Practice 3

Xabier Oyanguren Asua 1456628

We start importing the necessary packages and the three images we will start playing with.

In [1]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def plot(im, cmap='gray', title=None, size=(10,5)):
    %matplotlib inline
    plt.figure(figsize=size)
    plt.imshow(im, cmap=cmap)
    if title is not None:
        plt.title(title)
    plt.show()
```

In [2]:

```
L=cv2.imread("L.jpg")[:,:,::-1]
C=cv2.imread("C.jpg")[:,:,::-1]
R=cv2.imread("R.jpg")[:,:,::-1]
plot(np.hstack((L,C,R)), size=(30,5))
print( "      L", 45*" ", "C", 45*" ", "R")
```


```
      L                                              C
R
```

Note we name the left image as L, the center image as C and the right one R.

General Routines we will be using

Let us now define the functions to get the homography sending pixel coordinates of one image plane to the pixel positions of another camera plane. With it, we will be able to get any image as seen from the camera extrinsics of another image.

To get this homography we will employ the Direct Linear Transform method. For that we need to manually choose some corresponding points between the first and the rest of images, so we also define functions for that.

In [2]:

```
def choose_points(image, num_points=-1):
    %matplotlib tk
    plt.imshow(image)
    plt.title("Choose the points of correspondence")
    points = plt.ginput(n=num_points, timeout=-1, show_clicks=True)
    plt.show()
    return points

def build_homography_from_m_to_mp(m_points, mp_points):
    # mp alpha Hm
    # turned into Bh=0
    B = np.zeros((len(m_points)*2, 9), dtype=np.float64)
    for k, (mp, m) in enumerate(zip(mp_points, m_points)):
        B[k*2:(k+1)*2, :] = np.array([[0,0,0,-m[0], -m[1], -1, mp[1]*m[0], mp[1]*m[1], mp[1]],
                                      [m[0], m[1], 1, 0,0,0, -mp[0]*m[0], -mp[0]*m[1], -mp[0]]])
    u,s,vt = np.linalg.svd(B)
    h = vt.T[:,-1]
    H = h.reshape(3,3)
    H = H/H[2,2]
    # st now if we take [m']=Hm we know which pixel in image of m should be correspondent with m'
    return H

def homog_ops(H, inputs): # inputs expected to be [3,N] or [ 2,N]
    if inputs.shape[0]==2:
        inputs = np.vstack((inputs, np.ones(inputs.shape[-1])))
    out = H@inputs
    return (out/out[-1, :])[:-1,:]

def image_of_mp_as_seen_from_m( H_m_to_mp, image_m, image_mp ):
    # Image of m' as seen from image m-s camera
    image_mp_as_m = np.zeros(image_m.shape, image_m.dtype)
    dest_xys = np.array(np.meshgrid(
                np.arange(image_m.shape[1]), np.arange(image_m.shape[0]))
                   ).reshape(-1, image_m.shape[1]*image_m.shape[0])
    # [2 (x,y), image_m.shape[0]*image_m.shape[1]] both dest_xys and look_up_mps
    look_up_mps = np.round( homog_ops( H_m_to_mp, dest_xys ) ).astype(int)
    is_out_of_bounds = (look_up_mps[0]>=0) & (look_up_mps[0]<image_mp.shape[1]) & \
                       (look_up_mps[1]>=0) & (look_up_mps[1]<image_mp.shape[0])
    # [image_m.shape[0]*image_m.shape[1]]
    for dest_m, look_up_mp, is_out in zip(dest_xys.T, look_up_mps.T, is_out_of_bounds):
        if is_out:
            image_mp_as_m[dest_m[1],dest_m[0], :] = image_mp[look_up_mp[1],look_up_mp[0], :]

    return image_mp_as_m
```

Get the Corresponding points manually

Get the corresponding points of the the left image (L) with the center one (C) and the right one (R) with the center one, since we are looking to get a homography from the side images to the center image.

In [5]:

```
pointsLC = []
pointsRC = []
pointsLC.append(choose_points(L))
pointsLC.append(choose_points(C, len(pointsLC[0])))
pointsRC.append(choose_points(R))
pointsRC.append(choose_points(C, len(pointsRC[0])))
```

In [6]:

```
print(f"Gathered {len(pointsLC[0])} and {len(pointsRC[0])} points for each pair (L,C) and (R,C) respectively")
```

```
Gathered 27 and 32 points for each pair (L,C) and (R,C) respectively
```

Get the Homographies from the C image to the L and R for an output to input strategy

For this we use the routines we implemented in the beginning.

In [6]:

```
H_C_to_L = build_homography_from_m_to_mp(pointsLC[1], pointsLC[0]) # m is C mp are the rest
H_C_to_R = build_homography_from_m_to_mp(pointsRC[1], pointsRC[0]) # m is C mp are the rest

imageL_from_C = image_of_mp_as_seen_from_m(H_C_to_L, C, L)
imageR_from_C = image_of_mp_as_seen_from_m(H_C_to_R, C, R)

plot(np.hstack((imageL_from_C,C,imageR_from_C)), size=(30,5))
```

We clearly see that both L and R gave us more information than just the registered portions. We can get the whole images by computing where the four edges of L and R go in the projective plane of C, using the inverse homographies to those computed ones. Then, we will be able to define the whole canvas from the camera extrinsics of C that leaves no pixel of any of the three images out. We will get each of the images there and we will then blend them using the average intensities where the pixels get superimposed.

So first, get the four edges in C's coordinate system for C,L and R and get the rectangular canvas that gets all of them inside.

In [3]:

```
def get_edges_of_mp_in_m(H_m_to_mp, m):
    H_mp_to_m = np.linalg.inv(H_m_to_mp)
    mp_edges = np.array([[0,0], [0,m.shape[0]], [m.shape[1], m.shape[0]], [m.shape[1], 0]]).T
    return homog_ops( H_mp_to_m, mp_edges ) #[2, 4]
```

In [7]:

```
L_edges_in_C = get_edges_of_mp_in_m(H_C_to_L, L)
R_edges_in_C = get_edges_of_mp_in_m(H_C_to_R, R)
C_edges_in_C = np.array([[0,0], [0,C.shape[0]], [C.shape[1], C.shape[0]], [C.shape[1], 0]]).T

# in the coordinates of the C image:
min_width = np.ceil(min([L_edges_in_C[0].min(), R_edges_in_C[0].min(), 0]))
max_width = np.ceil(max([L_edges_in_C[0].max(), R_edges_in_C[0].max(), C.shape[1]]))
min_height = np.ceil(min([L_edges_in_C[1].min(), R_edges_in_C[1].min(), 0]))
max_height = np.ceil(max([L_edges_in_C[1].max(), R_edges_in_C[1].max(), C.shape[0]]))
```

In [4]:

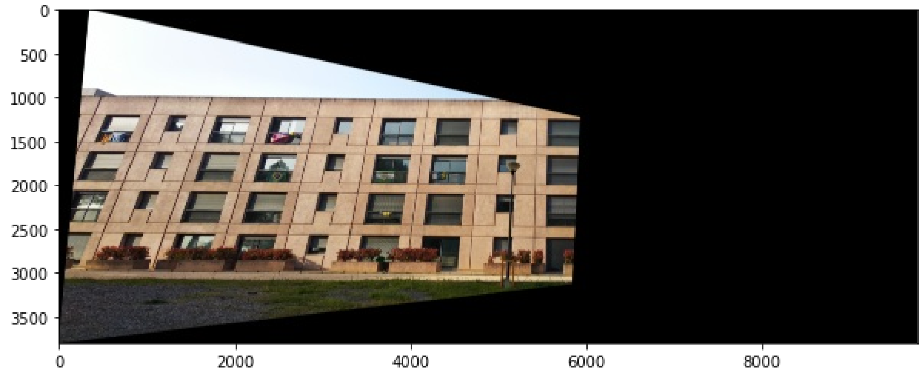
```
def fill_canvas_with_mp(H_m_to_mp, mp, canvas, min_width, min_height):
    dest_xys = np.array(np.meshgrid(
                np.arange(canvas.shape[1]), np.arange(canvas.shape[0]))
                    ).reshape(-1, canvas.shape[1]*canvas.shape[0])
    dest_xys_mp_coord_syst = dest_xys + np.array([[min_width], [min_height]])

    # [2 (x,y), canvas.shape[0]*canvas.shape[1]] both dest_xys and look_up_mps
    look_up_in_mp = np.round( homog_ops( H_m_to_mp, dest_xys_mp_coord_syst ) ).astype(int)
    is_out_of_bounds = (look_up_in_mp[0]>=0) & (look_up_in_mp[0]<mp.shape[1]) & \
                       (look_up_in_mp[1]>=0) & (look_up_in_mp[1]<mp.shape[0])
    # [m.shape[0]*m.shape[1]]
    for dest_can, look_up_mp, is_out in zip(dest_xys.T, look_up_in_mp.T, is_out_of_bounds):
        if is_out:
            canvas[dest_can[1],dest_can[0], :] = mp[look_up_mp[1],look_up_mp[0], :]
    return canvas
```

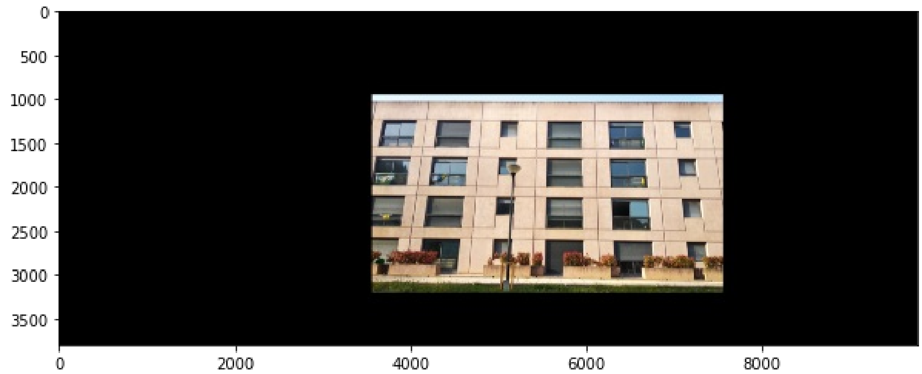
In [8]:

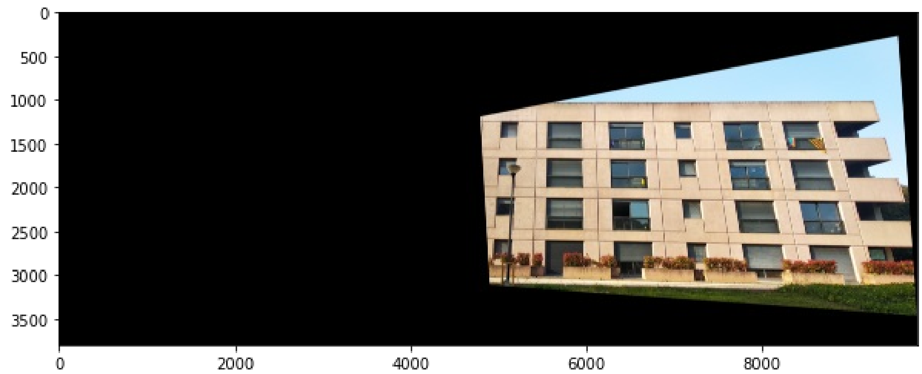
```
canvas = np.zeros( (int(max_height-min_height), int(max_width-min_width), 3), dtype=C.dtype )

canvas_L = fill_canvas_with_mp(H_C_to_L, L, canvas.copy(), min_width, min_height)
canvas_R = fill_canvas_with_mp(H_C_to_R, R, canvas.copy(), min_width, min_height)
canvas_C = fill_canvas_with_mp(np.array([[1,0,0],[0,1,0],[0,0,1]]), C, canvas.copy(), min_width, min_height)
```

In [10]:

```
plot(canvas_L)
plot(canvas_C)
plot(canvas_R)
```



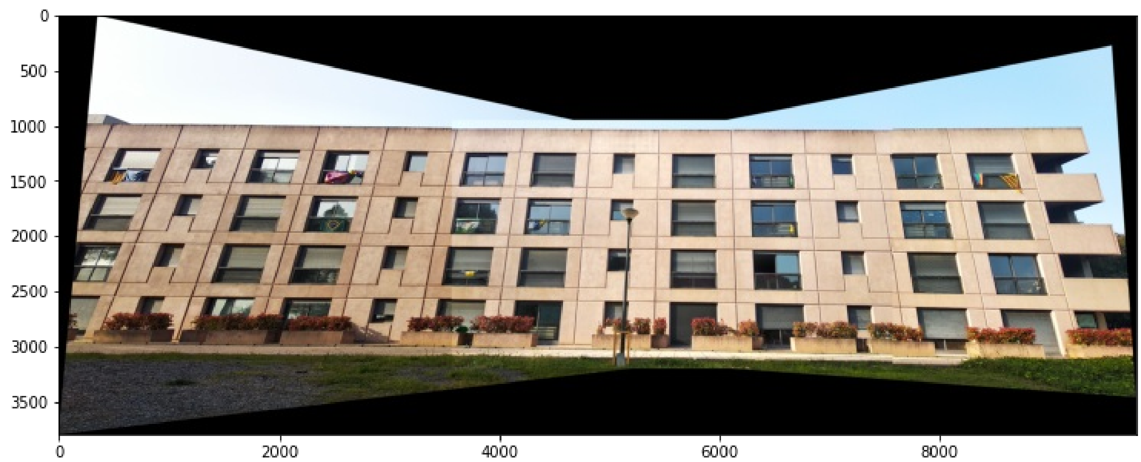




We can first attempt a simple blending, by simply getting one of the values of the images:

In [11]:

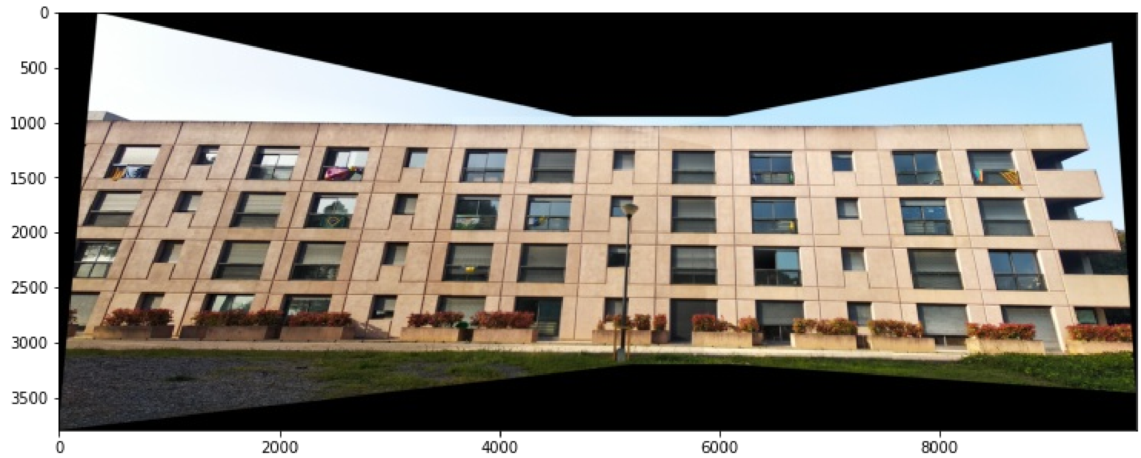
```
canvas_simple = np.where(canvas_C==0, canvas_R, canvas_C)
canvas_simple = np.where(canvas_simple==0, canvas_L, canvas_simple)
plot(canvas_simple, size=(30,5))
```



Even if it is not that perceptible for C and R, for C and L, the tonality change is very abrupt. We can improve this by getting the average of the images wherever they get superimposed.

In [12]:

```
mask_C = np.where(canvas_C.mean(axis=2)!=0, 1, 0)
mask_L = np.where(canvas_L.mean(axis=2)!=0, 1, 0)
mask_R = np.where(canvas_R.mean(axis=2)!=0, 1, 0)
mask_backg = np.where((canvas_C.mean(axis=2)==0) & (canvas_L.mean(axis=2)==0) &(canvas_R.mean(axis=2)==0), 1, 0)
mask_total = mask_C + mask_L + mask_R + mask_backg

blended = ((canvas_L.astype(np.float64)+canvas_C.astype(np.float64)+canvas_R.astype(np.float64)
                                              )/mask_total[:,:,np.newaxis]).astype(np.uint8)
plot(blended, size=(30,5))
```



Which is a way more reasonable result!

Finally, let us crop the result such that no black pixel without information is shown.

In [13]:

```
w_left = int(abs(L_edges_in_C[0,0]-L_edges_in_C[0,1]))
w_right = -int(abs(R_edges_in_C[0,2]-R_edges_in_C[0,3]))
h_top = int(-min_height)
h_bot = int(C.shape[0]-max_height)

blended_crop = blended[h_top:h_bot,w_left:w_right]
plot(blended_crop, size=(30, 10))
```

Automatic Corresponding Point Detection

We will employ a SIFT keypoint detector, with which to generate some descriptors of the locality of each keypoint in each image. Then, we can use a FLANN based keypoint matcher (instead of a brute force one), which will use an approximate nearest neighbour sort of method. We will filter the best matches (which are a reasonable distance apart) and we will use these matches to build the homography from one plane to the other one, just as we did manually. Yet, since there will be way more found correspondances than we did manually, where lots of them will be for sure noisy, it would be better to do a RANSAC employing the homography building technique, say DLT like we did, as the fitter for each RANSAC iteration.

For this openCV provides us with a `findHomography()` function, with a parameter allowing RANSAC to be used. Then we will be able to proceed as we did before.

At this point, since the procedure here and in what follows will be equivalent, we will create a function that does all this "homography+registering"-like pipeline for any two pairs of image. In a way that if we wish to do this for several images in a row, we can iteratively apply it to a pair, then the result with another image of the set etc.

In [5]:

```
def detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid( image_from, image_into
):
    if len(image_from.shape)==3: # then it is a color image
        image_from_g = cv2.cvtColor(image_from, cv2.COLOR_RGB2GRAY)
        image_into_g = cv2.cvtColor(image_into, cv2.COLOR_RGB2GRAY)

    # Initiate SIFT detector
    sift = cv2.SIFT_create()
    # find the keypoints and descriptors with SIFT
    keys_from, descr_from = sift.detectAndCompute(image_from_g,mask=None)
    keys_into, descr_into = sift.detectAndCompute(image_into_g,mask=None)

    # FLANN parameters - faster matcher than brute force
    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks=50)   # or pass empty dictionary
    flann = cv2.FlannBasedMatcher(index_params,search_params)
    matches = flann.knnMatch(descr_from, descr_into, k=2)

    # store all the good matches as per Lowe's ratio test.
    good_ones = []
    for m,n in matches:
        if m.distance < 0.7*n.distance:
            good_ones.append(m)

    # gather the correspondances we will pass to the RANSAC homography generator
    pts_from = np.float32([ keys_from[m.queryIdx].pt for m in good_ones ]).reshape(-1,1,2)
    pts_into = np.float32([ keys_into[m.trainIdx].pt for m in good_ones ]).reshape(-1,1,2)
    # Generate best homography using RANSAC
    H_into_to_from, mask = cv2.findHomography(pts_into, pts_from, cv2.RANSAC, 5.0) # get homography from image_in
to to image_from for output-input strategy
    # get the resulting RANSAC inliers
    inlier_match_mask = mask.ravel().tolist()
    # plot the inlier matches for sanity check
    draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                       singlePointColor = None,
                       matchesMask = inlier_match_mask, # draw only inliers
                       flags = 2)
    detected_inlier_match_im = cv2.drawMatches(image_from_g,keys_from, image_into_g, keys_into, good_ones,None,**
draw_params)

    # generate the canvas for the whole image
    from_edges_in_from = get_edges_of_mp_in_m(H_into_to_from, image_from)
    into_edges_in_into = np.array([[0,0], [0, image_into.shape[0]], [image_into.shape[1], image_into.shape[0]], [
image_into.shape[1], 0]]).T

    # decide the size of the canvas
    min_width = np.ceil(min([from_edges_in_from[0].min(), 0]))
    max_width = np.ceil(max([from_edges_in_from[0].max(), image_into.shape[1]]))
    min_height = np.ceil(min([from_edges_in_from[1].min(), 0]))
    max_height = np.ceil(max([from_edges_in_from[1].max(), image_into.shape[0]]))

    canvas = np.zeros( (int(max_height-min_height), int(max_width-min_width), 3), dtype=image_into.dtype )

    # fill the canvas for each view
    canvas_from = fill_canvas_with_mp(H_into_to_from, image_from, canvas.copy(), min_width, min_height)
    canvas_into = fill_canvas_with_mp(np.array([[1,0,0],[0,1,0],[0,0,1]]), image_into, canvas.copy(), min_width,
min_height)

    # Blend them using the average trick we explained in the previous section
    mask_from = np.where(canvas_from.mean(axis=2)!=0, 1, 0)
    mask_into = np.where(canvas_into.mean(axis=2)!=0, 1, 0)
    mask_backg = np.where((canvas_from.mean(axis=2)==0) & (canvas_into.mean(axis=2)==0) , 1, 0)
    mask_total = mask_from + mask_into + mask_backg

    blended = ((canvas_into.astype(np.float64)+canvas_from.astype(np.float64)
                                                )/mask_total[:,:,np.newaxis]).astype(np.uint8)
    return blended, detected_inlier_match_im
```

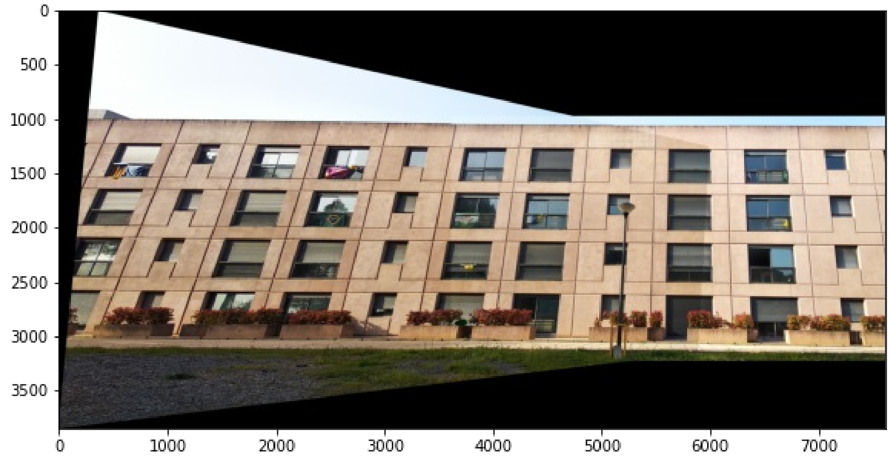
Lets apply it to L,C and then to L+C, R. We can also see the inlier keypoints that were left in the RANSAC.

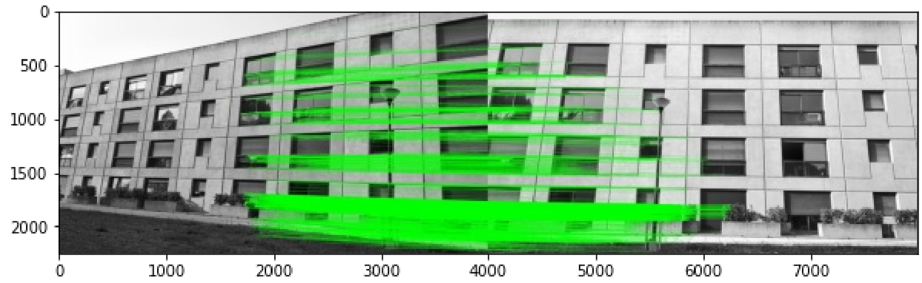
In [15]:

```
blendedLC, inliersLC = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(imag
e_from=L, image_into=C)
```

In [17]:

```
plot(blendedLC)
plot(inliersLC)
```





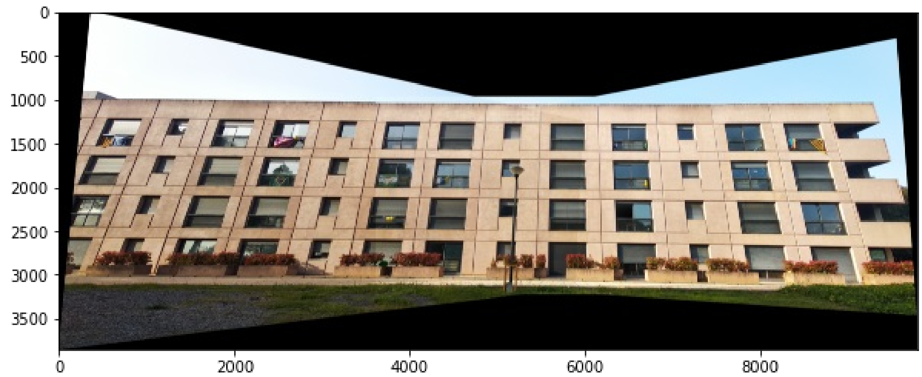
In [18]:

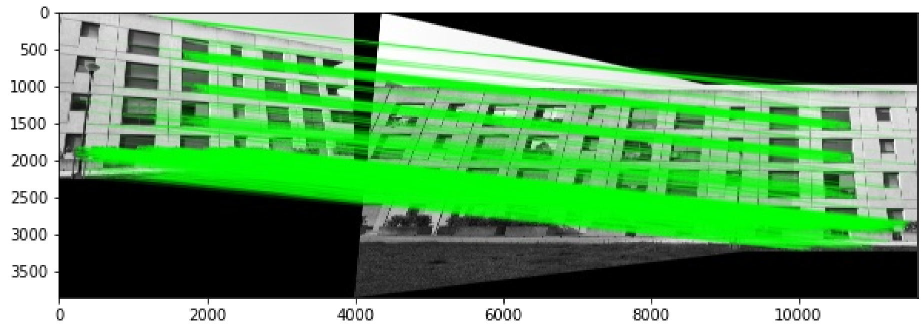
```
blendedLCR, inliersLC_R = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(i
mage_from=R, image_into=blendedLC)
```

And here the result:

In [20]:

```
plot(blendedLCR)
plot(inliersLC_R)
```



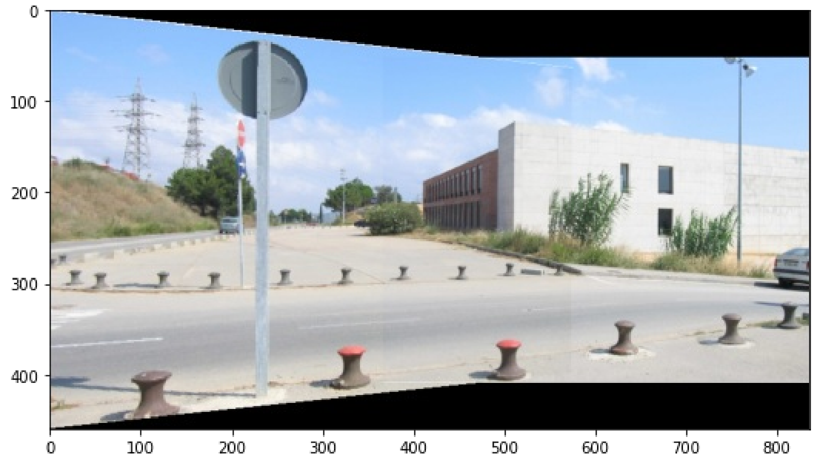


As we see, the result is at least visually, as good as what we would have obtained manually!

We can also apply this pipeline to the images given in the practice guide to get:

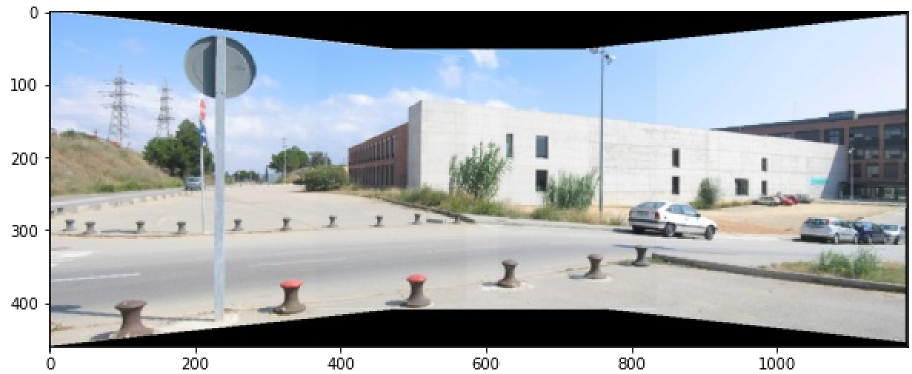
In [18]:

```
images=[]
for i in range(1, 4):
    images.append(cv2.imread(f"360_2/{i}.png")[:,:,::-1])

cummulative_mosaic1 = images[1]
for i in [0,2]:
    plot(cummulative_mosaic1)
    cummulative_mosaic1, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=images[i], image_into=cummulative_mosaic1)
```



In [19]:

```
plot(cummulative_mosaic1)
```



As we see, the pipeline works perfectly well!

Cylindrical Projection

In the practice guide, we are given the coordinate transformation that suffers a plane tangent to a cylinder when the plane is conically projected over the cylinder, with the projection pole inside the cylinder and over its rotation axis.

This means that, since if we have multiple images taken by just rotating the camera projection axis, all of them are tangent to a same cylinder, for each of them we can apply the geometrical transofrmation given by the formulas to get how the image would be formed as projected on the cylinder (conically). Because all of these images are tangent to the same cylinder where the projection pole is on the main axis of the cylinder, we know that these projections over the cylinder will necessarily be parts of the same image. Thus, they should be registrable in principle by simple translations. To do this registering, we could use visualy corresponding points and then get their relative translations in the cylinder's surface. The registering will only involve translations horizontal and vertically (vertically as well if the images were not taken really parallel to each other). Alternatively, we could employ an automatic interesting point detection, or we could register the images pairwise using Fourier phase correlation or such.

Yet another way to do this, allowing imperfect rotation of the camera along its camera axis (thus allowing the images not to be registrable with simple translations), is by generating homographies linking the cylindrically projected images. In principle, if a simple shift should be enough, then a translation is also a homography, so the result should be the same. Yet, we allow a bit more flexibility this way.

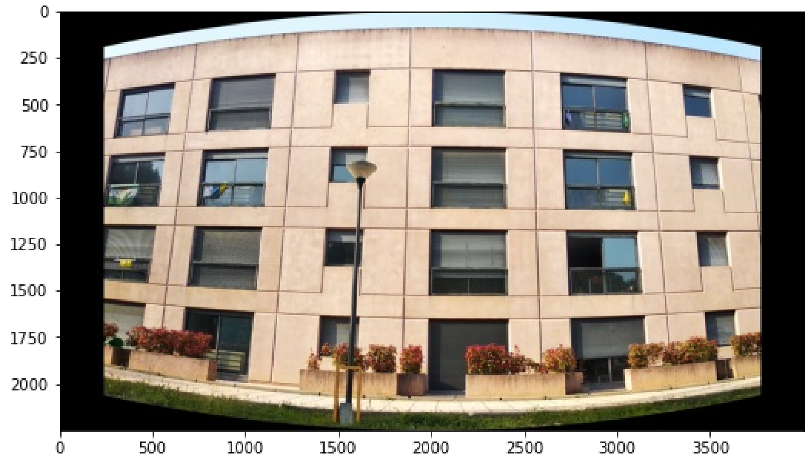
As for the equations of the cylindrical projection, we will assume the projection center is aligned with the image center (typically the value obtained in the camera calibrations). For the focal and scale factor, since we are using an FV-5 camera of a Xiaomi Redmi 5, we know the focal is about 35 mm long, with 1.3 microns per pixel. This means, in pixels, the focal is about 27000 pixels. Thus, we will need to choose f and s in the order of the thousands of pixels. We find that about 3000 gives us a reasonably curved cylinder as to try the 360 degree picture.

In [21]:

```
def centered_conical_projection_onto_cylinder( image_in_tangent_plane, f=3000, s =3000 ):
    result = np.zeros((image_in_tangent_plane.shape[0],image_in_tangent_plane.shape[1],3) , dtype=np.uint8) # we
know geometrically that the size of the output will be smaller than the input
    dest_xys = np.array(np.meshgrid(
                np.arange(result.shape[1]), np.arange(result.shape[0]))
                   ).reshape(-1, result.shape[1]*result.shape[0]) #[2, resultHxresultW]
    c_in = np.array(image_in_tangent_plane.shape)[:2]//2 # [h//2, w//2] aprox center
    c_out = np.array(result.shape)[:2]//2
    dest_xys_shifted = dest_xys - c_out[::-1, np.newaxis]
    look_up_xs_input = np.round(f*np.tan(dest_xys_shifted[0]/s)).astype(int) + c_in[1]
    look_up_ys_input = np.round(f/s*dest_xys_shifted[1]*np.sqrt(1+(np.tan(dest_xys_shifted[0]/s))**2)).astype(int
) + c_in[0]
    is_out_of_bounds = (look_up_xs_input>=0) & (look_up_xs_input<image_in_tangent_plane.shape[1]) & \
                        (look_up_ys_input>=0) & (look_up_ys_input<image_in_tangent_plane.shape[0])
    for dest_can, look_up_x, look_up_y, is_out in zip(dest_xys.T, look_up_xs_input, look_up_ys_input, is_out_of_b
ounds):
        if is_out:
            result[dest_can[1],dest_can[0], :] = image_in_tangent_plane[ look_up_y, look_up_x, :]
    return result
```

In [69]:

```
projectedL = centered_conical_projection_onto_cylinder(L)
plot(projectedL)
projectedC = centered_conical_projection_onto_cylinder(C)
plot(projectedC)
projectedR = centered_conical_projection_onto_cylinder(R)
plot(projectedR)
```

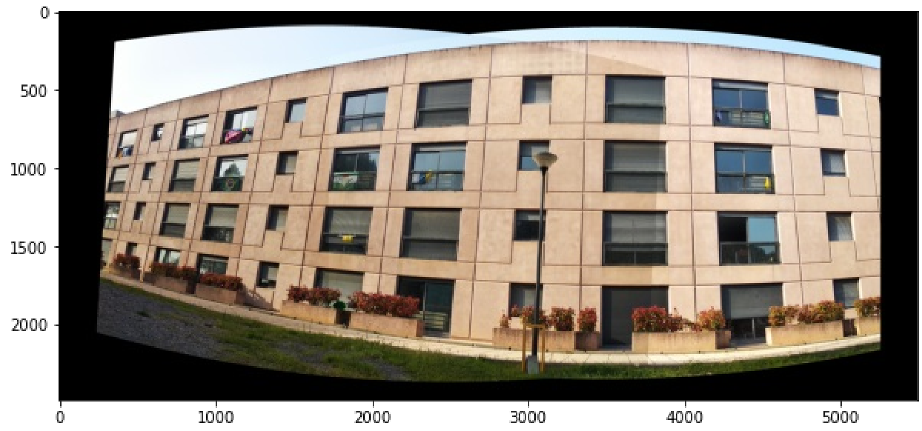


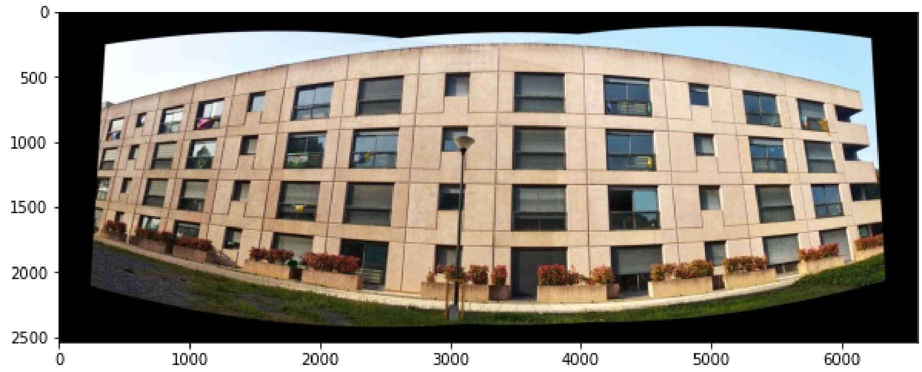
In [70]:

```
blendedLC_cyl, inliersLC_cyl = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_va
lid(image_from=projectedL, image_into=projectedC)
blendedLCR_cyl, inliersLC_R_cyl = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to
_valid(image_from=projectedR, image_into=blendedLC_cyl)
```

In [71]:

```
plot(blendedLC_cyl)
plot(blendedLCR_cyl)
```





As said, in reality the cylindrically projected images should be registrable with a simple translation (no projective change) if they were really tangent to the same cylinder of same projection pole. Yet, when the homography is generated we find a little perspective change as well! This is clearly due to the fact that the images are really not tangent to the same projection pole cylinder! (which is rather very hard to achieve by hand).

360: personal camera

After we have taken several photos as to build a 360 image, we will merge different continguous images in a pyramid-like organization (to safe resources) until we arrive to the 360 view.

In [ ]:

```
images=[]
images_cyl=[]
for i in range(1, 19):
    images.append(cv2.imread(f"360_1/{i}.jpg")[:,:,::-1])
    images_cyl.append(centered_conical_projection_onto_cylinder(images[-1], f=1000, s=1000))

cummulative_mosaic1 = images_cyl[0]
for im_cyl in images_cyl[1:4]:
    plot(cummulative_mosaic1)
    cummulative_mosaic1, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic1)

cummulative_mosaic2 = images_cyl[3]
for im_cyl in images_cyl[3:7]:
    plot(cummulative_mosaic2)
    cummulative_mosaic2, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic2)

cummulative_mosaic3 = images_cyl[6]
for im_cyl in images_cyl[6:10]:
    plot(cummulative_mosaic3)
    cummulative_mosaic3, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic3)

cummulative_mosaic4 = images_cyl[9]
for im_cyl in images_cyl[9:13]:
    plot(cummulative_mosaic4)
    cummulative_mosaic4, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic4)

cummulative_mosaic5 = images_cyl[12]
for im_cyl in images_cyl[12:16]:
    plot(cummulative_mosaic5)
    cummulative_mosaic5, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic5)

cummulative_mosaic6 = images_cyl[15]
for im_cyl in images_cyl[15:]:
    plot(cummulative_mosaic6)
    cummulative_mosaic6, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic6)

print("Obtained Cummulative mosaics:")
plot(cummulative_mosaic1)
plot(cummulative_mosaic2)
plot(cummulative_mosaic3)
plot(cummulative_mosaic4)
plot(cummulative_mosaic5)
plot(cummulative_mosaic6)
```

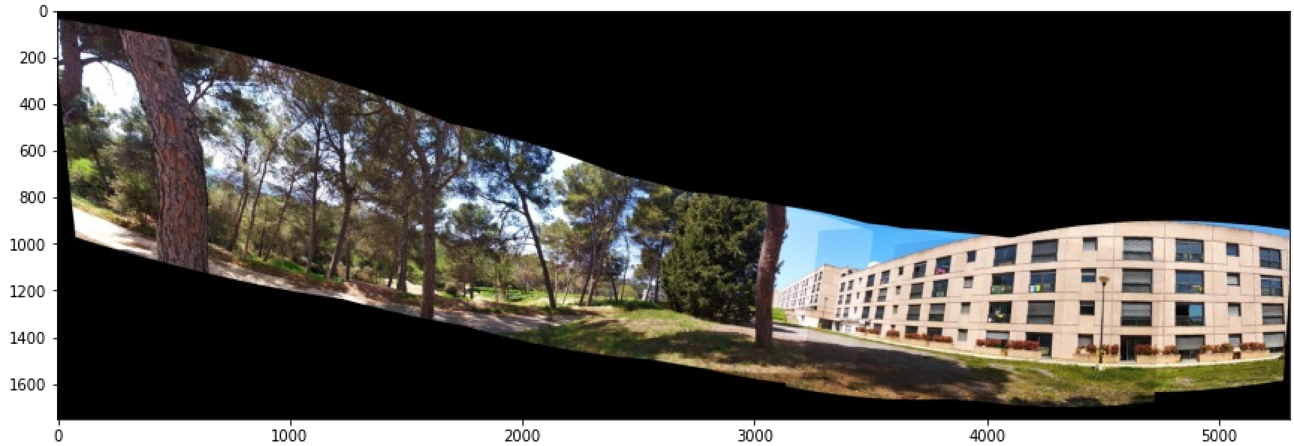
In [ ]:

```
cumml12, _ = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(cummulative_mo
saic2, cummulative_mosaic1)
plot(cumml12)
cumml126, _ = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(cummulative_m
osaic6, cumml12)
plot(cumml126)

cumml45, _ = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(cummulative_mo
saic4, cummulative_mosaic5)
plot(cumml45)
cumml453, _ = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(cummulative_m
osaic3, cumml45)
plot(cumml453)
```

Thus, we arrive at two final cummulative mosaics, which together will do a 360 view!

In [14]:

```
plot(cumml453[400:1500,50:5000], size=(30,5))
plot(cumml126[120:,200:5500], size=(30,5))
```



For the last merging, we will downscale the images, because if not the RAM cannot generate a big enough canvas.

In [ ]:

```
from skimage.transform import downscale_local_mean
cumml126_small = centered_conical_projection_onto_cylinder(
    downscale_local_mean(cumml126[120:,200:5500], (4,4,1)).astype(np.uint8), f=800, s=800)
cumml453_small = centered_conical_projection_onto_cylinder(
    downscale_local_mean(cumml453[400:1500,50:5000], (4,4,1)).astype(np.uint8), f=800, s=800)
plot(cumml453_small)
plot(cumml126_small)
```

In [49]:

```
total, _ = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(
    cumml126_small, cumml453_small)
```

In [52]:

```
plot(total[100:550, 100:1600], size=(30,10))
```

Certainly the camera was not rotated as expected!

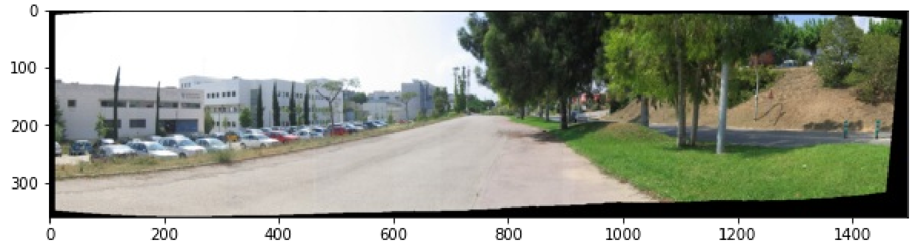
360 practice guide images

Lets now try a 360 image using the images given by the practice guide.

In [22]:

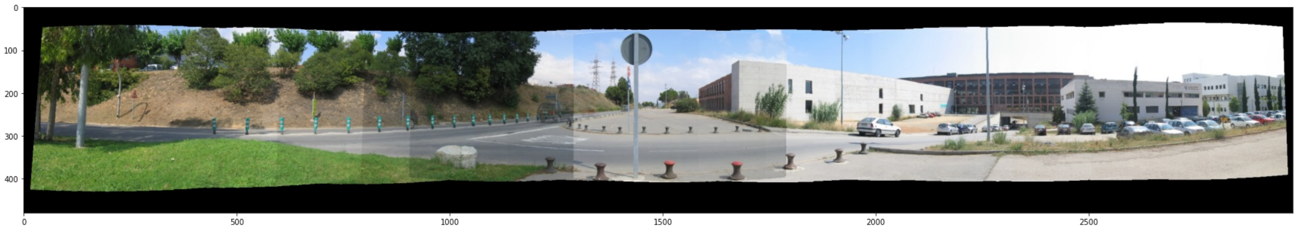
```
images=[]
images_cyl=[]
for i in range(1, 19):
    images.append(cv2.imread(f"360_2/{i}.png")[:,:,::-1])
    images_cyl.append(centered_conical_projection_onto_cylinder(images[-1], f=650, s=650))

cummulative_mosaic1 = images_cyl[0]
for im_cyl in images_cyl[1:7]:
    #plot(cummulative_mosaic1)
    cummulative_mosaic1, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic1)

cummulative_mosaic2 = images_cyl[6]
for im_cyl in images_cyl[6:13]:
    #plot(cummulative_mosaic2)
    cummulative_mosaic2, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic2)

cummulative_mosaic3 = images_cyl[12]
for im_cyl in images_cyl[12:]:
    #plot(cummulative_mosaic3)
    cummulative_mosaic3, inliers = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_t
o_valid(
                        image_from=im_cyl, image_into=cummulative_mosaic3)

print("Obtained Cummulative mosaics:")
plot(cummulative_mosaic1)
plot(cummulative_mosaic2)
plot(cummulative_mosaic3)
```

Obtained Cummulative mosaics:



In [23]:

```
cumml13, _ = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(cummulative_mo
saic3, cummulative_mosaic1)
plot(cumml13, size=(30,5))
cumml360, _ = detect_filter_keypoints_get_matches_find_ransac_homography_make_collage_crop_to_valid(cummulative_m
osaic2, cumml13)
plot(cumml360, size=(30,5))
```



We can see that the brightness change of the images is transferred to the final mosaic. A bit of image pre-processing to make all the images have the same range of light should correct this undesired effect.

END