CS 6476: Computer Vision, Fall 2019

PS3

Instructor: Devi Parikh

Due: Wednesday, October 8th, 11:59 pm

Instructions

Implement all of the functions described in this notebook. Then, apply the functions to generate mosaics of the provided images and some of your own.

When you are done save this notebook with the images and mosaics clearly visible.

Submit the notebook and other deliverables on Gradescope in the `PS3 Code` assignment (see the checklist below).

1 Programming: Image Mosaics [100 points]

In this exercise, you will implement an image stitcher that uses image warping and homographies to auto- matically create an image mosaic. We will focus on the case where we have two input images that should form the mosaic, where we warp one image into the plane of the second image and display the combined views. This problem will give some practice manipulating homogeneous coordinates, computing homography matrices, and performing image warps. For simplicity, we'll specify corresponding pairs of points manually using mouse clicks. For extra credit, you can optionally implement an automated correspondence process with local feature matching.

Python hints:

1. Useful Modules: `numpy`, `scipy`, `imageio`, `matplotlib`
2. Useful Functions: `numpy.linalg.eig`, `numpy.linalg.inv`, `numpy.tile`, `numpy.meshgrid`
3. There are some Python libraries that could do much of the work for this project.

However, to get practice with how the algorithms work, we want you to write your own code.

1. Getting Correspondences

Implement the `pick_points` function in `pick-points.py` (included in the zip). The function should allow a user to manually identified corresponding points from two views. Hint: look up **matplotlib event handling** and the **matplotlib.widgets.Cursor** class.

The results will be sensitive to the accuracy of the corresponding points; when providing clicks, choose distinctive points in the image that appear in both views.

To use `pick-points.py` run:

```
# general usage:
python3 pick-points.py <path/to/image1.jpg> <path/to/ima
ge2.jpg>

# for example
python3 pick-points.py wdc1.jpg wdc2.jpg
```

The selected points will be saved to `<path/to/image1.npy>` and `<path/to/image2.npy>` .

Include `pick-points.py` in your submission.

Imports

In [31]:
```
import imageio
import numpy as np
import matplotlib.pyplot as plt
##########################################################################
# TODO: Add additional imports
##########################################################################
import cv2
##########################################################################
#                                 END OF YOUR CODE
##########################################################################
```

2. Computing the Homography Parameters [20 points]

Implement the `compute_homography(t1, t2)` function as described below.

Be sure to handle homogenous and non-homogenous coordinates correctly. Look at

the notes on how to estimate a homography here (https://gatech.box.com/shared/static/yl4t92swxn4ffa928cec4lfz22csk86t.pdf).

> Note: Your estimation procedure may perform better if image coordinates range from 0 to 2. Consider scaling your measurements to avoid numerical issues.

Manually export your function into a file named `compute_homography.py`. Add (only) the required imports and submit this file.

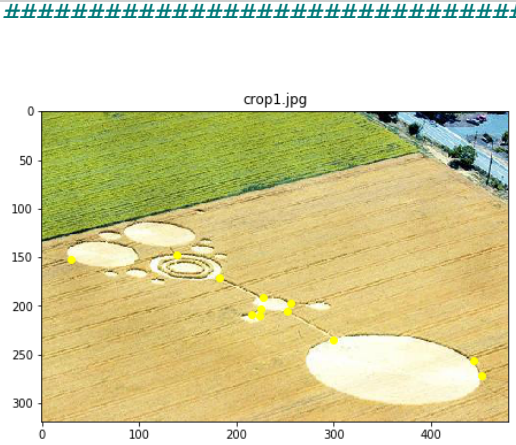
In [32]:
```
def compute_homography(t1, t2):
    """
    Computes the Homography matrix for corresponding image point

    The function should take a list of N ≥ 4 pairs of correspond
    from the two views, where each point is specified with its 2
    coordinates.

    Inputs:
    - t1: Nx2 matrix of image points from view 1
    - t2: Nx2 matrix of image points from view 2

    Returns a tuple of:
    - H: 3x3 Homography matrix
    """
    H = np.eye(3)
    ###########################################################
    # TODO: Compute the Homography matrix H for corresponding im
    ###########################################################
    max_val = max(t1.max(), t2.max())
    mats = []
    for i in range(t1.shape[0]):
        x = t1[i][0]
        y = t1[i][1]
        xp = t2[i][0]
        yp = t2[i][1]
        mat = np.array([
            [x, y, 1, 0, 0, 0, -x * xp, -y * xp, -xp],
            [0, 0, 0, x, y, 1, -x * yp, -y * yp, -yp]
        ])
        mats.append(mat)
    L = np.vstack(mats)
    eig_vals, eig_vecs = np.linalg.eig(L.T.dot(L))
    H = eig_vecs[:, np.argmin(eig_vals)].reshape((3, 3))
    ###########################################################
    #                                 END OF YOUR CODE
    ###########################################################
    return H
```

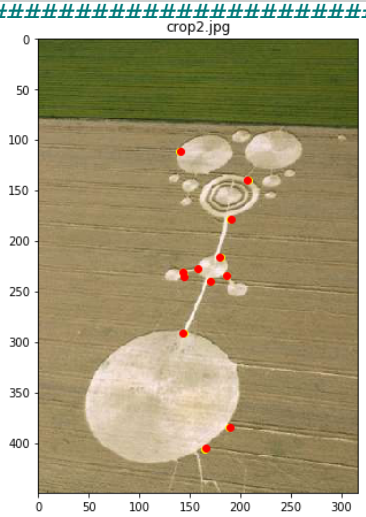
In [ ]:

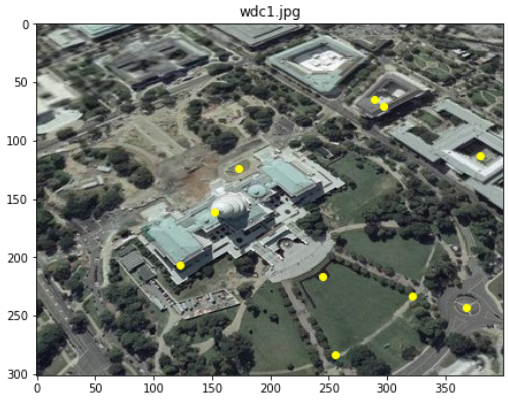
Verify that the homography matrix your function computed is correct by mapping the clicked image points from one view to the other, and displaying them on top of each respective image.

