Rana Mohamed Mostafa Barakat 6926
Mohab Mohamed Ali 7199

Assignment 2
Image Mosaics

We used matplotlib's interactive window to capture mouse clicks. The interactive window doesn't work with matplotlib inline so we used matplotlib qt instead using the following command:

```
%matplotlib qt
```
✓ 0.7s

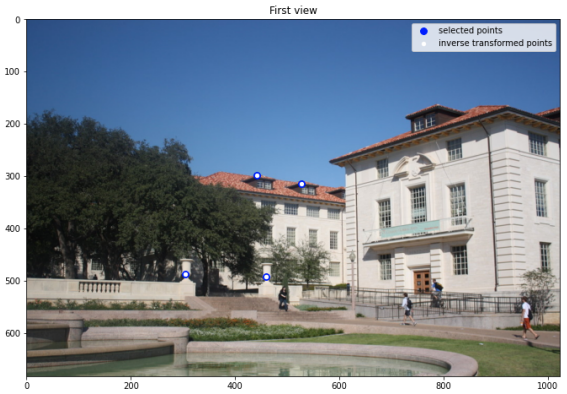
To capture mouse clicks and coordinates, we used an event listener embedded in a window that displays both images. The user selects one point from the left image and its corresponding point in the right image. This is implemented as shown.

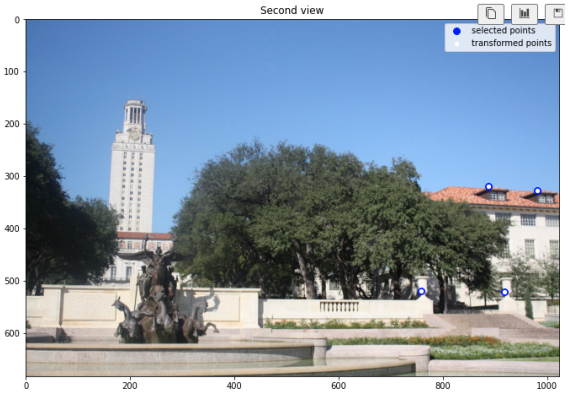
```python
def find_correspondences(img1, img2):
    points = []
    def onclick(event):
        global correspondonces
        # print('in event')
        ix, iy = np.round((event.xdata, event.ydata))
        points.append((int(ix), int(iy)))

        if len(points)>=8:
            fig.canvas.mpl_disconnect(cid)
            correspondonces = [(p1, p2) for p1, p2 in zip(points[0::2], points[1::2])]

    fig = plt.figure(figsize=(25,10))

    plt.subplot(1, 2, 1)
    plt.imshow(img1)
    plt.title('First view')

    plt.subplot(1,2,2)
    plt.imshow(img2)
    plt.title('Second view')

    cid = fig.canvas.mpl_connect('button_press_event', onclick)
    plt.show()
```

Next, it was time to compute the homography. This is simply done by constructing a system of linear equations in the form of a matrix and using SVD to solve for the parameters of the transformation matrix.

```python
def find_H(correspondences):
    P = []
    for ((x1,y1),(x2,y2)) in correspondences[:4]:
        rows = [[x1, y1, 1, 0, 0, 0, -x1*x2, -y1*x2, -x2],
                [0, 0, 0, x1, y1, 1, -x1*y2, -y1*y2, -y2]]
        P.extend(rows)

    _, __, V = np.linalg.svd(np.array(P))
    H = V[-1, :] / V[-1,-1]
    return H.reshape((3,3))
```

We can verify the correctness of the computed matrix by displaying each point selected by the user, its corresponding point also selected by user, as well as the computed point using the homography and the first point.





