locsDoG[i, 1] - patch_width // 2)
                           valid_locsDoG[i, 2]] < \
              gaussPyramid[(int(valid_locsDoG[i, 0] - patch_width // 2)
                           (int(valid_locsDoG[i, 1] - patch_width // 2)
                           valid_locsDoG[i, 2]]
        desc.append(list(map(lambda j: str(int(tau[j])), range(len(tau))

    newlocs = valid_locsDoG[:, -2: -4: -1]
    desc = np.stack(desc)

    return newlocs, desc

def rBriefLite(im):
    '''
    Given an image, detect the keypoints and describe the keypoints with

    INPUTS
        im - gray image with values between 0 and 1
    OUTPUTS
```

```
        locs - an m x 3 vector, where the first two columns are the imag
               of keypoints and the third column is the pyramid level c
        desc - an m x n bits matrix of stacked BRIEF descriptors.
               m is the number of valid descriptors in the image and wi
               n is the number of bits for the BRIEF descriptor
    '''
    # Hint: Use the provided "DoGdetector()" obtain the smoothed Gaussia
    # the keypoints, and use them as the input of "computeBrief()"

    locsDoG, gauss_pyramid = DoGdetector(im)
    [compareX, compareY] = np.load('data/testPattern.npy')

    locs, desc = computeRBrief(gauss_pyramid, locsDoG, compareX, compare

    return locs, desc

img1 = cv2.imread('data/model_chickenbroth.jpg')
# img1 = cv2.imread('data/chickenbroth_01.jpg')
locs1, desc1 = rBriefLite(img1)

img2 = cv2.imread('data/chickenbroth_01.jpg')

h, w, _ = img2.shape

M = cv2.getRotationMatrix2D((w / 2, h / 2), 20, 1.0)
rotated = cv2.warpAffine(img2, M, (w, h))
locs2, desc2 = rBriefLite(rotated)

matches = briefMatch(desc1, desc2)
plotMatches(img1, rotated, matches, locs1, locs2)
```

Result:



The idea to solve for rotation invariance was something similar to what ORB descriptor does. Took the patch and found its rotation and found the new points to compare by rotating with the same angle in the opposite direction. This would help resolve the patch rotation issue.

A fix for scale invariance would be to use a guassian pyramid. By having the gaussian pyramid of an image and comparing between combinations of the other image and different pyramid levels would lead to a scale invariant system.

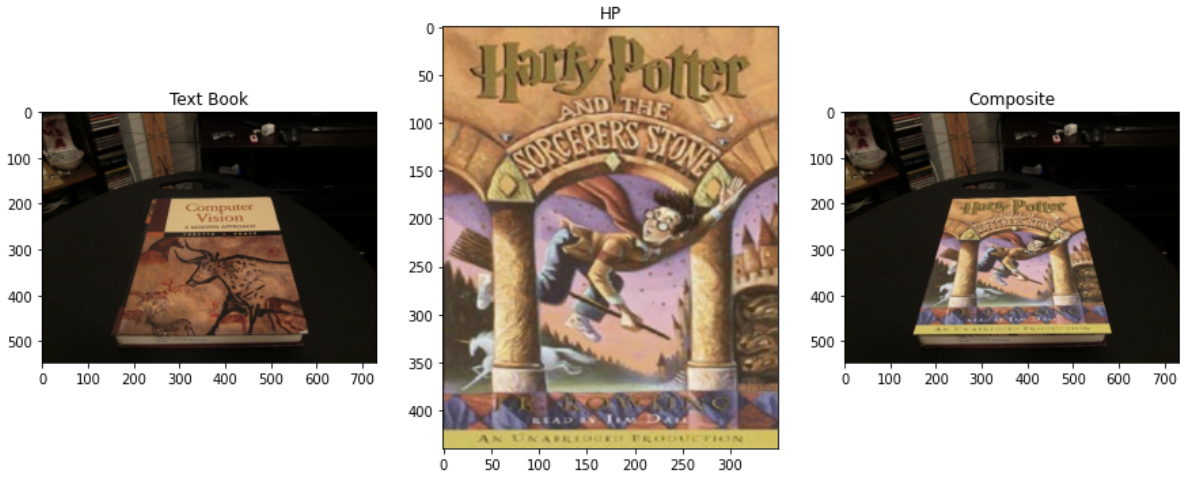
3.3 Automated Homography Estimation/Warping for Augmented Reality (10 points)

Implement the following steps:

1. Reads `cv-cover.jpg`, `cv-desk.png`, and `hp-cover.jpg`.
2. Computes a homography automatically using `computeH-ransac`.
3. Warps `hp-cover.jpg` to the dimensions of the `cv-desk.png` image using the OpenCV `warpPerspective` function.
4. At this point you should notice that although the image is being warped to the correct location, it is not filling up the same space as the book. Why do you think this is happening? How would you modify `hp-cover.jpg` to fix this issue?
5. Implement the function:
   `function [ composite-img ] = compositeH( H2to1, template, img)` to now compose this warped image with the desk image as in the following figures.
6. Include your resulting image in your write-up. Please also print the final H matrix in your writeup (normalized so the bottom right value is 1)

In [ ]:
```
def compositeH(H, template, img):

