[ELE00142M] Machine Vision and Human Machine Interaction

Lab 2: Localization and Tracking of Dynamic Objects

Part 2: Extrinsics

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 two weeks of labs (Week 4 - Week 5). The P/T/410 and P/T/411 laboratories are open for your 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

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

1. Pose Estimation with Reprojection

As you may recall, the *cv2.calibrateCamera()* function returns a vector of rotations *rvecs* and a vector of translations *tvecs* that represent the rotations and translations of the known calibration pattern that you used. These rotations and translations are *Extrinsic* camera parameters much as the intrinsic matrix and distortion coefficients are *Intrinsic* camera parameters. They describe the relative position or motion of the camera with respect to the pattern. And like the other parameters they are solved for with least-squares estimation from the transformation between the physical locations of points in 3D and the resulting pixels they correspond to in the 2D image. When rotation and translation are combined into a single matrix, it is called the *Extrinsic matrix.*

The mapping of feature points in a 3D scene to their corresponding features in a 2D image plane is a pose computation problem that is called the Perspective-n-Point (PnP) problem. The pose computation problem consists in solving for the rotation and translation that minimizes the reprojection error from 3D-2D point correspondences, as shown in Figure 1. The same Intrinsic and Extrinsic matrices as in the camera calibration process are used, with the objective this time to specifically solve for the 3D-2D correspondences and *Extrinsic Matrix* given that the camera parameters and *Intrinsic Matrix* are already known.

