Josh Holloway
ASU – Fall 2018 – Computer Vision EEE 598
ID: 1210205565
Assignment 3

Pre-requisite reading

i. 2D-Projective Space
ii. Computation of Homography by DLT Method
iii. Camera models
iv. Computation of Camera Projection Matrix P via DLT Method

- I pledge to have read the reading material in the assignment.

Practical Assignment

Calibration Using a 3D-Object

Par 1. Intrinsic Parameter Computation

Par 2. Intrinsic and Extrinsic Parameter Computation

Solution Methodology: I implement my solution in C++ using OpenCV after prototype a MATLAB implementation to provide a golden reference to compare against.

Solution Methodology for Part 1:

- $\boldsymbol{X}_i$ is the i^{th} provided 3D-world points in non-homogeneous coordinates in the same coordinate system as the camera.
- $\boldsymbol{x}_i$ is the i^{th} provided 2D-image point in non-homogeneous coordinates

Step 1: Apply the following transformation to $\boldsymbol{X}_i$ and $\boldsymbol{x}_i$

$$\boldsymbol{X}_i = \begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} \rightarrow \widehat{\boldsymbol{X}}_i = \begin{bmatrix} X_i/Z_i \\ Y_i/Z_i \\ 1 \end{bmatrix}$$

$$\boldsymbol{x}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \rightarrow \widehat{\boldsymbol{x}}_i = \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Step 2: Set up the following relationship:

$$\widehat{\boldsymbol{x}}_i = K\widehat{\boldsymbol{X}}_i$$

where $K = \begin{bmatrix} \alpha_x & s & p_x \\ & \alpha_x & p_y \\ & & 1 \end{bmatrix}$.

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \left.\begin{bmatrix} \alpha_x & s & p_x \\ & \alpha_x & p_y \\ & & 1 \end{bmatrix} \begin{bmatrix} X_i/Z_i \\ Y_i/Z_i \\ 1 \end{bmatrix}\right\} \Rightarrow \begin{cases} x_i = \left(\frac{X_i}{Z_i}\right)\alpha_x + \left(\frac{Y_i}{Z_i}\right)s + p_x \\ y_i = \left(\frac{Y_i}{Z_i}\right)\alpha_y + p_y \end{cases}$$

By stacking up sets of points columns for all N-points, we have

$$\underbrace{\widehat{\boldsymbol{x}}}_{(3\times 28)} = \underbrace{K'}_{(3\times 3)} \underbrace{\widehat{\boldsymbol{X}}}_{(3\times 28)}$$

Our goal is to solve for K'.
We can utilize the Moore-Penrose pseudo-inverse:

$$A^{\dagger} = (\mathrm{A}^T\mathrm{A})^{-1}\mathrm{A}^T$$

I used the SVD method to actually compute the pseudo- inverse $\widehat{\boldsymbol{X}}^{\dagger}$.

$$\widehat{\boldsymbol{x}}\widehat{\boldsymbol{X}}^{\dagger} = K'\widehat{\boldsymbol{X}}\widehat{\boldsymbol{X}}^{\dagger}$$

$$\underbrace{\widehat{\boldsymbol{x}}}_{(3\times 28)} \underbrace{\widehat{\boldsymbol{X}}^{\dagger}}_{(28\times 3)} = \underbrace{K'}_{(3\times 3)}$$

The obtained K-matrix found through this method was:
K:
[396.2827735090135, -6.259946786553686, 882.1424071221763;
0, 456.4725206426237, 673.1095365217135;
0, 0, 1]

Main source-code for part-1 – refer to provided full source code for details**:**

```
//================================================================
void part_1_method2()
{
	// Non-homogeneous coordinates
	Mat x = copy_2_mat(pts_2d); // 2D-points in image plane in reference to camera coordinate system
	Mat X = copy_2_mat(pts_3d); // 3D-world points world    in reference to camera coordinate system

	// Number of points
	const int N = x.cols;

	// Step 1: Un-homogenize the 3D points to get 2D points - result is 28 x 2
	Mat XX = from_homo(X);

	// Step 2: Homogenize the 2D points to get 3D points by appending row of ones
	Mat x_ = to_homo(x);
	Mat XX_ = to_homo(XX);

	// Step 3: Compute the Moore-Penrose Psuedo-inverse to estimate K
	Mat XX_inv(N, 3, CV_64FC1);
	invert(XX_, XX_inv, cv::DECOMP_SVD);

	// Step 4: Solve for P with computed Moore-Penrose Psuedo-Inverse
	auto K = x_ * XX_inv;

	// Step 5: Look at the resulting K matrix
	cout << "K:\n" << K;
}
```

Solution Methodology for Part 1:

$\boldsymbol{X}_i$ is the i^{th} provided 3D-world points in non-homogeneous coordinates in the same coordinate system as the camera

Strategy:

Step 1. Form a matrix A out of the 3D<->2D point correspondences provided according to H&Z eq. 7.2

Step 2. Apply SVD on A

Step 3. Extract right-most col-vector of V matrix produced from [U, SIGMA, V] = svd(A)

- Because equation 7.2 in H&Z is a homogeneous linear system we are trying to find the elements in the nullspace – i.e. all vector that map to zero through the matrix A.
- The elements in the nullspace correspond to any zero valued singular values.
- Since there is only one zero valued singular value (only the lower right diagonal element in the sigma matrix from the SVD) the nullspace of A is one-dimensional.
- This zero valued singular value corresponds to the right most column of the matrix V, which forms a basis for the nullspace of A.
- This right-most column is the vector we are looking for and contains all the parameters of the projection matrix.

Step 4. Reshape this vector to a 3x4 matrix - this is projection matrix P

Step 5. Determine how good our estimation of P was by projecting the set of 3D points into the image plane through our computed matrix P

Step 6. Convert the computed points in step 5 into their in-homogeneous form by dividing by the third element of each point.

Step 7. Observe the error between our computed 2D points from step 6 to the provided 2D points via a geometric error function

Step 8. We can perform a non-linear optimization technique at this point to minimize the geometric error from step 7. We can use Levenberg–Marquardt since the projection is non-linear due to that division by z to achieve a perspective projection

- I was unable to complete this part, however I am using this technique for part of my final project.

Step 9. Once a small geometric error has been achieved we take the corresponding matrix of P and apply the technique described in H&Z section 6.2.4 "Decomposition of the camera matrix" to decompose P into P = K * [R t]

DLT description:

The culminating effect of all the camera parameters is the mapping from 3D world coordinate to 2D image plane image coordinate:

$$\boldsymbol{x} \cong P\boldsymbol{X}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \cong \begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Camera Calibration problem definition (re-sectioning): Find P given a set of homogeneous points $\boldsymbol{x}_i \in \mathbb{R}^3$ and corresponding homogeneous world points $\boldsymbol{X}_i \in \mathbb{R}^4$.

$$\begin{bmatrix} wx_i \\ wy_i \\ w \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}$$

$$\begin{aligned} wx_i &= m_{00}X_i + m_{01}Y_i + m_{02}Z_i + m_{03} \\ wy_i &= m_{10}X_i + m_{11}Y_i + m_{12}Z_i + m_{13} \\ w &= m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23} \end{aligned}$$

$$\left.\begin{aligned} x_i &= (1/w)(m_{00}X_i + m_{01}Y_i + m_{02}Z_i + m_{03}) \\ y_i &= (1/w)(m_{10}X_i + m_{11}Y_i + m_{12}Z_i + m_{13}) \\ w &= m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23} \end{aligned}\right\} \quad \Rightarrow \quad \begin{cases} x_i = \dfrac{m_{00}X_i + m_{01}Y_i + m_{02}Z_i + m_{03}}{m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23}} \\ y_i = \dfrac{m_{10}X_i + m_{11}Y_i + m_{12}Z_i + m_{13}}{m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23}} \end{cases}$$

The i^{th}-point corresponds to a pair of equations.

$$\begin{aligned} x_i &= \frac{m_{00}X_i + m_{01}Y_i + m_{02}Z_i + m_{03}}{m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23}} \\ y_i &= \frac{m_{10}X_i + m_{11}Y_i + m_{12}Z_i + m_{13}}{m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23}} \end{aligned}$$

$$\begin{aligned} (m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23})x_i &= m_{00}X_i + m_{01}Y_i + m_{02}Z_i + m_{03} \\ (m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23})y_i &= m_{10}X_i + m_{11}Y_i + m_{12}Z_i + m_{13} \end{aligned}$$

$$\begin{aligned} (m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23})x_i - (m_{00}X_i + m_{01}Y_i + m_{02}Z_i + m_{03}) &= 0 \\ (m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23})y_i - (m_{10}X_i + m_{11}Y_i + m_{12}Z_i + m_{13}) &= 0 \end{aligned}$$

$$\begin{aligned} m_{20}X_ix_i + m_{21}Y_ix_i + m_{22}Z_ix_i + m_{23}x_i - m_{00}X_i - m_{01}Y_i - m_{02}Z_i - m_{03} &= 0 \\ m_{20}X_iy_i + m_{21}Y_iy_i + m_{22}Z_iy_i + m_{23}y_i - m_{10}X_i - m_{11}Y_i - m_{12}Z_i - m_{13} &= 0 \end{aligned}$$

$$\begin{bmatrix} X_i x_i & Y_i x_i & Z_i x_i & x_i & -X_i & -Y_i & -Z_i & -1 & 0 & 0 & 0 & 0 \\ X_i y_i & Y_i y_i & Z_i y_i & y_i & 0 & 0 & 0 & 0 & -X_i & -Y_i & -Z_i & -1 \end{bmatrix} \begin{bmatrix} m_{20} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{00} \\ m_{01} \\ m_{02} \\ m_{03} \\ m_{10} \\ m_{11} \\ m_{12} \\ m_{13} \end{bmatrix} = \mathbf{0}$$

Moving variables around, and using vector notation, we have the form of equation 7.2 in Hartley and Zissserman:

$$\begin{bmatrix} \mathbf{0}^T & -\boldsymbol{X}_i^T & y_i \boldsymbol{X}_i^T \\ \boldsymbol{X}_i^T & \mathbf{0}^T & -x_i \boldsymbol{X}_i^T \end{bmatrix} \tilde{P} = \mathbf{0}$$

By stacking up points in this fashion, we have enough equation to solve for $\widetilde{\boldsymbol{P}}$ with 5 and a half points, but six or more is best.

$$\begin{bmatrix} \mathbf{0}^T & -\boldsymbol{X}_1^T & y_1 \boldsymbol{X}_1^T \\ \boldsymbol{X}_1^T & \mathbf{0}^T & -x_1 \boldsymbol{X}_1^T \\ & \vdots & \\ \mathbf{0}^T & -\boldsymbol{X}_N^T & y_N \boldsymbol{X}_N^T \\ \boldsymbol{X}_N^T & \mathbf{0}^T & -x_N \boldsymbol{X}_N^T \end{bmatrix} \tilde{P} = \mathbf{0} \in \mathbb{R}^{2N}$$

$$A\tilde{P} = \mathbf{0} \quad \Rightarrow \quad \text{want to find nullspace of } A$$

$$\mathrm{A} = U\Sigma V^T$$

$$AV = UD$$

$$\underbrace{A}_{(2N\times 12)} \underbrace{[\boldsymbol{v}_1 \quad \cdots \quad \boldsymbol{v}_{12}]}_{(12\times 12)} = \underbrace{[\boldsymbol{u}_1 \quad \cdots \quad \boldsymbol{u}_{12}]}_{(2N\times 12)} \underbrace{\mathrm{diag}(\sigma_1, \dots, \sigma_{12})}_{(12\times 12)}$$

$$A\boldsymbol{v}_i = \sigma_i \boldsymbol{u}_i, \qquad i = 1,..,12$$

The $N - \mathrm{rank}(A)$ columns of V form an orthonormal basis set for A [Strang: Linear Algebra and its Applications – section 6.3]

$$\mathcal{N}(A) = \left[v_1 \quad \cdots \quad \underbrace{\boldsymbol{v}_{r+1} \quad \cdots \quad \boldsymbol{v}_N}_{r:N} \right] \quad \Rightarrow \quad AV_{r:N} = \mathbf{0}$$

For our A-matrix we have only $\sigma_{12} = 0$. Therefore $\boldsymbol{v}_{12} = \tilde{P}$.

$\tilde{P}$ contains our 12 elements we seek. We obtain the projection matrix P by reshaping $\tilde{P}$ to be of dimension 3×12, unrolling in row-major order.

Please enter the values of P, K, R and t in your report.

Answer the following questions:

Describe how the parameters K, R and t were obtained from the projection matrix P? [Hints: Let us write $P = [M\ p_4]$, where M is the first 3*3 sub-matrix of P. Now, the expression P = K[R t] implies that M = KR, where K is an upper triangular matrix and R is a rotation matrix. This is an instance of a very famous matrix decomposition.]

We perform a QR-decomposition as described in section 6.2.4 of Hartley and Zisserman.

We can find the camera center through analyzing the determinant of combinations of column vectors of the projection matrix. We then find the camera orientation and internal parameters by decomposing K and R, where K is the camera calibration matrix containing the intrinsic parameters and M is the first three columns of the projection matrix P.

This is where the QR-decomposition is used [what follows is paraphrasing from Hartley and Ziszerman]. The decomposition is used to form an upper-triangular (U) and orthogonal (Q) matrix. The decomposition of M=KR is achieved through this method, where R is the rotation matrix. The translation vector is further computed in this process.

The result is P = K*[R t], where K is the intrinsic 3x3 matrix, R is the extrinsic rotation matrix, and t is the 3x1 translation vector. R and t form the extrinsic parameters.

By projecting the originally provided 3D world points into the image plane through our estimate of the projection matrix, we obtain following reprojections seen in Figure 1.

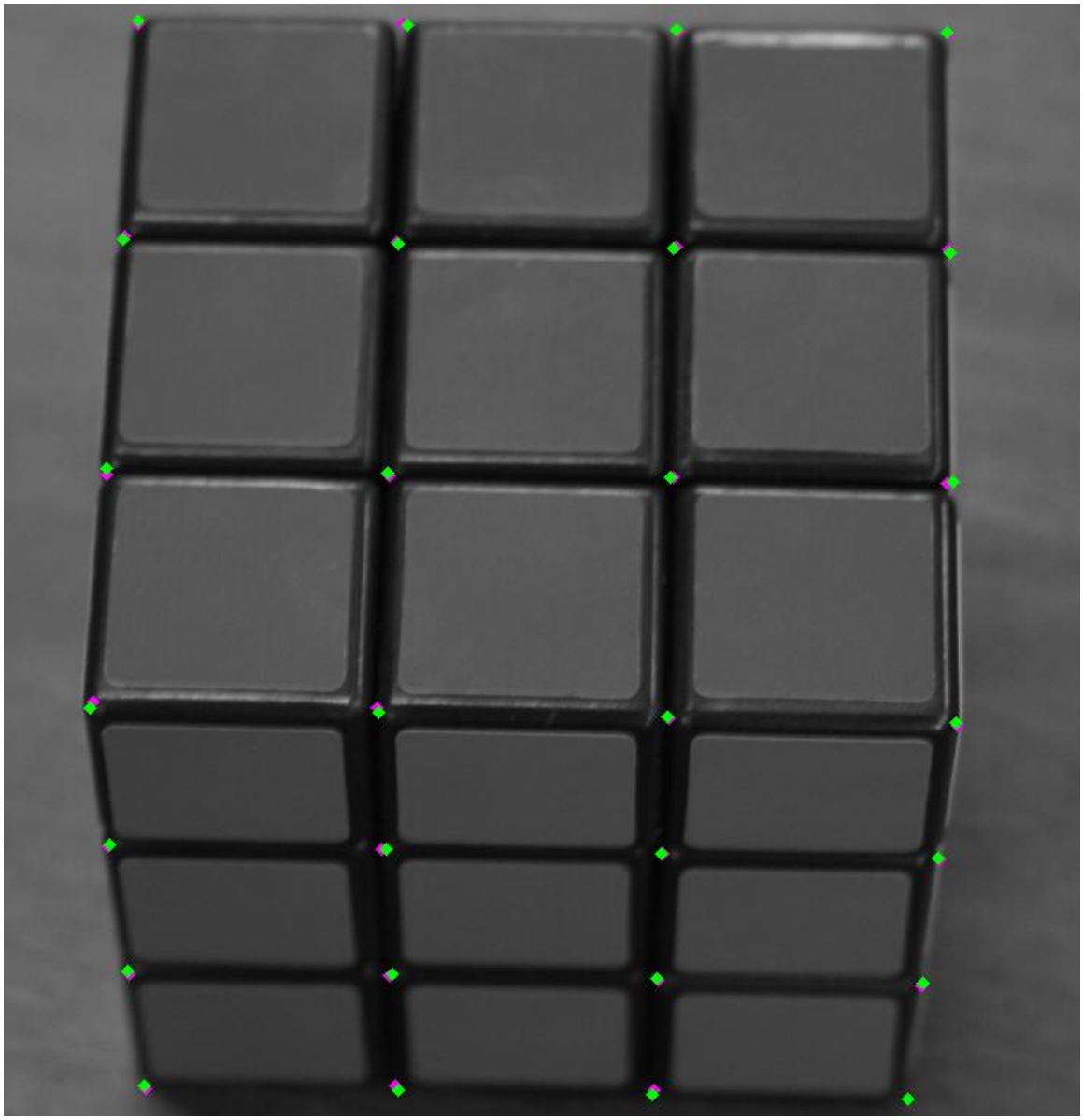

Figure 1: Green diamonds are the re-projected points and magenta are the original 2D points.

The reprojection error, as quantified via an L2-norm of the difference between the points, is given as follows.

L2 error = 0.441317

```
//====================================================================
void part_2()
{
	// Non-homogeneous coordinates
	Mat x = copy_2_mat(pts_2d); // 2D-points in image plane
	Mat X = copy_2_mat(pts_3d); // 3D-world points world

	// Map to homogeneous coordinates
	Mat x_ = to_homo(x);
	Mat X_ = to_homo(X);

	// Estimate projection matrix P
	Mat P = calibrate(x_, X_);

	// Re-project the 3D-world points into the 2D-image plane
	auto x_reproject_ = P * X_;
	//cout << "x_1_reproject" << x_1_reproject_;

	// Map from homo to non-homo coordinates:
	auto x_reproject = from_homo(x_reproject_);
	//cout << "x_reproject" << x_reproject;

	/// Compute re-projection error

	// Get into form that matches notation from LM-implementation notes:
	vector<double> x_u, x_v;      // row-vec
	vector<double> x_u_hat, x_v_hat;  // row-vec
	for (int j = 0; j < N; ++j) // itterate across cols
	{
		x_u.push_back(x.at<double>(0, j));
		x_v.push_back(x.at<double>(1, j));

		x_u_hat.push_back(x_reproject.at<double>(0, j));
		x_v_hat.push_back(x_reproject.at<double>(1, j));
	}

	// Form d-vector
	vector<double> d;
	double l2 = 0;
	for (int j = 0; j < N; ++j)
	{
		auto d_jx = x_u[j] - x_u_hat[j];
		auto d_jy = x_v[j] - x_v_hat[j];

		d.push_back(d_jx);
		d.push_back(d_jy);

		l2 += d_jx * d_jx + d_jy * d_jy;
	}
	l2 = sqrt(l2) / static_cast<double>(N);

	// Perform inner product with self, sqrt, and divide by double casted N
	auto l2_v2 = sqrt(std::inner_product(d.begin(), d.end(), d.begin(), 0)) / static_cast<double>(N);
	cout << "\nl2-norm with stl (something is wrong though) = " << l2_v2 << "\n";
	cout << "l2 error manual (correct) = " << l2 << "\n";

	// Look at image
	auto img = imread("rubik_cube.jpg", CV_LOAD_IMAGE_GRAYSCALE);
	Mat img_c;
	//Mat img_c2;
	cvtColor(img, img_c, cv::COLOR_GRAY2BGR);
	//cvtColor(img, img_c2, cv::COLOR_GRAY2BGR);

	// Draw points on image
	Scalar color;
	draw_points(img_c, x_u, x_v, color = Scalar(255, 0, 255));
	draw_points(img_c, x_u_hat, x_v_hat, color = Scalar(0, 2550, 0));

	/// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
        /// Decompose the projection matrix into P = K * [R t]
        /// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        Proj_struct proj(P);
        proj.P_to_KRt();

        cout << "\n in main function \n\n\n";
        cout << "\n\nK:\n" << proj.K;
        cout << "\n\nR:\n" << proj.R;
        cout << "\n\nt:\n" << proj.t;

        /// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        /// Protoyping stuff:
        /// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        // Link to MATLAB environment
        matlabClass matlab;
        matlab.passImageIntoMatlab(x_reproject);
        //matlab.passImageIntoMatlab(A);

        // Run MATLAB script that executes prototype
        matlab.command("assn3");

        imshow("test", img_c);
        waitKey(0);
}

//=========================================================================
//=========================================================================
//=========================================================================
//=========================================================================
//=========================================================================
Mat calibrate(const Mat& x_, const Mat& X_)
{
        // Function computes the Projection matrix P from a set (>= 6)
        // 3D<->2D point correspondences

        // Construct the matrix A corresponding to equation 7.2 (page 178 in H&Z)
        auto A = construct_A(x_, X_);

        // Drop an SVD on A
        Mat W, U, Vt;
        cv::SVD::compute(A, W, U, Vt);

        // Modify SVD
        Mat V = -Vt.t(); // SVD in OpenCV is different than MATLAB and Numpy

        // Extract right most column of V
        const size_t I = V.rows;
        const size_t J = V.cols;
        Mat v(I, 1, CV_64FC1); // Col-vector to store right most col of V
        for (int i = 0; i < I; ++i)
                v.at<double>(i, 0) = V.at<double>(i, J - 1); // Itterate down right-most col of V

       // Re-shape into 3x4 camera-projection matrix
        int cn = 0;
        int rows = 3;
        Mat P = v.reshape(cn, rows);
        //cout << "P:\n" << P;

        return P;
}
```

```
//==========================================================================
Mat to_homo(const Mat& x)
{
        // Map a vector in non-homogeneous coordinates (dimension m)
        //  to a vector in homogeneous coordinates     (dimension m+1)

        Mat x_ = Mat::ones(x.rows + 1, x.cols, CV_64FC1);
        for (int i = 0; i < x.rows; ++i)
                for (int j = 0; j < x.cols; ++j)
                        x_.at<double>(i, j) = x.at<double>(i, j);

        return x_;
}
//==========================================================================
Mat from_homo(const Mat& x_)
{
        // Map a vector in homogeneous coordinates     (dimension m+1)
        //  to a vector in non-homogeneous coordinates (dimension m)

        const size_t rows = x_.rows;
        const size_t cols = x_.cols;

        Mat x = Mat::ones(rows - 1, cols, CV_64FC1);
        for (int i = 0; i < rows - 1; ++i)
                for (int j = 0; j < cols; ++j)
                        x.at<double>(i, j) = x_.at<double>(i, j) / x_.at<double>(rows-1, j);

        return x;
}
//==========================================================================
Mat construct(const Mat& x_j, const Mat& X_j)
{
        // Construct a matrix corresponding to eq. 7.2 in Hartley and Zisserman

        // To match H&Z notation
        auto x = x_j.at<double>(0, 0);
        auto y = x_j.at<double>(0, 1);
        auto w = x_j.at<double>(0, 2);

        // Construct row-1
        Mat temp1 = Mat::zeros(1, 4, CV_64FC1);
        Mat temp2 = -w * X_j;
        Mat temp3 =  y * X_j;

        Mat row1;
        hconcat(temp1, temp2, row1);
        hconcat(row1, temp3, row1);
        //cout << "\nrow1:\n" << row1;

        // Construct row-2
        temp1 = w * X_j;
        temp2 = Mat::zeros(1, 4, CV_64FC1);
        temp3 = -x * X_j;

        Mat row2;
        hconcat(temp1, temp2, row2);
        hconcat(row2, temp3, row2);

        // Concatenate two rows rows
        Mat A_two_rows;
        vconcat(row1, row2, A_two_rows);

        return A_two_rows;
}
```

References

[1] R. Hartley and A. Zissermann, Multiview geometry, 2nd edition, Cambridge University Press.

[2] Z. Zhang. A flexible new technique for camera calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(11):1330-1334, 2000.