Computer Vision project: Camera Calibration with Zhang's Method

Matteo Boi

Course of AA 2022-2023 - DSSC

1 Problem statement

Camera calibration is an important task in computer vision and image processing, as it allows to determine the intrinsic and extrinsic parameters of a camera from a set of observed images. The intrinsic parameters include the focal length, principal point, and distortion coefficients. The extrinsic parameters include the position and orientation of the camera in the world coordinate system. One widely used method for camera calibration is Zhang's method, which uses a checkerboard pattern to determine the intrinsic and extrinsic parameters of the camera. The goal of this report is to implement and evaluate the performance of Zhang's method for camera calibration using a set of observed images and compare the results with and without radial distortion correction.

2 Zhang's method for camera calibration

The procedure is implemented as described in [1]. The goal is to find the parameters of the camera, therefore the calibration matrix K (intrinsic) and, for each image, the pair of R, t (extrinsic) from a set of images of a checkerboard. One important assumption is that the images being used for calibration do not lie on the same plane. This is crucial because if the images were on the same plane, the algorithm would not be able to triangulate the 3D coordinates of the points on the checkerboard and therefore to accurately estimate the parameters of the camera. For this reason the checkerboard is positioned at a different orientation in every image.

Twenty images $640x480$ pixels are loaded and the corners and dimensions of the checkerboard pattern are detected using MATLAB *detectCheckerboardPoints()* function in every image, as shown in the following code:

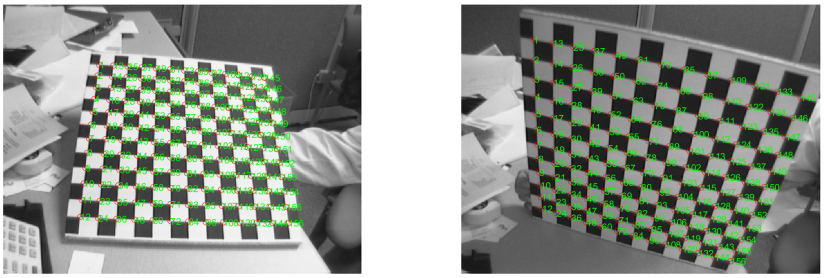

Figure 1: Detected checkerboard points on two of the twenty images.

```
iimage_full= 1:20;

for ii=1:length(iimage_full)
  imageFileName_full = "images" + filesep + "image" +
     num2str(iimage_full(ii)) + ".tif";

  imageData(ii).I = imread(imageFileName_full); %#ok

  [imageData(ii).XYpixel, boardsize] =
     detectCheckerboardPoints(imageData(ii).I, '
     PartialDetections', false); %#ok

end
```

ImageData is the MATLAB struct in which all the results will be stored, one row for each image. The detected points for two of the twenty images are shown in Fig. 1, where an index is assigned to each pair of coordinates (u_i, v_i) with i=1, ..., 156.

Assuming the real world reference system's origin in the first point detected and the X, Y axis parallel to the checkerboard, the corresponding real world coordinates (in mm) are computed given the size of the checkerboard (*boardsize* = $[13, 14]$) and the length of a square (*squaresize* = $30mm$).

```
for ii=1:length(imageData)
    XYpixel = imageData(ii).XYpixel;
    clear Xmm Ymm

    for jj=1:length(XYpixel)
```

```
        [row, col] = ind2sub(boardsize-1, jj);

        Xmm = (col-1) * squaresize;
        Ymm = (row-1) * squaresize;

        imageData(ii).XYmm(jj, :) = [ Xmm Ymm ];
    end
end
```

The method is based on the use of homographies. An homography is a projective transformation of a plane, which maps the points in one plane to the corresponding points in another plane. It is a non-linear map that can become linear by adopting homogeneous coordinates of the detected points $m_i = [x_i \ y_i \ z_i \ 1]^T$ and $m'_i = [u_i \ v_i \ 1]^T$ in the real world plane and image plane respectively. The homography can be represented by a 3x3 matrix, which can be computed given a minimum of four pairs of original and projected points via the Direct Linear Transform (DLT) method. The homography matrix is computed in the following code for every image:

```
for ii=1:length(imageData)
    XYpixel = imageData(ii).XYpixel;
    XYmm=imageData(ii).XYmm;
    A=[];
    b=[];
    for jj=1:length(XYpixel)
        Xpixel=XYpixel(jj,1);
        Ypixel=XYpixel(jj,2);
        Xmm=XYmm(jj,1);
        Ymm=XYmm(jj,2);

        m=[Xmm; Ymm; 1];
        O=[0;0;0];

        A=[A; m' O' -Xpixel*m' ; O' m' -Ypixel*m']; %#
            ok
        b=[b;0;0]; %#ok
    end

    [~,~,V]=svd(A);
    h=V(:,end);

    imageData(ii).H = reshape(h,[3 3])';
end
```

Through the computed homographies is now possible to get a first estimate of the calibration matrix K, that describes the intrinsic parameters (focal lengths in $x-$ and $y-$direction, center, skew angle) of the camera used. At least three different planes, and therefore homographies, are needed to be able to compute K, but in practice more planes mean better accuracy. First, the matrix V is constructed, a $[2N\text{x}6]$ matrix with N the numbers of images and the elements computed by the function *calculate_vij()* of the form

$$v_{iJ} = \begin{bmatrix} H_{i1}H_{j1} \\ H_{i1}H_{j2} + H_{i2}H_{j2} \\ H_{i2}H_{j2} \\ H_{i3}H_{j1} + H_{i1}H_{j3} \\ H_{i3}H_{j2} + H_{i2}H_{j3} \\ H_{i3}H_{j3} \end{bmatrix}$$

from every image's homography H.

```
V = [];
for ii = 1:length(imageData)
    v12 = calculate_vij(1,2, imageData(ii).H);
    v11 = calculate_vij(1,1, imageData(ii).H);
    v22 = calculate_vij(2,2, imageData(ii).H);

    V = [V; v12'; (v11 - v22)']; %#ok
end
```

Then the calibration matrix K is computed by singular value decomposition of V and the proper scale for K is obtained by imposing $K_{33} = 1$.

```
[~,~,S]=svd(V);
b = S(:,end);
B = [b(1) b(2) b(4); b(2) b(3) b(5); b(4) b(5) b(6)];

L=chol(B, "lower");
K = (L')^(-1);
K = K/K(3,3);
```

As last step, the function *compute_ExtrinsicParams()* is invoked to compute, given the matrix K and the homographies, the extrinsic parameters of each image (i.e. the rotation matrix R and translation vector t). These are the last two ingredients to be able to compute the perspective projection matrix P:

```
for ii=1:length(imageData)

    H = imageData(ii).H./imageData(ii).H(3,3);

    h1 = H(:,1);
    h2 = H(:,2);
    h3 = H(:,3);

    lambda = 1 / norm( K^-1*h1 );

    r1 = lambda * K^-1 * h1;
    r2 = lambda * K^-1 * h2;
    r3 = cross( r1, r2 );

    t = lambda * K^-1 * h3;
    R = [ r1 r2 r3 ];

    [U,~,V] = svd(R);
    R_prime = U*V';

    imageData(ii).R = R_prime;
    imageData(ii).t = t;
    imageData(ii).P = K*[R_prime t];
end
```

Obviously, representing both intrinsic and extrinsic parameters, P is characteristic for each image. The detailed mathematical calculation can be found in the original article [1].

One important observation needs to be made here, given the fact that homographies are computed using only points independent by the z-axis, the sign of the final matrix is dictated by the output of the singular value decomposition. This difference in the sign of the homography matrices is carried on during all the computations, but it makes an enormous difference in the meaning that the matrix R and therefore P assumes. In particular, it dictates if the z-axis of that particular reference system is in the same direction of the optical axis (going from the camera to the object) or the opposite. One possible solution and the one chosen here is to adopt an unanimous direction to divide each homography matrix by its last element. This subtlety doesn't show until, for example, one projects a solid object on the images, as done in the next section.

3 Cylinder superimposition

To check the goodness of the results obtained the next step is to project a cylinder on the images. Any $3D$ point projection on the images is possible

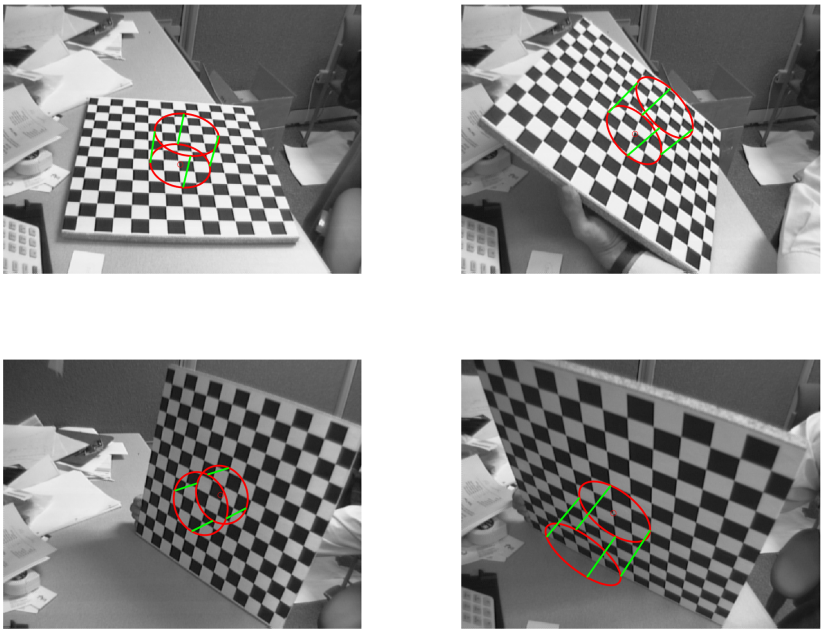

Figure 2: Superimposition of a cylinder in 4 of the 20 images used for calibration.

thanks to the previously computed perspective projection matrices. In fact each P represents the mapping between the $3D$ world and the image plane. It is a $3x4$ matrix that takes the homogeneous coordinates of a 3D point m and maps it to the homogeneous coordinates of the corresponding 2D image point m':

$$m' = Pm$$

The obtained m' will be of the form $[wu\ wv\ w]^T$ and the actual pixel coordinates (u, v) can be found dividing for w. One has to keep in mind that given the world reference frame used, the direction of the z-axis is from the camera to the checkerboard, meaning that an object has z-coordinates negative if it is placed on top of the checkerboard in this particular reference frame. The cylinder is obtained by generating 1000 points on a circle centered in a central pixel of the checkerboard, with a radius of $60mm$ and a height of $80mm$. The corresponding implementation of the MATLAB function is not shown here for space reasons, but can be found in Sec. 8. The results obtained on 4 of the images are shown in Fig. 2.

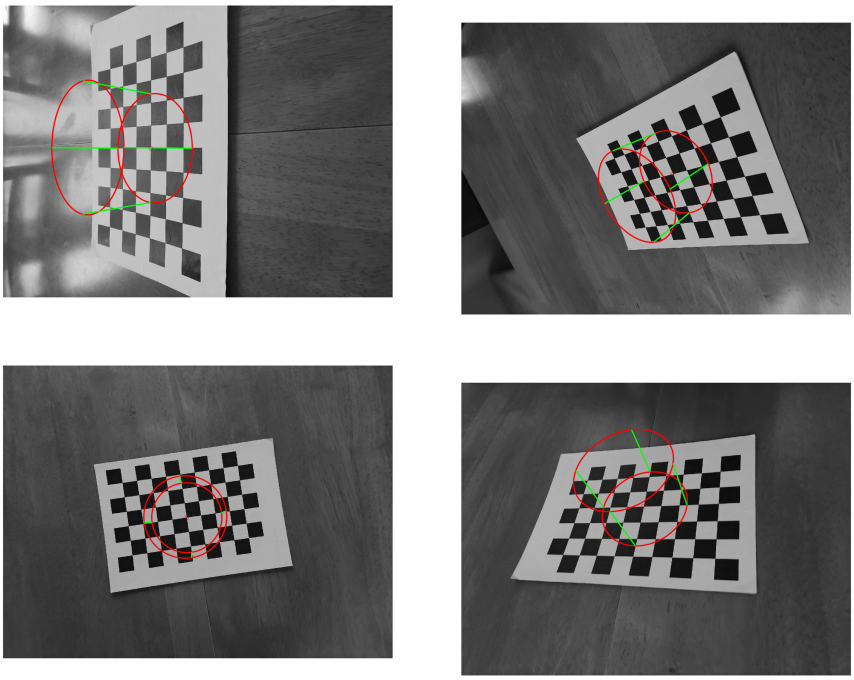

Figure 3: Superimposition of a cylinder in 4 of the 20 phone images used for calibration.

4 Phone camera calibration

In this section, the same procedures explained in sections 2, 3 are applied to a set of 20 images taken by a Xiaomi Android smartphone with a 48 Mpx camera. The checkerboard used this time is smaller than the previous one, with a pattern of $7x10$ squares and a squaresize of $25mm$. Firstly, the images ($3000x4000$ pixels) are loaded and converted from color to gray-scale through the MATLAB *rgb2gray()* function. Zhang's camera calibration procedure is used to calculate camera's intrinsic and extrinsic parameters and the perspective projection matrices are then used to superimpose a cylinder of the same type as before. Results are shown in Fig. 3 for four of the twenty images.

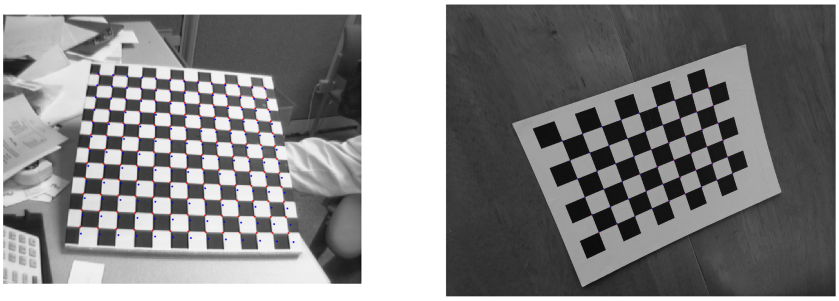

Figure 4: Comparison between detected and reprojected points in two images of the two sets.

5 Reprojection Error

The reprojection error is a measure of the difference between the observed image coordinates and the predicted image coordinates from the estimated intrinsic and extrinsic parameters of the camera. It is the sum of squared differences between the observed image points and the projected 3D points onto the image plane. Therefore it can be used to assess the quality of the camera calibration, as lower values indicate a better fit between the observed and predicted image points. For a set of $3D$ points $m = \{m_{(i)}, i = 1, ..., 156\}$ and matching image coordinates $(u, v) = \{(u_{(i)}, v_{(i)}), i = 1, ..., 156\}$, the error is calculated as:

$$\epsilon(P) = \sum_{i=1}^{n} \left(\frac{p_1^T m_{(i)}}{p_3^T m_{(i)}} - u_{(i)} \right)^2 + \left(\frac{p_2^T m_{(i)}}{p_3^T m_{(i)}} - v_{(i)} \right)^2$$

An interesting result is given by the comparison between the first set and second set of images. Even if the error cannot be compared numerically, given the different amount of detected points in the two sets of images, a visual comparison is possible and is shown in Fig. 4. While in the first case the error is evident, this is not the case for the second image, where the points are almost aligned. This is due a stronger radial distortion caused by the first camera, problem that will be addressed in the next section.

6 Radial distortion compensation

The goal of this section is to describe the procedure of radial distortion compensation on the first set of images. Radial distortion is a type of image distortion that occurs when light entering a camera lens is not perfectly collimated, causing points at different distances from the lens to be imaged differently. As a result, points that are farther away from the center of the image are stretched or compressed compared to points near the center. This leads to images with

a characteristic "barrel" or "pincushion" shape. The amount of radial distortion can be modeled mathematically and the model can be used to correct the distortion. This correction is particularly important in computer vision applications where accurate measurements are critical. Radial distortion can have a significant impact on the accuracy of camera calibration and pose estimation algorithms, so it is essential to model and correct for it in these applications.

The basic idea of the model is that a point in the real world is transformed through the matrix P in a set of undistorted image coordinates (u, v) to which a radial distortion is apllied to obtain a pair of distorted image coordinates $(\hat{u}, \hat{v})$. This distortion can be modeled through two radial distortion coefficients k_1, k_2 in a pair of equations of the form:

$$\begin{cases} \hat{u} = (u - u_0)(1 + k_1 r_d^2 + k_2 r_d^4) + u_0 \\ \hat{v} = (v - v_0)(1 + k_1 r_d^2 + k_2 r_d^4) + v_0 \end{cases} \tag{1}$$

$$r_d^2 = \left(\frac{u - u_0}{\alpha_u}\right)^2 + \left(\frac{v - v_0}{\alpha_v}\right)^2$$

with (u_0, v_0) the coordinates (in pixel) of the principal point, α_u, α_v the product of the reciprocal of the pixel size along u and v and the focal length respectively. All can be obtained from the calibration matrix K. A first estimation of k_1, k_2 can be obtained by using the detected pixel coordinates as distorted coordinates $(\hat{u}, \hat{v})$ and the reprojected coordinates through P as undistorted coordinates (u, v). Then defining the normalized coordinates as:

$$x = \frac{u - u_0}{\alpha_u} \qquad y = \frac{v - v_0}{\alpha_v}$$

a non-linear system is obtained from Eq. 1 that can be solved by a root-finding algorithm like Newton's method. The solution will be a new set of normalized undistorted coordinates from which a corrected version of H, K, R, t and P keeping into account radial distortion can be obtained. Then the process can be iterated until convergence of k_1, k_2 and P. The code for the process is shown below:

```
n = number_of_points_per_image
N = number_of_images
k0 = [0; 0]

for jj = 1:maxIter

    k = estimate_radialDistortionCoeffs(imageData, K,
        n);

    for kk = 1:N
        uv = imageData(kk).P * [imageData(kk).XYmm
            (:,1)';
```

```
            imageData(kk).XYmm(:,2)'; zeros(1,n);
               ones(1,n)];
        u = uv(1,:)./uv(3,:);
        v = uv(2,:)./uv(3,:);

        u_hat = imageData(kk).XYpixel(:,1)';
        v_hat = imageData(kk).XYpixel(:,2)';

        x = (u - u0)/alpha_u;
        y = (v - v0)/alpha_v;
        x_hat = (u_hat - u0)/alpha_u;
        y_hat = (v_hat - v0)/alpha_v;

        [x_new, y_new] = NewtonsMethod(x_hat, y_hat, x
           , y, k, tol, maxIter);
        u = x_new*alpha_u + u0;
        v = y_new*alpha_v + v0;

    end

    imageData = compute_Homographies(imageData, true);
    K = compute_IntrinsicParams(imageData);
    imageData = compute_ExtrinsicParams(imageData, K);

    if P_convCheck < tol || k_convCheck < tol
        break
    end
end
```

Again for space reasons, the functions *estimate_radialDistortionCoeffs()* and *NewtonsMethod()* can be found in Sec. 8.

7 Results

To finish, we can finally compared the reprojection error introduced in Sec. 5 obtained for the first set of images without and with radial distortion correction (RDC). The comparison is shown both in Fig. 5 where detected and reprojected points are plotted on the images and in Table 1, where the actual values of the reprojection error are reported for every image.

As shown by the table, there is clear drop in the error, going from an average of 5300.6 to 263.5. Therefore, a more accurate estimation of the true parameters of the camera has been found.

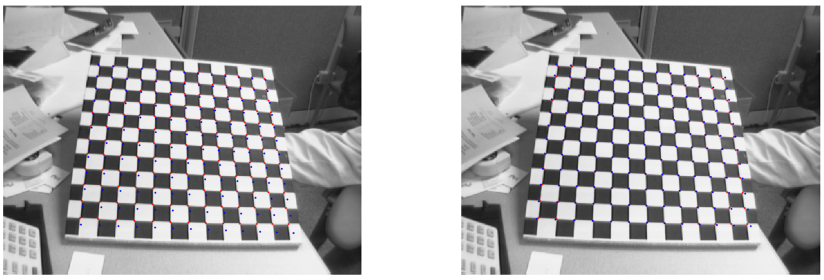

Figure 5: Comparison between detected and reprojected points in the same image without (left) and with (right) radial distortion correction.

	Without RDC	**With RDC**
image1	13997.205	21.789
image2	9172.554	105.782
image3	7112.369	131.311
image4	8968.398	617.366
image5	4464.627	432.369
image6	3248.541	258.089
image7	2276.491	96.524
image8	1747.089	135.999
image9	19805.657	529.573
image10	3788.629	352.684
image11	1719.555	197.408
image12	1265.401	158.899
image13	1557.079	179.996
image14	2556.671	248.590
image15	720.471	462.081
image16	9280.182	223.365
image17	7417.311	332.492
image18	671.658	415.245
image19	3949.767	212.508
image20	2292.555	158.679

Table 1: Comparison of the reprojection error obtained with and without radial distortion compensation on the first set of images.

8 Appendice

- Cylinder plot MATLAB code:

```
n = 1000;
R = 60;
height = 80;
center = imageData(1).XYmm(n/2 + 1,:);
x0 = center(1);
y0 = center(2);

it = linspace(1,n,n);
X = cos(2*pi/n*it)*R + x0;
Y = sin(2*pi/n*it)*R + y0;

center = [x0; y0; 0; 1];
coords_low = [X; Y; zeros(1,n); ones(1,length(X))
   ];
coords_high = [X; Y; -ones(1,n)*height; ones(1,
   length(X))];

for ii=1:length(imageData)
    figure
    imshow(imageData(ii).I, 'InitialMagnification'
       , 200);
    hold on


    proj_hom_low = imageData(ii).P*coords_low;
    proj_hom_high = imageData(ii).P*coords_high;
    proj_center = imageData(ii).P*center;
    proj_center = [proj_center(1)/proj_center(3);
       proj_center(2)/proj_center(3)];

    proj_low = [proj_hom_low(1, :)./proj_hom_low
       (3, :); proj_hom_low(2, :)./proj_hom_low(3,
        :)];
    proj_high = [proj_hom_high(1, :)./
       proj_hom_high(3, :); proj_hom_high(2, :)./
       proj_hom_high(3, :)];

    plot(proj_low(1, :), proj_low(2, :), 'r', '
       LineWidth', 2);
    plot(proj_high(1, :), proj_high(2, :), 'r', '
       LineWidth', 2);
    plot([proj_low(1,1) proj_high(1,1)], [proj_low
```

```
        (2,1) proj_high(2,1)], 'g', 'LineWidth', 2)
    plot([proj_low(1,n/2) proj_high(1,n/2)], [
        proj_low(2,n/2) proj_high(2,n/2)], 'g', '
        LineWidth', 2)
    plot([proj_low(1,n/4) proj_high(1,n/4)], [
        proj_low(2,n/4) proj_high(2,n/4)], 'g', '
        LineWidth', 2)
    plot([proj_low(1,n/4*3) proj_high(1,n/4*3)], [
        proj_low(2,n/4*3) proj_high(2,n/4*3)], 'g',
         'LineWidth', 2)
    plot(proj_center(1), proj_center(2), 'or')
end
```

- *estimate_radialDistortionCoeffs()* MATLAB function:

```
function k = estimate_radialDistortionCoeffs(
    imageData, K, n)
    A = [];
    b = [];
    for kk = 1:length(imageData)

        uv = imageData(kk).P * [imageData(kk).XYmm
            (:,1)'; imageData(kk).XYmm(:,2)'; zeros
            (1,n); ones(1,n)];
        u = uv(1,:)./uv(3,:);
        v = uv(2,:)./uv(3,:);

        u_hat = imageData(kk).XYpixel(:,1)';
        v_hat = imageData(kk).XYpixel(:,2)';

        u0 = K(1,3);
        v0 = K(2,3);
        alpha_u = K(1,1);
        alpha_v = K(2,2)*sin(acot(K(1,2)/alpha_u))
            ;

        x = (u - u0)/alpha_u;
        y = (v - v0)/alpha_v;

        rd_sqr = x.^2 + y.^2;

        for ii = 1:n
            A_i = [(u(ii) - u0)*rd_sqr(ii), (u(ii)
                - u0)*rd_sqr(ii)^2;
```

```
                    (v(ii) - v0)*rd_sqr(ii), (v(ii)
                        - v0)*rd_sqr(ii)^2];
            b_i = [u_hat(ii) - u(ii); v_hat(ii) -
                v(ii)];

            A = [A; A_i]; %#ok
            b = [b; b_i]; %#ok
        end

    end

    k = (A'*A)^-1*A'*b;
end
```

- *Newton's method function:*

```
function newxy = NewtonsMethod(x_hat, y_hat, x, y,
    k, tol, maxIter)

    for ll = 1:maxIter
        res = f(x_hat, y_hat, x, y, k(1), k(2));

        delta = [];
        for ii = 1:length(x)
            J = compute_jacobian(x(ii), y(ii), k
                (1), k(2));
            delta_i = -J^-1 * res(:,ii);
            delta = [delta, delta_i]; %#ok
        end

        x = x + delta(1,:);
        y = y + delta(2,:);

        if (norm(delta) < tol)
            break
        end
    end
    newxy = [x; y];

end
```

References

[1] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(11):1330–1334, nov 2000.