Having computed and verified the transformation matrix, we can now warp one image into the coordinate system of the other. First, we split the image by channels, warp each channel individually, then merge them back as shown.

```
def warp_multichannel(img, H):
    output_img = []
    for channel in cv.split(img):
        warped_channel, offset_x, offset_y = warp_singlechannel(channel, H)
        output_img.append(warped_channel)
    .....
    return cv.merge(output_img).astype(np.int64), offset_x, offset_y
```

For each channel, the first step is to compute the new image size. This can be done by computing the transformed corners, then computing the width and height.

```
(max_x, min_x), (max_y, min_y) = get_boundary(img, H)
width = int(np.ceil(max_x)-np.floor(min_x))
height = int(np.ceil(max_y)-np.floor(min_y))
output_shape = (height,width)
```

Since the image could have a starting point at non-zero points, we translate the image so that it's centred around 0 to be able to store it using regular array indexing. We also keep track of this translation so we can move it back when stitching the images.

```
if min_x<0:
    offset_x = (0-int(np.floor(min_x)))-1
else:
    offset_x = (0-int(np.ceil(min_x)))+1

if min_y<0:
    offset_y = (0-int(np.floor(min_y)))-1
else:
    offset_y = (0-int(np.ceil(min_y)))+1
```

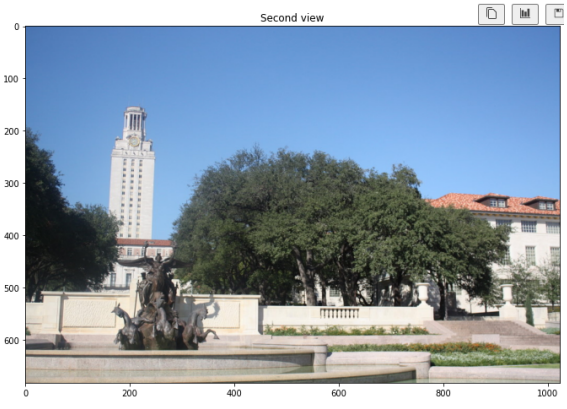
Next, we perform forward warping. For each pixel in the original image, we transform it using H. The result is usually sub-pixel so we use splatting to distribute the output among neighbouring pixels. This results in a very minor blurring effect but it greatly reduces the number of holes in the image.

```python
# Forward warp
for y in range(img.shape[0]): # for each row
    for x in range(img.shape[1]): # for each column
        intensity = img[y,x]
        pose = np.array([x,y,1])[:, np.newaxis] # homogoenous coordinates
        new_pose = np.matmul(H, pose).squeeze() # find new coordinates
        new_pose = new_pose/new_pose[2] # normalize
        new_pose[0]+=offset_x
        new_pose[1]+=offset_y
        x_new, y_new = new_pose[:2]
        if (np.ceil(x_new)-x_new + np.ceil(y_new)-y_new)==0: # if resulting coordinates are whole numbers
            output[int(y_new),int(x_new)] = intensity # update intensity
        else: # if not splatting is required
            points = set(get_nearest_points(new_pose[:2])) # get four nearest points
            for x_near,y_near in points: # for each point
                if x_near<0 or x_near>=width or y_near<0 or y_near>=height:
                    continue
                if (y_near,x_near) in splatted: # if point has been splatted before
                    n, avg_intensity = splatted[(y_near,x_near)] # get old number of splatting contributors
                    new_avg = (n*avg_intensity+intensity)/(n+1)
                    splatted[(y_near,x_near)] = (n+1, new_avg) # update by adding current intensity
                else: # otherwise
                    splatted[(y_near,x_near)] = (1, intensity) # avg intensity is current intensity with only 1 contributor
# Apply splatting
for point in splatted.keys():
    _, avg_intensity = splatted[point]
    output[point] = int(avg_intensity)
```

We follow this up by performing an inverse warp for any holes in the output. We take the empty spots in the output, compute the coordinates in the original image using the inverse transformation matrix, and apply the intensity. Similar to forward warping, this also usually results in sub-pixel values. We use interpolation to distribute the pixel intensity according to their proximity.

```python
# Inverse warp
H_inv = np.linalg.inv(H) # calculate inverse homography
to_warp = [(x,y) for y, x in zip(*np.where(output==0))] # find points to warp. i.e. points that were not updated during forward warp
for x,y in to_warp: # for each point
    x, y = x-offset_x, y-offset_y
    new_pose = np.array([x,y,1])[:, np.newaxis] # homogenous coords
    pose = np.matmul(H_inv, new_pose).squeeze() # apply inverse warping
    pose = (pose/pose[2])
    interp = interpolate(img, pose[:2]) # interpolate to find intensity
    x, y = x+offset_x, y+offset_y
    output[y,x] = int(interp)

return output, offset_x, offset_y
```

The result of the warp is shown below.



