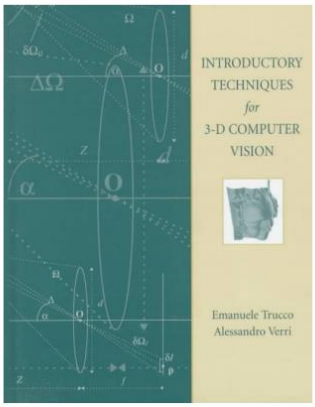
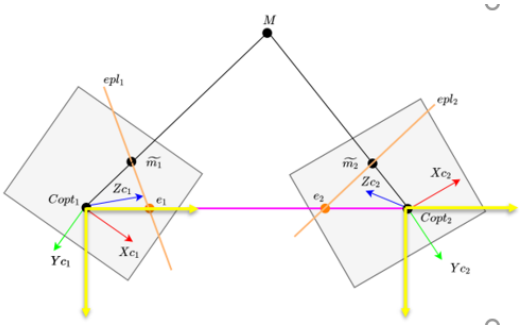
Rectification algorithm

There are different rectification algorithms. Matlab actually has a ToolBox to calibrate stereo cameras and rectify them. The following rectification algorithm is based on the book by Trucco & Verry, pp. 159.

1. Known $\boldsymbol{T}$ and $\boldsymbol{R}$ between cameras
2. Rotate left camera so that epipole el goes to infinity along horizontal axis
3. Apply same rotation to right camera to recover geometry
4. Rotate right camera by $\boldsymbol{R^{-1}}$
5. Adjust scale



1. To rectify image 1, we define a frame $\{\boldsymbol{Xcr1}, \boldsymbol{Yxc1}, \boldsymbol{Zcr1}\}$ with the origin $\boldsymbol{Copt1}$. $\boldsymbol{Xcr1}$ has the same direction as $\boldsymbol{T}$, i.e. $\boldsymbol{Xcr1} = \boldsymbol{T} / ||\boldsymbol{T}||$. Choose $\boldsymbol{Ycr1}$ perpendicular to $\boldsymbol{Xcr1}$ closest to the original $\boldsymbol{Yc1}$. $\boldsymbol{Ycr1} = \boldsymbol{cross}(\boldsymbol{Zc1}, \boldsymbol{Xcr1})$ = $\boldsymbol{cross}(\,[\boldsymbol{0}, \boldsymbol{0}, \boldsymbol{1}]', \boldsymbol{Xcr1})$. Finally, $\boldsymbol{Zcr1} = \boldsymbol{cross}(\boldsymbol{Xcr1}, \boldsymbol{Ycr1})$.
   If we define:
   $\boldsymbol{R_{rect}} = [\boldsymbol{Xcr1} \quad \boldsymbol{Ycr1} \quad \boldsymbol{Zcr1}]$, then ${}^{\boldsymbol{c1}}\boldsymbol{p} = \boldsymbol{R_{rect}} * {}^{\boldsymbol{cr1}}\boldsymbol{p}$; ${}^{\boldsymbol{cr1}}\boldsymbol{p} = \boldsymbol{R}^{\boldsymbol{T}}_{\boldsymbol{rect}} * {}^{\boldsymbol{c1}}\boldsymbol{p}$.



2. Calculate the homography $\boldsymbol{H_1}$ between the original image in camera 1 and the rectified image in the camera 1. This homography is $\boldsymbol{H_1} = \boldsymbol{K_1} * \boldsymbol{R}^{\boldsymbol{T}}_{\boldsymbol{rect}} * \boldsymbol{K_1^{-1}}$ as:

$$\tilde{\boldsymbol{m}}_{\boldsymbol{1rect}} = \boldsymbol{K_1} * \boldsymbol{R}^{\boldsymbol{T}}_{\boldsymbol{rect}} * \boldsymbol{K_1^{-1}} * \tilde{\boldsymbol{m}}_{\boldsymbol{1}} = \boldsymbol{H_1} * \tilde{\boldsymbol{m}}_{\boldsymbol{1}}$$

   Due to the fact that $\hat{\boldsymbol{m}}_{\boldsymbol{1}} = {}^{\boldsymbol{c1}}\hat{\boldsymbol{m}}_{\boldsymbol{1}} = \boldsymbol{R}^{\boldsymbol{T}}_{\boldsymbol{rect}} * {}^{\boldsymbol{cr1}}\hat{\boldsymbol{m}}_{\boldsymbol{1}} = \boldsymbol{R}^{\boldsymbol{T}}_{\boldsymbol{rect}} * \hat{\boldsymbol{m}}_{\boldsymbol{1rect}}$, and $\hat{\boldsymbol{m}}_{\boldsymbol{1}} = \boldsymbol{K_1^{-1}} * \tilde{\boldsymbol{m}}_{\boldsymbol{1}}$.

3. Calculate the homography $\boldsymbol{H_2}$ between the original image in camera2 and the rectified image in camera 2. We choose the focal length of camera 2 the same as the one in camera 1. First, we have to rotate the camera 2 to the camera 1 around the origin of the camera 2 by the rotation R:

   $^{||1}\widehat{\boldsymbol{m}}_{\boldsymbol{2}} = \boldsymbol{R} * \widehat{\boldsymbol{m}}_{\boldsymbol{2}}$, where $^{||1}\widehat{\boldsymbol{m}}_{\boldsymbol{2}}$ corresponds to $\widehat{\boldsymbol{m}}_{\boldsymbol{2}}$ in a frame parallel to 1.

   Then, $^{||1}\widehat{\boldsymbol{m}}_{\boldsymbol{2}}$ has to be transform by the same transformation $\boldsymbol{R}^{\boldsymbol{T}}_{\boldsymbol{rect}}$, in order to get the final transformation. Also, the focal length of the second camera is given as that of the first one. This homography is $\boldsymbol{H_2} = \boldsymbol{K_1} * \boldsymbol{R}^{\boldsymbol{T}}_{\boldsymbol{rect}} * \boldsymbol{R} * \boldsymbol{K_2^{-1}}$, as:

$$\widetilde{\boldsymbol{m}}_{\boldsymbol{2rect}} = \boldsymbol{K_1} * \boldsymbol{R}^{\boldsymbol{T}}_{\boldsymbol{rect}} * \boldsymbol{R} * \boldsymbol{K_2^{-1}} * \widetilde{\boldsymbol{m}}_{\boldsymbol{2}} = \boldsymbol{H_2} * \widetilde{\boldsymbol{m}}_{\boldsymbol{2}}$$

Rectification of a particular set of points is made by the previous formulae. To rectify images, use the Matlab function imwarp instead.

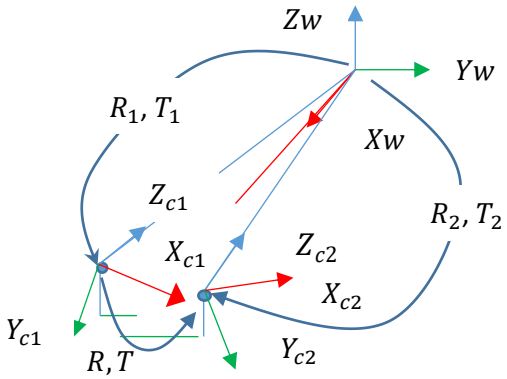
Rectificación en Matlab

The Matlab function **rectifyStereoImages** allows one the rectifying of images, given the **stereoParams** structure. Internally, this función calculates the homografías $\boldsymbol{H_1}, \boldsymbol{H_2}$, and a matriz $\boldsymbol{Q}$ to calculate the point (X, Y, Z) given the disparities in the rectified images. As these Matlab structures are not publically accesible, we have done reverse engineering in the Matlab Toolbox and make it accessible in a provided by us función **getHomoStereo**, that delivers the $\boldsymbol{H_1}$ and $\boldsymbol{H_2}$ homographies.

Suppose that we have 2 cameras. Camera 1 has a focal length f1 = 15mm and the pixel size = 5.0*1.0e-3mm. The size of the image is 1640x1024 and Cx = 1640/2, Cy = 1024/2, and has no distortions. The second camera is similar with a focal length of 16mm.

```
clear all; close all; clc;

% Definiciones de las cámaras
f1 = 15; f2 = 16;
sx = 5.0*1.0e-3; sy = sx;
SizeImX = 1280; SizeImY = 1024;

K1 = [f1/sx, 0, SizeImX/2; 0, f1/sy, SizeImY/2; 0, 0, 1];
K2 = [f2/sx, 0, SizeImX/2; 0, f2/sy, SizeImY/2; 0, 0, 1];
```

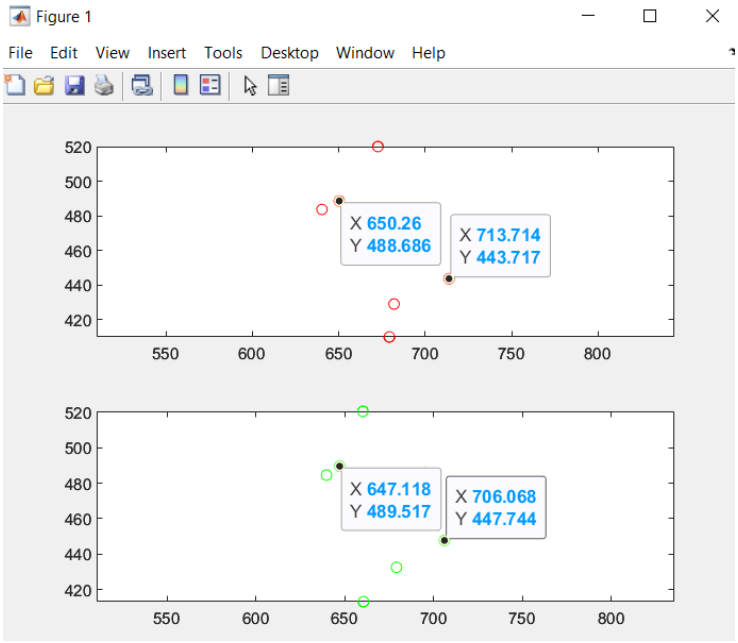
Imagine that the optical center of the camera 1 is placed in ${}^{w}\boldsymbol{T_1} = [\mathbf{1000}, -\mathbf{100}, \mathbf{200}]^T$ respect to the world frame. Also, assume that the axis $\boldsymbol{Z_{c1}}$ of camera 1 observes the origin of the world frame, i.e. ${}^{w}\boldsymbol{Z_{c1}} = -{}^{w}\boldsymbol{T_1}/||{}^{w}\boldsymbol{T_1}||$. The axis ${}^{w}\boldsymbol{X_{c1}}$ is perpendicular to ${}^{w}\boldsymbol{Z_{c1}}$ and closest to the $\boldsymbol{Y}$ axis in the world frame, i.e. the cross product of ${}^{w}\boldsymbol{Z_{c1}}$ and the $\boldsymbol{Z}$ axis in the world frame. The optical center of the camera 2 is placed in $[\mathbf{1100}, \mathbf{110}, \mathbf{210}]^T$ with respect to the world frame. We make the same assumptions as in camera 1 about the world frame.




```
% Transform definitions
w_T1 = [1000, -100, 200]';
w_T2 = [1100, +110, 210]';

w_Zcam1 = - w_T1/norm(w_T1);
w_Zcam2 = - w_T2/norm(w_T2);

w_Xcam1 = cross(w_Zcam1, [0; 0; 1])/norm(cross(w_Zcam1, [0; 0; 1]));
w_Xcam2 = cross(w_Zcam2, [0; 0; 1])/norm(cross(w_Zcam2, [0; 0; 1]));

% Elegir la transformacion entre el world y cam.
w_R1T1 = [w_Xcam1, cross(w_Zcam1, w_Xcam1), w_Zcam1, w_T1; 0, 0, 0, 1];
w_R2T2 = [w_Xcam2, cross(w_Zcam2, w_Xcam2), w_Zcam2, w_T2; 0, 0, 0, 1];
R1T1 = inv(w_R1T1);
R2T2 = inv(w_R2T2);
RT = R1T1 * w_R2T2;
R1 = R1T1(1:3, 1:3);
R2 = R2T2(1:3, 1:3);
R = RT(1:3, 1:3);
T1 = R1T1(1:3,4);
T2 = R2T2(1:3,4);
T = RT(1:3,4);
```

When we plot 8 points in both cameras, we observe that they are not rectified.

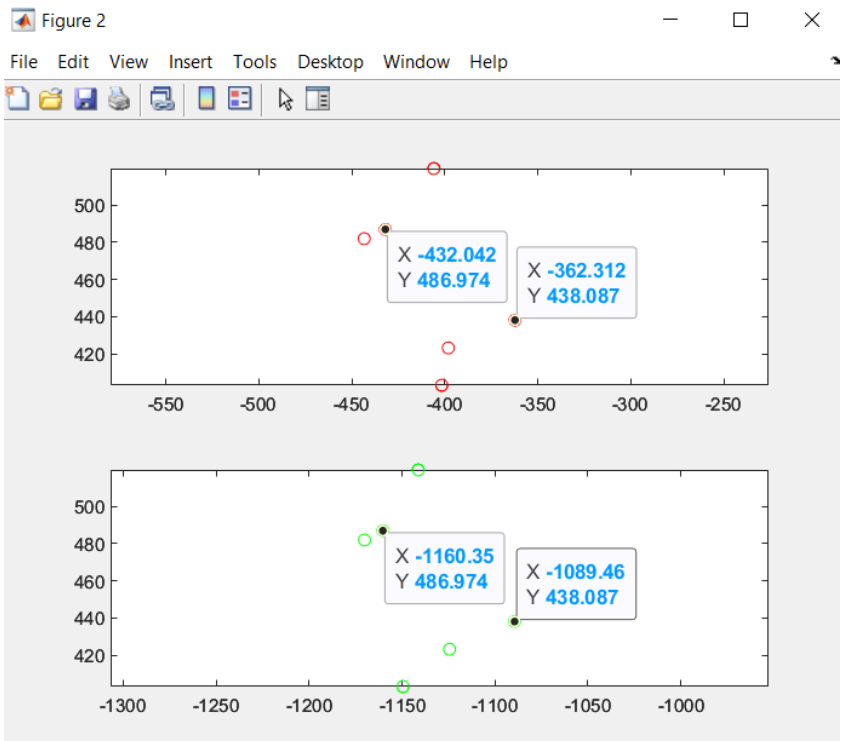
```
PuntosW = [5 3 9;1 0 10;3 8 20;7 20 11;20 9 1;3 14 29;10 24 25;30 10 40]';

m1 = K1 * [R1, T1] * [PuntosW; ones(1,size(PuntosW,2))];
m1 = m1 ./ m1(3,:);


m2 = K2 * [R2, T2] * [PuntosW; ones(1,size(PuntosW,2))];
m2 = m2 ./ m2(3,:);

% Display points
figure(2)
subplot(2,1,1)
plot(m1(1,:), m1(2,:), 'ro');
axis equal;

subplot(2,1,2)
plot(m2(1,:), m2(2,:), 'go');
axis equal;
```



Practical rectification

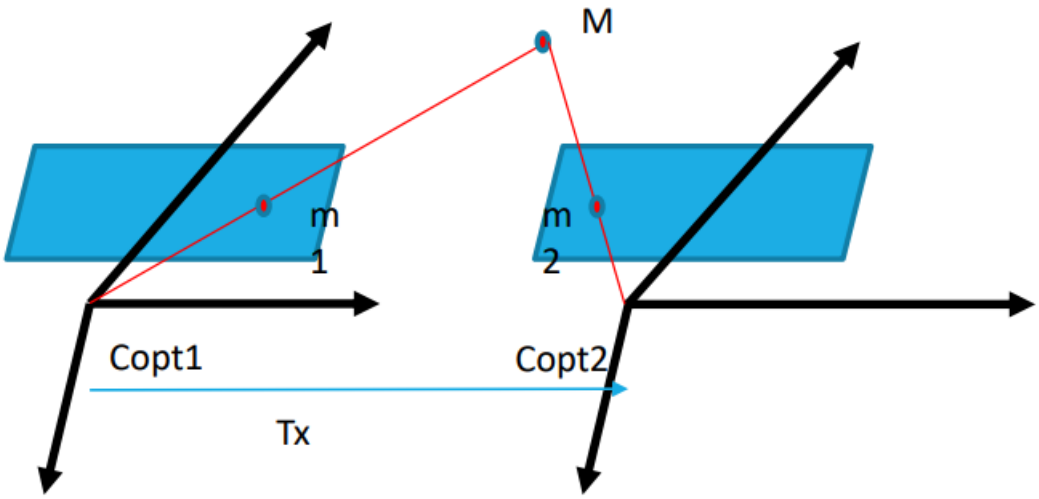
We rectify the 8 points previously used and verified that they are correctly rectified. For it, we calculate the homographies $H1$ and $H2$, getting in both images the same focal length corresponding to the first camera. We observe that the Y-component in both images is the same.

```
Xcr = T/norm(T);
Ycr = cross([0, 0, 1]', Xcr)/norm(cross([0, 0, 1]', Xcr));
Zcr = cross(Xcr, Ycr);
Rect1 = [Xcr'; Ycr'; Zcr'];
H1 = K1 * Rect1 * inv(K1);
H1 = H1 / H1(3, 3);
H2 = K1 * Rect1 * R * inv(K2);
H2 = H2 / H2(3, 3);

m1r = H1 * m1;
m1r = m1r./m1r(3,:);

m2r = H2 * m2;
m2r = m2r./m2r(3,:);

% Display points
figure(3)
subplot(2,1,1)
plot(m1r(1,:), m1r(2,:), 'ro');
axis equal;

subplot(2,1,2)
plot(m2r(1,:), m2r(2,:), 'go');
axis equal;
```



Triangulation and depth calculation

The stereo triangulation in the rectified cameras is easily calculated:



$$^{cam1}M = \lambda * K^{-1} * \tilde{m}_1; \; ^{cam2}M = {}^{cam1}M - [T_x \quad 0 \quad 0]^T = \lambda * K^{-1} * \tilde{m}_1 - [T_x \quad 0 \quad 0]^T$$

$$\tilde{m}_2 = K * {}^{cam2}M = K * \left(\lambda * K^{-1} * \tilde{m}_1 - [T_x \quad 0 \quad 0]^T\right) = \lambda * \tilde{m}_1 - K * [T_x \quad 0 \quad 0]^T$$

$$\tilde{m}_2 = \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda * x_1 \\ \lambda * y_1 \\ \lambda \end{bmatrix} - \begin{bmatrix} F & 0 & C_x \\ 0 & F & C_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} T_x \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 - \dfrac{F * T_x}{\lambda} \\ y_1 \\ 1 \end{bmatrix}$$

If we define the disparity as $disp = (x_1 - x_2)$, then:

$$Z = \lambda = \frac{F * T_x}{disp}$$

Further development leads to:

$$^{cam1}M = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \lambda * K^{-1} * \tilde{m}_1 = \lambda * \begin{bmatrix} F & 0 & C_x \\ 0 & F & C_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} * \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \lambda * \begin{bmatrix} \frac{1}{F} & 0 & -\frac{C_x}{F} \\ 0 & \frac{1}{F} & -\frac{C_y}{F} \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$^{cam1}M = \begin{bmatrix} \frac{F * T_x}{F * disp} & 0 & -\frac{C_x * F * T_x}{F * disp} \\ 0 & \frac{F * T_x}{F * disp} & -\frac{C_y * F * T_x}{F * disp} \\ 0 & 0 & \frac{F * T_x}{disp} \end{bmatrix} * \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{T_x}{disp} & 0 & -\frac{C_x * T_x}{disp} \\ 0 & \frac{T_x}{disp} & -\frac{C_y * T_x}{disp} \\ 0 & 0 & \frac{F * T_x}{disp} \end{bmatrix} * \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

That can be written with the following homography:

$$^{cam1}\tilde{M} = \lambda * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} T_x & 0 & 0 & -C_x * T_x \\ 0 & T_x & 0 & -C_y * T_x \\ 0 & 0 & 0 & F * T_x \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} x_1 \\ y_1 \\ disp \\ 1 \end{bmatrix}$$

As any homography can be substituted by another homography multiply by a scalar, dividing by T_x:

$$^{cam1}\tilde{M} = \lambda * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 0 & F \\ 0 & 0 & 1/T_x & 0 \end{bmatrix} * \begin{bmatrix} x_1 \\ y_1 \\ disp \\ 1 \end{bmatrix} = Q * \begin{bmatrix} x_1 \\ y_1 \\ disp \\ 1 \end{bmatrix}$$

```
%%%%% Calculate the stero point in function of the disparity
Tx = norm(T);

Q = [1, 0, 0, - K1(1, 3);...
     0, 1, 0, - K1(2, 3);...
     0, 0, 0, K1(1,1);...
     0, 0, 1/Tx, 0];

 displac = m1r(1, :) - m2r(1, :);

 ptsDisp = Q * [m1r(1:2, :); displac; ones(1, size(m1r, 2))];
 ptsDisp = ptsDisp ./ ptsDisp(4, :);
```

The obtained points are defined in the rectified cam1 frame. In order to compared them with the points in the world frame, we use the homogeneous coordinates between the world and rectified camera frame.

$$^{c1rect}\tilde{P} = \begin{bmatrix} R_{rect} & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} R_1 & T_1 \\ 0 & 1 \end{bmatrix} * {}^{w}\tilde{P}, \quad {}^{w}\tilde{P} = \begin{bmatrix} R_1 & T_1 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} R_{rect}{}^{T} & 0 \\ 0 & 1 \end{bmatrix} * {}^{c1rect}\tilde{P}$$

```
% Transform between world frame and rectfied cam1 frame
 RcTc1 = [Rect1, [0;0;0]; 0 0 0 1] * [R1,T1;0 0 0 1];
 ptsOrg = inv(RcTc1) * ptsDisp;

 err = ptsOrg(1:3,:) - PuntosW(1:3,:);
```

```
err =
  1.0e-12 *
   0.5684   0.2274  -0.1137   0.5684   0.1137   0.2274   0.4547        0
  -0.0995  -0.0853  -0.0853  -0.1421  -0.0995  -0.1137  -0.0995  -0.0995
   0.0853        0  -0.0284   0.0568   0.0284  -0.0284   0.0284  -0.0284
```

We verify that we recover the original world points.

Differences with Matlab

In Matlab, the calculation of the matrix *Q* is not directly available. Instead, there is a function **`rectifyStereoImages(I1,I2,stereoParams,OutputView),`** that rectifies two images, given the stereoParams. If we go inside this function, internally it call the function:

```
[H1, H2, R1, R2, camMatrix1, camMatrix2, Q, xBounds, yBounds, success] =
this.computeRectificationParameters(imageSize, outputView)
```

which calculates both homographies, as well as the matrix *Q*. If we look at the code, it is basically the same as the proposed one, with the difference that both cameras are aligned at the same time. To do it, a calculation of the axis and angle corresponding to the matrix *R* is performed, and the first camera is rotated to the left by half the angle, and the second camera is rotated to the right by the same quantity (`[Rl, Rr] = computeHalfRotations(this))`. With this process, both images planes are aligned. The X-axis has to be aligned with the *T* vector, and the homographies `Hleft, Hright, Q` are computed.

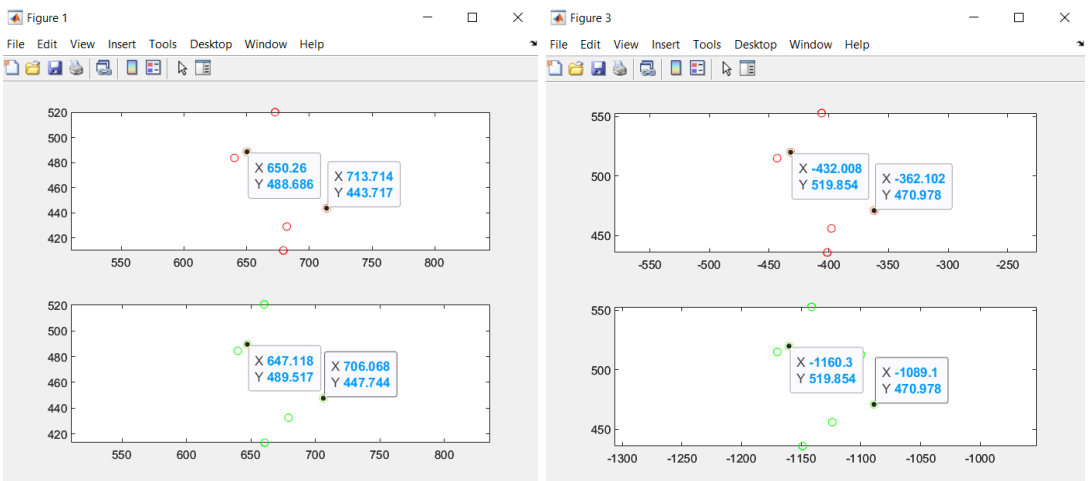
```
function [Hleft, Hright, R1, R2, camMatrix1, camMatrix2, Q, xBounds, yBounds, success] =
        computeRectificationParameters(this, imageSize, outputView)

    % Make the two image planes coplanar, by rotating each half way
    [Rl, Rr] = computeHalfRotations(this);

    % rotate the translation vector
    t = Rr * this.TranslationOfCamera2';

    % Row align the image planes, by rotating both of them such
    % that the translation vector coincides with the X-axis.
    RrowAlign = computeRowAlignmentRotation(t);

    % combine rotation matrices
    R1 = RrowAlign * Rl;
    R2 = RrowAlign * Rr;

    Kl = this.CameraParameters1.K;
    Kr = this.CameraParameters2.K;
    K_new = computeNewIntrinsics(this); % In OpenCV format

    Hleft  = projective2d((K_new * R1 / Kl)');
    Hright = projective2d((K_new * R2 / Kr)');

    % apply row alignment to translation
    t = RrowAlign * t;

    [xBounds, yBounds, success] = computeOutputBounds(this, ...
        imageSize, Hleft, Hright, outputView);

    K_new(1:2, 3) = K_new(1:2, 3) - [xBounds(1); yBounds(1)];
    camMatrix1 = [eye(3); zeros(1, 3)]  * K_new';
    camMatrix2 = [eye(3); [t(1), 0, 0]] * K_new';

    % [x, y, disparity, 1] * Q = [X, Y, Z, 1] * w
    cy = K_new(2,3);
    cx = K_new(1,3);
    f_new = K_new(2,2);
    Q = [1, 0,   0,       -cx;
         0, 1,   0,       -cy;
         0, 0,   0,       f_new;
         0, 0, -1/t(1),   0]';

end
```

Figure – Internal Matlab function

Matlab method of left and right rotation

The Matlab method has the advantage that the right and left cameras are halfway rotated, making a different solution that is closed to the final solution. However, the coordinates in the y direction of the rectified points are not so closed to the original points as our rectification method. The resulting homographies and matrix Q, as well as the rectified points are also slightly different.

```
cameraParams1 = cameraParameters('K', K1, 'RadialDistortion',[0 0]);
cameraParams2 = cameraParameters('K', K2, 'RadialDistortion',[0 0]);
stereoParams = stereoParameters(cameraParams1,cameraParams2, R, -R' * T);

% I got some of these functions entering inside the Matlab toolbox
[Hleft, Hright, R1new, R2new, camMatrix1, camMatrix2, Qnew, xBounds, yBounds] = ...
       computeRectificationParameters1(stereoParams, [1024, 1280], 'valid');

m1new = Hleft.T' * m1;
m1new = m1new ./ m1new(3,:);

m2new = Hright.T' * m2;
m2new = m2new ./ m2new(3,:);

% Display points
figure(4)
subplot(2,1,1)
plot(m1new(1,:), m1new(2,:), 'ro');
axis equal;

subplot(2,1,2)
plot(m2new(1,:), m2new(2,:), 'go');
axis equal;

disp('end');
```



In some cases, the images rectified with these homographies may not be visible in one of the images. In order to better this rectification, there are modifications (e.g. Multiple View Geometry in Computer Vision, Hartley & Zisserman, pp. 306) that make the points always visible.

Note: The necessary functions to run this program are in the Annex.

Exercise

1. Calculate the fundamental matrix with 8 points, adding noise to the observed images. Compare this calculation and the one with normalization. Also, calculate the relative rotation and translation between both cameras, give the essential matrix (see https://es.mathworks.com/help/vision/ref/relativecamerapose.html).

2. How would you calculate the original points using the matrix Q? How do you obtain in this case the matrix $\boldsymbol{R_{rect}}$ from the matrices `R1new, R2new?`

3. See also https://es.mathworks.com/help/vision/stereo-vision.html , https://es.mathworks.com/help/vision/ug/depth-estimation-from-stereo-video.html and https://es.mathworks.com/help/vision/ref/rectifystereoimages.html to rectify the camera images in Matlab, and how to compute the depth from stereo.

```
function [Hleft, Hright, R1, R2, camMatrix1, camMatrix2, Q, xBounds, yBounds, success] = ...
                computeRectificationParameters(stereoParams, imageSize, outputView)

            % Make the two image planes coplanar, by rotating each half way
            [Rl, Rr] = computeHalfRotations(stereoParams);

            % rotate the translation vector
            t = Rr * stereoParams.TranslationOfCamera2';

            % Row align the image planes, by rotating both of them such
            % that the translation vector coincides with the X-axis.
            RrowAlign = computeRowAlignmentRotation(t);

            % combine rotation matrices
            R1 = RrowAlign * Rl;
            R2 = RrowAlign * Rr;

            Kl = stereoParams.CameraParameters1.K;
            Kr = stereoParams.CameraParameters2.K;
            K_new = computeNewIntrinsics(stereoParams); % In OpenCV format

            Hleft  = projective2d((K_new * R1 / Kl)');
            Hright = projective2d((K_new * R2 / Kr)');

            % apply row alignment to translation
            t = RrowAlign * t;

            [xBounds, yBounds, success] = computeOutputBounds(stereoParams, ...
                imageSize, Hleft, Hright, outputView);

            K_new(1:2, 3) = K_new(1:2, 3) - [xBounds(1); yBounds(1)];
            camMatrix1 = [eye(3); zeros(1, 3)]  * K_new';
            camMatrix2 = [eye(3); [t(1), 0, 0]] * K_new';

            % [x, y, disparity, 1] * Q = [X, Y, Z, 1] * w
            cy = K_new(2,3);
            cx = K_new(1,3);
            f_new = K_new(2,2);
            Q = [1, 0,   0,       -cx;
                 0, 1,   0,       -cy;
                 0, 0,   0,       f_new;
                 0, 0, -1/t(1),   0]';

end
%--------------------------------------------------------------------
function [Rl, Rr] = computeHalfRotations(this)
r = vision.internal.calibration.rodriguesMatrixToVector(this.RotationOfCamera2');

% right half-rotation
Rr = vision.internal.calibration.rodriguesVectorToMatrix(r / -2);

% left half-rotation
Rl = Rr';
end

%--------------------------------------------------------------------
function K_new = computeNewIntrinsics(stereoParams)
% initialize new camera intrinsics
Kl = stereoParams.CameraParameters1.IntrinsicMatrix';
Kr = stereoParams.CameraParameters2.IntrinsicMatrix';

K_new=Kl;

% find new focal length
f_new = min([Kr(1,1),Kl(1,1)]);

% set new focal lengths
K_new(1,1)=f_new; K_new(2,2)=f_new;

% find new y center
```

```
cy_new = (Kr(2,3)+Kl(2,3)) / 2;

% set new y center
K_new(2,3)= cy_new;

% set the skew to 0
K_new(1,2) = 0;
end

%--------------------------------------------------------------------------
function [xBounds, yBounds, success] = computeOutputBounds(stereoParams, ...
    imageSize, Hleft, Hright, outputView)

% find the bounds of the undistorted images
[xBoundsUndistort1, yBoundsUndistort1] = ...
    computeUndistortBounds(stereoParams.CameraParameters1, ...
    imageSize, outputView);

undistortBounds1 = getUndistortCorners(xBoundsUndistort1, yBoundsUndistort1);

[xBoundsUndistort2, yBoundsUndistort2] = ...
    computeUndistortBounds(stereoParams.CameraParameters2, ...
    imageSize, outputView);
undistortBounds2 = getUndistortCorners(xBoundsUndistort2, yBoundsUndistort2);

% apply the projective transformation
outBounds1 =  Hleft.transformPointsForward(undistortBounds1);
outBounds2 = Hright.transformPointsForward(undistortBounds2);

if strcmp(outputView, 'full')
    [xBounds, yBounds, success] = computeOutputBoundsFull( ...
        outBounds1, outBounds2);
else % valid
    [xBounds, yBounds, success] = computeOutputBoundsValid(...
        outBounds1, outBounds2);
end
end

%--------------------------------------------------------------
function undistortBounds = getUndistortCorners(xBounds, yBounds)
undistortBounds = [xBounds(1), yBounds(1);
    xBounds(2), yBounds(1);
    xBounds(2), yBounds(2);
    xBounds(1), yBounds(2);];
end

%--------------------------------------------------------------------------
function RrowAlign = computeRowAlignmentRotation(t)

xUnitVector = [1;0;0];
if dot(xUnitVector, t) < 0
    xUnitVector = -xUnitVector;
end

% find the axis of rotation
rotationAxis = cross(t,xUnitVector);

if norm(rotationAxis) == 0 % no rotation
    RrowAlign = eye(3);
else
    rotationAxis = rotationAxis / norm(rotationAxis);

    % find the angle of rotation
    angle = acos(abs(dot(t,xUnitVector))/(norm(t)*norm(xUnitVector)));

    rotationAxis = angle * rotationAxis;

    % convert the rotation vector into a rotation matrix
    RrowAlign = vision.internal.calibration.rodriguesVectorToMatrix(rotationAxis);
end
end
function [xBounds, yBounds, isValid] = computeOutputBoundsFull(...
    outBounds1, outBounds2)
```

```
minXY = min(outBounds1);
maxXY = max(outBounds1);
outBounds1 = [minXY; maxXY];

minXY = min(outBounds2);
maxXY = max(outBounds2);
outBounds2 = [minXY; maxXY];

minXY = round(min([outBounds1(1,:); outBounds2(1,:)]));
maxXY = round(max([outBounds1(2,:); outBounds2(2,:)]));
xBounds = [minXY(1), maxXY(1)];
yBounds = [minXY(2), maxXY(2)];
if minXY(1) >= maxXY(1) || minXY(2) >= maxXY(2)
    isValid = false;
else
    isValid = true;
end
end

%-------------------------------------------------------------
function [xBounds, yBounds, isValid] = computeOutputBoundsValid(...
    outBounds1, outBounds2)

% Compute the common rectangular area of the transformed images
outPts = [outBounds1; outBounds2];
xSort   = sort(outPts(:,1));
ySort   = sort(outPts(:,2));
xBounds = zeros(1, 2, 'like', outBounds1);
yBounds = zeros(1, 2, 'like', outBounds2);

outBounds1 = round(outBounds1);
outBounds2 = round(outBounds2);
% Detect if there is a common rectangle area that is large enough
xmin1 = min(outBounds1(:,1));
xmax1 = max(outBounds1(:,1));
xmin2 = min(outBounds2(:,1));
xmax2 = max(outBounds2(:,1));

if (xmin1 >= xmax2) || (xmax1 <= xmin2) % no overlap
    isValid = false;
else
    xBounds(1) = round(xSort(4));
    xBounds(2) = round(xSort(5));
    yBounds(1) = round(ySort(4));
    yBounds(2) = round(ySort(5));
    if xBounds(2)-xBounds(1) < 0.4 * min(xmax1-xmin1, xmax2-xmin2) % not big enough
        isValid = false;
    else
        isValid = true;
    end
end
end
```