How does Panorama work?

Samrudha Kelkar Follow
May 17 · 4 min read ★

Image Stitching

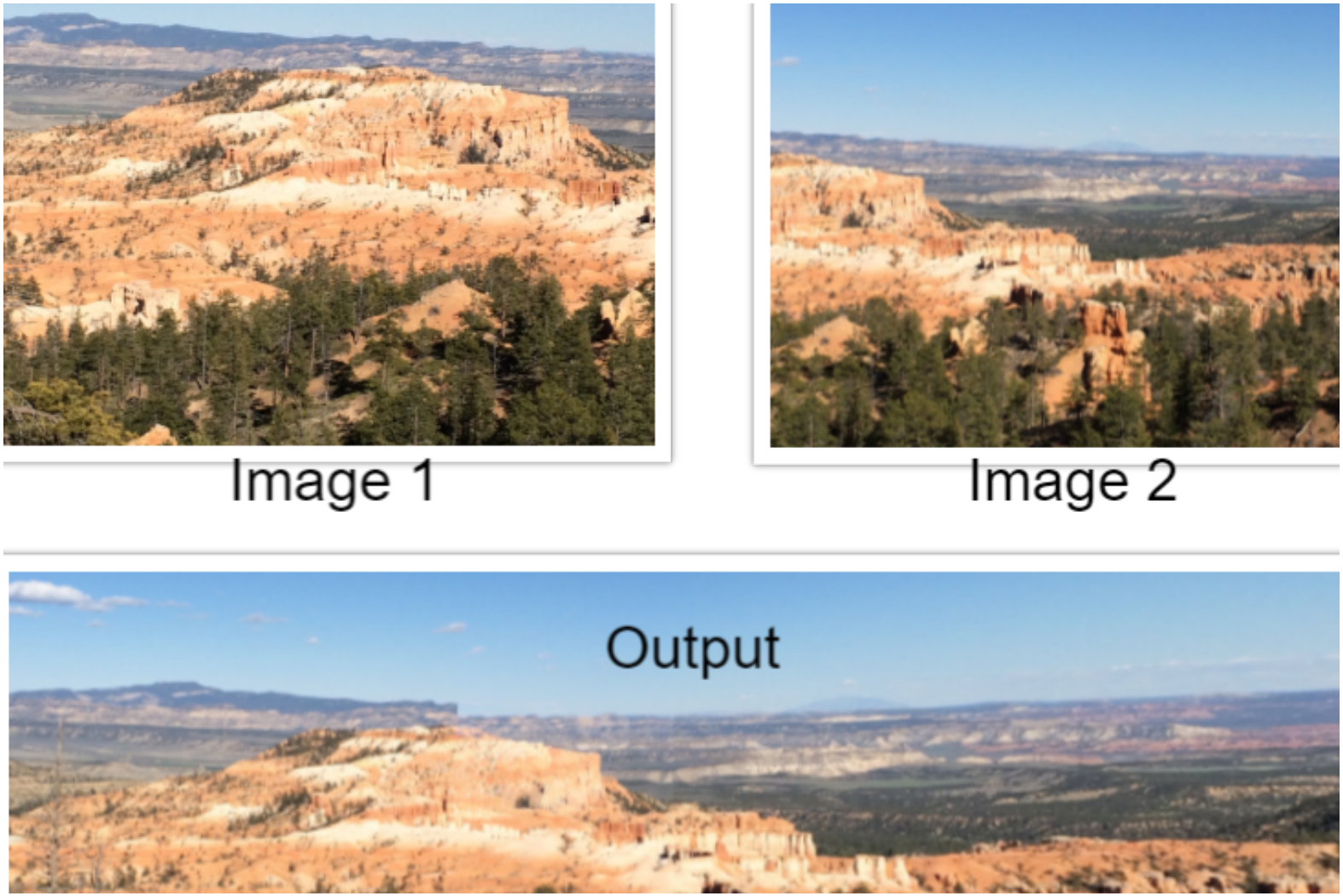
Panorama effect! Yes, that's what we will uncover in this post.

Image mosaicing/stitching is the task of sticking one/more input images. We have information spread across these images which we would like to see at once. In a single image!

We all have used Panorama mode on mobile camera. In this mode, the image-mosaicing algorithm runs to capture and combine images. But we can use the same algo offline also. See the image below, where we have pre-captured images and we want to combine them. Observe the hill is partially visible in both inputs.

When you have multiple images to combine, the general approach followed is to pairwise add 2 images each time. The output of last addition is used as an input for the next image.