    warped_img = cv2.warpPerspective(template, np.linalg.inv(H), img.sha
    composite_img = np.uint8(warped_img == 0) * img + warped_img

    return composite_img.astype(np.uint8)

im1 = cv2.cvtColor(cv2.imread("figure/cv_cover.jpg"), cv2.COLOR_BGR2RGB)
im2 = cv2.cvtColor(cv2.imread("figure/cv_desk.png"), cv2.COLOR_BGR2RGB)

locs1, desc1 = briefLite(im1)
locs2, desc2 = briefLite(im2)
matches = briefMatch(desc1, desc2)

H, inliers = computeH_ransac(matches, locs1, locs2)

print('H: {0}'.format(H / H[2, 2]))

template = cv2.cvtColor(cv2.imread("figure/hp_cover.jpg"), cv2.COLOR_BGR

# Resizing image to meet the im1 shape so that homography matrix works p
resized_template = cv2.resize(template, im1.shape[-2: -4: -1])

composite_img = compositeH(H, resized_template, im2)

fig, axes = plt.subplots(1, 3, figsize=(15, 15))

axes[0].imshow(im2)
axes[0].set_title('Text Book')
axes[1].imshow(resized_template)
axes[1].set_title('Resized HP')
axes[2].imshow(composite_img)
axes[2].set_title('Composite')
plt.show()
```

$$H = \begin{bmatrix} 2.35252906e+00 & 6.88051228e-01 & -6.86719269e+02 \\ 1.01883048e-01 & 4.25832288e+00 & -8.42107269e+02 \\ 1.96661190e-04 & 3.70549187e-03 & 1.00000000e+00 \end{bmatrix}$$



4.1 Image Stitching (5 pts)

Visualize the warped image. Please include the image and your H2to1 matrix (with the bottom right index as 1) in your writeup PDF, along with stating which image pair you used.

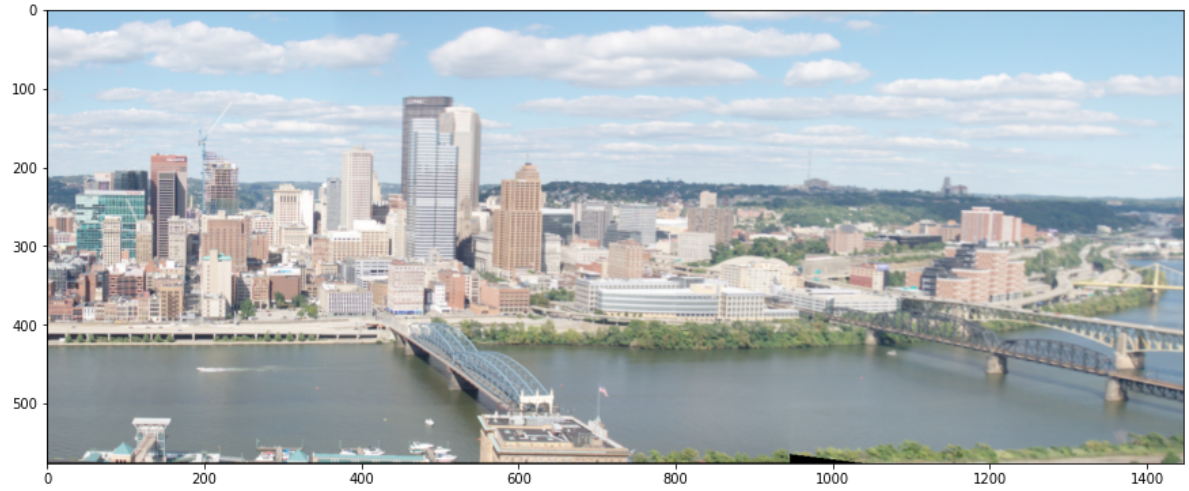
In [ ]:
```
def imageStitching(im1, im2, H2to1, pad=((0, 0), (0, 500), (0, 0))):
    '''
    Returns a panorama of im1 and im2 using the given
    homography matrix

