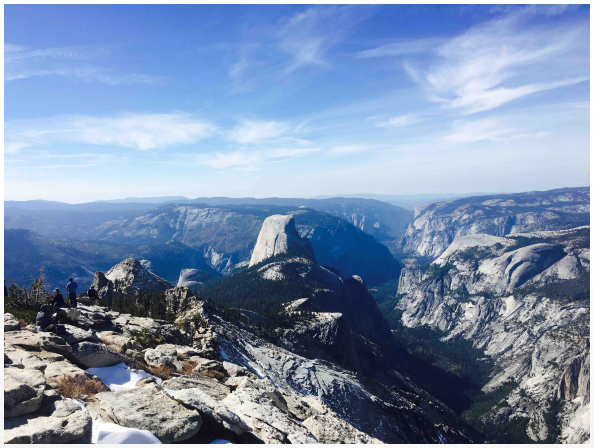
Computational Photography
Assignment #8
Panoramas

Ram Subramanian

Fall 2016

The Scene

- The pictures are of Yosemite valley taken from Cloud's rest
- Cylindrical panorama

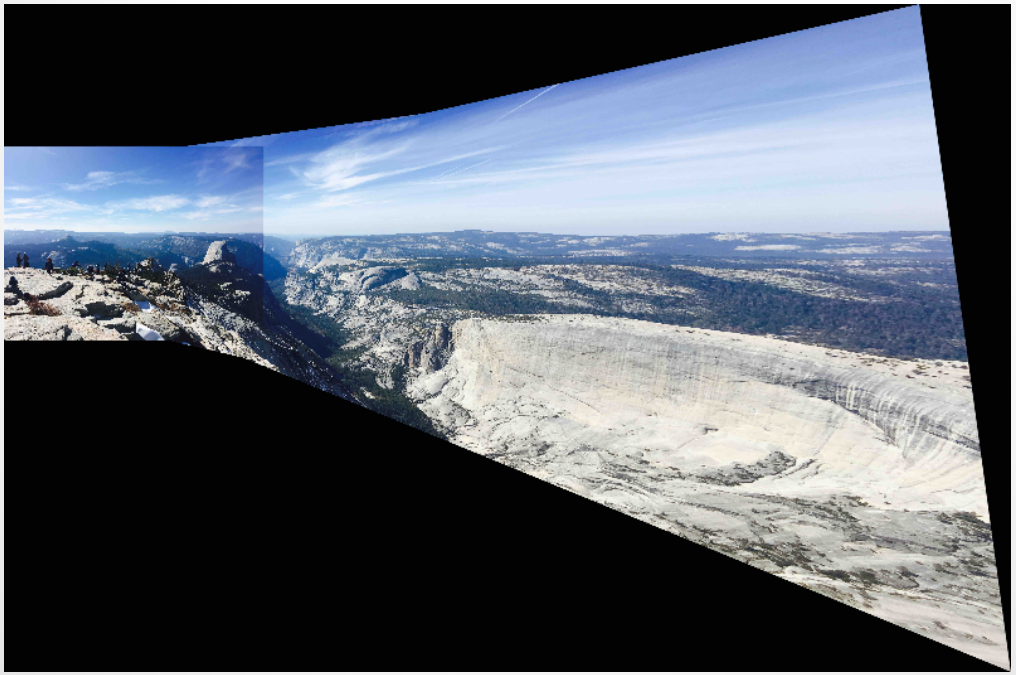




Results - Test Images

- I used alpha blending to merge the image together. The mask for the alpha blending is created from the final location of image_2 passed through a gaussian filter (with a large kernel).
- I'm not too happy with the results since you can see the boundaries very clearly and I feel as if the panorama could have been made that looks less 'warpy'.

Results - Original Images

- Better results here as you can see two of the constituent images were blended seamlessly, while the third wasn't done nearly as well.
- The picture is quite warped! This maybe because the camera didn't pivot along the same axis while taking each of the constituent pictures. Also, as mentioned in Piazza, I'm using my own set of libraries (setup through conda).



Implementation: getImageCorners() & findMatchesBetweenImages()

```
s = image.shape
corners = np.zeros((4, 1, 2), dtype=np.float32)
corners[0][0] = (0, 0)
corners[1][0] = (s[1], 0)
corners[2][0] = (s[1], s[0])
corners[3][0] = (0, s[0])
return corners
```

```
orb = cv2.ORB()
kp1, des1 = orb.detectAndCompute(image_1, None)
kp2, des2 = orb.detectAndCompute(image_2, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des1, des2)

matches = sorted(matches, key = lambda x:x.distance)
return kp1, kp2, matches[:num_matches]
```

getImageCorners() simply returns the co-ordinates of the 4 corners of the image provided. Since each image array is exactly the size of the image, the co-ordinates are simply a combination of 0, the width and height of the array/image.

findMatchesBetweenImages() analyses the images provided, builds a list of 'features', matches the feature points from one image to another (if any) and returns the requested number of best matches (by hamming distance). This is done so we can get a sense for how the camera moved between pictures and eventually to generate a homography for warping.

