Linear Algebra: Final Presentation

Judah Levin

June 2025

ICP Initialization

We want to recover a rigid transformation (rotation) $O \in O(d)$, and an unknown permutation matrix $S \in \text{Sym}(n)$, such that:

$$Q = OPS \tag{1}$$

Where d is the dimension (3 for coordinates) and $P, Q \in \mathbb{R}^{d \times n}$ are centered point clouds (their columns are points in $\mathbb{R}^d$ and have a mean of zero).

This is a nontrivial problem because:
1. There is no correspondence between points due to random shuffling.
2. We want a global alignment among potential noise, large n, occlusions, or nearly equal eigenvalues of their ellipsoids of inertia.

In *An Approach to Robust ICP Initialization*, Professor Kolpakov and Michael Werman propose an ICP initialization algorithm called E-init, in which they compute the ellipsoids of inertia defined by:

$$E_P = PP^T \tag{2}$$

$$E_Q = QQ^T \tag{3}$$

(the covariances matrices in $\mathbb{R}^{3\times3}$)
Next, they diagonalize the ellipsoids with np.linalg.eigh(), a 'divide and conquer strategy' of splitting a tridiagonal matrix into subproblems:

$$E_P = U_P \Lambda U_P^T \tag{4}$$

$$E_Q = U_Q \Lambda U_Q^T \tag{5}$$

where U_P, U_Q are orthogonal matrices with columns of eigenvectors and Λ is the diagonal matrix of real eigenvalues.
Then, they define:

$$U_0 = U_Q U_P^T \tag{6}$$

which is the matrix that transforms the ellipsoid E_P into E_Q. Let's derive that definition. Given the relation:

$$Q = OPS \implies E_Q = QQ^T = (OPS)(OPS)^T =$$

$$= OPSS^T P^T O^T = OPP^T O^T = OE_P O^T$$

Where $SS^T = I$ since S is a permutation matrix. We also have,

$$U_P^T E_P U_P = U_P^T (U_P \Lambda U_P^T) U_P = I \Lambda I = \Lambda$$

since $U_P \in O(d)$, the matrix of eigenvectors of E_P. Similarly,

$$U_Q^T E_Q U_Q = U_Q^T (U_Q \Lambda U_Q^T) U_Q = I \Lambda I = \Lambda$$

Since $E_Q = OE_P O^T$, this implies by substitution:

$$U_Q^T (OE_P O^T) U_Q = \Lambda$$

Multiply both sides on the right by U_Q^T and on the left by U_Q, getting:

$$U_Q (U_Q^T O E_P O^T U_Q) U_Q^T = I O E_P O^T I = O E_P O^T = U_Q \Lambda U_Q^T$$

Since $U_Q \in O(d)$, the matrix of eigenvectors of E_Q.
Recall:

$$E_P = U_P \Lambda U_P^T \implies O E_P O^T = O(U_P \Lambda U_P^T) O^T$$

Thus, since $OE_P O^T = U_Q \Lambda U_Q^T$, the equation above implies:

$$E_Q = O U_P \Lambda U_P^T O^T = U_Q \Lambda U_Q^T$$

It also follows that:

$$(OU_P)\Lambda(OU_P)^T = U_Q \Lambda U_Q^T \implies OU_P = U_Q$$

Now, while the directions of U_P and U_Q are the same, that is not necessarily true for signs of the eigenvectors. Thus, we use a diagonal matrix $D = \text{diag}(\pm 1, \pm 1, \pm 1) \in \text{Ref}(d)$. So:

$$U_Q = O U_P D \implies O = U_Q D U_P^T$$

since $D = D^T = D^{-1}$ (since D is diagonal and orthogonal) and $U_P^{-1} = U_P^T$ (since U_P is orthogonal). Now, we have derived the rotation O which maps $P \to Q$, but to map $Q \to P$, we must use O^{-1}. But, $O^{-1} = O^T$ since O is an orthogonal matrix. So, our matrix which rotates Q to get P is:

$$O^T = (U_Q D U_P^T)^T$$

But, we want to avoid diagonalizing U_Q. So, let:

$$U_0 = U_Q U_P^T \tag{7}$$

so that

$$U_0 U_P = (U_Q U_P^T) U_P = U_Q$$

By substitution:

$$O_D^T = (U_0 U_P D U_P^T)^T \tag{8}$$

To find the best rotation matrix O_D^T, we must find the optimal D (forget Π^* for now...):

$$D^* = \arg \min_{D \in \mathrm{Ref}(d)} \|P\Pi^* - O_D^\top Q\|_F \tag{9}$$

using the Frobenius norm, or the total pointwise error. How? As the equation states, we just loop through 2^d possible reflections and use the reflection with the least error.

