Image Stitching using OpenCV and Python (Creating Panorama Project)

In this project, we will use OpenCV with Python and Matplotlib in order to merge two images and form a panorama.

As you know, the Google photos app has stunning automatic features like video making, panorama stitching, collage making, and many more. In this exercise, we will understand how to make a panorama stitching using OpenCV with Python.

Skills used:

1. OpenCV
2. Python
3. Matplotlib

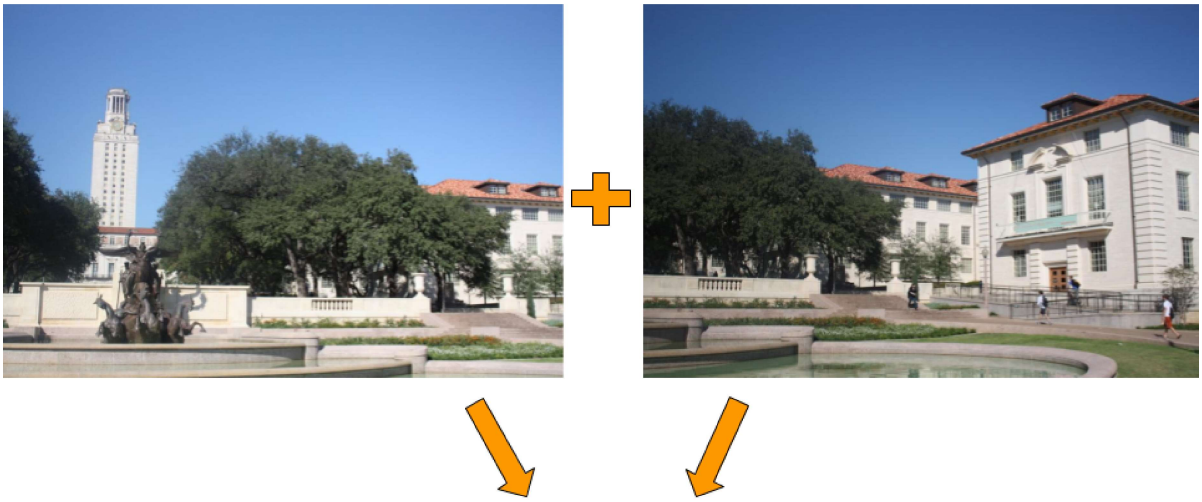
About the Project

Image Stitching may also be termed as Panorama completion.

Given two or more images of the same scenic view but might be caught from the same or different perspectives of the camera, the resultant image is expected to view the complete scene by merging the images on the basis of common image parts between one-another.

