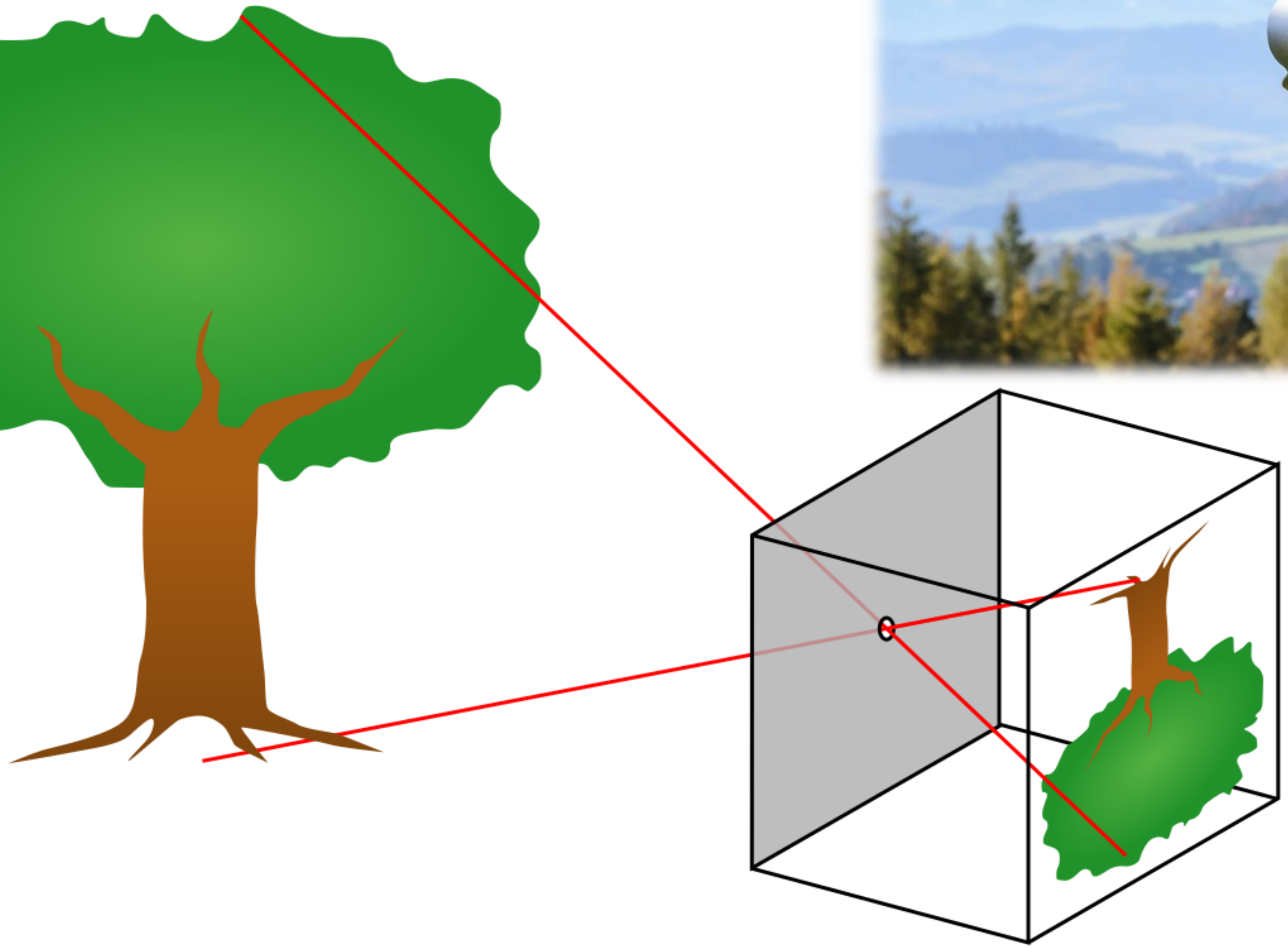
# Robotic Mapping & Localization

**Kaveh Fathian**
Assistant Professor
Computer Science Department
Colorado School of Mines

**Lec09: Two-View Geometry - Examples**

# Pinhole imaging

real-world
object



copy of real-world
object (inverted
& scaled)

# The camera as a coordinate transformation

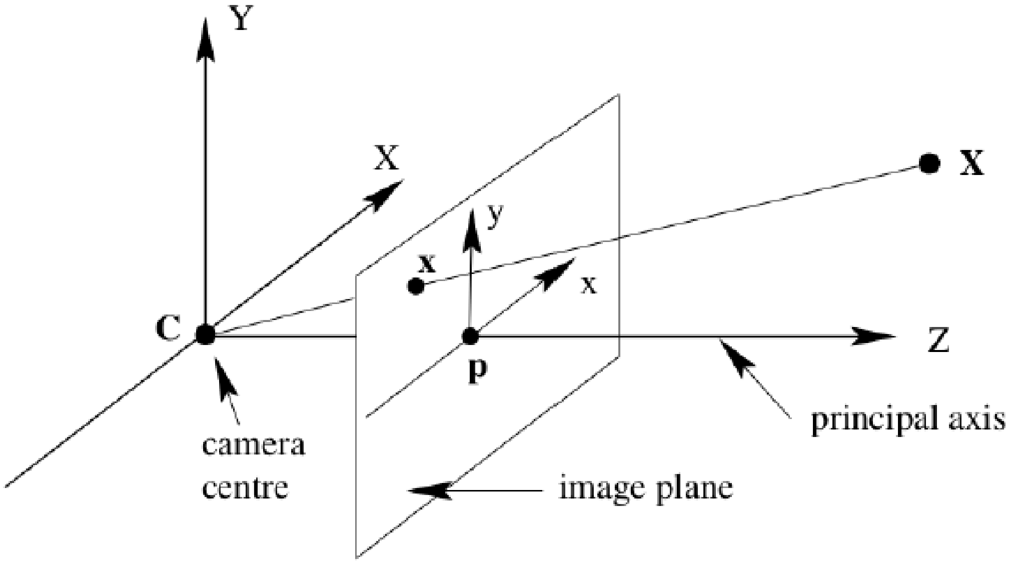
homogeneous coordinates

$$\mathbf{x} = \mathbf{P}\mathbf{X}$$

2D image point

camera matrix

3D world point



# The camera as a coordinate transformation

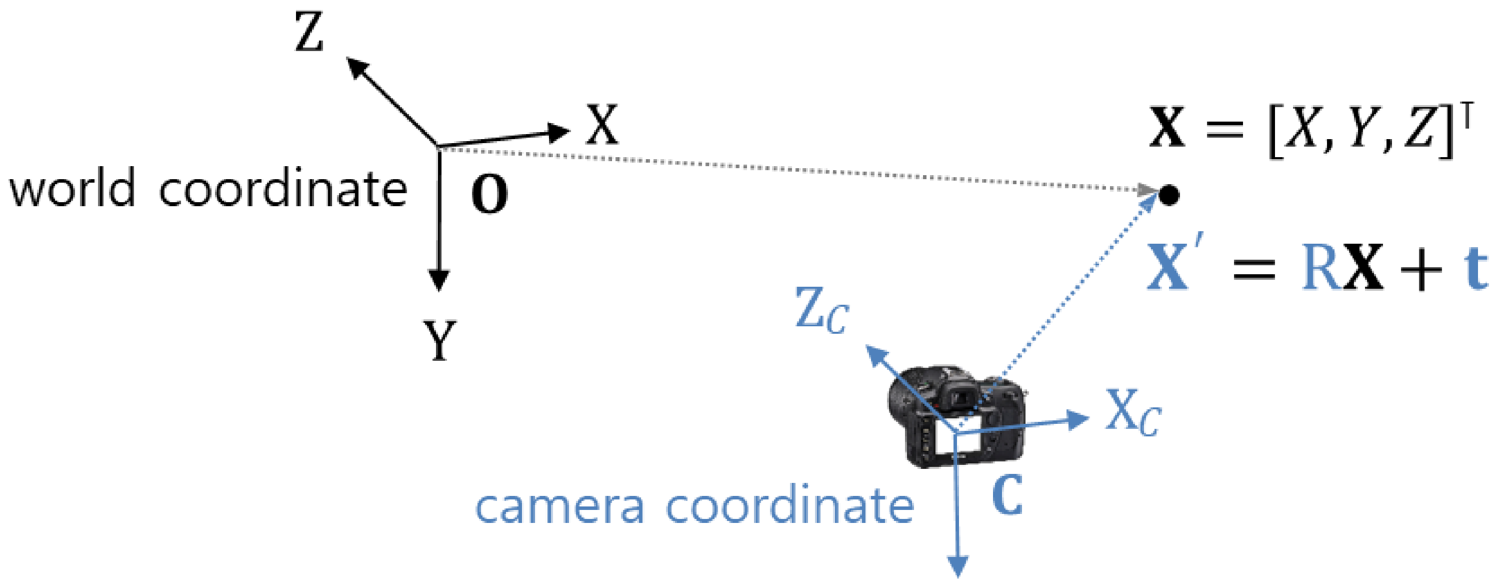
$$\mathbf{x} = \mathbf{P}\mathbf{X}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

| | | |
|---|---|---|
| homogeneous image coordinates 3 x 1 | camera matrix 3 x 4 | homogeneous world coordinates 4 x 1 |

# General pinhole camera matrix

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}]$$




$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|c} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{array}\right]$$

intrinsic parameters

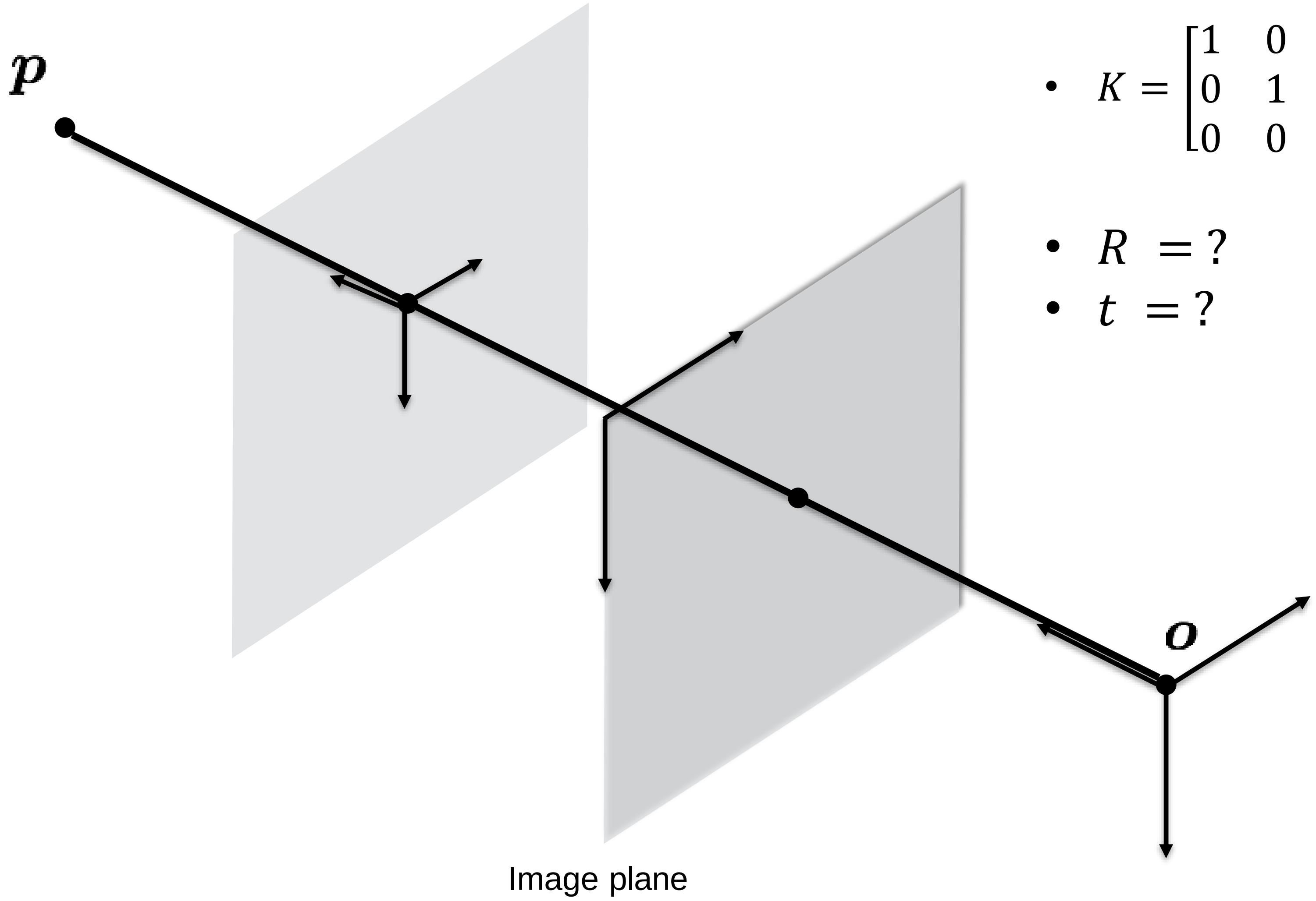
extrinsic parameters

$$\mathbf{R} = \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{bmatrix} \quad \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

3D rotation

3D translation

# Example




- $P = \begin{bmatrix} 1 & -2 & 0 & 9 \\ 0 & -2 & 1 & 9 \\ 0 & -1 & 0 & 4 \end{bmatrix}$

- $K = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}, \; K^{-1} = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix}$

- $R \; = ?$
- $t \; = ?$

# Example

Consider 2 camera views, with camera frames ${}^{1}F$ and ${}^{2}F$, and the world frame denoted by ${}^{w}F$. Assume that the $x$-$z$ axes of camera frames, and the $x$-$y$ axes of world frame are on the same plane, and they are located relative to each other as shown in the figure.

Assume that 3D points $p_1, \dots, p_6$ have the following coordinates (in the world frame)

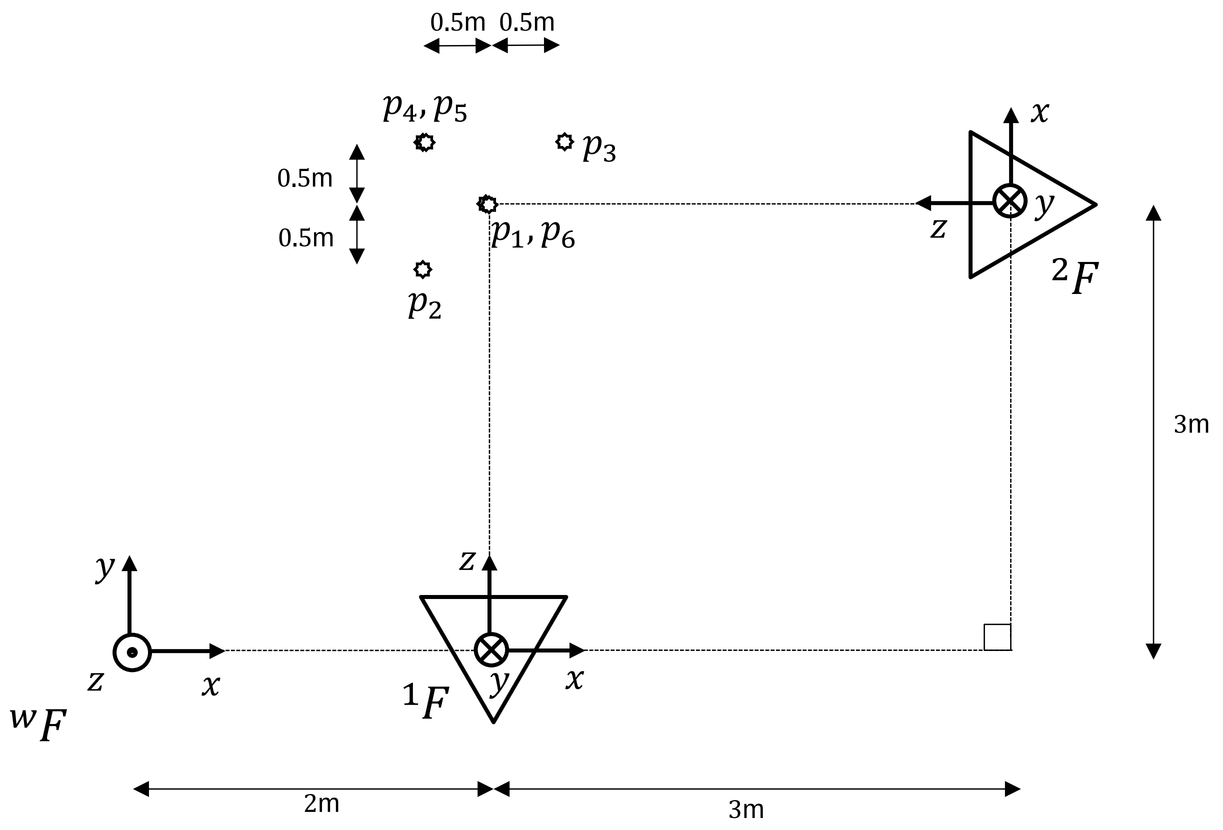
$$ {}^{w}p_1 = \begin{bmatrix} 2 \\ 3 \\ 0 \end{bmatrix}, \quad {}^{w}p_2 = \begin{bmatrix} 1.5 \\ 2.5 \\ 0 \end{bmatrix}, \quad {}^{w}p_3 = \begin{bmatrix} 2.5 \\ 3.5 \\ 0 \end{bmatrix}, $$

$$ {}^{w}p_4 = \begin{bmatrix} 1.5 \\ 3.5 \\ 0.5 \end{bmatrix}, \quad {}^{w}p_5 = \begin{bmatrix} 1.5 \\ 3.5 \\ -0.5 \end{bmatrix}, \quad {}^{w}p_6 = \begin{bmatrix} 2 \\ 3 \\ -0.5 \end{bmatrix} $$

Assume that the camera calibration matrix is

$$ K = \begin{bmatrix} 10 & 0 & 100 \\ 0 & 10 & 200 \\ 0 & 0 & 1 \end{bmatrix} $$

**Top view of 2 camera frames & 3D points**



# Questions

- Compute the pose of camera 1 in the world frame, that is, the rotation matrix and translation vector $({}^{w}R_1,\ {}^{w}t_1)$.
- Compute the relative pose of camera 2 in camera 1's frame, that is, $({}^{1}R_2,\ {}^{1}t_2)$.
- Compute the pose of camera 2 in the world frame, $({}^{w}R_2,\ {}^{w}t_2)$, using homogeneous transforms ${}^{w}H_1$, ${}^{1}H_2$ , constructed from matrices computed above.

Solution:

```
// pose of camera 1 in world frame
Eigen::Matrix3d Rw1;
Rw1 << 1, 0, 0,
       0, 0, 1,
       0, -1, 0;
Eigen::Vector3d tw1(2,0,0);

// pose of camera 2 in camera 1 frame
Eigen::Matrix3d R12;
R12 << 0, 0, -1,
       0, 1, 0,
       1, 0, 0;
Eigen::Vector3d t12(3,0,3);

Eigen::Matrix4d Hw1 = Eigen::Matrix4d::Identity();
Hw1.block<3, 3>(0, 0) = Rw1;  // rotation part
Hw1.block<3, 1>(0, 3) = tw1;  // translation part

Eigen::Matrix4d H12 = Eigen::Matrix4d::Identity();
H12.block<3, 3>(0, 0) = R12;  // rotation part
H12.block<3, 1>(0, 3) = t12;  // translation part

Eigen::Matrix4d Hw2 = Hw1 * H12;
std::cout << "Homogeneous Transformation Matrix Hw2:\n" << Hw2 << "\n\n";
```

```
Homogeneous Transformation Matrix Hw2:
 0  0 -1  5
 1  0  0  3
 0 -1  0  0
 0  0  0  1
```

# Questions

- What are the 3D coordinates of points in the camera frames? That is, compute ${}^{1}p_i$ and ${}^{2}p_i$ for all $i = 1, \ldots, 6$. To do this, compute & use homogeneous transformations ${}^{1}H_w$ and ${}^{2}H_w$ to transform coordinates of points ${}^{w}p_i$ into camera frames.

Solution:

```cpp
// coordiantes of points in the camera frames
Eigen::MatrixXd Ptsw;
Ptsw.resize(4,6);
Ptsw << 2, 1.5, 2.5, 1.5, 1.5, 2  ,
        3, 2.5, 3.5, 3.5, 3.5, 3  ,
        0, 0  , 0  , 0.5,-0.5,-0.5,
        1, 1  , 1  , 1  , 1  , 1  ;

Eigen::Matrix4d H1w = Hw1.inverse();
Eigen::Matrix4d H2w = Hw2.inverse();

Eigen::MatrixXd Pts1 = H1w * Ptsw;
Eigen::MatrixXd Pts2 = H2w * Ptsw;
std::cout << "Homogeneous coordiantes of points in camera frame 1:\n" << Pts1 << "\n\n";
std::cout << "Homogeneous coordiantes of points in camera frame 2:\n" << Pts2 << "\n\n";
```

```
Homogeneous coordiantes of points in camera frame 1:
   0 -0.5  0.5 -0.5 -0.5    0
   0    0    0 -0.5  0.5  0.5
   3  2.5  3.5  3.5  3.5    3
   1    1    1    1    1    1

Homogeneous coordiantes of points in camera frame 2:
   0 -0.5  0.5  0.5  0.5    0
   0    0    0 -0.5  0.5  0.5
   3  3.5  2.5  3.5  3.5    3
   1    1    1    1    1    1
```

# Questions

- ❑ Compute camera matrices $P_1$ and $P_2$.

Solution:

```
// calibration matrix
Eigen::Matrix3d K;
K << 10,  0, 100,
      0, 10, 200,
      0,  0,   1;

// extrinsic matrix
Eigen::Matrix<double, 3, 4> Ext1;
Ext1.block<3, 3>(0, 0) = H1w.block<3, 3>(0, 0);  // rotation part
Ext1.block<3, 1>(0, 3) = H1w.block<3, 1>(0, 3);  // translation part
// camera matrix
Eigen::Matrix<double, 3, 4> P1 = K * Ext1;
std::cout << "Camera matrix P1:\n" << P1 << "\n\n";

// extrinsic matrix
Eigen::Matrix<double, 3, 4> Ext2;
Ext2.block<3, 3>(0, 0) = H2w.block<3, 3>(0, 0);  // rotation part
Ext2.block<3, 1>(0, 3) = H2w.block<3, 1>(0, 3);  // translation part
// camera matrix
Eigen::Matrix<double, 3, 4> P2 = K * Ext2;
std::cout << "Camera matrix P2:\n" << P2 << "\n\n";
```

```
Camera matrix P1:
 10 100   0 -20
  0 200 -10  -0
  0   1   0  -0

Camera matrix P2:
-100   10    0  470
-200    0  -10 1000
  -1    0    0    5
```

# Questions

- ❑ What are 2D ***pixel*** coordinates of points on each image? Use camera matrices from the previous step to compute them.
- ❑ What are the 2D ***Cartesian*** coordinates of points in the camera frame? Note that they are in the camera frame (not image frame).

Solution:

```
//pixel coordiantes
Eigen::MatrixXd Pts1pix = P1 * Ptsw;
for (int i=0; i < Pts1pix.cols(); ++i) {Pts1pix.col(i) = Pts1pix.col(i) / Pts1pix(2, i);} // divide by last element due to homogeneous coordinates
std::cout << "Image points pixel coordinates in camera 1:\n" << Pts1pix << "\n\n";

Eigen::MatrixXd Pts2pix = P2 * Ptsw;
for (int i=0; i < Pts2pix.cols(); ++i) {Pts2pix.col(i) = Pts2pix.col(i) / Pts2pix(2, i);} // divide by last element due to homogeneous coordinates
std::cout << "Image points pixel coordinates in camera 2:\n" << Pts2pix << "\n\n";

//euclidean  coordiantes
Eigen::MatrixXd Pts1euc = K.inverse() * Pts1pix;
std::cout << "Image points euclidean coordinates in camera 1:\n" << Pts1euc << "\n\n";

Eigen::MatrixXd Pts2euc = K.inverse() * Pts2pix;
std::cout << "Image points euclidean coordinates in camera 2:\n" << Pts2euc << "\n\n";
```

```
Image points pixel coordinates in camera 1:
    100      98 101.429 98.5714 98.5714     100
    200     200     200 198.571 201.429 201.667
      1       1       1       1       1       1

Image points pixel coordinates in camera 2:
    100 98.5714     102 101.429 101.429     100
    200     200     200 198.571 201.429 201.667
      1       1       1       1       1       1
```

```
Image points euclidean coordinates in camera 1:
        0      -0.2  0.142857 -0.142857 -0.142857         0
        0         0         0 -0.142857  0.142857  0.166667
        1         1         1         1         1         1

Image points euclidean coordinates in camera 2:
        0 -0.142857       0.2  0.142857  0.142857         0
        0         0         0 -0.142857  0.142857  0.166667
        1         1         1         1         1         1
```

# Questions

- ❑ Using relative pose $({}^1R_2, {}^1t_2)$, compute the essential matrix $E$ between the camera views.
- ❑ Using $E$ and camera calibration matrix $K$ (provided earlier), compute the fundamental matrix $F$.

Solution:

```cpp
// essential matrix
Eigen::Matrix3d Tx;
Tx <<       0, -t12(2),  t12(1),
      t12(2),       0, -t12(0),
     -t12(1),  t12(0),       0;

Eigen::Matrix3d E = Tx * R12;
std::cout << "Essential matrix:\n" << E << "\n\n";

//fundamental matrix
Eigen::Matrix3d F = K.inverse().transpose() * E * K.inverse();
std::cout << "Fundamental matrix:\n" << F << "\n\n";
```

```
Essential matrix:
 0 -3  0
-3  0 -3
 0  3  0

Fundamental matrix:
    0 -0.03     6
-0.03     0   2.7
    6   3.3 -1200
```

# Questions

- Show that the essential matrix constraint $m_i^T\ E\ m_i' = 0$ holds for all points $i = 1, \dots, 6$. Here, $m_i$ and $m_i'$ are the Cartesian coordinates of image points in the camera frames, which was computed earlier. Hint: Make sure that points $m_i$ and $m_i'$ are in the right camera frames. That is, $m_i$ is either in ${}^1F$ or ${}^2F$ depending on how you computed $E$.

- Show that the fundamental matrix constraint $m_i^T\ F\ m_i' = 0$ holds for all points. Here, $m_i$ and $m_i'$ are the pixel coordinates of image points in the image planes.

- What are the epipolar lines in camera frame ${}^2F$? Compute the line coefficients using the essential matrix $E$ and points $m_i'$ .

Solu

```cpp
// essential matrix constraint
for (int i=0; i<Pts1euc.cols(); ++i) {
    Eigen::Vector3d m1 = Pts1euc.col(i).head<3>();
    Eigen::Vector3d m2 = Pts2euc.col(i).head<3>();
    double error = m1.transpose() * E * m2;
    std::cout << "pair " << i << ": essential mat constraint error = " << error << "\n";
}


// fundamental matrix constraint
for (int i=0; i<Pts1pix.cols(); ++i) {
    Eigen::Vector3d m1 = Pts1pix.col(i).head<3>();
    Eigen::Vector3d m2 = Pts2pix.col(i).head<3>();
    double error = m1.transpose() * F * m2;
    std::cout << "pair " << i << ": fundamental mat constraint error = " << error << "\n";
}
```

```
pair 0: essential mat constraint error = 0
pair 1: essential mat constraint error = 0
pair 2: essential mat constraint error = 0
pair 3: essential mat constraint error = 7.21645e-16
pair 4: essential mat constraint error = -7.77156e-16
pair 5: essential mat constraint error = 0
pair 0: fundamental mat constraint error = -1.27898e-13
pair 1: fundamental mat constraint error = -1.13687e-13
pair 2: fundamental mat constraint error = -9.23706e-14
pair 3: fundamental mat constraint error = 7.10543e-14
pair 4: fundamental mat constraint error = 1.42109e-14
pair 5: fundamental mat constraint error = -1.06581e-13
```

# Questions

- Show that the essential matrix constraint $m_i^T \, E \, m_i' = 0$ holds for all points $i = 1, \dots, 6$. Here, $m_i$ and $m_i'$ are the Cartesian coordinates of image points in the camera frames, which was computed earlier. Hint: Make sure that points $m_i$ and $m_i'$ are in the right camera frames. That is, $m_i$ is either in ${}^1F$ or ${}^2F$ depending on how you computed $E$.

- Show that the fundamental matrix constraint $m_i^T \, F \, m_i' = 0$ holds for all points. Here, $m_i$ and $m_i'$ are the pixel coordinates of image points in the image planes.

- What are the epipolar lines in camera frame ${}^2F$? Compute the line coefficients using the essential matrix $E$ and points $m_i'$ .

Solution:

```
// epipolar lines
std::cout << "epipoloar lines in camera 2: \n" << E.transpose() * Pts1euc  << "\n\n";
```

```
epipoloar lines in camera 2:
       0         0         0  0.428571 -0.428571      -0.5
       3       3.6   2.57143   3.42857   3.42857         3
       0         0         0  0.428571 -0.428571      -0.5
```

# Questions

- Points $p_1, p_2, p_3, p_6$ are coplanar, therefore, a [homography matrix ](homography matrix) can be computed for their images. Compute the homography matrix $H$ corresponding to these points.
  Hint: you will need the relative pose between the cameras, normal vector $n$ for the plane that contains the points, and its distance $d$ to the origin of the camera frame. Make sure that all values are in the *same* camera frame.
  Hint: ChatGPT's answer for how to compute $n$ and $d$ using 3 point coordinates:

what is the normal of plane constructed from three 3D points?

The normal of a plane constructed from three 3D points can be found using the cross product of two vectors that lie on the plane. Given three points in 3D space:

$$P_1(x_1, y_1, z_1), \quad P_2(x_2, y_2, z_2), \quad P_3(x_3, y_3, z_3)$$

**Steps to find the normal:**

1. **Construct two vectors on the plane** using the given points:

$$\mathbf{v_1} = P_2 - P_1 = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

$$\mathbf{v_2} = P_3 - P_1 = (x_3 - x_1, y_3 - y_1, z_3 - z_1)$$

2. **Compute the cross product** of the two vectors:

$$\mathbf{n} = \mathbf{v_1} \times \mathbf{v_2}$$

The cross product is given by the determinant:

$$\mathbf{n} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ (x_2 - x_1) & (y_2 - y_1) & (z_2 - z_1) \\ (x_3 - x_1) & (y_3 - y_1) & (z_3 - z_1) \end{vmatrix}$$

Expanding the determinant:

$$\mathbf{n} = \begin{bmatrix} (y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_1) \\ (z_2 - z_1)(x_3 - x_1) - (x_2 - x_1)(z_3 - z_1) \\ (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) \end{bmatrix}$$

This vector $\mathbf{n}$ is the normal of the plane.

how can I compute the distance of the plane constructed from these points to the origin?

To compute the distance of the plane constructed from three points to the origin $(0,0,0)$, follow these steps:

**Step 1: Find the Plane Equation**

A plane in 3D space can be represented by the equation:

$$Ax + By + Cz + D = 0$$

where:

- $(A, B, C)$ is the **normal vector** to the plane (which we computed earlier using the cross product),
- $D$ is a constant that can be determined using one of the three points.

Using a known point $P_1(x_1, y_1, z_1)$ on the plane:

$$Ax_1 + By_1 + Cz_1 + D = 0$$

Solving for $D$:

$$D = -(Ax_1 + By_1 + Cz_1)$$

**Step 2: Compute the Distance to the Origin**

The perpendicular distance $d$ from the origin $(0,0,0)$ to the plane is given by:

$$d = \frac{|Ax_0 + By_0 + Cz_0 + D|}{\sqrt{A^2 + B^2 + C^2}}$$

Since the origin is $(0,0,0)$, this simplifies to:

$$d = \frac{|D|}{\sqrt{A^2 + B^2 + C^2}}$$

where $A, B, C$ are the components of the normal vector.

**Final Formula**

$$d = \frac{|-(Ax_1 + By_1 + Cz_1)|}{\sqrt{A^2 + B^2 + C^2}}$$

This gives the shortest distance from the origin to the plane.

# Questions

- Points $p_1, p_2, p_3, p_6$ are coplanar, therefore, a [homography matrix](homography matrix) can be computed for their images. Compute the homography matrix $H$ corresponding to these points.
  Hint: you will need the relative pose between the cameras, normal vector $n$ for the plane that contains the points, and its distance $d$ to the origin of the camera frame. Make sure that all values are in the *same* camera frame.

$$^1H_2 = {}^1R_2 - \frac{^1t_2 \, . \, ^2n^T}{^2d}$$

Solution:

```
// normal vector computation
Eigen::Vector3d p1 = Pts2.col(0).head<3>();
Eigen::Vector3d p2 = Pts2.col(1).head<3>();
Eigen::Vector3d p3 = Pts2.col(2).head<3>();
Eigen::Vector3d p6 = Pts2.col(5).head<3>();

Eigen::Vector3d v1 = p2 - p1;
Eigen::Vector3d v2 = p6 - p1;
Eigen::Vector3d nrm = v1.cross(v2);
nrm.normalize(); // normalize the vector
std::cout << "plane normal in camera frame 1: \n" << nrm << "\n\n";

// distance to plane
double A = nrm(0), B = nrm(1), C = nrm(2);
double D = -(A * p6(0) + B * p6(1) + C * p6(2));
double dist = std::abs(D) / nrm.norm();
std::cout << "plane equation: " << A << "x + " << B << "y + " << C << "z + " << D << " = 0\n";
std::cout << "distance from origin to plane: " << dist << "\n\n";

//homography matrix
Eigen::Matrix3d H = R12 - (t12 * nrm.transpose()) / dist;
std::cout << "homography matrix H:\n" << H << "\n\n";
```

```
plane normal in camera frame 1:
-0.707107
        0
-0.707107

plane equation: -0.707107x + 0y + -0.707107z + 2.12132 = 0
distance from origin to plane: 2.12132

homography matrix H:
           1            0 -2.22045e-16
           0            1            0
           2            0            1
```

# Questions

- Show that the homography constraint $Hm_i = m_i'$ holds for all points $i = 1, 2,3,6$. Here, $m_i$ and $m_i'$ are the cartesian coordinates of image points in the camera frames.
  Hint: Make sure that points $m_i$ and $m_i'$ are in correct camera frames.

- Test the homography constraint $Hm_i = m_i'$ for points $i = 4, 5$. Does this constraint hold? If not, why?

Solution:

```
// check homography constraint
for (int i=0; i<Pts1euc.cols(); ++i) {
      Eigen::Vector3d m1 = Pts1euc.col(i).head<3>();
      Eigen::Vector3d m2 = Pts2euc.col(i).head<3>();
      Eigen::Vector3d Hm2 = H * m2;
      for (int i=0; i<Hm2.cols(); ++i) {Hm2.col(i) = Hm2.col(i) / Hm2(2, i);} // divide by last element due to homogeneous coordinates
      Eigen::Vector3d residual = Hm2 - m1;

      double error = residual.norm();
      std::cout << "pair " << i << ": homography constraint error = " << error << "\n";
}
```

```
pair 0: homography constraint error = 2.22045e-16
pair 1: homography constraint error = 5.55112e-17
pair 2: homography constraint error = 8.88178e-16
pair 3: homography constraint error = 0.255945
pair 4: homography constraint error = 0.255945
pair 5: homography constraint error = 2.23773e-16
```

# Questions

Implement the DLT algorithm using coplanar points $i = 1, 2,3,6$. To do so, follow the steps below and show all computed variables.

- ❑ Construct matrix $A$ from point correspondences
- ❑ Compute SVD of $A = U\, S\, V^T$
- ❑ Take vector $h$ as the last column of $V$ (that is, the singular vector of the smallest singular value)
- ❑ Reshape $h$ into a 3x3 homography matrix $H$.

- ❑ Is the estimated homography matrix computed above the same as the matrix computed earlier? If not, what is the difference?

$$\mathbf{A}\boldsymbol{h} = \mathbf{0}$$

$$\begin{bmatrix} \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix} \\ \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix} \\ \vdots \\ \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Solution:

The homography matrix estimated from DLT could be different than the matrix computed earlier because the homography matrix is always defined up to a ***scale factor***

Also, as points $p_1, p_2, p_3$ are *colinear*, the homography matrix is *degenerate* and has more *degrees of freedom*

# Questions

```cpp
Eigen::Matrix3d computeHomographyDLT(const Eigen::MatrixXd& pts1, const Eigen::MatrixXd& pts2, const std::vector<int>& indices) {
    int n = indices.size();
    Eigen::MatrixXd A(2 * n, 9); // construct (2n x 9) matrix A matrix for DLT

    for (int i = 0; i < n; ++i) {
        int idx = indices[i];
        double x1 = pts1(0, idx);
        double y1 = pts1(1, idx);

        double x2 = pts2(0, idx);
        double y2 = pts2(1, idx);

        // first row for this point
        A(2*i, 0) = -x1;
        A(2*i, 1) = -y1;
        A(2*i, 2) = -1;
        A(2*i, 3) = 0;
        A(2*i, 4) = 0;
        A(2*i, 5) = 0;
        A(2*i, 6) = x1 * x2 ;
        A(2*i, 7) = y1 * x2;
        A(2*i, 8) = x2;
        // second row for this point
        A(2*i+1, 0) = 0;
        A(2*i+1, 1) = 0;
        A(2*i+1, 2) = 0;
        A(2*i+1, 3) = -x1;
        A(2*i+1, 4) = -y1;
        A(2*i+1, 5) = -1;
        A(2*i+1, 6) = x1 * y2;
        A(2*i+1, 7) = y1 * y2;
        A(2*i+1, 8) = y2;
    }

    Eigen::JacobiSVD<Eigen::MatrixXd> svd(A, Eigen::ComputeFullV); // SVD of A
    // solution is the last column of V (last row of V^T)
    Eigen::MatrixXd V = svd.matrixV();
    Eigen::VectorXd h = V.col(V.cols() - 1);

    // reshape h into a 3x3 homography matrix
    Eigen::Map<Eigen::Matrix3d> H(h.data());
    return H.transpose(); // eigen uses column-major storage, so we need to transpose
}
```

```cpp
std::vector<int> point_indices = {0, 1, 2, 5}; // use points 1, 2, 3, and 6 (1-based indexing)
Eigen::Matrix3d H_est = computeHomographyDLT(Pts2euc, Pts1euc, point_indices); // compute homography

H_est /= H_est(2, 2); // normalize the homography matrix
std::cout << "Estimated Homography Matrix H_est:\n" << H_est << "\n\n";


// check homography constraint
for (int i=0; i<Pts1euc.cols(); ++i) {
    Eigen::Vector3d m1 = Pts1euc.col(i).head<3>();
    Eigen::Vector3d m2 = Pts2euc.col(i).head<3>();
    Eigen::Vector3d Hm2 = H_est * m2;
    // divide by last element due to homogeneous coordinates
    for (int i=0; i<Hm2.cols(); ++i) {Hm2.col(i) = Hm2.col(i) / Hm2(2, i);}
    Eigen::Vector3d residual = Hm2 - m1;

    double error = residual.norm();
    std::cout << "pair " << i << ": homography constraint error = " << error << "\n";
}
```

```
Estimated Homography Matrix H_est:
           1 -1.66242e-16 -1.10828e-16
 1.88859e-16     0.480571  1.63281e-17
           2     -3.11658            1

pair 0: homography constraint error = 1.12024e-16
pair 1: homography constraint error = 1.39577e-16
pair 2: homography constraint error = 6.76369e-17
pair 3: homography constraint error = 0.24789
pair 4: homography constraint error = 0.318752
pair 5: homography constraint error = 2.89604e-16
```