Mathematical Foundations of Laser Plane Calibration Algorithms

A detailed breakdown of Plücker Line and RANSAC-based methods

Abstract

This document provides a rigorous mathematical exposition of two common methods for calibrating the extrinsic relationship of a camera and a line laser projector. The goal is to determine the equation of the plane formed by the laser light within the camera's coordinate system. We analyze the shared geometric utilities and then delve into two distinct algorithmic approaches: one based on the intersection of planes using Plücker line coordinates, and another based on robustly fitting a plane to a 3D point cloud using a weighted RANSAC algorithm. Each code segment is presented and followed by a detailed mathematical justification.

1 Shared Geometric and Calibration Utilities

This section details the foundational functions used by both calibration pipelines. These routines handle camera modeling, pose estimation, and fundamental geometric conversions.

1.1 Camera Model and Intrinsic Calibration

The first step in any vision-based metrology task is to calibrate the camera's intrinsic parameters. We model the camera using the standard pinhole model, augmented with distortion parameters.

1.1.1 The Pinhole Camera Model

A 3D point in the camera's coordinate system, $P_c = (X_c, Y_c, Z_c)^T$, is projected onto the 2D image plane at coordinates (u, v) by the following transformation:

$$s \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} \tag{1}$$

where:

- K is the 3×3 camera intrinsic matrix.
- (f_x, f_y) are the focal lengths in units of pixels.
- (c_x, c_y) is the principal point, i.e., the coordinates of the optical axis's intersection with the image sensor.
- s is a scalar depth factor.

Real lenses introduce non-linear distortions. These are modeled by a set of coefficients, typically for radial distortion $(k_1, k_2, k_3, \dots)$ and tangential distortion (p_1, p_2). The calibration process finds both the matrix K and these distortion coefficients.

1.1.2 Code Implementation: Camera Calibration

```
def calibrate_camera(images: Iterable[str],
                     pattern_size: Tuple[int, int],
                     objp: np.ndarray,
                     criteria) -> CameraCalibrationResult:
    """Perform intrinsic calibration over a list of image paths."""
    objpoints, imgpoints = [], []
    gray_shape = None
    # ... (loop to find checkerboard corners in images) ...
    # objpoints contains known 3D corner locations
    # imgpoints contains detected 2D corner locations
    ret, cam_mtx, dist_coefs, rvecs, tvecs = cv2.calibrateCamera(
        objpoints, imgpoints, gray_shape, None, None)
    return CameraCalibrationResult(cam_mtx=cam_mtx,
                                   dist_coefs=dist_coefs,
                                   rvecs=rvecs, tvecs=tvecs)
```

The ‘cv2.calibrateCamera‘ function implements Zhang's calibration method. It takes a set of known 3D points on a calibration pattern (‘objpoints‘) and their corresponding 2D projections (‘imgpoints‘) from multiple views. It then solves for the intrinsic matrix K and the distortion coefficients that minimize the overall reprojection error: the geometric distance between the projected 3D points and their detected 2D counterparts.

1.2 Checkerboard Pose and Plane Recovery

Once the camera is calibrated, we can determine the 3D position and orientation (pose) of the checkerboard relative to the camera for each image.

1.2.1 Mathematical Formulation

The relationship between the checkerboard's world coordinate system (where corners lie on the $Z_w = 0$ plane) and the camera's coordinate system is given by a rigid body transformation $[R|t]$:

$$P_c = RP_w + t \tag{2}$$

where R is a 3×3 rotation matrix and t is a 3×1 translation vector. The ‘cv2.solvePnP‘ function finds the $[R|t]$ that best aligns the 3D object points with their observed 2D image points, given the camera intrinsics.
A plane is defined by the equation $\mathbf{n} \cdot X + D = 0$, where $\mathbf{n}$ is the unit normal vector and D is the signed distance from the origin to the plane along the normal. The checkerboard lies on the $Z_w = 0$ plane in its own coordinate frame. Its normal vector in this frame is $\mathbf{n}_w = (0, 0, 1)^T$. To find the plane's representation in the camera frame, we transform this normal vector and a point on the plane.

1. **Normal Transformation**: The normal vector in the camera frame, $\mathbf{n}_c$, is found by applying the rotation:

$$\mathbf{n}_c = R\mathbf{n}_w = R \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3}$$

   This is simply the third column of the rotation matrix R.

2. **Offset Calculation**: The origin of the checkerboard frame, $P_w = (0, 0, 0)^T$, is transformed to the point $P_c = t$ in the camera frame. This point lies on the plane. Substituting it into the plane equation:

$$\mathbf{n}_c \cdot t + D = 0 \implies D = -\mathbf{n}_c \cdot t \tag{4}$$

1.2.2 Code Implementation

```
def board_plane_from_pose(R: np.ndarray, t: np.ndarray) -> np.ndarray:
    """Return plane [nx, ny, nz, D] for checkerboard (Z=0 in board frame)."""
    n = R @ np.array([0.0, 0.0, 1.0])
    p0 = t.reshape(3)
    D = -float(n @ p0)
    return np.array([*n, D], dtype=float)
```

This code is a direct implementation of the mathematics described above.

1.3 From 2D Image Line to 3D Plane

A straight line in the 2D image is the projection of a 3D plane that passes through the camera's optical center (the origin of the camera coordinate system).

1.3.1 Mathematical Formulation

Let the 2D image line be given by the equation $au + bv + c = 0$. We can relate the image coordinates (u, v) to a 3D point (X_c, Y_c, Z_c) using the pinhole model equations:

$$u = f_x \frac{X_c}{Z_c} + c_x \quad \text{and} \quad v = f_y \frac{Y_c}{Z_c} + c_y \tag{5}$$

Substituting these into the line equation:

$$a\left(f_x \frac{X_c}{Z_c} + c_x\right) + b\left(f_y \frac{Y_c}{Z_c} + c_y\right) + c = 0 \tag{6}$$

Multiplying by Z_c to clear the denominator:

$$a(f_x X_c + c_x Z_c) + b(f_y Y_c + c_y Z_c) + cZ_c = 0 \tag{7}$$

Rearranging terms into the plane equation form $\mathbf{n} \cdot X_c + D = 0$:

$$(af_x)X_c + (bf_y)Y_c + (ac_x + bc_y + c)Z_c = 0 \tag{8}$$

This gives us the plane parameters:

$$\mathbf{n} = (af_x, bf_y, ac_x + bc_y + c)^T \tag{9}$$
$$D = 0 \tag{10}$$

The offset D is zero, confirming the plane passes through the origin. The code normalizes $\mathbf{n}$ to make it a unit vector.

1.3.2 Code Implementation

```
def plane_from_image_line(a: float, b: float, c: float,
                          fx: float, fy: float, cx: float, cy: float) -> np.ndarray:
    """Convert normalized 2D image line to 3D plane through camera center."""
    A = a * fx
    B = b * fy
    C = a * cx + b * cy + c
    n = np.array([A, B, C], dtype=float)
    norm = np.linalg.norm(n)
    if norm < 1e-9:  % Corrected from norm == 0
        raise ValueError('Degenerate line -> zero normal')
    n /= norm
    return np.array([n[0], n[1], n[2], 0.0], dtype=float)
```

1.4 Plücker Coordinates for 3D Line Representation

A 3D line can be defined as the intersection of two non-parallel planes. Plücker coordinates provide a convenient way to represent this line with two 3D vectors.

1.4.1 Mathematical Formulation

Consider two planes $P_1 : \mathbf{n}_1 \cdot X + d_1 = 0$ and $P_2 : \mathbf{n}_2 \cdot X + d_2 = 0$. The line of intersection has:

1. **Direction Vector (L)**: The line's direction must be orthogonal to both plane normals. It is therefore given by their cross product:

$$\mathbf{L} = \mathbf{n}_1 \times \mathbf{n}_2 \tag{11}$$

2. **Moment Vector (M)**: The moment vector is related to the line's distance from the origin. It is defined as $\mathbf{M} = \mathbf{p} \times \mathbf{L}$, where $\mathbf{p}$ is any point on the line. A more direct calculation from the plane parameters is:
$$\mathbf{M} = d_1\mathbf{n}_2 - d_2\mathbf{n}_1 \tag{12}$$

From the Plücker coordinates $(\mathbf{L}, \mathbf{M})$, the point on the line closest to the origin, $\mathbf{p}_0$, can be recovered by:
$$\mathbf{p}_0 = \frac{\mathbf{L} \times \mathbf{M}}{\|\mathbf{L}\|^2} \tag{13}$$

1.4.2 Code Implementation

```
def plucker_from_two_planes(P: np.ndarray, Q: np.ndarray) -> Tuple[np.ndarray, np.ndarra
    """Return (direction, moment) of line = intersection of planes P and Q."""
    n1, d1 = P[:3], P[3]
    n2, d2 = Q[:3], Q[3]
    d = np.cross(n1, n2)
    if np.linalg.norm(d) < 1e-9: % Corrected from == 0
        raise ValueError('Parallel planes -> no line')
    m = d1 * n2 - d2 * n1
    return d, m

def point_from_plucker(direction: np.ndarray, moment: np.ndarray) -> np.ndarray:
    """Closest point on line to origin: (d x m) / ||d||^2."""
    denom = float(direction @ direction)
    return np.cross(direction, moment) / denom
```

1.5 Ray-Plane Intersection

This is a fundamental geometric operation used to find the 3D location of a pixel projected onto a known plane.

1.5.1 Mathematical Formulation

A 3D ray originating from the camera center (origin) and passing through a pixel (u, v) can be parameterized as $P(s) = s \cdot \mathbf{v}_{\text{ray}}$, where $s > 0$ is a scalar distance and $\mathbf{v}_{\text{ray}}$ is a unit direction vector. This direction is calculated from the normalized image coordinates:

$$\mathbf{v}_{\text{ray}} = \text{normalize}\left(\begin{pmatrix} (u - c_x)/f_x \\ (v - c_y)/f_y \\ 1 \end{pmatrix}\right) \tag{14}$$

To find the intersection with a plane $\mathbf{n} \cdot X + D = 0$, we substitute the ray equation into the plane equation:
$$\mathbf{n} \cdot (s \cdot \mathbf{v}_{\text{ray}}) + D = 0 \tag{15}$$

Solving for s:
$$s = \frac{-D}{\mathbf{n} \cdot \mathbf{v}_{\text{ray}}} \tag{16}$$

The intersection point is then $P_{\text{intersect}} = s \cdot \mathbf{v}_{\text{ray}}$. An intersection exists only if the denominator is non-zero (ray and plane are not parallel) and if $s > 0$ (intersection is in front of the camera).

1.5.2 Code Implementation

```
def intersect_ray_plane(ray: np.ndarray, n: np.ndarray, D: float) -> Optional[np.ndarray
    """Intersect ray (origin=0, direction=ray) with plane n X + D = 0."""
    denom = float(n @ ray)
    if abs(denom) < 1e-9:
        return None
    s = -D / denom
    if s <= 0:
        return None
    return s * ray
```

2 Method 1: Calibration via Plücker Lines

This method treats the laser plane calibration as a problem of finding a plane that contains a set of 3D lines.

2.1 Principle

For each image, the observed laser stripe is a 3D line formed by the intersection of two planes:

1. Π_B: The physical plane of the checkerboard, whose equation is found from its pose.
2. Π_I: The image plane containing the camera center and the 2D laser stripe in the image.

The intersection of these two planes gives a 3D line, L_i, for each image i. All these lines L_i must, by definition, lie on the true laser plane, Π_L. If the laser plane has normal $\mathbf{n}_L$, then the direction vector $\mathbf{d}_i$ of every line L_i must be orthogonal to $\mathbf{n}_L$.

$$\mathbf{n}_L \cdot \mathbf{d}_i = 0 \quad \forall i \tag{17}$$

2.2 Algorithm

2.2.1 Step 1: Per-Image Line Extraction

For each input image, the following procedure is executed:

```
# ... find checkerboard corners and compute pose (R, tvec) ...
P_board = board_plane_from_pose(R, tvec)

# ... enhance image and fit 2D line (a,b,c) to the laser stripe ...
P_laser_img = plane_from_image_line(a,b,c, fx, fy, cx, cy)

# Intersect planes -> line (d=direction, m=moment)
d, m = plucker_from_two_planes(P_board, P_laser_img)
directions.append(d/np.linalg.norm(d))
moments.append(m)
```

This loop populates a list of direction vectors, 'directions', one for each valid view.

2.2.2 Step 2: Global Plane Normal Estimation

We now have a collection of N direction vectors $\{\mathbf{d}_1, \ldots, \mathbf{d}_N\}$. We seek a unit vector $\mathbf{n}_L$ that is orthogonal to all of them. This can be formulated as a system of linear equations:

$$\begin{pmatrix} --- & \mathbf{d}_1^T & --- \\ --- & \mathbf{d}_2^T & --- \\ & \vdots & \\ --- & \mathbf{d}_N^T & --- \end{pmatrix} \mathbf{n}_L = \mathbf{0} \tag{18}$$

This is an overdetermined system of the form $A\mathbf{x} = \mathbf{0}$. The optimal solution in a least-squares sense is the one that minimizes $\|A\mathbf{x}\|^2$ subject to $\|\mathbf{x}\| = 1$. This solution is the eigenvector corresponding to the smallest eigenvalue of the matrix A^TA. A numerically robust way to find this is to compute the Singular Value Decomposition (SVD) of A. If $A = U\Sigma V^T$, the solution is the last column of V, which corresponds to the smallest singular value.

```
Dmat = np.vstack(directions)
# Plane normal is singular vector with smallest singular value
_, _, Vt = np.linalg.svd(Dmat)
plane_normal = Vt[-1]
```

2.2.3 Step 3: Plane Offset (D) Estimation

With the normal $\mathbf{n}_L$ determined, we must find the offset D_L. Every line L_i must lie on the plane. We can extract a representative point $\mathbf{p}_i$ from each line (e.g., the point closest to the origin). For each such point, we have an estimate for the offset:

$$\mathbf{n}_L \cdot \mathbf{p}_i + D_L \approx 0 \implies D_{L,i} \approx -(\mathbf{n}_L \cdot \mathbf{p}_i) \tag{19}$$

To get a single robust value, we can aggregate all estimates $D_{L,i}$. Using the median is robust to outlier points that may arise from noisy line fits.

```
pts = []
for d, m in zip(directions, moments):
    p0 = point_from_plucker(d, m)
    pts.append(p0)
pts = np.vstack(pts)

proj = pts @ plane_normal
D_pre = -np.median(proj)
```

2.2.4 Step 4: Reference Frame Alignment

The computed offset D depends on the absolute position of the checkerboard during capture. To produce a consistent result, the plane's offset is re-referenced to pass through a canonical point, such as the mean of all observed checkerboard origins. The new offset D_{aligned} is calculated to satisfy $\mathbf{n}_L \cdot \mathbf{p}_{\text{ref}} + D_{\text{aligned}} = 0$.

```
ref_origin = compute_reference_origin(board_origins, 'mean')
D_offset = align_plane_offset(plane_normal, D_pre, ref_origin)
```

3 Method 2: Calibration via Weighted RANSAC

This method approaches the problem by generating a 3D point cloud of the laser plane and then robustly fitting a plane to these points.

3.1 Principle

For each image, every pixel belonging to the laser stripe can be back-projected into 3D space. This is achieved by casting a ray from the camera center through the pixel and finding its intersection with the known checkerboard plane for that view. By aggregating these 3D points from all images, we build a dense point cloud that samples the laser plane. A plane is then fitted to this cloud. To handle noise and outliers from imperfect laser detection, a robust fitting algorithm like RANSAC is employed.

3.2 Algorithm

3.2.1 Step 1: 3D Point Cloud Generation

For each image and for each pixel (u, v) identified as part of the laser stripe:

1. A 3D ray $\mathbf{v}_{\text{ray}}$ from the camera origin through (u, v) is computed.
2. The checkerboard plane $(\mathbf{n}_B, D_B)$ is recovered from the board's pose.
3. The intersection point P of the ray and the plane is calculated using 'intersect_ray_plane'.
4. This point P is added to a global list.
5. A weight w for this point is stored, typically proportional to the brightness of the pixel in an enhanced image (e.g., blue-channel difference). Brighter pixels are considered more reliable.

```
# In a loop over images and laser pixels (x,y):
ray = np.array([(x - cx)/fx, (y - cy)/fy, 1.0])
P = intersect_ray_plane(ray, n_board, d_board)
if P is not None:
    w = float(intensity[y, x])
    all_points.append(P)
    all_weights.append(w)
```

3.2.2 Step 2: Weighted RANSAC for Robust Plane Fitting

The RANSAC (RANdom SAmple Consensus) algorithm is an iterative method for estimating parameters of a model from a dataset containing outliers. The weighted variant prioritizes points considered more reliable.
The iterative process is as follows:

1. **Hypothesize**: Randomly sample 3 points from the cloud to define a candidate plane $(\mathbf{n}_k, D_k)$. The sampling is not uniform; the probability of picking a point is proportional to its weight w_i. This is efficiently implemented by sampling from the cumulative distribution of weights.

2. **Evaluate**: For the candidate plane, calculate the perpendicular distance of all other points to it: $\text{dist}_i = |\mathbf{n}_k \cdot \mathbf{p}_i + D_k|$. An "inlier" is a point where dist_i is below a predefined threshold. The score for the candidate plane is the sum of the weights of all its inliers: $\text{Score}_k = \sum_{i \in \text{inliers}} w_i$.

3. **Select**: After many iterations, the plane with the highest score is chosen as the best model. The set of inliers corresponding to this plane is also returned.

3.2.3 Step 3: Final Plane Refinement

After RANSAC has identified the set of inlier points, a more accurate plane is fitted to this cleaned subset using a weighted least-squares method. This is best solved via Principal Component Analysis (PCA).

1. Compute the weighted centroid $\mathbf{c}$ of the inlier points:

$$\mathbf{c} = \frac{\sum_{i \in \text{inliers}} w_i \mathbf{p}_i}{\sum_{i \in \text{inliers}} w_i} \tag{20}$$

2. Form the weighted, centered covariance matrix of the inlier points:

$$C = \sum_{i \in \text{inliers}} w_i (\mathbf{p}_i - \mathbf{c})(\mathbf{p}_i - \mathbf{c})^T \tag{21}$$

3. The eigenvector of C corresponding to the smallest eigenvalue is the normal vector $\mathbf{n}_L$ of the best-fit plane. This is equivalent to finding the last right singular vector from the SVD of the weighted, centered data matrix.

4. The plane offset is then calculated to ensure the plane passes through the centroid: $D_L = -\mathbf{n}_L \cdot \mathbf{c}$.

```
# In weighted_ransac function after finding best_inliers:
P = points[best_inliers]
w = weights[best_inliers]
w_norm = w / w.sum()
cent = (P * w_norm[:,None]).sum(axis=0)
Q = (P - cent) * np.sqrt(w_norm[:,None])
_,_,Vt = np.linalg.svd(Q, full_matrices=False)
n_refined = Vt[-1]
d_refined = -n_refined @ cent
```

The alignment step (Step 4 from the Plücker method) is then applied identically to ensure a consistent reference frame.