PROJECT: CAMERA CALIBRATION USING ZHANG METHOD

STUDENT: Anwar Nazig

COURSE: Computer Vision and pattern recognition 2020/2021

Introduction

This report describe from an algorithmic point of view the steps involved in camera calibration using Zhang method and describes the implementation of each step required from project assignment in the python language.

The aim of calibration is to find the effective projection transform, hence yielding significant information regarding the vision system such as intrinsic camera matrix, extrinsic parameters and the distortion coefficients. The basic model for a camera is a pinhole camera model.

POINT 0: **preparation phase**

The preliminary steps that has been taken in order to satisfy all the requirements are:

1. Take the given the image set of a checkerboard from different viewpoint,
2. Use the opencv function findChessboardCorners [[1]](#footnote-1)in order to find (u , v) coordinates for each point in the 3d space
3. Define real world coordinates of 3d points using checkerboard pattern of known size.

Four functions has been implemented in order to collect our data and start the calibration process:

GetCorners that takes as parameters the image path and other two that defines the chosen pattern (i.e. the number of corners per row and column to detect) and have as output the pixel coordinates of all detected points.

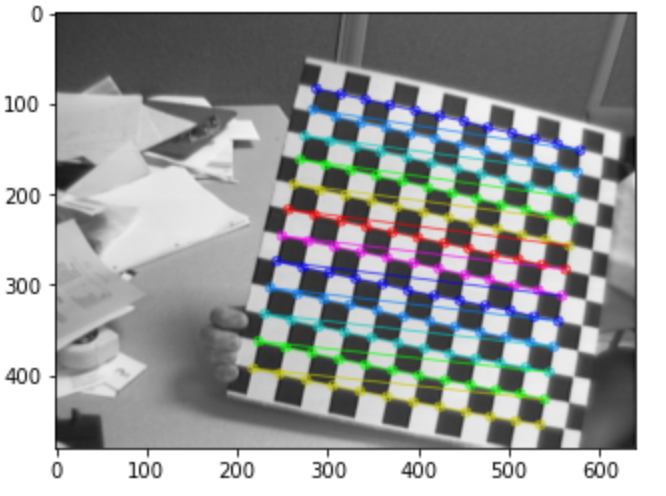


Figure 1:result of GetCorners

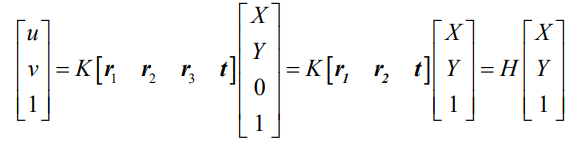
getWorldCoordinates that takes as inputs the set of found corner poinst , dimension of the 2 sides of a rectangle in millimetres and the number of rows that must be equal to the number of columns, the design choice behind this constraint is that we have assumed that checkerboards are composed just of squares. As an output we obtain the corresponding point in the 3D World Coordinate System assuming z = 0. (notice that the all 3d world point are already known and the function just associate the correspondence between 2d point and their relative 3d point).

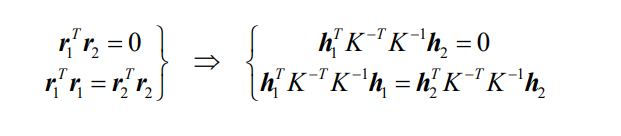
find\_detectable\_images finds what are the images of a given pattern for which the findChessboardCorners method detects the corners, this has been done because the method does not always detect corners for a certain pattern in all the images in the set. During experiments for the image set, the pattern that finds the most images is 12 x 12.

getHomographie find the homography matrix given the detected corners and the corresponding point in the world coordinate system.

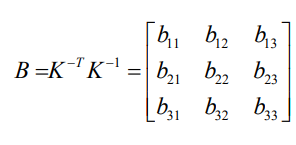
POINT 1: **calibrate using Zhang procedure, i.e. find the intrinsic parameters K and, for each image, the pair of R, t (extrinsic);**

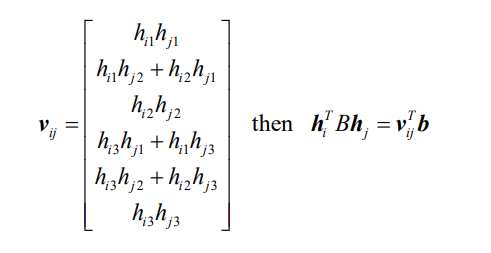
Zhang’s calibration method in short requires that the calibration object is planar such as a checkerboard , and requires at least 3 images in order to proceed. the 3D -2D relationship is described by a homography



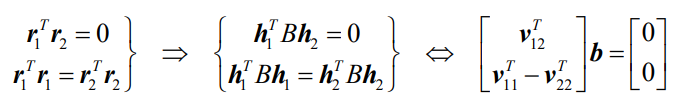
having 2 constraints on the intrinsic parameters due to the fact that 𝑅 is orthonormal.

The next step in the process is to isolate our unknown parameter in order to compute the matrix B.



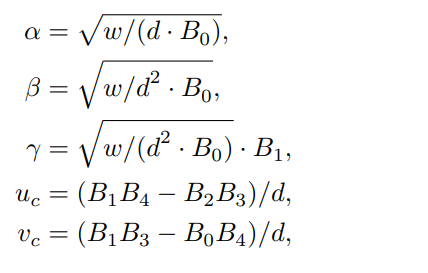
let’s define the matrix Vij where i, j ∈ {1, 2}, it’s easily verified that if:

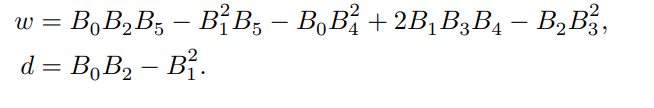
In conclusion this relation is obtained:

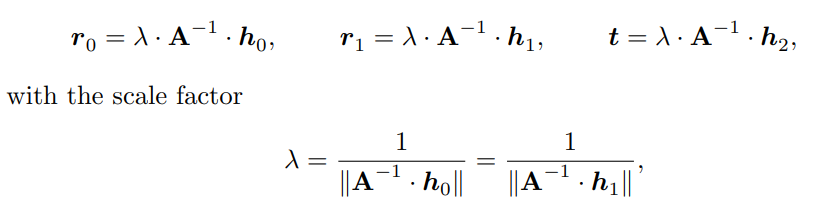


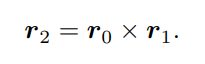
For n planes, by stacking equations of the above form, we get Vb = 0 where V is a 2n x 6 matrix. The solution for the system of equations is obtained by taking the right singular vector of V corresponding to the smallest singular value.

Now given B the matrix K can be found by using the Cholesky factorization or by estimating intrinsic para meters of K: alpha ,beta ,gamma, uc, vc .





The last step is to find extrinsic parameters R, t for each image by noticing that:



Where A is the calibration matrix K.

The resulting 3×3 matrix R = (r0 | r1 | r2 ) is most likely not a proper rotation matrix. However, there are proven techniques for calculating the most similar rotation matrix for a given 3×3 matrix using the singular value decomposition of R = UΣVT. so the orthogonal matrix is given by R’ = UVT.