    INPUT
        Warps img2 into img1 reference frame using the provided warpH()
        H2to1 - a 3 x 3 matrix encoding the homography that best matches
                equation.
    OUTPUT
        img_pano - the panorama image.
    '''

    padded_im1 = np.pad(im1, pad)
    warped_img = cv2.warpPerspective(im2, H2to1, padded_im1.shape[-2: -4

    # Generating weights for alpha blending.
    w1 = distance_transform_edt(np.pad(np.ones_like(im1[1: -1, 1: -1, 0]
    w2 = distance_transform_edt(np.pad(np.ones_like(im2[1: -1, 1: -1, 0]

    # Padding w1 as required to maintain right weight distribution.
    padded_w1 = np.pad(w1, pad[0: 2])
    padded_w1 = padded_w1.reshape(padded_w1.shape[0], -1, 1)

    # Warping weights for im2 to the target shape.
    warped_w2 = cv2.warpPerspective(w2, H2to1, padded_im1.shape[-2: -4:
    warped_w2 = warped_w2.reshape(warped_w2.shape[0], -1, 1)

    img_pano = np.uint8((padded_im1 * padded_w1 + warped_img * warped_w2

    return img_pano
```

$$H2to1 = \begin{bmatrix} 6.50985010e-01 & -4.40168516e-02 & 3.65903823e+02 \\ -7.99527543e-02 & 8.72003506e-01 & -1.59370556e+01 \\ -3.55709534e-04 & -1.73167455e-05 & 1.00000000e+00 \end{bmatrix}$$

Image1

Image2

Stitched Image



4.2 Image Stitching with No Clip (3 pts)

Visualize the warped image. Please include the image in your writeup PDF, along with stating which image pair you used.

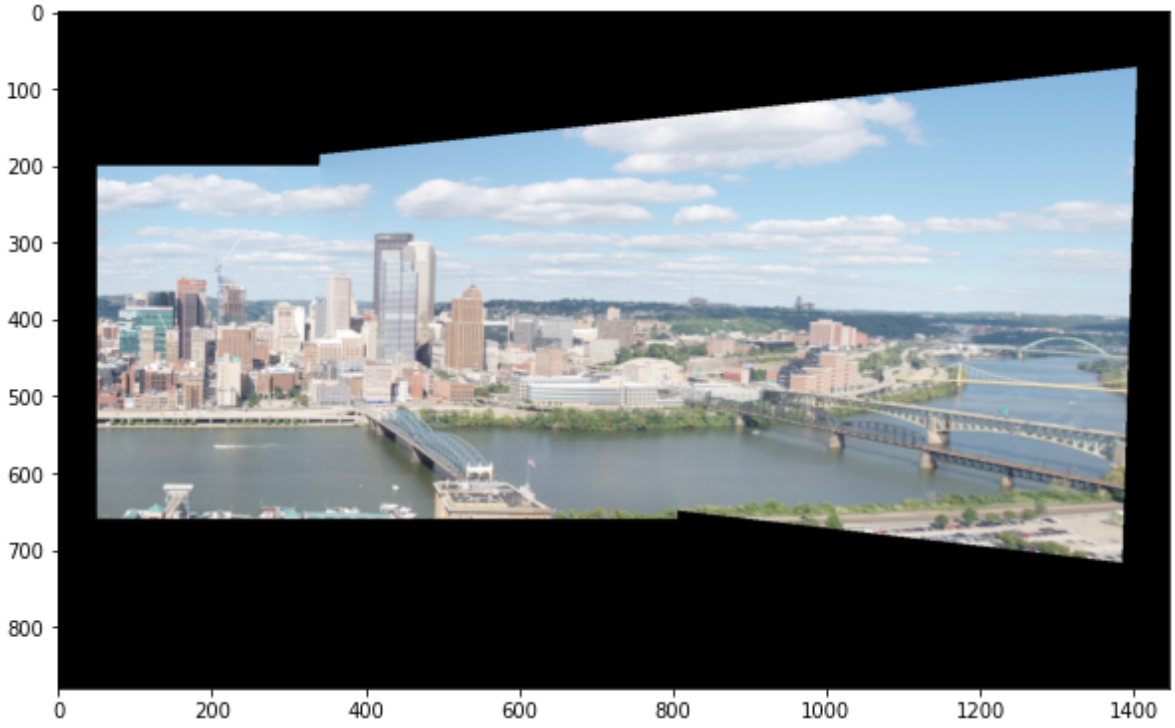
In [ ]:
```
def imageStitching_noClip(im1,
                          im2,
                          H2to1,
                          pad=500,
                          isPadWidth=True,
                          M=np.array([[0.8, 0, 50],
                                      [0, 0.8, 200],
                                      [0, 0, 1.]])):
    '''
    Returns a panorama of im1 and im2 using the given
    homography matrix without cliping.