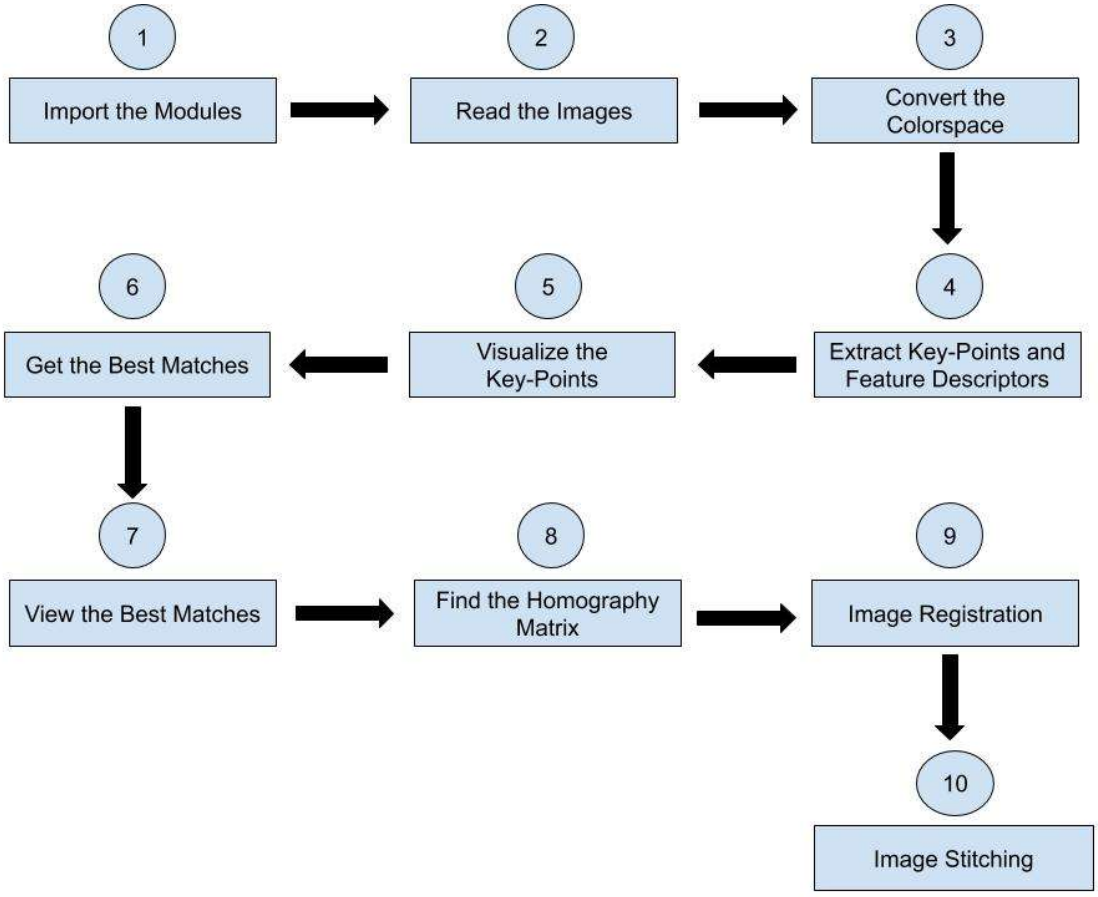
This is commonly coined as the image stitching problem.

In our current exercise, we shall merge two images to form the panorama depicted as follows. We shall mainly use OpenCV with Python and Matplotlib to do this.



Stitched Image is the resultant of merging the above two images to complete the panorama

WORKFLOW:



In [1]:
```
import cv2
cv2.__version__
```

Out[1]:
```
'3.4.4'
```

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Reading the Images

Now, let us read the images we want to stitch together to create a panorama.

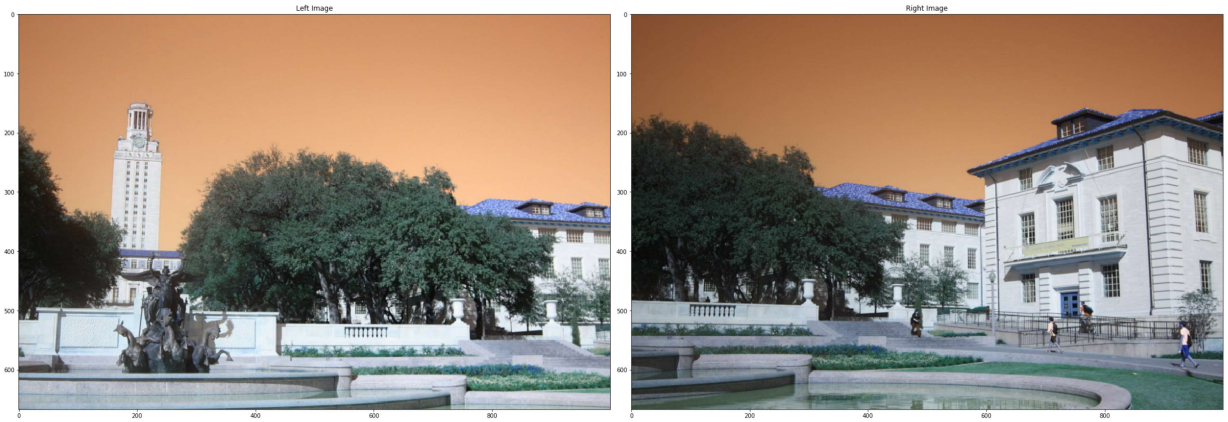
In [3]:
```
img_right = cv2.imread('/cxldata/projects/uttower_right.jpg')
img_left = cv2.imread('/cxldata/projects/uttower_left.jpg')
```

In [4]:
```
plt.figure(figsize=(30,20))

plt.subplot(1,2,1)
plt.title("Left Image")
plt.imshow(img_left)

plt.subplot(1,2,2)
plt.title("Right Image")
plt.imshow(img_right)


plt.tight_layout()
```



BGR to RGB and Grayscale

By default, a color image with Red, Green, and Blue channels will be read in reverse order; ie, Blue, Green, and Red by OpenCv.