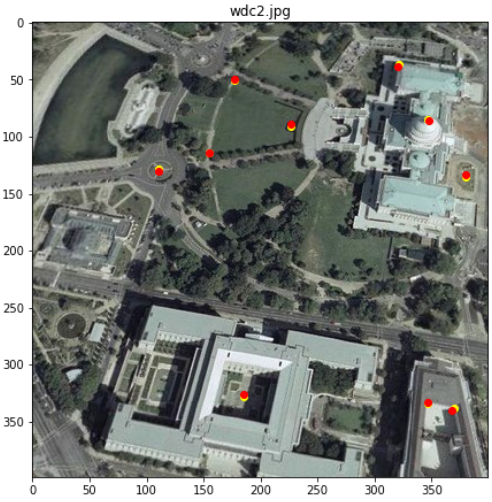
In [33]:
```
######################################################################
# TODO: Verify Homography matrix
######################################################################

# crop.jpg result

t1 = np.load('cc1.npy')
t2 = np.load('cc2.npy')
p = np.hstack([t1, np.ones((t1.shape[0], 1))]).T
q = np.hstack([t2, np.ones((t2.shape[0], 1))]).T
H = compute_homography(t1, t2)
H_dot_p = H.dot(p)
q_mapped = H_dot_p / H_dot_p[-1, :]

im1 = np.array(imageio.imread('crop1.jpg'))
im2 = np.array(imageio.imread('crop2.jpg'))
fig, ax = plt.subplots(2, 2, figsize=(16, 16))

ax[0, 0].imshow(im1)
ax[0, 0].set_title('crop1.jpg')
ax[0, 0].scatter(t1[:, 0], t1[:, 1], color='yellow')

ax[0, 1].imshow(im2)
ax[0, 1].set_title('crop2.jpg')
ax[0, 1].scatter(t2[:, 0], t2[:, 1], color='yellow')
ax[0, 1].scatter(q_mapped[0, :], q_mapped[1, :], color='red')

#### wdc.jpg result

t1 = np.load('wdc1.npy')
t2 = np.load('wdc2.npy')
p = np.hstack([t1, np.ones((t1.shape[0], 1))]).T
q = np.hstack([t2, np.ones((t2.shape[0], 1))]).T
H = compute_homography(t1, t2)
H_dot_p = H.dot(p)
q_mapped = H_dot_p / H_dot_p[-1, :]

im1 = np.array(imageio.imread('wdc1.jpg'))
im2 = np.array(imageio.imread('wdc2.jpg'))

ax[1, 0].imshow(im1)
ax[1, 0].set_title('wdc1.jpg')
ax[1, 0].scatter(t1[:, 0], t1[:, 1], color='yellow')
```

```
ax[1, 1].imshow(im2)
ax[1, 1].set_title('wdc2.jpg')
ax[1, 1].scatter(t2[:, 0], t2[:, 1], color='yellow')
ax[1, 1].scatter(q_mapped[0, :], q_mapped[1, :], color='red')

plt.show()

##########################################################################
#                                END OF YOUR CODE
##########################################################################
```









From the above figures we can see the points were corrected mapped from the first image to the second image through homogeneous coordinates transformation. The ground truth points were originally painted in yellow dots, and later completed covered by the computed points painted in red.

3. Warping Between Image Planes [30 points]

Write a function `warp_image, merge_image = warp_image(input_image, ref_image, H)` which takes as input an image `input_image`, a reference image `ref_image`, and a 3x3 homography matrix `H`, and returns 2 images as outputs. The first output image, `warp_image`, should be the the input image `input_image` warped according to `H` to be in the frame of the reference image `ref_image`. The second output image, `merge_image`, should be a single mosaic image with a larger field of view containing both the input images. *Note: the output images will have a different shape than the input images.*

To avoid holes, use an *inverse warp*. Calculate the bounding box in the reference frame of the destination image by warping all of the points from the source image into the reference frame of the destination. Then, sample all of the points in that bounding box from the proper coordinates in the source image.

Once you have the input image warped into the reference image's frame of reference, create a merged image showing the mosaic. Create a new image large enough to hold both the views; overlay one view onto the other, simply leaving it black wherever no data is available. Don't worry about artifacts that result at the boundaries.

In [34]:
```
def warp_image(input_image, ref_image, H):
    """
    Warps and merges an input image onto the reference image.

    Inputs:
    - input_image: input image
    - ref_image: reference image
    - H: 3x3 homography matrix

    Returns a tuple of:
    - warp_image: The input image warped according to H.
    - merge_image: A single mosaic image containing both of the
    """
    warp_image, merge_image = None, None
    ###############################################################
    # TODO: Compute the Homography matrix H for corresponding im
    ###############################################################
    bound_pts = np.array([[0, 0], [0, input_image.shape[0]],
                          [input_image.shape[1], 0], [input_image
    bound_pts = np.hstack([bound_pts, np.ones((bound_pts.shape[0

    bound_pts = H.dot(bound_pts)
    bound_pts = bound_pts / bound_pts[-1, :]

    H_inv = np.linalg.inv(H)

    min_row = int(bound_pts[0, :].min())
    max_row = int(bound_pts[0, :].max())
    min_col = int(bound_pts[1, :].min())
```

```
    max_col = int(bound_pts[1, :].max())
    warp_height = max_row - min_row
    warp_width = max_col - min_col

    merge_min_row = min(0, min_row)
    merge_max_row = max(max_row, ref_image.shape[0])
    merge_min_col = min(0, min_col)
    merge_max_col = max(max_col, ref_image.shape[1])
    merge_height = merge_max_row - merge_min_row
    merge_width = merge_max_col - merge_min_col

    warp_image = np.zeros((warp_height, warp_width, 3))
    for row in range(warp_image.shape[0]):
        for col in range(warp_image.shape[1]):
            ref_coor = np.array([row + min_row, col + min_col, 1
            input_coor = H_inv.dot(ref_coor)
            input_coor = input_coor / input_coor[-1]

            input_row, input_col = int(input_coor[1]), int(input
            if input_row >= 0 and input_row < input_image.shape[
            and input_col >= 0 and input_col < input_image.shape
                warp_image[row, col, :] = input_image[input_row,

    merge_image = np.zeros((merge_height, merge_width, 3))
    for row in range(merge_image.shape[0]):
        for col in range(merge_image.shape[1]):
            ref_coor = np.array([row + merge_min_row, col + merg
            input_coor = H_inv.dot(ref_coor)
            input_coor = input_coor / input_coor[-1]

            input_row, input_col = int(input_coor[1]), int(input
            if input_row >= 0 and input_row < input_image.shape[
            and input_col >= 0 and input_col < input_image.shape
                merge_image[row, col, :] = input_image[input_row
            else:
                ref_row = row + merge_min_row
                ref_col = col + merge_min_col
                if ref_row >= 0 and ref_row < ref_image.shape[1]
                and ref_col >= 0 and ref_col < ref_image.shape[0
                    merge_image[row, col, :] = ref_image[ref_col
    ##############################################################
    #                                END OF YOUR CODE
    ##############################################################
    return warp_image.transpose([1, 0, 2]), merge_image.transpos
```