    INPUTS
        im1 and im2 - images to be stitched.
        H2to1- the homography matrix.
        pad - Number of pixels to increase width by. Default 500
        M - scaling and translation matrix.
    OUTPUT
        img_pano - the panorama image.
    '''

    if isPadWidth:
        w, h = im1.shape[1] + pad, (im1.shape[0] * (im1.shape[1] + pad))
    else:
        w, h = (im1.shape[1] * (im1.shape[0] + pad)) // im1.shape[0], im

    out_size = (w, h)

    warp_im1 = cv2.warpPerspective(im1, M, out_size)
    warp_im2 = cv2.warpPerspective(im2, M @ H2to1, out_size)

    # Generating weights for alpha blending.
    w1 = distance_transform_edt(np.pad(np.ones_like(im1[1: -1, 1: -1, 0]
    w2 = distance_transform_edt(np.pad(np.ones_like(im2[1: -1, 1: -1, 0]

    # Warping weights for im1 to the target shape.
    warp_w1 = cv2.warpPerspective(w1, M, out_size)
    warp_w1 = warp_w1.reshape(warp_w1.shape[0], -1, 1)

    # Warping weights for im2 to the target shape.
    warp_w2 = cv2.warpPerspective(w2, M @ H2to1, out_size)
    warp_w2 = warp_w2.reshape(warp_w2.shape[0], -1, 1)

    img_pano = np.uint8((warp_im1 * warp_w1 + warp_im2 * warp_w2) / (war

    return img_pano

im1 = cv2.cvtColor(cv2.imread('data/incline_L.png'), cv2.COLOR_BGR2RGB)
im2 = cv2.cvtColor(cv2.imread('data/incline_R.png'), cv2.COLOR_BGR2RGB)

locs1, desc1 = briefLite(im1)
locs2, desc2 = briefLite(im2)
matches = briefMatch(desc1, desc2)

H, inliers = computeH_ransac(matches, locs1, locs2)
print('H: {0}'.format(H / H[2, 2]))
```

