Homework 7

Fangda Li
li1208@purdue.edu
ECE 661 - Computer Vision

October 31, 2016

1 Methodology

1.1 Iterative Closest Point (ICP)

Given a data point cloud, **Q**, and a model point cloud, **P**, the objective is to register **Q** in the reference frame of **P**. In other words, we want to find the transformation matrix T for **Q** such that **Q** after transformation properly aligns with **P**. Note that every 3D point in the point cloud is a row vector, i.e. **P** is a $N \times 3$ matrix if **P** consists of N points. The steps used in my ICP implementation are as follows:

1. For each 3D point in **Q**, find its nearest neighbor in **P** in terms of Euclidean distance. For computational efficiency, k-D tree is used for nearest neighbor search. Arrange the points in nearest neighbor pairs such that the ith point in **P'** corresponds to the ith point in **Q'**.

2. Next, calculate the centroids, P_c and Q_c , by taking the average of all the points in **P'** and **Q'**, respectively. Subtract the centroids from all the points in **P'** and **Q'** to obtain the "centered" point clouds $\mathbf{M}_P$ and $\mathbf{M}_Q$, respectively.

3. Subsequently, calculate the 3 by 3 correlation matrix C of $\mathbf{M}_P$ and $\mathbf{M}_Q$ using:

$$C = \mathbf{M}_Q^T \mathbf{M}_P. \tag{1}$$

   Decompose C with SVD:

$$C = UsV^T. \tag{2}$$

   As a result, the 3 by 3 rotation matrix R can be obtained with:

$$R = VU^T, \tag{3}$$

   and the corresponding translation vector is:

$$\overrightarrow{t} = P_c - RQ_c. \tag{4}$$

The 4 by 4 transformation matrix T then becomes:

$$T = \begin{pmatrix} R & \overrightarrow{t}^T \\ \overrightarrow{0} & 1 \end{pmatrix}. \tag{5}$$

4. Now, represent the points in $\mathbf{Q}$ in homogeneous coordinates and apply transformation matrix T on $\mathbf{Q}$ to obtain the "aligned" point cloud $\mathbf{Q}_t$.

5. Note that practically multiple iterations of step 1 through 4 might provide better "aligned" point cloud $\mathbf{Q}_t$. More specifically, at the beginning of each iteration, we substitute $\mathbf{Q}$ with $\mathbf{Q}_t$ from the previous iteration.

1.2 Converting Depth Image to Point Cloud

Given a depth image and the camera calibration matrix K, the physical coordinates of the points in point cloud is obtained with the following equation:

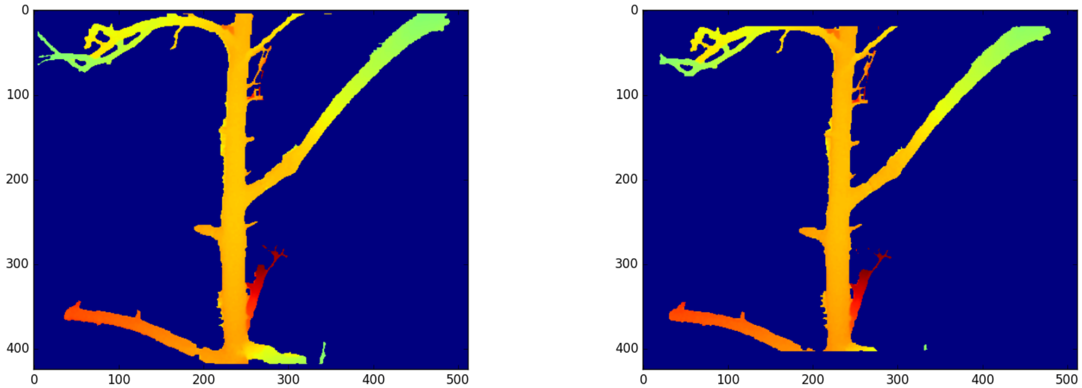
$$\overrightarrow{U} = D(\overrightarrow{u})K^{-1}\overrightarrow{u}, \tag{6}$$

where $\overrightarrow{U}$ is in physical coordinates, $\overrightarrow{u}$ is in pixel coordinates, and $D(\overrightarrow{u})$ is the depth value of $\overrightarrow{u}$. More specifically, the camera calibration matrix of Kinect 2 depth camera is given as:

$$K = \begin{pmatrix} 365 & 0 & 256 \\ 0 & 365 & 212 \\ 0 & 0 & 1 \end{pmatrix}. \tag{7}$$

