# EN-2550 Assignment 2

## Name : Ekanayaka S.D.

## Index No : 190162F

### GitHub Repository: [https://github.com/sasdil/EN-2550-Computer-Vision](https://github.com/sasdil/EN-2550-Computer-Vision)

---

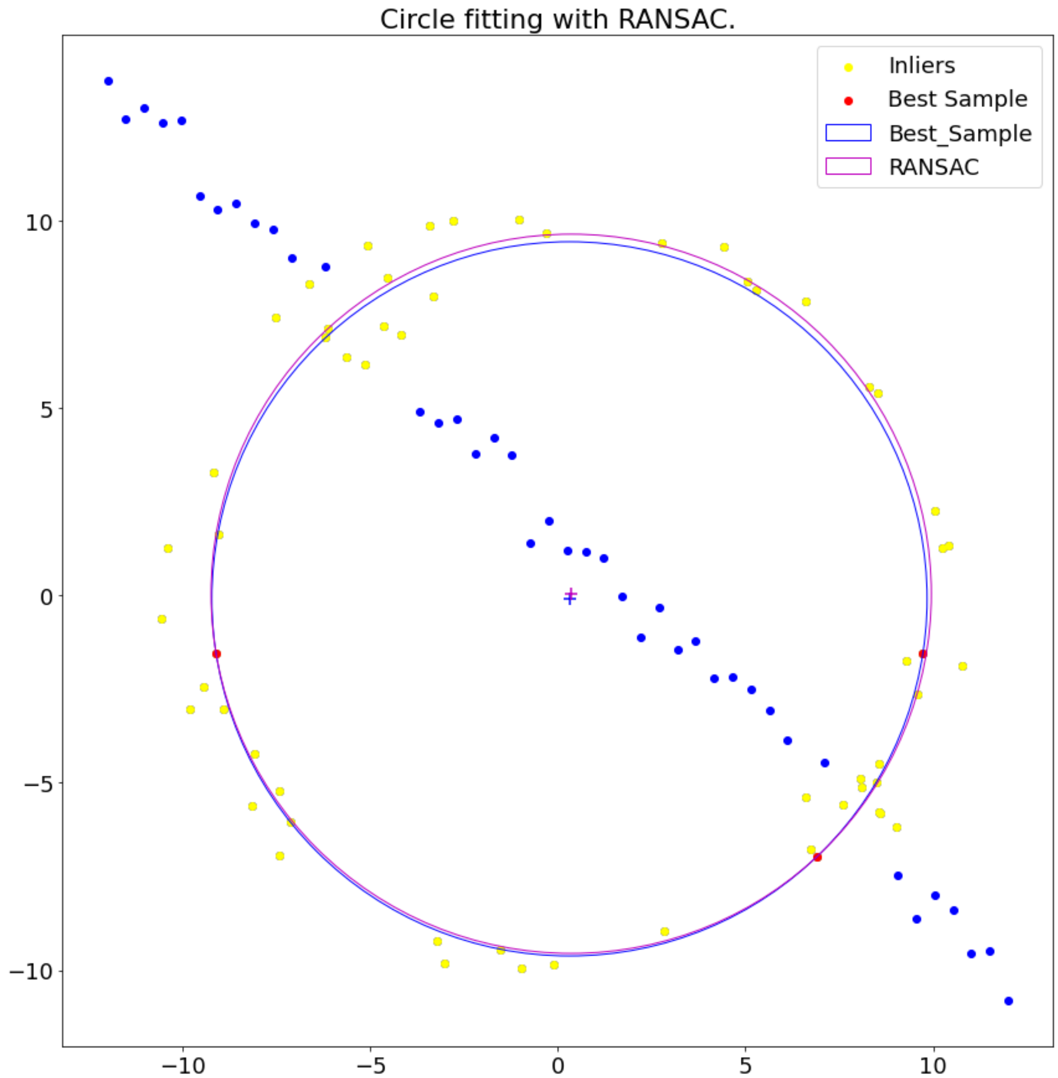
# Question 1

---

In below code, I implementing the RANSAC algorithm for Circle fitting. So for randomly choosed sample which consist with 3 point coordinates, we estimate the circle which go through those points. Then set a threshold value for check which circle is consist with most inliers.That belongs to best sample and then we put those inliers and best sample coordinates to get the RANSAC circle, So it is little bit deviate from best sample circle.

In [6]:

```python
#Class Object for Generating All required functions for RANSAC
class RANSAC_gen:
    def __init__(self, x_data, y_data, n):
        self.x_data = x_data
        self.y_data = y_data
        self.n = n
        self.d_max=15
        self.best_model = None
        self.point= None
        self.mod = None
        self.inliers = None
    #Function for Randomly take 3 points sample
    def random_sampling(self):
        sample = []
        save_ran = []
        count = 0

        # get three points from data
        while True:
            ran = np.random.randint(len(self.x_data))
            if ran not in save_ran:
                sample.append((self.x_data[ran], self.y_data[ran]))
                save_ran.append(ran)
                count += 1
                if count == 3:
                    break
        return sample
    #Function for genarate respective model
    def make_model(self, sample):

        pt1 = sample[0]
        pt2 = sample[1]
        pt3 = sample[2]

        A = np.array([[pt2[0] - pt1[0], pt2[1] - pt1[1]], [pt3[0] - pt2[0], pt3[1] - pt2[1]]])
        B = np.array([[pt2[0]**2 - pt1[0]**2 + pt2[1]**2 - pt1[1]**2], [pt3[0]**2 - pt2[0]**2 + pt3[1]**2 - pt2[1]**2]])
        inv_A = inv(A)

        c_x, c_y = np.dot(inv_A, B) / 2

        c_x, c_y = c_x[0], c_y[0]
        r = np.sqrt((c_x - pt1[0])**2 + (c_y - pt1[1])**2)
        return c_x, c_y, r
    #Function for filter out inliers
    def get_inliers(self,cx,cy,r):
        P=[]
        t=1.4
        xd=self.x_data
        yd=self.y_data
        for i in range(len(xd)):
            dis = np.sqrt((xd[i]-cx)**2 + (yd[i]-cy)**2)
            if (r-t<=abs(dis)<=r+t):
                P.append([xd[i],yd[i]])
        return (P)
    def eval_model(self, model):
        c_x, c_y, r = model
        P = self.get_inliers(c_x,c_y,r)
        return P
    #Find the best model by excuting functions
    def execute_ransac(self):
        # find best model
        for i in range(self.n):
            mod=self.random_sampling()
            model = self.make_model(mod)
            c_x, c_y, r = model
            d_temp = self.eval_model(model)
            if self.d_max < len(d_temp):
                self.best_model = model
                self.d_max = len(d_temp)
                self.mode = mod
                self.inliers = d_temp
```




59

## Question 2

In here first we take 4 point coordinates given by user.(In class object 'Click' function is belongs to that task). These points refer to destination points which use to compute homography later. Afterwards we take vertices of source image as source points and then calculate the homography using 'cv.findHomography()' inbuilt function. Resulting images are shown in the below.

The blending of the image is done by using "cv2.addWeighted()" inBuilt fuction to make necessary adjustments of the final stitched image. The Important code parts are given below.

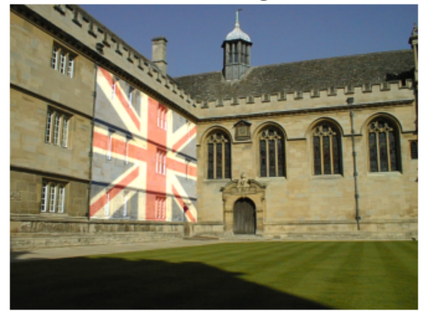
In [7]:

```
class warp_gen:
    def __init__(self,im1,im2,count,Points):
        self.im1 = im1
        self.im2 = im2
        self.count = count
        self.Points = Points
    #function for get coordinates according to mouse clicks
    def Click(self,event,x,y,flags,param):
        if event == cv2.EVENT_LBUTTONDOWN:
            self.Points.append([x,y])
            self.count+=1
    #Function for homography calcultions and warpping/ Blending of the image
    def process(self):
        wname = "Image"
        cv2.namedWindow(winname=wname)
        cv2.setMouseCallback(wname, self.Click)
        while self.count<4 :
            cv2.imshow(wname,self.im2)
            cv2.waitKey(1)
        cv2.destroyAllWindows()
        if (len(self.Points)==4):
            im_src = self.im1
            h, w, c = im_src.shape
            pts_src = np.array([[0,0],[w-1, 0],[w-1, h-1],[0,h-1]])
            im_dst =self.im2
            pts_dst = np.array(self.Points)
            h, status = cv2.findHomography(pts_src, pts_dst)
            im_out = cv2.warpPerspective(im_src, h, (im_dst.shape[1],im_dst.shape[0]))

            #Blend the image to get finale proper image
            result = cv2.addWeighted(im_dst,1,im_out ,0.4, 0)
```

source image

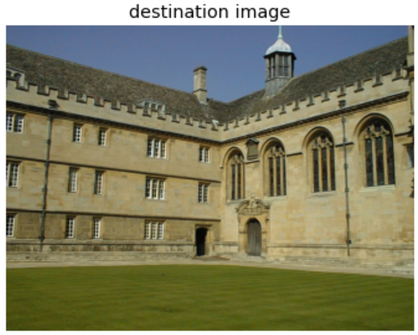
destination image

Blended image

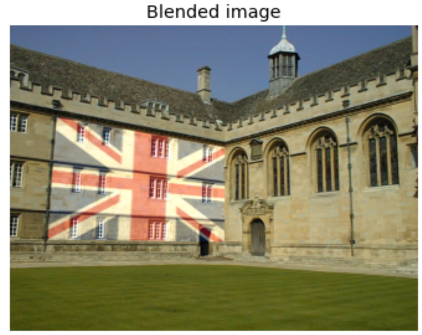


source image

destination image



Blended image



---

# Question 3

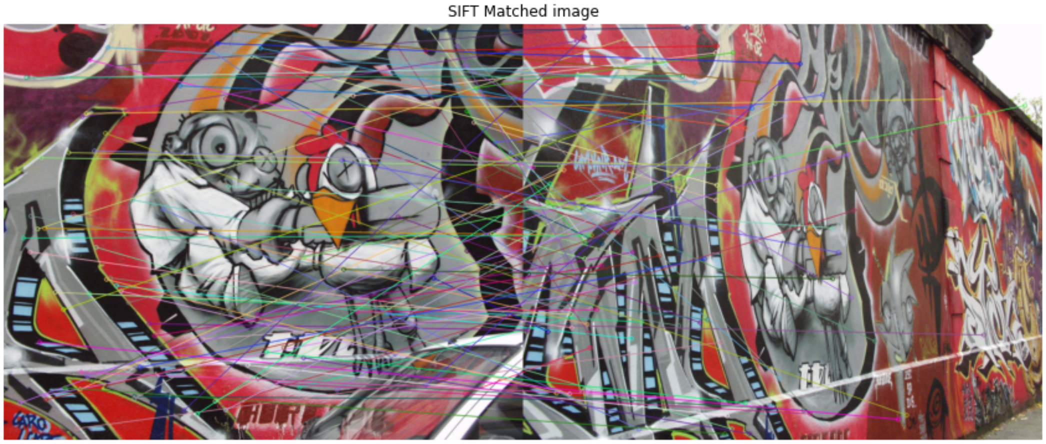
---

## Part (A)

SIFT features of two images matched using following code. Mainly this function returns the matches and keypoints as outputs.

In [1]:
```
def siftmatch(img1,img2):
    sift = cv.SIFT_create()
    kp1, descriptors_1 = sift.detectAndCompute(img1, None)
    kp2, descriptors_2 = sift.detectAndCompute(img2, None)
    bf1 = cv.BFMatcher(cv.NORM_L1, crossCheck = True)
    matches1 = bf1.match(descriptors_1, descriptors_2)
    sortmatches1 = sorted(matches1, key = lambda x:x.distance)

    return matches1,[kp1,kp2]
```

SIFT Matched image



## Part (B) & Part (C)

In here basically I calculate homography of image 1 to 5 using saperate homograpies of imagea 1 to 2 , 2 to 3, 3 to 4, 4 to 5.Then we multiply those homography matrices reversely to obtain the 1 to 5 image homography. It is very hard to calculate the homography of the image 1 to 5 directly. Because the homography of those two iamges is very high. So using Homography function, we can calculate homography matrix according to the RANSAC algorithm. The main code parts are given below.

In [7]:
```
def SSD(corres, h):
    pts1 = np.transpose(np.matrix([corres[0].item(0), corres[0].item(1), 1]))
    estimatep1 = np.dot(h, pts1)
    estimatep2 = (1/estimatep1.item(2))*estimatep1
    pts2 = np.transpose(np.matrix([corres[0].item(2), corres[0].item(3), 1]))
    error = pts2 - estimatep2
    return np.linalg.norm(error)
def Homography(correspondences):
    #Loop through correspondences and create assemble matrix
    Lst = []
    for corr in correspondences:
        p1 = np.matrix([corr.item(0), corr.item(1), 1])
        p2 = np.matrix([corr.item(2), corr.item(3), 1])

        a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2),
```

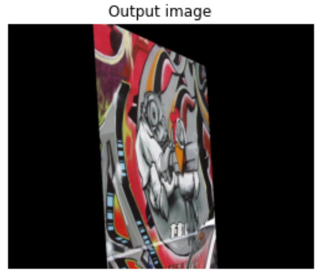
```
                p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
        a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2), 0, 0, 0,
                p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
        Lst.append(a1)
        Lst.append(a2)
    matrixA = np.matrix(Lst)
    #svd composition
    u, s, v = np.linalg.svd(matrixA)
    #reshape the min singular value into a 3 by 3 matrix
    h = np.reshape(v[8], (3, 3))
    #normalize and now we have h
    h = (1/h.item(8)) * h
    return h
def ransac(corr, thresh):
    maxInliers = []
    finalH = None
    for i in range(1000):
        #find 4 random points to calculate a homography
        corr1 = corr[random.randrange(0, len(corr))]
        corr2 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((corr1, corr2))
        corr3 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr3))
        corr4 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr4))
        #call the homography function on those points
        h = Homography(randomFour)
        inliers = []
        for i in range(len(corr)):
            d = SSD(corr[i], h)
            if d < 5:
                inliers.append(corr[i])
        if len(inliers) > len(maxInliers):
            maxInliers = inliers
            finalH = h
        if len(maxInliers) > (len(corr)*thresh):
            break
    return finalH, maxInliers
def corr_list(matches1,key):
    correspondenceList1 = []
    keypoints1 = [key[0],key[1]]
    for match in matches1:
        (x1, y1) = keypoints1[0][match.queryIdx].pt
        (x2, y2) = keypoints1[1][match.trainIdx].pt
        correspondenceList1.append([x1, y1, x2, y2])
    return correspondenceList1
#calculate homographies
match1,ky1=siftmatch(img1,img2)
correspondenceList1=corr_list(match1,ky1)
corrs1 = np.matrix(correspondenceList1)
finalH1, inliers1 = ransac(corrs1, 0.6)
match2,ky2=siftmatch(img2,img3)
correspondenceList2=corr_list(match2,ky2)
corrs2 = np.matrix(correspondenceList2)
finalH2, inliers2 = ransac(corrs2, 0.6)
match3,ky3=siftmatch(img3,img4)
correspondenceList3=corr_list(match3,ky3)
corrs3 = np.matrix(correspondenceList3)
finalH3, inliers3 = ransac(corrs3, 0.6)
match4,ky4=siftmatch(img4,img5)
correspondenceList4=corr_list(match4,ky4)
corrs4 = np.matrix(correspondenceList4)
finalH4, inliers4 = ransac(corrs4, 0.6)
#Obtaining the homography matrix of 1 to 5
H = finalH4 @ finalH3 @ finalH2 @ finalH1
print(H)
```

```
[[ 6.51222636e-01  7.03255113e-02  2.20540605e+02]
 [ 2.31063212e-01  1.19780873e+00 -2.55386339e+01]
 [ 5.43289009e-04 -4.18605266e-06  1.00140169e+00]]
```

source image

destination image

Output image



After calculating the homography using above RANSAC algorithm, in here we campare it with the actual homography matrix to observe the accuracy of the above code. For that we get the Sum of Square Difference between those two matrix.So we achieve reasonable value for it.

In [13]:

```
Original_Homography = [ [6.2544644e-01,5.7759174e-02,2.2201217e+02],
    [2.2240536e-01,1.1652147e+00,-2.5605611e+01],
    [4.9212545e-04,-3.6542424e-05,1.0000000e+00]]
Calculated_Homography = [[ 6.51222636e-01 , 7.03255113e-02 , 2.20540605e+02],
 [ 2.31063212e-01 , 1.19780873e+00, -2.55386339e+01],
 [ 5.43289009e-04, -4.18605266e-06 , 1.00140169e+00]]
Original_Homography =np.array(Original_Homography)
Calculated_Homography = np.array(Calculated_Homography)

SSD_Calc= np.sum(np.sum((Original_Homography-Calculated_Homography)*(Original_Homography-Calculated_Homography)))
print("SSD Value =",SSD_Calc)
```

```
SSD Value = 2.171951103855168
```