4. Apply System to Provided Images [15 points]

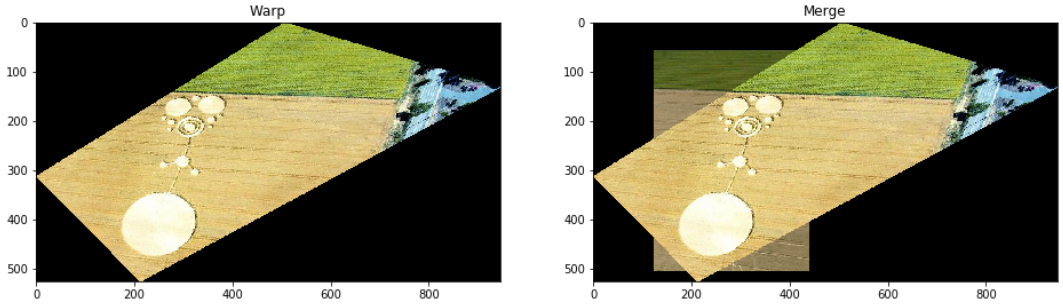
4a. Apply System to crop1.jpg and crop2.jpg

Apply your system to `crop1.jpg` and `crop2.jpg` using the corresponding points

`cc1.npy` and `cc2.npy`. The images and points are included in the zip file.

Display the warped and mosaic images in this notebook. Make sure both images are visible when you save (and submit) the notebook.

In [35]:
```
##################################################################
# TODO: Apply system to Pair 1
##################################################################
t1 = np.load('cc1.npy')
t2 = np.load('cc2.npy')
H = compute_homography(t1, t2)

im1 = np.array(imageio.imread('crop1.jpg'))
im2 = np.array(imageio.imread('crop2.jpg'))
warped_image, merge_image = warp_image(im1, im2, H)
warped_image = warped_image.astype(np.uint8)
merge_image = merge_image.astype(np.uint8)

fig, ax = plt.subplots(1, 2, figsize=(16, 8))

ax[0].imshow(warped_image)
ax[0].set_title('Warp')

ax[1].imshow(merge_image)
ax[1].set_title('Merge')

plt.show()
##################################################################
#                                END OF YOUR CODE
##################################################################
```

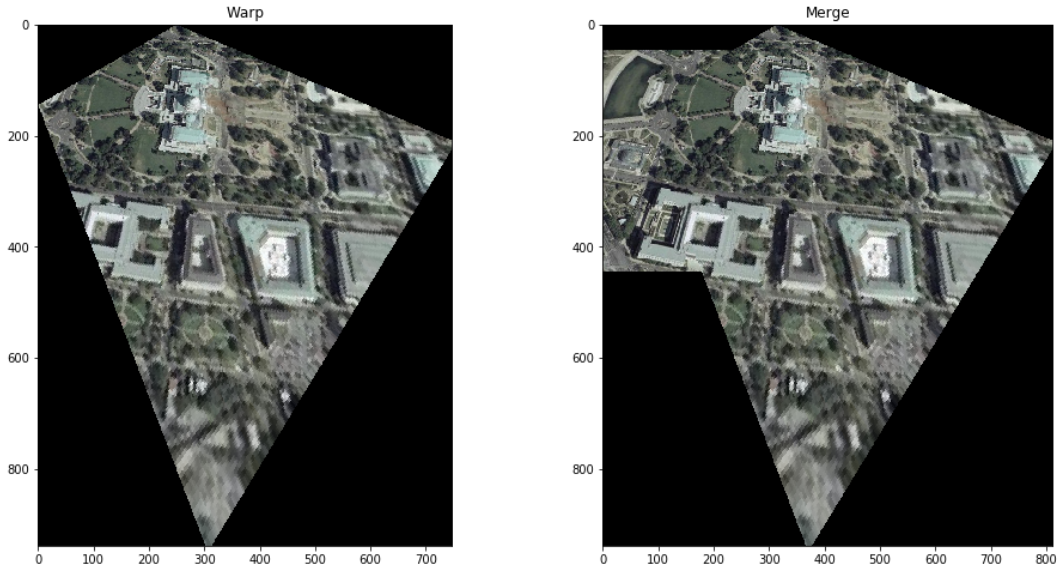


4b. Apply System to wdc1.jpg and wdc2.jpg

Apply your system to `wdc1.jpg` and `wdc2.jpg` using an appropriate choice of (manually selected) corresonding points. Only the images are included in the zip file.

Display the warped and mosaic images in this notebook. Make sure both images are visible when you save (and submit) the notebook.

In [36]:
```
##########################################################################
# TODO: Apply system to Pair 2
##########################################################################
t1 = np.load('wdc1.npy')
t2 = np.load('wdc2.npy')
H = compute_homography(t1, t2)

im1 = np.array(imageio.imread('wdc1.jpg'))
im2 = np.array(imageio.imread('wdc2.jpg'))
warped_image, merge_image = warp_image(im1, im2, H)
warped_image = warped_image.astype(np.uint8)
merge_image = merge_image.astype(np.uint8)

fig, ax = plt.subplots(1, 2, figsize=(16, 8))

ax[0].imshow(warped_image)
ax[0].set_title('Warp')

ax[1].imshow(merge_image)
ax[1].set_title('Merge')

plt.show()
##########################################################################
#                                END OF YOUR CODE
##########################################################################
```

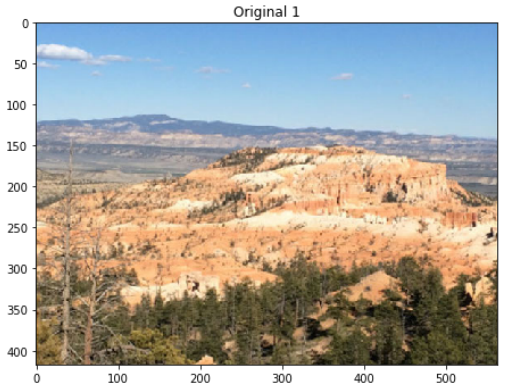


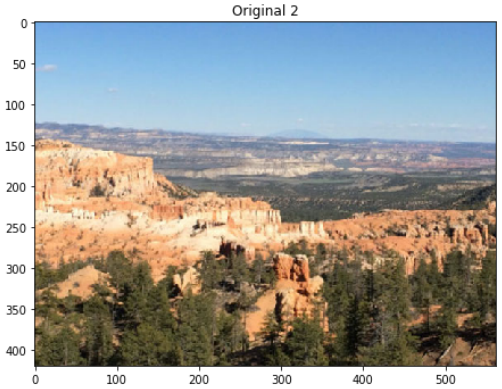
5. Custom Mosaic [20 points]

Show one additional example of a mosaic you create using images that you have taken. You might make a mosaic from two or more images of a broad scene that requires a wide angle view to see well. Or, make a mosaic using two images from the same room where the same person appears in both.

Display the original images and the mosaic in this notebook. Make sure all of the images are visible when you save (and submit) the notebook.

In [37]:
```
##########################################################################
# TODO: Apply system to create a custom mosaic
##########################################################################
t1 = np.load('landscape1.npy')
t2 = np.load('landscape2.npy')
H = compute_homography(t1, t2)

im1 = np.array(imageio.imread('landscape1.jpg'))
im2 = np.array(imageio.imread('landscape2.jpg'))
warped_image, merge_image = warp_image(im1, im2, H)
warped_image = warped_image.astype(np.uint8)
merge_image = merge_image.astype(np.uint8)

fig, ax = plt.subplots(2, 2, figsize=(16, 16))

ax[0, 0].imshow(im1)
ax[0, 0].set_title('Original 1')

ax[0, 1].imshow(im2)
ax[0, 1].set_title('Original 2')

ax[1, 0].imshow(warped_image)
ax[1, 0].set_title('Warp')

ax[1, 1].imshow(merge_image)
ax[1, 1].set_title('Merge')

plt.show()
##########################################################################
#                                END OF YOUR CODE
##########################################################################
```





6. Custom Warp [15 points]

Warp one image into a frame region in the second image. To do this, let the points from the one view be the corners of the image you want to insert in the frame, and let the corresponding points in the second view be the clicked points of the frame (rectangle) into which the first image should be warped. Use this idea to replace one surface in an image with an image of something else. For example – overwrite a billboard with a picture of your dog, or project a drawing from one image onto the street in another image, or replace a portrait on the wall with someone else's face, or paste a Powerpoint slide onto a movie screen, etc.

Display the original images and the mosaic in this notebook. Make sure all of the images are visible when you save (and submit) the notebook.

In [38]:
```
##########################################################################
# TODO: Apply system to create a custom warp
##########################################################################
t1 = np.load('carpet1.npy')
t2 = np.load('carpet2.npy')
H = compute_homography(t1, t2)

im1 = np.array(imageio.imread('carpet1.jpg'))
im2 = np.array(imageio.imread('carpet2.jpg'))
warped_image, merge_image = warp_image(im1, im2, H)
warped_image = warped_image.astype(np.uint8)
merge_image = merge_image.astype(np.uint8)

fig, ax = plt.subplots(2, 2, figsize=(16, 16))

ax[0, 0].imshow(im1)
ax[0, 0].set_title('Original 1')

ax[0, 1].imshow(im2)
ax[0, 1].set_title('Original 2')

ax[1, 0].imshow(warped_image)
ax[1, 0].set_title('Warp')

ax[1, 1].imshow(merge_image)
ax[1, 1].set_title('Merge')

plt.show()
##########################################################################
#                                END OF YOUR CODE
##########################################################################
```

2 [OPTIONAL] Extra Credit [up to 10 points each, max 30 points]

Add as any more cells as needed to implement the following. Please include all implementation and results in the final submission to be considered for extra credit.

1. Automatic Interest Point Detection + Local Feature Matching

Replace the manual correspondence stage with automatic interest point detection and local feature matching. Check out available code here to compute the local interest points and features:

http://www.vlfeat.org/overview/sift.html (http://www.vlfeat.org/overview/sift.html)

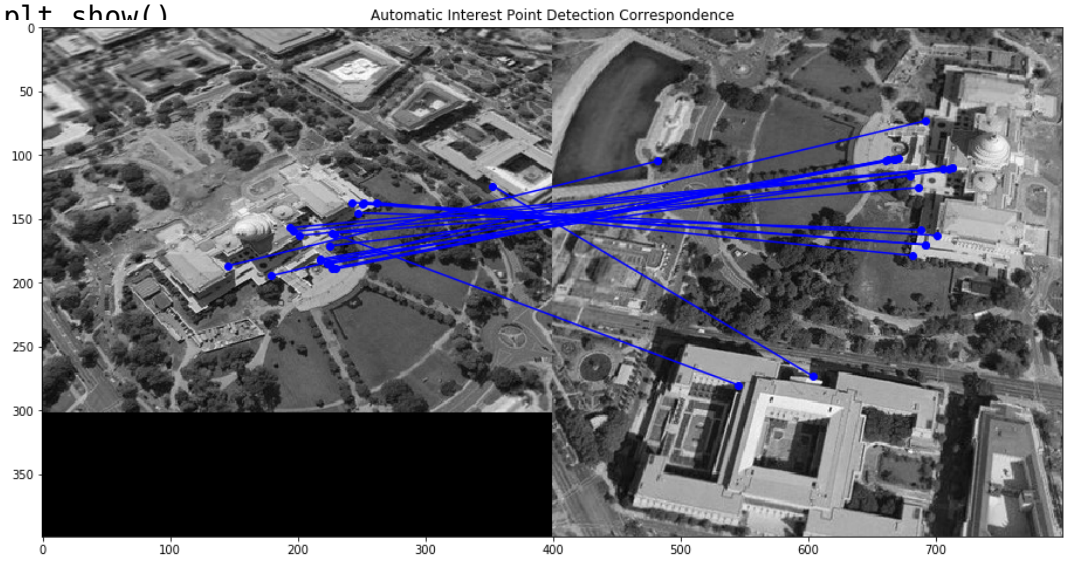
http://www.robots.ox.ac.uk/~vgg/research/affine/detectors.html (http://www.robots.ox.ac.uk/~vgg/research/affine/detectors.html)