```
img_pano = imageStitching_noClip(im1, im2, H)

plt.rcParams['figure.figsize'] = (10, 10)
plt.imshow(img_pano)
plt.show()
```

Image1

Image2

Stitched Image



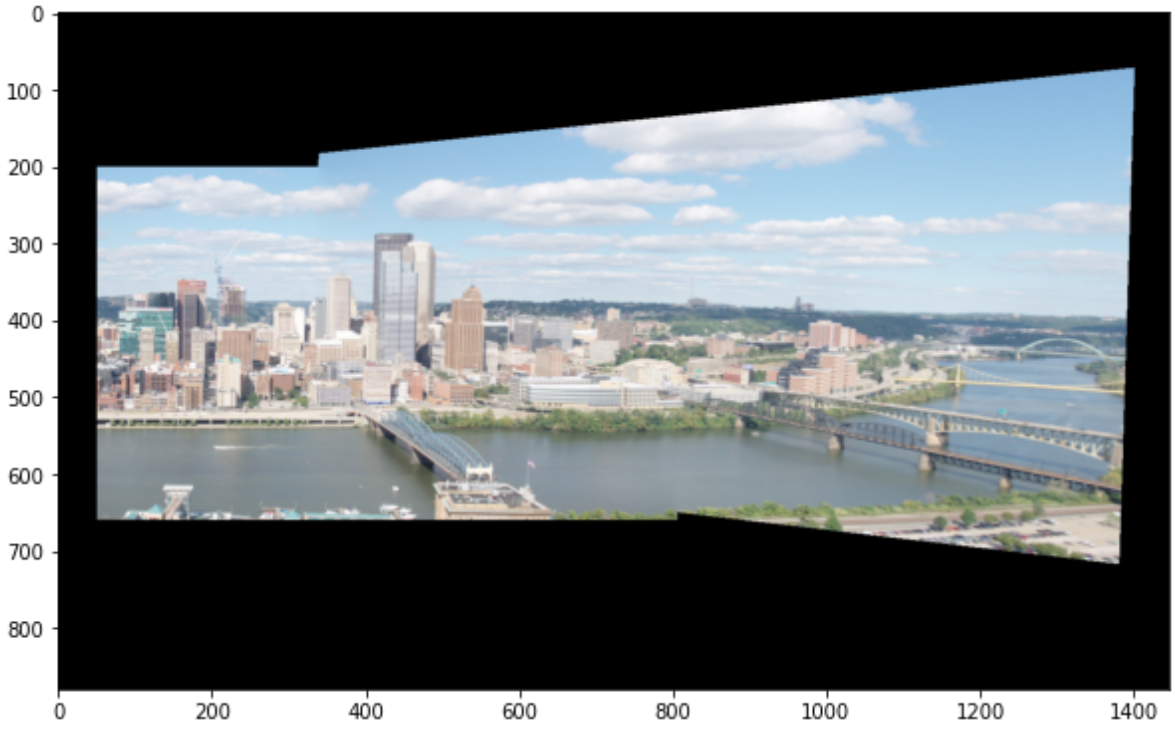
4.3 Generate Panorama (2 pts)

Save the resulting panorama on the full sized images and include the figure and computed homography matrix in your writeup.

In [ ]:
```
def generatePanorama(im1, im2):
    '''
    Gnerate a panorama from two images.

    INPUTS
        im1 and im2 - images to be stitched.
    OUTPUT
        img_pano - the panorama image.
    '''
    im1 = cv2.cvtColor(im1, cv2.COLOR_BGR2RGB)
    im2 = cv2.cvtColor(im2, cv2.COLOR_BGR2RGB)

    locs1, desc1 = briefLite(im1)
    locs2, desc2 = briefLite(im2)
    matches = briefMatch(desc1, desc2)
    H, inliers = computeH_ransac(matches, locs1, locs2)
    print('H: {0}'.format(H / H[2, 2]))


    img_pano = imageStitching_noClip(im1, im2, H)

    return img_pano

im1 = cv2.imread('data/incline_L.png')
im2 = cv2.imread('data/incline_R.png')

img_pano = generatePanorama(im1, im2)

