

# Lecture 18

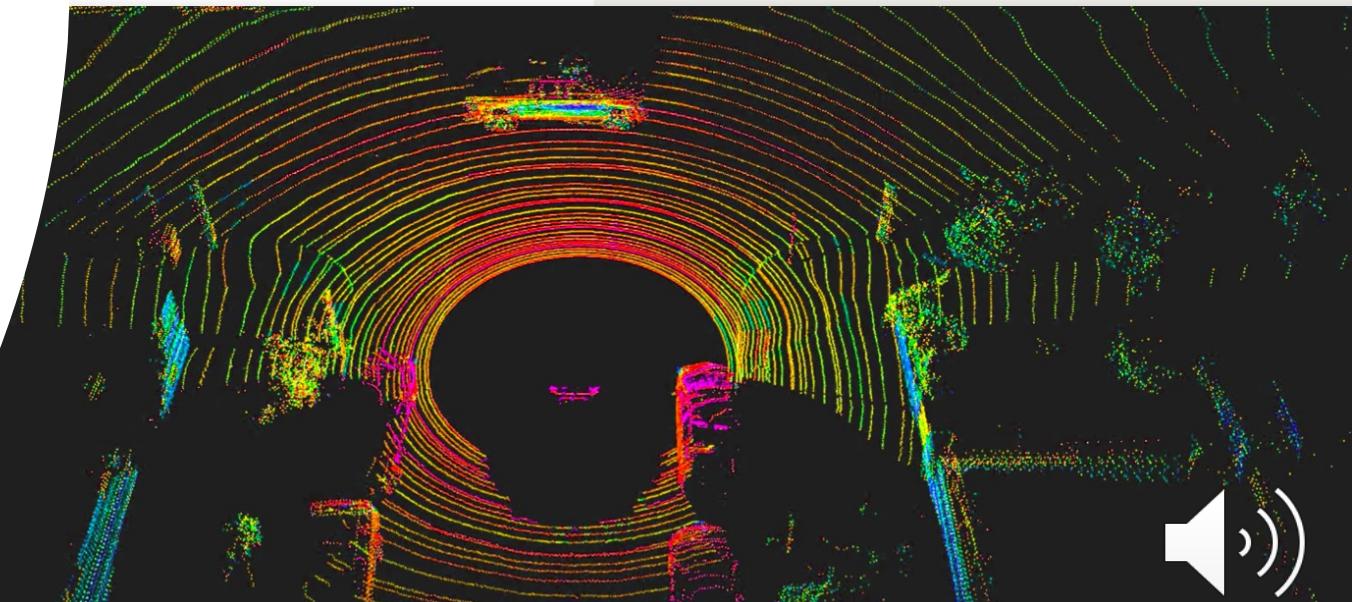
## ICP and SLAM

CS 3630



# LIDAR

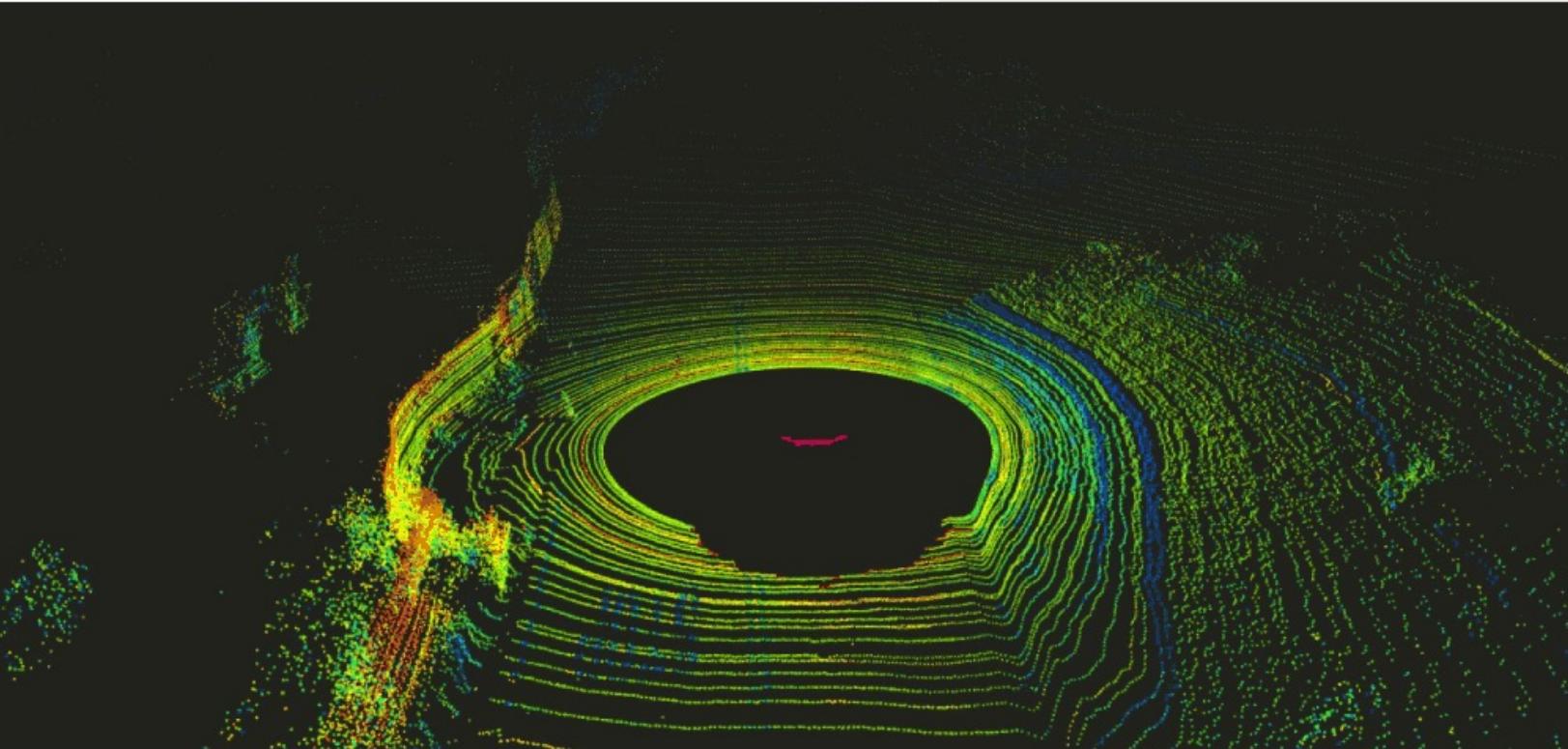
- Superpowers:
  - 360 Visibility
  - Accurate depth!
- Almost all AV prototypes have them (not all 360)



[Images and exposition take from excellent Voyage Blog post](https://news.voyage.auto/an-introduction-to-lidar-the-key-self-driving-car-sensor-a7e405590cff)

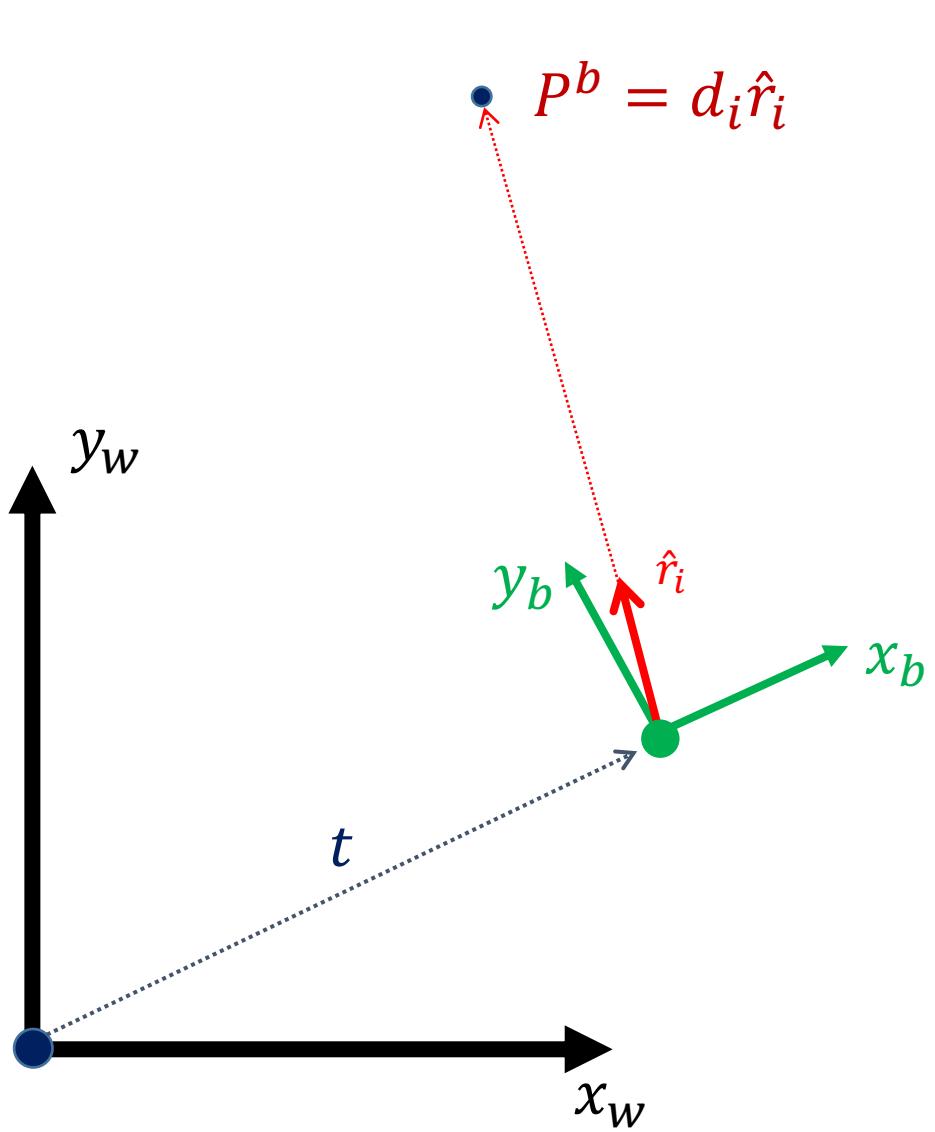
<https://news.voyage.auto/an-introduction-to-lidar-the-key-self-driving-car-sensor-a7e405590cff>

# Example



[Images and exposition take from  
excellent Voyage Blog post](#)

# Coordinate Transformations for LIDAR Data



$$R_b^w = \begin{bmatrix} x_b \cdot x_w & y_b \cdot x_w \\ x_b \cdot y_w & y_b \cdot y_w \end{bmatrix}$$

$$t^w = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$\boxed{\mathbf{P}^w = R_b^w \mathbf{P}^b + \mathbf{t}^w}$$

Or, we can write this using homogeneous transformations as:

$$\begin{bmatrix} \mathbf{P}^w \\ 1 \end{bmatrix} = \begin{bmatrix} R_b^w & \mathbf{t}^w \\ 0_2 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}^b \\ 1 \end{bmatrix}$$



# Localization using ICP

As the sensor moves through the world, it collects a data set (a point cloud) for multiple positions.

The localization problem is to infer the pose of the sensor, given the point clouds for successive scans.

This requires:

1. finding correspondences between data points in successive images, and
2. computing the relative pose for two successive scans, given the set of point correspondences.

We'll solve this problem using the ***Iterative Closest Points*** algorithm, also known as ***ICP***.



# Localization with LIDAR

- ICP = **Iterated Closest Points**:
- Call current scan  $S$ , map  $M$
- Predict pose from motion model:  
use other sensors if available
- Iterate:
  - For every point  $s$ : find closest  $m$
  - Re-estimate pose
- In practice:
  - outlier rejection to account for moving objects, unmodeled structures, parked cars etc...

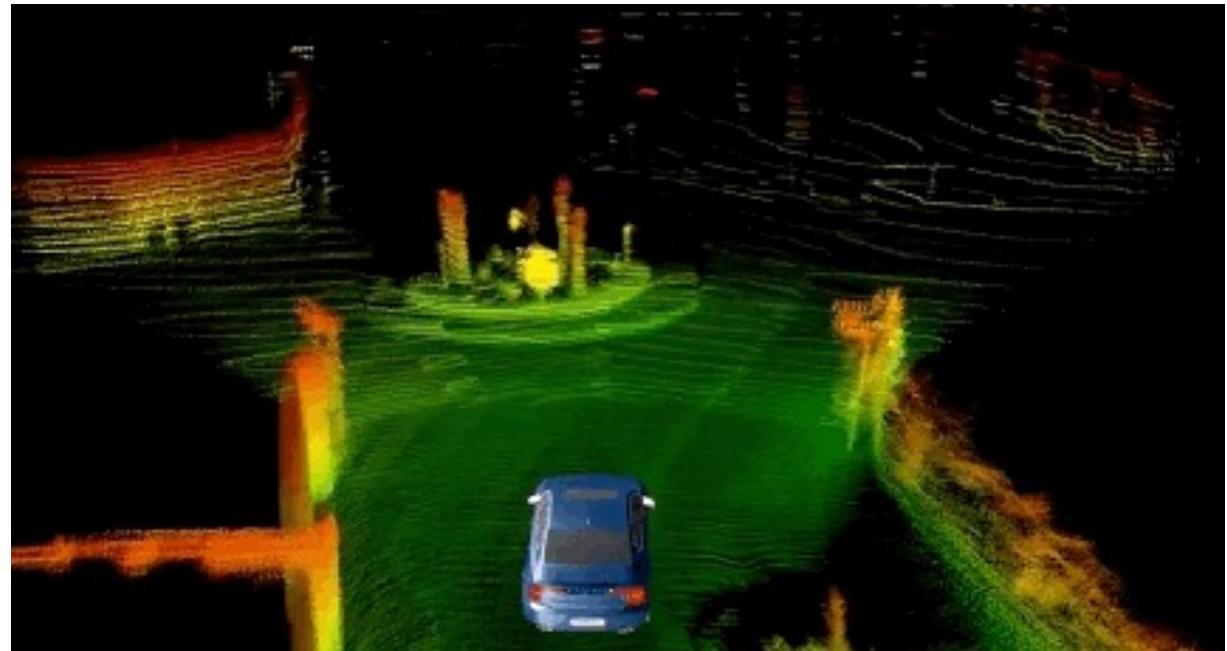
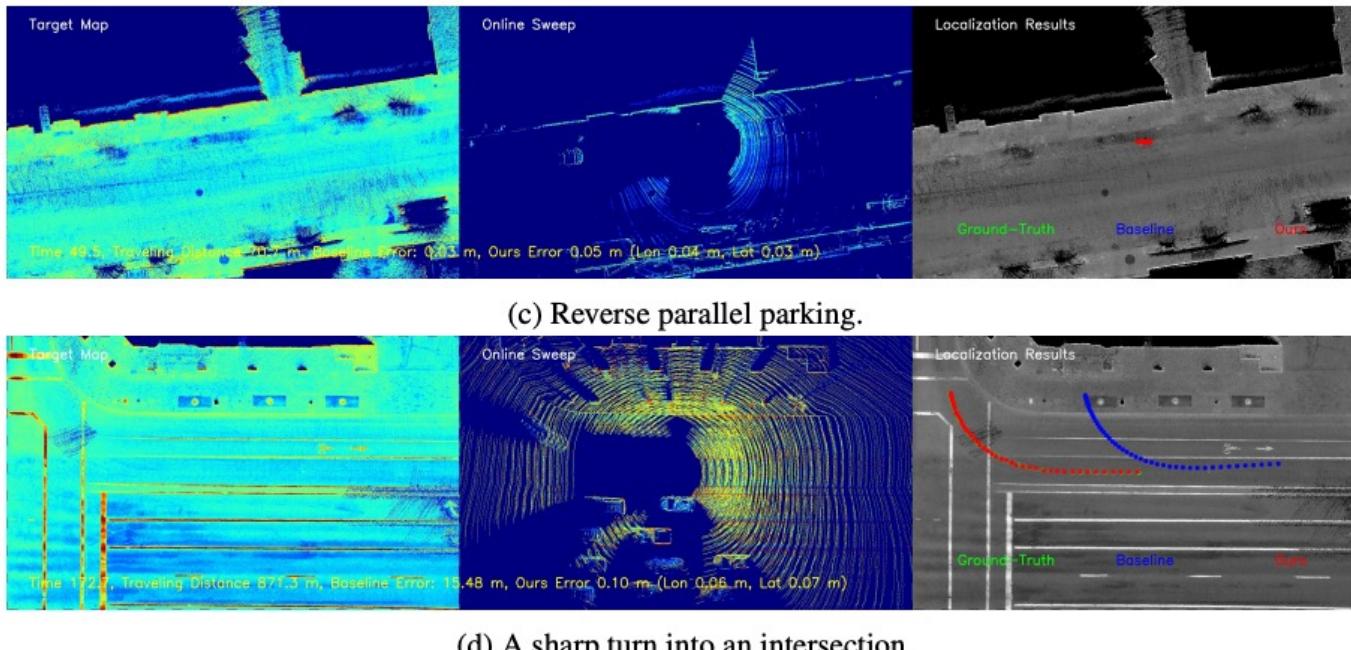


Image Credits: Innoviz



# Still an active area of research

- E.g., recent paper from Uber ATG
- “reliable and accurate localization remains an open problem,”
- “[ICP] can lead to high-precision localization, but remain vulnerable in the presence of geometrically non-distinctive or repetitive environments, such as tunnels, highways, or bridges”



Ioan Andrei Bărsan<sup>\*,1,2</sup> Shenlong Wang<sup>\*,1,2</sup> Andrei Pokrovsky<sup>1</sup> Raquel Urtasun<sup>1,2</sup>  
<sup>1</sup>Uber ATG, <sup>2</sup>University of Toronto  
{andreib, slwang, andrei, urtasun}@uber.com

**Abstract:** In this paper we propose a real-time, calibration-agnostic and effective localization system for self-driving cars. Our method learns to embed the online LiDAR sweeps and intensity map into a joint deep embedding space. Localization is then conducted through an efficient convolutional matching between the embeddings. Our full system can operate in real-time at 15Hz while achieving centimeter level accuracy across different LiDAR sensors and environments. Our experiments illustrate the performance of the proposed approach over a large-scale dataset consisting of over 4000km of driving.