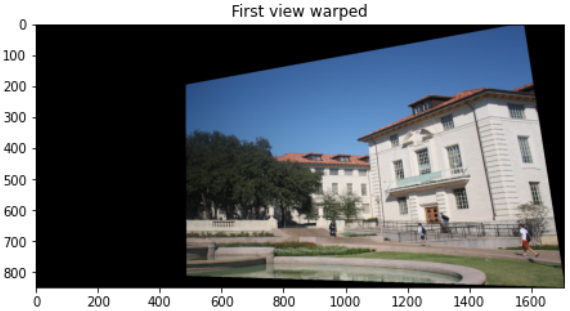
The next step is to create an image that can fit both the warped image and the second image. We can do this by finding the minimum origin from both images as well as the maximum edge in both images. This would result in an image that can fit both views. Since we centred the warped image around 0 earlier, we need to keep the offset in mind during calculations.

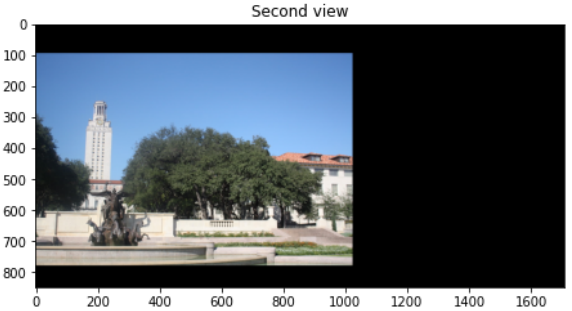
```python
def create_img_containers(img1, img2, offset_x, offset_y):
    min_x, max_x = min(0-offset_x, 0), max(img1.shape[1]-offset_x, img2.shape[1])
    min_y, max_y = min(0-offset_y, 0), max(img1.shape[0]-offset_y, img2.shape[0])
    shape = (max_y-min_y, max_x-min_x, 3)
    img1_warped_adjusted = np.zeros(shape, dtype=np.int64)
    img2_t = np.zeros(shape, dtype=np.int64)

    def fill_image(img, shape, offset_x, offset_y):
        container = np.zeros(shape, dtype=np.int64)
        for x in range(img.shape[0]):
            for y in range(img.shape[1]):
                container[x+offset_y, y+offset_x] = img[x,y]
        return container

    if offset_x>0:
        if offset_y>0:
            img2_t = fill_image(img2, shape, offset_x, offset_y)
            img1_warped_adjusted = fill_image(img1, shape, 0, 0)
        else:
            img2_t = fill_image(img2, shape, offset_x, 0)
            img1_warped_adjusted = fill_image(img1, shape, 0, -offset_y)
    else:
        if offset_y>0:
            img2_t = fill_image(img2, shape, 0, offset_y)
            img1_warped_adjusted = fill_image(img1, shape, -offset_x, 0)
        else:
            img2_t = fill_image(img2, shape, 0, 0)
            img1_warped_adjusted = fill_image(img1, shape, -offset_x, -offset_y)

    return img1_warped_adjusted, img2_t
```

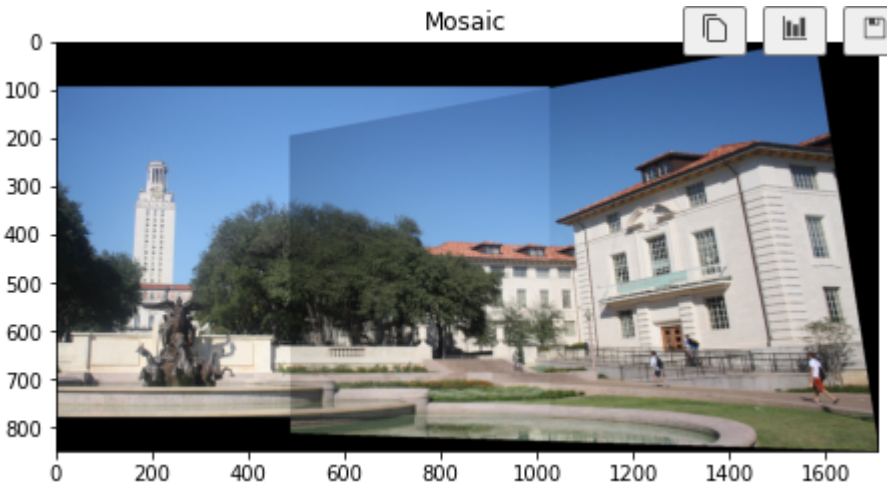
To overlay the images, we can simply pass over the new image and check each pixel, if it exists in both images, take the average value, otherwise, take the existing pixel value.

```python
def overlay(img1, img2):
    op = np.zeros(img2.shape, dtype=np.int64)
    for i in range(op.shape[0]):
        for j in range(op.shape[1]):
            if np.all(img1[i,j]==0):
                op[i,j] = img2[i,j]
            elif np.all(img2[i,j]==0):
                op[i,j] = img1[i,j]
            else:
                op[i,j] = (img1[i,j]+img2[i,j])/2
    return op
```

