Iterative Closest Point Algorithm

Dr. Francis Colas

11.11.2011

ETH Zürich

Introduction

Several problems:

- find object into a scene,
- estimate motion of sensor,
- compare two scenes,
- merge observations into a map,
- ...

Finding objects

Input:

- description of an object (mesh),
- scene (point clouds);

Objective:

- detect object,
- estimate position,
- estimate some parameters.

Sensor Motion

Input:

- sensor data (image or point cloud) at time t_1,
- sensor data at time t_2;

Objectives:

- estimate motion of the sensor,
- associate part of the data between frames,
- estimate some sensor parameters.

ETH Zürich

Scene Comparison

Input:

- sensor data from first point of view (time, space or sensor),
- sensor data from second point of view;

Objectives:

- find the differences,
- find the similarities.

Mapping

Input:

- some map,
- sensor data;

Objectives:

- integrate new data into map,
- find location of new data,
- finding overlap between map and data.

Introduction

Several problems:

- find object into a scene,
- estimate motion of sensor,
- compare two scenes,
- merge observations into a map,
- ...

Solution:

- ICP: Iterative Closest Point algorithm.

Iterative Closest Point

Definition:

- find transformation between 2 set of points;

Input:

- reference point cloud,
- data point cloud;

Output:

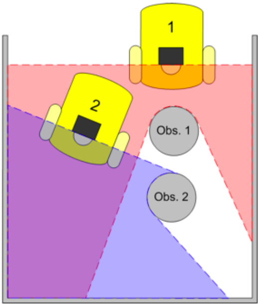
- transformation between reference and data:
  - 3 degrees of freedom in 2D,
  - 6 degrees of freedom in 3D.

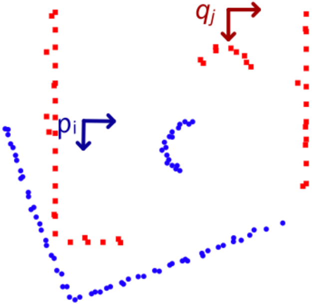
Algorithm

Outline:

- find corresponding points in both clouds,
- compute transformation minimzing error,
- move data point according to the transformation,
- loop...

Algorithm



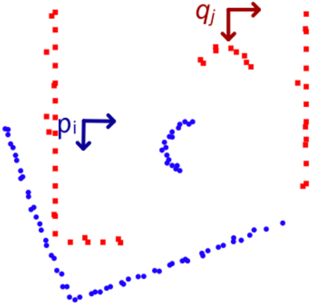


Images courtesy François Pomerleau

Algorithm

1. Preprocessing
2. Matching
3. Weighting
4. Rejection
5. Error
6. Minimization

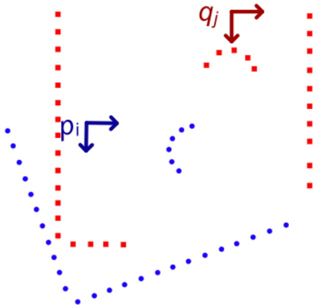
(Steps defined in Rusinkiewicz 01)



Autonomous Systems Lab

Algorithm

1. Preprocessing
2. Matching
3. Weighting
4. Rejection
5. Error
6. Minimization

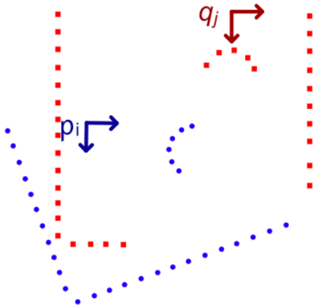


ETH zürich

Autonomous Systems Lab

Algorithm

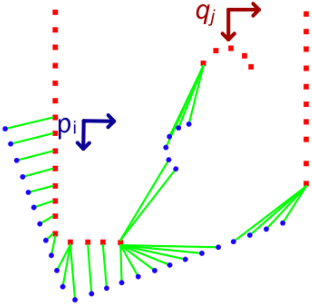
1. Preprocessing
2. **Matching**
3. Weighting
4. Rejection
5. Error
6. Minimization



ETH Zürich

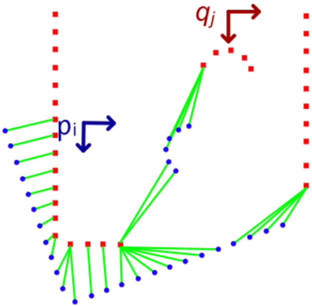
Algorithm

1. Preprocessing
2. **Matching**
3. Weighting
4. Rejection
5. Error
6. Minimization



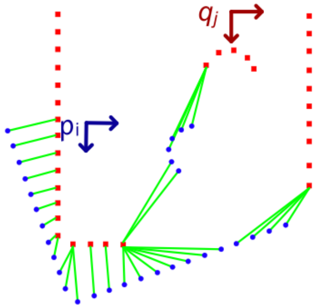
Algorithm

1. Preprocessing
2. Matching
3. **Weighting**
4. Rejection
5. Error
6. Minimization



Algorithm

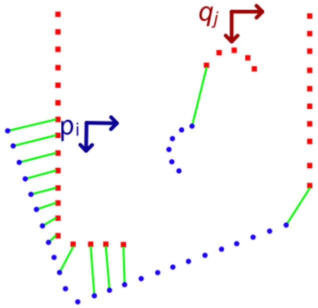
1. Preprocessing
2. Matching
3. Weighting
4. **Rejection**
5. Error
6. Minimization



Autonomous Systems Lab

Algorithm

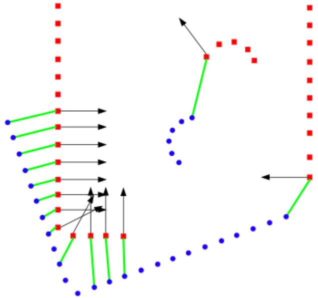
1. Preprocessing
2. Matching
3. Weighting
4. **Rejection**
5. Error
6. Minimization



ETH Zürich

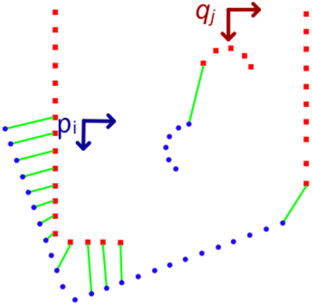
Algorithm

1. Preprocessing
2. Matching
3. Weighting
4. Rejection
5. **Error**
6. Minimization



Algorithm

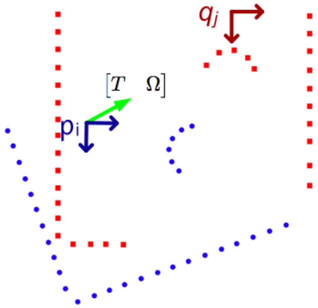
1. Preprocessing
2. Matching
3. Weighting
4. Rejection
5. Error
6. **Minimization**



Autonomous Systems Lab

Algorithm

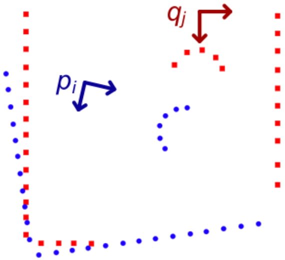
1. Preprocessing
2. Matching
3. Weighting
4. Rejection
5. Error
6. **Minimization**



ETH Zürich

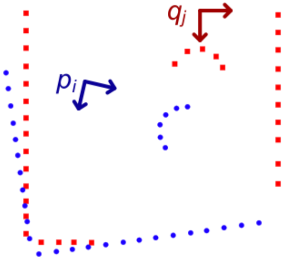
Algorithm

1. Preprocessing
2. Matching
3. Weighting
4. Rejection
5. Error
6. **Minimization**



Algorithm

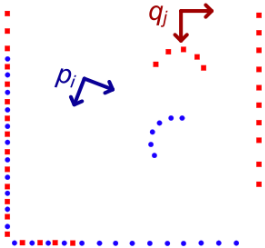
1. Preprocessing
2. Matching
3. Weighting
4. Rejection
5. Error
6. **Minimization**



Algorithm

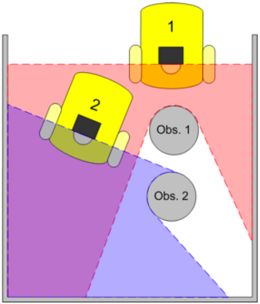
1. Preprocessing
2. Matching
3. Weighting
4. Rejection
5. Error
6. **Minimization**

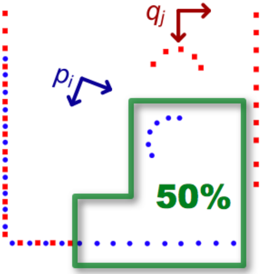
End conditions



ETH Zürich

Algorithm





ICP

Algorithm:

1. preprocessing
2. matching
3. weighting
4. rejection
5. error
6. minimization
7. loop to 2. unless convergence.

ICP

Algorithm:

1. preprocessing:
   - clean data,
2. matching
3. weighting
4. rejection
5. error
6. minimization
7. loop to 2. unless convergence.

ICP

Algorithm:

1. preprocessing
2. matching:
   - associate points from reference to data,
   - use neighbor search,
   - can use features,
3. weighting
4. rejection
5. error
6. minimization
7. loop to 2. unless convergence.

Autonomous Systems Lab

ICP

Algorithm:

1. preprocessing
2. matching
3. weighting:
   - change importance of some pairs,
4. rejection
5. error
6. minimization
7. loop to 2. unless convergence.

ETH zürich

ICP

Algorithm:

1. preprocessing
2. matching
3. weighting
4. rejection:
   - discard some pairs,
5. error
6. minimization
7. loop to 2. unless convergence.

ICP

Algorithm:

1. preprocessing
2. matching
3. weighting
4. rejection
5. error:
    - compute error for each pair,
6. minimization
7. loop to 2. unless convergence.

ETH Zürich

ICP

Algorithm:

1. preprocessing
2. matching
3. weighting
4. rejection
5. error
6. minimization:
   - find best transform;
7. loop to 2. unless convergence.

Summary

ICP:

- iterative algorithm,
- estimate matches,
- minimize error;

Features:

- similar to EM scheme (like Baum-Welch),
- local optimization,
- sensitive to overlap/outliers.

Matching

Objective:

- find point in reference to associate to each point in data,
- based on position,
- also normal vectors, colors...

Means:

- additional information as additional dimensions of point
- nearest neighbor search.

Done a lot:

- needs to be efficient.

Matching

Objective:

- find point in reference to associate to each point in data,
- based on position,
- also normal vectors, colors...

Means:

- additional information as additional dimensions of point
- nearest neighbor search.

Done a lot:

- needs to be efficient.

Matching

Objective:

- find point in reference to associate to each point in data,
- based on position,
- also normal vectors, colors...

Means:

- additional information as additional dimensions of point
- nearest neighbor search.

Done a lot:

- needs to be efficient.

Nearest Neighbor Search

Two main approaches:

- look at all points: *linear search*,
- look only where you want: *space partitioning*.

Features:

- exact or appoximate,
- complexity with respect to dimension.

Linear search

Exhaustive:

- compute distance for every point,
- keep current maximum;

Complexity:

- linear in number of points,
- depends on dimensionality only for distance function.

k-d tree

k-dimensional tree:

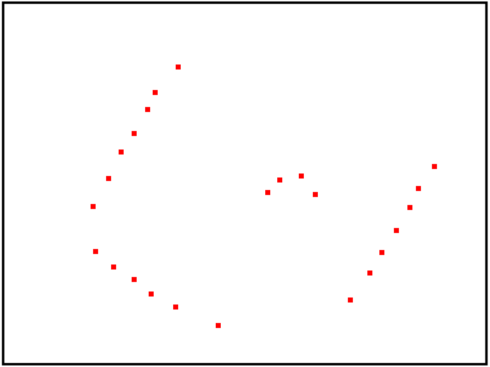
- split in 2 on middle or median point,
- according to the widest dimension,
- until a given bucket size.

Search:

- tree search,
- go down the tree to bucket,
- linear search in bucket.

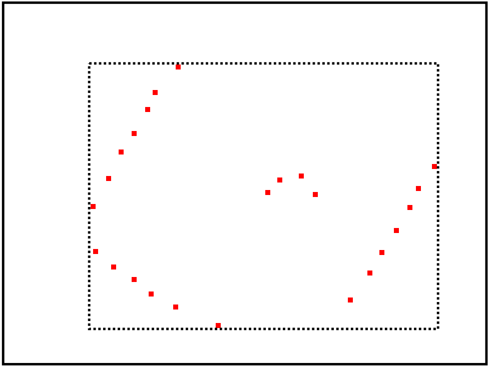
k-d tree

Build the tree for reference:



k-d tree

Build the tree for reference:

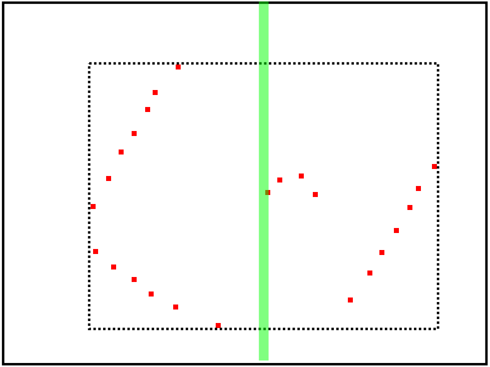


ETH Zürich

Autonomous Systems Lab

k-d tree

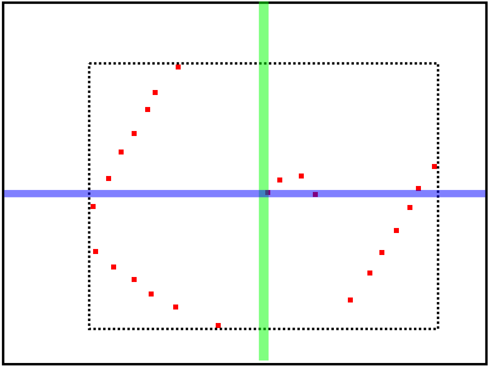
Build the tree for reference:



ETH Zürich

k-d tree

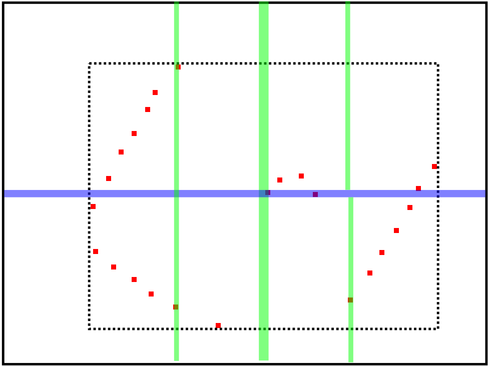
Build the tree for reference:



Autonomous Systems Lab

k-d tree

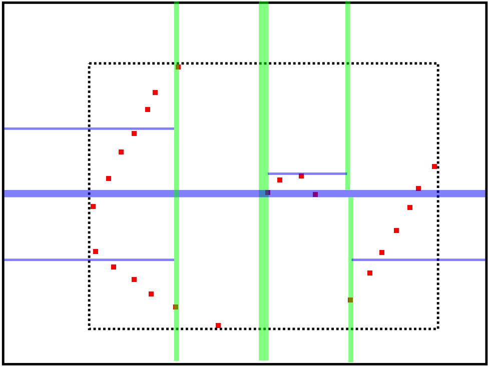
Build the tree for reference:



ETH zürich

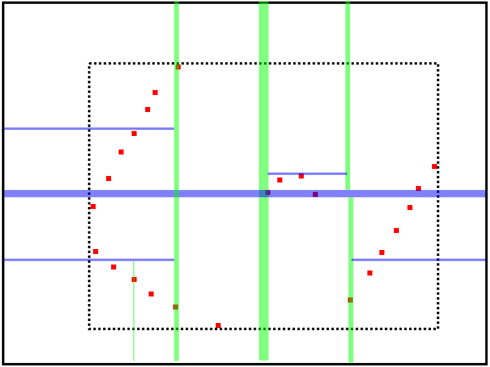
k-d tree

Build the tree for reference:



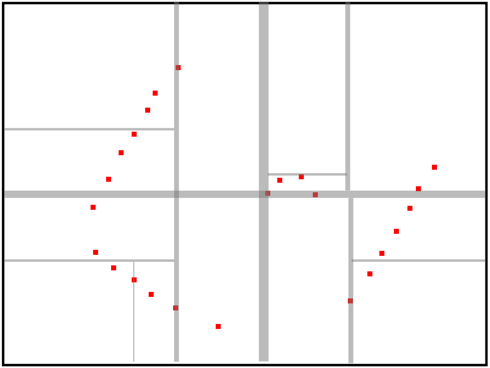
k-d tree

Build the tree for reference:



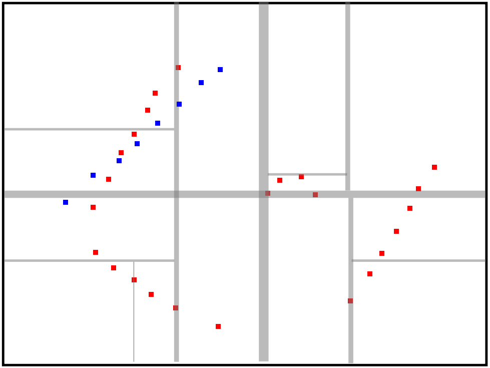
Nearest Neighbor Search in a *k*-d Tree

Looking for the neighbors of data points:



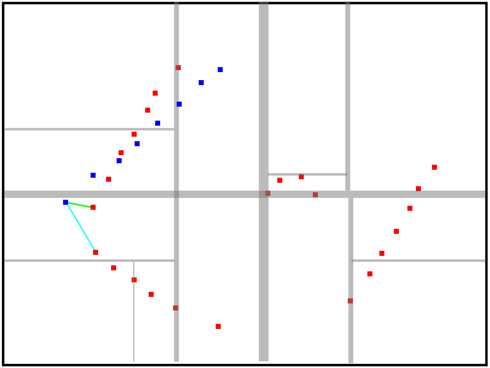
Nearest Neighbor Search in a k-d Tree

Looking for the neighbors of data points:



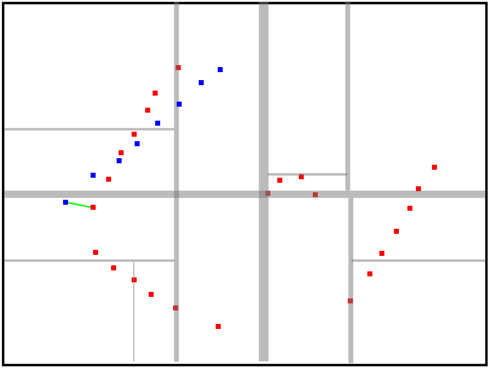
Nearest Neighbor Search in a k-d Tree

Looking for the neighbors of data points:



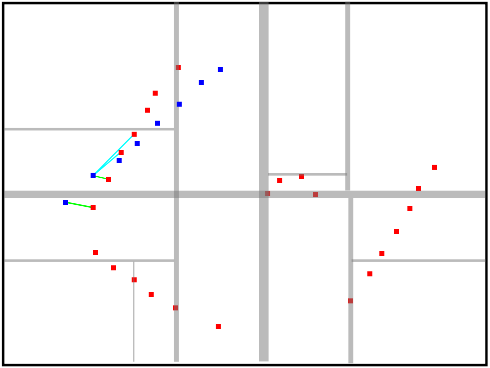
Nearest Neighbor Search in a k-d Tree

Looking for the neighbors of data points:



Nearest Neighbor Search in a *k*-d Tree

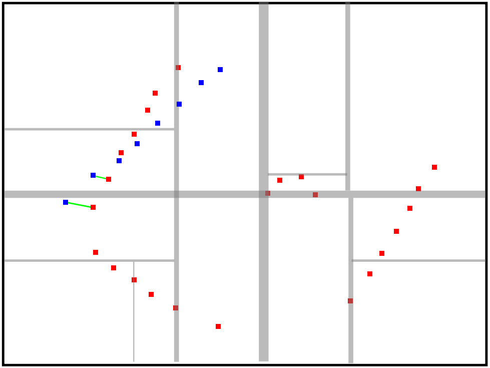
Looking for the neighbors of data points:



ETH Zürich

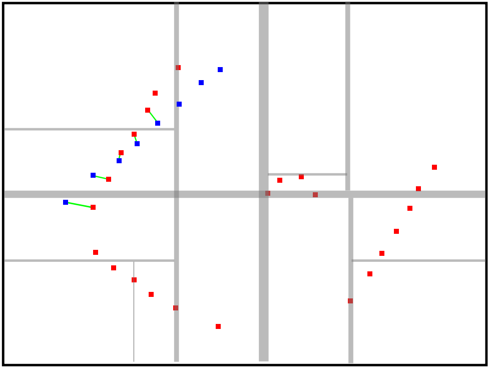
Nearest Neighbor Search in a k-d Tree

Looking for the neighbors of data points:



Nearest Neighbor Search in a k-d Tree

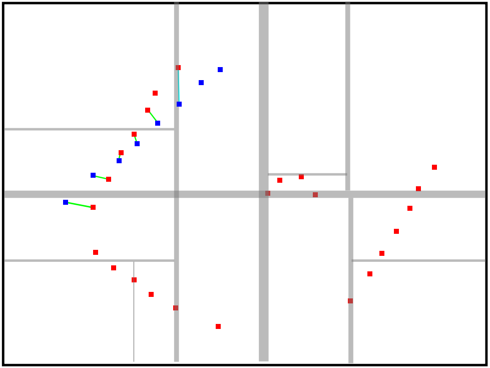
Looking for the neighbors of data points:



ETH Zürich

Nearest Neighbor Search in a k-d Tree

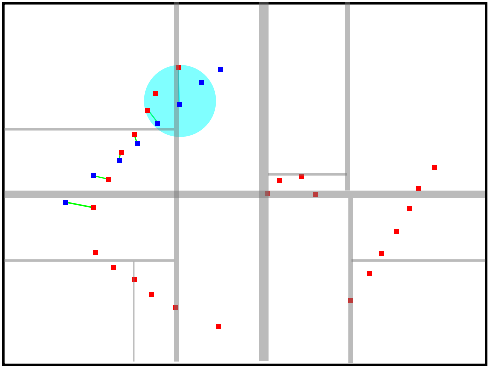
Looking for the neighbors of data points:



ETH zürich

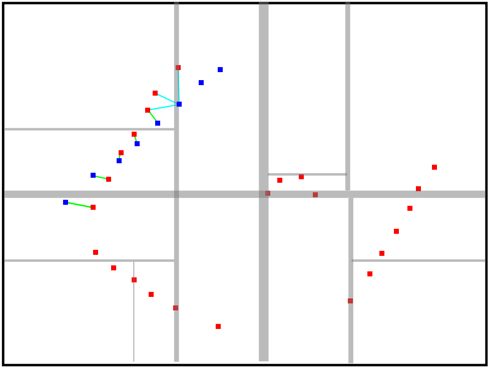
Nearest Neighbor Search in a k-d Tree

Looking for the neighbors of data points:



Nearest Neighbor Search in a k-d Tree

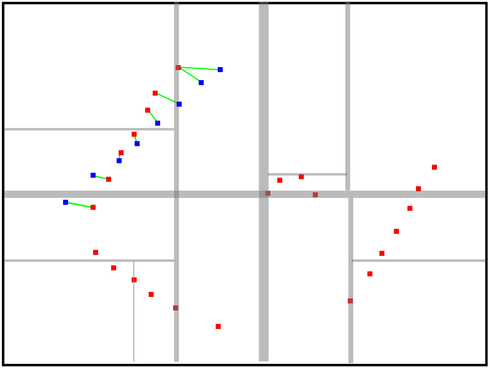
Looking for the neighbors of data points:



ETH zürich

Nearest Neighbor Search in a k-d Tree

Looking for the neighbors of data points:



Nearest Neighbor Search in a k-d Tree

Search:

- go down the tree,
- find closest point in the leaf,
- check when getting back up;

Complexity:

- normally $O(\log(N))$,
- worst case is linear,
- high dimension: worst case.

Building a k-d tree:

- complex: $O(N \log^2(N))$,
- efficient in low dimension (≤ 20),
- worthwhile for repetitive queries.

Summary

Nearest neighbor search:

- linear search,
- space partitioning;

Linear search:

- exhaustive,
- depends on the number of points,
- good for really high dimension or low number of points;

Space partitioning:

- *k*-d tree,
- several variants,
- more complex to build,
- faster for many points in not too high dimension.

Error Minimization

Several error functions:

- point-to-point:

$$E(\mathbf{R},\mathbf{t}) = \sum_i = 1^N \|\mathbf{R}\mathbf{p}_i + \mathbf{t} - \mathbf{m}_i\|^2$$

  where $\mathbf{m}_i$ is the reference point corresponding to data point $\mathbf{p}_i$.
- point-to-plane:

$$E(\mathbf{R},\mathbf{t}) = \sum_i = 1^N \|\mathbf{R}\mathbf{p}_i + \mathbf{t} - \mathbf{m}'_i\|^2$$

  where $\mathbf{m}'_i$ is projection of $\mathbf{p}_i$ on the closest surface
- scaled:

$$E(\mathbf{R},\mathbf{t},\mathbf{S}) = \sum_i = 1^N \|\mathbf{R}\mathbf{S}\mathbf{p}_i + \mathbf{t} - \mathbf{m}_i\|^2$$

Error Minimization

Getting the best transformation:

$$\mathbf{R}, \mathbf{t} = \operatorname*{arg\,min}_{\mathbf{R},\mathbf{t}} E(\mathbf{R}, \mathbf{t})$$

Iterative process:

$$\mathbf{R}_k, \mathbf{t}_k = \operatorname*{arg\,min}_{\mathbf{R},\mathbf{t}} \sum_i = 1^N \|\mathbf{R}\mathbf{p}_i^k + \mathbf{t} - \mathbf{m}_i^k\|^2$$

where $\mathbf{p}_i^k = \mathbf{R}_{k-1}\mathbf{p}_i^{k-1} + \mathbf{t}_{k-1}$.
Solution:

- point-to-point: closed-form solutions with SVD, quaternions...
- point-to-plane: no closed-form solution; linearization or Levenberg-Marquardt.

Summary

ICP:

- matching,
- error minimization,

Matching:

- linear nearest-neighbor search,
- k-d tree NNS;

Error minimization:

- point-to-point: standard with closed-form solution,
- point-to-plane: better but with approximate techniques.