We could fix this issue by using cv2.COLOR_BGR2RGB transformation effect on the image.

So, we shall define a function fixColor to return the RGB form of the given image.

Also, let us get the gray-scle form of the two images. Generally, because of the reduced complexity of Grayscale form over the RGB format, grayscale images are preferred to process the images.

In [5]:
```
def fixColor(image):
    return(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
```

In [6]:
```
img1 = cv2.cvtColor(img_right, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor( img_left, cv2.COLOR_BGR2GRAY)
```

In [7]:
```
plt.figure(figsize=(30,20))

plt.subplot(2,2,1)
plt.title("Left Image")
plt.imshow(fixColor(img_left))

plt.subplot(2,2,2)
plt.title("Grayscale of Left Image")
plt.imshow(img2)

plt.subplot(2,2,3)
plt.title("Right Image")
plt.imshow(fixColor(img_right))

plt.subplot(2,2,4)
plt.title("Grayscale of Right Image")
plt.imshow(img1)

plt.tight_layout()
```

Note:

cv2.cvtColor() method is used to convert an image from one color space to another.

cv2.COLOR_BGR2RGB returns image in RGB format, which was initially in BGR format as read by cv2.imread().

cv2.COLOR_BGR2GRAY returns image in Grayscale format, which was initially in BGR format as read by cv2.imread().

Getting Key-Points and Feature Descriptors

In order to stitch two images, we should first find out the matching image features between both of them, so that we could stitch them based on these features.

Image features are small patches that are useful to compute similarities between images. An image feature is usually composed of a feature keypoint and a feature descriptor.

The key point usually contains the patch 2D position and other stuff if available such as scale and orientation of the image feature.

The descriptor contains the visual description of the patch and is used to compare the similarity between image features.

Let us use ORB algorithm from OpenCV to extract key-points and descriptors for each image. Though there are other alternatives to ORB - like SIFT and SURF - ORB wins over them as it is as efficient as them in computation-wise, performance-wise and ost importantly patent-wise. ORB is free while SIFT and SURF are patented.

In [8]:
```
#Create an object of cv2.ORB_create() and let it be named as orb.
orb = cv2.ORB_create()
```

In [9]:
```
#Now use detectAndCompute method of this orb object and pass img1 as an argument to
kp1, des1 = orb.detectAndCompute(img1,None)
```

In [10]:
```
kp2, des2 = orb.detectAndCompute(img2,None)
```

note: cv2.ORB_create() which creates an object for ORB. We could use this object to get the key-points and descriptors for the images using detectAndCompute method of orb object.

Visualizing Key-Points on the Images

Let us view the original image and the one with key-points marked to see what key-points were extracted.

In [11]:
```
img_right_kp = cv2.drawKeypoints(img_right, kp1, np.array([]), color=(0,0, 255))
```

Get the img_right_kp, the resultant of plotting img_right with its key-points kp1 drawn using cv2.drawKeypoints. Observe, we pass np.array([]) in place of output image. The np.array([]) will be the output image with key-points drawn. If you want to draw the key-points to be drawn on the same input image, you could replace it with that image. In order to maintain modularity, let's pass a new empty NumPy array.

In [12]:
```
img_left_kp = cv2.drawKeypoints(img_left, kp2, np.array([]), color=(0,0, 255))
```

Get the img_left_kp, the resultant of plotting img_left with its key-points kp2 drawn in cv2.drawKeypoints.

In [13]:
```
#Visualize the img_left_kp and img_right_kp side-by-side.
```

```
plt.figure(figsize=(30,20))
plt.subplot(1,2,1)
plt.imshow(fixColor(img_left_kp ))

plt.subplot(1,2,2)
plt.imshow(fixColor(img_right_kp ))
plt.tight_layout()
```

Getting the Best Matches

The obtained descriptors in one image are to be recognized in the other image too. This is because, once the matching features are recognized, the images could be stitched based on these matching features.