plt.rcParams['figure.figsize'] = (10, 10)
plt.imshow(img_pano)
plt.show()
```

$$H2to1 = \begin{bmatrix} 6.70688040e-01 & -3.24766846e-02 & 3.61588728e+02 \\ -7.62290199e-02 & 8.90118506e-01 & -2.09211372e+01 \\ -3.42501786e-04 & -2.92305590e-06 & 1.00000000e+00 \end{bmatrix}$$

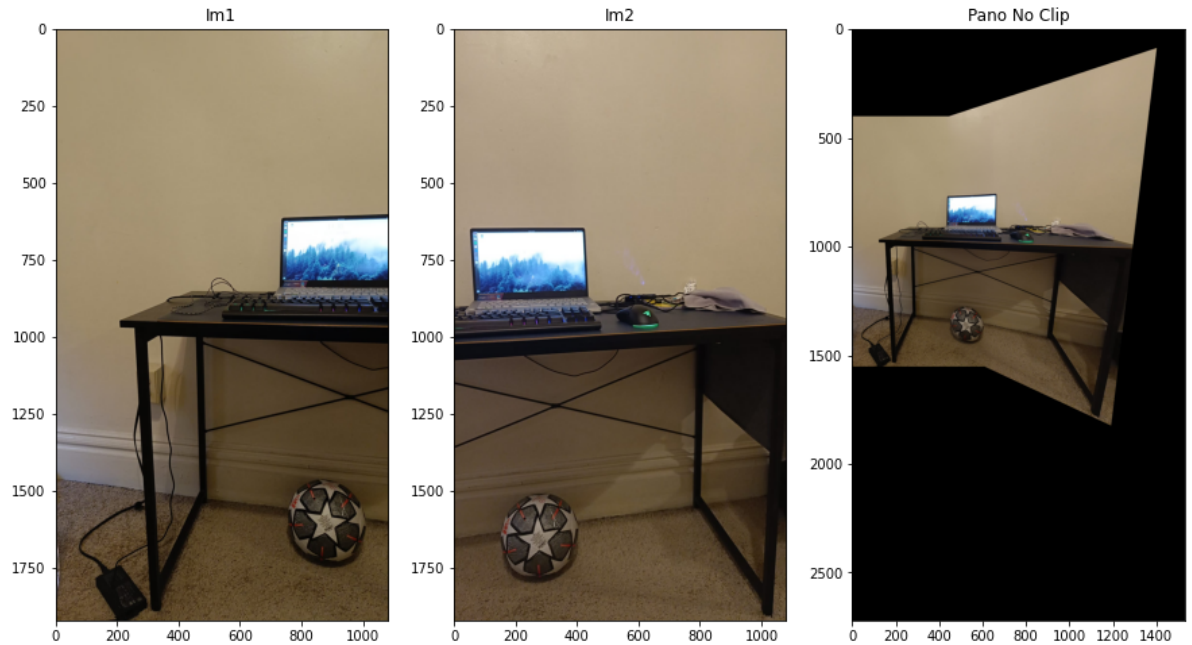
Panorama



4.4 extra credits (3 pts)

Collect a pair of your own images (with your phone) and stitch them together using your code from the previous section. Include the pair of images and their result in the write-up.

In [ ]:
```
im1 = cv2.imread('data/custom_l.jpeg')
im2 = cv2.imread('data/custom_r.jpeg')

im1 = cv2.cvtColor(im1, cv2.COLOR_BGR2RGB)
im2 = cv2.cvtColor(im2, cv2.COLOR_BGR2RGB)

locs1, desc1 = briefLite(im1)
locs2, desc2 = briefLite(im2)
matches = briefMatch(desc1, desc2)

# plotMatches(im1, im2, matches, locs1, locs2)

H, inliers = computeH_ransac(matches, locs1, locs2)
print('H: {0}'.format(H / H[2, 2]))

M = np.array([[0.6, 0, 0],
              [0, 0.6, 400],
              [0, 0, 1.]])

