Image Processing - 67829

Exercise 4: Panorama Registration & Stitching

Due date: 15/01/17

1 Overview

In this exercise you will be guided through the steps we discussed in class to perform automatic panorama generation. The input of such an algorithm is a sequence of images scanning a scene from left to right (due to camera rotation - homography between images), with significant overlap in the field of view of consecutive frames. The output will be a large field of view image of the scene combining all input images. This exercise covers the following steps:

- Registration: The geometric transformation between each consecutive image pair is found by detecting *Harris feature points*, extracting their *MOPS-like descriptors*, matching these descriptors between the pair and fitting a homography transformation that agrees with a large set of inlying matches using the *RANSAC algorithm*.
- Stitching: The above image-pair homographies are multiplied to give homographies transforming all frames into a common coordinate system. The panorama is rendered in this common coordinate system by *back-warping* from each frame into a vertical strip of the panorama.

2 Background and Dependencies

You will need the functions `read_image`, `build_gaussian_pyramid`, `blur_spatial` and `pyramid_blending` from the previous exercises. Include these and all of their dependencies in a file named sol4_utils.py, imported from your sol4.py. Don't forget to add this file to your submission. Nevertheless, the implementation of these functions will not affect your grade.

3 Image Pair Registration

In this section we will focus on computing the geometric transformation between a pair of consecutive frames, I_i and I_{i+1}, of some image sequence. For an example of such a frame pair, have a look at the example input image sequences provided in the supplementary images. Although the provided input sequences are of RGB images, in the current registration phase we will only require I_i and I_{i+1} to be grayscale images (load them accordingly).

3.1 Feature point detection and descriptor extraction

As discussed in class, not all points in an image are good candidates for matching. To detect points in a frame that can be localized well and that will be likely reproduced in the consecutive frame we will use the *Harris corner detector*. To do this you will implement a function `harris_corner_detector` that gets a grayscale image and returns (x, y) locations that represent corners in the image. We will not implement the Scale Invariant version of the Harris Detector.
You should implement the following algorithm:

- Get the I_x and I_y derivatives of the image using the filters $[1, 0, -1], [1, 0, -1]^T$ respectively.
- Blur the images: ${I_x}^2$, ${I_y}^2$, $I_x I_y$. You may use your `blur_spatial` function from `ex2` with `kernel_size=3`.
- Then, for each pixel you will have the following matrix M:

$$\begin{pmatrix} {I_x}^2 & I_x I_y \\ I_y I_x & {I_y}^2 \end{pmatrix}$$

- As you learned in class, the eigenvalues of this matrix tells us about the intensity changes of a window around this pixel, in a small neighborhood of the pixel. No intensity change means a constant gray level over the entire window. A big change in one direction (one large eigenvalue) means this window contains an edge. A big change in two directions (two large eigenvalues) means that this window contains a corner. One way to measure how big are the two eigenvalues is

$$R = det(M) - k(trace(M))^2$$

  We will use $k = 0.04$.

- Finding R for every pixel results in a response image R. The corners are the local maximum points of R. To find those points you should use the supplied `non_maximum_suppression` function, which gets a response image as an input, thresholds out areas with low response and returns a binary image contains the local maximum points.
- Return the xy coordinates of the corners.

The function should implement the following API:

```
pos = harris_corner_detector (im)
where
im − grayscale image to find key points inside.
pos − An array with shape (N,2) of [x,y] key points locations in im.
```

We would prefer though that the feature positions we locate in each image are spread-out and not concentrated in one small region of the image (a region that happens to have a lot of texture and contrast). For this purpose you are provided with the `spread_out_corners` function. Have a look at it. This function splits the input image `im` into $n \times m$ approximately equal sub-images and runs your `harris_corner_detector` on each. `spread_out_corners` should be used in your code instead of calling `harris_corner_detector` directly. You are encouraged to experiment with the values of the `n` and `m` parameters but a good starting point should be `n=7` and `m=7`. To visualize the detected corner positions, display the image and then plot the points using `plt.scatter` of `plt.plot` with `'.'` (see tips section).

Now that we've detected feature points in two images, we would like to extract descriptors at their locations. The type of descriptor you are guided to implement here is a simplified version of the MOPS descriptor presented in class. To sample this descriptor at position $p = (x, y)$ in an image I we will first need to prepare a 3-level Gaussian pyramid $G_I[l]$ of this image, where the index l here is used to denote the level ($l = 0$ being the original image). The descriptor we will extract, denoted by d, is a 7×7 matrix of normalized image intensities. These should be sampled in the third pyramid level image, $G_I[2]$, in a 7×7 patch centered around p_{l_3}, which is the location of the point $p = (x, y)$ in the 3rd level. In general you can transform point coordinates between any two pyramid levels l_i and l_j in the following way

$$p_{l_j} = (x_{l_j}, y_{l_j}) = 2^{l_i - l_j} p_{l_i} = 2^{l_i - l_j}(x_{l_i}, y_{l_i}) \tag{1}$$

Note that the coordinates at which we would like to sample this 7×7 patch at level 3 of the pyramid won't necessarily be integer valued. You will therefore need to sample them at these sub-pixel coordinates by interpolating within the pyramid's 3rd level image properly. To do this use `map_coordinates` from

`scipy.ndimage` (see the Tips section). Once this 7×7 intensity matrix has been sampled we would like to normalize it so that the resulting descriptor is invariant to certain changes of lighting. Denoting by $\tilde{d}$ the sampled descriptor matrix, the final descriptor matrix is $d = (\tilde{d} - \mu)/||\tilde{d} - \mu||$ where μ is the mean $\tilde{d}$ and $||.||$ is the euclidean norm operation (use `np.linalg.norm`). Descriptor sampling should be implemented in the `sample_descriptor` function having the following interface

```
desc = sample_descriptor(im, pos, desc_rad)
where
im − grayscale image to sample within.
pos − An array with shape (N,2) of [x,y] positions to sample descriptors in im.
desc_rad − "Radius" of descriptors to compute (see below).
desc − A 3D array with shape (K,K,N) containing the ith descriptor at desc(:,:,i). The per−descriptor dimensions KxK
    are related to the desc_rad argument as follows K = 1+2∗desc_rad.
```

Note that `sample_descriptor` already expects the grayscale image `im` to be the 3rd level pyramid image. Note also that to obtain 7×7 descriptors, `desc_rad` should be set to 3.

The `sample_descriptor` function should be called by a wrapper function you will implement called `find_features` which has the following interface

```
pos, desc = find_features (pyr)
pyr − Gaussian pyramid of a grayscale image having 3 levels.
pos − An array with shape (N,2) of [x,y] feature location per row found in the (third pyramid level of the) image. These
    coordinates are provided at the pyramid level pyr[0].
desc − A feature descriptor array with shape (K,K,N).
```

This function is responsible for both the feature detection and the descriptor extraction. `find_features` detects feature points in the pyramid and samples their descriptors. This function should call the functions `spread_out_corners` for getting the keypoints, and `sample_descriptor` for sampling a descriptor for each keypoint.

3.2 Matching descriptors

After obtaining the descriptor matrices D_i and D_{i+1} (the *desc* matrices returned by `find_features`) extracted from images I_i and I_{i+1}, we would now like to match features in one frame to the corresponding features in the other. Note that the number of features in D_i detected in frame i will in general differ from the number of features in D_{i+1} detected in frame $i+1$. We denote by $D_{i,j}$ the jth feature descriptor detected in the ith frame. The match score we choose between two descriptors will simply be their dot-product (flattened to 1D arrays). Therefore $S_{j,k} = D_{i,j} \cdot D_{i+1,k}$ would be the match score between the

jth descriptor in frame i and the kth descriptor in frame $i+1$. We will say that descriptors $D_{i,j}$ and $D_{i+1,k}$ match if the following three properties hold

- $S_{j,k} \geq 2_{nd}max\{S_{j,l} \mid l \in \{0..f_1 - 1\}\}$ where f_1 is the number of features in frame $i+1$. Explanation: $S_{j,k}$ is in the best 2 features that match feature j in image i, from all features in image $i+1$.
- $S_{j,k} \geq 2_{nd}max\{S_{l,k} \mid l \in \{0..f_0 - 1\}\}$ where f_0 is the number of features in frame i. Explanation: $S_{j,k}$ is in the best 2 features that match feature k in image $i+1$, from all features in image i.
- $S_{j,k}$ is greater than some predefined minimal score (called `min_score` in the following).

Note that these dot-products are necessarily in the range $[-1, 1]$ because of the way we normalized the descriptors. You should tweak the `min_score` parameter until you find that you're obtaining good matches. A good value to start with is `min_score=0.5`. In general, reasonable values for would be in the range $[0, 1)$. The function `match_features` performing this matching procedure should be implemented with the following interface

match_ind1, match_ind2 = match_features (desc1, desc2, min_score)
where
desc1 − A feature descriptor array with shape (K,K,N1).
desc2 − A feature descriptor array with shape (K,K,N2).
min_score − Minimal match score between two descriptors required to be regarded as corresponding points.
match_ind1 − Array with shape (M,) and dtype **int** of matching indices in desc1.
match_ind2 − Array with shape (M,) and dtype **int** of matching indices in desc2.

It will match feature descriptors in desc1 and desc2.
Note:

- The descriptors of the `j`th match are `desc1[:,:,match_ind1[j]]` in image I_i and `desc2[:,:,match_ind2[j]]` in image I_{i+1}.
- The number of feature descriptors N1 generally differs from N2.
- The value of M depends on the actual number of corresponding pairs your algorithm will find in the images.

3.3 Registering the transformation

We will now use the feature point matches found above to compute the most fitting homography that transforms I_i to the coordinate system of I_{i+1}. To do this we will first need to implement a function that

applies a homography transformation on a set of points. This is the `apply_homography` function you are required to implement

pos2 = apply_homography(pos1, H12)
where
pos1 − An array with shape (N,2) of [x,y] point coordinates.
H12 − A 3x3 homography matrix.
pos2 − An array with the same shape as pos1 with [x,y] point coordinates in image i+1 obtained from transforming pos1 using H12.

As a reminder to how homographies transform points - given a point (x_1, y_1) in coordinate system 1 and a homography matrix $H_{1,2}$, then `apply_homography` will transform this point to (x_2, y_2) in coordinate system 2 in the following way

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = H_{1,2} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \end{bmatrix} \Big/ \tilde{z}_2 \tag{3}$$

We will now use `apply_homography` in our RANSAC (Random SAmple Consensus) homography fitting code. Let's first recall how RANSAC operates. Given 2 sets of N matched points P_1 and P_2 s.t. $P_{1,j}$ are the x-y coordinates of the jth match in image 1 and $P_{2,j}$ are the x-y coordinates of the jth match in image 2

- Pick a random set of 4 point matches from the supplied N point matches. Let's denote their indices by J. We call these two sets of 4 points in the two images $P_{1,J}$ and $P_{2,J}$.
- Compute the homography $H_{1,2}$ that transforms the 4 points $P_{1,J}$ to the 4 points $P_{2,J}$. As discussed in class, there is a closed form solution that does this. To simplify matters you have been provided with the `least_squares_homography` function that performs this step.
- Use $H_{1,2}$ to transform the full set of points P_1 in image 1 to the transformed set P_2' (using the above `apply_homography`) and compute the squared euclidean distance $E_j = ||P'_{2,j} - P_{2,j}||^2$ for $j = 0..N-1$. Mark all matches having $E_j <$ `inlier_tol` as inlier matches and the rest as outlier matches for some constant threshold `inlier_tol`.

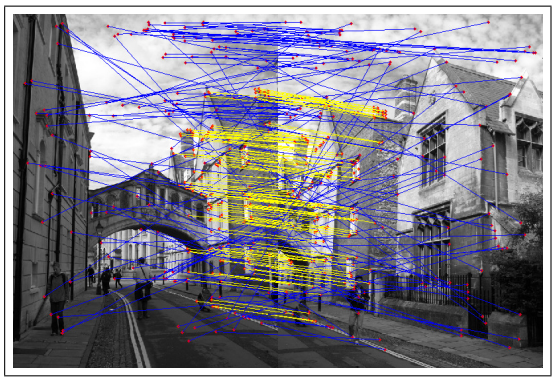
RANSAC performs several iterations (later denoted `num_iters`) of these 3 steps, keeping a record of the largest inlier match set J_{in} it has come upon. Once these iterations are complete the homography is recomputed over the matches J_{in}. To do this you will simply need to rerun `least_squares_homography` on these inlier point matches $P_{1,J_{in}}$ and $P_{2,J_{in}}$ and obtain the final least squares fit of the homography over the largest inlier set. Your RANSAC implementation should have the following interface

H12, inliers = ransac_homography(pos1, pos2, num_iters, inlier_tol)
where
pos1,pos2 − Two Arrays, each with shape (N,2) containing n rows of [x,y] coordinates of matched points.
num_iters − Number of RANSAC iterations to perform.
inlier_tol − inlier tolerance threshold.
H12 − A 3x3 normalized homography matrix.
inliers − An Array with shape (S,) where S is the number of inliers, containing the indices in pos1/pos2 of the maximal set of inlier matches found.

To visualize the full set of point matches and the inlier matches detected by RANSAC implement the following display function

display_matches(im1, im2, pos1, pos2, inliers)
im1, im2 − two grayscale images
pos1, pos2 − Two arrays with shape (N,2) each, containing N rows of [x,y] coordinates of matched points in im1 and im2 (i.e. the match of the ith coordinate is pos1[i,:] in im1 and pos2[i,:] in im2).
inliers − An array with shape (S,) of inlier matches (e.g. see output of ransac_homography)

This function should display a horizontally concatenated image (use `np.hstack` of an image pair `im1` and `im2`, with the matched points provided in `pos1` and `pos2` overlayed correspondingly as red dots. Each outlier match, say at index j, is denoted by plotting a blue line between `pos1[j,:]` and the horizontally shifted `pos2[j,:]`. Similarly, inlier matches are denoted by plotting a yellow line between the matched points (see section 8 for tips). As an example of how the output should look like, below is the figure resulting from running `display_matches` on an image pair on one of the provided example sequences together with its point matches `pos1`, `pos2` (obtained from `find_features` + `match_features`) and inlier index set `inliers` obtained from `ransac_homography`.



This figure shows the large number of outlier matches the RANSAC algorithm had to deal with. As opposed to the outlier match set (shown in blue) the inlier set (shown in yellow) shows a more consistent motion. Also note that although many features detected in the cloud texture were probably matched correctly, these were marked as outliers. This is probably due to the fact that the clouds had enough time to move between these two shots, and due to the value of `inlier_tol`. Your results will differ from the above example due to the random nature of RANSAC.

4 Panorama Stitching

Our efforts in the previous section eventually provided us with the set of registered homographies $H_{i,i+1}$, for $i = 0..M-2$, between consecutive frames in a given image sequence of M frames I_i. We would now like to use these homographies to stitch these M frames into one combined panorama frame.

4.1 Transforming to a common coordinate system

The first stage in doing this is to pick a coordinate system in which we would like the panorama to be rendered. We will (somewhat arbitrarily) choose this coordinate system to be the coordinate system of the middle frame I_m in our image sequence, where `m = (M-1)//2`. What we mean by this is that the resulting panorama image will be composed of frame I_m with all the other frames back-warped so that they properly align with it. To achieve this we will need to translate the set of homographies $H_{i,i+1}$ that transform image coordinates in frame I_i to frame I_{i+1} into a different set of homographies $\bar{H}_{i,m}$ that transform image coordinates in frame I_i to this middle frame I_m.

First we note that given 2 homography matrices $H_{a,b}$ and $H_{b,c}$ and a point p_a in coordinate system a, we can transform the point to coordinate system c by first operating with homography $H_{a,b}$ to obtain an intermediate point p_b in system b and then by operating with homography $H_{b,c}$ to obtain p_c in system c. Each time we operate with a homography H on a point $p = (x, y)$ we do two things: first we multiply the

homogeneous column 3-vector $\tilde{p} = (x, y, 1)^T$ from the left by H and then we re-normalize to keep the 3rd element equal to 1. When we operated on p_a first with $H_{a,b}$ and then with $H_{b,c}$ we could have equally well multiplied the homogeneous vector $\tilde{p}_a$ once from the left by the matrix $H_{b,c}H_{a,b}$ and then normalized only once at the very end. We see then that multiplying the homography matrices $H_{a,b}$ and $H_{b,c}$ has produced a new homography matrix $H_{a,c} = H_{b,c}H_{a,b}$ that now transforms points from coordinate system a directly to c.

We may now use this property of homographies to obtain $\bar{H}_{i,m}$ from $\{H_{i,i+1} : i = 0..M-1\}$. We do this as follows

- For $i < m$ we set $\bar{H}_{i,m} = H_{m-1,m} * ... * H_{i+1,i+2} * H_{i,i+1}$
- For $i > m$ we set $\bar{H}_{i,m} = H^{-1}_{m,m+1} * ... * H^{-1}_{i-2,i-1} * H^{-1}_{i-1,i}$
- For $i = m$ we set $\bar{H}_{i,m}$ to the 3×3 identity matrix $I =$ `np.eye(3)`

This procedure should be implemented in the following function

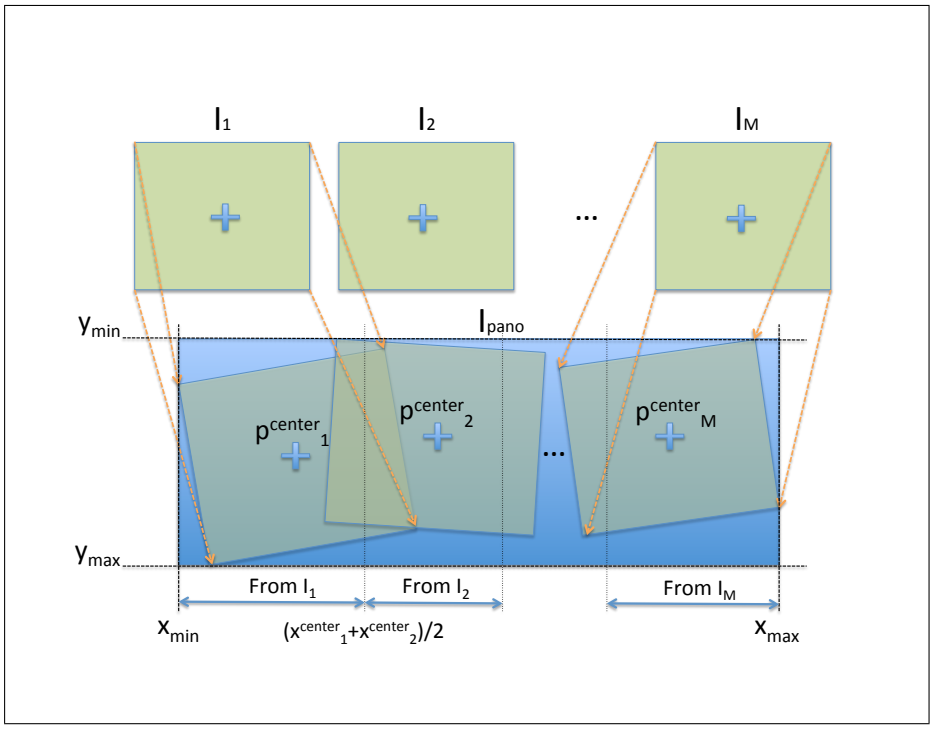
H2m = accumulate_homographies(H_successive, m)
where
H_successive − A list of M−1 3x3 homography matrices where H_successive[i] is a homography that transforms points from coordinate system i to coordinate system i+1.
m − Index of the coordinate system we would like to accumulate the given homographies towards.
H2m − A list of M 3x3 homography matrices, where H2m[i] transforms points from coordinate system i to coordinate system m.

Note: In this exercise homography matrices should always maintain the property that H[2,2]==1. This should be done by normalizing them as follows before using them to perform transformations H /= H[2,2]. In this function's interface the list of homographies `H_successive` corresponds in the above discussion to the set of homographies $H_{i,i+1}$ and the returned `H2m` corresponds to the set $\bar{H}_{i,m}$.

4.2 Rendering the panorama

Now that we have the set of M homographies $\bar{H}_{i,m}$ transforming pixel coordinates in frame I_i to the panorama coordinate system, we now proceed to describing the way in which the panorama frame I_{pano} is rendered. First we will need to define where we want I_{pano} to be rendered. We would like this region to be large enough to include all pixels from all frames I_i. To do so we compute where the 4 corner pixel coordinates (top-left, top-right, bottom-right, bottom-left) of each frame I_i get mapped to by $\bar{H}_{i,m}$, denoting these positions in the panorama coordinate system by $p_i^{corner_k}$ where $k = 1..4$. The coordinates of a rectangle bounding the set of $4 * M$ corners $p_i^{corner_k}$ denoted $x_{max}, x_{min}, y_{max}, y_{min}$, will then define

the region in which the panorama image I_{pano} should be rendered. We will now define what parts of I_{pano} should be obtained from which frame I_i. We divide the panorama to M vertical strips, each covering a portion of the full lateral range $[x_{min}, x_{max}]$. The boundaries between these strips are taken to be the following $M-1$ x-coordinates: $(x_i^{center} + x_{i+1}^{center})/2$ for all $i = 0..M-2$, where we denote by x_i^{center} the x-coordinate of the center of the ith image p_i^{center} in the panorama coordinate system (obtained again using $\bar{H}_{i,m}$). This division of the panorama to vertical strips is shown in the following figure



Back-warping of the strips should then be performed as follows. Initially, prepare the final RGB panorama image I_{pano} with dimensions $(y_{max} - y_{min} + 1) \times (x_{max} - x_{min} + 1) \times 3$. For every input image prepare coordinate strips X_coord and Y_coord using the function `np.meshgrid` to hold the x and y coordinates of each of the panorama pixels originates from that image. Recall that these should be in the two ranges $[x_{min}, x_{max}]$ and $[y_{min}, y_{max}]$. Each strip spans the full Y range $[y_{min}, y_{max}]$ and part of the X range (with a small overlap for stitching).

Now, for each strip $i = 0..M-1$, the coordinate mesh created for this strip, denoted X^i and Y^i should be transformed by the **inverse** homography $\bar{H}_{i,m}^{-1}$ using `apply_homography` **back** to the coordinate system of frame i. We call these X'^i and Y'^i. These back-warped coordinates can now be used to interpolate the image with `map_coordinates`. The interpolated pixel values should be inserted to strip i of the panorama frame.

4.3 Stiching

You should use your `pyramid_blending` function from ex3, for blending each new strip with the current panorama image. We want you to think how you can do that given the image strips from the input images. You may choose not to do so - just putting the strips side by side, but you will lose some points for this part.

The function implementing the panorama rendering should have the following interface

panorama = render_panorama(ims, Hs)
ims − A list of grayscale images. (Python list)
Hs − A list of 3x3 homography matrices. Hs[i] is a homography that transforms points from the coordinate system of ims [i] to the coordinate system of the panorama. (Python list)
panorama − A grayscale panorama image composed of vertical strips, backwarped using homographies from Hs, one from every image in ims.

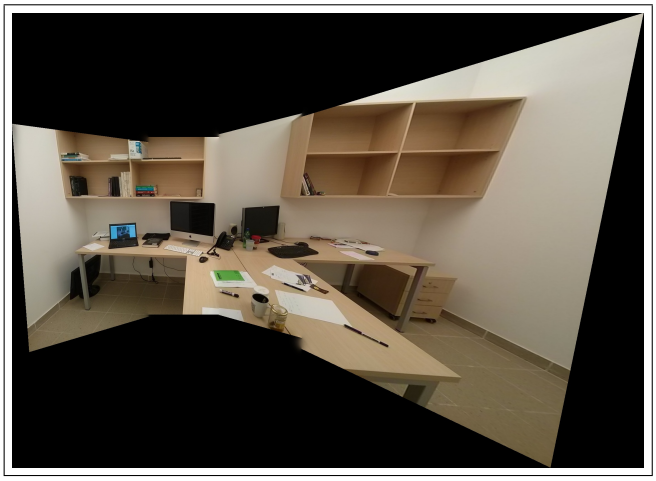
5 Tying it all Together

To ease the development burden, the main function, `generate_panorama` responsible for all the housekeeping has been provided. `generate_panorama` essentially goes through the following steps:

- Read grayscale frames (using your `read_image`).
- Find feature points and compute their descriptors (using your `gaussian_pyramid` and `find_features`).
- Match feature points (using your `match_features`).
- Register homography pairs (using your `ransac_homography`).
- Display inlier and outlier matches for the result of `ransac_homography` function (using your `display_matches`).
- Transform the homographies (using your `accumulate_homographies`).
- Load the RGB frames.
- Render panorama of each color channel (using your `render_panorama`).

Have a look at `generate_panorama`'s code so that you understand this flow. Once your implementation is complete you can also execute the provided `example_panoramas.py`. This script calls `generate_panorama` several times to generate the panoramas for all the provided example image sequences. The results should look something like the images below.

(a) oxford



(b) office

(c) backyard

You are allowed to alter `generate_panorama` as long as it still functions as described in this definition to generate panoramas. More specifically `generate_panorama` contains some predefined values for the constants `num_iters`, `inlier_tol` (for RANSAC) and `min_score` which you are encouraged to change to suite your needs.

6 Your Panorama

You should submit with your exercise one panorama image sequence having the same naming convention as the example sequences. Place this image sequence in the `external` directory. Also submit a script called `my_panorama.py` which operates in a similar manner to `example_panoramas.py` and that produces a panorama out of your image sequence. This resulting panorama should be saved back into the `external` directory. Don't include the images we provided in your submission. Make sure your images are taken with rotation only around the camera center (as much as possible). Make sure the size of the images is not too large (no more than approx. 600x600 pixels).

7 Bonus - 10 points

Change the way the different strips are combined in the `render_panorama` function by performing dynamic-programming stitching. Describe your solution in a file named bonus.txt. To test yourself make sure your method works well with images with parallax, where there is also translation of the camera between images (so without a special stitching, there will be artifacts in the resulted panorama). Submit the parallax sequence of images in your external folder.

8 Tips & Guidelines

- Start early. This is the largest exercise and it will also be weighted accordingly in the final grade.
- Avoid using loops when you can. Creating a panorama of 2,3,4 frames should not take more than 20,35,50 seconds respectively, using the CS-lab computers (might be slightly slower if you have implemented the bonus part).
- We have added a timeout to the presubmit script. It gives your code a long time, 180 seconds for a 2-image panorama. Make sure you pass the script before submission. If your code is slow use timers to analyze it and identify the bottleneck. Avoid unnecessary loops, and use vectorized operations wherever you can.

- Try to debug each function you implement first separately from the whole program.
- Don't forget to document additional functions you use properly.
- Your submission file should be formatted according to the submission guidelines.
- After rendering an image using e.g. `plt.imshow` you can use `plt.plot` to overlay points and lines on the image.
- You can control the parameters of the plotted points or lines through keyword arguments. E.g. `plt.plot(x, y, color='green', linestyle='dashed', marker='o', markerfacecolor='blue', markersize=12, lw=.5)`. see `http://matplotlib.org/api/lines_api.html#matplotlib.lines.Line2D` for many more options.
- To use the function `plt.plot` to draw a single thin blue line between two wide red points (`x[0]`,`y[0]`) and (`x[1]`,`y[1]`), you can execute `plt.plot(x, y, mfc='r', c='b', lw=.4, ms=10, marker='o')`.
- The functions we've provided are located in sol4_add.py, you should import it from your sol4.py.
- In `ransac_homography` you may find the package `np.random` useful, especially `np.random.permutation`.
- Use `map_coordinates` with `order=1` and `prefilter=False` for linear interpolation.
- The sizes of the supplied images are not powers of 2, if it causes you a problem when reconstructing images from laplacian pyramids (should occur only in stitching) you may crop the images or use less pyramid levels.
- There might be a slight difference in your results between different runs due to the randomness of the RANSAC process. If you feel that is the case, use more iterations to ensure it converges to homographies which are good enough.

Good luck and enjoy!