Now, we are mainly going to do the following:

- Use BFMatcher():
  - The Brute-Force matcher(cv2.BFMatcher()) is simple. It takes the descriptor of one feature in the first set and is matched with all other features in the second set using some distance calculation. And the closest one is returned.
  - For BF matcher, first we have to create the BFMatcher object using cv2.BFMatcher(). It takes two optional params:
    - First param is normType: It specifies the distance measurement to be used. For descriptors like ORB, BRIEF, BRISK etc, cv2.NORM_HAMMING should be used, which used Hamming distance as measurement.
    - Second param is a boolean variable, crossCheck: This is false by default. If it is true, Matcher returns only those matches with value (i,j) such that i-th descriptor in set A has j-th descriptor in set B as the best match and vice-versa. That is, the two features in both sets should match each other. It provides consistent results.
- Use match method of BFMatcher object: : By calling this method, we will be returned the best matches.

In [17]:
```
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

In [19]:
```
#Call the match method of bf object and pass the descriptors of the 2 images as inpu
matches = bf.match(des1,des2)
```

`matches` is a list of `DMatch` objects. This DMatch object has the following attributes:

- DMatch.distance - Distance between descriptors. The lower, the better it is.
- DMatch.trainIdx - Index of the descriptor in train descriptors (des1)
- DMatch.queryIdx - Index of the descriptor in query descriptors (des2)
- DMatch.imgIdx - Index of the train image(right image)

In [20]:
```
matches = sorted(matches, key = lambda x:x.distance)
```

Viewing the Best Matches

From all the keypoints extracted by the BFMatcher, let us plot and view 30 of the best common key-points between the left and right images.

We shall do this in 3 steps:

- Create a dictionary `draw_params` which mentions the color of the lines marking the matches between the images, and `flags=2` which says to show only those 30 key-points and not any other key-points. (You could experiment this by removing flags)
- Use `cv2.drawMatches` which returns the image drawn with the 30 common key-points between the right and left images `img_right` and `img_left`, as per the properties mentioned in `draw_params`. We need to pass the `fixColor(img_right), kp1, fixColor(img_left), kp2, matches[:30]` as arguments for this method.
- Finally, display the image returned by `cv2.drawMatches` using matplotlib's `plt.imshow`.

In [25]:
```
draw_params = dict(matchColor = (255,255,0), # draw matches in yellow color
                   flags = 2)
```

Here, we have chosen to use yellow color to draw the matches, and used flags=2 that indicates to show only those 30 key-points which are being drawn now and don't show others for a neater look.

In [26]:
```
#Use the following code to get the image with 30 of the common key-points matched be
matched_features_image = cv2.drawMatches(fixColor(img_right), kp1, fixColor(img_left

plt.figure(figsize=(30,20))
plt.imshow(matched_features_image)
```

Out[26]:
```
<matplotlib.image.AxesImage at 0x7f5d6064b6a0>
```

Finding Homography Matrix

It's time to align the images now.

We need to align the images because, though both images could form a panorama, the two could differ in terms of angle, translation, size, etc, probably caused due to orientational difference in the camera while capturing the photos. Thus, we need to find a transformation matrix to perform this alignment, which ensures compatibility to stitch and form a panorama.

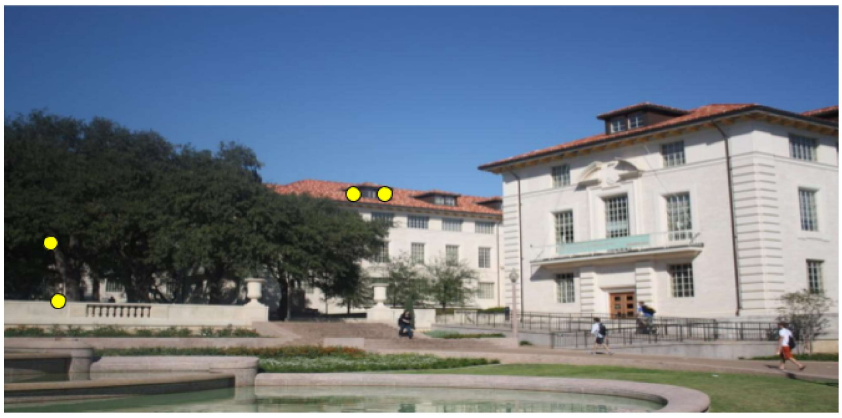
A Homography Matrix is a 3x3 transformation matrix that maps the points in one image to the corresponding points in the other image.

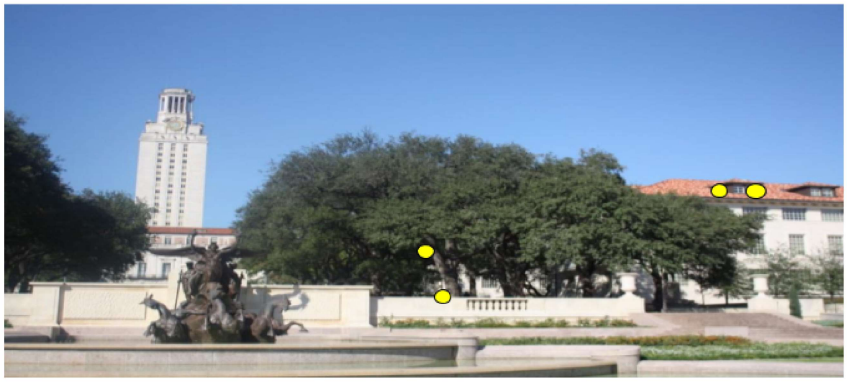
$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

where (x1, y1) and (x2, y2) could be thought of the co-ordinates of the matching key-points in both the images respectively.

A homography matrix is used with the best matching points, to estimate a relative orientation transformation within the two images. Simply put, it is the perspective transformation matrix between two planes - here, the two images.

In order to find a homography matrix, at least 4 matching key-points are needed.





Note:

cv2.findHomography finds a perspective transformation between two planes - here, the two images.

cv2.RANSAC is a method used to compute a homography matrix.

In [29]:
```
#Get the best matches - matches - between the right and left images. If there are at
if len(matches) >= 4:
    src = np.float32([ kp1[m.queryIdx].pt for m in matches ]).reshape(-1,1,2)
    dst = np.float32([ kp2[m.trainIdx].pt for m in matches ]).reshape(-1,1,2)

    H, masked = cv2.findHomography(src, dst, cv2.RANSAC, 5.0)
else:
    raise AssertionError("Can't find enough keypoints.")
```

Image Registration

Now that we found the homography for transformation, we can now proceed to warp and display the right-side image in an orientation suitable to be stitched with the left image. This is also known as image registration.

Note:

`cv2.warpPerspective` applies a perspective transformation to an image, given the image, homography matrix, and size of the output image and returns the transformed output image.

In [33]:
```
dst = cv2.warpPerspective(img_right,H,(img_left.shape[1] + img_right.shape[1], img_l
```

In [34]:
```
#Visualize the output image dst.
plt.figure(figsize=(30,20))
plt.title('Warped Image')
plt.imshow(fixColor(dst))
```

Out[34]:
```
<matplotlib.image.AxesImage at 0x7f5d6061d0b8>
```

Stitching the Images

And finally comes the last part, the stitching of the images.

We shall place the left image on the `dst` whose shape is left-image width plus right-mage width.

Then, we shall save the resultant file - the final stitched panorama image - and display it.

Note:

`cv2.imwrite` method saves the given image as a file with the given name.

In [37]:
```
#We are placing the left image on the dst whose shape is left-image width plus right
dst[0:img_left.shape[0], 0:img_left.shape[1]] = img_left
```

In [38]:
```
#Now let us store the resultant stitched image dst as resultant_stitched_panorama.jp
cv2.imwrite('resultant_stitched_panorama.jpg',dst)
```

Out[38]:
```
True
```

In [39]:
```
plt.figure(figsize=(30,20))
plt.title('Stitched Image')
plt.imshow(fixColor(dst))
```

Out[39]:
```
<matplotlib.image.AxesImage at 0x7f5d60579198>
```


Stitched Image
0
100
200
300
400
500
600
0
250
500
750
1000
1250
1500
1750

In [ ]: