Image Processing - 67829

Exercise 4: Panorama Registration & Stitching

Due date: 27/12/2015

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. look at Tirgul 6, silde 7 at the example in the top-right corner) different parts of a scene, 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 *backwarping* from each frame into a vertical strip of the panorama.

2 Background and Dependencies

You may find it helpful to review the material that was presented in exercise classes 6-8. You will also require the files `imReadAndConvert.m` , `GaussianPyramid.m`,`blurInImageSpace.m` and `pyramidBlending.m` that you submitted in exercises 1 and 3. Don't forget to include these files and all of their dependencies in your submission.

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 `ex4.zip`. These are located in the directory `ex4/data/inp/examples`. 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 the 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 `HarrisCornerDetector` that gets a grayscale image and returns (x, y) locations in the image represent corners. We will not implement the Scale Invariant version of 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]` respectively.
- Blur the images: ${I_x}^2$, ${I_y}^2$, I_xI_y. You should use your `blurInImageSpace.m` function from `ex3` with `kernelSize=3` (please cancel the imshow in this function).
- Then, for each pixel you will have the following matrix M:

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

- 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 a pixel. No intensity change means a a constant grey level in the window. A big change in one direction (one big eigenvalue) means that this window contains an edge. A big change in two directions (two big 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 a response image R. The corners are the local maximum points of R. To find this points you should use the supplied `nonMaximumSuppression` function, which gets a response image as an input, threshold out areas with low response and returns a binary image that contains the local maximum points.
- Return the xy coordinates of the corners.

You should implement the following api:

```
function pos = HarrisCornerDetector( im )
% HARRISCORNERDETECTOR Extract key points from the image.
% Arguments:
% im − nxm grayscale image to find key points inside.
% pos − A nx2 matrix of [x,y] key points positions 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 (in the `ex4/code` directory) with the `spreadOutCorners` function. Have a look at its documentation. This function splits the input image `im` into $n \times m$ approximately equal sub-images and runs your `HarrisCornerDetector` on each. `spreadOutCorners` should be used in your code instead of calling `HarrisCornerDetector` directly. You are encouraged to experiment with the values of the `n`, `m` and `maxNumCorners` parameters but a good starting point should be `n=7` and `m=7`. To visualize the detected corner positions, display the image using `imshow`, `hold on;` and the detected point positions using `plot` (see section 8 for their combined usage).

Now that we've detected feature point position matrices for two frames, we would like to extract descriptors at these positions. 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 ($l = 1$ being the heighest resolution level). The descriptor itself 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, $G_I(3)$, in a 7×7 patch centered around p_{l3}, 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_1 and l_2 in the following way

$$p_{l_2} \equiv (x_{l_2}, y_{l_2}) = 2^{l_1 - l_2}(p_{l_1} - 1) + 1 = 2^{l_1 - l_2}\left[(x_{l_1}, y_{l_1}) - 1\right] + 1 \tag{1}$$

where we are minding to take care of Matlab's 1-based indexing. 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 Matlab's `interp2` function mentioned in exercise class 6. As usual, also refer to `interp2`'s documentation. 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 unnormalized descriptor matrix, then the final decriptor matrix is $d = (\tilde{d} - \mu)/||\tilde{d} - \mu||$ where μ is the mean of all elements of the matrix $\tilde{d}$ and $||.||$ is the euclidean norm operation (use Matlab's `norm(window(:))` function). Descriptor sampling should be implemented in the `sampleDescriptor` function having the following interface

```
function desc = sampleDescriptor(im,pos,descRad)
% SAMPLEDESCRIPTOR Sample a MOPS−like descriptor at given positions in the image.
% Arguments:
% im − nxm grayscale image to sample within.
% pos − A Nx2 matrix of [x,y] descriptor positions in im.
% descRad − "Radius" of descriptors to compute (see below).
% Returns:
% desc − A kxkxN 3−d matrix containing the ith descriptor
% at desc(:,:,i). The per−descriptor dimensions kxk are related to the
% descRad argument as follows k = 1+2∗descRad.
```

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

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

```
function [pos,desc] = findFeatures(pyr)
% FINDFEATURES Detect feature points in pyramid and sample their descriptors.
% This function should call the functions spreadOutCorners for getting the keypoints, and
% sampleDescriptor for sampling a descriptor for each keypoint
% Arguments:
% pyr − Gaussian pyramid of a grayscale image having 3 levels.
% Returns:
% pos − An Nx2 matrix of [x,y] feature positions per row found in pyr. These
% coordinates are provided at the pyramid level pyr{1}.
% desc − A kxkxN feature descriptor matrix.
```

This function is responsible for both the feature detection and the descriptor extraction.

3.2 Matching descriptors

After obtaining the descriptor matrices D_i and D_{i+1} (the *desc* matrices returned by `findFeatures`) 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. Therefore $S_{j,k} \equiv D_{i,j} \cdot D_{i+1,k}$ 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

- $S_{j,k} \geq 2_{nd}max\{S_{j,l} \ | l \in \{1..f_1\}\}$ 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 $i+1$.
- $S_{j,k} \geq 2_{nd}max\{S_{l,k} \ | l \in \{1..f_0\}\}$ 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 `minScore` 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 `minScore` parameter until you find that you're obtaining good matches. A good value to start with is `minScore=0.5`. In general reasonable values for `minScore` would be in the range $[0, 1)$. The function `matchFeatures` performing this matching procedure should be implemented with the following interface

```
function [ind1,ind2] = matchFeatures(desc1,desc2,minScore)
% MATCHFEATURES Match feature descriptors in desc1 and desc2.
% Arguments:
% desc1 − A kxkxn1 feature descriptor matrix.
% desc2 − A kxkxn2 feature descriptor matrix.
% minScore − Minimal match score between two descriptors required to be
% regarded as matching.
% Returns:
% ind1,ind2 − These are m−entry arrays of match indices in desc1 and desc2.
%
% Note:
% 1. The descriptors of the ith match are desc1(ind1(i)) and desc2(ind2(i)).
% 2. The number of feature descriptors n1 generally differs from n2
% 3. ind1 and ind2 have the same length.
```

3.3 Registering the transformation

We will now use the feature point matches found above to compute the most fitting homography that transforms between the two frames I_i and 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 `applyHomography` function you are required to implement

```
function pos2 = applyHomography(pos1,H12)
% APPLYHOMOGRAPHY Transform coordinates pos1 to pos2 using homography H12.
% Arguments:
% pos1 - An nx2 matrix of [x,y] point coordinates per row.
% H12 - A 3x3 homography matrix.
% Returns:
% pos2 - An nx2 matrix of [x,y] point coordinates per row 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 $H_{1,2}$, `applyHomography` should transform these points 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 `applyHomography` 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 `leastSquaresHomography` 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_1' (using the above `applyHomography`) and compute the squared euclidean distance $E_j \equiv ||P'_{1,j} - P_{2,j}||^2$ for $j = 1..N$. Mark all matches having $E_j <$ `inlierTol` as inlier matches and the rest as outlier matches for some constant threshold `inlierTol`.

RANSAC performs several iterations (later denoted `numIters`) 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 `leastSquaresHomography` on these inlying 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

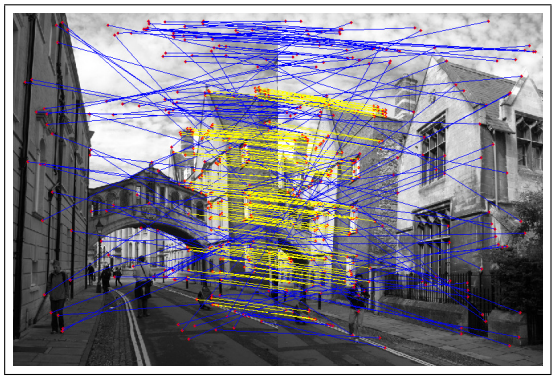
```
function [H12,inliers] = ransacHomography(pos1,pos2,numIters,inlierTol)
% RANSACHOMOGRAPHY Fit homography to maximal inliers given point matches
% using the RANSAC algorithm.
% Arguments:
% pos1,pos2 - Two nx2 matrices containing n rows of [x,y] coordinates of
% matched points.
% numIters - Number of RANSAC iterations to perform.
% inlierTol - inlier tolerance threshold.
% Returns:
% H12 - A 3x3 normalized homography matrix.
% inliers - A kx1 vector where k 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 inlying matches detected by RANSAC implement the following display function

```
function displayMatches(im1,im2,pos1,pos2,inliers)
% DISPLAYMATCHES Display matched pt. pairs overlayed on given image pair.
% Arguments:
% im1,im2 - two grayscale images
% pos1,pos2 - nx2 matrices containing n rows of [x,y] coordinates of matched
% points in im1 and im2 (i.e. the i'th match's coordinate is
% pos1(i,:) in im1 and and pos2(i,:) in im2).
% inliers - A kx1 vector of inlier matches (e.g. see output of
% ransacHomography.m)
```

This function should display a horizontally concatenated image `[im1,im2]` 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 shifted `pos2(j,:)`. Similarly inlier matches are denoted by plotting a yellow line between the match points (see section 8

for an explanation of how you can use the functions `imshow` and `plot` to do this). As an example of how the output should look like, below is the figure resulting from running `displayMatches` on an image pair in one of the provided example sequences together with its match points `pos1`, `pos2` (obtained from `findFeatures` + `matchFeatures`) and inlier index set `inlind` (obtained from `ransacHomography`).



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.

4 Panorama Stitching

Our efforts in the previous section eventually provided us with the set of registered homographies $H_{i,i+1}$, for $i = 1..M-1$, 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 = ceil(M/2)`. What we mean by this is that the resulting panorama image will be composed of frame I_m with all the other frames backwarped 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 renormalize 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 $H_{a,c} \equiv H_{b,c}H_{a,b}$ that now transforms from coordinate system a directly to c.

We may now use this property of homographies to obtain $\bar{H}_{i,m}$ from $H_{i,i+1}$. 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 indentity matrix $I =$ `eye(3)`

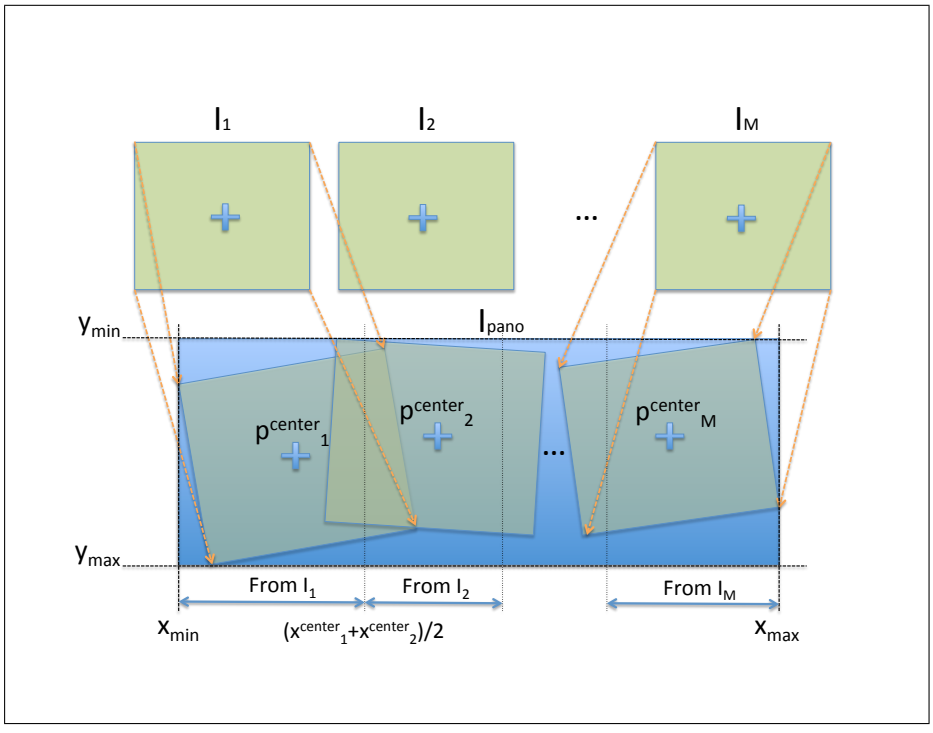
This procedure should be implemented in the following function

```
function Htot = accumulateHomographies(Hpair,m)
% ACCUMULATEHOMOGRAPHY Accumulate homography matrix sequence.
% Arguments:
% Hpair − Cell array of M−1 3x3 homography matrices where Hpair{i} is a
% homography that transforms between coordinate systems i and i+1.
% m − Index of coordinate system we would like to accumulate the
% given homographies towards (see details below).
% Returns:
% Htot − Cell array of M 3x3 homography matrices where Htot{i} transforms
% coordinate system i to the coordinate system having the index m.
% Note:
% In this exercise homography matrices should always maintain
% the property that H(3,3)==1. This should be done by normalizing them as
% follows before using them to perform transformations H = H/H(3,3).
```

In this function interface the cell array of homographies `Hpair` corresponds in the above discussion to the set of homographies $H_{i,i+1}$ and the returned `Htot` 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 describe 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 $i = 1..M-1$, 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



Backwarping of the strips should then be performed as follows. Initially, 3 matrices having dimensions $(y_{max} - y_{min} + 1) \times (x_{max} - x_{min} + 1)$ should be prepared. One is allocated for the I_{pano} image intensities and the other two denoted X_{pano}, Y_{pano} should be generated using the function `meshgrid` to hold the

x and y coordinates of each of the panorama pixels. Recall that these should be in the two ranges $[x_{min}, x_{max}]$ and $[y_{min}, y_{max}]$. Now, for each strip $i = 1..M$ the elements in X_{pano} and Y_{pano} within this strip, denoted $X^{strip-i}$ and $Y^{strip-i}$ should be transformed by $\bar{H}_{i,m}^{-1}$ using `applyHomography` *back* to the coordinate system of frame i. We call these $X'^{strip-i}$ and $Y'^{strip-i}$. These backwarped coordinates can now be used when calling the function `interp2` to perform the backwarp operation from frame I_i into strip i of the panorama frame.

4.3 Stiching

You should use your `pyramidBlending` function from ex3, for blending each new strip with the current panorama image. We want you to think how you can do that that given the images strips from the input images. You may choose not to do that - 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

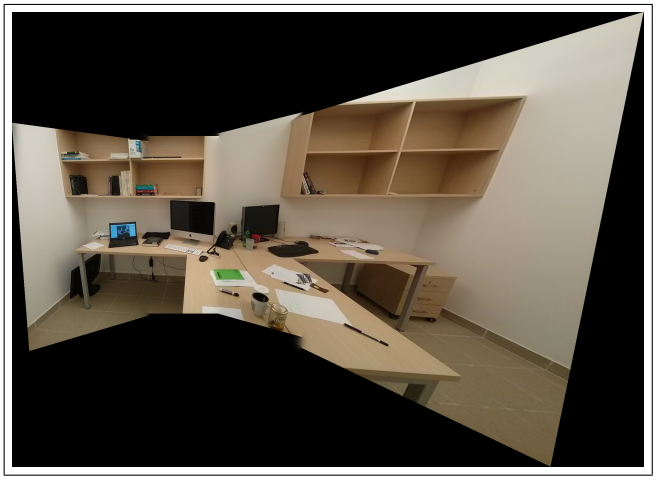
```
function panorama = renderPanorama(im,H)
% RENDERPANORAMA Renders a set of images into a combined panorama image.
% Arguments:
% im − Cell array of n grayscale images.
% H − Cell array of n 3x3 homography matrices transforming the ith image
% coordinates to the panorama image coordinates.
% Returns:
% panorama − A grayscale panorama image composed of n vertical strips that
% were backwarped each from the relevant frame im{i} using homography H{i}.
```

5 Tying it all Together

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

- Read grayscale frames (using your `imReadAndConvert.m`).
- Find features and compute their descriptors (using your `gaussianPyramid.m`, `findFeatures.m`).
- Match feature points (using your `matchFeatures.m`).
- Register homography pairs (using your `ransacHomography.m`).
- Display inlier and outlier matches for the result of `ransacHomography` function (using your displayMatches.m).

(a) oxford



(b) office

(c) backyard

- Transform the homographies (using your accumulateHomographies.m).
- Load the RGB frames.
- Render panorama of each color channel (using your renderPanorama.m).

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

You are allowed to alter `generatePanorama.m` as long as it still functions as described in this defini-

tion to generate panoramas. More specifically `generatePanorama.m` contains some predefined values for the constants `maxNumFeatures`, `ransacNumIters`, `ransacInlierTol` and `minMatchScore` which you are encouraged to change to suite your needs.

6 Your Panoramas

You should submit with your exercise two panorama image sequences having the same naming convention as the example sequences. Place these image sequences in the `ex4/data/inp/mine` directory. Also submit a script called `myPanorama.m` which operates in a similar manner to `examplePanoramas.m` and that produces panoramas out of your image sequences. These two resulting panoramas should be saved into the `ex4/data/out/mine` directory. Make sure that your images are taken with rotation only around the camera center (as much as possible). You can look at Tirgul 6, slide 7 at the example in the top-right corner. And make sure that the images are not bigger than 600x600 pixels.

7 Bonus Questions - 5 points

Change the way the different strips are combined in the `renderPanorama` function by performing dynamic-programming stitching as described in class. Describe your solution in the README file. To test yourself make sure that your method works well with the 'parallax' input images, where there is also translation in the camera between images (so without a special stitching, there will be artifacts in the resulted panorama).

8 Tips & Guidelines

- Start early. This a the large 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 15,20,25 seconds respectively, using the CS-lab computers (even if you implemented the bonus part).
- Try to debug each function you implement first separately from the whole program.
- You can add functions of your own either as sub-functions in existing m-files or as additional m-files. Don't forget to include these additional m-files in your submission, to document these functions properly, and to mention their purpose in your README file.
- Your submission file should also be packed in an archive called ex4.zip.

- After rendering an image in a Matlab figure using e.g. `imshow` you can use `plot` to overlay points and lines without clearing the figure by executing the `hold on` directive before performing the `plot` commands. When you're done plotting the overlay execute `hold off`.
- To use the function `plot` to draw a set of K red dots at the positions defined by the K-element arrays `x` and `x`, execute `plot(x,y,'r.')`. Here the character `'r'` tells plot to use the color red and `'.'` tells it to plot dots.
- To use the function `plot` to draw a single blue line between the points (`x(1)`,`y(1)`) and (`x(2)`,`y(2)`), execute `plot(x,y,'b-')` using the 2-element arrays `x` and `y`.

Good luck and enjoy!