**Implementation steps.**

COMPUTING INTRINSIC PARAMETERS

The first step consists in computing for each image it’s homography by using the method getHomographie described before and each matrix is saved in a list homographies.

To obtain the calibration matrix get\_intrinsics\_parameters method is called by passing to it the list of homographies. The method have an internal function v\_pq that is called 2N times (i.e for each homography two vectors are built v12 and the v22-v11 ) building the vector V.

Now that we have V we compute the singular value decomposition on V by calling the numpy function linalg.svd(V) [[2]](#footnote-2)and we take the right singular vector of V corresponding to the smallest singular value.

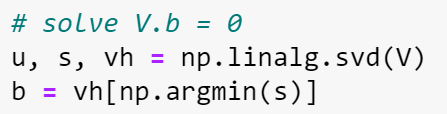


Figure 2: code to obtain b

The last step consist in computing the Cholesky decomposition of B or use the close form approximation. The first approach has not been used duo to the fact that the obtained B till this point might not be positive definite as required by the Cholesky decomposition.

once K parameters has been estimated as described before the K matrix is built as below:

COMPUTING EXTRINSIC PARAMETERS

To compute the extrinsic parameters for each image we call the implemented method estimate\_view\_transform by passing to it the calibration matrix K and the image homography, the returned result is a vector E compose in this way [r0 r1 r2 t], R and t are obtained by simply taking the first 3 columns of E, while t by taking the last column of E.

Notice that all the extrinsic parameters (i.e. for each image R and t ) are saved inside a list called E\_s that will be used in the next points.

POINT 2: **choose one of the calibration images and compute the total reprojection error for all the grid points (adding a figure with the reprojected points);**

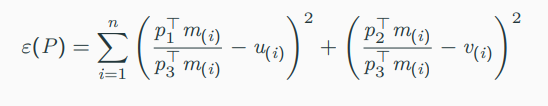
To calculate the reprojection error we must find the perspective projection matrix P, defined as:

So given the calibration matrix and the extrinsic matrix E we can obtain P for each image by calling the implemented method getP. At this point we can estimate u and v as

where mi is the homogeneous coordinate of the 3d point.

In practice to get these value for u and v we call the implemented function called reprojection\_error that takes as input parameters Extrinsics, K calibration matrix , corner pixel coordinates and the image we want to compute the reprojection error for.

Re-projection error gives a good estimation of how exact is the found parameters. Given the intrinsic and the perspective projection we’ve calculated the error by using the geometric residual that we got with our transformation and the corner finding algorithm.



To find the average error we calculate the arithmetical mean of the errors calculate for all the calibration images and obtained a mean value equal to 362.69 , maximum value and minimum are respectively 742.55, 133.74.

As reference in this report we take image named image2.tiff having a reprojection error equal to 133.74

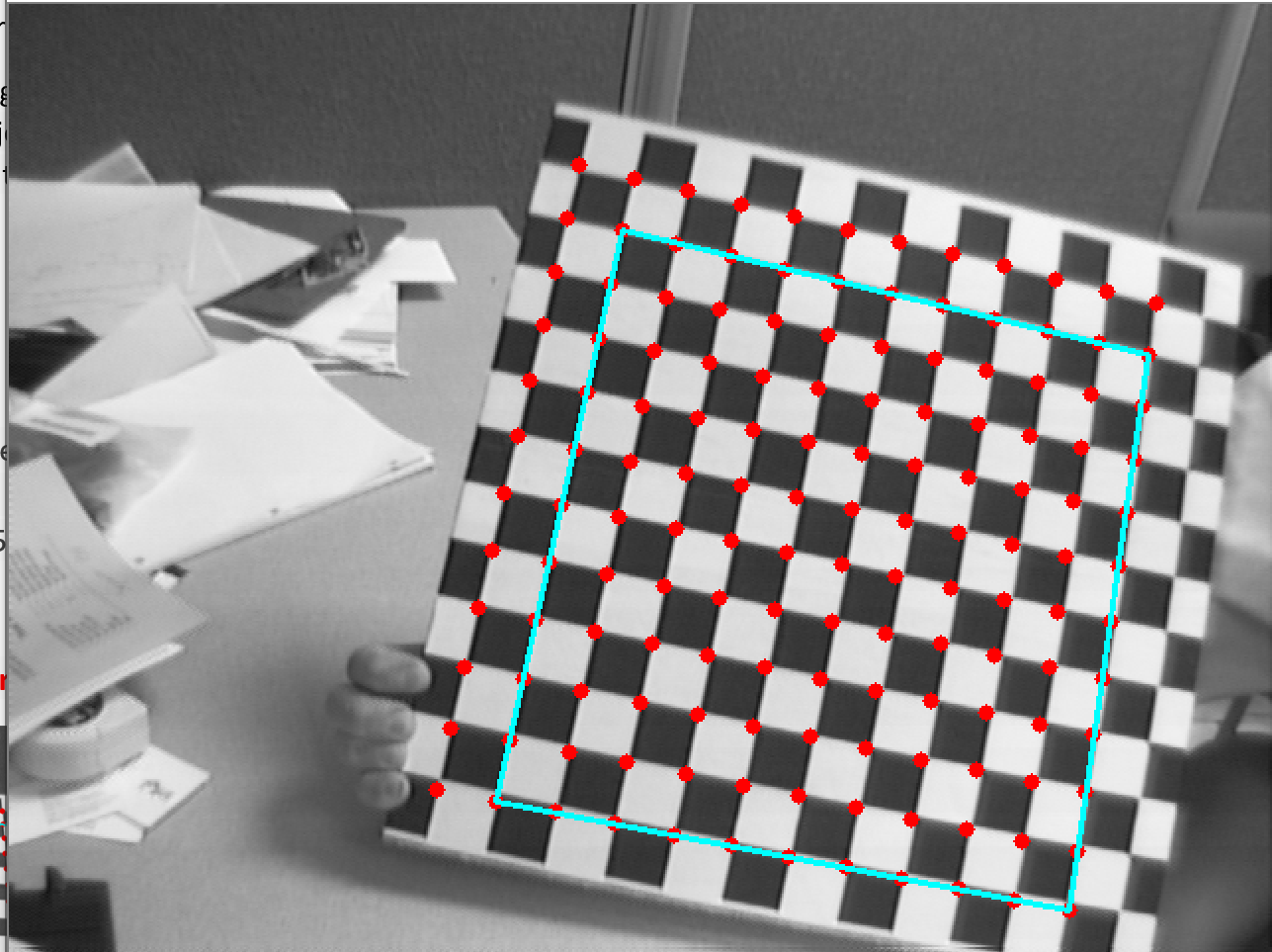
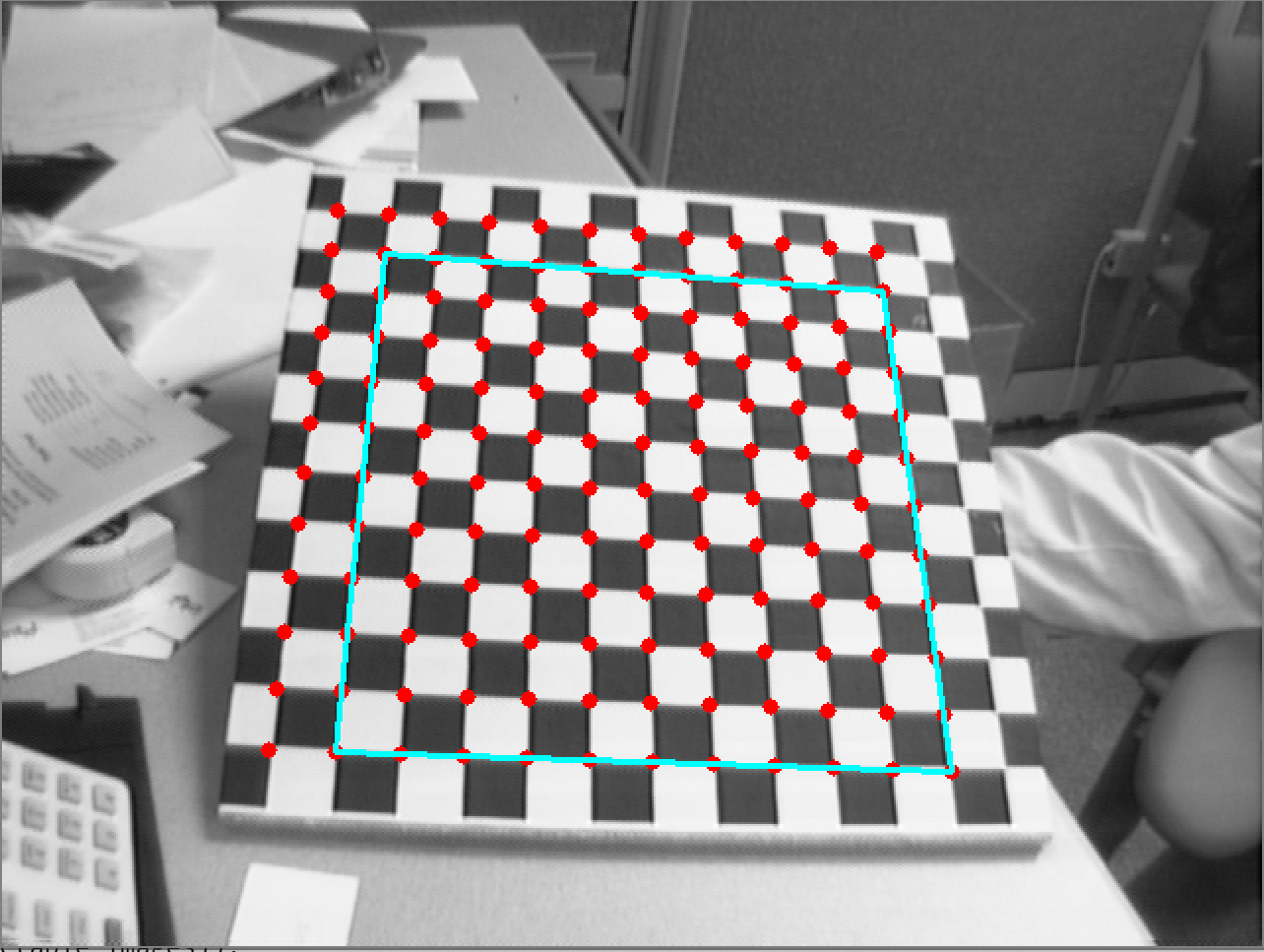
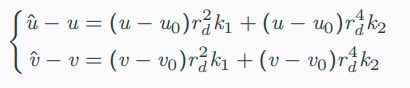


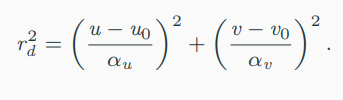
Figure 3: In red, point that has been reprojected for each image, in sky blue the reprojected square.

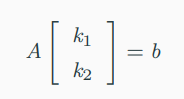
POINT 3**: add radial distortion compensation to the basic Zhang’s calibration procedure;**

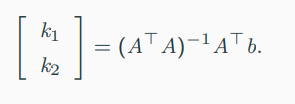
All calculations done so far has been done under the hypothesis that the inner camera transformation follows the pinhole projection model. more precisely, the distortions introduced by real lens systems were ignored so far. In this phase, a non-linear lens distortion model is used in combination with the projection pipeline and its parameters are calculated from the observed images.

This step is accomplished by **first estimating the distortion coefficients k1 and k2** using a linear least-squares fitting. More precisely assuming the perspective projection matrix P is known we can compute the ideal undistorted coordinates (u , v ) from the world coordinates (x,y,z) , square of the radius rd2  and by stacking all our correspondences and a single value decomposition is performed to find the estimate of k1 and k2.









To implement this step estimate\_lens\_distortion method is used to obtain the coefficients k1 and k2 , the method in input gets the calibration matrix, the perspective projection matrix the pixel coordinates and their world correspondences. The function basically builds the system of equations, extracts the matrix A and b and applies the last formula to get k1 and k2.

The **second step consist in compensating for radial distortion** by finding the new ideal coordinates (u,v ) by solving a nonlinear system. The solutions are obtained by an iterative method (newton\_krylov[[3]](#footnote-3)).

The implementation of this second step follows the following pipeline:

1. Function called GetNewPixels has been defined, it gets as inputs: the world coordinates , a set of pixel coordinates (in the first iteration pixel coordinates are the one detected by the checkerboard ) and the initial 2d detected coordinates. For each image the method returns the new (u,v) coordinates, for each image the coefficients (k1,k2 ), the calibration matrix K common to all images and a set of perspective projections where each P relative to the corresponding image. In summary the functions do all the calibration steps from finding homography to computing lens distortion described before and then by passing all this data to another function called getUandV it return the new coordinate for (u,v).
2. the function getUandV solves the nonlinear system describe previously , in short the function first define our system of equations and then by calling the python function newton\_krylov it iteratively tries to find the solutions, first by a given initial guess and if no solution is found it tries two more times with different guesses.

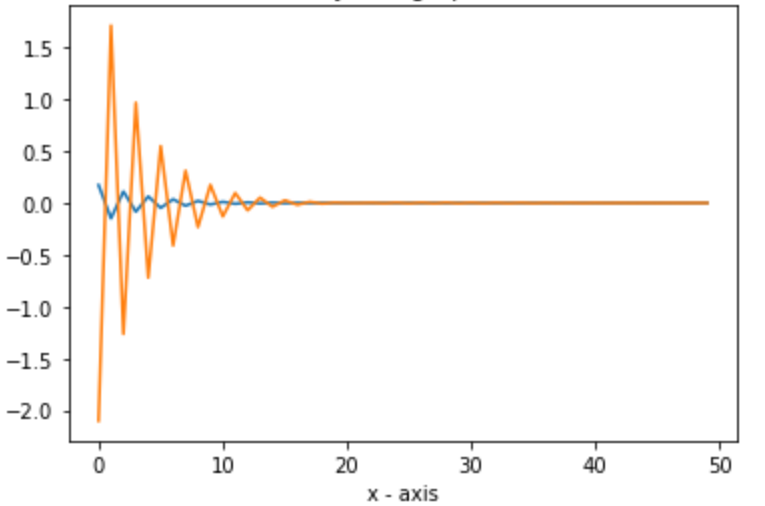
The **last step consist in refining our parameters**. More precisely the GetNewPixels function is called in loop until convergence of P, k1 and k2. Notice that the function is called in the loop starting from second iteration and with the only difference that the parameter 2d coordinates we are giving now to the function are the one computed previously as shown in the example:



In the first iteration we will use set\_of\_corners that corresponds to the detected corners in checkerboard initially. While in the loop we will use the computed new corners as shown below



In conclusion the convergence of coefficient k1,k2 and P requires usually a number of iteration about 20 as we can see from the figure below where in orange we have the difference between the previous k1 and actual k1. The same for k2 looking in blue.



Now by using the using the undistort[[4]](#footnote-4) opencv method we can visualize the result of our radial compensation as in the following images.

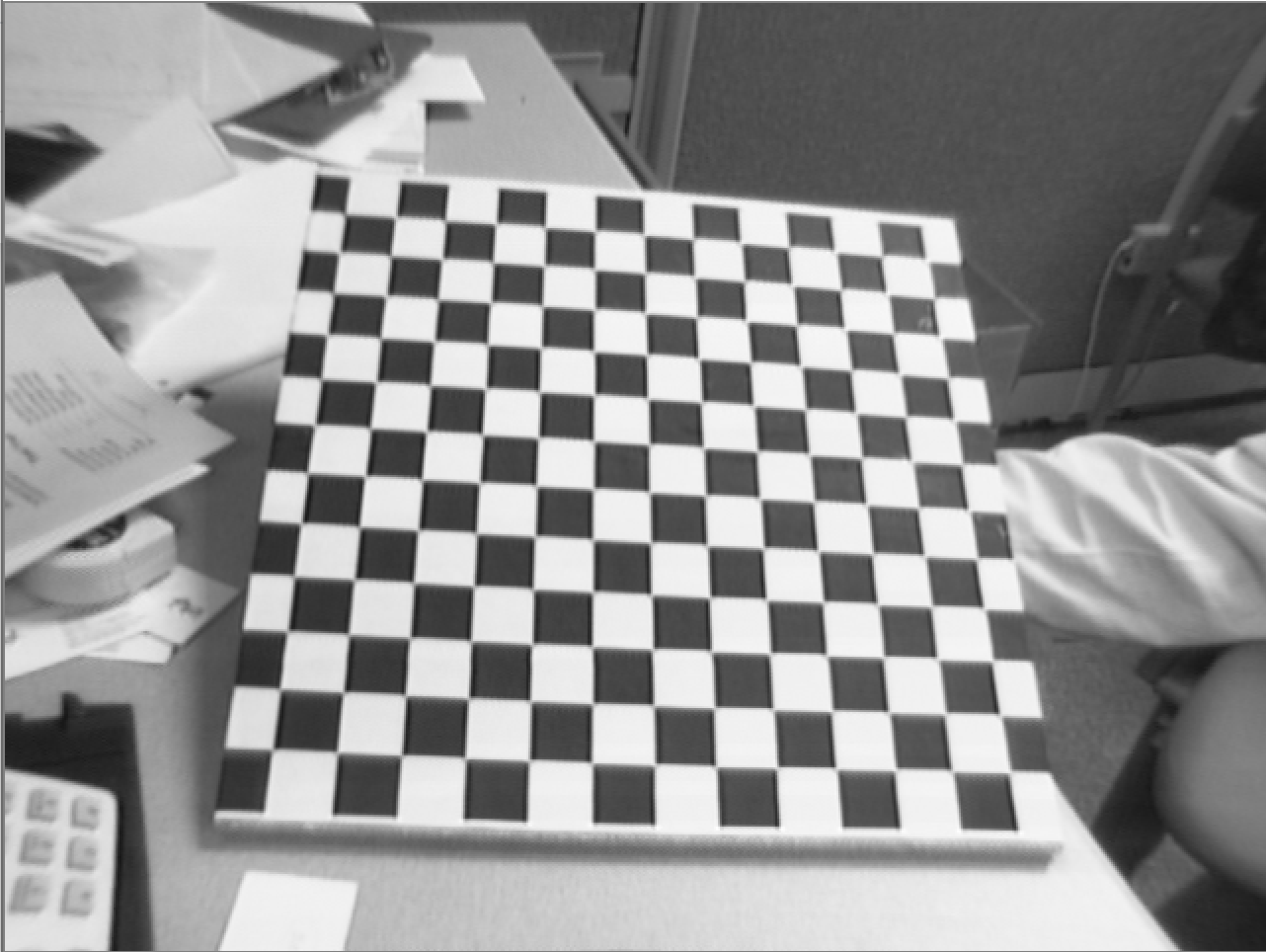
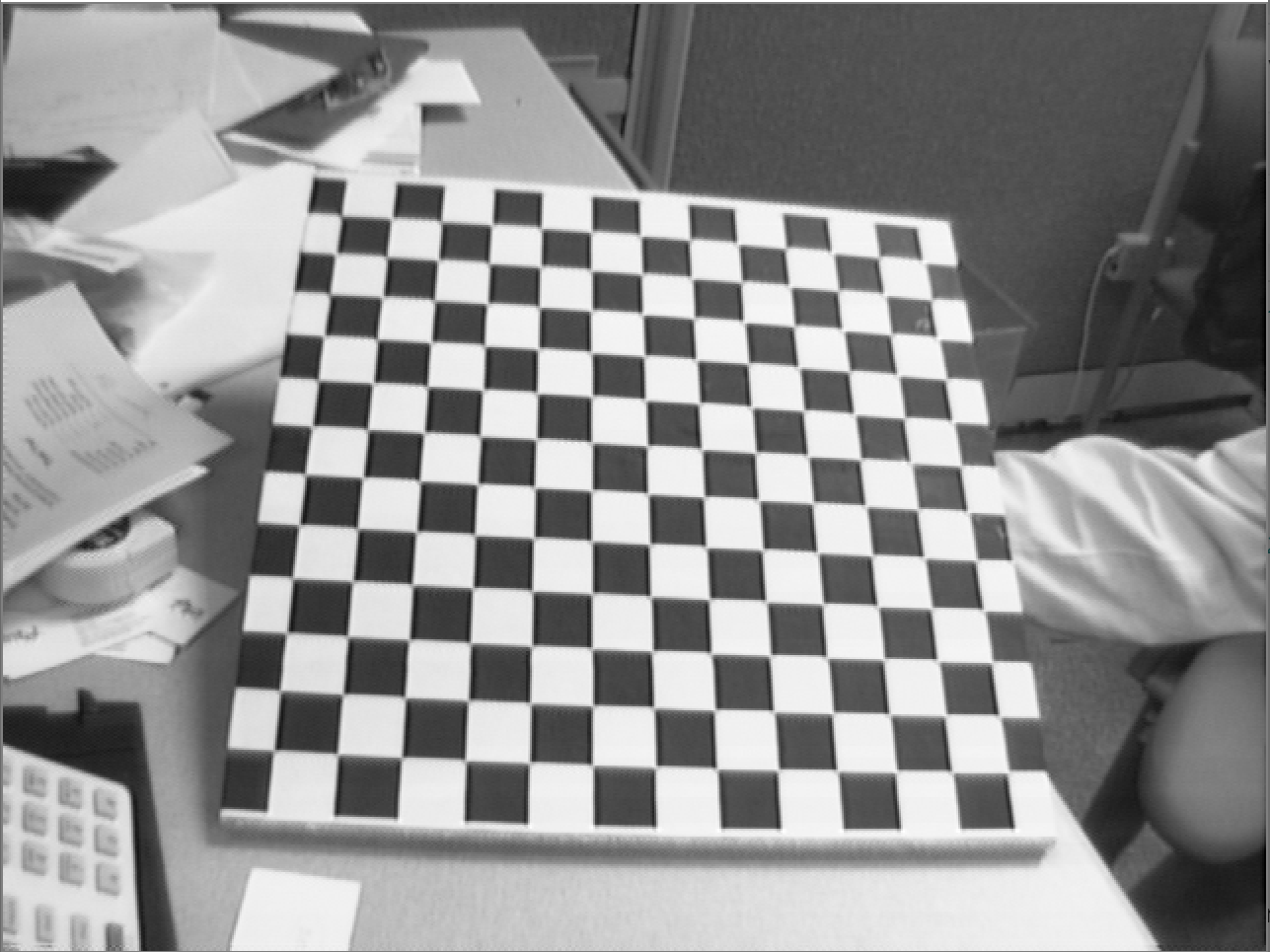


Figure 4:left distorted image, right the undistorted image

POINT 4: **compute the total reprojection error with radial distortion compensation and make a comparison to the one without compensation;**

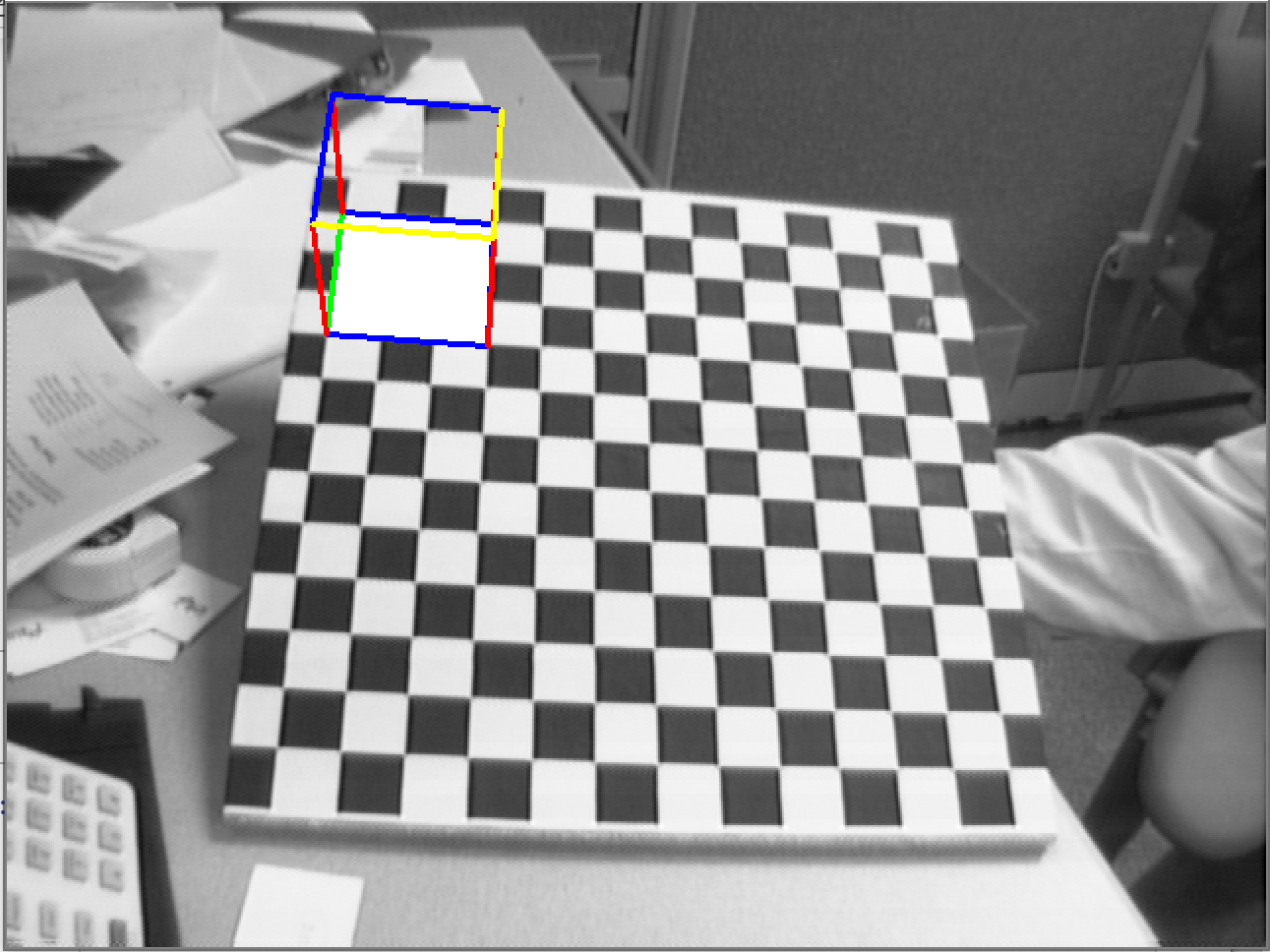
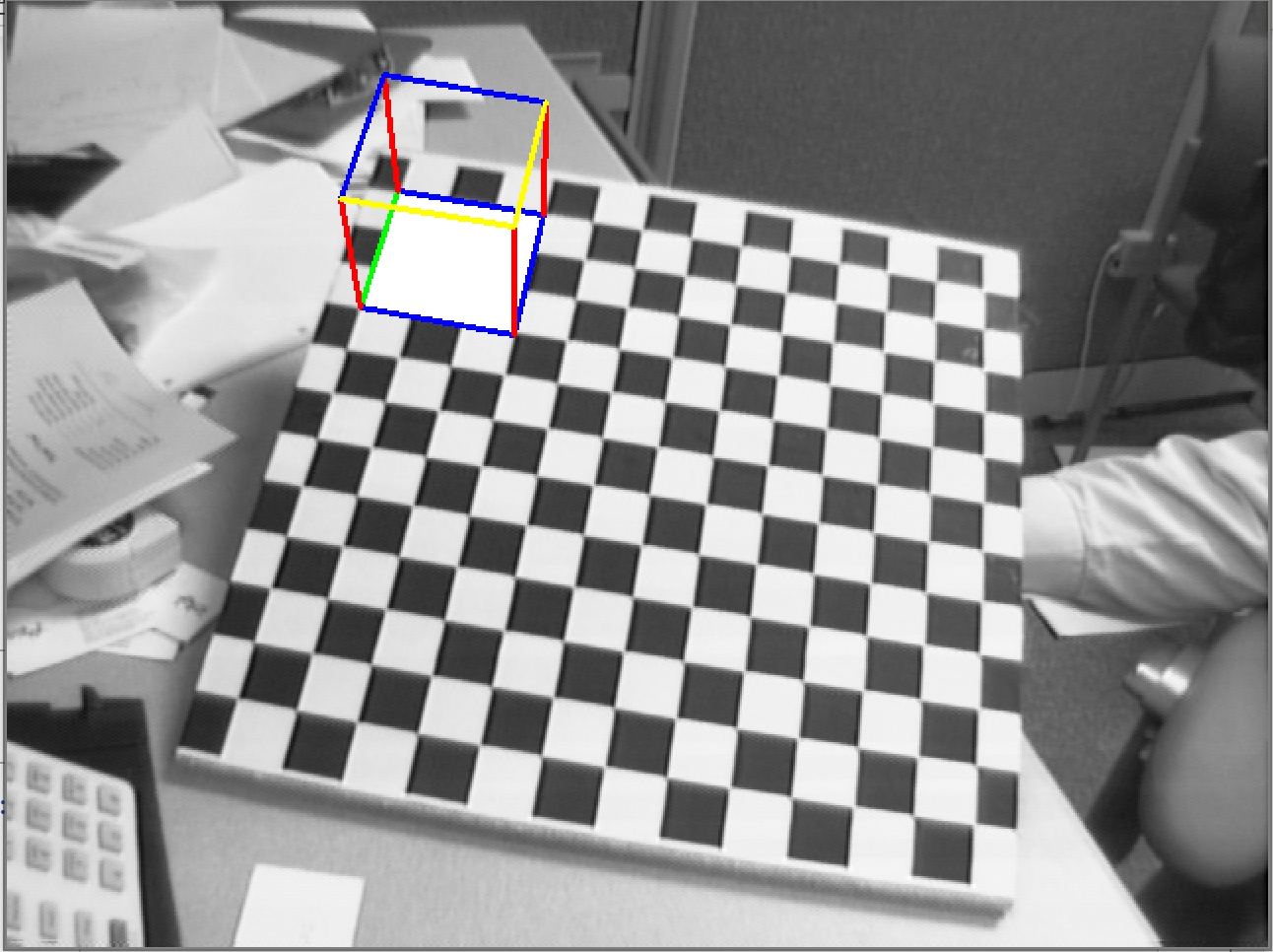
To compute the total reprojection with radial distortion we use the geometric residual

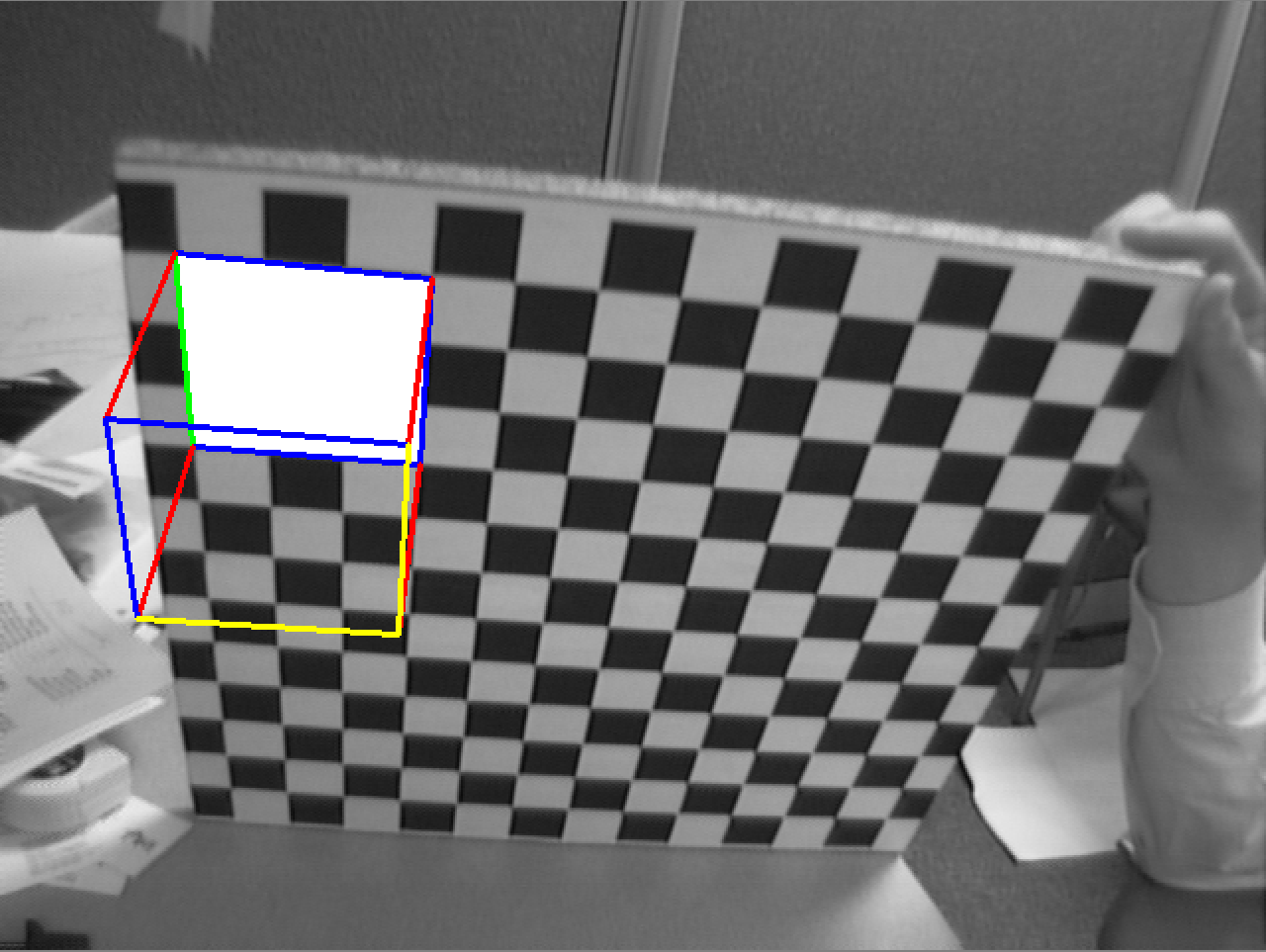
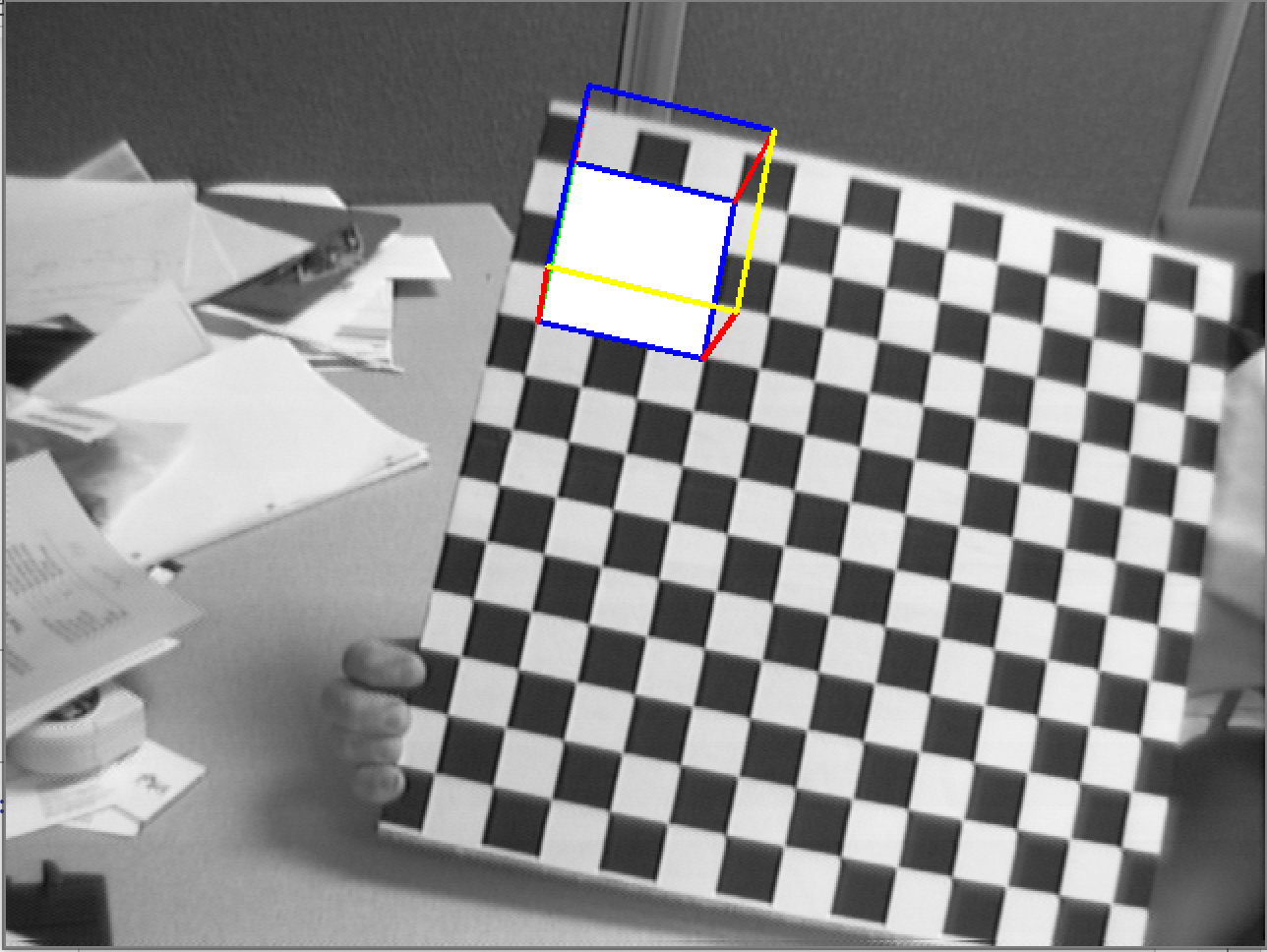
Where are the pixel coordinates detected initially and subjected to the distortion phenomena, while are the pixel coordinates obtained after distortion compensation.

The reprojection error obtained for our reference image (image2.tiff ) as described before Is about 18.18 , and among all the images the maximum reprojection error obtained is about 73.

The difference between the two reprojection errors (i.e. the distorted and undistorted) amounts to 133 – 18 = 115.

POINT 5**: superimpose an object (for instance, a cylinder as in Fig. 1), to the calibration plane, in all the images employed for the calibration;**The goal now is to take all the images for which we’ve computed the extrinsic’s and intrinsic’s and super impose a cube having origin at a chosen coordinates and side size as shown in the following images:

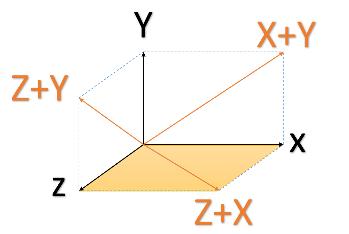
 

drawCube method has been implemented which takes in input the following parameters: the image number, the perspective projection matrix, coordinates of the origin (i.e. origin in the example [0,0,0], but can be any other desired point) and side dimension (in our example the dimension is 90mm).

two internal functions are called inside the drawCube method: getP that has been already explained and planProject, that given the perspective projection matrix and a 3d point finds the corresponding pixel coordinates in the image.

The cube is obtained by first finding the three cartesian axis x,y,z, for example axis x is defined as [side dimension , 0 , 0,1] in homogeneous coordinates and z as [0,0,-side\_dimension,1] , here side\_dimension is taken negative in order to show z axis as coming out of the checkerboard and not vice versa. To each of these vectors a perspective projection is applied in order to obtain the pixel coordinates. At this step given the 3 axis we can apply a reasonable combination of the three vectors in order to obtain the remaining points of the cube as shown in the image below.

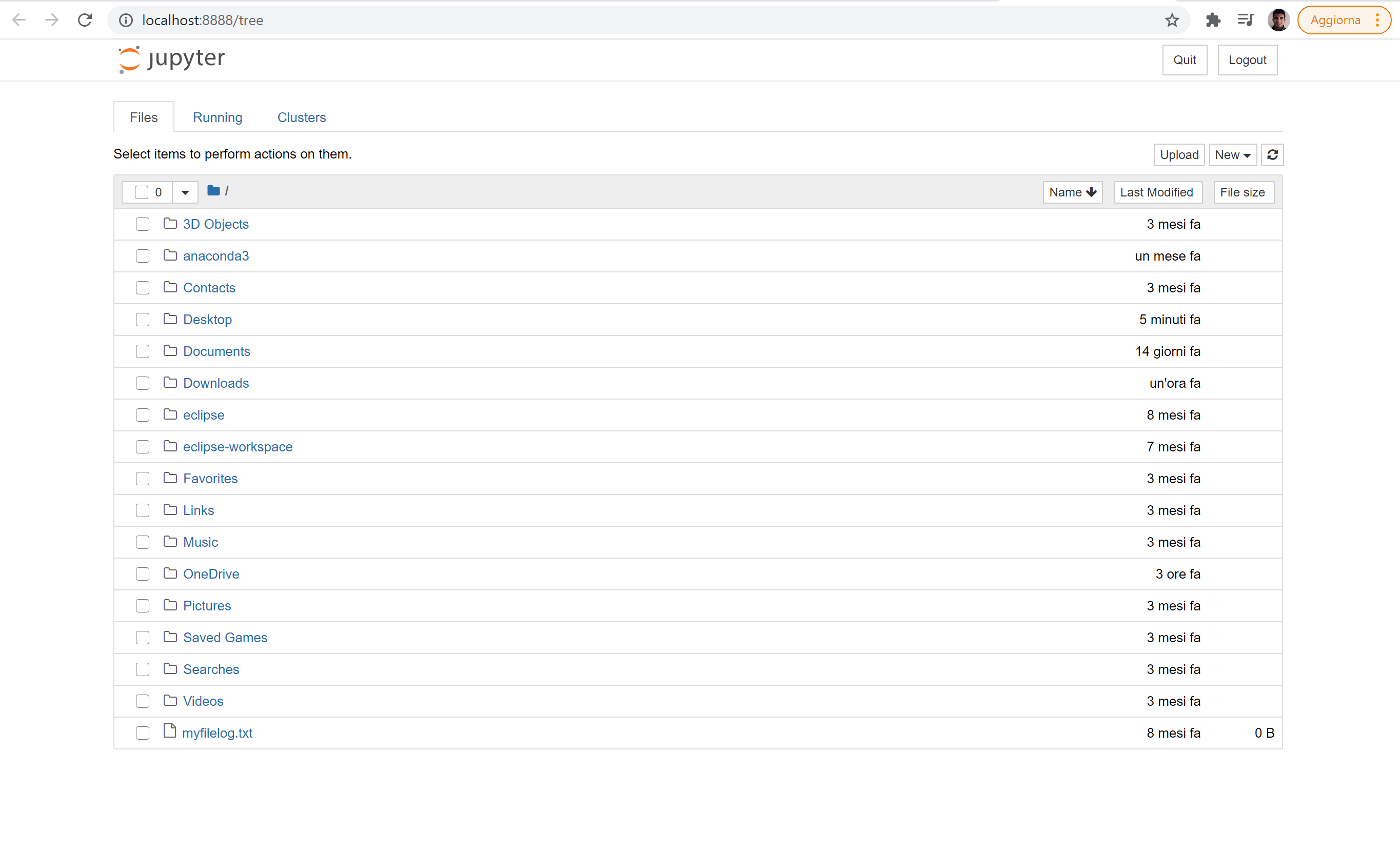


Extra : HOW TO USE THE CODE

First install anaconda  <https://www.anaconda.com/products/individual>

Second install from the command line open cv by executing : pip install opencv-python

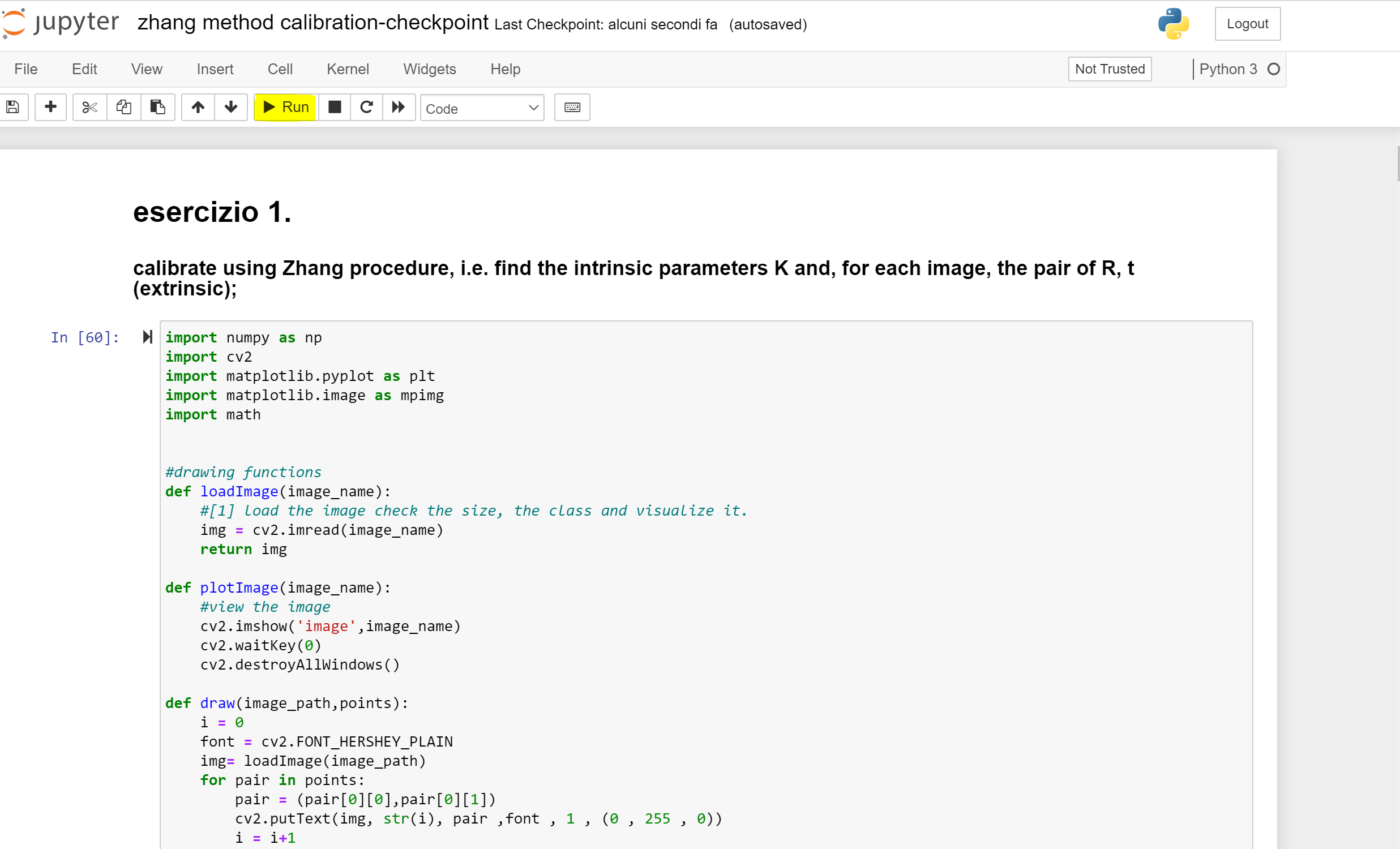
Third run jupyter notebook and the following view is shown in the browser



Now upload the project folder which contains the images and the notebook. First is suggested to create a folder called images and inside the folder upload the given images. See the image in the next page



Now just click on the notebook and run it :



Each point of the assignments is highlighted in the notebook

1. findChessboardCorners: https://docs.opencv.org/master/d9/d0c/group\_\_calib3d.html#ga93efa9b0aa890de240ca32b11253dd4a [↑](#footnote-ref-1)
2. linalg.svd(V) https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html [↑](#footnote-ref-2)
3. newton\_krylov: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton\_krylov.html [↑](#footnote-ref-3)
4. Undistort: https://docs.opencv.org/3.4/da/d54/group\_\_imgproc\_\_transform.html#ga69f2545a8b62a6b0fc2ee060dc30559d [↑](#footnote-ref-4)