Display the automatically detected and matched points in this notebook.

In [39]:
```
def match_points(im0, im1):
    sift = cv2.ORB_create()

    kp0, des0 = sift.detectAndCompute(im0, None)
    kp1, des1 = sift.detectAndCompute(im1, None)

    bf = cv2.BFMatcher()
    matches = sorted(bf.match(des0, des1), key=lambda k:k.distan

    t1 = np.zeros((len(matches), 2))
    t2 = t1.copy()

    for i, mat in enumerate(matches):
        im0_idx, im1_idx = mat.queryIdx, mat.trainIdx
        t1[i, :] = kp0[im0_idx].pt
        t2[i, :] = kp1[im1_idx].pt

    return t1, t2
```

In [40]:
```
wdc1 = cv2.imread('./wdc1.jpg', 0)
wdc2 = cv2.imread('./wdc2.jpg', 0)
t1, t2 = match_points(wdc1, wdc2)

t2_shift = t2.copy()
t2_shift[:, 0] = t2_shift[:, 0] + wdc1.shape[1]
```

```
wdc = np.zeros((max(wdc1.shape[0], wdc2.shape[0]), wdc1.shape[1]
wdc[:wdc1.shape[0], :wdc1.shape[1]] = wdc1
wdc[:wdc2.shape[0], wdc1.shape[1]:] = wdc2

fig, ax = plt.subplots(1, 1, figsize=(16, 8))
ax.imshow(wdc, cmap='gray')
ax.scatter(t1[:, 0], t1[:, 1], color='blue')
ax.scatter(t2_shift[:, 0], t2_shift[:, 1], color='blue')
for i in range(t1.shape[0]):
    ax.plot([t1[i, 0], t2_shift[i, 0]], [t1[i, 1], t2_shift[i, 1
plt.title('Automatic Interest Point Detection Correspondence')
plt.show()
```



We can see most of the points were matched correctly.

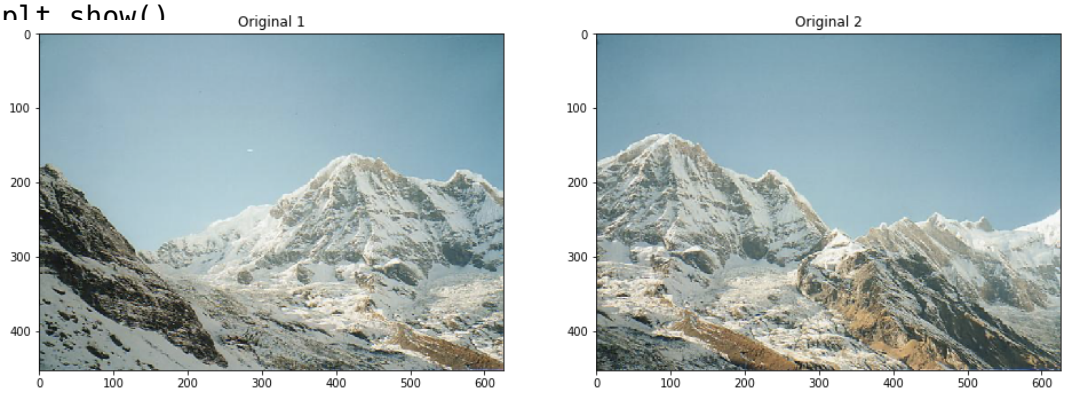
2. RANSAC

Implement RANSAC for robustly estimating the homography matrix from noisy correspondences. Show with an example where it successfully gives good results even when there are outlier (bad) corre- spondences given as input. Compare the robust output to the original (non-RANSAC) implementation where all correspondences are used.

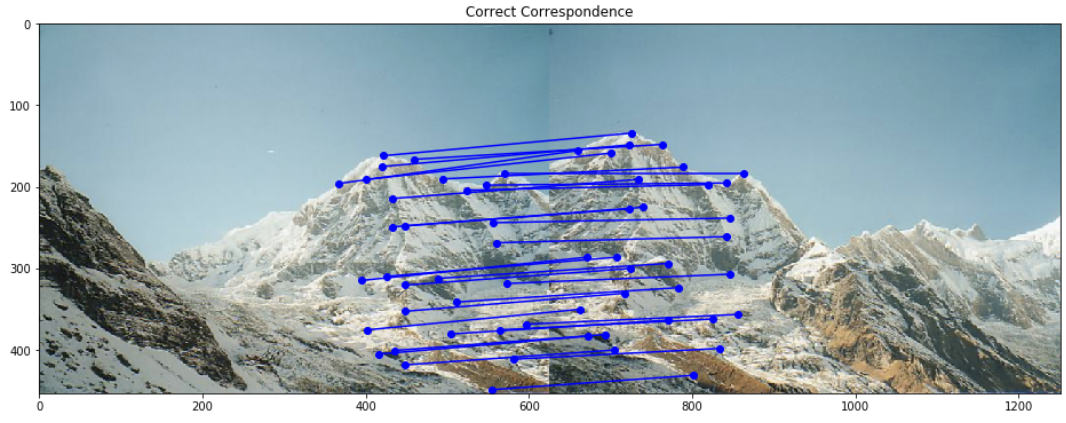
Display the original images and results in this notebook. Make sure all of the images are visibel when you save (and submit) the notebook.