**Keywords:** Deep Learning, Localization, Map-based Localization

## 1 Introduction

One of the fundamental problems in autonomous driving is to be able to accurately localize the vehicle in real time. Different precision requirements exist depending on the intended use of the localization system. For routing the self-driving vehicle from point A to point B, precision of a few meters is sufficient. However, centimeter-level localization becomes necessary in order to exploit high definition (HD) maps as priors for robust perception, prediction, and safe motion planning.

Despite many decades of research, reliable and accurate localization remains an open problem, especially when very low latency is required. Geometric methods, such as those based on the iterative closest-point algorithm (ICP) [1, 2] can lead to high-precision localization, but remain vulnerable in the presence of geometrically non-distinctive or repetitive environments, such as tunnels, highways, or bridges. Image-based methods [3, 4, 5, 6] are also capable of robust localization, but are still behind geometric ones in terms of outdoor localization precision. Furthermore, they often require capturing the environment in different seasons and times of the day as the appearance might change dramatically.

A promising alternative to these methods is to leverage LiDAR intensity maps [7, 8], which encode information about the appearance and semantics of the scene. However, the intensity of commercial LiDARs is inconsistent across different beams and manufacturers, and prone to changes due to environmental factors such as temperature. Therefore, intensity based localization methods rely heavily on having very accurate intensity calibration of each LiDAR beam. This requires careful fine-tuning of each vehicle to achieve good performance, sometimes even on a daily basis. Calibration can be a very laborious process, limiting the scalability of this approach. Online calibration is a promising solution, but current approaches fail to deliver the desirable accuracy. Furthermore, maps have to be re-captured each time we change the sensor, e.g., to exploit a new generation of LiDAR.

In this paper, we address the aforementioned problems by learning to perform intensity based localization. Towards this goal, we design a deep network that embeds both LiDAR intensity maps and online LiDAR sweeps in a common space where calibration is not required. Localization is then simply done by searching exhaustively over 3-Dof poses (2D position on the map manifold plus rotation), where the score of each pose can be computed by the cross-correlation between the embeddings. This allows us to perform localization in a few milliseconds on the GPU.

We demonstrate the effectiveness of our approach in both highway and urban environments over 4000km of roads. Our experiments showcase the advantages of our approach over traditional methods, such as the ability to work with uncalibrated data and the ability to generalize across different LiDAR sensors.

# Topics

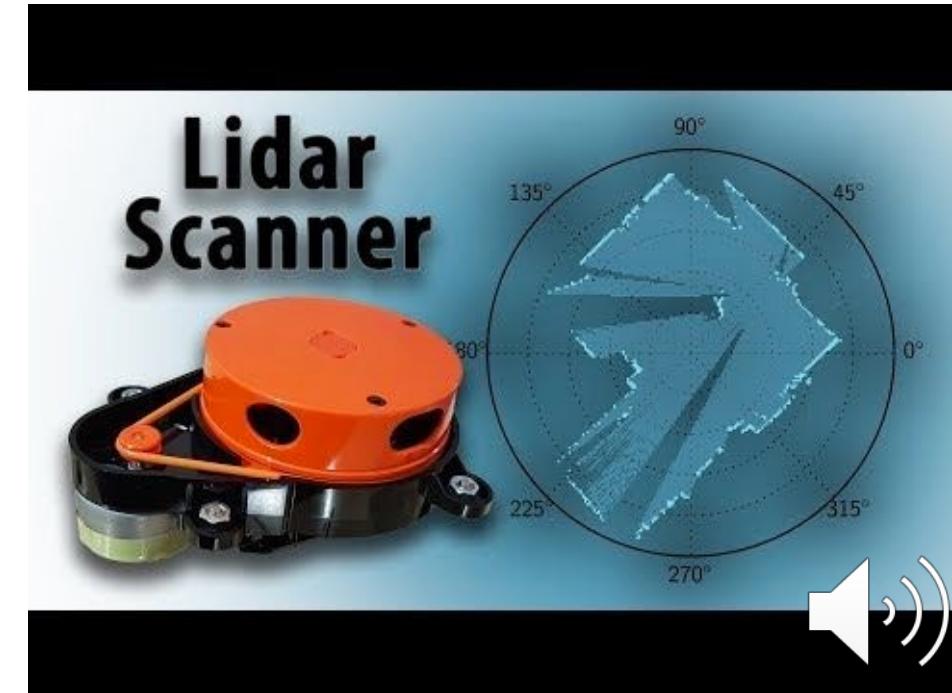
- 1. Applications**
- 2. Basic ICP Algorithm**
- 3. ICP Variants**

Includes slides adapted from Marc Pollefeys and James Hayes.

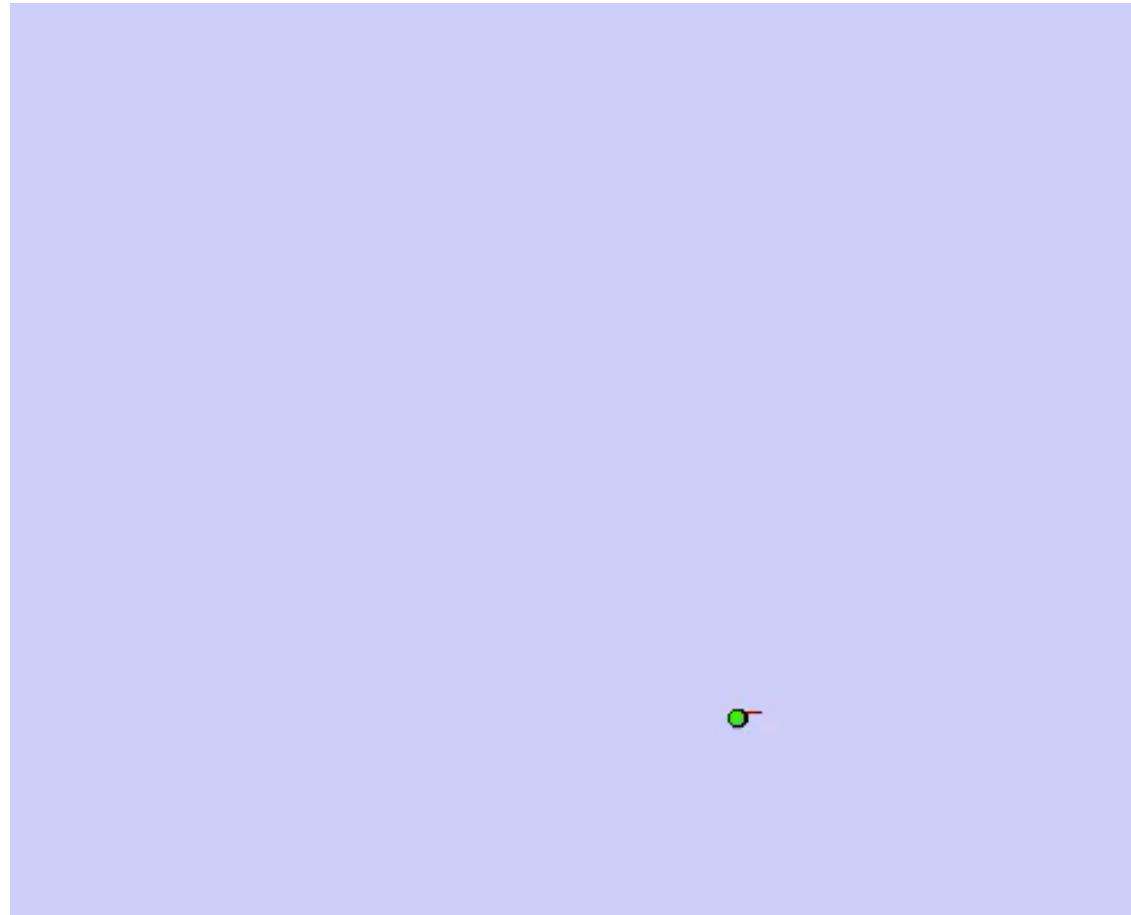


# Motivation

- 3D Scanners are becoming more and more prevalent
- 2D lasers now built into vacuuming robots
- Sensor of choice in autonomous vehicles
- Simple way to do SLAM



