AAE 490 Autonomous Perching Quadcopter

Yucheng Chen

Background and Overview

The research work that I conducted this semester is the continuation of the work in the past summer. In the past summer I developed a RANSAC based algorithm to recognize a cylinder pattern in a 3D point cloud stream and simulated the algorithm in MATLAB. This semester, I primarily implemented the algorithm in Python and integrated the code into Robot Operating System(ROS) and tested the program against the 3D data from Microsoft Kinect.

This report consists of three part. The first part is the explanations of the minor algorithmic modification of the program in the diameter calculations, given that, unlike the simulation, from the data collected by Microsoft Kinect, only the front half of a cylinder can be detected. The second part is basically the manual for using the cylinder detection software, including the required software package and Linux commands. The third part is the future work needed to improve the solution.

A Minor Algorithmic Modification

Problem statement:
The original algorithm to calculate the diameter is straightforward: randomly select 3 points and fit a circle from the them. However, in the read data from the Microsoft Kinect, only a half circle can be detected. Therefore, the 3 points could just be selected from the same half circle, which would cause a relatively greater error in diameter estimation. That is to say: we want the 3 points locate in different half circles.

Solution:
Use an adaptive way to increase the accuracy. In the very beginning, use the 3 points in the same half circle to fit a circle. Starting from the second round, first select 3 points p_1, p_2, p_3 from the half circle, then use the center calculated from the last round to project one point to the other half circle as p_4. Finally, use p_1, p_2, p_4 to estimate a circle. The result is much more accurate.

User Manual of the Cylinder Detection Software

I. Operating system: Ubuntu Linux 16.04
Required software package: 1). Python 2.7.X 2). ROS Kinetic 3). openni (Microsoft Kinect driver)
Sensor/3D scanner: Microsoft Kinect

II. Steps to run the program:
   i. Connect the Microsoft Kinect to a lap top. Open the terminal and go to the work station of ROS packages. In my laptop, it is catkin_ws. Then, type this command:

```
yucheng@yucheng-laptop:~/catkin_ws$ ls
build  devel  src
yucheng@yucheng-laptop:~/catkin_ws$ source devel/setup.bash
```

The command is to set up set ROS environment and connect ROS packages and nodes.

ii. Use openni to launch the Microsoft Kinect by typing the command:

```
yucheng@yucheng-laptop:~/catkin_ws$ roslaunch openni_launch openni.launch
```

Then wait until the following message appears, which means the Microsoft is ready to work.

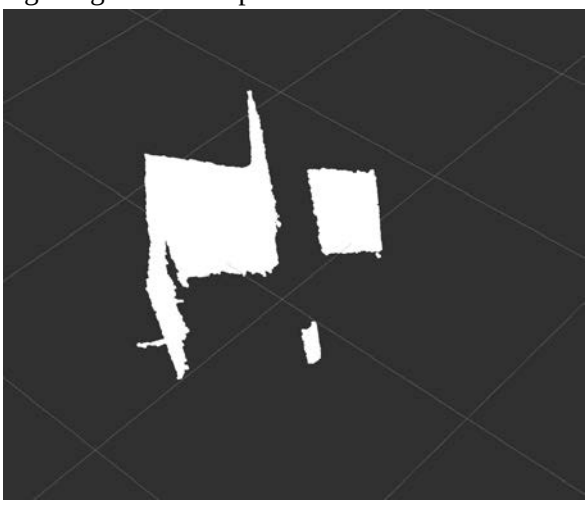
```
[ INFO] [1481929052.103498183]: Number devices connected: 1
[ INFO] [1481929052.103590512]: 1. device on bus 001:10 is a SensorV2 (2ae) from PrimeSense (45
e) with serial id 'A00366912577112A'
[ INFO] [1481929052.104386858]: Searching for device with index = 1
[ INFO] [1481929052.138644504]: Opened 'SensorV2' on bus 1:10 with serial number 'A003669125771
12A'
[ INFO] [1481929052.166376754]: rgb_frame_id = 'camera_rgb_optical_frame'
[ INFO] [1481929052.166677350]: depth_frame_id = 'camera_depth_optical_frame'
[ WARN] [1481929052.170653205]: Camera calibration file /home/yucheng/.ros/camera_info/rgb_A003
66912577112A.yaml not found.
[ WARN] [1481929052.170703002]: Using default parameters for RGB camera calibration.
[ WARN] [1481929052.170753580]: Camera calibration file /home/yucheng/.ros/camera_info/depth_A0
0366912577112A.yaml not found.
[ WARN] [1481929052.170790974]: Using default parameters for IR camera calibration.
```

After the Microsoft Kinect has been launched, we can visualize the point cloud retrieved by the 3D scanner by starting a new terminal source the steup.bash file and type:

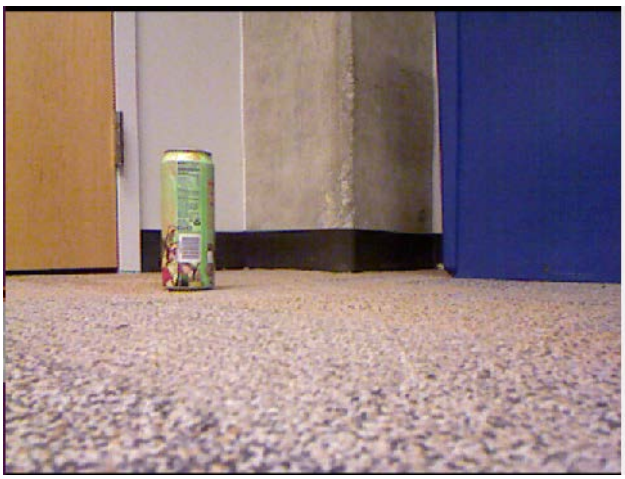
```
yucheng@yucheng-laptop:~/catkin_ws$ rosrun rviz rviz
```

A window will pop out and the 3D point cloud will show up.

The following image is the 3D point cloud:



The following image is the RGB image



iii. Run a ROS node PointXYZ.cpp to convert the raw 3D data, which can't be interpreted directly to the Cartesian coordinate in x, y, z. To run the code, run the command in a new terminal:

```
^Cyucheng@yucheng-laptop:~/catkin_wsrosrun vision_proj PointXYZ
Data sent
Data sent
Data sent
Data sent
Data sent
```

Whenever "Data sent" is printed on the screen, it means a 3D point cloud stream is sent out to been processed.

iv. Run the ROS node cylinderfit.py in a new terminal to receive the point cloud stream sent by PointXYZ.cpp and extract a cylinder from it.

```
yucheng@yucheng-laptop:~/catkin_ws$ rosrun vision_proj cylinderfit.py
Data recieved. Processing Now...
radius  0.0316078962831
center  [-0.14257702  0.71935066  0.01792107]
orientation  [ 0.01829142 -0.01332953  0.99974384]
distance  0.712824451001
Time elapsed  2.98338699341

Data recieved. Processing Now...
radius  0.0356064238807
center  [-0.14136944  0.72498059  0.00192826]
orientation  [-0.00364804  0.00394907 -0.99998555]
distance  0.712806484222
Time elapsed  3.00639390945
```

When the line "Data received. Processing Now…" is printed on the screen, it means a stream of 3D data is received by the node and processing. When is calculation process is completed, five lines of the results will print out.

The "radius" means the radius of the cylinder can in meters. The real diameter of the can is 6.3 cm. Therefore, the calculated radius is accurate.

The "center" means the relative position of the cylinder can to the 3D scanner (i.e. The scanner is located at the origin of the coordinate system).

The "orientation" means the orientation of the axis of the cylinder. The cylinder is vertically placed on the ground, so we expect that the magnitude of z component of the orientation is far greater than the other two components. The printed result is as expected because the magnitude of z component is close to 1 and other two are close to zero.

The "distance" measures the distance the closest point from the cylinder to the origin in meters. The distance value is consistent with $\sqrt{center[0]^2 + center[1]^2} - radius$.

The "Time elapsed" shows the time consumed to finish the computation for the point cloud stream in second. In order to achieve high level of accuracy, the RANSAC algorithm has to iterate 150 times. Initially the sequential program cost nearly 5 minutes. Then I imported the Python multiprocessing module to separate the iterations into 5 threads and conduct the computation concurrently. The number of threads is usually "#cores in the computer + 1". To check the number of cores, in the terminal, type:

```
yucheng@yucheng-laptop:~/catkin_ws$ nproc
4
```

Future Work and Considerations

The work is current in a good stage to the final product. However, there are a couple a limitations given the requirement of the quadcopter: 1). Although the computation time is reduced to ~3 seconds, there are still several frames lag. In the practical use, the application should return the real time result. 2). The algorithm is largely dependent on the number of the points. In this report, the algorithm is tested against a small can, if the program receives a large number of points, the execution time is not predictable. To resolve this issue, I have two possible approaches to improve the performance in the holistic view of the project. 1). After inspecting the executing time of each segments of the code, I found that the calculation of the orientation is almost real time. The most time consuming part is radius calculation. I am considering if we can use another sensor and method to obtain the radius information. 2). Before the 3D processing approach, I tried the machine learning method. I failed because the number of the training samples is not enough. At this point, I attempt to pick up the machine learning method again, but this requires more sophisticated computer vision knowledge because the 3D information should be dug out from the 2D images.

Appendix

Source code:

Cylinderfit.py

```
#!/usr/bin/env python
import rospy
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as la
from sklearn.neighbors import NearestNeighbors
#from mpl_toolkits.mplot3d import Axes3D
from numpy.linalg import inv
from vision_proj.msg import Pts
from multiprocessing import Process, Queue
import time
def callback(msg):
    start = time.time()
    data = np.zeros((3,msg.size))
    for k in range(msg.size):
        data[0,k] = msg.X[k]
        data[1,k] = msg.Z[k]
        data[2,k] = -msg.Y[k]
    print "Data recieved. Processing Now..."
    """
    fig = plt.figure()
    ax = fig.add_subplot(111,projection = '3d')
    x1 = np.squeeze(np.asarray(data[0]))
    y1 = np.squeeze(np.asarray(data[1]))
    z1 = np.squeeze(np.asarray(data[2]))
    ax.scatter(x1,y1,z1)
    plt.show()
    """
    normals = pts2norm(data.T)
    normV,vn1,vn2 = plane_ransac(normals,250,0.01,0.05)
```

```
    mat = np.matrix([vn1,vn2,normV])
    pcl = proj2plane(data,mat)
    qs = []
    tasks = []
    nproc = 5
    for i in range(nproc):
      q = Queue()
      p = Process(target = circle_ransac,args=(pcl,20,0.01,0.05,q))
      qs.append(q)
      tasks.append(p)
    for task in tasks:
      task.start()
    for task in tasks:
      task.join()
    radius = -1
    size = -1
    ctr = [0,0,0]
    for q in qs:
      val = q.get()
      if val[1] > size:
        radius = val[0]
        ctr = val[2]
        size = val[1]

    end = time.time()
    print 'radius ',radius
    print 'center ',ctr
    print 'orientation ',normV
    print 'distance ', msg.dist
    print 'Time elapsed ', end - start
    print ''
#The function uses RANSAC
def circle_ransac(pts,iterNum,thd,thr,q):
    ptNum = pts.shape[1]
    thInlr = round(thr*ptNum)
    optsz = -1
    radius = -1
    ctr = [0,0,0]
    inlier = np.array([])
    for i in range(iterNum):
       dist = np.zeros((1,ptNum))
       perm = np.random.permutation(ptNum)
       sampleIdx = perm[0:3]
       ptSample = pts[:,sampleIdx]
       p1 = np.squeeze(np.asarray(ptSample[:,0]))
       p2 = np.squeeze(np.asarray(ptSample[:,1]))
       p3 = np.squeeze(np.asarray(ptSample[:,2]))
       if iscollinear(p1,p2,p3):
         print 'collinear'
         continue
       center,r,v1n,v2n = circlefit3d(p1,p2,p3)
       un = np.cross(v1n,v2n)
       un = un/np.sqrt(np.dot(un,un))

       for k in range(ptNum):
          vector = np.squeeze(np.asarray(pts[:,k]))
```

```
            proj = np.cross(un,(vector-center.T))
            dist[:,k] = np.sqrt(np.dot(proj,proj))
        inlridx = np.nonzero(abs(dist-r)<thd)
        inlrsz = np.size(inlridx)
        if inlrsz < thInlr:
          continue
        if (inlrsz > optsz):
          optsz = inlrsz
          radius = r
          ctr = center
          inlier = pts[:,inlridx]
    q.put([radius,optsz,ctr])

def proj2plane(pts,mat):
    sol = inv(mat)*pts
    sol[2,:] = 0
    return mat*sol
def circlefit3d(p1,p2,p3):
    v1 = p2 - p1
    v2 = p3 - p1
    l1 = np.sqrt(np.dot(v1,v1))
    l2 = np.sqrt(np.dot(v2,v2))
    v1n = v1/l1
    v2n = v2/l2
    nv = np.cross(v1n,v2n)
    v2nb = np.cross(v1n,nv)
    v2n = v2nb/np.sqrt(np.dot(v2nb,v2nb))
    cor1 = np.dot(v2,v1n)
    cor2 = np.dot(v2,v2n)
    scale1 = 0.5*l1
    scale2 = 0.5*(cor2 - (l1-cor1)*cor1/cor2)
    center = p1+scale1*v1n+scale2*v2n
    rv = p1-center
    r = np.sqrt(np.dot(rv,rv))
    return (center,r,v1n,v2n)
def iscollinear(p1,p2,p3):
    v1 = p2-p1
    uv = v1/np.sqrt(np.dot(v1,v1))
    t = (p3[0]-p1[0])/uv[0]
    pt = p1+t*uv
    erv = p3-pt
    err = np.sqrt(np.dot(erv,erv))
    if err < 0.00001:
      return True
    else:
      return False
def pts2norm(pts):
    nbrs = NearestNeighbors(n_neighbors = 100, algorithm = 'ball_tree').fit(pts)
    dist,indices = nbrs.kneighbors(pts)
    m = pts.shape[0]
    n = np.zeros((3,m))
    for i in range(m):
        kn = pts[indices[i,0:100],:]
        #print kn
        kn_col = np.sum(kn,axis=0)/100.0
        #print kn_col
```

```
        P1 = (kn-np.tile(kn_col,(100,1))).T
        P2 = (kn-np.tile(kn_col,(100,1)))
        P = np.dot(P1,P2)
        #print P
        D,V = la.eig(P)
        ind = np.argmin(D)
        n[:,i] = V[:,ind]
    return n
def plane_ransac(pts,iterNum,thd,thr):
    ptNum = pts.shape[1]
    thInlr = round(thr*ptNum)
    inlrsz = np.zeros((1,iterNum))
    uns = np.zeros((3,iterNum))
    v1s = np.zeros((3,iterNum))
    v3s = np.zeros((3,iterNum))
    for i in range(iterNum):
        perm = np.random.permutation(ptNum)
        sampleIdx = perm[0:2]
        ptSample = pts[:,sampleIdx]
        p1 = ptSample[:,0]
        p2 = ptSample[:,1]
        p3 = -p1
        if iscollinear(p1,p2,p3):
          print 'collinear'
          continue
        un,v1,v3 = fitplane(p1,p2,p3)
        arrange = np.tile(p1,(ptNum,1)).T
        dist = np.dot(un,(pts-arrange))
        inlieridx = np.nonzero(abs(dist)<thd)
        if np.size(inlieridx) < thInlr:
          continue
        else:
          inlrsz[:,i] = np.size(inlieridx)
          uns[:,i] = un
          v1s[:,i] = v1
          v3s[:,i] = v3
    candid = np.argmax(inlrsz)
    normV = uns[:,candid]
    vn1 = v1s[:,candid]
    vn3 = v3s[:,candid]
    return (normV,vn1,vn3)
def fitplane(p1,p2,p3):
    v1 = p2-p1
    v1 = v1/np.sqrt(np.dot(v1,v1))
    v2 = p3-p1
    v2 = v2/np.sqrt(np.dot(v2,v2))
    v3 = np.cross(v1,v2)
    v3 = v3/np.sqrt(np.dot(v3,v3))
    return (v3,v1,v2)

def main():
   rospy.init_node('CylinderFitter',anonymous = True)
   rospy.Subscriber('chatter',Pts,callback)
   rospy.spin()
```

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

PointXYZ.cpp

```
#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>
#include <vision_proj/Pts.h>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/Float32MultiArray.h"
#include <math.h>
ros::Publisher pub;
typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
void callback(const PointCloud::ConstPtr& msg)
{

int j = 0;
int step = 5;
double dist = 0;
vision_proj::Pts p;
pcl::PointXYZ pt;
for (int i = 0; i < 307200;i=i+step) {
   pt = msg->points[i];
   if (!isnan(pt.x) && !isnan(pt.y) && !isnan(pt.z)){
      double length = 0;
      if ((length = sqrt(pt.x*pt.x+pt.y*pt.y+pt.z*pt.z)) < 1) {
        dist = dist + length;
        p.X[j] = pt.x;
        p.Y[j] = pt.y;
        p.Z[j] = pt.z;
        j = j + 1;
      }
   }
}
p.size = j;
p.dist = dist/p.size;
pub.publish(p);
printf("Data sent\n");

}
int main(int argc, char** argv)
{
  ros::init(argc, argv, "sub_pcl");
  ros::NodeHandle nh;
  while (ros::ok()) {
    pub = nh.advertise<vision_proj::Pts>("chatter", 1000);
    ros::Subscriber sub = nh.subscribe<PointCloud>("/camera/depth/points", 1, callback);
    ros::spin();
  }
  //ros::spin();

}
```