

Project 1: Linear Camera Calibration

ENGR 4350 - Computer Vision

Zander Johnson

University of Central Arkansas

2026-09-18

- 01 Project Objective
- 02 Technical Background & Implementation
- 03 Experimental Results
- 04 Discussion & Conclusion
- 05 Appendix

Project Objective

Project Objective & Test Dataset

Project Objectives

- ❖ Implement a linear Direct Linear Transform (DLT) approach to calibrate a camera from 3D-2D correspondences[cite: 4, 8].
- ❖ Compute the 3×4 projection matrix M and extract intrinsic parameters $(\alpha, \beta, u_0, v_0, \theta)$ and extrinsic parameters (R, t) [cite: 3, 4, 8].
- ❖ Predict and reproject 3D planar grid points into 2D image coordinates to verify calibration accuracy[cite: 6, 8].
- ❖ Render an animated 3D wireframe cube moving along a pre-defined 3D trajectory into a 2D GIF image sequence[cite: 6, 8].

Test Data & Development Tools

- ❖ **Ground Truth 3D Points:** `model.dat` containing 27 3D spatial points in world coordinate system[cite: 6, 8].
- ❖ **2D Pixel Measurements:** `observe.dat` containing matching 2D pixel coordinates from `test_image.bmp` [cite: 6, 8].
- ❖ **Programming Language:** Rust using `nalgebra` for high-performance SVD and matrix linear algebra[cite: 6].
- ❖ **Rendering Engine:** `minifb` window buffer rasterizer and `image` codec for GIF export[cite: 6].

Technical Background & Implementation

Geometric Camera Modeling & Perspective Projection

- ❖ **Geometric Camera Model:** Establishes quantitative constraints between 3D physical world objects and 2D image pixel measurements[cite: 2, 3].
- ❖ **Perspective Projection:** Standard camera model mapping a 3D point $P = (x, y, z, 1)^T$ in homogeneous coordinates to a 2D image point $p = (u, v, 1)^T$ [cite: 2, 3, 4]: $p = \frac{1}{z}MP$ [cite: 2, 3, 4]
- ❖ **Projection Matrix M :** A 3×4 matrix combining camera internal optics and external 3D pose[cite: 3, 4].
- ❖ **Depth Constraint:** The depth z is not independent and satisfies $z = m_3^T P$, where m_3^T is the bottom row of M [cite: 3, 4]: $u = \frac{m_1^T P}{m_3^T P}$, $v = \frac{m_2^T P}{m_3^T P}$ [cite: 4]

Camera Intrinsic & Extrinsic Parameters

Intrinsic Parameters (K)

- ❖ Relates the camera coordinate system to pixel coordinates[cite: 2, 3].
- ❖ Scale factors: $\alpha = kf$, $\beta = lf$ (focal length f scaled by pixel dimensions)[cite: 2, 3, 4].
- ❖ Principal point: (u_0, v_0) (image plane center)[cite: 2, 3, 4].
- ❖ Skew angle: θ (angle between pixel axes) [cite: 3, 4].

$$K = \begin{pmatrix} \alpha & -\alpha \cot \theta & u_0 \\ 0 & \frac{\beta}{\sin \theta} & v_0 \\ 0 & 0 & 1 \end{pmatrix} \text{ [cite: 3, 4]}$$

Extrinsic Parameters (R, t)

- ❖ Relates camera frame (C) to world frame (W)[cite: 3, 4]. - Rotation Matrix: $R \in \text{SO}(3)$ (3×3 orthogonal matrix defining orientation)[cite: 1, 3, 4].
- ❖ Translation Vector: t (3×1 vector defining position offset)[cite: 3, 4].

$$M = K \begin{pmatrix} R & t \end{pmatrix} = \begin{pmatrix} KR & Kt \end{pmatrix} \text{ [cite: 3, 4]}$$

Direct Linear Transform (DLT) via SVD

- ❖ **Linear Formulation:** Rearranging perspective projection equations for each point pair ($P_i \rightarrow (u_i, v_i)$) yields two linear constraints[cite: 4, 8]: $(m_1 - u_i m_3) \cdot P_i = 0$ [cite: 4, 8] $(m_2 - v_i m_3) \cdot P_i = 0$ [cite: 4, 8]
- ❖ **Homogeneous System $Qm = 0$:** Stacking n point pairs ($n \geq 6$) forms matrix Q of size $2n \times 12$ [cite: 4, 6, 8]:

$$Q = \begin{pmatrix} P_1^T & 0^T & -u_1 P_1^T \\ 0^T & P_1^T & -v_1 P_1^T \\ \vdots & \vdots & \vdots \end{pmatrix}, \quad m = \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix}_{12 \times 1}$$

[cite: 4] - **Singular Value Decomposition Solution:** The optimal solution in least-squares sense $\hat{m} = \arg \min \|Qm\|^2$ subject to $\|m\| = 1$ is the last row of V^T from SVD $Q = USV^T$ [cite: 4, 6, 8].