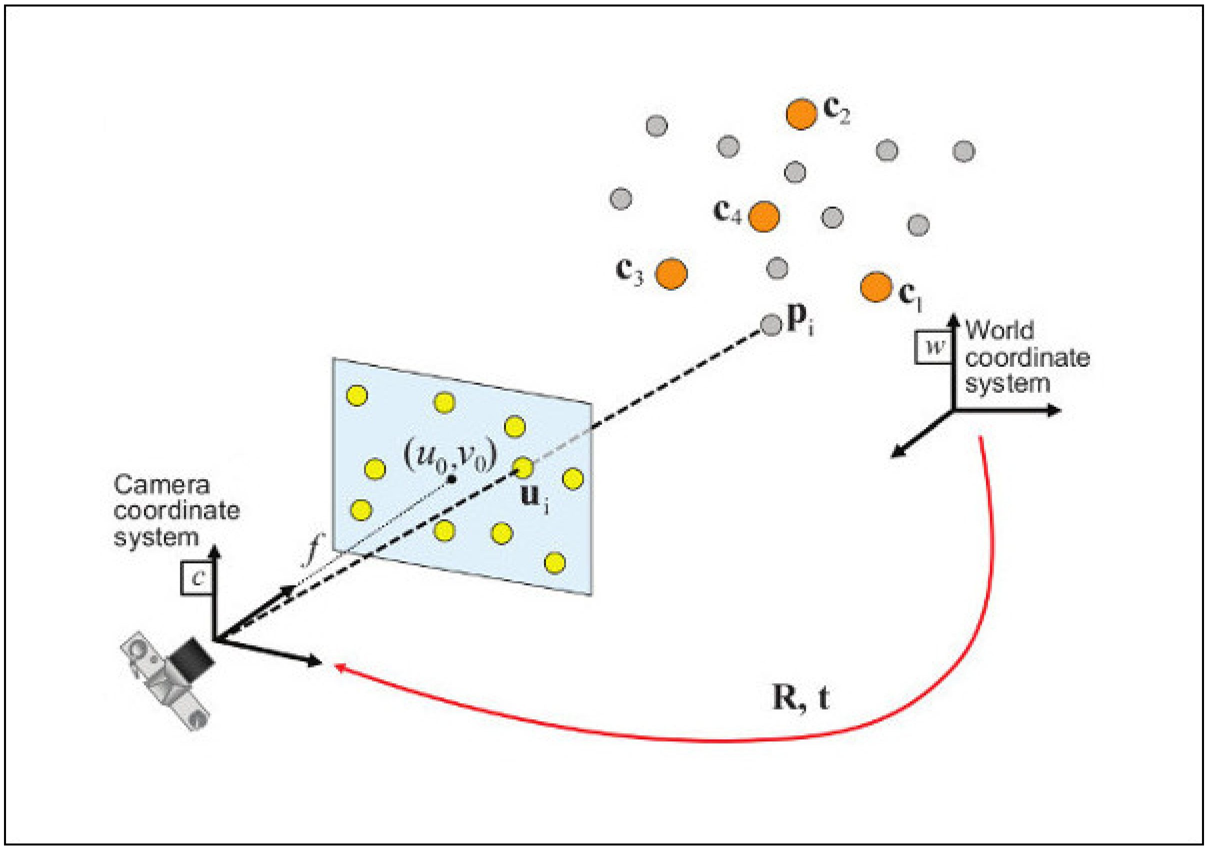

Figure 1: Coordinate Systems of the PnP Problem

There is a way to re-project these points to where they would appear in an image of a 3D scene, and doing this will help you to understand the process of reprojection. Start with your image-capturing code from the last part of the lab. You may want to comment out the lines that save calibration data and images if you want to keep your calibration safe.

First we need something to draw on a localized object. You can draw a simple three-colored Cartesian axis marker given an origin (first element in *originpts* and a set of vectors (three *imgpts*) with this function:

```
def draw(img, originpts, imgpts):
	origin = tuple(originpts[0].ravel().astype(int))
	img = cv2.line(img, origin, tuple(imgpts[0].ravel().astype(int)), (255,0,0), 5)
	img = cv2.line(img, origin, tuple(imgpts[1].ravel().astype(int)), (0,255,0), 5)
	img = cv2.line(img, origin, tuple(imgpts[2].ravel().astype(int)), (0,0,255), 5)
	return img
```

You also need to define an axis in the image on to which to project the points. Such as

```
axis = np.float32([[3,0,0], [0,3,0], [0,0,-3]]).reshape(-1,3)
```

Just enter *axis* into the command line to see what it looks like - just a matrix that defines three orthogonal vectors in a 3D coordinate system. Put both the function and axis vectors near the start of your program, you only need to define them once.

You can solve the PnP problem with the OpenCV function *cv2.solvePnP()* which takes in the camera parameters in *mtx* and *dist* as well as the object points and image points, and can use a variety of solution methods to obtain the Extrinsic parameters *rvecs* and *tvecs*.

```
ret, rvecs, tvecs = cv2.solvePnP(objpoints[imgnum], imgpoints[imgnum], mtx, dist)
```

By solving the PnP problem, this function produces rotation and translation vectors similarly to *cv2.calibrateCamera()*, by finding the relationship between a set of 3D points and their known 2D projections. Knowing these vectors, as well as the camera intrinsics, we can *reproject* any point in our 3D coordinate system into where it would appear in the 2D image. OpenCV provides the *cv2.projectPoints()* function which takes in all of these parameters and reprojects our *axis* in to our *image* as if it were attached to the 3D object that our *rvecs* and *tvecs* have localized, and taking into account intrinsics *mtx* and *dist*.

```
projpoints = cv2.projectPoints(axis, rvecs, tvecs, mtx, dist)
```

You can then draw the axis marker with

```
img = draw(img, imgpoints[imgnum], projpoints)
```

Put these three lines of code after your main program loop (down where you saved images using *cv2.imwrite()* and inside a loop, for example starting with `for imgnum, img in enumerate(imglist):` so that you can loop through all images that you took. Go through the process of calibration again. You should now see axis markers attached to the first chessboard image point, correctly rotated to align with the image! This kind of reprojection is very useful for both Machine Vision debugging and Augmented Reality applications, as we can attach a virtual object to anything in the scene! If you have a known good calibration (and reasonably fast computer), you can even start a video stream, find the chessboard, and re-project the object in real time video.

How accurate is the position and pose of the reprojected axis marker? If it seems to be a little bit off you can try it with replacing *img* in your code with the undistorted image *dst* from previously, which was undistorted using the intrinsic camera parameters (however the difference is usually very small!)

2. Disparity Mapping

Disparity mapping makes use of the differences in the positions of features between two images to estimate their relative distance from the camera. The geometry of disparity mapping using a pair of cameras in a “stereo vision” configuration is shown in Figure 1.

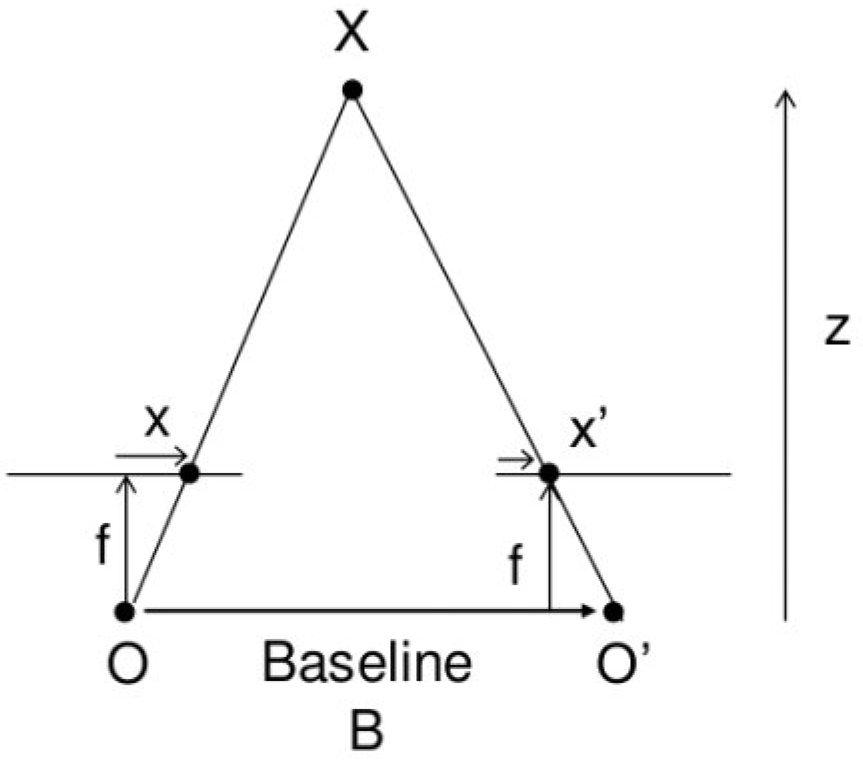

Figure 2: Stereo Vision Disparity Coordinate System

We can use the “disparity” or difference in position of features along the *x* axis connecting the two camera positions on the image plane (*x - x’)* to solve for the axial distance from the focal points of the cameras. From the geometry of similar triangles, we see that with a baseline *B* between the camera positions and a focal length *f* of the cameras

$$disparity = (x - x') = B * f / Z$$

Solving for Z we find that *Z = B * f / (x - x')* which is the characteristic equation of stereo vision.

As you only have one camera, we will "simulate" a stereo pair by taking two images in succession very close to each other. To do this it is recommended that you create a new Python program file, and start with the "Digital Camera" example from the last part of the lab. This time since we are focused on gathering pairs of images, make `number_of_images = 2` so that you only capture images a pair at a time.

So as to not have to calibrate the camera all over again, you should define your camera's intrinsic parameters from what you recorded previously in the last part of the lab. Note that OpenCV uses nested numpy arrays for these dense 2D matrices so you can hand-define them (substituting appropriate numbers in) with

```
mtx = np.array([[fx, 0, cx],[0, fy, cy], [0, 0, 1]])
```

If you saved them as CSV files, you can read them back in using numpy's convenient *np.genfromtxt()* function, e.g.

```
mtx = np.genfromtxt('intrinsic_matrix.csv', delimiter=',')
dist = np.genfromtxt('distortion_coefficients.csv', delimiter=',')
```

Note that if you tried saving rvecs and tvecs, they use nested lists that do not work well with the CSV writer, so you may have to strip the square brackets out if you want to re-use them (though it's not needed in this lab) with something like

```
with open('rotation_vectors.csv', 'r') as file:
     text = file.read().translate({ord('['): None, ord(']'): None})
     with open('rotation_vectors2.csv', 'w') as file:
     file.write(text)
```

For the maximum of dimensional accuracy, you should also undistort the image after taking it (though it will likely make very little difference as you saw in the previous part of the lab) with an operation such as

```
h, w = img.shape[:2]
newcameramtx, (x, y, w, h) =
cv2.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1, (w,h))
```

```
dst = cv2.undistort(img, mtx, dist, None, newcameramtx)
img = dst[y:y+h, x:x+w]
```

As you did before, you should put the processing of multiple images after the *cv2.waitKey(1)* main capture loop exits and you show the "Captured Image". Make sure you save the image with `imglist.append(img)`.

The simplest way to create a disparity map in OpenCV is to create a Stereo Block Matcher with *cv2.StereoBM.create()*. *StereoBM* uses the assumption of *Epipolar Geometry* to compute disparity efficiently by comparing features only along horizontal lines in the image. This assumption is made by observing that if two cameras have extrinsic translation only along a horizontal *Epipolar Plane*, then disparity in their images will occur only along the line of that plane, as illustrated in Figure 3.

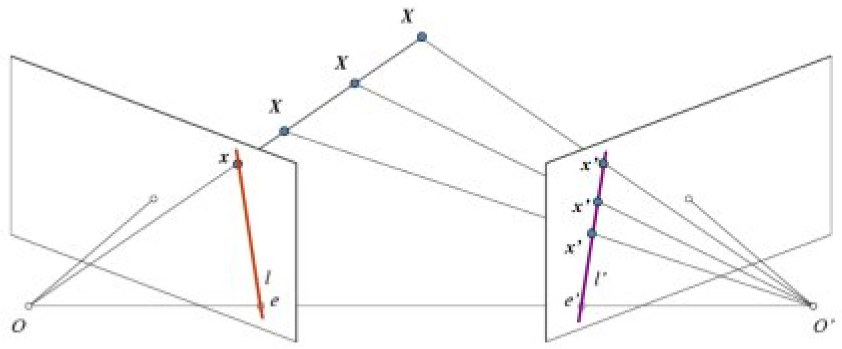

Figure 3: Epipolar Geometry

If we are using only the left camera (*O*), we can't find the 3D point corresponding to the point x in image because every point on the line *OX* projects to the same point on the image plane of *O*. But including the right image (*O'*), different points on the line *OX* project to different points *x'* in the right plane of *O'*. So with these two images, we can remove ambiguity and triangulate the correct 3D point

Therefore **you must ensure that images are taken with purely horizontal disparity**. The easiest way to do this is to slide your camera across the table surface while capturing two separate images. Also, because relatively small blocks are compared for matching, it works best when **very small movements are used** (on the order of centimeters). You will also have to ensure that you have at least two images saved in *imglist* to get a disparity between them, so after you after you press a key to capture and save the image, create a conditional with

```
if(len(imglist) > 1):
```

All following stereo block matcher code must be indented so it is considered to be within this conditional code block.

The block matcher only operates on **greyscale** images so you will need to convert both images to greyscale. In Python, the last image in an array can be conveniently accessed as index *-1* and the second-last can be accessed as index *-2*, and so on. You can convert the last two images taken in sequence with

```
gray1 = cv2.cvtColor(imglist[-1], cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(imglist[-2], cv2.COLOR_BGR2GRAY)
```

Create the Stereo Block Matcher with default parameters using

```
stereo = cv2.StereoBM.create(numDisparities = 16, blockSize = 21)
```

Then compute the disparity map itself with

```
disparity = stereo.compute(gray1, gray2)
```

The disparity map is an array of 16-bit signed integers which is different from the usual OpenCV image matrix of planes of 8-bit unsigned integers. If you have *matplotlib* installed you can display it directly with `matplotlib.pyplot.imshow(disparity, 'gray')` and then `matplotlib.pyplot.show()`, or you can *normalize* it to show as an image using *cv2.imshow()* with

```
disparity = cv2.normalize(disparity, None, alpha=0,
beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
```

Normalize the image and then show it as a Captured Image (don't forget to put a cv2.waitKey(0) afterwards or it will not draw the image immediately).

Now, try to capture pairs of images with your program and see how the disparity maps look. Remember to move the camera in very small distances between images, and keep the cameras horizontal on the *Epipolar Plane*. You may find initially that images look “fragmented” (too much movement, or movement off the plane) or have large dark areas (insufficient features for block matching). Look at the *cv2.StereoBM* documentation to see what other functionality is available. You may be able to adjust the parameters slightly to improve your disparity quality. A list of the adjustable parameters is given below and detailed in this Depth Map from Stereo Images tutorial. Which parameters do you think would help to improve your disparity maps?

- numDisparities: How many pixels to slide the window over. The larger it is, the larger the range of visible depths, but more computation is required.
- blockSize: The linear size of the blocks compared by the algorithm. The size should be odd (as the block is centered at the current pixel). Larger block size implies smoother, though less accurate disparity map. Smaller block size gives more detailed disparity map, but there is higher chance for algorithm to find a wrong correspondence.
- TextureThreshold: filters out areas that don't have enough texture for reliable matching
- MinDisparity: the offset from the *x*-position of the left pixel at which to begin searching.
- UniquenessRatio: If the best matching disparity is not sufficiently better than every other disparity in the search range, the pixel is filtered out.
- PreFilterSize and PreFilterCap: The pre-filtering phase, which normalizes image brightness and enhances texture in preparation for block matching. Normally you should not need to adjust these.

Note that since block/feature matching is used, for surfaces having no texture, finding the corresponding point becomes difficult. The same situation occurs when the texture has repetitive patterns. So disparity mapping (as with most matching) will work best when the scene contains many unique patterns.

Now, a tricky question: did we measure the baseline *B* between our camera positions somehow? **No we didn’t!** So why are you still able to get a disparity map with an unknown camera baseline? It is important to note that finding disparity *alone* between two images does not require any calibration. However, without a known baseline *there is no way to know what physical distance the disparity corresponds to*. Hence, doing this calculation with a known baseline will allow you to estimate distance using the disparity map.

3. Feature Matching

While disparity mapping is relatively simple and useful if a known baseline is provided, for more general 3D machine vision applications we want to be able to determine the motion of the camera itself from a sequence of images and features. This “ego-motion estimation” (self-motion) is also commonly known as *visual odometry* as it can be used in the place of localization sensors to estimate the movement of a camera. We can in addition use it to obtain a camera baseline.

Visual odometry starts with the identification of features in a sequence of images (not unlike the sequential stereo vision you tried above). However instead of block matching along epipolar lines we will now use more general feature matching methods to estimate the common keypoints between images. If we are not using an extremely large number of features, we can afford to use brute-force matching to improve the potential performance of the algorithm.

Start by finding ORB features in the main capture loop of your program and writing the keypoints to the image so that you can see them

```
orb = cv2.ORB_create()
kp = orb.detect(img, None)
kp, des = orb.compute(img, kp)
img_orb = cv2.drawKeypoints(img, kp, None, flags=0)
```

You can display *img_orb* with *cv2.imshow()* while retaining the original *img* for further processing.

Now, when you save an image with *imglist.append(img)* also save the keypoints *kp* **and** the descriptors themselves *des* in a list with

```
imgpoints.append(kp)
imgdescs.append(des)
```

Make sure to initialize these lists to empty at the beginning of your program with

```
imgpoints = []
imgdescs = []
```

After you check for more than one image saved with `if (len(imglist) > 1):`, in the block where you did stereo matching you can also do the keypoint matching. It may be easier to understand if you unpack the images, keypoints, and descriptors into separate variables rather than indexing lists all the time

```
img1 = imglist[-1]
img2 = imglist[-2]
kp1 = imgpoints[-1]
kp2 = imgpoints[-2]
des1 = imgdescs[-1]
des2 = imgdescs[-2]
```

Create a BFMatcher object with distance measurement parameter set to *cv.NORM_HAMMING* (since we are using ORB we use Hamming distance for matching). You can also switch on the *crossCheck* parameter to further compare matches for better results.

```
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

The *Matcher.match()* method then obtains the best matches between the two sets of keypoint descriptors.

```
matches = bf.match(des1, des2)
```

We then want to sort the matches in ascending order of their distances so that best matches (with low distance) come to front.

```
matches = sorted(matches, key = lambda x:x.distance)
```

Then draw the matches onto an image with

```
img_matches = cv2.drawMatches(img1, kp1, img2, kp2, matches,
None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

Show *img_matches* with *cv2.imshow*. Do the matches you have look reasonable? Note that if you have too many matches at once you can select for example the 10 best matches (because they were sorted above) by replacing `matches` with `matches[:10]` (the 10 first elements of the array) in any function that uses them. Consider how many obviously bad matches you may have in your image and try this out with a number that estimates the number of good matches you think you have. Do the bad matches go away?

4. Visual Odometry

The last step that we need to estimate Extrinsic motion for Visual Odometry is to use the feature matches we have made to triangulate the left and right camera positions in 3D space given that we now know which points on Epipolar Lines correspond to each image. The transformation from one projective plane to another that represents the relative positions of our two cameras is called a *Homography* which etymologically, approximately means "similar drawing". As with all geometric transformations, a Homography is typically represented by a Matrix that transforms all the points from one image into another. This “Homography Matrix”, in multiple-view geometry terminology, is called the *Fundamental Matrix (F)*. And it is absolutely fundamental to 3D machine vision!

We can find the *Fundamental Matrix* from our features in pixel coordinates, then we need to convert it into an Extrinsic Homogeneous Matrix that represents the translation and rotation of our second camera with respect to our first. In multiple-view geometry terminology, this is called the *Essential Matrix (E)*, which is absolutely essential for ego-motion estimation! The *Essential Matrix* describes the translation and rotation of cameras in Epipolar Geometry as shown in Figure 4, without the Intrinsic calibration of the camera that turns 3D coordinates into 2D image coordinates. The Essential and Fundamental matrices are related by $E = K'^T * F * K$, where K is the Intrinsic matrix of the camera.

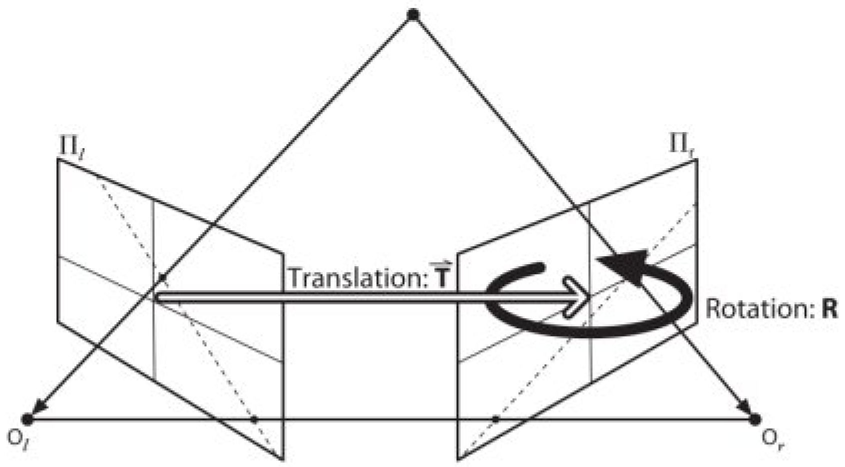

Figure 4: Geometry of the Essential Matrix

The first things you need to do is unpack the matches 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 now use the *cv2.findFundamentalMat()* function to find the Fundamental Matrix corresponding to these matched keypoints.

```
F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_LMEDS)
```

Print out the value of F. Can you make a guess at what its row and column elements correspond to?

You can find the Essential matrix through its relational equation to F, or you can use the OpenCV function *cv2.findEssentialMat()* that finds it directly by using the Intrinsic Matrix to transform the points directly to 3D correspondences.

```
E, mask = cv2.findEssentialMat(pts1, pts2, mtx, cv2.LMEDS, prob=0.999)
```

Print out the value of E and compare it to F. What are the similarities and differences?

In these solvers, we have used the least-median-of-squares (*cv2.LMEDS*) method that estimates the parameters by solving a nonlinear minimization problem. Another option is RANdom SAmple Consensus (cv2.*RANSAC*) that an iterative method for estimating a mathematical model from a data set that contains outliers. The RANSAC algorithm works by identifying the outliers in a data set and estimating the desired model using data that does not contain outliers. It may be useful to try as an alternative if you are not getting good homographies due to outlying incorrect matches of feature points. In both cases, the *prob* parameter specifies the probability of a correct solution to aim for (the larger the number, the harder the algorithm will work to find a correct solution).

The last step is to estimate the rotation *R* and translation *t* matrices for ego-motion estimation from the Essential Matrix E you have obtained. There is of course a function for this: *cv2.recoverPose()* called with

```
_, R, t, _ = cv2.recoverPose(E, pts1, pts2, mtx)
```

Put all these functions together in your program and try obtaining pairs of images - several in a row if you wish - and print out the corresponding values of *R* and *t* for each transformation.

Look at the results carefully.

- Do the values of *t* resemble the vectors of translation along which you moved the camera in each step?
- Do the values of *R* (in radians) look like a rotation matrix describing the rotation of the camera
- Do these values increase or decrease as you increase or decrease the camera motion along an axis?

Remember that Visual Odometry relies on having good matches between each set of poses, so if you are not getting good results, try moving the camera less or looking at scenes with more distinct features. You may also want to revisit the parameters of the functions you have used

Congratulations, you can now track the motion of a camera! (to some degree of accuracy…how much?)

Now, remember the disparity image that we obtained without a baseline? Since you have ego-motion estimation of the camera, you can use the translation distance *t* as the baseline distance for each pair of images. Try applying the Stereo Vision equation $Z = B * f / (x - x')$ to the disparity map that you obtained (which is a map of $(x - x')$, remember to use the un-normalized version!) Can you obtain an estimate of actual depths?

OPTIONAL ACTIVITY 1: Draw the Epipolar Lines

It may be interesting to actually see the epipolar lines drawn on your images. Following the Epipolar Geometry Tutorial, once you have the Fundamental Matrix you can find the epilines. Epilines corresponding to the points in the first image are drawn on the second image. We can define a function at the beginning of your program to draw these lines on the images.

```
def drawlines(img1,img2,lines,pts1,pts2):
    r,c = img1.shape
    img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
    img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
    for r,pt1,pt2 in zip(lines,pts1,pts2):
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1] ])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
        img1 = cv2.line(img1, (x0,y0), (x1,y1), color,1)
        img1 = cv2.circle(img1,tuple(pt1),5,color,-1)
        img2 = cv2.circle(img2,tuple(pt2),5,color,-1)
    return img1,img2
```

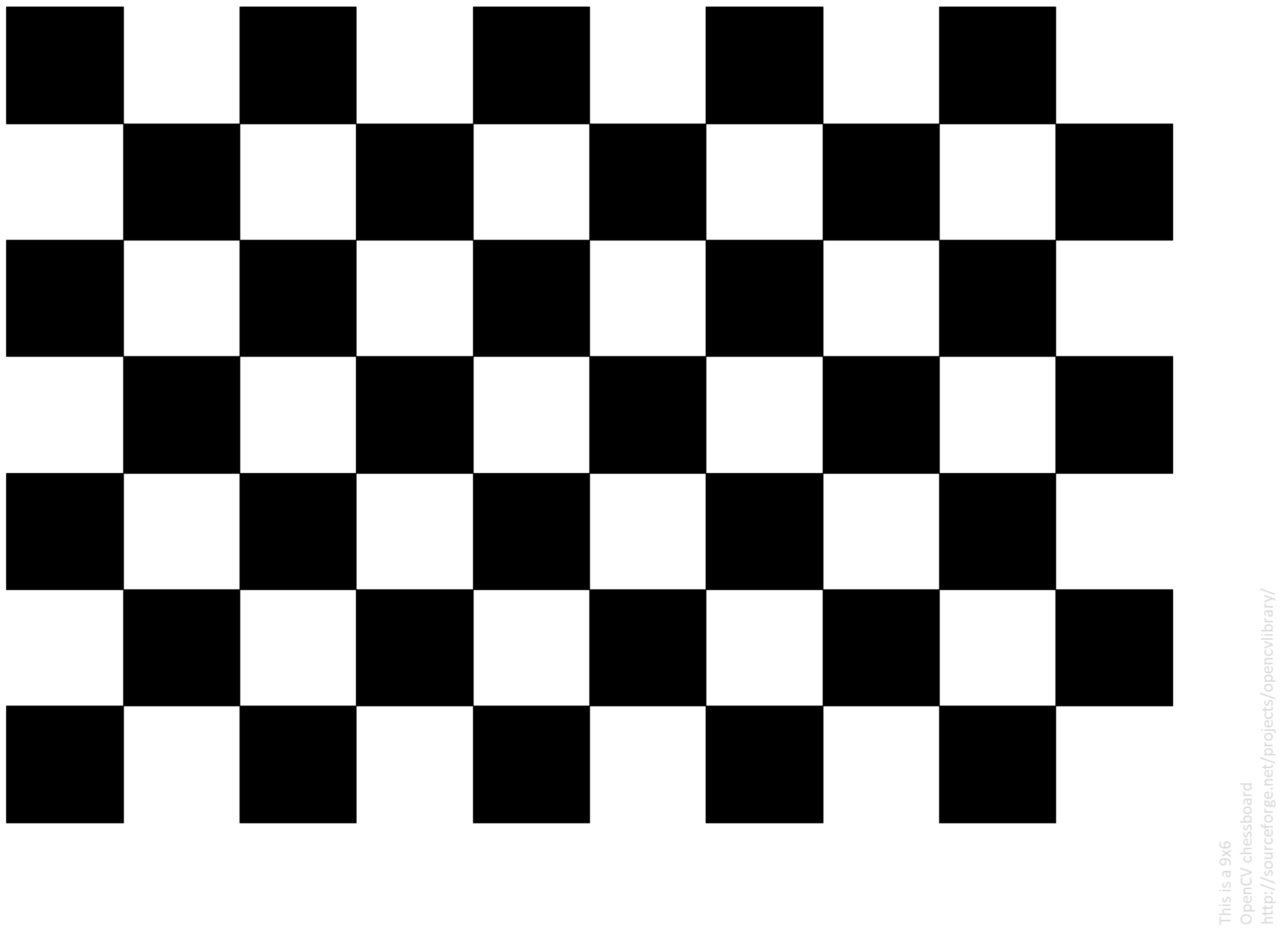

Figure 3: Chessboard Calibration Pattern (pattern_chessboard.png)

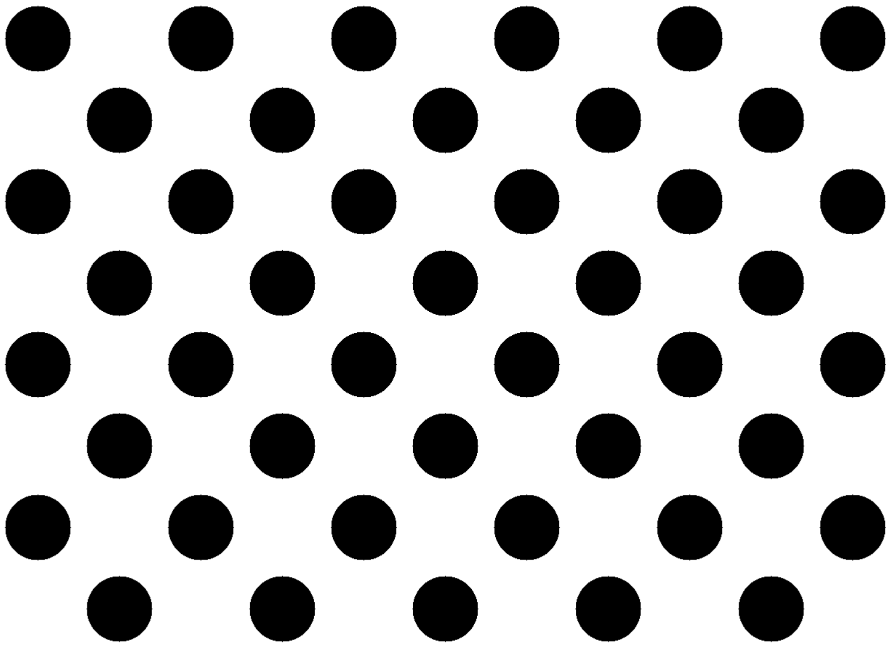

Figure 4: Circles Calibration Pattern (pattern_acircles.png)