2 Results and Discussion

2.1 Results



(a) Depth image 1 (b) Depth image 2

Figure 1: Input depth images

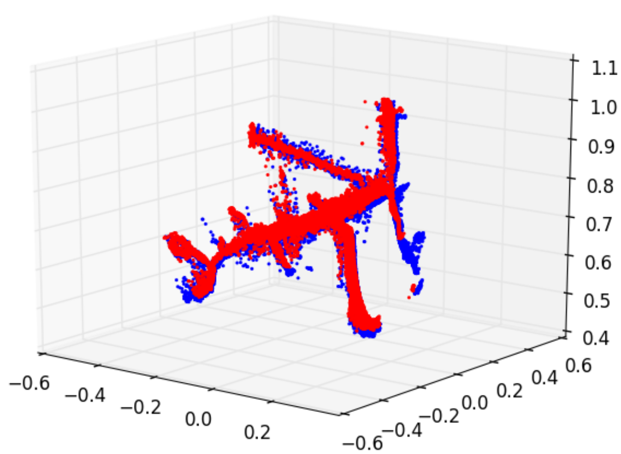


Figure 2: Point clouds before alignment

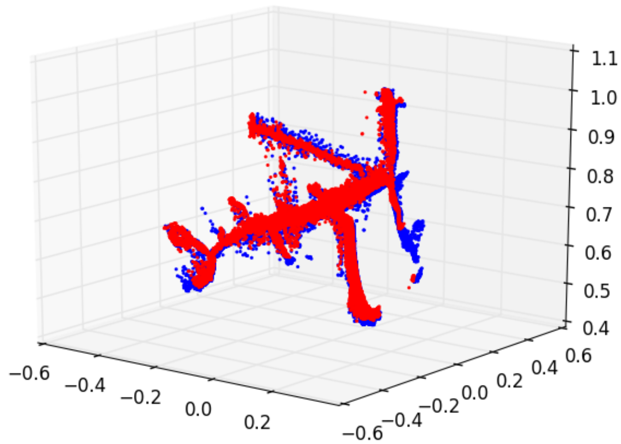


Figure 3: Point clouds after 1 iteration

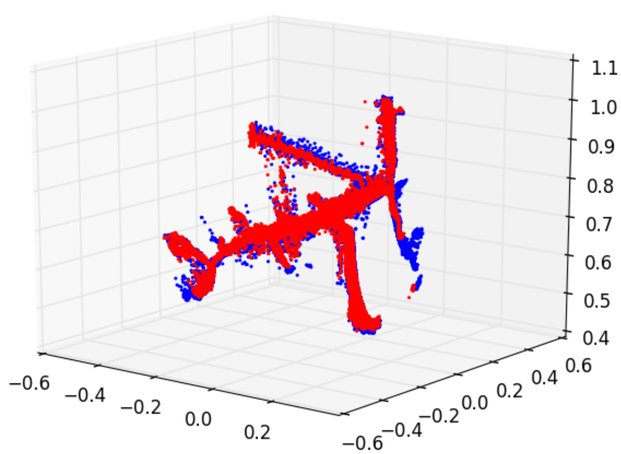


Figure 4: Point clouds after 3 iterations

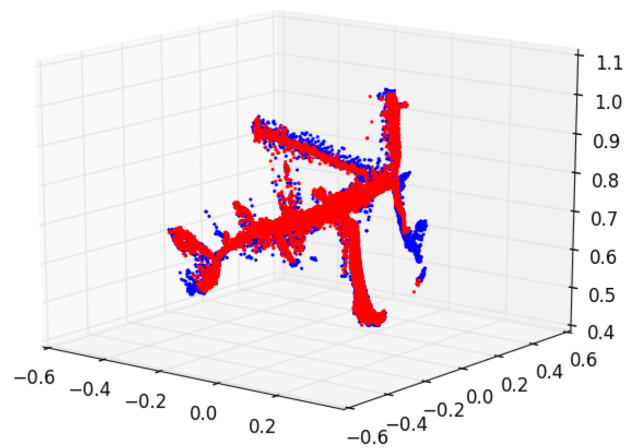


Figure 5: Point clouds after 10 iterations

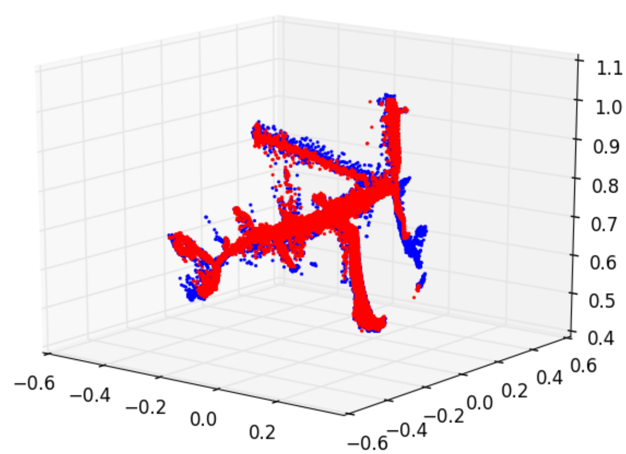


Figure 6: Point clouds after 20 iterations

2.2 Discussion

1. In order for ICP to work, we must start with a good initial estimation of the transformation matrix.

2. The two input point clouds in this experiment are initially already very close to each other, as shown in Figure 2. Consequently, 3 iterations of ICP is shown to be sufficient (Figure 4), compared to the suggested 20 iterations (Figure 6).

