1. check if input is valid:
   (1) correspondences >= 8
   (2) points_0.size() = points_1.size()
2. normalize process - to get W(Wf = 0):
   - construct transform matrix:

     tx = sum(x coordinates) / N

     ty = sum(y coordinates) / N

     (tx, ty) - use the image center as new origin of coordinate system

     avg_distance = sum(distance from p to (tx, ty) - NOT squared distance):

     ```
     // scale factor
     double sum_dist = 0;
     for (const auto& p : points)
         sum_dist += sqrt((p.x() - tx) * (p.x() - tx) + (p.y() - ty) * (p.y() - ty));
     double avg_dc = sum_dist / N;
     ```

     where 'points' is the input 2d points set

     scale = sqrt(2) / avg_distance
   - normalize points:

     p_normalized = transform_matrix * p

     points_0 and points_1 both are normalized
3. get Fundamental matrix
   - use SVD to solve W and get initial_F

     construct W - use **normalized** points

     $$\boxed{\begin{cases} p_i = (u_i, v_i, 1) \\ p_i' = (u_i', v_i', 1) \end{cases} + \quad p'^T F\, p = 0}$$

     $$\Downarrow$$

     $$\begin{bmatrix} u_i u_i' & v_i u_i' & u_i' & u_i v_i' & v_i v_i' & v_i' & u_i & v_i & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0$$

     $$\begin{bmatrix} u_1u_1' & v_1u_1' & u_1' & u_1v_1' & v_1v_1' & v_1' & u_1 & v_1 & 1 \\ u_2u_2' & v_2u_2' & u_2' & u_2v_2' & v_2v_2' & v_2' & u_2 & v_2 & 1 \\ u_3u_3' & v_3u_3' & u_3' & u_3v_3' & v_3v_3' & v_3' & u_3 & v_3 & 1 \\ u_4u_4' & v_4u_4' & u_4' & u_4v_4' & v_4v_4' & v_4' & u_4 & v_4 & 1 \\ u_5u_5' & v_5u_5' & u_5' & u_5v_5' & v_5v_5' & v_5' & u_5 & v_5 & 1 \\ u_6u_6' & v_6u_6' & u_6' & u_6v_6' & v_6v_6' & v_6' & u_6 & v_6 & 1 \\ u_7u_7' & v_7u_7' & u_7' & u_7v_7' & v_7v_7' & v_7' & u_7 & v_7 & 1 \\ u_8u_8' & v_8u_8' & u_8' & u_8v_8' & v_8v_8' & v_8' & u_8 & v_8 & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0$$

     solve W:

     (1) Matrix W(m, n); // W is a m by n matrix, m(=rows), n(=cols)

     (2) svd_decompose(W, U, S, V); // W = U * S * V.transpose()

```
// solve W using SVD
Matrix U(m, m, 0.0);   // initialized with 0s
Matrix S(m, n, 0.0);   // initialized with 0s
Matrix V(n, n, 0.0);   // initialized with 0s
svd_decompose(W, U, S, V);

// Form F using the last column of V
Vector vlc = V.get_column(V.cols() - 1);  // get the last column of V
F.set_row(0, { vlc[0], vlc[1], vlc[2] });
F.set_row(1, { vlc[3], vlc[4], vlc[5] });
F.set_row(2, { vlc[6], vlc[7], vlc[8] });
```

- use SVD to decompose initial_F
- rank-2 approximation - make S(2, 2) = 0
- denormalize F

$$F = T'^T F_q T$$

- scale F such that F(2, 2) = 1.0

```
// decompose F using SVD
int m = initial_F.rows();
int n = initial_F.cols();

Matrix U(m, m, 0.0);   // initialized with 0s
Matrix S(m, n, 0.0);   // initialized with 0s
Matrix V(n, n, 0.0);   // initialized with 0s
svd_decompose(initial_F, U, S, V);

// rank-2 approximation - make S(2, 2) = 0
S(2, 2) = 0;

// update F
F = U * S * V.transpose();

// denormalize F: F = T'_transpose() * F * T scale F such that F(2, 2) = 1.0
F = T_.transpose() * F * T;

// scale F such that F(2, 2) = 1.0 (F is up to scale)


const double scale = 1.0 / F(2, 2);
F = F * scale;  // scale F
```

4. get Essential matirx

- $E = K^T F K$ where K is the intrinsic matrix

If TWO different cameras are used:

$$E = [t_\times]R = K'^T F K$$

K - camera 1 (usually camera 1 CRS is also the World CRS)

K' - camera 2

5. get possible R, t combinations from E (4 combinations in total)

- Essential matrix from fundamental matrix
- Relative pose from essential matrix
  - SVD of E
  - determinant(R) > 0
    - Two potential values
  - T up to a sign
    - Two potential values
    - Last column of U
    - Corresponds to smallest singular value

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad E = U\Sigma V^T$$

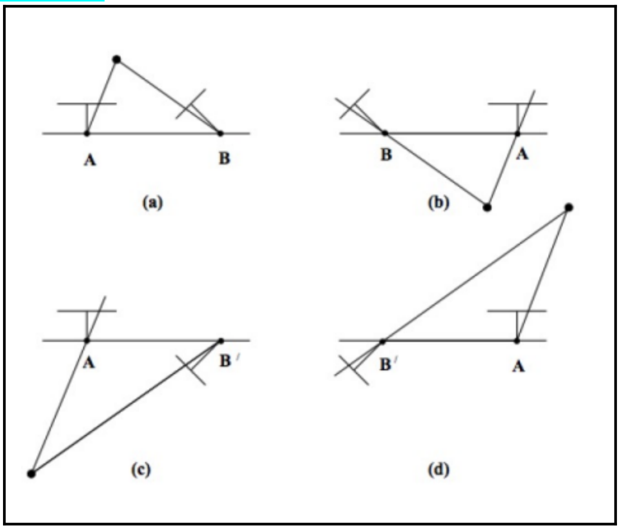
$$R = (\det UWV^T)UWV^T \text{ or } (\det UW^TV^T)UW^TV^T$$

$$t = \pm U \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \pm u_3$$

24

In practice, det(UWV.transpose()) and det(UW.transpose()V.transpose()) are both > 0 and (should be) equal(or close) to 1

6. find best R, t



Four different settings of R, t

in order to find best R,t, we need to use correspondences from two pictures to recover the 3d points, and for these 3d points, they need to:

(1) z coordinates in World CRS > 0 (camera-1 World CRS)

(2) z coordinates in Relative CRS >0 (camera-2 Relative CRS)

transform point3d from camera 1 to camera 2:

Q = R * P + t

recover 3d points from correspondences

Two image points

$$p = MP = (x, y, 1)$$

$$p' = M'P = (x', y', 1)$$

By the definition of the cross product

$$p \times (MP) = 0$$

Similar constraints can also be formulated for p' and M'.

$$x(M_3P) - (M_1P) = 0$$
$$y(M_3P) - (M_2P) = 0$$
$$x(M_2P) - y(M_1P) = 0$$

$$A = \begin{bmatrix} xM_3 - M_1 \\ yM_3 - M_2 \\ x'M'_3 - M'_1 \\ y'M'_3 - M'_2 \end{bmatrix}$$

$$AP = 0$$

M = K[I 0] - M is 3 by 4 matrix
M' = K[R t] - M'is 3 by 4 matrix
M1, M2, M3 - each row of M
the same with M'

A is a 4 by 4 matrix;

use SVD to decompose A, the last column of V has 4 elements, and its cartesian() coordinates is the coordinates of 3d points in World CRS

```cpp
// construct matrix A:
//  xM3  - M1
//  yM3  - M2
// x'M3' - M1'
// y'M3' - M2'
double x  = points_0[i].x(), y  = points_0[i].y();
double x_ = points_1[i].x(), y_ = points_1[i].y();

Matrix44 A;

A.set_row(0, x  * M3  - M1 );
A.set_row(1, y  * M3  - M2 );
A.set_row(2, x_ * M3_ - M1_);
A.set_row(3, y_ * M3_ - M2_);

// solve A using SVD
int m = A.rows();
int n = A.cols();
Matrix U(m, m, 0.0);   // initialized with 0s
Matrix S(m, n, 0.0);   // initialized with 0s
Matrix V(n, n, 0.0);   // initialized with 0s
svd_decompose(A, U, S, V);

Vector4D vlc = V.get_column(V.cols() - 1);  // get the last column of V
points_3d.emplace_back();
points_3d.back() = vlc.cartesian();  // add the 3d point to the result vector
```

transform these recovered 3d points to camera-2 CRS:
Q = R * P + t where P is the 3d point in World CRS

Then for best R, t:

3D points must be in front of both cameras

- First camera
  - P.z > 0 ?
- Second camera
  - P in 2nd camera's coordinate system: Q = R * P + t
  - Q.z > 0 ?