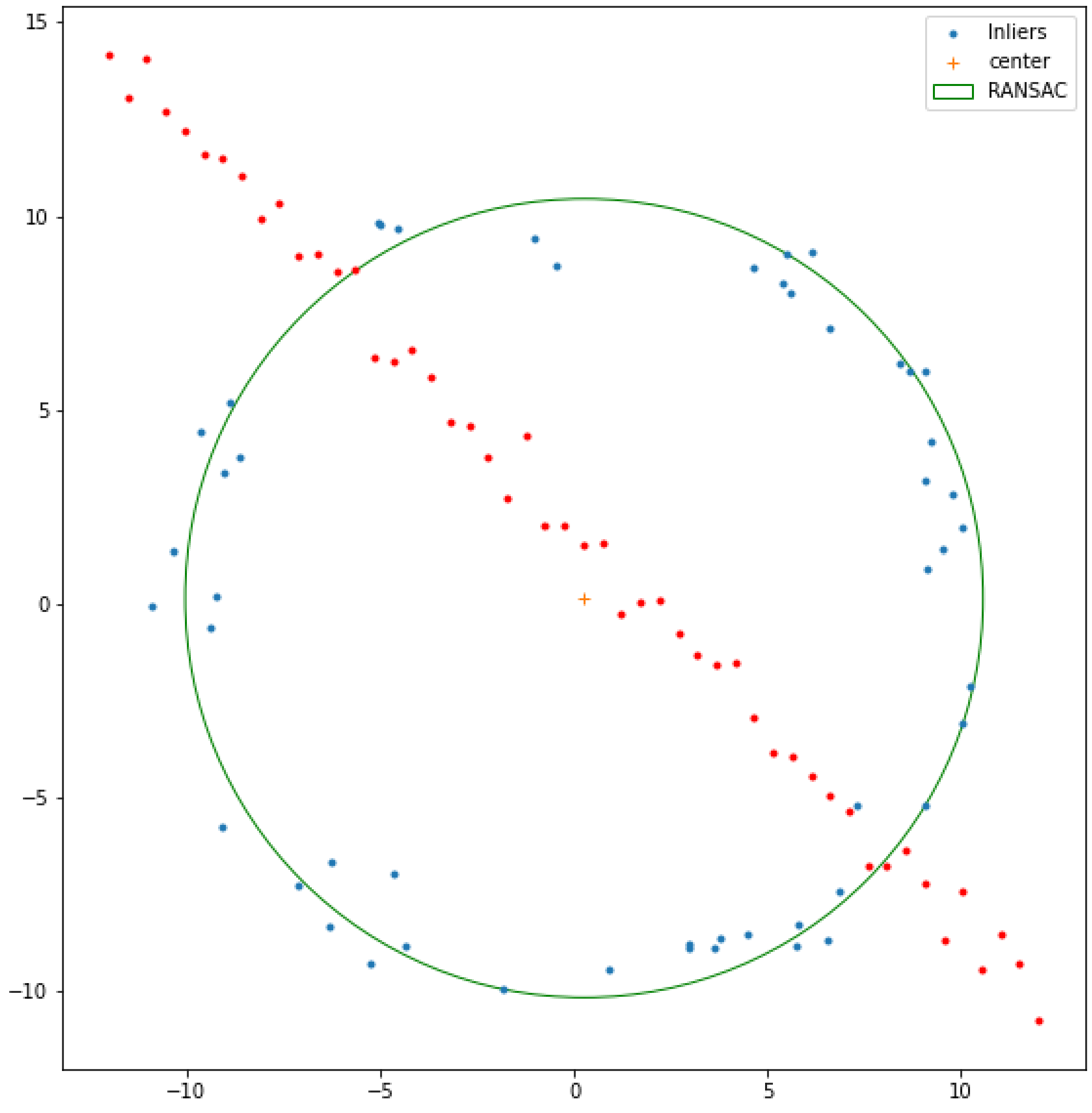
190531L Rukmal.M.A.D

Q(1)

In [80]:
```
import numpy as np
from scipy.optimize import minimize
from scipy import linalg
import matplotlib.pyplot as plt

#np.random.seed(0)
N = 100
half_n = N//2
r = 10
s = r/16
t = np.random.uniform (0,2*np.pi,half_n)
n = s*np.random.randn(half_n)
x,y = (r + n)*np.cos( t ),( r + n)*np.sin( t )
X_circ = np.hstack((x.reshape(half_n,1),y.reshape(half_n,1))) #list of cordinates of p
figure, axes = plt.subplots(figsize=(10,10))
#RANSAC Algorithm
N = 80; #number of repititive fittings
Inlier_ratio=0.9 # Inlier ratio
marginal_dis = 1; #distance from circle that points are considered as inliers
critic_inlier_count = 40
for i in range(0,N,1):
    #choosing 3 random points to uniquely define a circle
    ind_1,ind_2,ind_3 =np.random.randint(50),np.random.randint(50),np.random.randint(5
    coord_1,coord_2,coord_3 = X_circ[ind_1],X_circ[ind_2],X_circ[ind_3]
    x1,y1,x2,y2,x3,y3=coord_1[0],coord_1[1],coord_2[0],coord_2[1],coord_3[0],coord_3[1
    #calculating centre and radius of the circle
    c = (x1-x2)**2 + (y1-y2)**2
    a = (x2-x3)**2 + (y2-y3)**2
    b = (x3-x1)**2 + (y3-y1)**2
    s = 2*(a*b + b*c + c*a) - (a*a + b*b + c*c)
    cent_x = (a*(b+c-a)*x1 + b*(c+a-b)*x2 + c*(a+b-c)*x3) / s
    cent_y = (a*(b+c-a)*y1 + b*(c+a-b)*y2 + c*(a+b-c)*y3) / s
    ar = a**0.5
    br = b**0.5
    cr = c**0.5
    radi = ar*br*cr / ((ar+br+cr)*(-ar+br+cr)*(ar-br+cr)*(ar+br-cr))**0.5

    inlier_count=0;
    #distance to the points
    for j in range(0,50):
        dis=((X_circ[j][0]-cent_x)**2 +(X_circ[j][1]-cent_y)**2)**0.5
        Is_inlier=abs(dis-radi)<=marginal_dis
        if(Is_inlier):
            inlier_count+=1

    if(critic_inlier_count<=inlier_count):
        axes.plot(x,y,".",label="Inliers")
        axes.plot(cent_x,cent_y,'+',label='center')
        draw_circle = plt.Circle((cent_x,cent_y), radi,fill=False,color='g',label='RAN
        axes.set_aspect(1)
        axes.add_artist(draw_circle)
        plt.legend()
        break

#plotting rest of points
m,b = -1,2
s=r/16
x_line = np.linspace(-12,12,half_n)
y_line = m*x_line+b + s*np.random.randn(half_n)
X_line = np.hstack((x.reshape(half_n,1), y.reshape(half_n,1)))
X = np.vstack((X_circ,X_line))
plt.plot(x_line,y_line,".",color='r')
plt.show()
```



Q(2)

In [24]:

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

im1 = cv.imread(r'back_image.jpg',cv.IMREAD_COLOR)
assert im1 is not None
```

```
im2 = cv.imread(r'front_image1.jpg',cv.IMREAD_COLOR)
assert im2 is not None
img1_plt = cv.cvtColor(im1,cv.COLOR_BGR2RGB)
fig,ax = plt.subplots(figsize=(12,5))
ax.imshow(img1_plt)
plt.show()
```

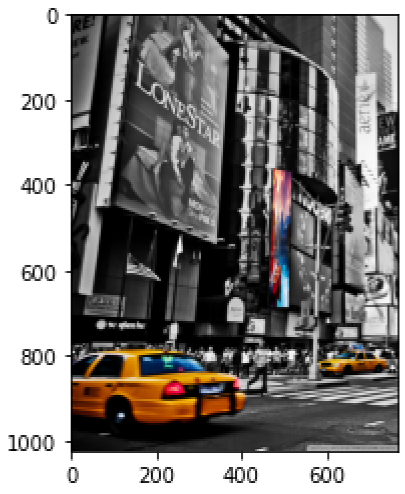
In this problem first we need to click four points of the planar surface. Therefore, define function for this purpose. When it is calling,the points are printed on the console. As well when clicking the corners need some procedure. First click the top left, next top right, next bottom left and last bottom right corner.

In [19]:
```
# fuction use to identify points using mouse clicks
import cv2 as cv

def click_event(event, x, y, flags, params):
    if event == cv.EVENT_LBUTTONDOWN:
        print(x, ' ', y)
        cv.imshow('image', img)

img = cv.imread(r'back_image.jpg', 1)
cv.imshow('image', img)
cv.setMouseCallback('image', click_event)
cv.waitKey(0)
cv.destroyAllWindows()
```

```
153   275
```

In [23]:
```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
%matplotlib inline

backimg = cv2.imread(r'back_image.jpg')
frontimg = cv2.imread(r'front_image1.jpg')
assert backimg is not None
assert frontimg is not None
```

```
#four corners of the back image to be replaced----------------------------------------
pts_backimg = np.array([[474,363], [516,371], [508,685], [462,685]])

#four corners of our front image
pts_frontimg = np.array([[0, 0], [frontimg.shape[1] - 1, 0], [frontimg.shape[1] - 1, f

#Calculate homography matrix
homographyMatrix, status = cv.findHomography(pts_frontimg, pts_backimg)

#warp frontimg to backimg
result1 = cv.warpPerspective(frontimg, homographyMatrix, (backimg.shape[1], backimg.sh

#removing the rest of area in back image
cv.fillConvexPoly(backimg, pts_backimg, 0, 16)

#Add warped image to the backimage
result = backimg + result1a

#display image
result = cv.cvtColor(result, cv.COLOR_BGR2RGB)
plt.imshow(result)
plt.show()
```

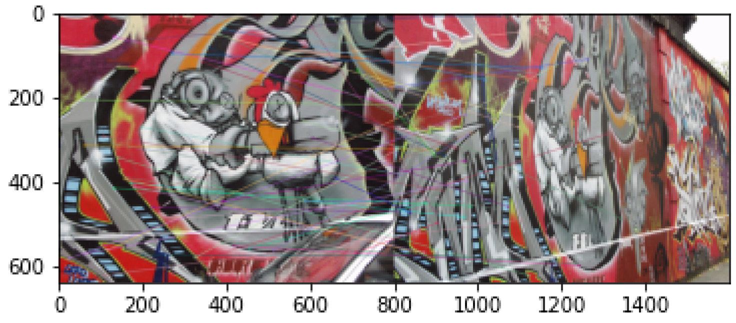


Q(3)

a)

In [92]:
```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

img1 = cv.imread(r'img1.ppm')
img2 = cv.imread(r'img5.ppm')
assert img1 is not None
assert img2 is not None

img1 = cv.cvtColor(img1, cv.COLOR_BGR2RGB)
img2 = cv.cvtColor(img2, cv.COLOR_BGR2RGB)
sift = cv.SIFT_create()
keypoints_1, descriptors_1 = sift.detectAndCompute(img1,None)
keypoints_2, descriptors_2 = sift.detectAndCompute(img2,None)
```

```
bf = cv.BFMatcher(cv.NORM_L1, crossCheck=True)
matches = bf.match(descriptors_1,descriptors_2)
matches = sorted(matches, key = lambda x:x.distance)
img3 = cv.drawMatches(img1, keypoints_1, img2, keypoints_2, matches[:50], img2, flags=
plt.imshow(img3)
plt.show()
```



b)

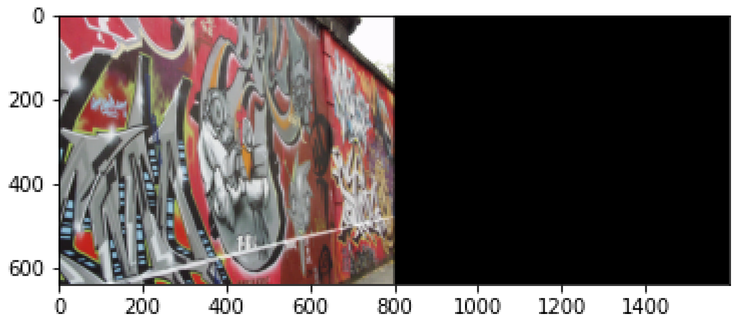
In [25]:
```
import cv2
import numpy as np
import getopt
import sys
import random

#
# Read in an image file, errors out if we can't find the file
#
def readImage(filename):
    img = cv2.imread(filename, 0)
    if img is None:
        print('Invalid image:' + filename)
        return None
    else:
        print('Image successfully read...')
        return img


# This draws matches and optionally a set of inliers in a different color
# Note: I lifted this drawing portion from stackoverflow and adjusted it to my needs b
# include the drawMatches function
def drawMatches(img1, kp1, img2, kp2, matches, inliers = None):
    # Create a new output image that concatenates the two images together
    rows1 = img1.shape[0]
    cols1 = img1.shape[1]
    rows2 = img2.shape[0]
    cols2 = img2.shape[1]

    out = np.zeros((max([rows1,rows2]),cols1+cols2,3), dtype='uint8')

    # Place the first image to the left
    out[:rows1,:cols1,:] = np.dstack([img1, img1, img1])

    # Place the next image to the right of it
    out[:rows2,cols1:cols1+cols2,:] = np.dstack([img2, img2, img2])

    # For each pair of points we have between both images
    # draw circles, then connect a line between them
    for mat in matches:

        # Get the matching keypoints for each of the images
        img1_idx = mat.queryIdx
        img2_idx = mat.trainIdx

        # x - columns, y - rows
        (x1,y1) = kp1[img1_idx].pt
        (x2,y2) = kp2[img2_idx].pt

        inlier = False

        if inliers is not None:
            for i in inliers:
                if i.item(0) == x1 and i.item(1) == y1 and i.item(2) == x2 and i.item(
                    inlier = True

        # Draw a small circle at both co-ordinates
        cv2.circle(out, (int(x1),int(y1)), 4, (255, 0, 0), 1)
        cv2.circle(out, (int(x2)+cols1,int(y2)), 4, (255, 0, 0), 1)

        # Draw a line in between the two points, draw inliers if we have them
        if inliers is not None and inlier:
            cv2.line(out, (int(x1),int(y1)), (int(x2)+cols1,int(y2)), (0, 255, 0), 1)
        elif inliers is not None:
            cv2.line(out, (int(x1),int(y1)), (int(x2)+cols1,int(y2)), (0, 0, 255), 1)

        if inliers is None:
            cv2.line(out, (int(x1),int(y1)), (int(x2)+cols1,int(y2)), (255, 0, 0), 1)

    return out

#
# Runs sift algorithm to find features
#
def findFeatures(img):
    print("Finding Features...")
    sift = cv2.SIFT()
    keypoints, descriptors = sift.detectAndCompute(img, None)

    img = cv2.drawKeypoints(img, keypoints)
    cv2.imwrite('sift_keypoints.png', img)

    return keypoints, descriptors

#
# Matches features given a list of keypoints, descriptors, and images
#
def matchFeatures(kp1, kp2, desc1, desc2, img1, img2):
    print("Matching Features...")
    matcher = cv2.BFMatcher(cv2.NORM_L2, True)
    matches = matcher.match(desc1, desc2)
    matchImg = drawMatches(img1,kp1,img2,kp2,matches)
    cv2.imwrite('Matches.png', matchImg)
    return matches


#
# Computers a homography from 4-correspondences
#
```

```
def calculateHomography(correspondences):
    #loop through correspondences and create assemble matrix
    aList = []
    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)
              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.ite
              p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2
        aList.append(a1)
        aList.append(a2)

    matrixA = np.matrix(aList)

    #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


#
#Calculate the geometric distance between estimated points and original points
#
def geometricDistance(correspondence, h):

    p1 = np.transpose(np.matrix([correspondence[0].item(0), correspondence[0].item(1),
    estimatep2 = np.dot(h, p1)
    estimatep2 = (1/estimatep2.item(2))*estimatep2

    p2 = np.transpose(np.matrix([correspondence[0].item(2), correspondence[0].item(3),
    error = p2 - estimatep2
    return np.linalg.norm(error)


#
#Runs through ransac algorithm, creating homographies from random correspondences
#
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 = calculateHomography(randomFour)
        inliers = []
        for i in range(len(corr)):
            d = geometricDistance(corr[i], h)
            if d < 5:
                inliers.append(corr[i])

        if len(inliers) > len(maxInliers):
            maxInliers = inliers
            finalH = h
        print "Corr size: ", len(corr), " NumInliers: ", len(inliers), "Max inliers: '

        if len(maxInliers) > (len(corr)*thresh):
            break
    return finalH, maxInliers


#
# Main parses argument list and runs the functions
#
def main():
    args, img_name = getopt.getopt(sys.argv[1:],'', ['threshold='])
    args = dict(args)

    estimation_thresh = args.get('--threshold')
    print "Estimation Threshold: ", estimation_thresh
    if estimation_thresh is None:
        estimation_thresh = 0.60

    img1name = str(img_name[0])
    img2name = str(img_name[1])
    print("Image 1 Name: " + img1name)
    print("Image 2 Name: " + img2name)

    #query image
    img1 = readImage(img_name[0])
    #train image
    img2 = readImage(img_name[1])

    #find features and keypoints
    correspondenceList = []
    if img1 is not None and img2 is not None:
        kp1, desc1 = findFeatures(img1)
        kp2, desc2 = findFeatures(img2)
        keypoints = [kp1,kp2]
        matches = matchFeatures(kp1, kp2, desc1, desc2, img1, img2)
        for match in matches:
            (x1, y1) = keypoints[0][match.queryIdx].pt
            (x2, y2) = keypoints[1][match.trainIdx].pt
            correspondenceList.append([x1, y1, x2, y2])

        corrs = np.matrix(correspondenceList)

        #run ransac algorithm
        finalH, inliers = ransac(corrs, estimation_thresh)
        print("Final homography: ", finalH)

        matchImg = drawMatches(img1,kp1,img2,kp2,matches,inliers)
        cv2.imwrite('InlierMatches.png', matchImg)

        f = open('homography.txt', 'w')
        f.write("Final homography: \n" + str(finalH)+"\n")
        f.write("Final inliers count: " + str(len(inliers)))
        f.close()


if __name__ == "__main__":
    main()
print("Given Homography")
H_given = np.array([[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]])
H_given
```

```
  Input In [25]
    print "Corr size: ", len(corr), " NumInliers: ", len(inliers), "Max inliers: ", l
en(maxInliers)
    ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?
```

c)

```
In [104…
H = np.array([[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]])

dst = cv.warpPerspective(img1,H,((img1.shape[1] + img2.shape[1]), img2.shape[0])) #wra
dst[0:img2.shape[0], 0:img2.shape[1]] = img2 #stitched image
cv.imwrite('output.jpg',dst)
dst = cv.cvtColor(dst, cv.COLOR_BGR2RGB)
plt.imshow(dst)
plt.show()
```



```
In [ ]:
```