Registration

Joey Wilson, Maani Ghaffari

March 22, 2024

UNIVERSITY OF MICHIGAN

CURLY
Explore the Unknown

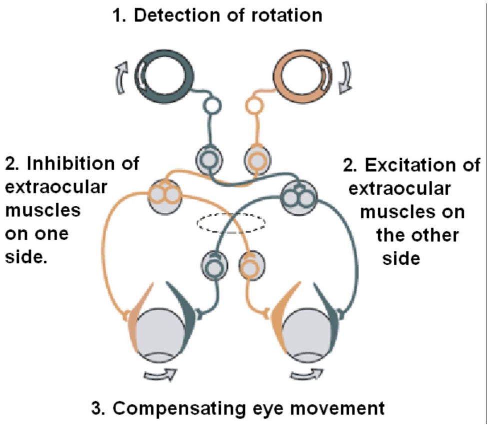
Overview

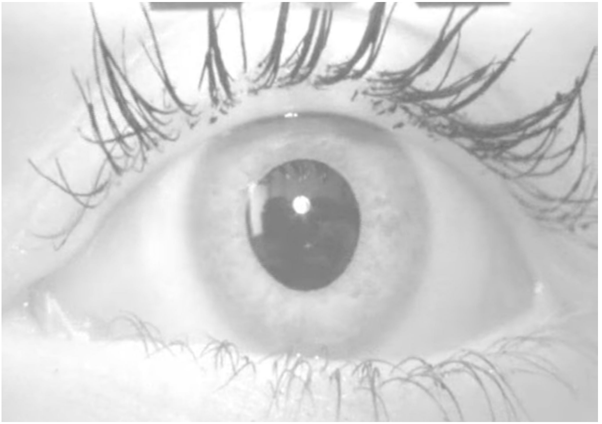
1. Why registration
2. Iterative Closest Point
3. Registration notebook (icp.ipynb - niosus/notebooks · GitHub)

Motivation

Registration in Practice

- Exteroceptive and proprioceptive cues to motion
    - Example: Balancing on one foot with eyes closed, or watching moving object
    - [Vestibulo–ocular reflex - Wikipedia]



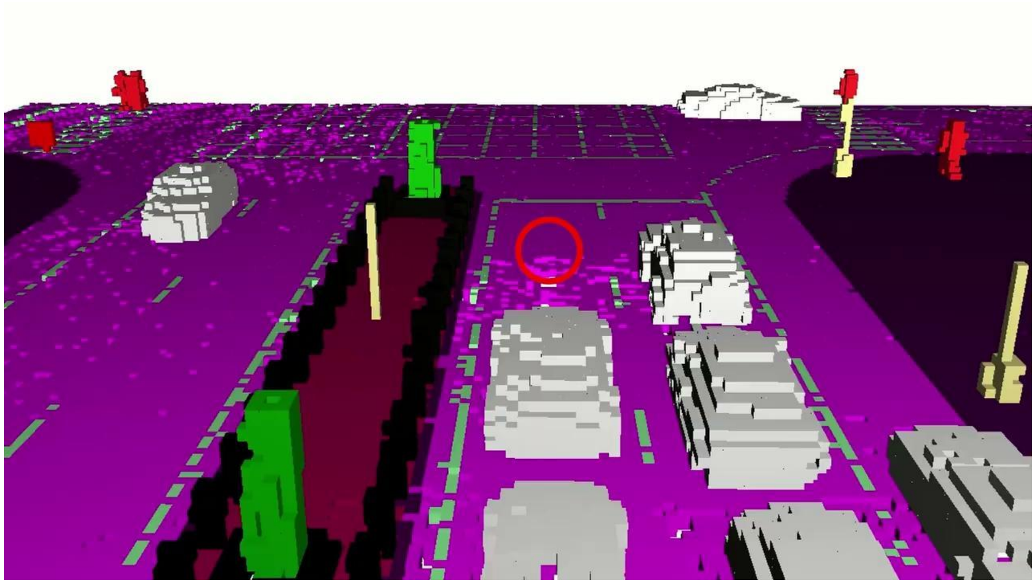


World Models through Registration

- Combine localization cues to aggregate exteroceptive data

Compared to Odometry

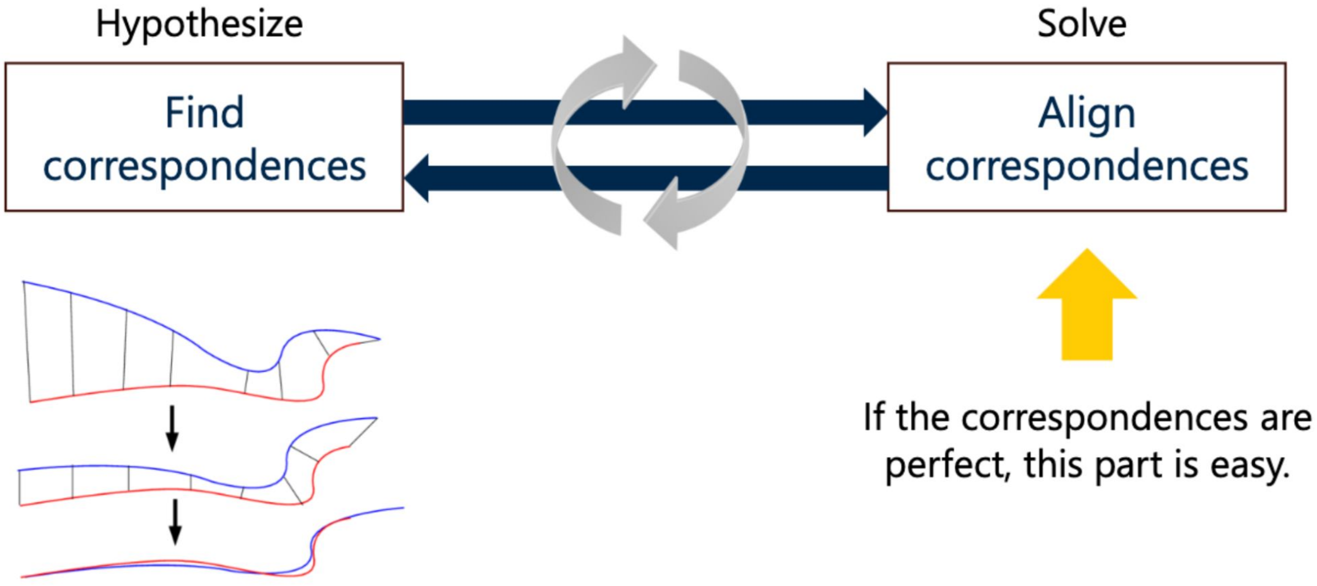
- Proprioception provides clues to static motion, but what about dynamic objects?



J. Wilson et al, "MotionSC: Data Set and Network for Real-Time Semantic Mapping in Dynamic Environments," IEEE Robot. Autom. Letter., vol. 7, no. 3, pp. 8439–8446, 2022.

Overview

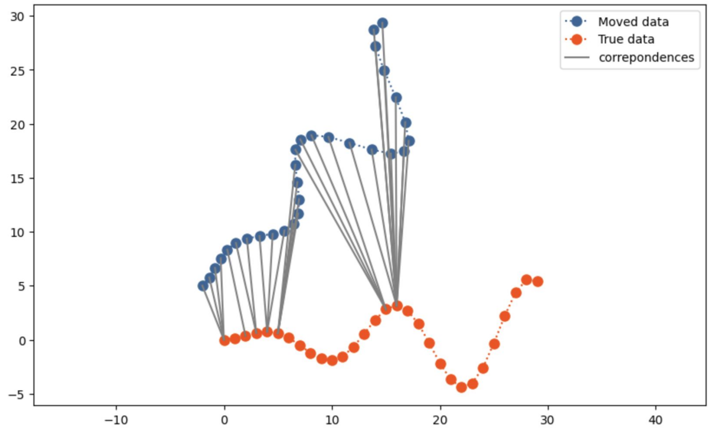
Iterative Closest Point



Associate Target and Source

- Find nearest neighbor between target and source points

$$i_k = \arg\min_k \|x_k^t - T \cdot x_i^s\|$$
$$\mathcal{I} := \{i_k\}$$

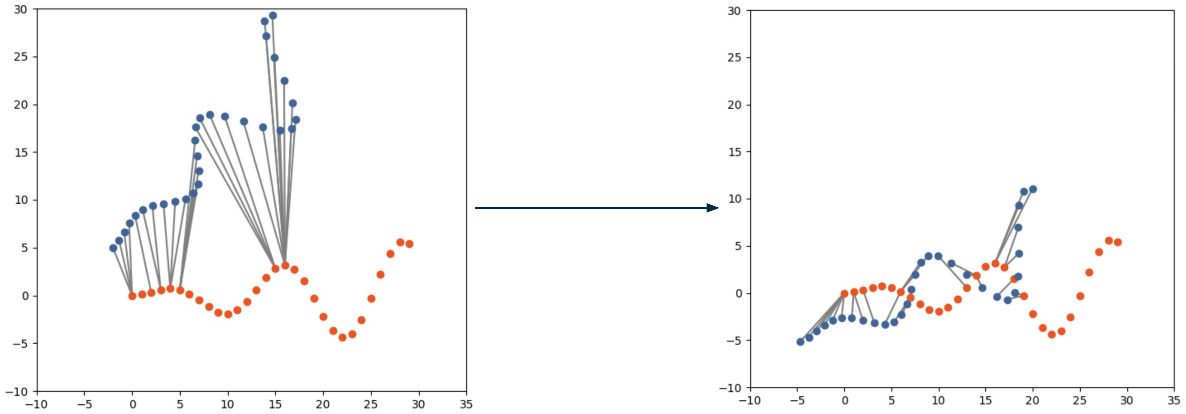


Update Transformation Matrix

- Minimize the residual **of the correspondences**… then repeat!

$$r_k(T) := x_k^t - T \cdot x_k^s$$

$$T^{\mathrm{OPT}} = \underset{T \in \mathrm{SE}(3)}{\arg\min} \sum_{k \in \mathcal{I}} \|r_k(T)\|^2$$



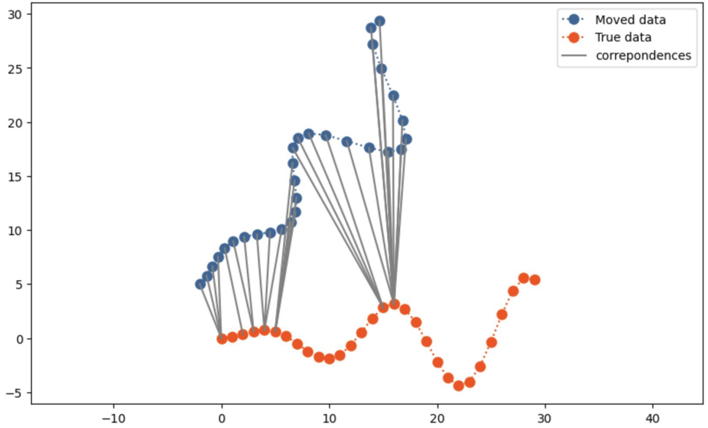
Implementation

Associate Target and Source

- Find nearest neighbor between target and source points

$$i_k = \arg\min_k \| x_k^t - T \cdot x_i^s \|$$

$$\mathcal{I} := \{ i_k \}$$




```python
def get_correspondence_indices(P, Q):
    """For each point in P find closest one in Q."""
    p_size = P.shape[1]
    q_size = Q.shape[1]
    correspondences = []
    for i in range(p_size):
        p_point = P[:, i]
        min_dist = sys.maxsize
        chosen_idx = -1
        for j in range(q_size):
            q_point = Q[:, j]
            dist = np.linalg.norm(q_point - p_point)
            if dist < min_dist:
                min_dist = dist
                chosen_idx = j
        correspondences.append((i, chosen_idx))
    return correspondences
```

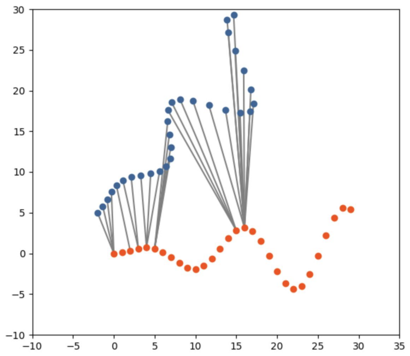
Define the Error

- Minimize the distance between associated points

$$e_n = \mathbf{R}\mathbf{p}_n + \mathbf{t} - \mathbf{q}_n$$

$$\mathbf{E} = \sum_n ||e_n||^2$$

```python
def error(x, p_point, q_point):
    rotation = R(x[2])
    translation = x[0:2]
    prediction = rotation.dot(p_point) + translation
    return prediction - q_point
```



Gauss Newton's Method

- Minimize the distance between associated points

$$e_n = \mathbf{R}\mathbf{p}_n + \mathbf{t} - \mathbf{q}_n$$
$$\mathbf{E} = \sum_n ||e_n||^2$$

$$\mathbf{x} = [x, y, \theta]^T$$

- Second order expansion

$$\mathbf{E}(\mathbf{x} + \Delta\mathbf{x}) \approx \mathbf{E}(\mathbf{x}) + \mathbf{E}'(\mathbf{x})\Delta\mathbf{x} + \frac{1}{2}\mathbf{E}''(\mathbf{x})\Delta\mathbf{x}^2$$

$$0 = \frac{d}{d\Delta\mathbf{x}}\left(\mathbf{E}(\mathbf{x}) + \mathbf{E}'(\mathbf{x})\Delta\mathbf{x} + \frac{1}{2}\mathbf{E}''(\mathbf{x})\Delta\mathbf{x}^2\right) \longrightarrow \mathbf{H}\Delta\mathbf{x} = -\mathbf{E}'(\mathbf{x})$$

$$0 = \mathbf{E}'(\mathbf{x}) + \mathbf{E}''(\mathbf{x})\Delta\mathbf{x}$$

Gauss Newton's Method

- Minimize the distance between associated points

$$e_n = \mathbf{R}\mathbf{p}_n + \mathbf{t} - \mathbf{q}_n$$

$$\mathbf{E} = \sum_n ||e_n||^2$$

$$\mathbf{x} = [x, y, \theta]^T$$

- Second order expansion $\mathbf{H}\Delta\mathbf{x} = -\mathbf{E}'(\mathbf{x})$
    - $\mathbf{E}'(\mathbf{x}) = 2\mathbf{J}(\mathbf{x})\mathbf{e}(\mathbf{x})$
    - $\mathbf{H} \approx 2\mathbf{J}(\mathbf{x})^T\mathbf{J}(\mathbf{x})$

We just need to compute the Jacobian!

Minimization

- Gauss Newton Method (generalization of Newton's method)

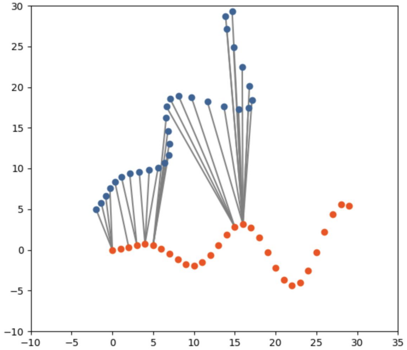
$$e_n = \mathbf{R}\mathbf{p}_n + \mathbf{t} - \mathbf{q}_n$$

$$\mathbf{E} = \sum_n ||e_n||^2$$

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{E}'(\mathbf{x})$$

$$\mathbf{x} = [x, y, \theta]^T$$

- Solve system of equations where **H** is Hessian of **E**

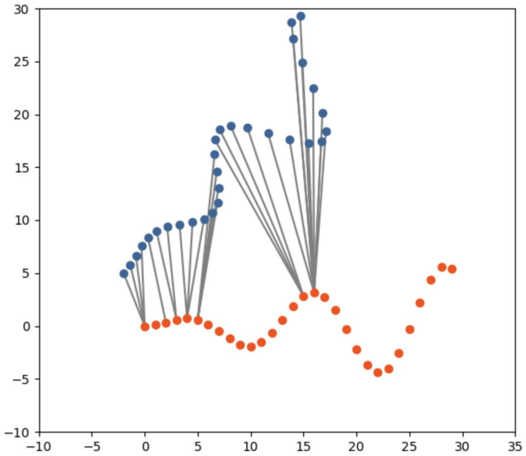


Jacobian

- For point-pair n, Jacobian can be written as:

$$\mathbf{J_n} = \frac{\partial \mathbf{e}_n}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{e}_n^x}{\partial t_x} & \frac{\partial \mathbf{e}_n^x}{\partial t_y} & \frac{\partial \mathbf{e}_n^x}{\partial \theta} \\ \frac{\partial \mathbf{e}_n^y}{\partial t_x} & \frac{\partial \mathbf{e}_n^y}{\partial t_y} & \frac{\partial \mathbf{e}_n^y}{\partial \theta} \end{bmatrix}$$

$$e_n = \mathbf{R}\mathbf{p}_n + \mathbf{t} - \mathbf{q}_n \qquad \mathbf{x} = [x, y, \theta]^T$$

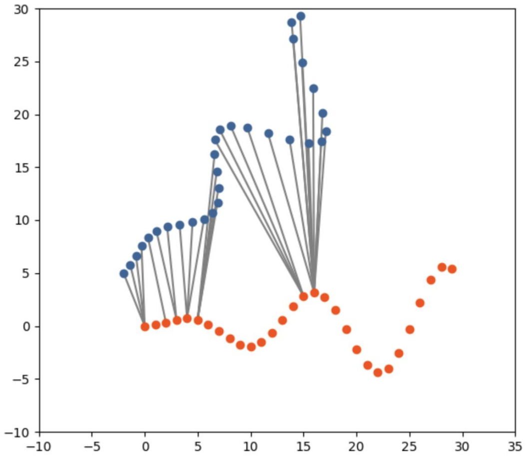


Jacobian of Translation

- For point-pair n, Jacobian can be written as:

$$\mathbf{J_n} = \frac{\partial \mathbf{e}_n}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{e}_n^x}{\partial t_x} & \frac{\partial \mathbf{e}_n^x}{\partial t_y} & \frac{\partial \mathbf{e}_n^x}{\partial \theta} \\ \frac{\partial \mathbf{e}_n^y}{\partial t_x} & \frac{\partial \mathbf{e}_n^y}{\partial t_y} & \frac{\partial \mathbf{e}_n^y}{\partial \theta} \end{bmatrix}$$

$$e_n = \mathbf{R}\mathbf{p}_n + \mathbf{t} - \mathbf{q}_n \qquad \mathbf{x} = [x, y, \theta]^T$$



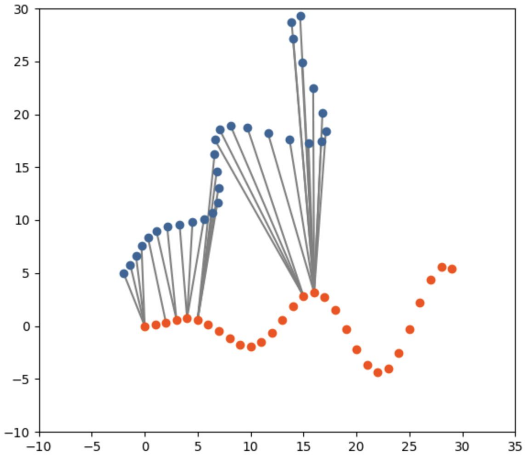
- Only t term -> identity matrix $\begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}$

Jacobian of Rotation

- For point-pair n, Jacobian can be written as:

$$\mathbf{J_n} = \frac{\partial \mathbf{e}_n}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{e}_n^x}{\partial t_x} & \frac{\partial \mathbf{e}_n^x}{\partial t_y} & \frac{\partial \mathbf{e}_n^x}{\partial \theta} \\ \frac{\partial \mathbf{e}_n^y}{\partial t_x} & \frac{\partial \mathbf{e}_n^y}{\partial t_y} & \frac{\partial \mathbf{e}_n^y}{\partial \theta} \end{bmatrix}$$

$$e_n = \mathbf{R}\mathbf{p}_n + \mathbf{t} - \mathbf{q}_n \qquad \mathbf{x} = [x, y, \theta]^T$$



- Derivative of rotation matrix multiplied by p

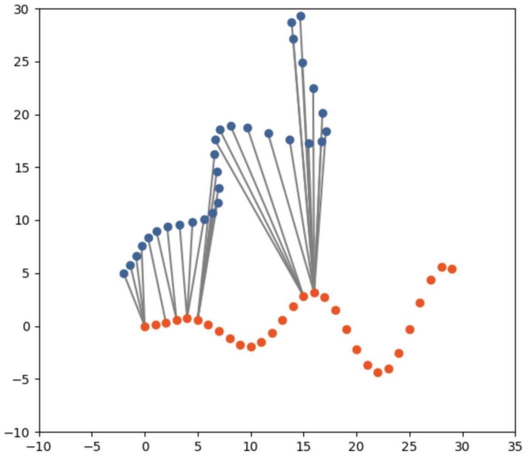
$$\frac{\partial}{\partial \theta}\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} -\sin\theta & -\cos\theta \\ \cos\theta & -\sin\theta \end{bmatrix} \longrightarrow \begin{matrix} -\sin\theta\, p_i^x - \cos\theta\, p_i^y \\ \cos\theta\, p_i^x - \sin\theta\, p_i^y \end{matrix}$$

Jacobian

- For point-pair n, Jacobian can be written as:

$$\mathbf{J_n} = \frac{\partial \mathbf{e}_n}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{e}_n^x}{\partial t_x} & \frac{\partial \mathbf{e}_n^x}{\partial t_y} & \frac{\partial \mathbf{e}_n^x}{\partial \theta} \\ \frac{\partial \mathbf{e}_n^y}{\partial t_x} & \frac{\partial \mathbf{e}_n^y}{\partial t_y} & \frac{\partial \mathbf{e}_n^y}{\partial \theta} \end{bmatrix}$$

$$e_n = \mathbf{R}\mathbf{p}_n + \mathbf{t} - \mathbf{q}_n \qquad \mathbf{x} = [x, y, \theta]^T$$



- Full Jacobian:

$$\begin{bmatrix} 1 & 0 & -\sin\theta\, p_i^x - \cos\theta\, p_i^y \\ 0 & 1 & \cos\theta\, p_i^x - \sin\theta\, p_i^y \end{bmatrix}$$

Jacobian Implementation

- Full Jacobian: $\begin{bmatrix} 1 & 0 & -\sin\theta\, p_i^x - \cos\theta\, p_i^y \\ 0 & 1 & \cos\theta\, p_i^x - \sin\theta\, p_i^y \end{bmatrix}$

```
def jacobian(x, p_point):
    theta = x[2]
    J = np.zeros((2, 3))
    J[0:2, 0:2] = np.identity(2)
    J[0:2, [2]] = dR(theta).dot(p_point)
    return J
```

Optimization

- Compute system of equations
  - Initialize Hessian **H** and gradient **g** to zeroes
  - For each pair of points, increment **g** and **H**

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathbf{g} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{H} \rightarrow \mathbf{H} + \mathbf{J}_n^T \mathbf{J}_n$$

$$\mathbf{g} \rightarrow \mathbf{g} + \mathbf{J}_n^T \mathbf{e}_n$$

```python
def prepare_system(x, P, Q, correspondences
    H = np.zeros((3, 3))
    g = np.zeros((3, 1))
    for i, j in correspondences:
        p_point = P[:, [i]]
        q_point = Q[:, [j]]
        e = error(x, p_point, q_point)
        J = jacobian(x, p_point)
        H += J.T.dot(J)
        g += J.T.dot(e)
    return H, g
```

Solve System of Equations

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{g} \Longrightarrow \Delta\mathbf{x} = -\mathbf{H}^{-1}\mathbf{g}$$

```
H, g, chi = prepare_system(x, P, Q, correspondences, kernel)
dx = np.linalg.lstsq(H, -g, rcond=None)[0]
```

All Together: Iterate Matching and Update

```
def icp_least_squares(P, Q, iterations=30, kernel=lambda distance: 1.0):
    x = np.zeros((3, 1))
    chi_values = []
    x_values = [x.copy()]  # Initial value for transformation.
    P_values = [P.copy()]
    P_copy = P.copy()
    corresp_values = []
    for i in range(iterations):
        rot = R(x[2])
        t = x[0:2]
        correspondences = get_correspondence_indices(P_copy, Q)
        corresp_values.append(correspondences)
        H, g, chi = prepare_system(x, P, Q, correspondences, kernel)
        dx = np.linalg.lstsq(H, -g, rcond=None)[0]
        x += dx
        x[2] = atan2(sin(x[2]), cos(x[2])) # normalize angle
        chi_values.append(chi.item(0))
        x_values.append(x.copy())
        rot = R(x[2])
        t = x[0:2]
        P_copy = rot.dot(P.copy()) + t
        P_values.append(P_copy)
    corresp_values.append(corresp_values[-1])
    return P_values, chi_values, corresp_values
```

Notebook

Want to learn more?

- Check out Open3D: ICP registration - Open3D 0.18.0 documentation
- Try the full notebook: icp.ipynb - niosus/notebooks · GitHub