[ELE00142M] Machine Vision and Human Machine Interaction

Lab 2: Localization and Tracking of Dynamic Objects

Part 3: Homographies

Lecturer: Mark A Post <mark.post@york.ac.uk>

Demonstrator: John Bateman <john.bateman@york.ac.uk>

Technician: Alejandro Pascual San Roman <alejandro.pascualsanroman@york.ac.uk>

Aims and Objectives

In this lab session you will be experimenting with the Open Source Computer Vision Library (OpenCV) using a USB camera on a laboratory desktop workstation. The focus of this lab will be the use of a calibrated camera to localize and track features in 3-D space.

It is expected that you will complete all of these activities in three weeks of labs (Week 4 - Week 6) completing one part (in one document) of the lab per week. The P/T/410 and P/T/411 laboratories are open for use outside of scheduled lab times in case you need extra time to finish. You can also work at home by installing the Python language and OpenCV-Python on your own computer.

Learning outcomes

- Repeated image capture, processing and saving in OpenCV
- Understanding of camera coordinate systems and calibration
- Calibration of a camera using a calibration pattern
- Undistortion of images given camera parameters
- Disparity mapping using stereo block matching

- Visual Odometry using estimation of Homographies
- Reconstruction of Panoramas from Multiple Images

Hardware and Software required

- Python 3 with OpenCV-Python and numpy libraries installed
- Logitech C930 or similar USB camera with autofocus
- Calibration image files *pattern_chessboard.png* and *pattern_acircles.png* provided on the module webpage. They are also provided at the end of this document.

Tasks

10. Reconstruction through Homography

If you have completed all the tasks given in the previous parts of this lab, you should now copy your code into a new text file to make a Python program that performs the following operations (it should be exactly in this order if you have followed the steps carefully):

1. Read in a camera calibration (intrinsic matrix and distortion coefficients) that you have previously obtained by calibrating your camera.
2. Capture images one by one from a camera and store them in a list.
3. Undistort each image using the intrinsic matrix and distortion coefficients.
4. Detect keypoints in each undistorted image and store them in a list.
5. Extract the descriptors for each undistorted image and store them in a list.
6. After two or more images have been captured, take the previous two images and match their descriptors to obtain a list of matches.
7. (Calculating a disparity image is optional at this point, you can remove any Stereo Block Matcher code that you have remaining in your new program.)
8. Sort the matches in descending order with the best match first.
9. Find the Fundamental and Essential matrix from the matched keypoints and recover the pose from the Essential matrix.

If you don’t have all this working, take some time to go back to previous parts of the lab. This is a fairly complex program when you delve into what each of the

OpenCV functions does, but is reduced into a relatively concise set of function calls and encapsulates the most well established methodologies for sensing and reconstructing three-dimensional environments from machine vision hardware.

We will now focus on the final step: reconstructing the environment from all of the visual information that you have gathered. The process of “stitching” together a “panorama” from multiple images is probably familiar to you, as it is now commonly available on most mobile devices, digital cameras, and a variety of free online software such as Hugin. Image stitching makes use of the concept of a *Homography* introduced previously, which is a linear transformation between two image planes. The Essential Matrix and Fundamental Matrix are both special kinds of Homographies as they refer to camera-calibrated and uncalibrated mappings of image points from 3 dimensional space. As we are only interested in directly mapping image-space pixel coordinates for the process of stitching, we can use a general Homography matrix without calibration (assuming you have already undistorted your images) to relate the two image planes from two different camera positions as shown in Figure 1. The goal of image stitching is to produce a single “panoramic” image plane composed of multiple smaller images taken at different angles.

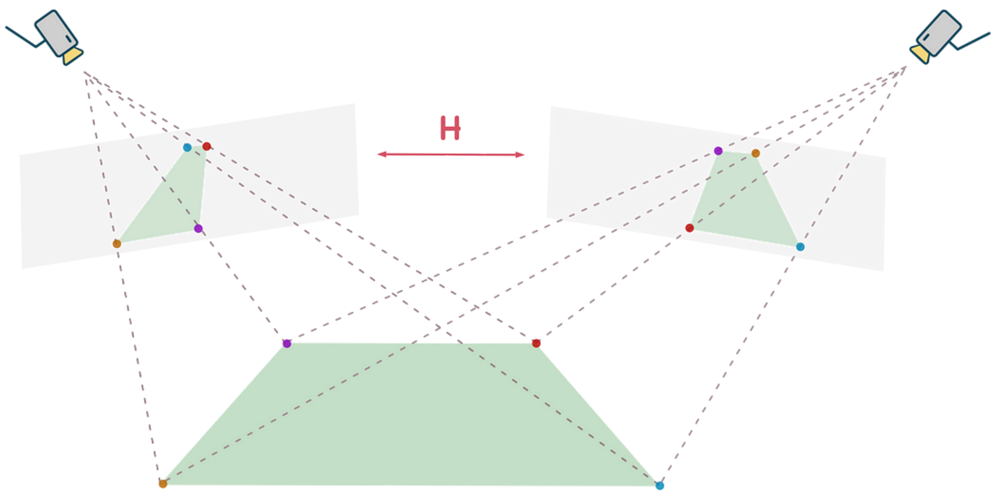

Figure 1: A Homography between two images

This general Homography matrix *H* is a *3x3* transformation matrix that, when multiplied by the (*homogeneous*) pixel coordinates in one image, gives the pixel coordinates of the other image, effectively “mapping” one image’s features on to another’s image plane. Mapping one plane on to another means that geometric linear transformations of the image must be used such as translation, rotation, scaling, aspect, affine, and perspective transformations (effectively “stretching” and “warping” the first image plane until it matches the other image plane). The Homography matrix is a single compound matrix that performs these transformations all at once (much as the Intrinsic and Extrinsic matrices compound together the transformations that occur when light enters your camera).

First, initialize a new window to contain the stitched image, with `cv2.namedWindow("Stitched Image")` or a similar name. Remember to `add a cv2.destroyWindow("Stitched Image")` to the end of your program to clean it up also.

If you have not done so already, make sure that you have unpacked the matches from your choice of descriptor matcher into their respective keypoint locations so that the Fundamental Matrix can be calculated.

```
pts1 = []
pts2 = []
for m in matches:
pts2.append(kp2[m.trainIdx].pt)
pts1.append(kp1[m.queryIdx].pt)
```

You will also need to convert them to *int32* format

```
pts2 = np.int32(pts2)
pts1 = np.int32(pts1)
```

You can obtain the Homography matrix from *int32* format matched keypoints of two images in the same manner as the Fundamental and Essential matrices, using the cv2.findHomography() function as

```
H, mask = cv2.findHomography(pts1, pts2, cv2.RANSAC, 5.0)
```

Note that it is important to get the Homography *from* image 1 to image 2 as we will be using image 2 (the first captured image, if you have defined it that way) as the target image plane, so *pts2* from *img2* must be the second (destination) argument.

In this example we are directing OpenCV to use the *cv2.RANSAC* (RANdom Sample Consensus) solution implementation, which is faster and more robust to outliers. The slower but more meticulous *cv2.LMEDS* implementation is also available. If you are finding that your homographies are not performing well you may want to try both of these options to see which works better.

Insert this in your program after the *cv2.findFundamentalMat()* and *cv2.findEssentialMat()* calls. Print out the *H* matrix and compare it to what the *F* and *E* matrices look like for a set of matched images. What are the similarities and differences that you notice?

11. Warping the Image

Once you have found the Homography matrix *H*, you still need to apply it in such a way as to map the two images on to a common image plane so it appears they are part of the same image. For simplicity, we will assume that the first image uses the "target" image plane and map subsequent images on to that same coordinate system. The *cv2.warpPerspective()* function in OpenCV takes as arguments the Homography matrix *H*, the size of the target image (obtainable using the *shape* 2D tuple attribute of a numpy.ndarray object), and the input image to be *warped* on to the target image plane. By using the detected Homography, the size of the resulting image will also be scaled to match the dimensions of the target image because the matched features indicate this scaling. Use it with

```
warped = cv2.warpPerspective(img1, H, (img2.shape[1], img2.shape[0]))
```

Now you can display the warped image effectively in the plane of the target image by superimposing the two images. Adding two images together (of the same size as defined by the *shape()* attributes) is easy in OpenCV as you can just use the overloaded plus '+' operator. Display the images together with

```
cv2.imshow("Stitched Image", img2+warped)
cv2.waitKey(0)
```

Try stitching a few pairs of images together. You may find that sometimes the Homography results in a terrible match or distorted images, and this is usually caused by bad feature matches between the images. Remember that for good image stitching you need to have images that are:

1. similar enough that many correct matches are found between them to obtain a Homography (you may want to use the Chessboard paper or other complex patterns to test with that contain many regular features);
2. overlapping sufficiently such that they do not need to be extremely warped or distorted to align on a common image plane (i.e. not a very large angle or linear distance between camera positions).

Contrary to the case with Stereo Vision though, notice that because there is no assumption of a common baseline and common axis for stereo vision, you *can* rotate the camera around its axis and the Homography will still be able to align the images with each other.

Optional Task: Filter your Matches

If you are having difficulty getting good matches, consider implementing a K-nearest Neighbors matcher and using Lowe's ratio test for descriptors to filter your matches and only keep those that are within a certain ratio of each other. If you want to try this out, you can replace your matching code with code similar to the following (typical values for Lowe's ratio are usually in the range *[0.7, 0.8]*):

```
ratio = 0.7
matches = []
bf = cv2.BFMatcher()
knnMatches = bf.knnMatch(des1, des2, k=2)
for m,n in knnMatches:
    if m.distance < n.distance * ratio:
        matches.append(m)
```

12. Blending Images Together

If you have succeeded in stitching two images together accurately, you may notice that just adding them together directly results in a pretty bad looking image. This is because adding pixel values directly will often cause the components to overflow their bit length limitations leading to some pretty strange looking colours on high-intensity pixels.

You can improve the quality of the image by *blending* the images together instead of just adding pixel values, which determines each pixel value through a *Pixel Operator Function*, such as the dyadic (two-input) Linear Blend Operator.

$$g(x) = (1 - \alpha)\, f_0(x) + \alpha\, f_1(x)$$

This operator takes in two pixel values $f_0(x)$ and $f_1(x)$ and combines them into a target pixel value $g(x)$. By varying the weighting parameter α from $0 \rightarrow 1$ one image can be faded, or "dissolved" into another, which is a common transition effect used in slide shows, pictures, and videos. The common use of the linear parameter α (which is often included as a pixel value "transparency" component alongside *B*, *G*, and *R* components) has led to the well-known name of *"Alpha Blending"*.

In this case as we do not need to favor one image over another, we can use a weighting parameter of $\alpha = 0.5$ and blend the two images together with

```
alpha = 0.5
blended = cv2.addWeighted(warped, alpha, img2, 1 - alpha, 0)
```

Show the resulting blended image with *cv2.imshow()* instead of the added image

```
cv2.imshow("Stitched Image", blended)
cv2.waitKey(0)
```

Try your program on several images of different scenes and objects in the room, and with images from different locations.

- Are there some objects that work better than others for stitching?
- How much rotational difference or translational difference can you allow between camera positions before the Homography does not succeed?
- How many feature points do you need to make a successful Homography?

13. Multiple-Image Panoramas

Your final task in this lab is to take everything you have learned so far, and modify your program to **stitch multiple images into a single panorama!** You will probably want to re-think your program structure a little, have the first image that you take saved as a target image plane and then compare every other image to it before blending them with the target image. This will mean that you will need to take a very wide-viewing target image so as to not restrict other images from different locations from sharing feature points with it. Can you think of other creative ways to stitch a panorama together that do not rely on a single initial image? Can you perhaps use the panorama itself as your target image by searching it for features to compare after adding each image to it, or even take a large list of images and sort through them for good matches before pairing them together in a set of sub-panoramas?...

If you want to place your target image on a larger canvas so that you can add more images to it, you can try the following snippet of code, which pastes a smaller image on to a larger canvas filled with only colour. Here, *y_offset+height <= blank_image.shape[0]* and *x_offset+width <= blank_image.shape[1]*, and it is usually good to set the offset so that the initial target image is in the centre of the target image plane.

```
height, width = img1.shape[:2]
blank_image = np.zeros((3840,2160,3), np.uint8)
blank_image[:,:] = (255,255,255)
l_img = blank_image.copy()
x_offset = 3840/2-1920/2
y_offset = 2160/2-1280/2
l_img[y_offset:y_offset+height, x_offset:x_offset+width] = img1.copy()
```

You then need to re-project the source image on to the target image by obtaining new matches and a new homography such as in

```
H, mask = cv2.findHomography(pts1, l_img, cv2.RANSAC, 5.0)
warped = cv2.warpPerspective(img1, H, (l_img.shape[1], l_img.shape[0]))
```

This is most easily done as a single main loop of your program, in which all new images have feature detection performed, and are then matched with a common target image plane. To make a large panorama you will need to either warp your existing keypoints for new images on to the target image plane, or just re-detect features on the panorama each time a new image is added. You may also want to change the blending parameters

Once you are finished with the activities of this lab, please ask a demonstrator to check your work to receive credit that will count towards your final grade.

OPTIONAL ACTIVITY 3: Finding Objects in a Scene using Homographies

Another application for Homographies and Linear Transformations is to locate a source object (represented by one image, usually as the only object in the image) in a larger target scene, or even a panorama you have created.

You should already have enough information from this lab to be able to figure out how to do it, as it is again just a matter of finding keypoints in both the source object image and the target scene, matching them, and then finding the homography required to map the object on to the scene. To help guide you, OpenCV has a tutorial on using Feature Matching + Homography to find objects. It is worth going through because this example uses SIFT feature points, FLANN matching, and the Lowe’s ratio test. It is useful to try and integrate these different approaches into your Python programs as you will get to see how different algorithms affect the results of your image processing.

For an additional challenge (or perhaps a project idea), once you have matched the object in the scene, try to use Disparity Mapping or the Fundamental Matrix to estimate the 3D location of the object with respect to the camera!

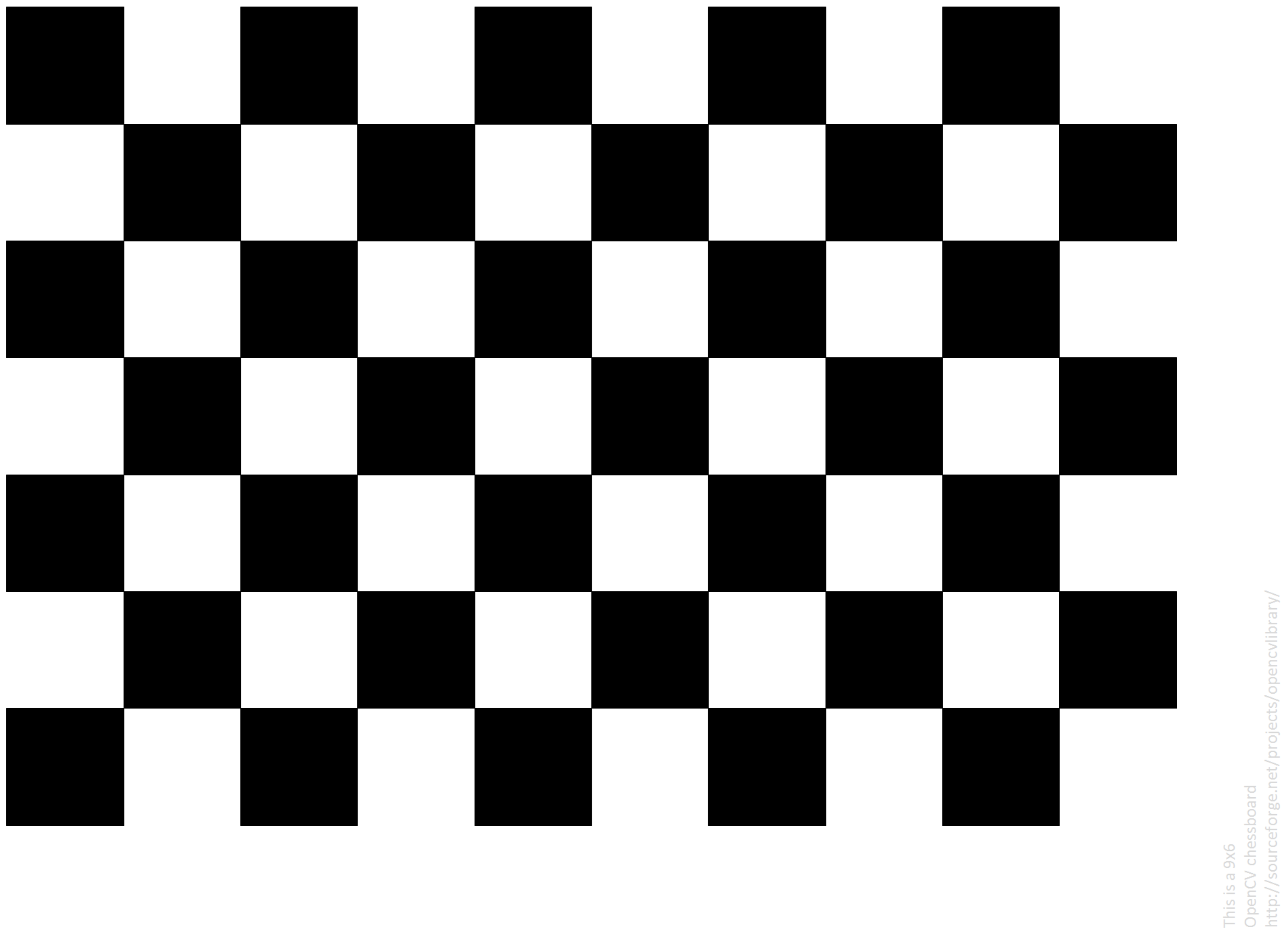

Figure 2: Chessboard Calibration Pattern (pattern_chessboard.png)

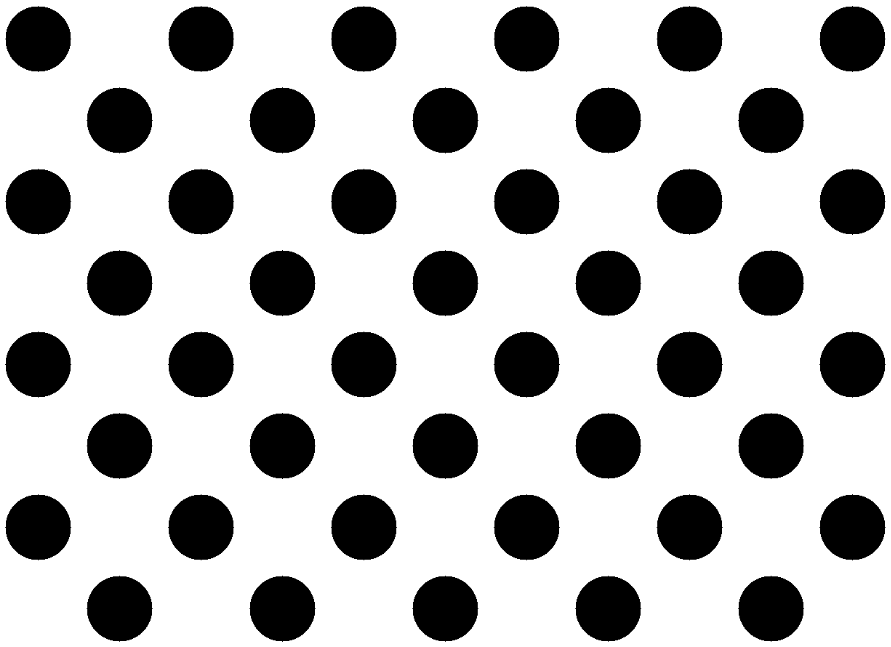

Figure 3: Circles Calibration Pattern (pattern_acircles.png)