img_pano = imageStitching_noClip(im1, im2, H, pad=800, isPadWidth=False,

fig, axes = plt.subplots(1, 3, figsize=(15, 15))

axes[0].imshow(im1)
axes[0].set_title('Im1')
axes[1].imshow(im2)
axes[1].set_title('Im2')
axes[2].imshow(img_pano)
axes[2].set_title('Pano No Clip')
plt.show()
```



4.5 extra credits (2 pts)

Collect at least 6 images and stitch them into a single noClip image. You can either collect your own, or use the PNC Park images (http://www.cs.jhu.edu/~misha/Code/SMG/PNC3.zip) from Matt Uyttendaele. We used the PNC park images (subsmapled to 1/4 sized) and ORB keypoints and descriptors for our reference solution.

YOUR ANSWER HERE...

Question 5: Poisson Image Stitching (15 points)

Write a function called `poisson-blend(background,foreground,mask)` which takes 3 equal sized images (background and foreground as RGB, mask as binary) and solves the Poisson equation, using gradients from foreground and boundary conditions from the background.

The problem will be manually graded. Please include results from both the `(fg1,bg1,mask1)` and `(fg2,bg2,mask2)` images in your write-up.

In [ ]:
```
def is_border(mask, x, y, c):

    if (mask[x - 1, y, c] == 0) or (mask[x + 1, y, c] == 0) or \
        (mask[x, y - 1, c] == 0) or (mask[x, y + 1, c] == 0):
        return True

    return False

def poisson_blend(bg, fg, mask):

    blended_img = np.uint8(np.uint8(mask == 0) * bg + np.uint8(mask != 0
    plt.imshow(blended_img)
    plt.show()

    mask_nzx, mask_nzy = np.nonzero(mask[:, :, 0])
    mask_nz = [(int(x), int(y)) for x, y in zip(mask_nzx, mask_nzy)]
    N = len(mask_nz)

    A = np.eye(N, dtype=np.float16) * 4

    # Iterate through every non-zero pixel and fill A.
    for p in range(N):

        points = [(mask_nz[p][0] - 1, mask_nz[p][1]),
                  (mask_nz[p][0] + 1, mask_nz[p][1]),
                  (mask_nz[p][0], mask_nz[p][1] - 1),
                  (mask_nz[p][0], mask_nz[p][1] + 1)]

        for x, y in points:
            if (x, y) not in mask_nz:
                continue

            A[p, mask_nz.index((x, y))] = -1

    # Solve Ax=b for each channel.
    for c in range(blended_img.shape[-1]):
        b = np.zeros(N)

        for j in range(N):
            b[j] = 4 * fg[mask_nz[j][0], mask_nz[j][1], c] \
                - fg[mask_nz[j][0] - 1, mask_nz[j][1], c] \
                - fg[mask_nz[j][0] + 1, mask_nz[j][1], c] \
                - fg[mask_nz[j][0], mask_nz[j][1] - 1, c] \
                - fg[mask_nz[j][0], mask_nz[j][1] + 1, c]

            if (is_border(mask, mask_nz[j][0], mask_nz[j][1], c)):
                if (mask[mask_nz[j][0] - 1, mask_nz[j][1], c] == 0):
                    b[j] += bg[mask_nz[j][0] - 1, mask_nz[j][1], c]
                else:
                    b[j] += fg[mask_nz[j][0] - 1, mask_nz[j][1], c]

                if (mask[mask_nz[j][0] + 1, mask_nz[j][1], c] == 0):
                    b[j] += bg[mask_nz[j][0] + 1, mask_nz[j][1], c]
                else:
                    b[j] += fg[mask_nz[j][0] + 1, mask_nz[j][1], c]
```

```
                if (mask[mask_nz[j][0], mask_nz[j][1] - 1, c] == 0):
                    b[j] += bg[mask_nz[j][0], mask_nz[j][1] - 1, c]
                else:
                    b[j] += fg[mask_nz[j][0], mask_nz[j][1] - 1, c]

                if (mask[mask_nz[j][0], mask_nz[j][1] + 1, c] == 0):
                    b[j] += bg[mask_nz[j][0], mask_nz[j][1] + 1, c]
                else:
                    b[j] += fg[mask_nz[j][0], mask_nz[j][1] + 1, c]

        i = np.linalg.solve(A, b)
        i = (i - np.min(i)) * 255. / (np.max(i) - np.min(i))
        blended_img[mask_nzx, mask_nzy, c] = i

    return np.uint8(blended_img)
```

FG:

BG:

Mask:

Result:

While the resulting image does not look right, I feel the algorith implemented is right. So, if you could help figure out what I am doing wrong, that would help a lot.

In [ ]: