In [ ]:

```
Index Number: 190108X                          Name: Chathuranga M.M.P.                          Assignment 2
```

GitHub Link: https://github.com/ChathurangaMMP/EN2550_Image_Processing_Exercises/tree/master/Assignment%202

Question 1

The RANSAC algorithm is created as an object and it has used to estimate a circle.

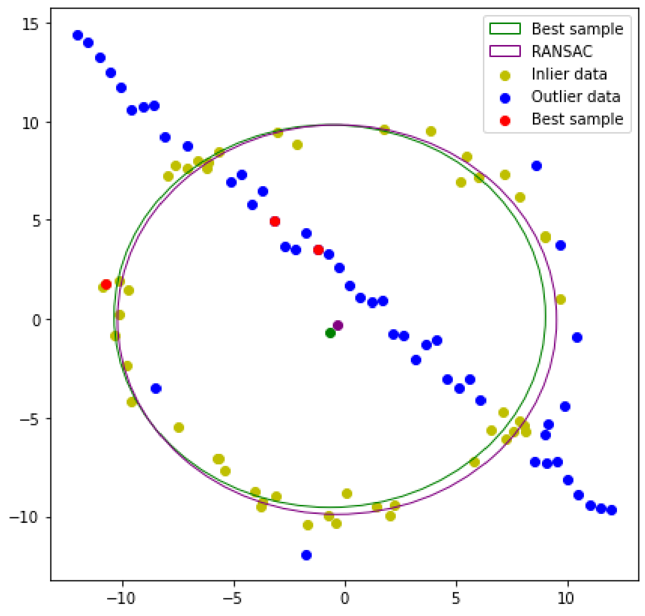
In [3]:

```python
class RANSAC:
    '''this class performs the RANSAC algorithm'''
    def __init__(self,x_data,y_data,min_samples,dis_thres,num_samps):
        # initialize the variables
        self.x_data=x_data
        self.y_data=y_data
        self.s=min_samples # minimum number of points that need to choose
        self.t=dis_thres # the thresh hold distance for comparison
        self.N=num_samps # the number of samples required to get reasonable accuracy
        self.best_model=None
        self.inliers=[]
        self.outliers=[]
        self.points=[]
    def distance(self,p1,p2): # calculate the distance between two points
        return ((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)**0.5
    def random_sample(self): # generate random sample of data points
        self.points=[]
        i=0
        while i<self.s:
            ind=random.randint(0,len(self.x_data)-1)
            pnt=(self.x_data[ind],self.y_data[ind])
            if pnt not in self.points: # avoid duplicating points
                self.points.append(pnt)
                i+=1
    def circle_equation(self): # calculate the equation of a circle
        pt1,pt2,pt3=self.points
        Y = np.array([[pt2[0] - pt1[0], pt2[1] - pt1[1]], [pt3[0] - pt2[0], pt3[1] - pt2[1]]])
        Z = 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]])
        inverse_Y = linalg.inv(Y)
        c_x, c_y = np.dot(inverse_Y, Z) / 2
        cx, cy = c_x[0], c_y[0]
        r = np.sqrt((cx - pt1[0])**2 + (cy - pt1[1])**2)
        return cx, cy, r # return the center coordinates and the radius
    def inliers_filter(self):
        # calculate the inliers and outlier
        new_inliers=[]
        new_outliers=[]
        xc,yc,r=self.circle_equation()
        for i in range(len(self.x_data)):
            dist=self.distance((self.x_data[i],self.y_data[i]),(xc,yc))
            if abs(dist-r)<=self.t:
                new_inliers.append((self.x_data[i],self.y_data[i]))
            else:
                new_outliers.append((self.x_data[i],self.y_data[i]))
        if len(self.inliers)<len(new_inliers): # update the best model using calculated data
            self.inliers=new_inliers
            self.outliers=new_outliers
            self.best_model=(xc,yc,r)
    def model_finder(self): # find the best fitting model by iterating N times
        for i in range(self.N):
            self.random_sample()
            self.inliers_filter()
        return self.best_model

# finding the best sample
best_samp=RANSAC(X[:,0],X[:,1],3,1,35)
best_samp.model_finder()
# using the data of the best sample, find the RANSAC estimation
ransac=RANSAC(np.array(best_samp.inliers)[:,0],np.array(best_samp.inliers)[:,1],3,1,35)
ransac.model_finder()
```

In [6]:

```python
'''The RANSAC circle estimation is done by applying the RANSAC algorithm over the best sample that found early.
As you can see, the both estimations are very similar and the inliers and outliers have selected very accurately
by the implemented algorithm.'''
```

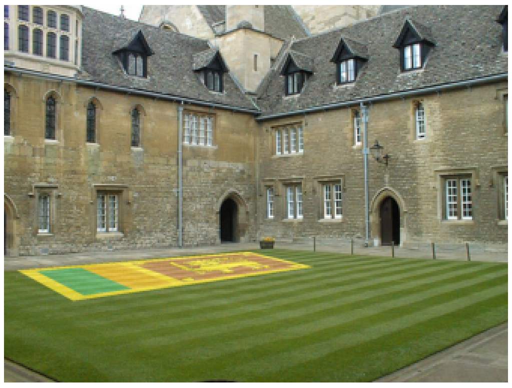


Question 2

In this approach, user can choose the position where the second image needs to superimpose, by mouse clicking. The image slection for this task is done by considering the flat spaces, walls, planes in the images. Then we can properly put the images into that position.

In [8]:
```
dst_points=[]
def get_coord(img_name):
    # this function gets the required positions as mouse clicked points from the user
    global dst_points
    img=cv.imread(img_name)
    count=0
    def click_event(event, x, y, flags, params):
        # track the left button click of the mouse
        if event==cv.EVENT_LBUTTONDOWN:
            dst_points.append([x,y]) # get the point
            cv.circle(img, (x,y), 2, [0,0,255], 2) # draw a small dot in the clicked position
            cv.imshow('image', img)
    # display the image and get mouse clicks
    cv.namedWindow('image',cv.WINDOW_AUTOSIZE)
    cv.imshow('image', img)
    cv.setMouseCallback('image', click_event)
    while count<4:
        cv.waitKey(1)
        count+=1
    cv.waitKey(0)
    cv.destroyAllWindows()
    return dst_points
def homography(img_fg,img_bg):
    # find the homography and combine two images
    fh,fg=img_fg.shape[0],img_fg.shape[1]
    pts_src = np.array([[0, 0], [0, fg], [fh, 0],[fh, fg]])
    pts_dst = np.array(get_coord('hall.jpg'))
    h, status = cv.findHomography(pts_src, pts_dst)
    im_out = cv.warpPerspective(img_fg, h, (img_bg.shape[1],img_bg.shape[0]))
    return im_out
def plot_align(img_fg,img_bg): # only the important line is shown here
    ax.imshow(cv.cvtColor(cv.addWeighted(img_bg,1,homography(img_fg,img_bg),0.5,0), cv.COLOR_BGR2RGB))
```

In [8]:
```
plot_align(img_flag,img_hall)
```



Question 3 (a)

In [10]:
```
sift = cv.SIFT_create() # make the SIFT object
```

```python
# using the SIFT object calculate the key points and destination points
key_p1, des1 = sift.detectAndCompute(gray1,None)
key_p4, des4 = sift.detectAndCompute(gray4,None)
key_p5, des5 = sift.detectAndCompute(gray5,None)
bf = cv.BFMatcher() # make the matcher object
# using this matcher, match the destination points
matches15, matches14, matches45 = bf.knnMatch(des1,des5,k=2), bf.knnMatch(des1,des4,k=2), bf.knnMatch(des4,des5,k=2)
# filter out the best matches after comparing with a thresh hold
good15, good14, good45 = [], [], []
for m,n in matches15:
    if m.distance < 0.8*n.distance: good15.append([m])
for m,n in matches14:
    if m.distance < 0.75*n.distance: good14.append([m])
for m,n in matches45:
    if m.distance < 0.75*n.distance: good45.append([m])
img_new = cv.drawMatchesKnn(img1,key_p1,img5,key_p5,good15,None,flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

Question 3 (b)

When we compute the homography matrix directly using the images 1 and 5, the resulting matrix has many differences compared to the given homography matrix in the dataset and the stitched image is also not accurate. This result can be found in the last image of the 'Figure 01'.

Hence, rather than computing the homography matrix directly using the images 1 and 5, in here, the matrix has calculated in two steps. First, the images 1 and 4 are selected and computed the homography matrix of it. Then the matrix is calculated by considering the images 4 and 5. The multiplication of these two matrices gives the homography matrix for the images 1 and 5. It is more accurate compaired to previous matrix. When we compare the stitched image of this approach (Figure 02) with the image which is stitched using the given homography matrix in the dataset, we can see that they are very close to each other. The Sum of Square Differences(SSD) value comparison between these two generated matrices and the given matrix in the dataset reflects the above result. The H matrix used for calculation in Figure 01 gives a higher SSD value compaired to the H matrix used for calculation in Figure 02.

If we calculate the homography matrices for the image pairs 1&2,2&3,3&4,4&5 and multiply them(H_4&5 @ H_3&4 @ H_2&3 @ H_1&2) to form a homography matrix, it also gives a similar result to our second method, but it needs more computational power.

In [11]:
```python
dst_pts15 = np.float32([key_p1[m[0].queryIdx].pt for m in good15]).reshape(-1, 2)
src_pts15 = np.float32([key_p5[m[0].trainIdx].pt for m in good15]).reshape(-1, 2)
dst_pts14 = np.float32([key_p1[m[0].queryIdx].pt for m in good14]).reshape(-1, 2)
src_pts14 = np.float32([key_p4[m[0].trainIdx].pt for m in good14]).reshape(-1, 2)
dst_pts45 = np.float32([key_p4[m[0].queryIdx].pt for m in good45]).reshape(-1, 2)
src_pts45 = np.float32([key_p5[m[0].trainIdx].pt for m in good45]).reshape(-1, 2)
def random_points(src_pts, dst_pts, N):
    # generate N number of random points from 'src_pts' and 'dst_pts'
    rnd = np.random.choice(len(src_pts), N)
    src = [src_pts[i] for i in rnd]
    dst = [dst_pts[i] for i in rnd]
    return np.asarray(src, dtype=np.float32), np.asarray(dst, dtype=np.float32)
def H_mat(src, dst, N):
    # calculate the 'H' matrix - Homography matrix
    A = []
    for i in range(N):
        x, y = src[i][0], src[i][1]
        x_p, y_p = dst[i][0], dst[i][1]
        A.append([x, y, 1, 0, 0, 0, -x * x_p, -x_p * y, -x_p])
        A.append([0, 0, 0, x, y, 1, -y_p * x, -y_p * y, -y_p])
    A = np.asarray(A)
    u, s, vh = np.linalg.svd(A)
    l = vh[-1, :] / vh[-1, -1]
    H = l.reshape(3, 3)
    return H
def ransac_homography(src_pts, dst_pts):
    # calculate and find the best homography matrix using RANSAC
    I = 0
    Src = []
    Dst = []
    n=400 # number of times to calculate homography samples
    for i in range(n):
        src_pnt, dst_pnt = random_points(src_pts, dst_pts, 4)
        H = H_mat(src_pnt, dst_pnt, 4)
        inlines = 0
        src_lines = []
        dst_lines = []
        for p, q in zip(src_pts, dst_pts): # find the lines
```

```
                p1U = (np.append(p, 1)).reshape(3, 1)
                qe = H.dot(p1U)
                qe = (qe / qe[2])[:2].reshape(1, 2)[0]
                if cv.norm(q - qe) < 50: # make a comparison with a predefined value
                    inlines += 1
                    src_lines.append(p)
                    dst_lines.append(q)
            if inlines > I: # update the source and destination lines
                I = inlines
                Src = src_lines.copy()
                Src = np.asarray(Src, dtype=np.float32)
                Dst = dst_lines.copy()
                Dst = np.asarray(Dst, dtype=np.float32)
        H_f = H_mat(Src, Dst, I) # calculate the final H matrix
        return H_f
```

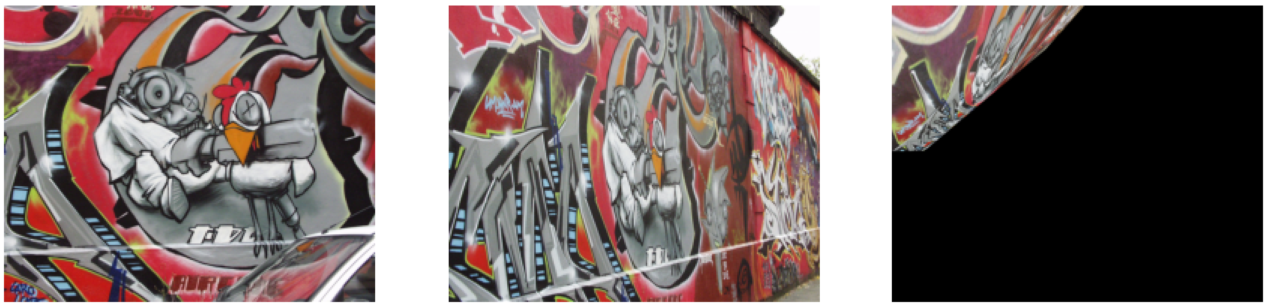
In [12]:
```
H14 = ransac_homography(src_pts14, dst_pts14) # calculate H matrix for image 1 and 4
H45 = ransac_homography(src_pts45, dst_pts45) # calculate H matrix for image 4 and 5
H15=np.matmul(H14,H45) # calculate H matrix for image 1 and 5
H15
```

Out[12]:
```
array([[ 2.13197318e+00, -1.39500513e-01, -4.74951988e+02],
       [-4.01092899e-01,  9.43529357e-01,  1.06309328e+02],
       [-9.97640001e-04,  1.21499726e-04,  1.26559657e+00]])
```

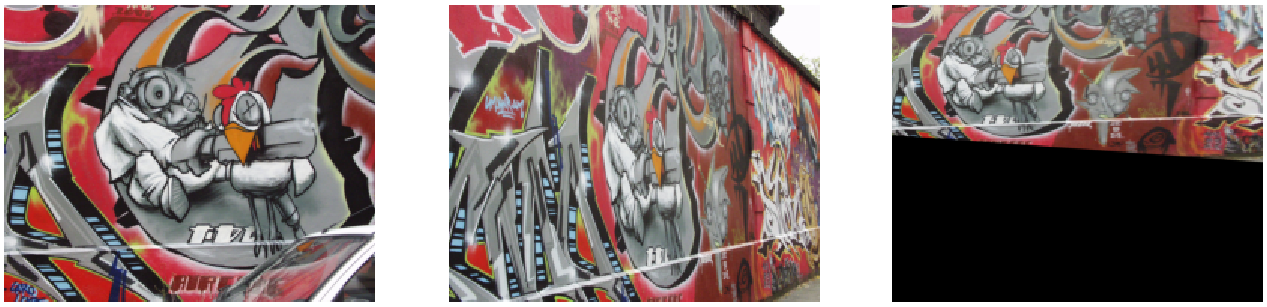
In [13]:
```
H15_direct = ransac_homography(src_pts15, dst_pts15) # calculate H matrix from image 1 to 5 directly
H_given=[]
with open(r'graf/H1to5p') as f: # read the given H matrix
    H_given=np.array([[float(h) for h in line.split()] for line in f])
```

Question 3 (c)

In [17]:
```
'''Figure 01'''  # stich image 1 to image 5 using the directly calculated H matrix
dst = cv.warpPerspective(img5,H15_direct, ((img1.shape[1] + img5.shape[1]), img5.shape[0]+ img5.shape[0]))
```



In [18]:
```
'''Figure 02'''  # stich image 1 to image 5 using the H matrix calculated using img1, img4 and img5
dst = cv.warpPerspective(img5,H15, ((img1.shape[1] + img5.shape[1]), img5.shape[0]+ img5.shape[0]))
```



In [19]:
```
'''Figure 03'''  # stich image 1 to image 5 using the given H matrix
dst=cv.warpPerspective(img5,np.linalg.inv(H_given),((img1.shape[1] + img5.shape[1]), img5.shape[0]))
```