> Assumption is that each pair of images under consideration do have certain features common.



Left and Right images stitched together using local features of interest points

Let’s see the step by step procedure for creating a panorama image:

Step 1] Read Input Images

Take images with some overlapping structure/object. If you are using own camera then make sure you do not change camera properties while taking pictures. Moreover, do not take many similar structures in the frame. We do not want to confuse our tiny little algorithm.

Now read both first two images in OpenCV.

```
def createPanorama(input_img_list):
    images = []
    dims = []
    for index, path in enumerate(input_img_list):
        print (path)
        images.append(cv2.imread(path))
        dims.append(images[index].shape)
    return images, dims
```

Step 2] Compute SIFT features

Detect features/interest points for both images. These points are unique identifiers which are used as *markers*. We will use SIFT features. It is a popular local features detection and description algorithm. It is used in many computer vision object matching tasks. Some other examples of feature descriptors are SURF, HOG.

SIFT uses a pyramidal approach using DOG (difference of gaussian). Features thus obtained will be invariant to scale. It is good for panorama kind of applications wherein images might have features variations in rotations, scale, lighting, etc.

```
## Define feature type
feature_typye = cv2.xfeatures2d.SIFT_create()

points1, des1 = features.detectAndCompute(image1,  None)
points2, des2 = features.detectAndCompute(image2,  None)
```

Here *points1* is a list of key points whereas *des1* is a list of descriptors expressed in the feature space of SIFT. Each descriptor will be a 1x128 vector

Step 3] Match strong interest points

Now we will be matching points based on vector representation. We will assume a certain threshold for deciding whether two points are ***near*** or **not**

OpenCV has inbuilt FLANN based Matcher for this purpose. It is a **histogram based** matching technique which computes the *distance* for 2 points described in SIFT feature space.

```
## Define flann based matcher
matcher = cv2.FlannBasedMatcher()
matches = matcher.knnMatch(des1,des2,k=2)

# important features
imp = []
for i, (one, two) in enumerate(matches):
    if one.distance < dist_threshold*two.distance:
    imp.append((one.trainIdx, one.queryIdx))
```

Step 4] Calculate the homography

Now that we have matching points identified in both images, we will use them to get the *generic relationship* between images. This ***relationship*** is defined in the literature as homography. It is a relationship between image1 and image2 described as a matrix.

We need to ***transform*** one image into other image's space using the homography matrix. We can do either way from image 1 to 2 or image 2 to 1 since the homography matrix (3x3 in this case) will be square and non-singular. Only thing is that we need to be consistent while passing points to homography.

I have used my own RANSAC based approach to get a Homography matrix. But you can also use inbuilt OpenCV function *cv2.findHomography*()

```
### RANSAC
def ransac_calibrate(real_points , image_points, total_points,
image_path, iterations):

index_list = list(range(total_points))
    iterations = min(total_points - 1, iterations)
    errors = list(np.zeros(iterations))
    combinations = []
    p_estimations=[]

for i in range(iterations):
        selected = random.sample(index_list,4)
        combinations.append(selected)
        real_selected =[]
        image_selected =[]

for x in selected:
            real_selected.append(real_points[x])
            image_selected.append(image_points[x])

p_estimated = dlt_calibrate(real_selected, image_selected, 4)

not_selected = list(set(index_list) - set(selected))
        error = 0
        for num in tqdm(not_selected):

# get points from the estimation
            test_point = list(real_points[num])
```

```
            test_point = [int(x) for x in test_point]
            test_point = test_point + [1]

try:
                xest, yest = calculate_image_point(p_estimated,
np.array(test_point), image_path)
            except ValueError:

                continue

error = error + np.square(abs(np.array(image_points[num])-
np.asarray([xest,yest])))
#            print("estimated  :",np.array([xest, yest])  )
#            print("actual :",image_points[0])
#            print("error :",error)
        errors.append(np.mean(error))
        p_estimations.append(p_estimated)

p_final = p_estimations[errors.index(min(errors))]
    return p_final ,errors, p_estimations
```

Step 5] Transform images into the same space

Compute the second image transformed coordinates using the homography matrix output of step 4.

> ***image2_transformed = H*image2***

Step 6] Let's do the stitching...

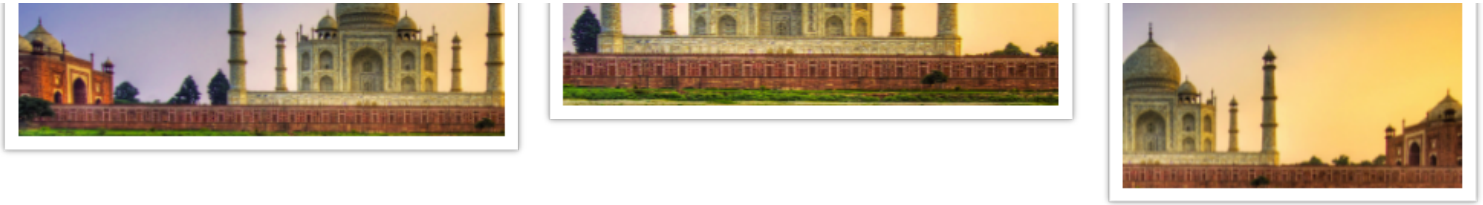
After computing transformed images we get two images each having some information separate and some common w.r.t. other. When it is separate the other image will have 0 intensity value at the corresponding location

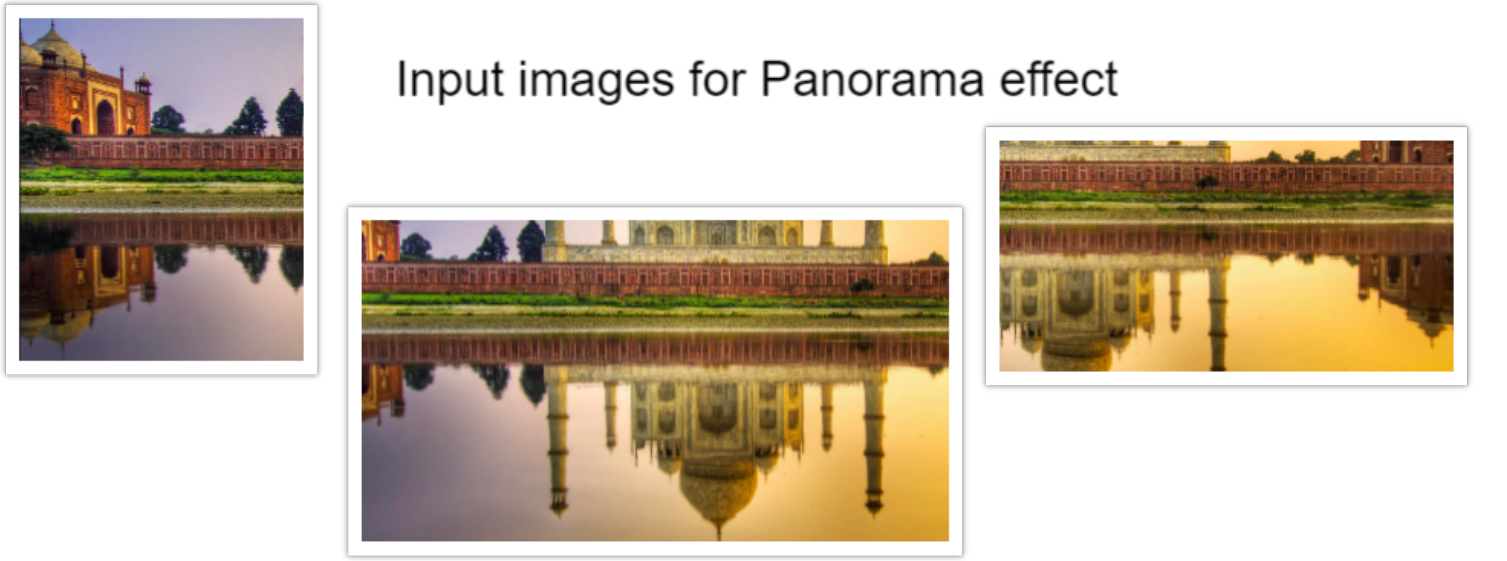
We need to fuse with the help of this info at every location while keeping overlapping info intact.

It is a pixel level operation which technically can be optimized by doing image level add, subtract, bitwise_and, etc, but I found the output was getting compromised in doing this manipulation. Output generated with old-school for-loop based approach to choose the maximum pixel from the corresponding input pixels worked better.

```
## get maximum of 2 images
for ch in tqdm(range(3)):
    for x in range(0, h):
        for y in range(0, w):
            final_out[x, y, ch] = max(out[x,y, ch], i1_mask[x,y,
ch])
```

See the output below:




Input images for Panorama effect

Although the post was just a small guide, you can refer to the entire code available here.

Machine Learning | Image Processing | Computer Vision | Panorama | Panoramic Image Stitching