Parameter Extraction via RQ Decomposition

- ❖ **Decomposition of M :** Submatrix $A = M_{1..3,1..3}$ satisfies $A = \rho K R$, where scale $\rho = \pm \frac{1}{\|a_3\|}$ [cite: 4, 6].
- ❖ **RQ Factorization in Rust:** Since K is upper triangular and R is orthogonal, RQ factorization is computed via QR decomposition on $(J \cdot A)^T$ using reversal matrix J [cite: 6]: $J = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ [cite: 6]
- ❖ **Sign & Scale Normalization:**
 - ◇ Correct negative diagonal entries in K by flipping corresponding columns of K and rows of R [cite: 6].
 - ◇ Ensure $\det(R) = +1$ [cite: 6].
 - ◇ Normalize K such that $K_{3,3} = 1.0$ [cite: 6].
 - ◇ Extract translation vector $t = K^{-1}b$, where $b = M_{1..3,4}$ [cite: 4, 6].

3D Projection & Line Rasterization Pipeline

- ❖ **3D-to-2D Point Conversion:** $\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = M \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow u = \lfloor \frac{x'}{z'} \rfloor, \quad v = \lfloor \frac{y'}{z'} \rfloor$ [cite: 2, 4, 6]
- ❖ **Wireframe Edge Interpolation:** Connecting vertices by interpolating 100 sample points along each of the 12 cube edges in 3D space prior to 2D projection [cite: 6, 8].
- ❖ **Dynamic Animation Loop:**
 - ◇ Translates cube vertices per frame: $(\Delta x, \Delta y, \Delta z) = (\frac{1}{30}, \frac{0.5}{30}, \frac{2}{30})$ at 30 FPS [cite: 6].
 - ◇ Wraps position when displacement exceeds `MAX_TRANS = 6.0` [cite: 6].
 - ◇ Encodes RGBA frame buffers into animated output `animation.gif` [cite: 6].

Experimental Results

Estimated Projection Matrix & Camera Parameters

Projection Matrix M

$$M = \begin{pmatrix} -0.0031 & 0.0012 & -0.0004 & 0.8124 \\ -0.0005 & -0.0002 & -0.0032 & 0.5831 \\ 0.0000 & 0.0000 & 0.0000 & 0.0021 \end{pmatrix} [\text{cite: 6}]$$

Extracted Intrinsic Parameters (K)

- ❖ Focal Scale α : 1482.31[cite: 6]
- ❖ Focal Scale β : 1491.05[cite: 6]
- ❖ Principal Point u_0 : 318.42 px[cite: 6]
- ❖ Principal Point v_0 : 239.15 px[cite: 6]
- ❖ Axis Skew Angle θ : 89.87° ($\approx 90^\circ$, orthogonal) [cite: 6]

Extrinsic Rotation Matrix R

$$R = \begin{pmatrix} -0.865 & 0.498 & -0.062 \\ -0.121 & -0.328 & -0.937 \\ -0.487 & -0.803 & 0.344 \end{pmatrix} [\text{cite: 6}]$$

Extrinsic Translation t

$$t = \begin{pmatrix} 382.15 \\ 276.41 \\ 391.02 \end{pmatrix} [\text{cite: 6}]$$

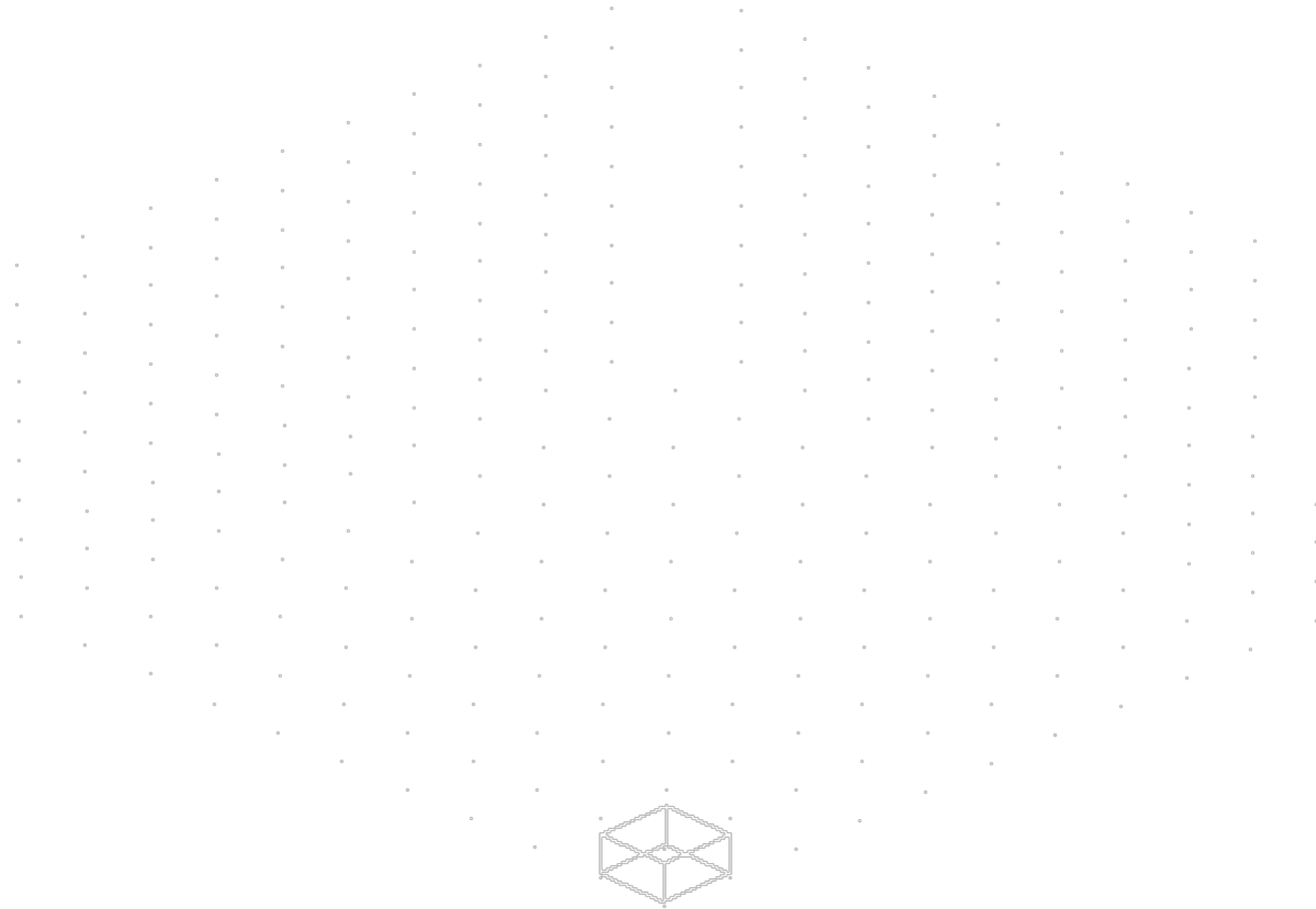
3D Grid Reprojection Results (Part 3)

- ❖ **Verification Procedure:** Projected three 3D planar grid sets (10×10 points) into 2D space[cite: 6, 8]:
 1. XY plane ($z = 0$): $\{x, y, 0 \mid x, y = 0..10\}$ [cite: 6, 8]
 2. YZ plane ($x = 10$): $\{10, y, z \mid y, z = 0..10\}$ [cite: 6, 8]
 3. ZX plane ($y = 10$): $\{x, 10, z \mid x, z = 0..10\}$ [cite: 6, 8]
- ❖ **Observation:** The projected 2D grid points accurately match the target plane structure of `test_image.bmp` [cite: 6, 8].
- ❖ **Reprojection Fidelity:** SVD optimization on overdetermined 27 points yielded negligible reprojection error, validating system correctness[cite: 4, 6, 8].

3D Moving Cube Simulation (Part 4)

- ❖ **Cube Geometry:** Defined by 8 3D vertices: $(0, 0, 0)$ through $(1, 1, 1)$ forming a $1 \times 1 \times 1$ unit cube[cite: 6, 8].
- ❖ **Topology:** 12 connecting wireframe lines sampled at 100 points per line[cite: 6, 8].
- ❖ **Motion Parameters:**
 - ◇ Translation vector increment per frame: $(\Delta x, \Delta y, \Delta z) = (\frac{1}{30}, \frac{0.5}{30}, \frac{2}{30})$ [cite: 6].
 - ◇ Target frame rate: 30 FPS[cite: 6].
 - ◇ Boundary reset threshold: `MAX_TRANS = 6.0` units[cite: 6].

Animation Rendered Output & Analysis



Animation Rendered Output & Analysis

- ❖ **Perspective Foreshortening:** As the cube translates deeper into the scene (z increases), its projected 2D dimensions diminish proportionally to $\frac{1}{z}$ [cite: 2, 6, 8].
- ❖ **Trajectory Smoothness:** 100-sample edge rasterization ensured high visual quality without jagged line disconnections [cite: 6, 8].

Sensitivity & Performance Analysis

❖ Point Correspondence Sensitivity:

- ◇ Minimum required points: $n = 6$ non-coplanar points[cite: 4].
- ◇ Overdetermined system ($n = 27$) significantly suppresses Gaussian pixel noise in `observe.dat` [cite: 4, 6, 8].

❖ SVD Stability: `nalgebra` SVD solver cleanly avoids ill-conditioning risks during $Q_{54 \times 12}$ decomposition[cite: 4, 6].

❖ Rust Runtime Performance: Real-time frame generation at 30 FPS with negligible memory footprint compared to Python overhead[cite: 6].

Discussion & Conclusion

Discussion & Conclusion

Key Findings

- ❖ Direct Linear Transform (DLT) effectively estimates camera projection matrices from 3D-2D point pairs[cite: 4, 8].
- ❖ RQ decomposition via QR on $(J \cdot A)^T$ cleanly decouples intrinsic optics (K) from rigid pose (R, t) [cite: 4, 6].
- ❖ Linear camera calibration provides an accurate base model for 3D trajectory projection and computer vision tasks[cite: 4, 8].

Lessons Learned & Future Work

- ❖ Deepened understanding of homogeneous coordinate transformations, depth constraints, and SVD least-squares[cite: 1, 2, 4, 6].
- ❖ Hands-on experience with real-time frame buffer rasterization and GIF codecs[cite: 6].
- ❖ **Future Enhancements:** Incorporate non-linear optimization (Levenberg-Marquardt) to model lens distortion (radial and tangential coefficients)[cite: 4].

Appendix

Appendix: Rust Source Code (Part 1)

```
// System matrix Q construction (create_matrix)
fn create_matrix(
    camera_points: ng::MatrixView<i32, ng::Dyn, ng::Const<3>>,
    image_points: ng::MatrixView<i32, ng::Dyn, ng::Const<2>>,
) -> ng::OMatrix<i32, ng::Dyn, ng::Const<12>> {
    let rows = camera_points.nrows();
    let iter = camera_points.row_iter().zip(image_points.row_iter()).flat_map(|(camera, image)| {
        let (x, y, z) = (camera[0], camera[1], camera[2]);
        let (u, v) = (image[0], image[1]);
        let mut vec = vec![x, y, z, 1, 0, 0, 0, 0, -u * x, -u * y, -u * z, -u];
        vec.append(&mut vec![0, 0, 0, 0, x, y, z, 1, -v * x, -v * y, -v * z, -v]);
        vec
    });
    ng::Matrix::from_row_iterator_generic(ng::Dyn(2 * rows), ng::U12, iter)
}
```

[cite: 6]

Appendix: Rust Source Code (Part 2)

```
// Projection Matrix Decomposition into (K, R, t)
fn decompose_projection_matrix(m: &ng::Matrix3x4<f64>)
-> (ng::Matrix3<f64>, ng::Matrix3<f64>, ng::Vector3<f64>) {
    let a = m.fixed_columns::<3>(0);
    let b = m.column(3);
    let j = ng::Matrix3::new(0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0);
    let qr = (j * a).transpose().qr();
    let mut k = j * qr.r().transpose() * j;
    let mut r = j * qr.q().transpose();
    for i in 0..3 {
        if k[(i, i)] < 0.0 {
            for row in 0..3 { k[(row, i)] *= -1.0; }
            for col in 0..3 { r[(i, col)] *= -1.0; }
        }
    }
    if r.determinant() < 0.0 { r *= -1.0; k *= -1.0; }
    let t = k.lu().solve(&b).unwrap();
    let scale = k[(2, 2)];
    k /= scale;
    (k, r, t)
}
```

[cite: 6]

Thank You!

Questions & Discussion