My images were captured at 4032x3024 resolution. Increasing the number of feature points exponentially increased the time taken to stitch the panorama together, but since the quality of features and matches were good, increasing the number didn't have distinguishable results in the final image.

Implementation: findHomography()

```
image_1_points = np.array([[image_1_kp[match.queryIdx].pt] for match in matches], dtype=np.float32)
image_2_points = np.array([[image_2_kp[match.trainIdx].pt] for match in matches], dtype=np.float32)
H, _ = cv2.findHomography(image_1_points, image_2_points, method=cv2.RANSAC, ransacReprojThreshold=5.0)
return H
```

In order to find homography, I first build a list of points from image_1 that were matched to some other points in image_2 and do the same for image_2 (vice-versa). The elements from the former list were matched to the corresponding elements in the latter list.

Then I simply pass these lists to cv2.findHomography() to calculate a transformation matrix.

Lower the re-projection threshold, more matches are eliminated possibly leading to a less robust homography – but this is highly dependent on the quality of matches in the first place.

Implementation: getBoundingCorners()

```
xu = np.squeeze(getImageCorners(image_1))
xu = np.concatenate((xu, np.ones((4, 1))), axis=1)
xu = np.dot(homography, xu.T).T
x = np.array([[i/k, j/k] for (i, j, k) in xu])
y = np.squeeze(getImageCorners(image_2))
z = np.vstack((x, y))
min_xy = np.array([np.amin(z[:,0]), np.amin(z[:,1])])
max_xy = np.array([np.amax(z[:,0]), np.amax(z[:,1])])
return min_xy, max_xy
```

getBoundingCorners(), given two images and a homography to warp image_1 to 2, will warp just the four corners of image_1 into the co-ordinates of image_2 (i.e. find out where to position image_1 on 2) and calculate the minimum size of the output image that will contain all of image_1 and image_2. So this function is called to calculate the size of output image and the location on output image at which to place image_1.

Implementation: warpCanvas()

```
size = tuple(np.round(max_xy - min_xy).astype(np.int))
T = np.array([[1, 0, -min_xy[0]], [0, 1, -min_xy[1]], [0, 0, 1]])
H = np.dot(T, homography)    warped_image = cv2.warpPerspective(image, H, size)
return warped_image
```

warpCanvas() warps a picture given a homography. Additionally, it takes a translation parameter that is used to position the image at the appropriate location in the warped_image (in this case, to line it up with the features in image_2).

Implementation: blendImagePair()

```
img2 = np.zeros(warped_image.shape)
img2[point[1]:point[1]+image_2.shape[0], point[0]:point[0]+image_2.shape[1]] = image_2
mask = img2.copy()    mask[np.where(mask > 0)] = 1
mask = cv2.GaussianBlur(mask, (11, 11), 0.7)
output_image = warped_image * (1 - mask) + img2 * mask
# mask1 = warped_image.copy()    # mask2 = img2.copy()
# mask1[mask1 > 0] = 1
# mask2[mask2 > 0] = 1
# common_mask = mask1 * mask2 * 0.5
# mask1 = mask1 - common_mask
# mask2 = mask2 - common_mask
# output_image = warped_image * mask1 + img2 * mask2
```

I tried two approaches here (one is commented out). The first is to simply take the boundaries of image_2, blur it using a large enough kernel and sigma for the transition and apply the simple alpha blend with both images. This produced better results that the other method, which was to have separate masks for each image.

Discussion & Analysis

- With simple alpha blending there were always double-images caused by imperfect warping/alignment. So replacing it with a hybrid alpha-insertion method gave much better results (for obvious reasons).
- Then to take care of boundaries caused by differing intensities, I tried to blur the alpha mask with a large kernel and sigma – this helped remove one of the seams whereas the second seam was just too strong. Any larger kernel resulted in MemoryError's.
- I was surprised to see things like numpy indexing, broadcasting, etc, which are usually super fast take a few seconds for even moderately sized arrays. Implementing this algorithm in C++ would definitely yield much better results IMO.
- If I could restart this assignment, I would try and rotate the camera about the same axis to take all of my constituent pictures to see if that affects the amount of warping done by the homography.

References

List the sources for your ideas. You do not need to list the lectures.

Include items such as technical papers, Wikipedia articles, websites, and significant Piazza threads