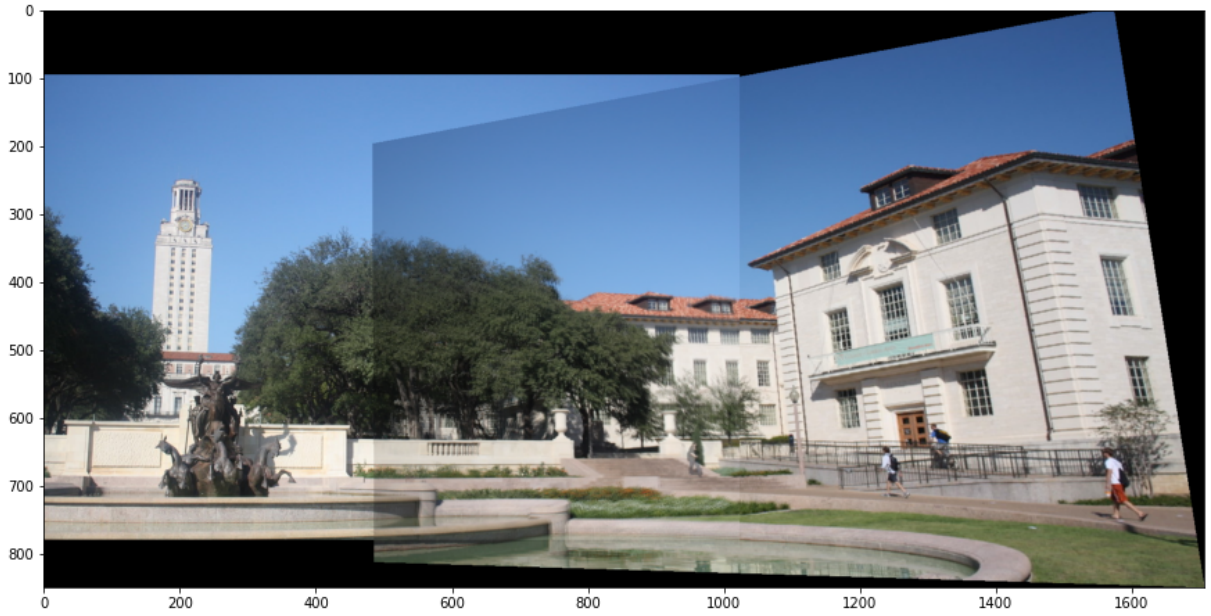
This is the result of the above operations:







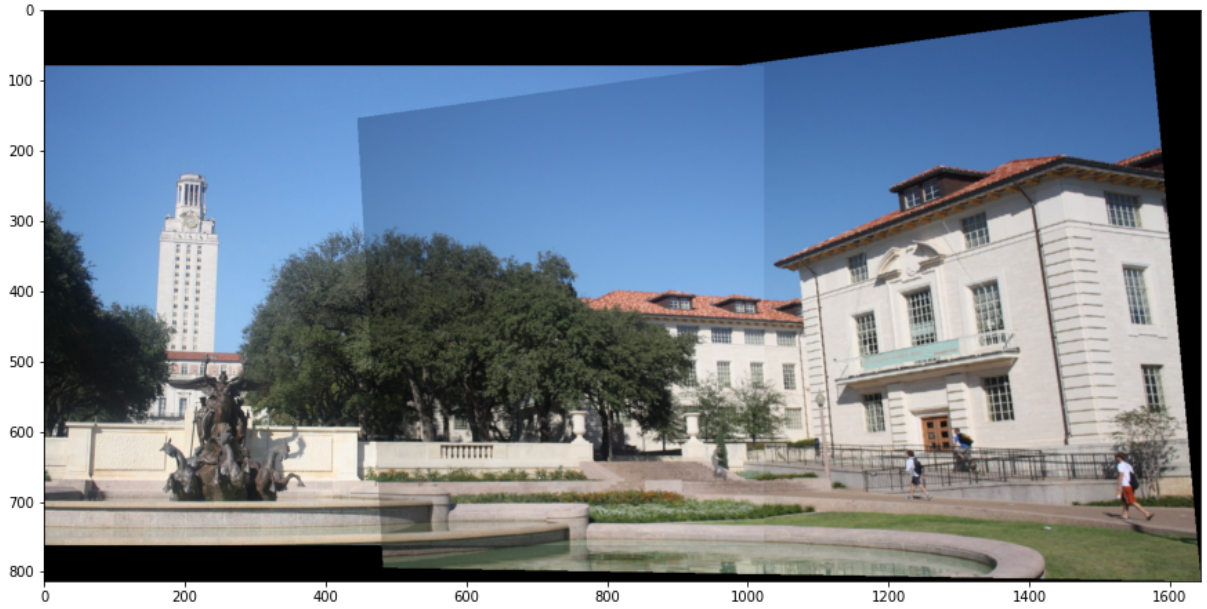
Additional tests clearly show that user input has a huge impact on the resulting transformation:



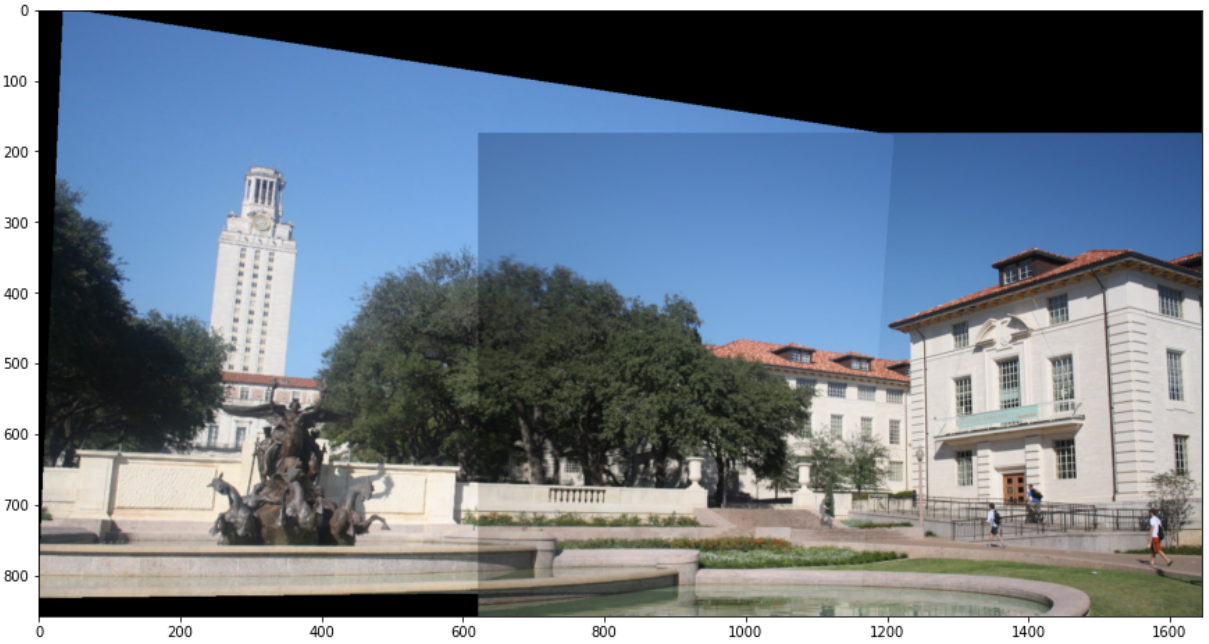
To avoid this, we also implemented a function that matches features between the two images using opencv's SIFT implementation and brute force matcher. RANSAC is then applied to determine the best homography.

```python
def find_homography_cv(img1, img2):
    gray1 = cv.cvtColor(img1, cv.COLOR_RGB2GRAY)
    gray2 = cv.cvtColor(img2, cv.COLOR_RGB2GRAY)

    sift = cv.SIFT_create()

    kp1, des1 = sift.detectAndCompute(gray1, None)
    kp2, des2 = sift.detectAndCompute(gray2, None)

    bf = cv.BFMatcher()

    matches = bf.knnMatch(des1, des2, k=2)

    good_matches = []
    for m,n in matches:
        if m.distance < 0.7 * n.distance:
            good_matches.append(m)

    src_pts = np.float32([kp1[g.queryIdx].pt for g in good_matches]).reshape(-1,1,2)
    dst_pts = np.float32([kp2[g.trainIdx].pt for g in good_matches]).reshape(-1,1,2)

    h, _ = cv.findHomography(src_pts, dst_pts, cv.RANSAC, 5.0)

    return h
```

