Computation of the Fundamental and Essential matrix

based on the 8-Point algorithm (https://en.wikipedia.org/wiki/Eight-point_algorithm)

The fundamental or the Essential matrix can be obtained by the observation of 8 or more non-planar corresponding points. From the Fundamental matrix, we obtain:

$$\tilde{\boldsymbol{m}}_{\boldsymbol{2}}^{\boldsymbol{T}} * \boldsymbol{F} * \tilde{\boldsymbol{m}}_{\boldsymbol{1}} = [\boldsymbol{m}_{2x} \quad \boldsymbol{m}_{2y} \quad \boldsymbol{1}] * \begin{bmatrix} \boldsymbol{F}_{11} & \boldsymbol{F}_{12} & \boldsymbol{F}_{13} \\ \boldsymbol{F}_{21} & \boldsymbol{F}_{22} & \boldsymbol{F}_{23} \\ \boldsymbol{F}_{31} & \boldsymbol{F}_{32} & \boldsymbol{F}_{33} \end{bmatrix} * \begin{bmatrix} \boldsymbol{m}_{1x} \\ \boldsymbol{m}_{1y} \\ \boldsymbol{1} \end{bmatrix} = \boldsymbol{0}$$

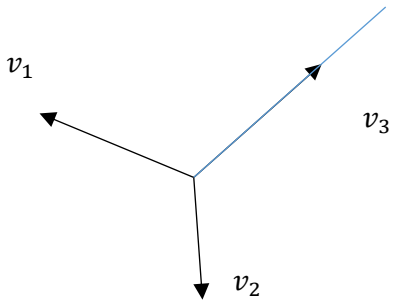
This can be put as linear function of the ciefficients of ***F***:

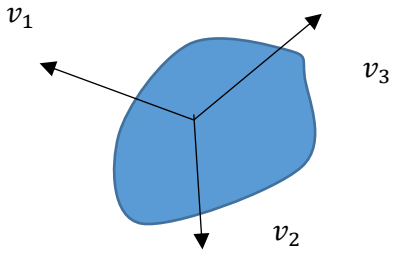
$$[\boldsymbol{m}_{2x} * \boldsymbol{m}_{1x} \quad \boldsymbol{m}_{2x} * \boldsymbol{m}_{1y} \quad \boldsymbol{m}_{2x} \quad \boldsymbol{m}_{2y} * \boldsymbol{m}_{1x} \quad \boldsymbol{m}_{2y} * \boldsymbol{m}_{1y} \quad \boldsymbol{m}_{2x} \quad \boldsymbol{m}_{2x} \quad \boldsymbol{m}_{2x} \quad 1] * \begin{bmatrix} \boldsymbol{F}_{11} \\ \boldsymbol{F}_{12} \\ \boldsymbol{F}_{13} \\ \boldsymbol{F}_{21} \\ \boldsymbol{F}_{22} \\ \boldsymbol{F}_{23} \\ \boldsymbol{F}_{31} \\ \boldsymbol{F}_{32} \\ \boldsymbol{F}_{33} \end{bmatrix} = \boldsymbol{0}$$

That results in a matrix ***A * F = 0.*** This resulting a solution of the least eigenvalue of $A^T * A$, or the smallest eigenvalue of the singular value decomposition of A. One of the properties of the fundamental matrix is that its determinant is zero. Thus, we can decompose $F = U * S * V^T$, and equal the smallest singular value of S to zero, $\hat{F} = U * \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} * V^T$, making $\hat{F}$ the closest one to F, but with a zero determinant.

The problem with the solution is that the matrix $A^T * A$ can be ill-conditioned, so that the biggest eigenvalue divided by the second smallest eigenvalue is very large. Effectively, the smallest eigenvalue of $A^T * A$ is always practically zero. If the second smallest eigenvalue is large, then the solution of ***A * F = 0*** is on the direction of the smallest eigenvalue v_3. If the second smallest eigenvalue is almost zero, then the solution is on the plane formed by v_2 and v_3, having infinite solutions on this plane, that make ***A * F = 0.***

.

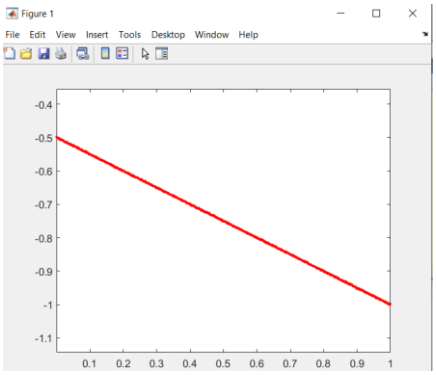




In order to understand it, we are going to make a simpler solution of a straight 2D line.

Suppose a line the line equation $\tilde{p}^T * \tilde{n} = [x \quad y \quad 1] * [a \quad b \quad c]^T = 0$. In matlab:

```
n = [1 2 1]';
n = n / norm(n);
factor = 1.0;
x = (0:0.001:1) * factor;
y = -(n(3) + x * n(1)) / n(2);
% We add noise to the y values
sigmaN = 0.001 * factor;
yn = y + sigmaN * randn(size(y));

plot(x, y); axis equal;
hold on
plot(x, yn ,'r.');
```



We solve both with least eigenvalues and svd.

```
% Solve A * n = 0 ======================================
A = [x', yn', ones(size(x'))];
disp('Atr * A = ');
A' * A
eigA = eig(A' * A)

[V, L] = eig(A' * A);
n1 = V(:, 1);
Err_n1 = min(abs(n1 - n/norm(n)), abs(n1 + n/norm(n)))
disp(strcat('n1(1) / n1(2) = ', num2str(n1(1) / n1(2))));

%=========================================================
[U, S, V1] = svd(A);
n2 = V1(:, end);
Err_n2 = min(abs(n2 - n/norm(n)), abs(n2 + n/norm(n)))
disp(strcat('n2(1) / n2(2) = ', num2str(n2(1) / n2(2))));
```

In this case, the matrix A'*A is well conditionned:

Atr * A = 1.0e+03 *

0.3338	**-0.4171**	**0.5005**
-0.4171	**0.5839**	**-0.7507**
0.5005	**-0.7507**	**1.0010**

eigA =
1.0e+03 *
0.0000
0.0678
1.8510

Err_n1 = 1.0e-04 * [0.2028 0.1034 0.4096]
n1(1) / n1(2) = 0.50002

Err_n2 = 1.0e-04 * [0.2028 0.1034 0.4096]
n2(1) / n2(2) = 0.50002

In this case $A^T * A$ has similar values and the quotient of the largest and second largest eigenvalue is 27, making the matrix well-conditioned. If the factor = 1000, then we obtain the following values:

Atr * A = 1.0e+08 *

3.3383	**-1.6716**	**0.0050**
-1.6716	**0.8370**	**-0.0025**
0.0050	**-0.0025**	**0.0000**

eigA =
1.0e+03 *
0.0000
0.0678
1.8510

Err_n1 = [0.3316 0.6605 0.5765]
n1(1) / n1(2) = 0.49127

Err_n2 = [0.3316 0.6605 0.5765]
n2(1) / n2(2) = 0.49127

In the second case, x, and y has been multiplied by 1000. It is like if in the first case we had units in meter, and in the second case in mm. We appreciate that $A^T * A$ has very different values with the last row and column very different from the first one. In addition, the quotient of the largest and second largest eigenvalue is 471969.3289, making the matrix ill conditioned. This implies that the solution is almost a linear combination of the second and third eigenvectors, as we observe in the error with the original equation of the line. In addition, we observe that the quotient between the first and second value of the obtained *n* is different from the original 0.5.

In order to make this fitting more robust, we transform the points (normalization) so that the matrix $A^T * A$ becomes well-conditioned.

Data normalization

One way of doing the normalization is to making the standard deviation of each variable equal to +1, and centered in the mean value, by means of a linear function *T*:

$$\tilde{\boldsymbol{p}}_{\boldsymbol{n}} = [\boldsymbol{x}_{\boldsymbol{n}} \quad \boldsymbol{y}_{\boldsymbol{n}} \quad \mathbf{1}]^{\boldsymbol{T}} = \boldsymbol{T} * \tilde{\boldsymbol{p}} = \boldsymbol{T} * [\boldsymbol{x} \quad \boldsymbol{y} \quad \mathbf{1}]^{\boldsymbol{T}}$$

Where, $\tilde{p}_n$ are the homogeneous coordinates of the normalized point *p*.

In Matlab, this function is calculated in the following way:

```
function T = getNormMat(x, y)
  xMax = max(x); xMin = min(x);
  yMax = max(y); yMin = min(y);

  stdX = std(x);
  stdY = std(y);
  mX = mean(x);
  mY = mean(y);

  T = [1/stdX, 0, -mX / stdX; 0, 1/stdY, -mY / stdY; 0, 0, 1];
end
```

With this transformation, the variables are transformation, so that their standard values are equal to 1. Other similar transformation are possible. In particular, Matlab uses an internal function call **normalizedPoints** (inside the **estimateFundamentalMatrixFunction**, based on the book by Hartley and Zisserman).

If we have the linear equation of the straight line:

$$[\boldsymbol{x}_{\boldsymbol{i}} \quad \boldsymbol{y}_{\boldsymbol{i}} \quad \mathbf{1}] * \begin{bmatrix} \boldsymbol{n}_{\boldsymbol{x}} \\ \boldsymbol{n}_{\boldsymbol{y}} \\ \mathbf{1} \end{bmatrix} = \mathbf{0}$$

And:

$$[\boldsymbol{x}_{\boldsymbol{in}} \quad \boldsymbol{y}_{\boldsymbol{in}} \quad \mathbf{1}]^{\boldsymbol{T}} = \boldsymbol{T} * \tilde{\boldsymbol{p}} = \boldsymbol{T} * [\boldsymbol{x}_{\boldsymbol{i}} \quad \boldsymbol{y}_{\boldsymbol{i}} \quad \mathbf{1}]^{\boldsymbol{T}}$$

Then:

$$[\boldsymbol{x}_{\boldsymbol{in}} \quad \boldsymbol{y}_{\boldsymbol{in}} \quad \mathbf{1}] = [\boldsymbol{x}_{\boldsymbol{i}} \quad \boldsymbol{y}_{\boldsymbol{i}} \quad \mathbf{1}] * \boldsymbol{T}^{\boldsymbol{T}}$$

The minimization with the normalized points result in:

$$[\boldsymbol{x}_{\boldsymbol{in}} \quad \boldsymbol{y}_{\boldsymbol{in}} \quad \mathbf{1}] * \begin{bmatrix} \boldsymbol{n}_{\boldsymbol{nx}} \\ \boldsymbol{n}_{\boldsymbol{ny}} \\ \mathbf{1} \end{bmatrix} = [\boldsymbol{x}_{\boldsymbol{i}} \quad \boldsymbol{y}_{\boldsymbol{i}} \quad \mathbf{1}] * \boldsymbol{T}^{\boldsymbol{T}} * \begin{bmatrix} \boldsymbol{n}_{\boldsymbol{nx}} \\ \boldsymbol{n}_{\boldsymbol{ny}} \\ \mathbf{1} \end{bmatrix} = \mathbf{0}$$

Where, $\begin{bmatrix} n_{nx} \\ n_{ny} \\ 1 \end{bmatrix}$ are the equation of the fitted line in normalized coordinates.

Thus:

$$\boldsymbol{s} * \begin{bmatrix} \boldsymbol{n}_{\boldsymbol{x}} \\ \boldsymbol{n}_{\boldsymbol{y}} \\ \mathbf{1} \end{bmatrix} = \boldsymbol{T}^{\boldsymbol{T}} * \begin{bmatrix} \boldsymbol{n}_{\boldsymbol{nx}} \\ \boldsymbol{n}_{\boldsymbol{ny}} \\ \mathbf{1} \end{bmatrix}$$

The code in Matlab for the normalized coordinates is:

```
%=========================================================
% We normalize the points x, y
T = getNormMat(x, yn);
p1 = T * [x; yn; ones(size(x))];
A1 = p1';
[U, S, V3] = svd(A1);
eigA1 = eig(A1' * A1)

n3 = V3(:, end);
n3uN = T' * n3;
n3uN = n3uN / norm(n3uN);
Err_n3 = min(abs(n3uN - n/norm(n)), abs(n3uN + n/norm(n)))
n3uN(1) / n3uN(2)
```

For the case of the factor = 1, the results are better by a factor of 10.

eigA1 =
1.0e+03 *
0.0000
1.0010
2.0000

Err_n3 = 1.0e-04 * [0.2737 0.0914 0.4565]
n1(1) / n1(2) = 0.5000

For the case of the factor = 1000, the results are better by several factor of magnitude.

eigA1 =
1.0e+03 *

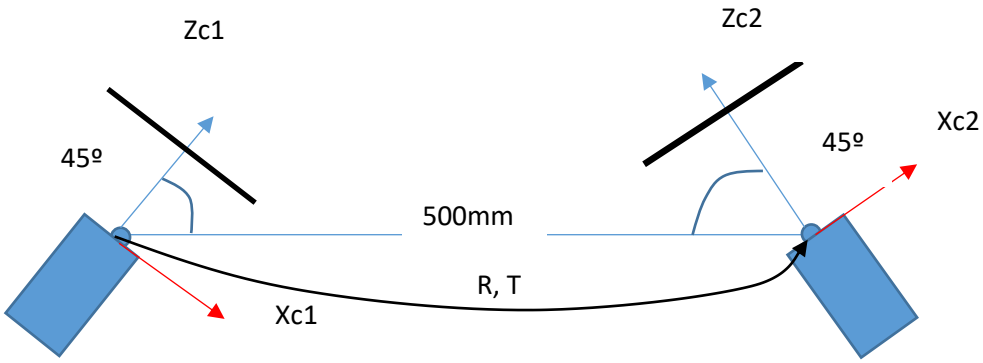
0.0000
1.0010
2.0000

Err_n3 = [0.0054 0.0106 0.0278]
n1(1) / n1(2) = 0.5000

Exercise

For the previous example, in which you calculated the Fundamental and essential matrix, calculate both matrices based on projection of 10 non-coplanar points in the world, using the 8 point algorithm with normalization. Compare the results.

I give again the scenario: Suppose that we have two cameras, with focal length f1 = 25 mm, f2 = 35mm. Both cameras have a pixel size of 5 microns, and a resolution of 1280 x 1024 pixels. We assume that the lenses don´t have distortion and that they are tilted in the Y axes by 45º as they are shown in the Figure. The optical center is at half the size of the camera, i.e. Cx = 1280/2, Cy = 1024/2.



From the Essential matrix, we can derive R and T. However, the method and its proof are complicated (see the book by Hartley and Zisserman). You can use the **relativeCameraPose** from Matlab (https://es.mathworks.com/help/vision/ref/relativecamerapose.html), and compare with the real R, T.

Note: Matlab recently change the way it calculates the camera geometry. Previous to 2022, it used to compute the camera transformation in a transposed way:

$$\gamma * [x_{cam} \quad y_{cam} \quad 1] = [x_w \quad y_w \quad z_w \quad 1] * \begin{bmatrix} R^T \\ T^T \end{bmatrix} * K^T$$

From 2022 on the transformation is the same as the rest of the vision community:

$$\gamma * [x_{cam} \quad y_{cam} \quad 1]^T = K * [R \quad T] * [x_w \quad y_w \quad z_w \quad 1]^T$$

The relativeCameraPose function has been changed by the function estrelpose (https://es.mathworks.com/help/vision/ref/estrelpose.html) with the new notation.

Matlab Stereo Programming

Matlab has functions that define cameras as well as stereo definitions. As the versions Matlab2022b and previous ones are different, I will give you both versions:

Version before Matlab2022b

```
f1 = 25; f2 = 35; Cx = 1280; Cy = 1024; sx = 5 * 1.0e-3; sy = 5 *
1.0e-3;

A1 = [f1 / sx, 0, Cx / 2; 0, f1 / sy, Cy / 2; 0 0 1];
A2 = [f2 / sx, 0, Cx / 2; 0, f2 / sy, Cy / 2; 0 0 1];

R = [0 0 -1; 0 1 0; 1 0 0];
T = [500 * cosd(45); 0; 500 * sind(45)];

radialDistortion = [0 0]';
cameraParams1 = cameraParameters('IntrinsicMatrix', A1',...
                                 'RadialDistortion',radialDistortion);
cameraParams2 = cameraParameters('IntrinsicMatrix', A2',...
                                 'RadialDistortion',radialDistortion);

R1 = eye(3); T1 = [0; 0; 0];
R2 = R'; T2 = - R' * T;

P1 = cameraParams1.IntrinsicMatrix' * [R1, T1];
P2 = cameraParams2.IntrinsicMatrix' * [R2, T2];

% Create the point M in the world and project it
M = [10; 20; 350];

pim1 = P1 * [M; 1];
pim1 = pim1/pim1(3);

pim2 = P2 * [M; 1];
pim2 = pim2/pim2(3);

stereoParams = stereoParameters(cameraParams1,cameraParams2, R, -R' *
T);

% 8 points-algoritm

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]';

pimw1 = P1 * [PuntosW; ones(1, size(PuntosW, 2))];
pimw1 = pimw1 ./ pimw1(3,:);
%pimw1 = pimw1(:,1:2);
pimw2 = P2 * [PuntosW; ones(1, size(PuntosW, 2))];
pimw2 = pimw2 ./ pimw2(3,:);
%pimw2 = pimw2(:,1:2);


fRANSAC1 = estimateFundamentalMatrix(pimw1(1:2,:)',...
pimw2(1:2,:)','Method','RANSAC',...
'NumTrials',2000,'DistanceThreshold',1e-4);

fRANSAC1 = fRANSAC1/fRANSAC1(3,3);
[Rest, Test] = relativeCameraPose(stereoParams.EssentialMatrix, ...
                                  cameraParams1, cameraParams2, ...
                                  pimw1(1:2,:)', pimw2(1:2,:)');
```

Version after Matlab2022b

```
% Define cameras
f1 = 25; f2 = 35; Cx = 1280; Cy = 1024; sx = 5 * 1.0e-3; sy = 5 * 1.0e-3;

A1 = [f1 / sx, 0, Cx / 2; 0, f1 / sy, Cy / 2; 0 0 1];
A2 = [f2 / sx, 0, Cx / 2; 0, f2 / sy, Cy / 2; 0 0 1];

R = [0 0 -1; 0 1 0; 1 0 0];
T = [500 * cosd(45); 0; 500 * sind(45)];

radialDistortion = [0 0]';

cameraParams1 = cameraParameters('K', A1,
'RadialDistortion',radialDistortion);
cameraParams2 = cameraParameters('K', A2,
'RadialDistortion',radialDistortion);

R1 = eye(3); T1 = [0; 0; 0];
R2 = R'; T2 = - R' * T;

P1 = cameraParams1.K * [R1, T1];
P2 = cameraParams2.K * [R2, T2];

% Create the point M in the world and project it
M = [10; 20; 350];

pim1 = P1 * [M; 1];
pim1 = pim1/pim1(3);

pim2 = P2 * [M; 1];
pim2 = pim2/pim2(3);

stereoParams = stereoParameters(cameraParams1,cameraParams2, R, -R' * T);

% 8 points-algoritm

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]';

pimw1 = P1 * [PuntosW; ones(1, size(PuntosW, 2))];
pimw1 = pimw1 ./ pimw1(3,:);
%pimw1 = pimw1(:,1:2);
pimw2 = P2 * [PuntosW; ones(1, size(PuntosW, 2))];
pimw2 = pimw2 ./ pimw2(3,:);
%pimw2 = pimw2(:,1:2);


fRANSAC1 = estimateFundamentalMatrix(pimw1(1:2,:)',...
                                     pimw2(1:2,:)','Method','RANSAC',...
                                   'NumTrials',2000,'DistanceThreshold',1e-4);

fRANSAC1 = fRANSAC1/fRANSAC1(3,3);
```

```
[Rest, Test] = relativeCameraPose(stereoParams.EssentialMatrix, ...
                                  cameraParams1, cameraParams2, ...
                                  pimw1(1:2,:)', pimw2(1:2,:)');
focalLength    = [cameraParams1.K(1,1) cameraParams1.K(2,2)];
```

```
principalPoint = [cameraParams1.K(1,3), cameraParams1.K(2,3)];
imageSize      = [cameraParams1.K(1,3)*2, cameraParams1.K(2,3)*2];
% Create a camera intrinsics object.
intrinsics1 = cameraIntrinsics(focalLength,principalPoint,imageSize);

focalLength    = [cameraParams2.K(1,1) cameraParams2.K(2,2)];
principalPoint = [cameraParams2.K(1,3), cameraParams2.K(2,3)];
imageSize      = [cameraParams2.K(1,3)*2, cameraParams2.K(2,3)*2];
% Create a camera intrinsics object.
intrinsics2 = cameraIntrinsics(focalLength,principalPoint,imageSize);

[tformRT] = estrelpose(stereoParams.EssentialMatrix, ...
                                    intrinsics1, intrinsics2, ...
                                    pimw1(1:2,:)', pimw2(1:2,:)');
[tformRT.R, tformRT.Translation'; 0 0 0 1]
```

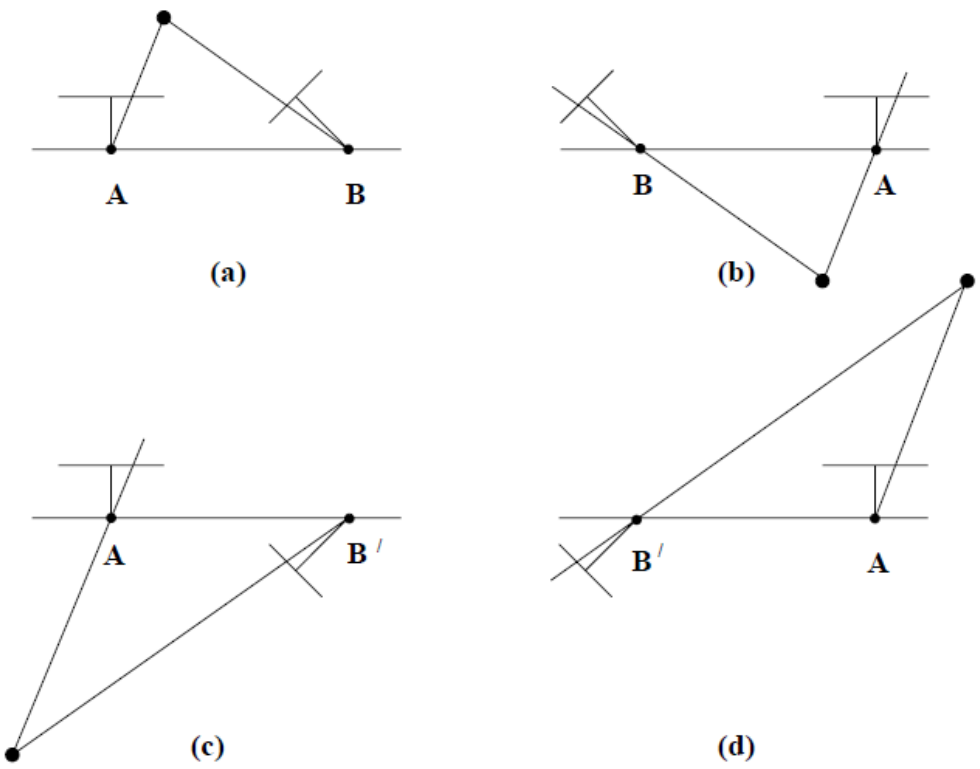
```
>> stereoParams.FundamentalMatrix / stereoParams.FundamentalMatrix(3,3)
ans =

     0   -0.0000    0.0014
  -0.0000       0    0.0153
   0.0014   -0.0173    1.0000

>> fRANSAC1

fRANSAC1 =

     0   -0.0000    0.0014
  -0.0000       0    0.0153
   0.0014   -0.0173    1.0000
```

Calculation of the rotation and Translation from Essential Matrix

First, only the rotation R and the direction of the vector T can be estimated, as the Essential matrix is of rank 2. This can also be observed that two cameras with the same rotation and the translation scaled by a factor gives an Essential matrix multiplied by this factor, and becomes another possible Essential matrix, i.e. if *E* is an Essential matrix, so it is $\boldsymbol{k} * \boldsymbol{E}$, as $\widehat{\boldsymbol{m}}_{\boldsymbol{2}}^{\boldsymbol{T}} * \boldsymbol{k} * \boldsymbol{E} * \widehat{\boldsymbol{m}}_{\boldsymbol{1}} = \boldsymbol{k} * \left(\widehat{\boldsymbol{m}}_{\boldsymbol{2}}^{\boldsymbol{T}} * \boldsymbol{E} * \widehat{\boldsymbol{m}}_{\boldsymbol{1}}\right) = \boldsymbol{0}$.

The solution is to decompose the Essential matrix $\boldsymbol{E} = [\boldsymbol{R}^{\boldsymbol{T}} * \boldsymbol{T}]_{\boldsymbol{x}} * \boldsymbol{R}^{\boldsymbol{T}} = \boldsymbol{S} * \boldsymbol{R}^{\boldsymbol{T}}$, where *S* is a skew-symmetric matrix. This solution can be decomposed using the singular value decomposition, i.e. $\boldsymbol{E} = \boldsymbol{U} * \boldsymbol{D} * \boldsymbol{V}^{\boldsymbol{T}}$. It can also be shown (see book of Hartley and Zisserman chap9, https://en.wikipedia.org/wiki/Essential_matrix) that a skew symmetric matrix can be decomposed as: $\boldsymbol{S} = \boldsymbol{U} * \boldsymbol{Z} * \boldsymbol{U}^{\boldsymbol{T}}$, where $Z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$. $\boldsymbol{R}^{\boldsymbol{T}} = \boldsymbol{U} * \boldsymbol{W} * \boldsymbol{V}^{\boldsymbol{T}}$ or $\boldsymbol{R}^{\boldsymbol{T}} = \boldsymbol{U} * \boldsymbol{W}^{-\boldsymbol{1}} * \boldsymbol{V}^{\boldsymbol{T}}$, where $W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$. The Matlab function **`relativeCameraPose`** needs to pass as arguments the Essential or Fundamental matrix, the camera parameters, and image points. However, inside it uses the function **`decomposeEssentialMatrix(E)`**. This function checks the 4 possible solutions for the rotation. The function **`chooseRealizableSolution`** allows one to choose the real solution, i.e. the one that has the points in front of the camera (see following Figure from Hartley and Zisserman).

Note: In case that the robot is mounted on the tip of the robot, a very good estimation of the pose is known, and it is possible to choose the right solution using only **`decomposeEssentialMatrix(E)`**.



Four possible solutions (from Hartley and Zisserman)

Matlab Internal Decomposition of Essential matrix

```
function [Rs, Ts] = decomposeEssentialMatrix(E)

% Fix E to be an ideal essential matrix
[U, D, V] = svd(E);
e = (D(1,1) + D(2,2)) / 2;
D(1,1) = e;
D(2,2) = e;
D(3,3) = 0;
E = U * D * V';

[U, ~, V] = svd(E);

W = [0 -1 0; 1 0 0; 0 0 1];
Z = [0 1 0; -1 0 0; 0 0 0];

% Possible rotation matrices
R1 = U * W * V';
R2 = U * W' * V';

% Force rotations to be proper, i. e. det(R) = 1
if det(R1) < 0
    R1 = -R1;
end

if det(R2) < 0
    R2 = -R2;
end

% Translation vector
Tx = U * Z * U';
t = [Tx(3, 2), Tx(1, 3), Tx(2, 1)];

Rs = cat(3, R1, R1, R2, R2);
Ts = cat(1, t, -t, t, -t);

% Determine which of the 4 possible solutions is physically
realizable. A physically realizable solution is the one which puts
reconstructed 3D points in front of both cameras. There could be two
solutions if the R and t are extracted from homography matrix
function [R, t, validFraction] = chooseRealizableSolution(Rs, Ts,
cameraParams1, cameraParams2, points1, points2)
numNegatives = zeros(1, size(Ts, 1));

camMatrix1 = cameraMatrix(cameraParams1, eye(3), [0 0 0]);
for i = 1:size(Ts, 1)
    camMatrix2 = cameraMatrix(cameraParams2, Rs(:,:,i)', Ts(i, :));
    m1 = triangulateMidPoint(points1, points2, camMatrix1,
camMatrix2);
    m2 = bsxfun(@plus, m1 * Rs(:,:,i)', Ts(i, :));
    numNegatives(i) = sum((m1(:,3) < 0) | (m2(:,3) < 0));
end

val = min(numNegatives);
idx = find(numNegatives == val);

validFraction = 1 - (val / size(points1, 1));

R = zeros(3, 3, length(idx), 'like', Rs);
t = zeros(length(idx), 3, 'like', Ts);
```

```
for n = 1 : length(idx)
    R0 = Rs(:,:,idx(n))';
    t0 = Ts(idx(n), :);

    tNorm = norm(t0);
    if tNorm ~= 0
        t0 = t0 ./ tNorm;
    end
    R(:,:,n) = R0;
    t(n, :) = t0;
end
```