In [41]:
```
mountain1 = np.array(imageio.imread('mountain1.png')).astype(np.
mountain2 = np.array(imageio.imread('mountain2.png')).astype(np.

fig, ax = plt.subplots(1, 2, figsize=(16, 8))
```

```
ax[0].imshow(mountain1)
ax[0].set_title('Original 1')

ax[1].imshow(mountain2)
ax[1].set_title('Original 2')

plt.show()
```

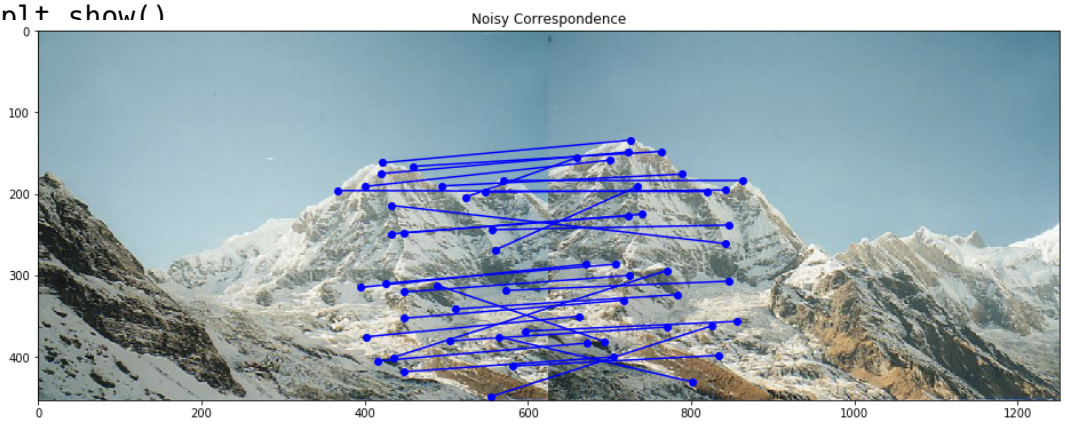


In [42]:
```
t1 = np.load('mountain1.npy')
t2 = np.load('mountain2.npy')

t2_shift = t2.copy()
t2_shift[:, 0] = t2_shift[:, 0] + mountain1.shape[1]

mountain = np.hstack([mountain1, mountain2])

fig, ax = plt.subplots(1, 1, figsize=(16, 8))
ax.imshow(mountain)
ax.scatter(t1[:, 0], t1[:, 1], color='blue')
ax.scatter(t2_shift[:, 0], t2_shift[:, 1], color='blue')
for i in range(t1.shape[0]):
    ax.plot([t1[i, 0], t2_shift[i, 0]], [t1[i, 1], t2_shift[i, 1
plt.title('Correct Correspondence')
plt.show()
```

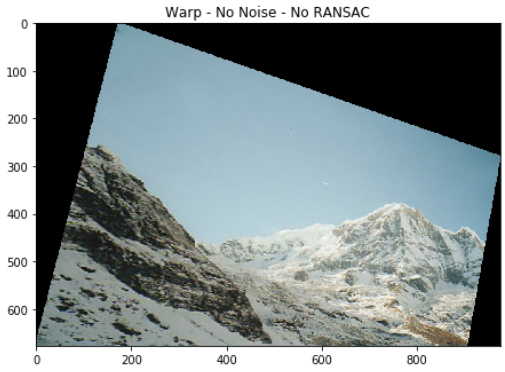


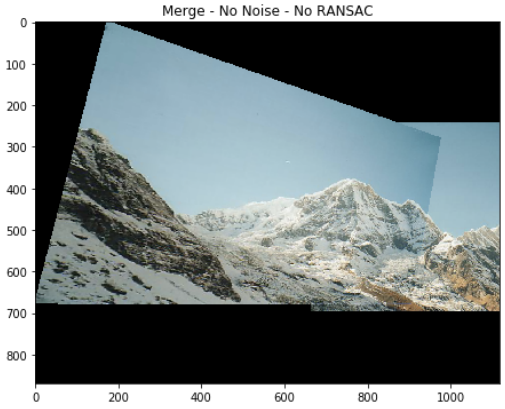
In [43]:
```
# alter this to simulate noise
noise_swaps = [(4, 8), (10, 11), (16, 21), (27, 29)]
```

```
t1_noise = t1.copy()
for swap1, swap2 in noise_swaps:
    temp = t1_noise[swap1, :].copy()
    t1_noise[swap1, :] = t1_noise[swap2, :]
    t1_noise[swap2, :] = temp
t2_noise = t2

t2_shift_noise = t2_noise.copy()
t2_shift_noise[:, 0] = t2_shift_noise[:, 0] + mountain1.shape[1]

mountain = np.hstack([mountain1, mountain2])

fig, ax = plt.subplots(1, 1, figsize=(16, 8))
ax.imshow(mountain)
ax.scatter(t1_noise[:, 0], t1_noise[:, 1], color='blue')
ax.scatter(t2_shift[:, 0], t2_shift[:, 1], color='blue')
for i in range(t1.shape[0]):
    ax.plot([t1_noise[i, 0], t2_shift_noise[i, 0]], [t1_noise[i,
plt.title('Noisy Correspondence')
plt.show()
```