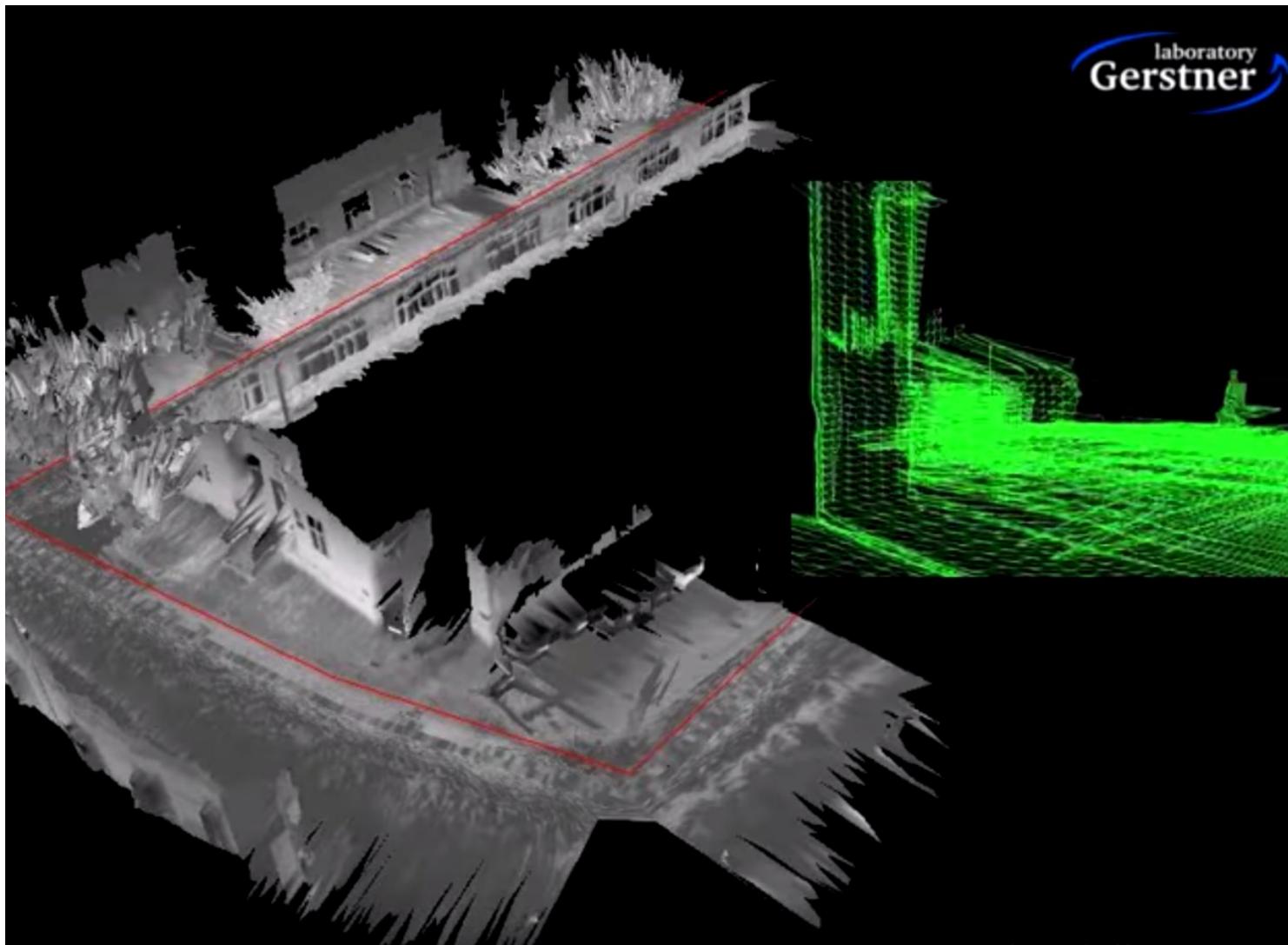
# ICP in Robotics



<https://www.youtube.com/watch?v=Ni8OFNyC5RY>  
[https://www.youtube.com/watch?v=9rTkUZ7HV\\_o](https://www.youtube.com/watch?v=9rTkUZ7HV_o)



