CS6476 - PS4: Harris, SIFT, RANSAC

Martin Saint-Jalmes (msaintjalmes3)

In [1]:

```
# Importing Numpy, OpenCV and Matplotlib
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#Importing python utilities
import math
from timeit import default_timer as timer
from itertools import izip

#imports for interactive jupyter
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

1. Harris corners

1.1 X and Y gradients

In [2]:

```
def gradient(image, alongX = True, scale = 1):
    image = image.astype(np.int32)
    return image[scale:] - image[:-1*scale] if alongX else image[:,scale:] - image[:,:-
1*scale]
```

In [3]:

```
def diffTo255(gradient_image):
    matrix = gradient_image + 127
    return ((matrix.astype(np.float32) / np.max(matrix))*255).astype(np.uint8)
```

In [4]:

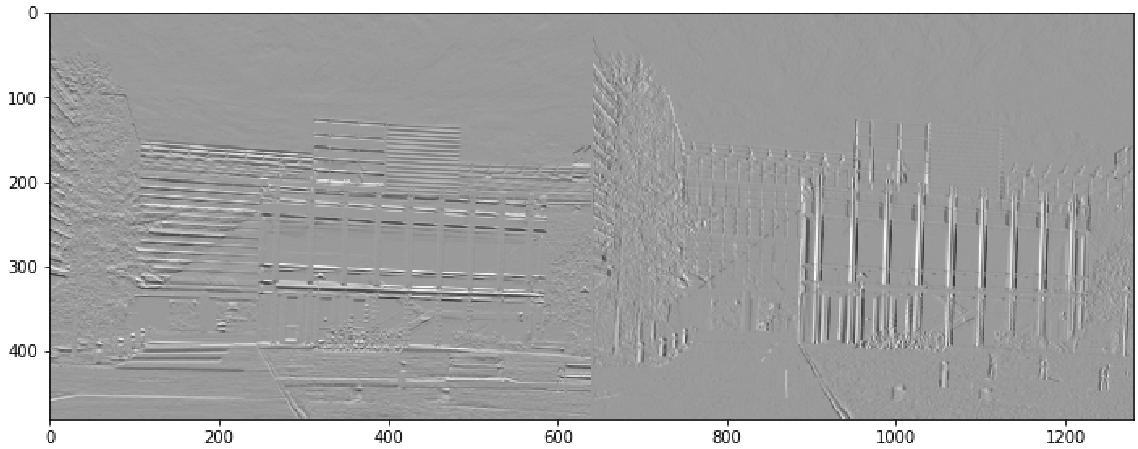
```
transA = cv2.imread('./transA.jpg', cv2.IMREAD_GRAYSCALE)
simA = cv2.imread('./simA.jpg', cv2.IMREAD_GRAYSCALE)
```

In [5]:

```
#Interact bypass for report generation
#@interact(scale = (1,30), gauss_sigma = (1,10), filter_size = (3,31,2))
def transA_grad_images(scale = 1, gauss_sigma = 2, filter_size = 3):
    transA_blur = cv2.GaussianBlur(transA, (filter_size, filter_size), gauss_sigma, cv2
.BORDER_REFLECT)
    transA_gradX = gradient(transA_blur, scale = scale)
    transA_gradY = gradient(transA_blur, alongX = False, scale = scale)

    transA_gradX_pad = np.zeros_like(transA) # cannot hstack because of dimensions, fil
l with 0s
    margin = transA.shape[0] - transA_gradX.shape[0]
    transA_gradX_pad[margin:] = transA_gradX
    transA_gradY_pad = np.zeros_like(transA)
    margin = transA.shape[1] - transA_gradY.shape[1]
    transA_gradY_pad[:, margin:] = transA_gradY

    transA_gradpair = np.hstack((transA_gradX_pad, transA_gradY_pad))

    plt.figure(figsize=(12, 9))
    plt.imshow(diffTo255(transA_gradpair), cmap='gray', vmin = 0, vmax = 255)
```

In [6]:

```
#Manual call for report generation
transA_grad_images()
```

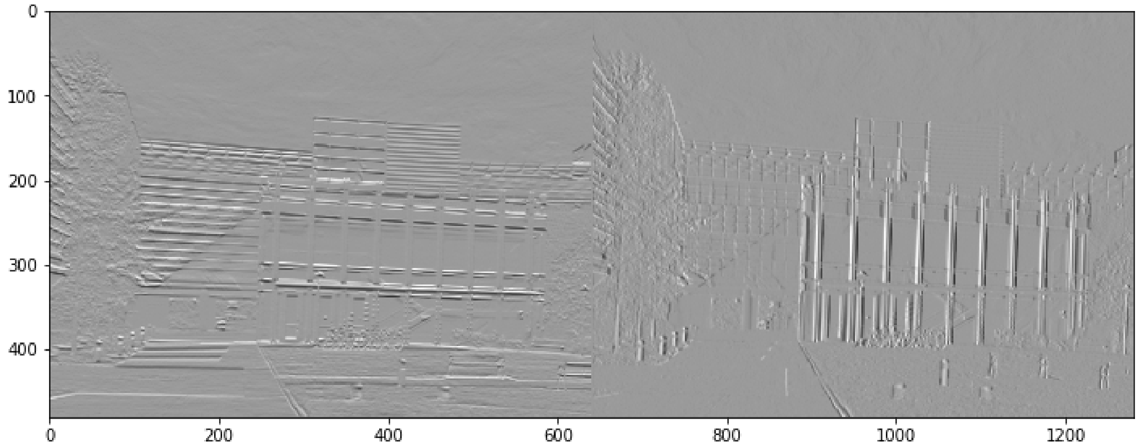


In [7]:

```
#Interact bypass for report generation
#@interact(scale = (1,30), gauss_sigma = (1,10), filter_size = (3,31,2))
def simA_grad_images(scale = 1, gauss_sigma = 2, filter_size = 3):
    simA_blur = cv2.GaussianBlur(transA, (filter_size, filter_size), gauss_sigma, cv2.B
ORDER_REFLECT)

    simA_gradX = gradient(simA_blur, scale = scale)
    simA_gradY = gradient(simA_blur, alongX = False, scale = scale)

    simA_gradX_pad = np.zeros_like(simA) # cannot hstack because of dimensions, fill wi
th 0s
    margin = simA.shape[0] - simA_gradX.shape[0]
    simA_gradX_pad[margin:] = simA_gradX
    simA_gradY_pad = np.zeros_like(simA)
    margin = simA.shape[1] - simA_gradY.shape[1]
    simA_gradY_pad[:, margin:] = simA_gradY

    simA_gradpair = np.hstack((simA_gradX_pad, simA_gradY_pad))

    plt.figure(figsize=(12, 9))
    plt.imshow(diffTo255(simA_gradpair), cmap='gray', vmin = 0, vmax = 255)
```

In [8]:

```
#Manual call for report generation
simA_grad_images()
```



1.2. Harris values

In [9]:

```
def Harris(image, window_size, alpha = 0.04, cust_weights = None, gradient_scale = 1, g
auss_sigma = 4, filter_size = 11):
    if gauss_sigma is None and filter_size is None:
        image_blur = image
    else:
        image_blur = cv2.GaussianBlur(image, (filter_size, filter_size), gauss_sigma, c
v2.BORDER_REFLECT)
    Gradx = gradient(image_blur, scale = gradient_scale)
    Grady = gradient(image_blur, alongX = False, scale = gradient_scale)

    #code to fill with zeros
    Gradx_pad = np.zeros_like(image, dtype=np.float32)
    margin = image.shape[0] - Gradx.shape[0]
    Gradx_pad[margin:] = Gradx.copy()
    Grady_pad = np.zeros_like(image, dtype=np.float32)
    margin = image.shape[1] - Grady.shape[1]
    Grady_pad[:, margin:] = Grady.copy()
    #Gradx = diffTo255(Gradx_pad) #DO NOT USE
    #Grady = diffTo255(Grady_pad) #DO NOT USE
    Gradx = Gradx_pad
    Grady = Grady_pad

    #to avoid repeating operations within loop
    Gradx_sq = Gradx ** 2
    Grady_sq = Grady ** 2
    Gradx_times_Grady = np.multiply(Gradx, Grady)

    margin = (window_size - 1) /2
    R = np.zeros((image.shape[0] - 2 * margin, image.shape[1] - 2 * margin), dtype = np
.float32)

    if cust_weights == None:
        gaussian = cv2.getGaussianKernel(window_size, margin)
        weights = np.outer(gaussian, gaussian)
        weights /= weights.sum()
    else:
        weights = cust_weights

    for row in np.arange(margin, image.shape[0] - margin):
        for col in np.arange(margin, image.shape[1] - margin):
            Ix = Gradx[row-margin:row+margin+1, col-margin:col+margin+1]
            Iy = Grady[row-margin:row+margin+1, col-margin:col+margin+1]
            Ix_sq = Gradx_sq[row-margin:row+margin+1, col-margin:col+margin+1]
            Iy_sq = Grady_sq[row-margin:row+margin+1, col-margin:col+margin+1]
            Ix_times_Iy = Gradx_times_Grady[row-margin:row+margin+1, col-margin:col+mar
gin+1]
            M = np.array((np.sum(np.multiply(weights, Ix_sq)), np.sum(np.multiply(weigh
ts, Ix_times_Iy)), \
                          np.sum(np.multiply(weights, Ix_times_Iy)), np.sum(np.multiply
(weights, Iy_sq)))).reshape(2,2)
            R[row-margin, col-margin] = np.linalg.det(M) - alpha * np.trace(M)**2

    return R
```

In [10]:

```
transB = cv2.imread('./transB.jpg', cv2.IMREAD_GRAYSCALE)
simB = cv2.imread('./simB.jpg', cv2.IMREAD_GRAYSCALE)
```

In [11]:

```
transA_harris = Harris(transA, 7, gauss_sigma = 2, filter_size = 3)
transB_harris = Harris(transB, 7, gauss_sigma = 2, filter_size = 3)
simA_harris = Harris(simA, 7, gauss_sigma = 2, filter_size = 3)
simB_harris = Harris(simB, 7, gauss_sigma = 2, filter_size = 3)
```

In [12]:

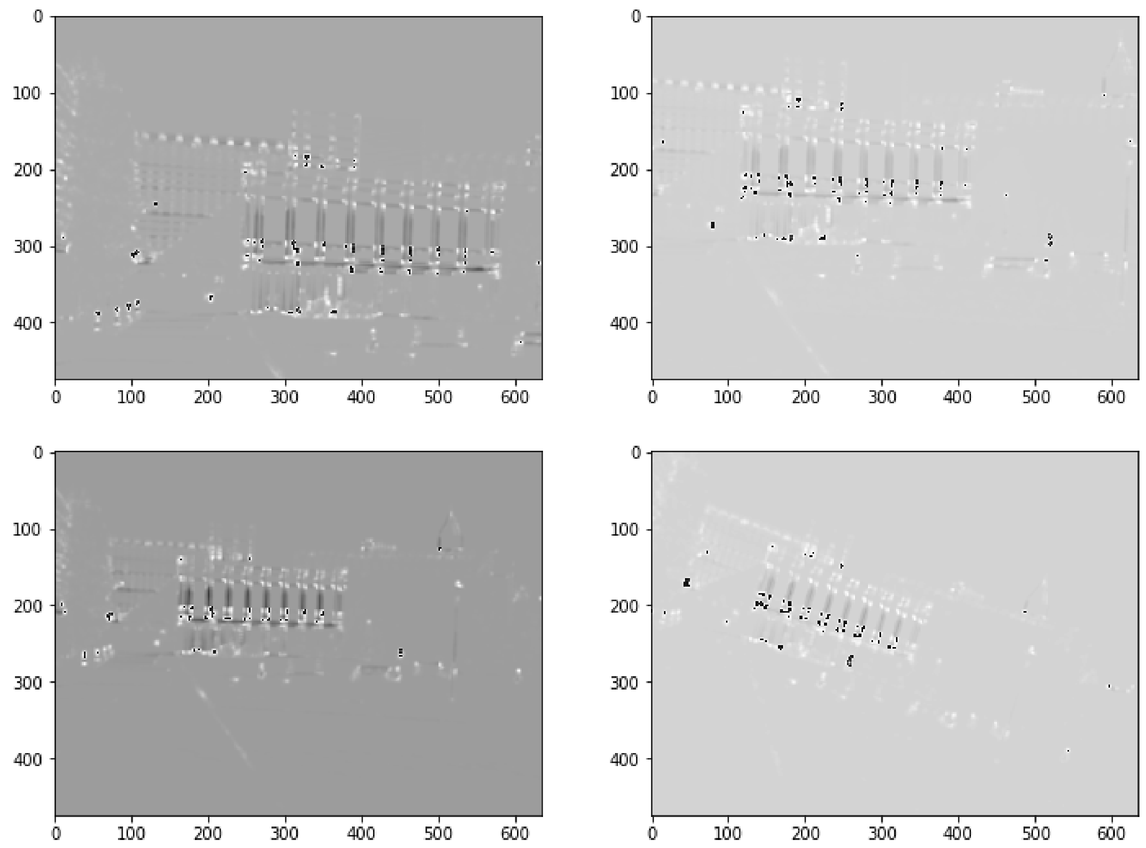
```
def normalize_255(matrix):
    return (((np.min(matrix) + matrix.astype(np.float32)) / np.max(matrix))*255).astype
(np.uint8)
```

In [13]:

```
plt.figure(figsize=(12, 9))
plt.subplot(2,2,1)
plt.imshow(normalize_255(transA_harris), cmap='gray', vmin = 0, vmax = 255)
plt.subplot(2,2,2)
plt.imshow(normalize_255(transB_harris), cmap='gray', vmin = 0, vmax = 255)
plt.subplot(2,2,3)
plt.imshow(normalize_255(simA_harris), cmap='gray', vmin = 0, vmax = 255)
plt.subplot(2,2,4)
plt.imshow(normalize_255(simB_harris), cmap='gray', vmin = 0, vmax = 255)
```

Out[13]:

```
<matplotlib.image.AxesImage at 0x1ee396d8>
```



1.3. Non-maxima suppression and tresholding

In [14]:

```
def max_filter(matrix, size = 10):
    new_matrix = matrix.copy()
    (r_boundary, c_boundary) = matrix.shape
    for r in range(size, r_boundary - size + 1):
        for c in range(size, c_boundary - size + 1):
            argmax = np.unravel_index(np.argmax(new_matrix[r-size:r+size, c-size:c+size
]), (2*size, 2*size))
            max = new_matrix[r-size + argmax[0], c-size + argmax[1]]
            new_matrix[r-size:r+size, c-size:c+size] = 0
            new_matrix[r-size + argmax[0], c-size + argmax[1]] = max
    return new_matrix
```

In [15]:

```
def threshold_filter(matrix, n_points = 300):
    flattened = matrix.ravel()
    best = np.argsort(flattened)[-n_points:]
    mask = np.zeros(flattened.shape, dtype=bool)
    mask[best] = True
    flattened[~mask] = 0
    matrix = flattened.reshape(matrix.shape)
    return matrix
```

In [16]:

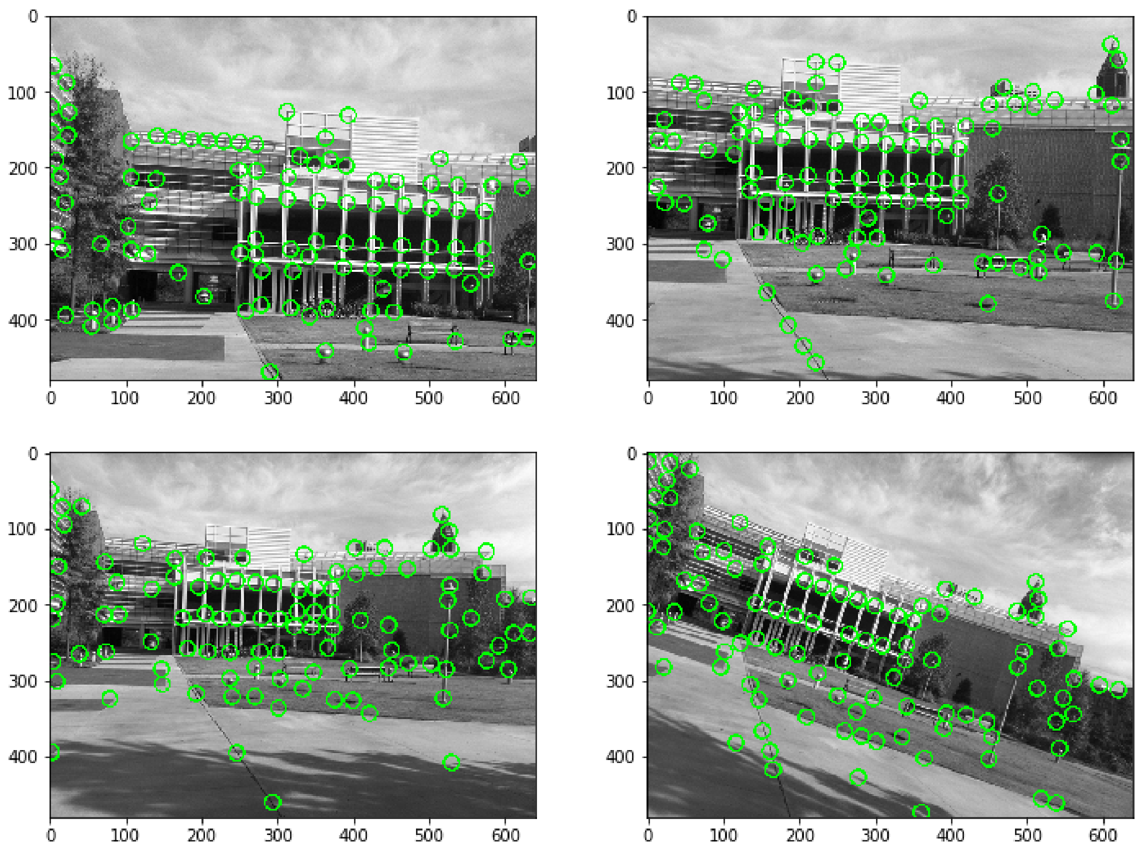
```
transA_corners = threshold_filter(max_filter(transA_harris), 100)
transA_corners_nz = np.array(np.nonzero(transA_corners)).T
transB_corners = threshold_filter(max_filter(transB_harris), 100)
transB_corners_nz = np.array(np.nonzero(transB_corners)).T
simA_corners = threshold_filter(max_filter(simA_harris), 100)
simA_corners_nz = np.array(np.nonzero(simA_corners)).T
simB_corners = threshold_filter(max_filter(simB_harris), 100)
simB_corners_nz = np.array(np.nonzero(simB_corners)).T
```

In [17]:

```
plt.figure(figsize=(12, 9))
plt.subplot(2,2,1)
transA_col = cv2.cvtColor(transA, cv2.COLOR_GRAY2RGB)
for nonzero in transA_corners_nz:
    cv2.circle(transA_col, tuple(nonzero[::-1]), radius = 10, color = (0,255,0), thickn
ess=2, lineType=8, shift=0)
plt.imshow(normalize_255(transA_col), vmin = 0, vmax = 255)
plt.subplot(2,2,2)
transB_col = cv2.cvtColor(transB, cv2.COLOR_GRAY2RGB)
for nonzero in transB_corners_nz:
    cv2.circle(transB_col, tuple(nonzero[::-1]), radius = 10, color = (0,255,0), thickn
ess=2, lineType=8, shift=0)
plt.imshow(normalize_255(transB_col), vmin = 0, vmax = 255)
plt.subplot(2,2,3)
simA_col = cv2.cvtColor(simA, cv2.COLOR_GRAY2RGB)
for nonzero in simA_corners_nz:
    cv2.circle(simA_col, tuple(nonzero[::-1]), radius = 10, color = (0,255,0), thicknes
s=2, lineType=8, shift=0)
plt.imshow(normalize_255(simA_col), vmin = 0, vmax = 255)
plt.subplot(2,2,4)
simB_col = cv2.cvtColor(simB, cv2.COLOR_GRAY2RGB)
for nonzero in simB_corners_nz:
    cv2.circle(simB_col, tuple(nonzero[::-1]), radius = 10, color = (0,255,0), thicknes
s=2, lineType=8, shift=0)
plt.imshow(normalize_255(simB_col), vmin = 0, vmax = 255)
```

Out[17]:

```
<matplotlib.image.AxesImage at 0x1f4505c0>
```



The corner detector works reasonably well. In particular, it manages to not detect anything in the sky (that would make poor corners for matching).
It is not perfect, of course, as there are some corners detected with the shadows of the tree with the sim pair (although they are sufficiently distinct for us to use them in this case because there is no time difference).
The line on the ground is also a place where we surprinsingly detect corners. It might be the case that because of perspective and irregularities, the neighborhood might be sufficiently unique compared to other patches along the same line.
Finally, there are some corners that are not detected on both images within each pair (mostly the repetitive patterns on the top fence of the building. The reason is mostly because of the thresholding that was performed to 100 corners. A better, more significant region was select instead.

2. SIFT features

2.1 Keypoints

In [18]:

```
def angles(image, degrees = True):
    Grady = gradient(image, alongX=False)
    Gradx = gradient(image)

    Gradx_pad = np.zeros_like(image, dtype=np.float32)
    margin = image.shape[0] - Gradx.shape[0]
    Gradx_pad[margin:] = Gradx.copy()
    Grady_pad = np.zeros_like(image, dtype=np.float32)
    margin = image.shape[1] - Grady.shape[1]
    Grady_pad[:, margin:] = Grady.copy()
    Gradx = Gradx_pad
    Grady = Grady_pad
    if degrees:
        return (np.degrees(np.arctan2(Grady, Gradx)) + 180)%360
    return np.arctan2(Grady, Gradx)

transA_angles = angles(transA)
transB_angles = angles(transB)
simA_angles = angles(simA)
simB_angles = angles(simB)
```

In [19]:

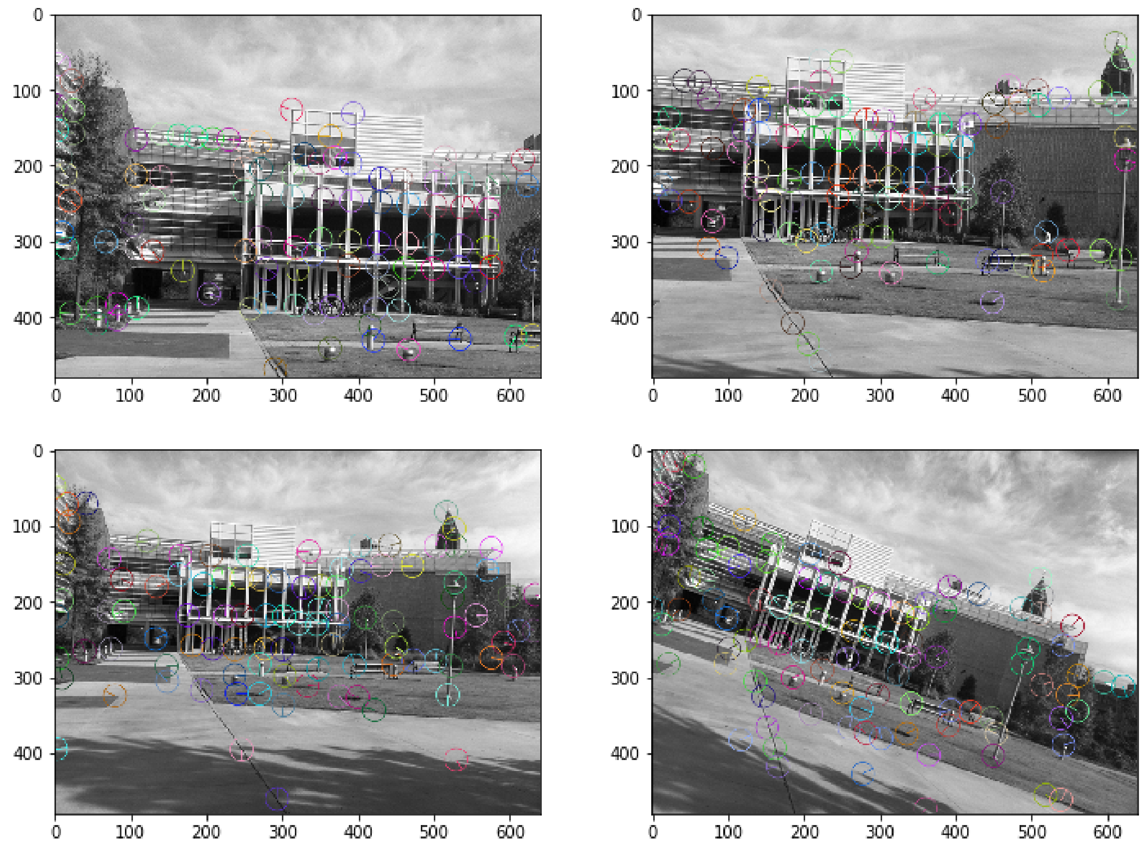
```
def Keypoints(corners, angles, scale=30): #scale 1 is too small
    return [cv2.KeyPoint(corner[1], corner[0], scale, angles[corner[0], corner[1]], _oc
tave=0) for corner in corners]

transA_kp = Keypoints(transA_corners_nz, transA_angles)
transB_kp = Keypoints(transB_corners_nz, transB_angles)
simA_kp = Keypoints(simA_corners_nz, simA_angles)
simB_kp = Keypoints(simB_corners_nz, simB_angles)
```

In [20]:

```
plt.figure(figsize=(12, 9))
plt.subplot(2,2,1)
transA_kp_img = cv2.drawKeypoints(transA,transA_kp,None,flags=cv2.DRAW_MATCHES_FLAGS_DR
AW_RICH_KEYPOINTS)
plt.imshow(normalize_255(transA_kp_img), vmin = 0, vmax = 255)
plt.subplot(2,2,2)
transB_kp_img = cv2.drawKeypoints(transB,transB_kp,None,flags=cv2.DRAW_MATCHES_FLAGS_DR
AW_RICH_KEYPOINTS)
plt.imshow(normalize_255(transB_kp_img), vmin = 0, vmax = 255)
plt.subplot(2,2,3)
simA_kp_img = cv2.drawKeypoints(simA,simA_kp,None,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RIC
H_KEYPOINTS)
plt.imshow(normalize_255(simA_kp_img), vmin = 0, vmax = 255)
plt.subplot(2,2,4)
simB_kp_img = cv2.drawKeypoints(simB,simB_kp,None,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RIC
H_KEYPOINTS)
plt.imshow(normalize_255(simB_kp_img), vmin = 0, vmax = 255)
```

Out[20]:

```
<matplotlib.image.AxesImage at 0x2070def0>
```



2.2. Matching

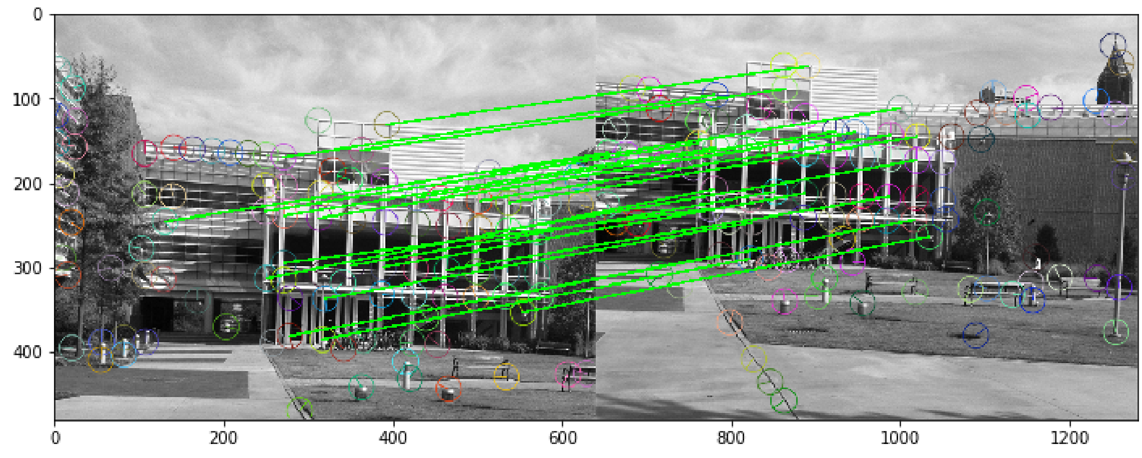
This part uses the updated (Spring 2015) supplemental material from Pr. Bobick for PS4, with code to use for OpenCV-Python.
https://docs.google.com/document/d/1-2pLrbVFySuCzJjoWxJ7kZQ1tZLNgdH49Bhmlkc7XgM/view
(https://docs.google.com/document/d/1-2pLrbVFySuCzJjoWxJ7kZQ1tZLNgdH49Bhmlkc7XgM/view)

In [21]:

```
def match(image1, image2, keypoints1, keypoints2, ratio = None):
    image1 = cv2.drawKeypoints(image1,keypoints1,None,flags=cv2.DRAW_MATCHES_FLAGS_DRAW
_RICH_KEYPOINTS)
    image2 = cv2.drawKeypoints(image2,keypoints2,None,flags=cv2.DRAW_MATCHES_FLAGS_DRAW
_RICH_KEYPOINTS)
    imagepair = np.hstack((image1, image2))

    sift = cv2.xfeatures2d.SIFT_create()
    _, descriptors1 = sift.compute(image1, keypoints1) #anon. we don't need keypoints a
gain
    _, descriptors2 = sift.compute(image2, keypoints2)

    keypoints_coord = []

    if ratio is None: #get the best possible match
        matcher = cv2.BFMatcher(normType=cv2.NORM_L2, crossCheck=True)
        matches = matcher.match(descriptors2, descriptors1)
    else: #only select matches where 2-NN is very dissimilar to 1-NN (1-NN/2-NN distanc
e ratio small)
        matcher = cv2.BFMatcher(normType=cv2.NORM_L2)
        matches = np.array(matcher.knnMatch(descriptors2, descriptors1, 2))
        matches = matches[ [match[0].distance/match[1].distance < ratio for match in ma
tches], 0]

    for match in matches:
        img1_desc_id = match.trainIdx #train descriptor index
        img2_desc_id = match.queryIdx #query descriptor index
        kp1 = tuple(np.array(keypoints1[img1_desc_id].pt, dtype = np.int16))
        kp2 = tuple(np.array(keypoints2[img2_desc_id].pt, dtype = np.int16) + [image1.s
hape[1], 0])
        keypoints_coord.append([kp1, kp2])
        cv2.line(imagepair, kp1, kp2, color = (0,255,0), thickness=2, lineType=8, shift
=0)

    plt.figure(figsize=(12, 9))
    plt.imshow(imagepair, cmap='gray', vmin = 0, vmax = 255)

    return np.array(keypoints_coord)
```

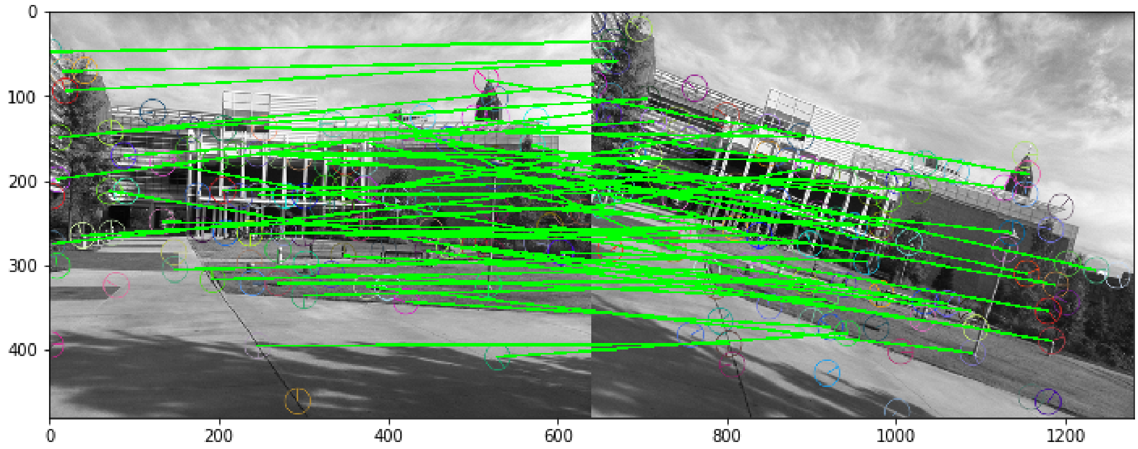
In [22]:

```
trans_matches = match(transA, transB, transA_kp, transB_kp, ratio = 0.8)
```



In [23]:

```
sim_matches = match(simA, simB, simA_kp, simB_kp, ratio = None)
```



Note: Due to a limitation in Python's OpenCV, it wasn't possible to use knn and perform a ratio test (selecting matches where 2-NN is very dissimilar to 1-NN (1-NN/2-NN distance ratio small)) using cross-check matching. In the similarity case, it is important as a lot of matches involve the same descriptor and yields poor results.
So, a conserative approach was used at this step, keeping a lot of the matches (with a lot of lines appearing on the image above) that we will filter with RANSAC to only keep the best ones.

3. RANSAC

In [24]:

```
def RANSAC(image1, image2, matches, n_pairs, delta, max_iter = 100):
    imagepair = np.hstack((image1, image2))

    best_inliers = 0
    best_list = []
    best_tr = []

    for i in np.arange(max_iter):
        inliers = 0
        inlier_list = []
        ids = np.random.choice(matches.shape[0], size = n_pairs)
        sample = matches[ids,:,:]
        if n_pairs == 1: #translation case
            [t_col, t_row] = [sample[0,1,0] - sample[0,0,0], sample[0,1,1] - sample[0,0,1]]
            transformation = np.array([t_col - image1.shape[1], t_row]).reshape(2,1)

            for i, match in enumerate(matches):
                if np.sqrt((match[1,0] - match[0,0] - t_col)**2 + (match[1,1] - match[0,1] - t_row)**2) < delta:
                    inlier_list.append(i)
                    inliers += 1

        elif n_pairs == 2: #similarity case
            A = np.array(([sample[0,0,0], -sample[0,0,1], 1, 0], [sample[0,0,0], sample[0,0,1], 0, 1],
                          [sample[1,0,0], -sample[1,0,1], 1, 0], [sample[1,0,0], sample[1,0,1], 0, 1])).reshape(4,4)
            b = np.array(sample[0,:,:]).reshape(4,1)
            b -= np.array([0,0,image1.shape[1],0]).reshape(4,1) #accounting for pair (-img1 cols)
            H = np.linalg.lstsq(A, b, rcond = None)[0]
            transformation = np.array([H[0], -H[1], H[2], H[1], H[0], H[3]]).reshape(2,3)

            for i, match in enumerate(matches):
                pred = np.dot(transformation, np.append(match[0,:],  [1]))
                if np.sqrt((match[1,0] - pred[0] - image1.shape[1])**2 + (match[1,1] - pred[1])**2) < delta:
                    inlier_list.append(i)
                    inliers += 1

        else:
            raise Exception

        if inliers > best_inliers:
            best_inliers = inliers
            best_list = inlier_list
            best_tr = transformation

    return matches[best_list], best_inliers * 1.0 / matches.shape[0], transformation
```

3.1 Translation with transA and transB

In [25]:

```
trans_consensus, trans_consensus_rate, tr_vect = RANSAC(transA, transB, trans_matches,
1, 20, max_iter = 100)
```

In [26]:

```
tr_vect #column and row translation
```

Out[26]:

```
array([[-131],
       [ -95]], dtype=int64)
```

In [27]:

```
trans_consensus_rate
```

Out[27]:

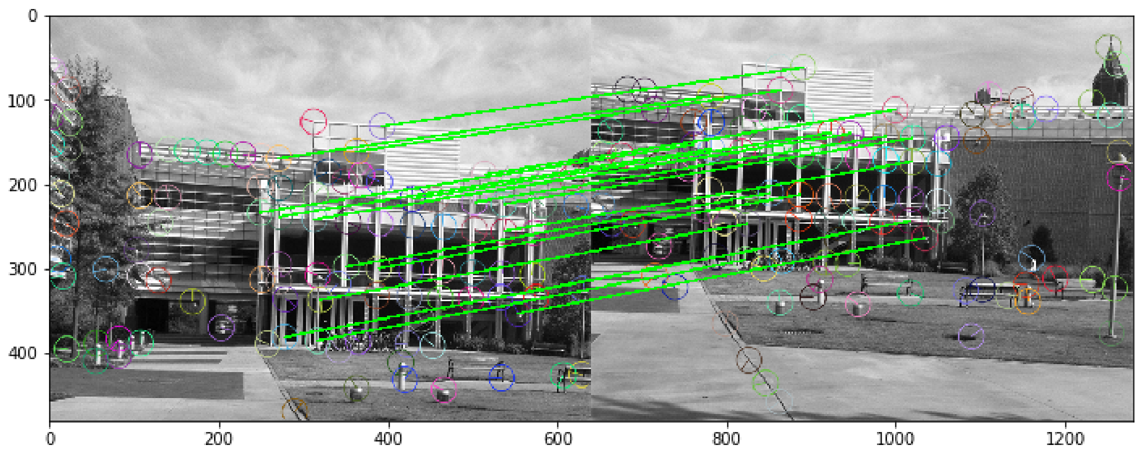
```
0.8260869565217391
```

In [28]:

```
trans_pair = np.hstack((transA_kp_img, transB_kp_img))
for match in trans_consensus:
    cv2.line(trans_pair, tuple(match[0]), tuple(match[1]), color = (0,255,0), thickness
=2, lineType=8, shift=0)
plt.figure(figsize=(12, 9))
plt.imshow(trans_pair, cmap='gray', vmin = 0, vmax = 255)
```

Out[28]:

```
<matplotlib.image.AxesImage at 0x236ec518>
```



The results we get obviously depend on the tolerance we set within the RANSAC algorithm (to consider matches to be inliers or outliers). For a tolerance of 20 ("pixels", although the L2-distance is used), the translation vector that was found is $(-131, -95)$ from the first to the second image (column, row). 82.6% of matches made up the biggest consensus set within 100 iterations.

3.2 Similarity with simA and simB

In [29]:

```
sim_consensus, sim_consensus_rate, sim_matrix = RANSAC(simA, simB, sim_matches, 2, 20,
max_iter = 2000)
```

In [30]:

```
sim_matrix
```

Out[30]:

```
array([[-2.17081851e-01,  1.23333333e+01, -1.51693120e+03],
       [-1.23333333e+01, -2.17081851e-01,  2.44773547e+03]])
```

In [31]:

```
sim_consensus_rate
```

Out[31]:

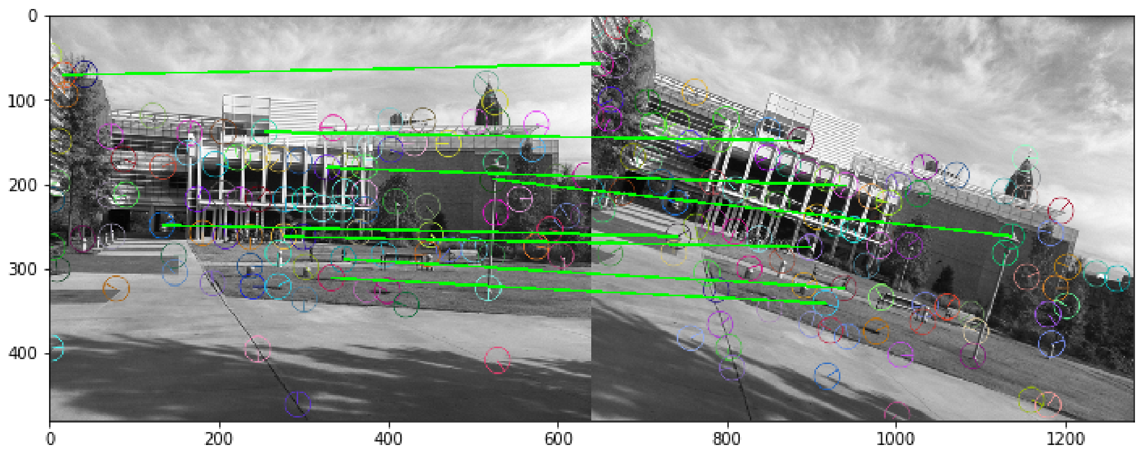
```
0.14814814814814814
```

In [32]:

```
sim_pair = np.hstack((simA_kp_img, simB_kp_img))
for match in sim_consensus:
    cv2.line(sim_pair, tuple(match[0]), tuple(match[1]), color = (0,255,0), thickness=2
, lineType=8, shift=0)
plt.figure(figsize=(12, 9))
plt.imshow(sim_pair, cmap='gray', vmin = 0, vmax = 255)
```

Out[32]:

```
<matplotlib.image.AxesImage at 0x209a6a90>
```



As a result from the imprecise matching (no ratio test), some of the matches aren't exact (e.g. on the lamp post or on the patch of grass on the left).

The results we get depend on the tolerance we set within the RANSAC algorithm (to consider matches to be inliers or outliers). For a tolerance of 20 ("pixels", although the L2-distance is used), the transformation matrix that was found is $\begin{bmatrix} -0.217 & 12.3 & -1517 \\ -12.3 & -0.217 & 2448 \end{bmatrix}$ from the first to the second image. 14.8% of matches made up the biggest consensus set within 2000 iterations.