CGRA352 Assignment 4

Matching, warping and applications

30 marks

In this project you will work with image local feature matching, image warping and its typical applications: image alignment and video stabilization. The input data is a sequence of images from one video shot. You will write functions that take the images as input, process them to find the matched local feature points, estimate the transformation between image pairs by RANSAC, and warp the images to align the content. For the completion and challenge, you will implement a video stabilizer to generate stable visual content for video sequence.

Core (13 marks)

Feature matching and homography transformation estimation

The input images are frames of a video. The file names are in the following format:

`Frame<frame_number>.jpg`

Where **`frame_number`** is the original frame number in the video. When loading the input you can either parse in these files programmatically or hardcode them in.

1. (5 marks) SIFT feature extraction and matching

Extract the SIFT feature points on the images **Frame039.jpg** and **Frame041.jpg**. Then find the matching pairs of feature points on the two images, and draw a line between all the matched pairs across the two images (from **Frame039.jpg** on top, to **Frame041.jpg** on the bottom). You are allowed to use **ALL** of the high-level API provided by the OpenCV C++ package for SIFT feature points. In the OpenCV packages with latest versions (>=OpenCV 4.4.0), the SIFT points need to be detected using the following way, where the feature points will be stored in “keypoints_1”:

```
(Make sure you have #include <opencv2/features2d.hpp>)
Ptr<cv::SIFT> sift = sift->create();
std::vector<KeyPoint> keypoints_1;
Mat descriptors_1;
sift->detect(img_1, keypoints_1); //
```

**It is NOT recommended to use other ways to call SIFT functions if you are using OpenCV’s official package. Other ways are not stable during the test.

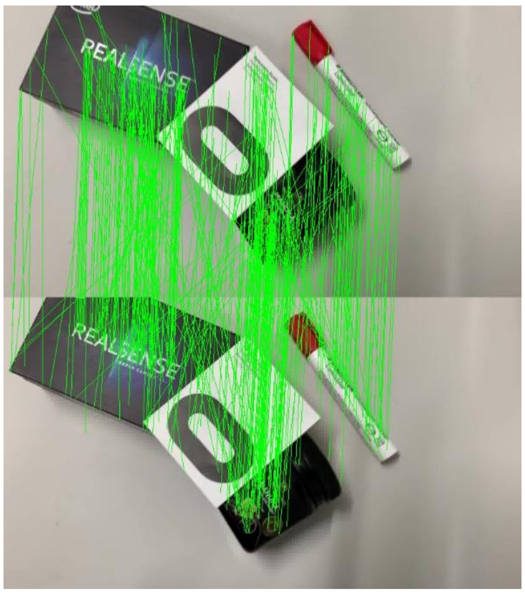
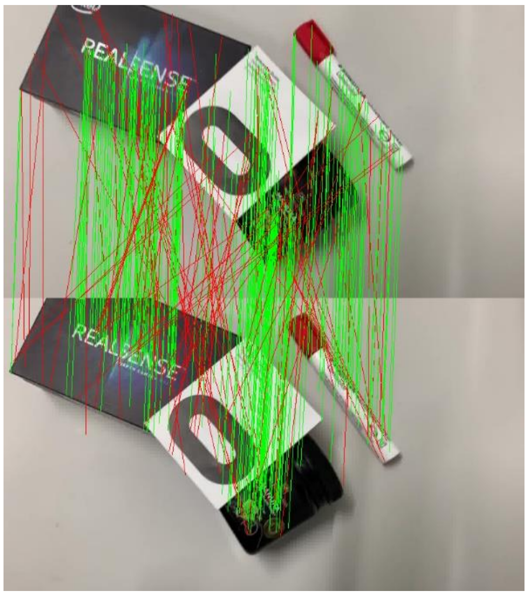
**The above functions DON’T work with OpenCV 3. Make sure you are using an OpenCV version >=4.4

Other useful functions to do the matching include:

```
cv::BFMatcher
std::vector<cv::DMatch>
```

Note : setting the second argument (cross-check) of `BFMatcher` to true will ensure that no feature is used more than once for all the pairs produced. This will reduce the error when calculating a homography matrix and produce a better result.

Your submission should include a single image with both input images vertically stacked and with green lines drawn between the matched feature point pairs.



Example output for SIFT algorithm. (left) Part 1 of Core, where all pairs are drawn in green with **Frame039.jpg** *on top and* **Frame041.jpg** *on the bottom. (right) Part 2 of Core, where all inlier pairs are drawn in green and outlier pairs are drawn in red.*

2. (5 marks) Estimate homography transformation

Implement a RANSAC-based homography transformation matrix estimation function for estimating the best homography transformation between the given image pair (from **Frame039.jpg** to **Frame041.jpg**). Implement the following steps:

1) Generate feature pairs using SIFT
2) Iterate a fixed number of times, e.g. 100:
   a) Select four feature pairs (at random)
   b) Compute the homography transform H for the pairs **exactly**.
   c) Compute *inliers* amongst all of the pairs, where the mapping error of the transformed point q with the target position p is less than some epsilon. This can be done as ($|p_i - Hq_i| < \varepsilon$).
   d) If the number of inlier pairs is greater than previous iteration's, save the current homography transform H and the number of inlier pairs.
3) Re-compute homography transform H estimation on the largest set of inlier pairs

To estimate a homography matrix H between 4 given pair of matching feature points in each iteration, and the homography matrix H estimation in the step 3 (where you use more than 4 pairs of matching points in a least square way), you are allowed to use the following function:

```
Mat cv::findHomography(InputArray srcPts, InputArray dstPts, ...)
```

Important – DO NOT USE CV_RANSAC in this function: When using this function you must keep the third optional parameter defaulted as 0. This is because OpenCV implements a RANSAC version of H matrix estimation. You are not permitted to use the built-in RANSAC method for this part of the assignment.

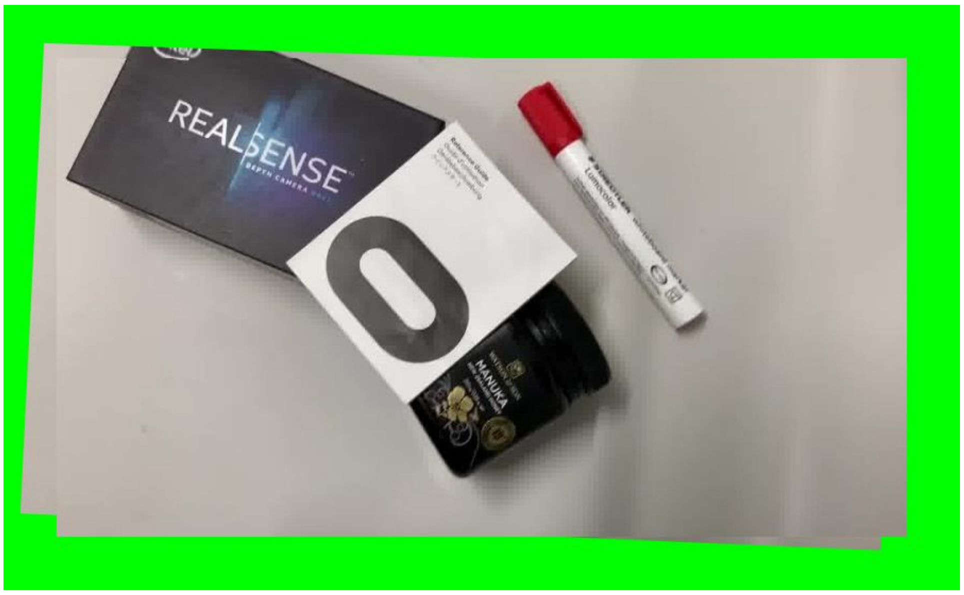
Your submission should include a single image with both input images vertically stacked and with <u>green lines between inlier pairs</u> and <u>red lines between outlier pairs</u>.

3. (3 marks) Generate an image aligned result

Use the function you implemented in Step 2 to estimate the homography transformation between **Frame039.jpg** and **Frame041.jpg**. Then use the homography matrix to warp one of the images to align the content to another image (either way is fine). You may use the warping function provided by OpenCV to warp an image:

```
cv::warpPerspective(InputArray Origin_mage, OutputArray
warped_image, Mat H, size cv::Size(cols, rows))
```

Your submission should include a single image that contains the aligned content from both images with the invalid pixels by a solid color (typically black or white, but any color is acceptable). You could just choose pixel value from one of the input images in the overlapped aligned content.



A possible arrangement of overlaying two frames with green marking the invalid regions.

Completion (10 marks)

A video stabilizer

Implement a video stabilization method. Use the homography transformation between the neighboring frames as the proxy of the actual "path" of the camera. In a stable video, the H matrices between neighboring frames should change smoothly, so if we can make the homography transformation between neighboring frames more similar, we will have a stabilized video.

Because the transformation parameters are continuous with respect to the content changes, it means if we want to get a smooth "path" of the visual content, we have to smooth the parameter sequence. For example, suppose the *Hi* matrix is the homography transformation between frame *I*i and frame *I*i-1, it could be written as:

$$\mathrm{H_i} = \begin{bmatrix} h_{11}^{i} & h_{12}^{i} & h_{13}^{i} \\ h_{21}^{i} & h_{22}^{i} & h_{23}^{i} \\ h_{31}^{i} & h_{32}^{i} & h_{33}^{i} \end{bmatrix}$$

$$I_{i-1} = H_i I_i$$

The cumulative transformations from the first frame to the frame *i* could be represented by:

$$I_0 = H_1 \dots H_{i-1} H_i I_i = \widetilde{H_i} I_i$$

We denote $\widetilde{H_i}$ as:

$$\widetilde{H_i} = \begin{bmatrix} \tilde{h}_{11}^{i} & \tilde{h}_{12}^{i} & \tilde{h}_{13}^{i} \\ \tilde{h}_{21}^{i} & \tilde{h}_{22}^{i} & \tilde{h}_{23}^{i} \\ \tilde{h}_{31}^{i} & \tilde{h}_{32}^{i} & \tilde{h}_{33}^{i} \end{bmatrix}$$

This matrix $\widetilde{H_i}$ is a transform from frame *i* to the first frame, so we can take it as a measurement of the **actual motion of the visual content**. Every entry in the $\widetilde{H_i}$ matrix is representing a freedom of the transformation; take translation as an example, if $\tilde{h}_{13}^{i}$ and $\tilde{h}_{13}^{i-1}$ is very different, it means for frame *i* and frame *i-1*, the horizontal moving speed has a significant change, meaning that it is shaking. If we can reduce the difference between the parameters at the same position in the neighboring $\widetilde{H}$ matrices, the content will change more slowly.

To smooth the sequences of values $\{\tilde{h}_{mn}^{1}, \tilde{h}_{mn}^{2}, \dots, \tilde{h}_{mn}^{i}\}$ in $\widetilde{H}$ matrices we can use a 1-D Gaussian filter or median filter. Alternatively, all 9 sequence values in the matrices can be smoothed together by a weighted sum of the neighboring matrices. For example, for a Gaussian filter with a window size of 5, the new cumulative motion matrix can be:

$$\widetilde{H_i}' = \frac{\boldsymbol{w_{-2}}\widetilde{H}_{i-2} + \boldsymbol{w_{-1}}\widetilde{H}_{i-1} + \boldsymbol{w_0}\widetilde{H}_i + \boldsymbol{w_1}\widetilde{H}_{i+1} + \boldsymbol{w_2}\widetilde{H}_{i+2}}{\boldsymbol{w_{-2}} + \boldsymbol{w_{-1}} + \boldsymbol{w_0} + \boldsymbol{w_1} + \boldsymbol{w_2}}$$

Where $\{\boldsymbol{w_{-2}}, \boldsymbol{w_{-1}}, \boldsymbol{w_0}, \boldsymbol{w_1}, \boldsymbol{w_2}\} = \{0.1, 0.3, 0.5, 0.3, 0.1\}$. You may need to choose a larger window size for the Gaussian filter for a more stable result. You can also perform the Gaussian filter several times on the cumulative homography transformation matrix sequence to make it more stable.

The motion $\widetilde{H_i}'$ is a smoothed motion that transforms a stabilized image I_i' in to the first frame, so we need a homography transformation U_i that transforms the current image I_i into the same space as I_i'. This can be calculated as:

$$\widetilde{H_i} I_i = \widetilde{H_i}' I_i'$$
$$U_i I_i = I_i'$$
$$U_i = \widetilde{H_i}'^{-1} \widetilde{H_i}$$

You can then use $U_i I_i$ to warp the image sequence to get the following stabilized visual content:

Stable045.png

Stable046.png

Stable047.png

Stable048.png

Stable049.png

Stable050.png

Stable051.png

Stable052.png

Stable053.png

Stable054.png

Stable055.png

Stable056.png

An example of the possible output for the stabilized frames. The change is subtle but there should be clear regions in each frame that show the image has been shifted (shown here in green).

Note: You may find it helpful to start with a small number of frames first (maybe 10) and use software to playback your output as a video. We recommend DJV: http://djv.sourceforge.net/

Your submission should include all 102 frames of stabilized video named:

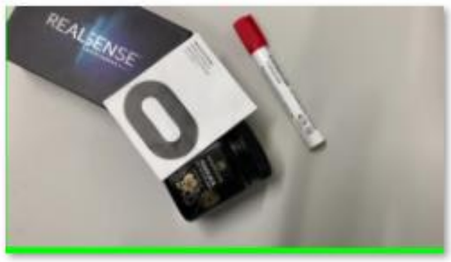
`Stable<frameNumber>.png`

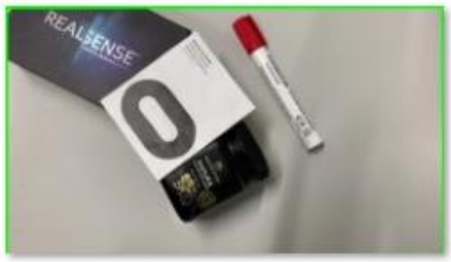
where `<frameNumber>` is the number of the frame in the sequence starting with "000".

Challenge (5 marks)

Best cropping window

The images after updating transformations will look like the following:





To generate a final stabilized result you need to find the best cropping window with the same aspect ratio as the original frames (and with the same window performed on every frame) to cut off all the invalid pixels (green in the above examples) for the whole video.

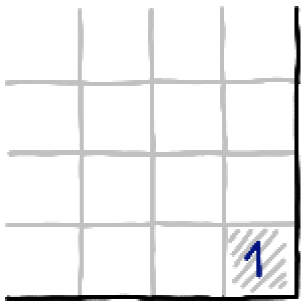
First, to implement this cropping algorithm, you have to store the masks of the valid regions of all the transformed images. This could be done by transforming a fully white mask using the transformation U_i for every *i*, then in the transformed mask, the white pixels will all be valid and anything else (black pixels) will all be invalid. The next step is to generate a new mask M, whose valid pixels are also valid pixels in every transformed mask. This can be done naively by iterating through every image and setting the pixel to valid if and only if every image is valid at that same pixel.

The problem becomes to find the largest inscribed rectangular region in this new mask M. Finding the largest inscribed rectangle is harder than the largest inscribed square because the rectangle has two degrees of freedom (height, width) compared to a square which just has one (size). So first we resize all the masks to a square, find the largest inscribed square, and then resize it back, which results in a ratio-preserved rectangular cropping window.

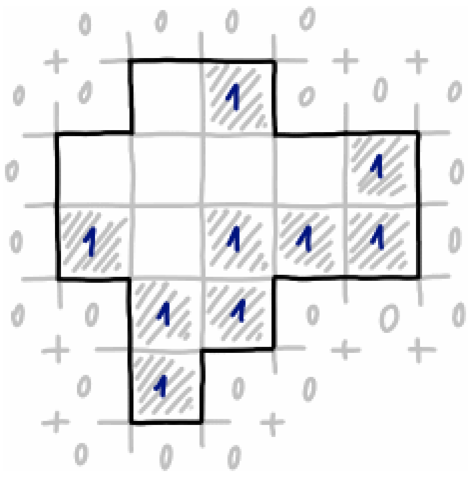
Largest Inscribed Square

A possible straightforward algorithm for finding largest inscribed square is to iterate over mask pixels and for every white pixel, increase square sizes from 1 until a black pixel is encountered. This algorithm works but is very slow. Consider a white mask of size N by N pixels. Such mask will need to check N^3 squares leading to a cubic time complexity in image size. Most checks are unnecessary as for each large square checked, all its sub-squares are checked as well, which obviously cannot be larger than the original square.

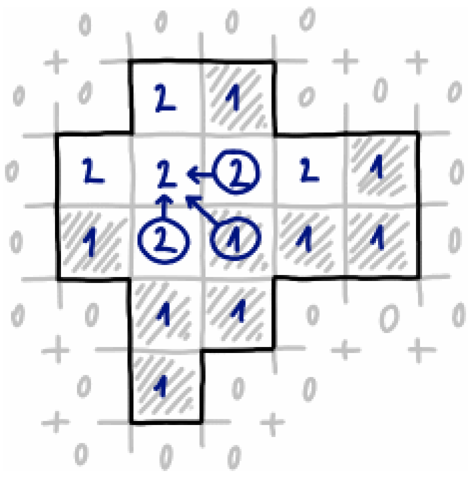
The problem consists of smaller sub-problems and can be solved conveniently by first finding smaller squares and then combining them into larger ones, re-using already known values, in a technique called dynamic programming. Instead of scanning the image from the top-left to the bottom-right corner, we scan it from the bottom-right to the top-left corner. If the bottom-right pixel is white, we know for sure that the largest square having top left corner at the same position is 1 pixel large:



The square-searching algorithm fills a two-dimensional array S to contain the largest inscribed square size whose top-left corner is at that pixel position. The array S is of the same size as the image since its entries correspond to image pixels. The array is progressively filled with positive integers as the algorithm scans through the image. For any black pixel, **S[x,y]** is **0** because there can be no valid square with its top left corner black. **S[x,y]** is **1** for any white pixel at position (x, y) that has a black pixel neighbor at position (x+1, y), (x, y+1) or (x+1, y+1), because that black pixel neighbor means that the largest valid square at that position can be only 1x1 pixels:



From here you can compute the size of the largest square at given position when we know square sizes at the three neighboring positions (x+1, y), (x, y+1) or (x+1, y+1):



For example, the 1×1 square at the position to which the three arrows point can stretch two pixels to the right, two pixels to the bottom and just one pixel diagonally. Since width and height of the square must be equal, the largest square size is restricted by the smallest neighbor:

$$S[x,y] = \min(S[x+1,y], S[x,y+1], S[x+1,y+1]) + 1$$

This formula only applies when scanning the image in a bottom-up, right-left fashion, which ensures all three needed neighbors are always known.

Your submission should include all 59 frames of cropped stabilized video named:

`Crop<frameNumber>.png`

where `<frameNumber>` is the number of the frame in the sequence starting with "00".

Report (2 marks)

You should submit a 1-2 page PDF document reflecting your experience. In this report, you should include:

- Brief introduction of your functions in your programs.
- How to run your program to perform the functions required by the assignment.
- The results of core, completion and challenge (depending on how much you have done).

Practical matters

You are required to submit both your program and any supplementary material including the report. You may complete the assignment in a single or multiple programs, as long as it meets the assignment specification. Please zip your program(s) source and other material into a single compressed file to upload. You are required to show your program can run on the ECS machine OR your own computer.