The resulting stitch is a lot cleaner with minimal blur.



We also experimented with the order of the stitch (which image gets warped). The results show that this choice can affect the quality of the final output as shown below by reversing the order.



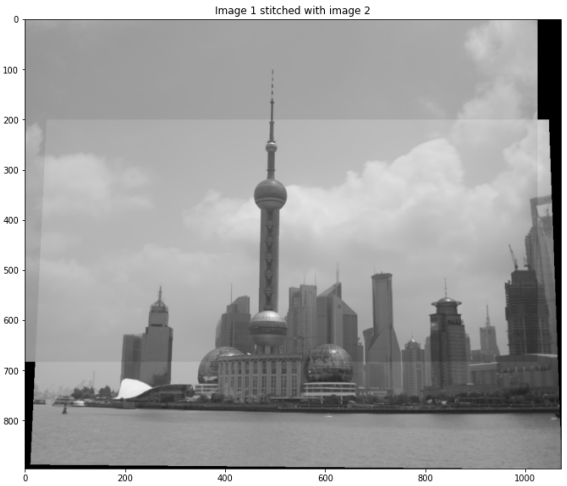
We wrapped all of this in 1 function call. You input the images and the correspondences and receive the output stitch.

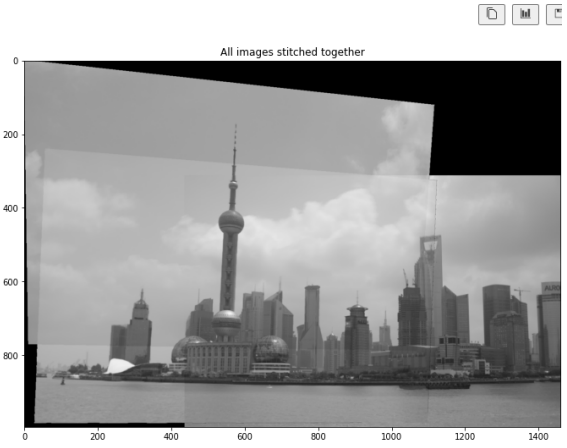
```python
def stitch_images(img1, img2, correspondences=None, plot_correspondences=False):
    if correspondences is None:
        H = find_homography_cv(img1, img2)
    else:
        H = find_H(correspondences)
        if plot_correspondences:
            show_correspondences(img1, img2, correspondences, H)

    img1_warped, offset_x, offset_y = warp(img1, H)

    img1_container, img2_container = create_img_containers(img1_warped, img2, offset_x, offset_y)

    mosaic = overlay(img1_container, img2_container)

    return mosaic.astype(np.uint8)
```

BONUS:

Due to the simplicity of the wrapper function, stitching three images was very simple. Stitch 2 images then add the third.

```python
mid_mosaic = stitch_images(img1, img2)

final_mosaic = stitch_images(mid_mosaic, img3)

fig = plt.figure(figsize=(25,10))

plt.subplot(1, 2, 1)
plt.imshow(mid_mosaic)
plt.title('Image 1 stitched with image 2')

plt.subplot(1,2,2)
plt.imshow(final_mosaic)
plt.title('All images stitched together')

plt.show()
```





You are prompted at every stage for correspondences or you can simply use SIFT descriptors to find the matching features.