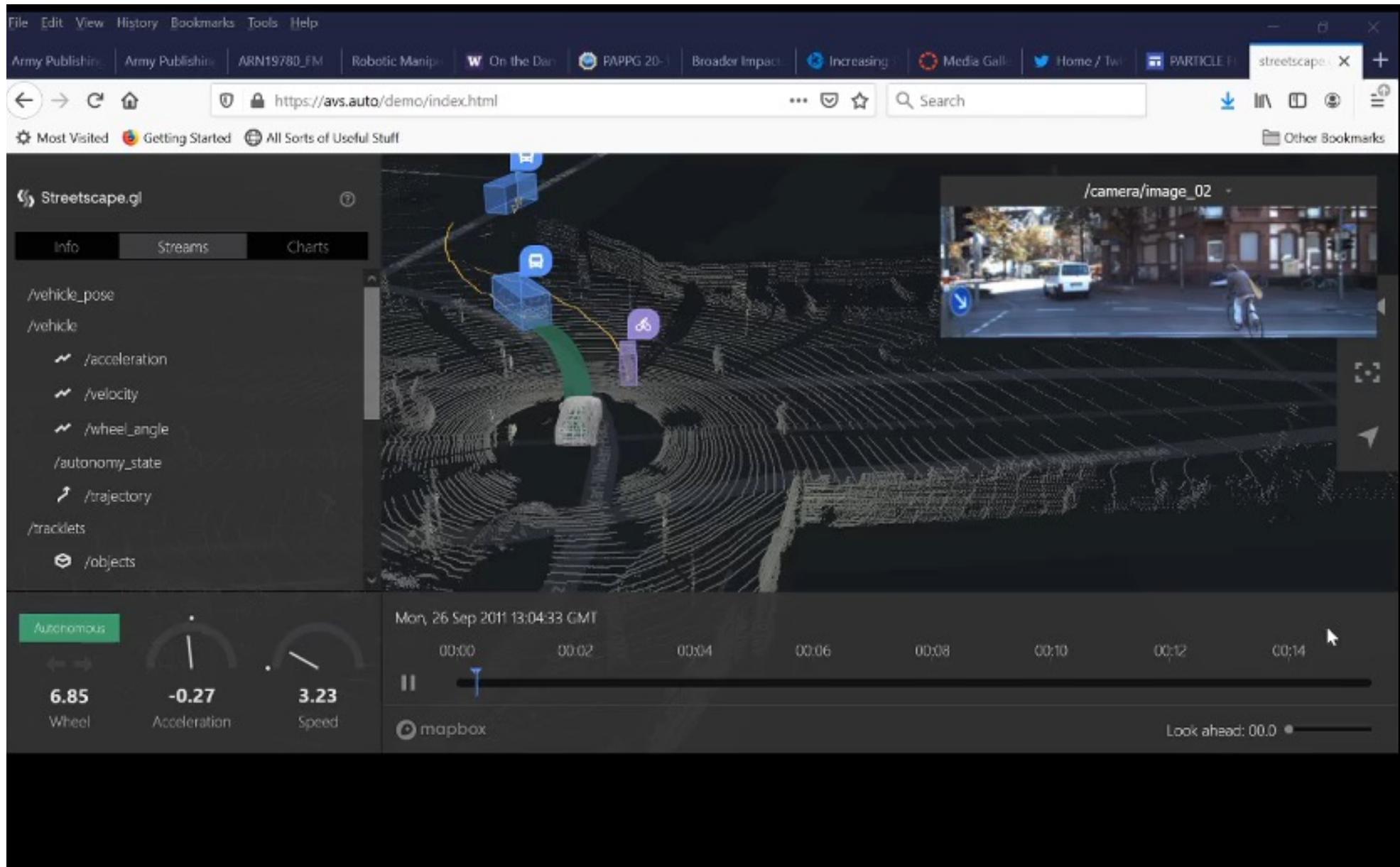
# ICP Outdoors



Gerstner Lab, Prague

# Uber AVS

Try it live: <https://avs.auto/demo/index.html>



# Aligning 3D Data

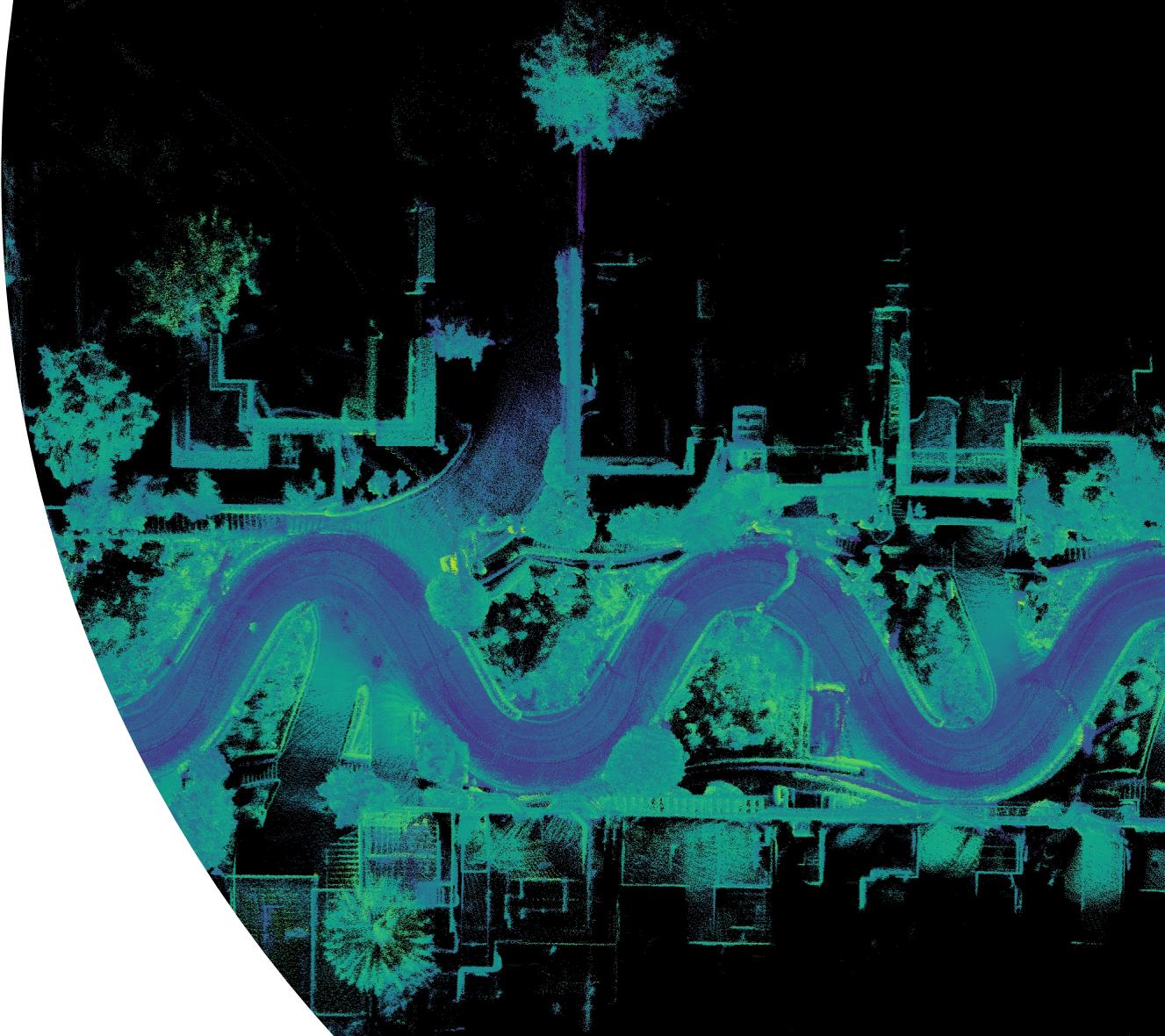
- How to find corresponding points?
- How to calculate a transform between two different point clouds?

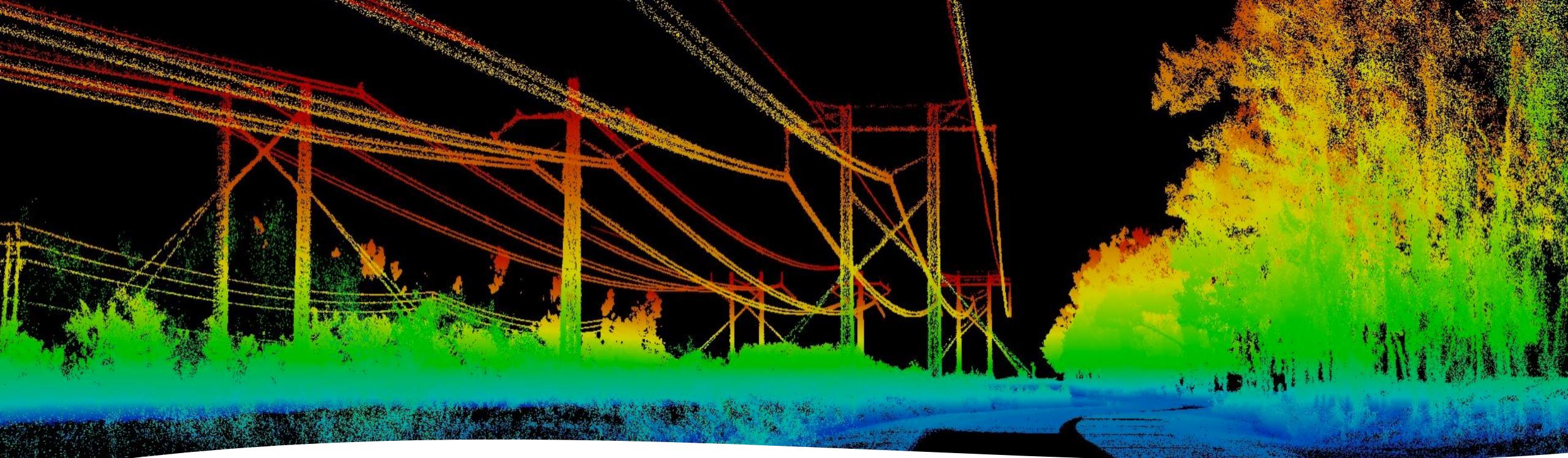


# Fitting and Alignment: Methods

---

- Global optimization / Search for parameters
  - Least squares fit
  - Robust least squares
  - Other parameter search methods
- Hypothesize and test
  - Generalized Hough transform
  - RANSAC
- *Iterative Closest Points (ICP)*





<https://www.terra-drone.net/angola/lidar-powerlines/>

# Iterative Closest Points (ICP) Algorithm

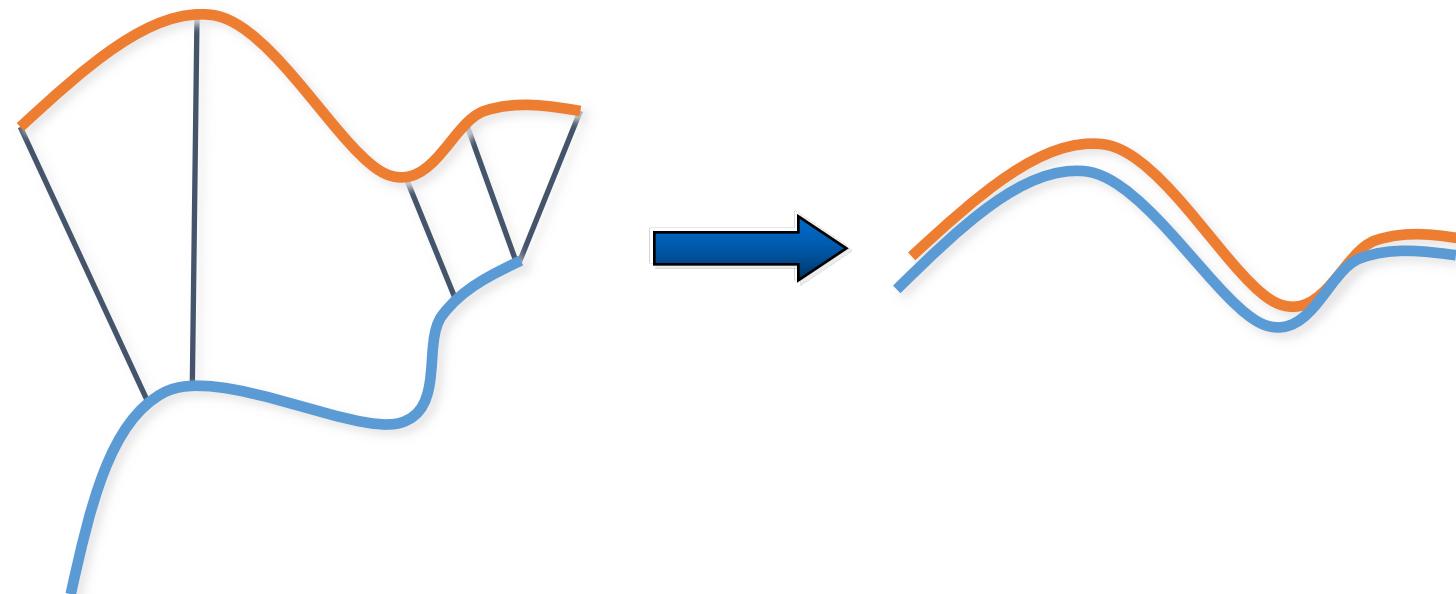
**Goal:** estimate transform between two dense sets of points

1. **Initialize** transformation (e.g., compute difference in means and scale)
2. **Assign** each point in {Set 1} to its nearest neighbor in {Set 2}
3. **Estimate** transformation parameters using least squares
4. **Transform** the points in {Set 1} using estimated parameters
5. **Repeat** steps 2-4 until change is very small



# Aligning 3D Data

Assume closest points correspond to each other,  
compute the best transform...