Alignment Problem

While we embarked on a rigorous proof to recover $O \in O(d)$, we left out a fundamental issue. How do we compute the error if we don't know the point correspondence? Otherwise, it would be like trying to match two bunnies that have been run over by a truck! So, within the loop through reflections, we need to estimate the permutation matrix $S \in \mathrm{Sym}(n)$, i.e., estimate correspondences to properly compute error.

In the E-init paper, Professor Kolpakov uses a nearest neighbor matching algorithm called k-d trees. For some point cloud $X \in \mathbb{R}^{d \times n}$ the tree begins forming by partitioning the space along some dimension $k = 1, 2, \ldots, d$. For some query point $x_1 = x_{11}, x_{12}, \ldots, x_{1d}$ choose a hyperplane orthogonal to dimension k through some median coordinate $x_2 = x_{21}, x_{22}, \ldots, x_{2d}$, where each side of the hyperplane is a subtree. Then, store the distance between x_1 and x_2. Repeat recursively for $k = k + 1$, storing the best distance r. When you hit $k = d$, at the leaf node, check if the distance to the hyperplane in dimension d is less than r. If not, you're done. If so, you must check the distance coordinate on the other side of that hyperplane. Then, if that distance is less, update r, and check the distance to the next hyperplane in $d - 1$. Continue backtracking this way until the distance to some $d - k > r$, then you're done (because there is no place left to go). The time complexity is $O(n \log n)$.

However, nearest neighbors assumes that E-init works well enough to transform corresponding points close together. This is not always the case for a couple reasons. First, in a uniform point cloud, each point is approximately equidistant to its neighbors, so k-d trees may not draw meaningful hyperplanes. Second, if the eigenvalues are too similar along each principal axis of an ellipsoid, approaching a sphere, then trying to compute the optimal rotation will be unstable (this can happen even if the point clouds are not spheres themselves).

Sinkhorn-Hungarian Solution

$S \in \mathrm{Sym}(n)$ has a single 1 in each row and column with zeroes elsewhere, and in our case it is used to switch the columns of P (multiplying by S on the right

side). But, to estimate S, we can find a doubly stochastic matrix:

A *doubly stochastic matrix* (also called a *soft permutation matrix*) is a square matrix

$$A = (a_{ij}) \in \mathbb{R}^{n \times n}$$

whose entries are nonnegative real numbers, such that each row and each column sums to 1. That is,

$$\sum_{j=1}^{n} a_{ij} = 1 \quad \text{for all } i = 1, \ldots, n, \quad \text{and} \quad \sum_{i=1}^{n} a_{ij} = 1 \quad \text{for all } j = 1, \ldots, n.$$

Thus, a doubly stochastic matrix is both left stochastic and right stochastic, and individual rows and columns can be represented as probability distributions. Each entry a_{ij} represents the probability of a column i in P (a point in P) becoming a column j in Q.

Birkhoff-von-Neumann decomposition. If A is a soft permutation matrix, then:

$$\exists \theta_1, \ldots, \theta_n \geq 0, \ \sum_{i=1}^{n} \theta_i = 1$$

and permutation matrices $P_1, \ldots, P_n$ such that $A = \theta_1 P_1 + \cdots + \theta_n P_n$. It follows that A is a convex combination of $P_1, \ldots, P_n$.

Let B_n be the set of all soft permutation matrices $A \in \mathbb{R}^{n \times n}$. It follows that B_n is the smallest convex set with vertices $P_1, \ldots, P_n$, making B_n a convex hull, that is, $B_n = \text{conv}(\text{Perm}(n))$. B_n is also called the Birkhoff polytope.

Thus, if we can figure out A (a soft permutation matrix), then we are trying to find the closest vertex to it in B_n, the optimal estimate for S (a permutation matrix with 1's and 0's). This is a linear assignment problem, which can be solved by the Hungarian algorithm.

To find A^*, we must solve an optimal transport problem (entropy-regularized) with the cost matrix $C \in \mathbb{R}^{n \times n}$:

$$\max_{A \in B_n} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} \, C_{ij} + \varepsilon \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} (\ln A_{ij} - 1) \tag{10}$$

where $A \in \mathbb{R}^{n \times n}$ is the soft permutation matrix, $B_n \subset \mathbb{R}^{n \times n}$ is the Birkhoff polytope of doubly stochastic matrices, $C_{ij} = \|p_i - q_j O_D^T\|_2^2$ is the cost, or the squared ℓ_2-norm) between source point p_i and transformed target point q_j. The second term is derived from Shannon entropy, which is large with uniform or near-zero entries and low with mass concentrated in a few entries. This regularization amplifies uncertainty or 'softness,' while keeping the problem convex, where $\varepsilon > 0$ is the entropy regularization parameter. So, to minimize

$$f(A_{ij}) = A_{ij} C_{ij} + \epsilon A_{ij} \ln A_{ij} - \epsilon A_{ij}$$

we take the partial derivative w.r.t to A_{ij} and set to zero. We can do this because the function is differentiable, its feasible region is a convex polytope and the negative entropy term is also convex.

$$\frac{d}{dA_{ij}}(A_{ij}C_{ij} + \epsilon A_{ij}\ln A_{ij} - \epsilon A_{ij}) = C_{ij} + \epsilon + \epsilon \ln A_{ij} - \epsilon = C_{ij} + \epsilon \ln A_{ij} = 0$$

This implies $C_{ij} = -\epsilon \ln A_{ij} \implies \frac{C_{ij}}{-\epsilon} = \ln A_{ij}$. Exponentiate both sides:

$$A_{ij} = e^{\frac{C_{ij}}{-\epsilon}}$$

However, to satisfy the double stochasticity of A_{ij}, according to *Sinkhorn's theorem*, we can represent:

$$A = \text{diag}(u)K\text{diag}(v) \tag{11}$$

and

$$A_{ij} = u_i K_{ij} v_j \tag{12}$$

such that $K \in \mathbb{R}^{n\times n}$ with strictly positive elements, and diag(u) and diag(v) have strictly positive diagonal elements.

Since we know $C_{ij} = \|p_i - q_j O_D^T\|_2^2$, C stores all non-negative pairwise costs. Now, let

$$K_{ij} = e^{\frac{C_{ij}}{-\epsilon}} \tag{13}$$

where $K \in \mathbb{R}^{n\times n}$ with strictly positive elements.

It follows, by *Sinkhorn's algorithm*, that:

$$\sum_{i=1}^{n} A_{ij} = 1 \implies v_j \sum_{i=1}^{n} u_i K_{ij} = 1 \implies v_j = \frac{1}{(K^T u)_j}$$

and

$$\sum_{j=1}^{n} A_{ij} = 1 \implies u_i \sum_{j=1}^{n} K_{ij} v_j = 1 \implies u_i = \frac{1}{(Kv)_i}$$

Step 0: Initialization. Set

$$u^{(0)} = v^{(0)} = \mathbf{1} \in \mathbb{R}^n.$$

Step 1: Row Normalization. Given $v^{(k)}$, update

$$u^{(k+1)} = \frac{1}{Kv^{(k)}} \quad \text{(elementwise division)}$$

Step 2: Column Normalization. Given $u^{(k+1)}$, update

$$v^{(k+1)} = \frac{1}{K^\top u^{(k+1)}} \quad \text{(elementwise division)}$$

Step 3: Repeat Until Convergence. Repeat Steps 1–2 until the matrix $A^{(k)} = \text{diag}(u^{(k)})K\text{diag}(v^{(k)})$ satisfies:

$$\|A^{(k)}\mathbf{1} - \mathbf{1}\| < \text{tol} \quad \text{and} \quad \|(A^{(k)})^\top \mathbf{1} - \mathbf{1}\| < \text{tol}.$$

Finally, we have derived the soft permutation matrix with A using *Sinkhorn*. Now, if you recall, we still need to solve the linear assignment problem, which basically means taking the maximum likelihood that a given column i in P will be permuted to some column j in Q. Thus, we are trying to solve the linear program:

$$\Pi^* = \arg \min_{\Pi \in \text{Perm}(n)} \langle A, \Pi \rangle_F \tag{14}$$

which means we are trying to maximize overlap of Π with our soft permutation matrix A (the frobenius norm is like a matrix dot product).

The Hungarian algorithm works as follows:
1. Subtract the row minimum from each row
2. Subtract the column minimum from each column (a kind of normalization)
3. Find the smallest nonzero entry δ
4. Subtract δ from nonzero elements
5. Add 2δ to zero elements
6. For each row, assign the row as a column in the same position of its zero entry (e.g. entry 2 $\rightarrow$ column 2), starting with the rows with the fewest zeros
7. Assign subsequent rows as columns which haven't been taken yet (like Sudoku)
8. Once Π^* is computed, we apply the permutation matrix to P, getting $P\Pi^*$. This is the arrangement of P which best matches the points in $O_D^\top Q$. See (9).

The Hungarian algorithm is guaranteed to return a vertex of the Birkhoff polytope B_n because it solves a linear program over B_n, and every such linear program achieves its minimum at a vertex. Since the vertices of B_n are exactly the permutation matrices, and the algorithm searches only among them, its convergence to an optimal vertex Π^* is guaranteed.

The problems with this are three-fold:
1. The Hungarian algorithm converges in (worst-case) $O(n^3)$ time. And in large n-dimensional space, there may be multiple vertices nearby some A, leading to large permutation error.
2. We may end up with underflow error for A, given that K is the exponentiation of C, which may be a rough estimate with large distances as entries.
Proposed solutions:
1. Use PCA filters to reduce n.
2. Tune ϵ, and dynamically change based on std of the cost matrix to capture a higher uncertainty, or less flat A.

PCA Filter

Query a point $x_i \in X$, where $X \in \mathbb{R}^{d \times n}$ (the full point cloud), and find its k nearest neighbors (e.g. 20) using k-d trees. Then, produce a centered matrix with each row as a neighbor, and compute the $d \times d$ covariance matrix. Next, use eigen-decomposition to find the eigenvalues, i.e., the principal axis lengths, $\frac{1}{\sqrt{\lambda_k}}$ for $k = 1, \ldots, d$ of the local ellipsoid of inertia (by definition of an ellipsoid formula). Then, we want to estimate curvature or anisotropy (how 'stretched' the local neighborhood is) κ_i, which we can do in several ways by manipulating λ. By experiment, the scoring method with the least error (though not by much) is:

$$\kappa_i = \frac{\lambda_3 - \lambda_1}{\lambda_1} \tag{15}$$

where λ_3 is the largest eigenvalue and λ_1 is the smallest. This shows how uneven the spread is, where a 0 indicates a spherical ellipsoid of inertia (due to similar eigenvalues) or uniform neighbor distribution, and 1 indicates an elongated neighbor distribution, implying a more interesting geometric structure. Finally, we retain the top highest scores based on a keep ratio, or percentage of n (e.g. 30%), thereby reducing n to a subsample of bin-like, differentiated geometries.

PCA filtering also allows compute a best rotation matrix in the filtered points cloud, which can be applied to the full one, since the filtered data is a good approximation.

Results

Side by side, we compare PCA-filtered Sinkhorn & Hungarian code vs. Kolpakov's original code. First, I sub-sampled some n of the point cloud for each method, computing the ground truths for the rotation matrix of that sampled point cloud. Then, each method outputs a 3×3 rotation matrix, whose error can be determined against the ground truth with δ_o. The PCA-filter, for instance, computes the same-sized rotation matrix even though it is doing so for a filtered cloud (e.g. 300 points). However, alignment and permutation error wouldn't be possible to compare since they are of different sizes (e.g. 300×300 vs $n \times n$). The following are concatenated results over the same random seeds (important in matrix initialization and comparability).

Table 1: Summary of Rotation Error Metrics
(n = 1000, bunny.ply, average over 100 seeds)

Method	Metric	Mean	Median	Std
E–Init + Sinkhorn + Hungarian	Rotation Error δ_o	1.423252	2.000000	1.189266
	Time per Seed (s)		0.482412	
E–Init Only	Rotation Error δ_o	1.582157	2.000000	1.287160
	Time per Seed (s)		0.038381	

Table 2: Summary of Rotation Error Metrics
(n = 1000, cow.ply, average over 100 seeds)

Method	Metric	Mean	Median	Std
E–Init + PCA + Sinkhorn + Hungarian	Rotation Error δ_o	1.501880	2.000000	1.106508
	Time per Seed (s)		0.432520	
E–Init Only	Rotation Error δ_o	1.605671	2.000000	1.192401
	Time per Seed (s)		0.033516	

Table 3: Summary of Rotation Error Metrics
(n = 1000, brain.ply, average over 100 seeds)

Method	Metric	Mean	Median	Std
E–Init + PCA + Sinkhorn + Hungarian	Rotation Error δ_o	1.291127	2.000000	1.083047
	Time per Seed (s)		0.444098	
E–Init Only	Rotation Error δ_o	1.144853	2.000000	1.024359
	Time per Seed (s)		0.031834	

Table 4: Summary of Rotation Error Metrics
(n = 1000, elephant.ply, average over 100 seeds)

Method	Metric	Mean	Median	Std
E–Init + Sinkhorn + Hungarian	Rotation Error δ_o	1.164853	2.000000	1.021331
	Time per Seed (s)		0.376670	
E–Init Only	Rotation Error δ_o	1.197990	2.000000	1.060575
	Time per Seed (s)		0.032513	

Across multiple .ply files, I can conclude that the PCA-filtered Sinkhorn Hungarian version outperforms the original in rotation error, yet takes on average .4 seconds longer per run and on brain.ply it performs worse.