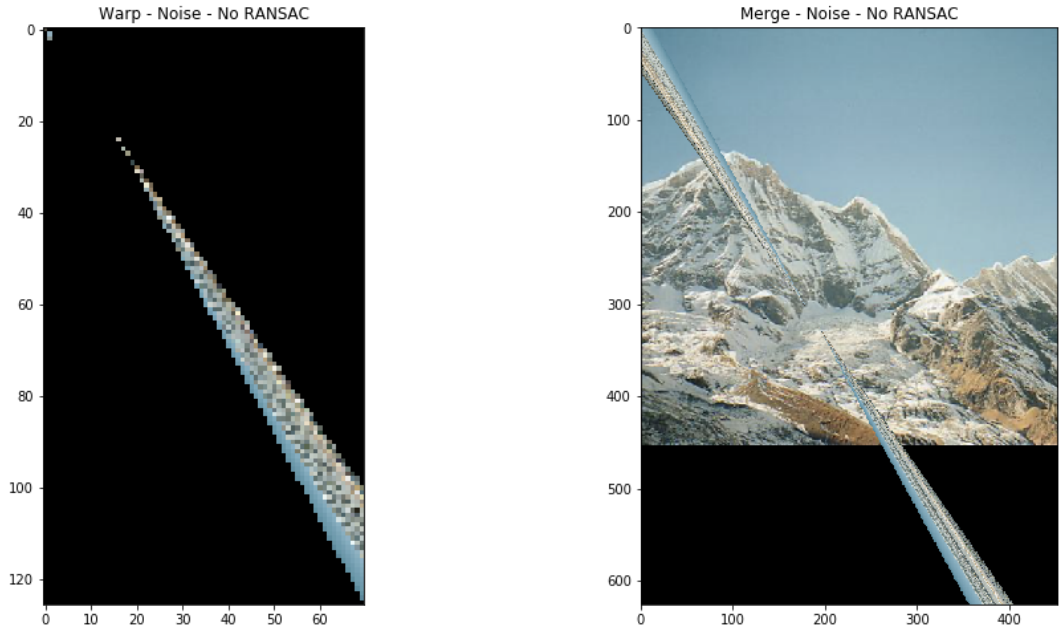


In [44]:
```
H = compute_homography(t1, t2)

warped_image, merge_image = warp_image(mountain1, mountain2, H)
warped_image = warped_image.astype(np.uint8)
merge_image = merge_image.astype(np.uint8)

fig, ax = plt.subplots(1, 2, figsize=(16, 8))

ax[0].imshow(warped_image)
ax[0].set_title('Warp - No Noise - No RANSAC')

ax[1].imshow(merge_image)
ax[1].set_title('Merge - No Noise - No RANSAC')

plt.show()
```





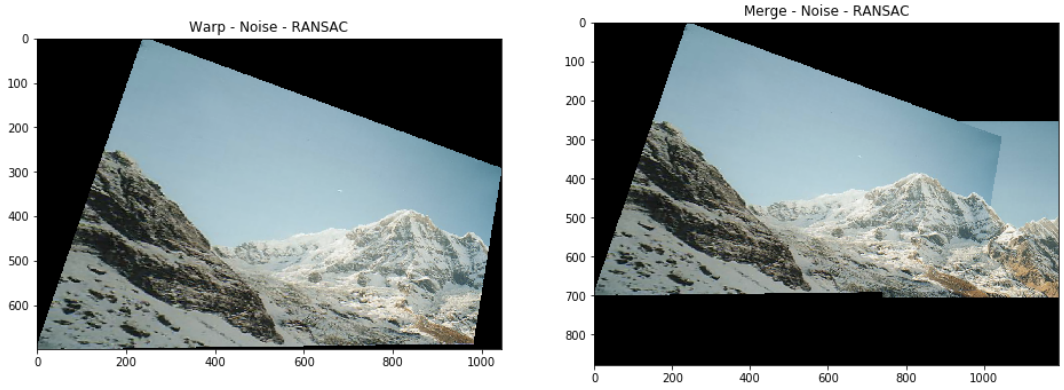
In [45]:
```
H = compute_homography(t1_noise, t2_noise)

warped_image, merge_image = warp_image(mountain1, mountain2, H)
warped_image = warped_image.astype(np.uint8)
merge_image = merge_image.astype(np.uint8)

fig, ax = plt.subplots(1, 2, figsize=(16, 8))

ax[0].imshow(warped_image)
ax[0].set_title('Warp - Noise - No RANSAC')

ax[1].imshow(merge_image)
ax[1].set_title('Merge - Noise - No RANSAC')

plt.show()
```



In [46]:
```
def compute_homography_ransac(t1, t2, n_selection=10, min_inlier
    """
    Computes the Homography matrix for corresponding image point

    The function should take a list of N ≥ 4 pairs of correspond
    from the two views, where each point is specified with its 2
    coordinates.

    Inputs:
    - t1: Nx2 matrix of image points from view 1
    - t2: Nx2 matrix of image points from view 2
    - n_selection: size of seed group to compute initial H
    - min_inlier: proportion of inlier sufficiently large to ter
    - dist_tol: maximum error/euclidean distance tolerated to be
    - max_iter: maximum number of shots before giving up
```

```
    Returns a tuple of:
    - H: 3x3 Homography matrix
    """
    H = np.eye(3)
    inlier = 0
    t1_homo_T = np.ones((t1.shape[1] + 1, t1.shape[0]))
    t1_homo_T[:2, :] = t1.T
    iteration = 0
    best_idx = None
    best_inlier = 0
    while inlier < min_inlier and iteration < max_iter:
        iteration += 1
        idxs = np.random.choice(t1.shape[0], n_selection)
        sub_t1 = t1[idxs, :]
        sub_t2 = t2[idxs, :]
        init_H = compute_homography(sub_t1, sub_t2)
        t2_homo_T = init_H.dot(t1_homo_T)
        t2_homo_T = t2_homo_T / t2_homo_T[-1, :]
        t2_predict = t2_homo_T[:2, :].T
        dists = np.sqrt(np.sum((t2_predict - t2)**2, axis=1))
        inlier_idxs = dists < dist_tol
        inlier_count = np.count_nonzero(inlier_idxs)
        inlier = inlier_count / t1.shape[0]
        if best_idx is None or inlier > best_inlier:
            best_idx = idxs
            best_inlier = inlier
        if inlier > min_inlier:
            break
    return compute_homography(t1[best_idx, :], t2[best_idx, :])
```

In [47]:
```
H = compute_homography_ransac(t1_noise, t2_noise)

warped_image, merge_image = warp_image(mountain1, mountain2, H)
warped_image = warped_image.astype(np.uint8)
merge_image = merge_image.astype(np.uint8)

fig, ax = plt.subplots(1, 2, figsize=(16, 8))

ax[0].imshow(warped_image)
ax[0].set_title('Warp - Noise - RANSAC')

ax[1].imshow(merge_image)
ax[1].set_title('Merge - Noise - RANSAC')

plt.show()
```



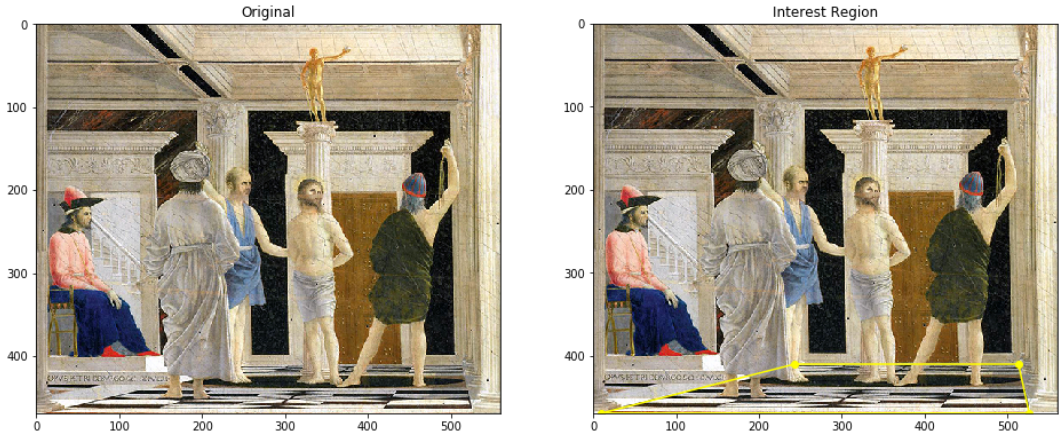
3. Rectification

Rectify an image with some known planar surface (say, a square floor tile, or the rectangular face of a building facade) and show the virtual fronto-parallel view. In this case there is only one input image. To solve for H, you define the correspondences by clicking on the four corners of the planar surface in the input image, and then associating them with hand-specified coordinates for the output image. For example, a square tile's corners from the non-frontal view could get mapped to [0 0; 0 N; N 0; N N] in the output.

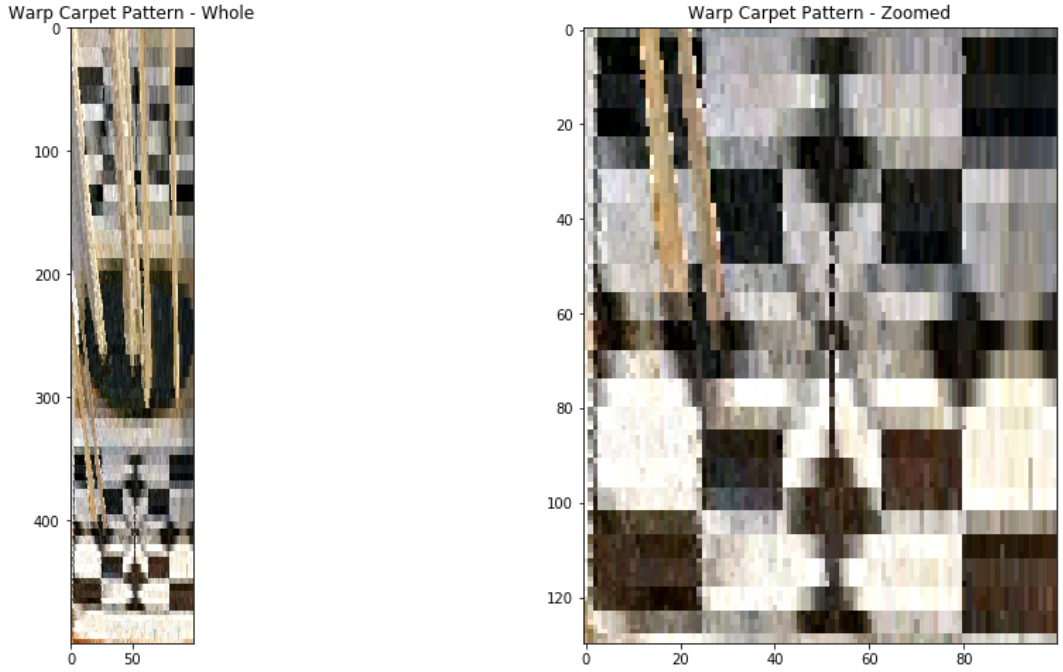
Display the original and rectified images in this notebook. Make sure both images are visible when you save (and submit) the notebook.

In [48]:
```
christ = imageio.imread('christ.jpg')
christ_back = imageio.imread('christ-back.png')[:, :, :3]

fig, ax = plt.subplots(1, 2, figsize=(16, 8))

ax[0].imshow(christ)
ax[0].set_title('Original')
```

```
anchors = np.array([[243, 410], [513, 410], [526, 469], [10, 469
ax[1].imshow(christ)
ax[1].scatter(anchors[:, 0], anchors[:, 1], color='yellow')
ax[1].plot(list(anchors[:, 0]) + [anchors[0, 0]], list(anchors[:
ax[1].set_xlim(0, christ.shape[1])
ax[1].set_ylim(christ.shape[0], 0)
ax[1].set_title('Interest Region')
plt.show()
```



In [49]:
```
height = 500
width = 100
anchors_mapped = np.array([[0, 0], [width, 0], [width, height],
H = compute_homography(anchors, anchors_mapped)

_, merge_image = warp_image(christ, christ_back, H)
merge_image = merge_image.astype(np.uint8)[:height, :width, :]

cropped2 = merge_image[345:475, :, :]
fig, ax = plt.subplots(1, 2, figsize=(16, 8))
ax[0].imshow(merge_image)
ax[0].set_title('Warp Carpet Pattern - Whole')
ax[1].imshow(cropped2)
ax[1].set_title('Warp Carpet Pattern - Zoomed')
plt.show()
```



4. Short Video

Make a short video in the style of the HP commercial (https://www.youtube.com/watch?v=2RPl5vPEoQk)'s video which you saw in class. Building on #3 above, let the frame in the output video move to different positions over time, and warp the framed image into the correct position for every video frame in the sequence.

Name the video `ps3-extra-credit.mp4` *and submit it.*

NOT IMPLEMENTED

3 Deliverable Checklist

- [ ] `ps3.ipynb`
- [ ] `compute_homography.py`
- [ ] `pick-points.py`
- [ ] [extra credit] `ps3-extra-credit.mp4`