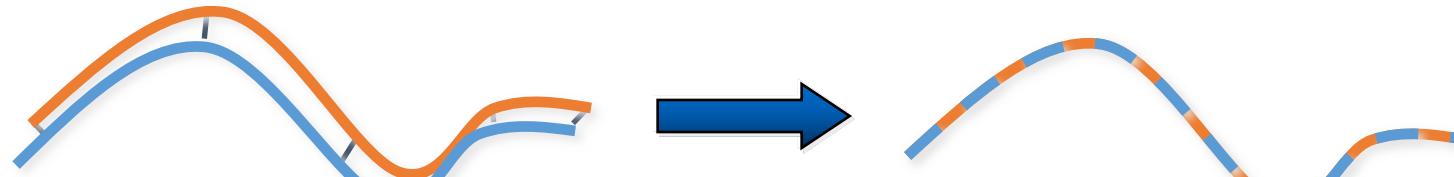


# Aligning 3D Data

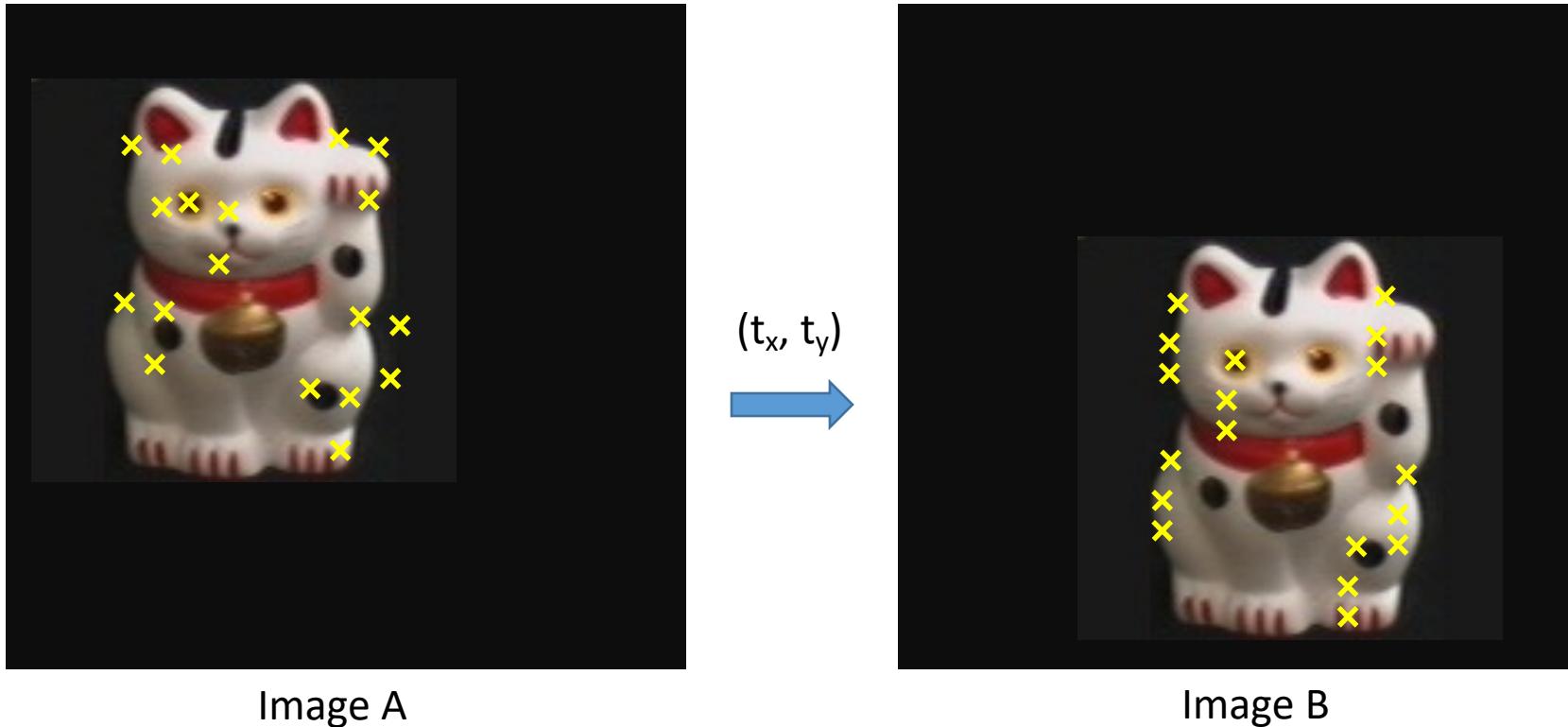
... and iterate to find alignment

Iterated Closest Points (ICP) [Besl & McKay 92]

Converges if starting position “close enough”



# Example: solving for translation



## ICP solution

1. Find nearest neighbors for each point
2. Compute transform using matches
3. Move points using transform
4. Repeat steps 1-3 until convergence

$$\begin{bmatrix} x_i^B \\ y_i^B \end{bmatrix} = \begin{bmatrix} x_i^A \\ y_i^A \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$



# Finding Nearest Neighbors

- Points in Image A are  $\mathcal{P}^A = \{P_1^A, \dots, P_n^A\}$
- Points in Image B are  $\mathcal{P}^B = \{P_1^B, \dots, P_n^B\}$
- For each point in  $P_i^A \in \mathcal{P}^A$  compute

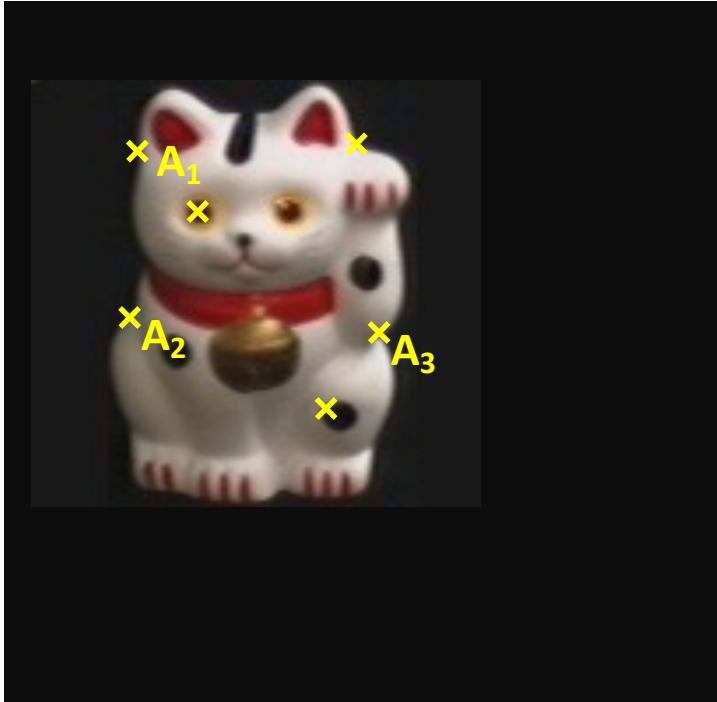
$$P^B = \arg \min_{p \in \mathcal{P}^B} \|p - P_i^A\|$$

- This gives us a set of matching pairs:

$$\{(P_1^A, P_{j_1}^B), \dots, (P_n^A, P_{j_n}^B)\}$$



# Example: solving for translation

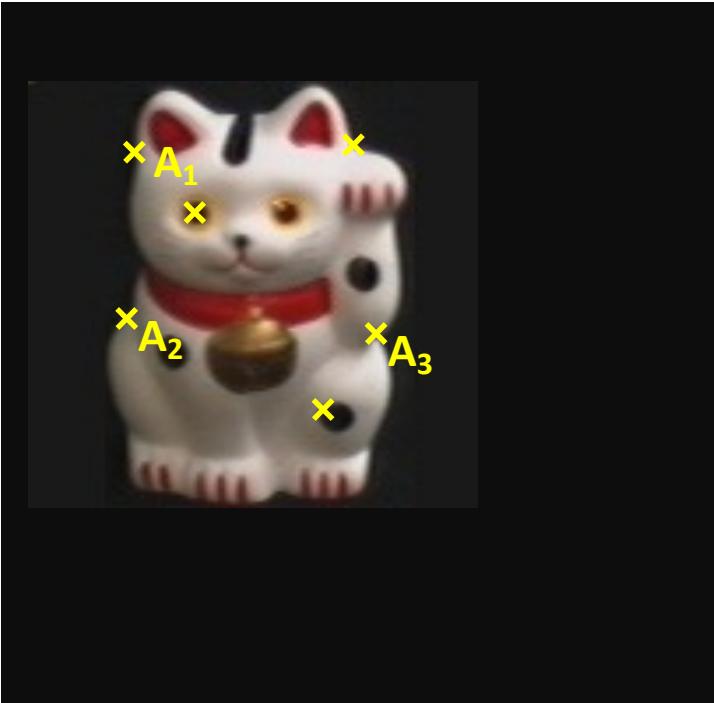


In this example, the matching process would yield

$$\{(A_1, B_2), (A_2, B_3), (A_3, B_1)\}$$



# Example: solving for translation

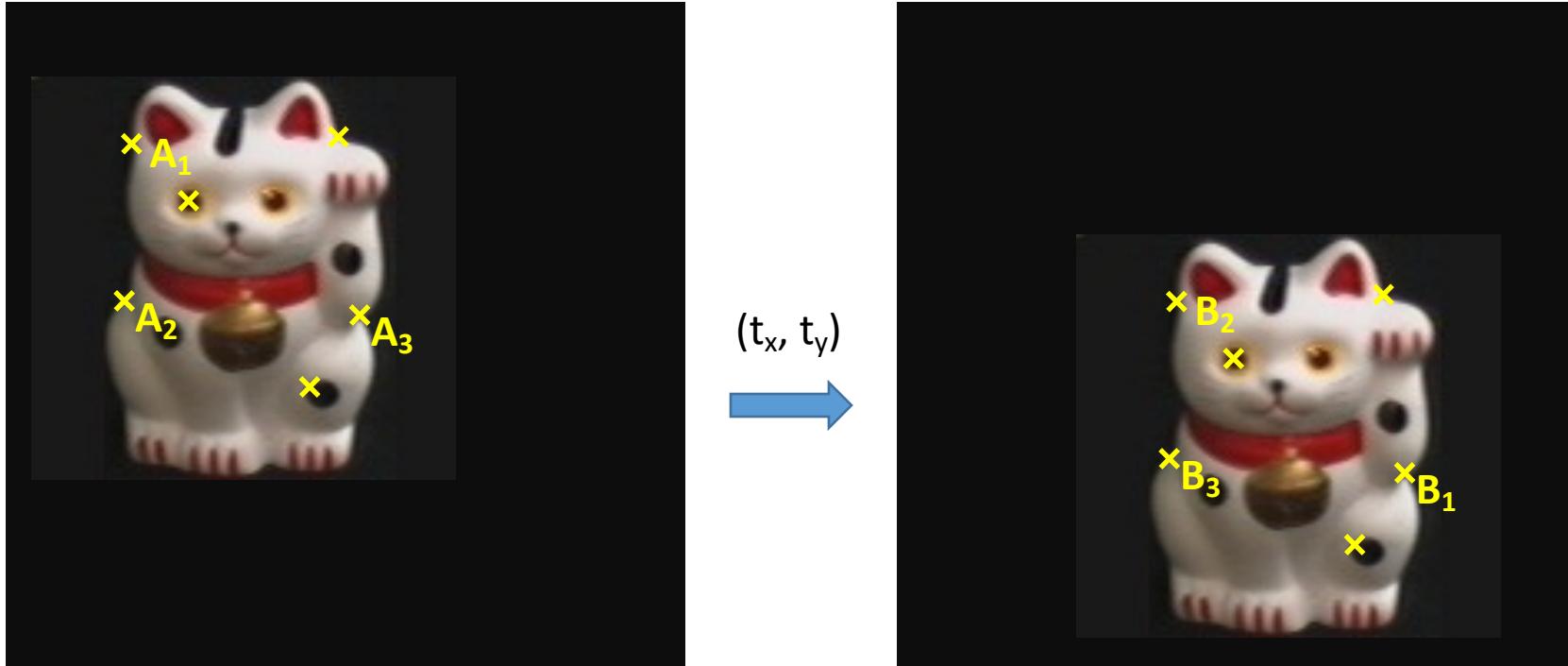


Given matched points in  $\{(A_1, B_2), (A_2, B_3), (A_3, B_1)\}$ , estimate the translation of the object

$$\begin{bmatrix} x_i^B \\ y_i^B \end{bmatrix} = \begin{bmatrix} x_i^A \\ y_i^A \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$



# Example: solving for translation



## Least squares solution

1. Write down objective function
2. Derived solution
  - a) Compute derivative
  - b) Compute solution
3. Computational solution
  - a) Write in form  $Ht=b$
  - b) Solve using pseudo-inverse  $t^* = H^+b$

$$\begin{bmatrix} x_i^B \\ y_i^B \end{bmatrix} = \begin{bmatrix} x_i^A \\ y_i^A \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x_1^B - x_1^A \\ y_1^B - y_1^A \\ \vdots \\ x_n^B - x_n^A \\ y_n^B - y_n^A \end{bmatrix}$$

(we can't use inverse  $t=H^{-1}b$  as  $H$  is not square  $\rightarrow$  pseudo-inverse)



# Solving for Translation

We have two equations for each correspondence:

$$\begin{aligned}t_x &= x_1^B - x_1^A \\t_y &= y_1^B - y_1^A \\t_x &= x_2^B - x_2^A \\t_y &= y_2^B - y_2^A \\\vdots \\t_x &= x_n^B - x_n^A \\t_y &= y_n^B - y_n^A\end{aligned}$$

*We can write this as a matrix equation:*

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x_1^B - x_1^A \\ y_1^B - y_1^A \\ \vdots \\ x_n^B - x_n^A \\ y_n^B - y_n^A \end{bmatrix} \rightarrow Ht = b$$

Useful facts about  $H$ :

$$H \in \mathbb{R}^{2n \times 2}$$

$$H^T \in \mathbb{R}^{2 \times 2n}$$

$$(H^T H)^{-1} \in \mathbb{R}^{2 \times 2}$$

$$(H^T H)^{-1} H^T \in \mathbb{R}^{2 \times 2n}$$

*Now, we can build the pseudoinverse for  $H$ :*

1.  $(H^T H)^{-1} H^T H = I$
2.  $(H^T H)^{-1} H^T = H^+$
3.  $(H^T H)^{-1} H^T H = H^+ H$   
 $= (H^T H)^{-1} H^T H = I$

*There are conditions and caveats for this, but we won't worry about these in this class.*

Cool: We can “invert” a non-square matrix!



# Computing the Translation (just simple algebra)

$$H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \quad H^T = \begin{bmatrix} 1 & 0 & \dots & 1 & 0 \\ 0 & 1 & \dots & 0 & 1 \end{bmatrix}$$

↗

$$H^T H = \begin{bmatrix} 1 & 0 & \dots & 1 & 0 \\ 0 & 1 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} n & 0 \\ 0 & n \end{bmatrix} \quad (H^T H)^{-1} = \begin{bmatrix} \frac{1}{n} & 0 \\ 0 & \frac{1}{n} \end{bmatrix}$$

$$\rightarrow H^+ = (H^T H)^{-1} H^T = \begin{bmatrix} \frac{1}{n} & 0 \\ 0 & \frac{1}{n} \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 1 & 0 \\ 0 & 1 & \dots & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{n} & 0 & \dots & \frac{1}{n} & 0 \\ 0 & \frac{1}{n} & \dots & 0 & \frac{1}{n} \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} \frac{1}{n} & 0 & \dots & \frac{1}{n} & 0 \\ 0 & \frac{1}{n} & \dots & 0 & \frac{1}{n} \end{bmatrix} \begin{bmatrix} x_1^B - x_1^A \\ y_1^B - y_1^A \\ \vdots \\ x_n^B - x_n^A \\ y_n^B - y_n^A \end{bmatrix} = \begin{bmatrix} \frac{1}{n} \sum_i x_i^B - x_i^A \\ \frac{1}{n} \sum_i y_i^B - y_i^A \end{bmatrix}$$

OK... it's a bit disappointing to do all of that work only to find that we estimate the translation as the average translation of all the points, but the good news is that this pseudoinverse approach generalizes to more interesting cases.



# Estimating Pose for $SO(2)$

- Suppose two sets of data points differ only by a rotation (e.g., the sensor merely rotates to collect successive scans).
- To start, suppose we have only one data point in each 2D image:

$$P^A = \begin{bmatrix} a_x \\ a_y \end{bmatrix}, \quad P^B = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

- Consider the expression

$$P^A(P^B)^T = \begin{bmatrix} a_x \\ a_y \end{bmatrix} [b_x \quad b_y] = \begin{bmatrix} b_x a_x & b_y a_x \\ b_x a_y & b_y a_y \end{bmatrix}$$

- Now, recall the construction of the rotation matrix  $R_b^a$

$$R_B^A = \begin{bmatrix} x_b \cdot x_a & y_b \cdot x_a \\ x_b \cdot y_a & y_b \cdot y_a \end{bmatrix}$$

There's a striking similarity between these, and this motivates (but does not prove or even really explain) the following:

- Let  $H = \sum P_i^A (P_i^B)^T$  where the sum is taken over corresponding pairs.
- The best estimate for the rotation matrix  $\hat{R}_B^A$  is the matrix  $R \in SO(2)$  that is closest to  $H$  in the least squares sense!



# Estimating Pose for $SE(2)$

- We know how to solve for the rotation in the case where the two point sets differ only by rotation.
- Suppose now that there is also a translation.
- Since all points undergo the same translation, we compute the centroid for each point set, and then use the translation between the two centroids as the estimate of translation:

$$C^A = \frac{1}{N} \sum P_i^A, \quad C^B = \frac{1}{N} \sum P_i^B$$

- Now, merely “subtract out the translation, and we can use the previous method to estimate the rotation:

$$H = \sum (P_i^A - C^A)(P_i^B - C^B)^T$$

and, as before,  $\hat{R}_B^A$  is the matrix  $R \in SO(2)$  that is closest to  $H$  in the least squares sense!

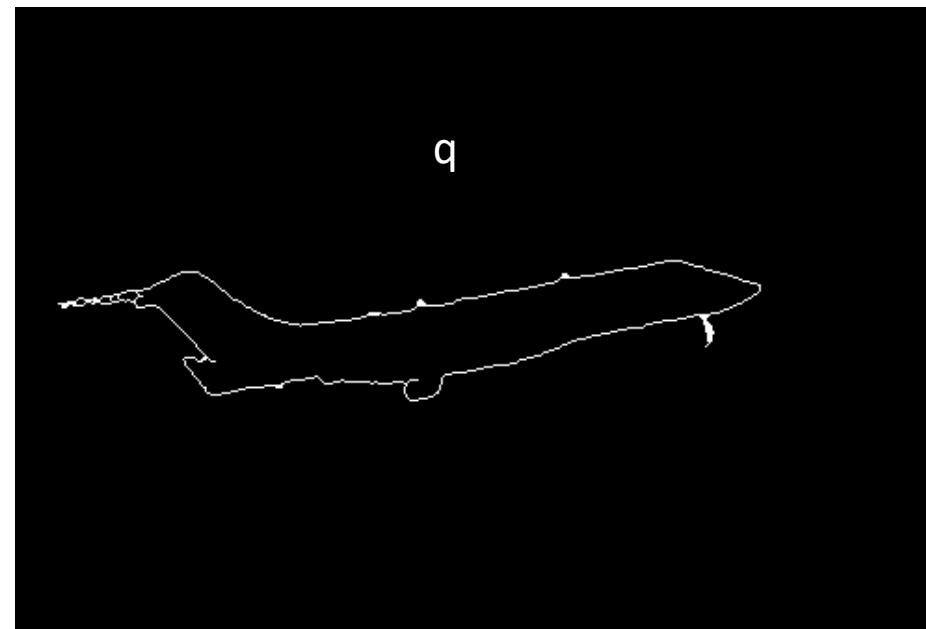
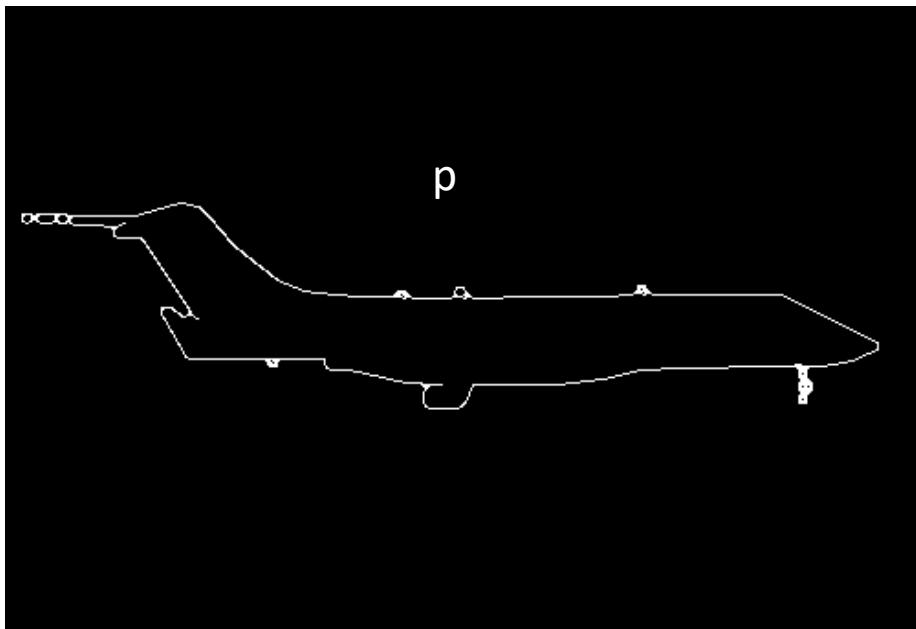
- For the translation, solve

$$C^A = \hat{R}_B^A C^B + \hat{t}_B^A$$



# Example: aligning boundaries in images

1. Extract edge pixels  $p_1..p_n$  and  $q_1..q_m$
2. Compute initial transformation (e.g., compute translation and scaling by center of mass, variance within each image)
3. Get nearest neighbors: for each point  $p_i$  find corresponding  $\text{match}(i) = \underset{j}{\operatorname{argmin}} \text{dist}(p_i, q_j)$
4. Compute transformation  $T$  based on matches
5. Warp points  $p$  according to  $T$
6. Repeat 3-5 until convergence



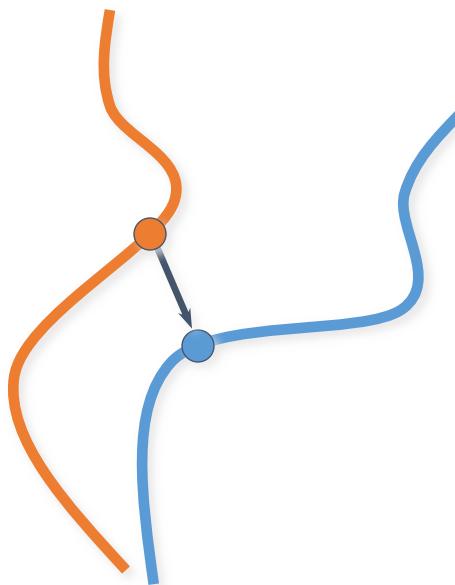
# ICP Variants

- Classic ICP algorithm not real-time
- To improve speed: examine stages of ICP and evaluate proposed variants
- [Rusinkiewicz & Levoy, 3DIM 2001]
  1. Selecting source points (from one or both meshes)
  2. Matching to points in the other mesh
  3. Weighting the correspondences
  4. Rejecting certain (outlier) point pairs
  5. Assigning an error metric to the current transform
  6. Minimizing the error metric



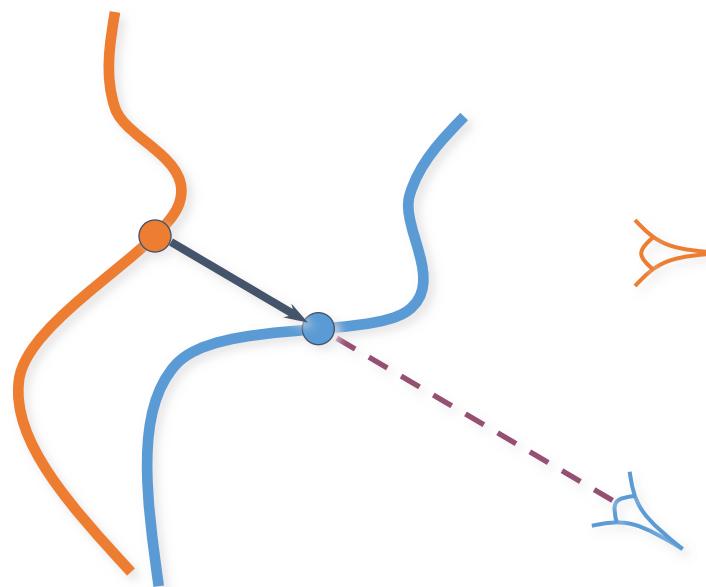
# Finding Corresponding Points

- Finding closest point is most expensive stage of ICP
  - Brute force search –  $O(n)$
  - Spatial data structure (e.g., k-d tree) –  $O(\log n)$
  - Voxel grid –  $O(1)$ , but large constant, slow preprocessing



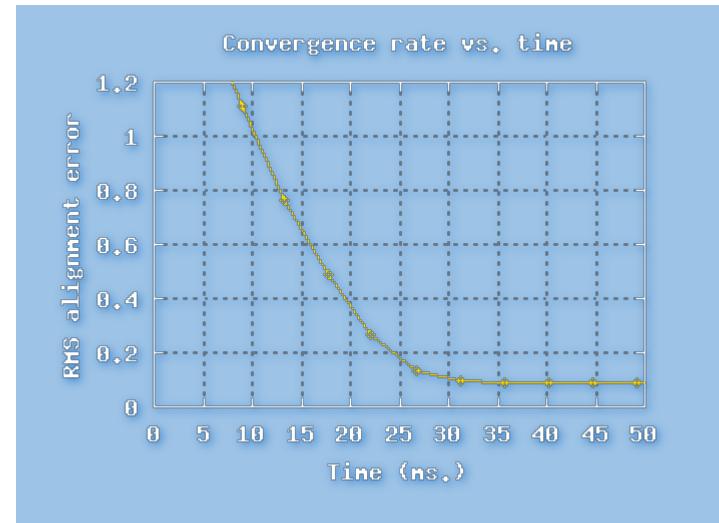
# Finding Corresponding Points

- For range images, simply project point [Blais 95]
  - Constant-time, fast
  - Does not require precomputing a spatial data structure



# High-Speed ICP Algorithm

- ICP algorithm with projection-based correspondences, point-to-plane matching can align meshes in a few tens of ms.  
(cf. over 1 sec. with closest-point)



[Rusinkiewicz & Levoy, 3DIM 2001]



# Summary

1. Applications are plentiful
2. Basic ICP Algorithm is simple (but slow)
3. ICP Variants can speed up