3. Using k-D tree for nearest neighbor search provides significant speed up.

3 Source Code

3.1 icp.py

```
#!/usr/bin/python
import numpy as np
from matplotlib import pyplot as plt
from sklearn.neighbors import NearestNeighbors

def icp(data, model, nIteration):
        '''
                Find the transformation matrix that transforms data to model.
                @data: np.ndarray of data point cloud, Mx3
                @model: np.ndarray of model point cloud, Nx3
                @nIteration: int of ICP iterations
                @return: np.ndarray of transformation matrix, 4x4
        '''
        T = np.zeros((4,4))
        Q = data.astype(np.float32)
        P = model.astype(np.float32)
        Qt = Q
        # Initialize the K-D tree with model points
        # FIXME: add threshold delta = 0.1?
        pNN = NearestNeighbors(n_neighbors=1, metric='euclidean').fit(P)
        for i in range(nIteration):
                # Find the NN pairs first
                _, indices = pNN.kneighbors(Qt)
                # Get rid of the extra dimension in the indices
                indices = np.squeeze(indices)
                print 'indices', indices
                # Obtain the NN pairs
                Qp = Qt
                Pp = P[indices,:]
                # Find the centroids
```

```
                Qc = np.sum(Qp, axis=0) / Qp.shape[0]
                Pc = np.sum(Pp, axis=0) / Pp.shape[0]
                print 'Qc', Qc, 'Pc', Pc
                # Subtract the centroid from the point pairs
                MQ = Qp - Qc
                MP = Pp - Pc
                # Compute the correlation matrix
                C = np.dot(MQ.transpose(), MP)
                print 'C', C
                # Decompose C using SVD and compute rotation and translation
                U,s,V = np.linalg.svd(C)
                print U.shape, V.shape, s.shape
                R = np.dot(V.transpose(),U.transpose())
                t = Pc.transpose() - np.dot(R, Qc.transpose())
                # Construct 4x4 transformation matrix T
                T[:3,:3] = R
                T[:3,3] = t
                T[3,3] = 1.
                print 'T', T
                # Transform our original data with transformation matrix
                Qt = np.dot( T, np.transpose( np.hstack(( Qp, np.ones((Qp.shape[0],1)) )
                Qt = Qt[:3,:].transpose()
                print Qt.shape
        return Qt
```

3.2 main.py

```
#!/usr/bin/python
import numpy as np
import cv2
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from icp import icp

def read_depth_image(filename):
        '''
                Read the given .txt file into a float numpy array.
        '''
        with open(filename, 'r+') as f:
                rows = f.readlines()
        rows = [r.split() for r in rows]
        for i,r in enumerate(rows):
                rows[i] = map(float, r)
        return np.array(rows)
```

```python
def depth_to_pc(dimg):
	'''
		Convert depth image into point cloud using the intrinsic camera matrix
		@return: Nx3 np.ndarray of points in the point cloud
	'''
	h,w = dimg.shape
	# Intrinsic camera matrix
	K = np.array([[365., 0., 256.],[0., 365., 212.],[0., 0., 1.]])
	K_inv = np.linalg.inv(K)
	# Preallocate memory for the point cloud
	npts = np.count_nonzero(dimg)
	pc = np.zeros((npts, 3))
	Y,X = np.nonzero(dimg)
	for i in range(npts):
		y,x = Y[i],X[i]
		u = np.array([x, y, 1])
		pc[i,:] = dimg[y,x] * np.dot(K_inv, u)
	return pc

def read_pc_from_file(filename):
	'''
		Read the point cloud from file.
		Each point is stored as a row vector.
	'''
	dimg = read_depth_image(filename)
	print 'Reading depth image...', dimg.shape
	pc = depth_to_pc(dimg)
	print 'Converted to point cloud...', pc.shape
	return pc

def main():
	# Each point is a row vector
	data = read_pc_from_file('images/depthImage1ForHW.txt')
	model = read_pc_from_file('images/depthImage2ForHW.txt')

	# Toy PCs for debugging
	# x = np.linspace(0, 2*np.pi, 100)
	# y = np.zeros(100)
	# z1 = np.sin(x)
	# z2 = np.sin(x) + 0.2
	# data = np.vstack((x,y,z1)).transpose()
	# model = np.vstack((x,y,z2)).transpose()

	# Plot the two orginal point clouds
	fig = plt.figure()
```

```
        ax = fig.gca(projection='3d')
        ax.scatter(data[:,0], data[:,1], data[:,2], c='b', marker='.', edgecolor='none',
        ax.scatter(model[:,0], model[:,1], model[:,2], c='r', marker='.', edgecolor='non
        ax.view_init(elev=16, azim=-52)
        # plt.savefig('./images/before.png')
        plt.show()
        aligned = icp(data, model, 20)
        fig = plt.figure()
        ax = fig.gca(projection='3d')
        ax.scatter(aligned[:,0], aligned[:,1], aligned[:,2], c='b', marker='.', edgecolo
        ax.scatter(model[:,0], model[:,1], model[:,2], c='r', marker='.', edgecolor='non
        ax.view_init(elev=16, azim=-52)
        # plt.savefig('./images/after.png')
        plt.show()

if __name__ == '__main__':
        main()
```