Robotic Sensing, Manipulation and Interaction
Coursework 3 - Code Report

Alejandro Halpern - Andras Nagy - Tim Andersson - Xingjian Lu

April 14, 2021

Question 1

1.1)

The simplest way of speeding up the cylinder segmentation is to reduce the number of points in the pointcloud. To do this, 2 methods where implemented using the PCL library.

The first method was down-sampling the point cloud by generating a voxel grid on top of it, and then combining all the points inside each voxel into 1, with the colour and position set to be the average of the previous points in the voxel. This effectively lowers the number of points, at the cost of a decreased point-cloud density/resolution, which might lead to higher inaccuracies in the cylinder pose results.

The second method was the removal of outliers, that is, points that due to measurement errors are significantly separated from its neighbours. These erroneous points can cause innacuracies down the line, in, for example, the estimation of surface normals. Thus, removing them will increase the accuracy of the solution and decrease the number of points in the cloud. The only downside to this method is that it introduces an extra computational cost that may not be justifiable if the point cloud has few or no outliers in the first place.

Both methods were implemented using PCL's VoxelGrid and StatisticalOutlierRemoval classes as follows:

```
/** \brief Given the pointcloud, apply VoxelGrid and StatisticalOutlierRemoval
  * to the pointcloud
  *
  * @param cloud - Pointcloud to be filtered
  * @param leafx, leafy, leafz - size of leaf on x, y, z direction
  * @param meanK - meanK of StatisticalOutlierRemoval
  * @param stdThresh - standard deviation threshold
  *                    of StatisticalOutlierRemoval
  */
void filterEnvironment (pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud,
                        float leafx, float leafy, float leafz,
                        int meanK, double stdThresh)
{
  // initialise a voxel_grid and apply filter to the cloud
```

```
    pcl::VoxelGrid<pcl::PointXYZRGB> voxel_grid;
    voxel_grid.setInputCloud (cloud);
    voxel_grid.setLeafSize(leafx,leafy,leafz);
    voxel_grid.filter(*cloud);

    // initialise a StatisticalOutlierRemoval and apply filter to the cloud
    pcl::StatisticalOutlierRemoval<pcl::PointXYZRGB> sor;
    sor.setInputCloud(cloud);
    sor.setMeanK(meanK);
    sor.setStddevMulThresh(stdThresh);
    sor.filter(*cloud);
}
```

1.2)

According the the coursework description, for this question, we are given the rough area that the object lies on, which is (-0.05, 0.16, 0.92) with 0.25m error in every direction. We assume "every direction" to be x, y, and z direction which would give us a cubic area instead of a spherical one. In order to create such filter, `fastFilterEnvironment()` is implemented as follows:

```
void fastFilterEnvironment (pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud,
                            double xl, double xu, double yl, double yu,
                            double zl, double zu)
{
    pcl::PassThrough<pcl::PointXYZRGB> pass;
    pass.setInputCloud (cloud);

    pass.setFilterFieldName ("x");
    // min and max values in x axis to keep
    pass.setFilterLimits (xl, xu);
    pass.filter (*cloud);

    pass.setFilterFieldName ("y");
    // min and max values in y axis to keep
    pass.setFilterLimits (yl, yu);
    pass.filter (*cloud);

    pass.setFilterFieldName ("z");
    // min and max values in z axis to keep
    pass.setFilterLimits (zl, zu);
    pass.filter (*cloud);
}
```

Where a PassThrough filter is used to set the limit on x, y, and z direction to be within 0.25m of the coordinate provided.

1.3)

In order to publish the cylinder pose, three changes are made to the original code.

First, a new public variable `ros::Publisher c_pose_pub` is created and initialised in the constructor of class `CylinderSegment` with the following:

```
c_pose_pub = nh.advertise<geometry_msgs::PoseStamped>("\cylinder_pose", 1);
```

Second, a new function `publishCylinderPose()` is created to initialise a PoseStamped message from a Pose message and publish it.

```
void publishCylinderPose (geometry_msgs::Pose cylinder_pose);
```

This function is called in function `CloudCB()` whenever a point cloud message is received and a cylinder is detected from the point cloud.

Third, the function `addCylinder()` is separated into two functions. The part of `addCylinder()` that calculates the position and orientation of the detected cylinder is taken out of the function and converted into a new function `getCylinderPose()` and the remaining part of it is left unchanged. The function declaration of both functions are shown below:

```
geometry_msgs::Pose getCylinderPose();
void addCylinder (geometry_msgs::Pose cylinder_pose)
```

The reason for dividing `addCylinder()` into two functions is that we want the cylinder pose to be calculated so that it can be published while we don't want a cylinder object to be added to the planning scene every time we get input from camera. Because the input point cloud message does not change over time and the filtering and segmentation algorithms are static, the pose and size of the cylinder detected doesn't change unless a new filter is enables/disabled. Hence, two boolean variables `replace` and `points_not_found` are setup and will be updated with key presses (see section Handling key press) so that the cylinder is only added when it is first detected and whenever a filter is enabled/disabled (key press detected) the cylinder is replaced with one with updated pose and size.

helper functions and general setup

Handling key press

This question requires certain filtering function in `cylinder_segment.cpp` to be triggered when key 'f' or 'p' is pressed. To achieve this, in `cw3.cpp`, the key presses are detected in function `make_choice()` and a `std_msgs::Char` carrying the character on the key is published to topic `/choices` by a publisher `choice_pub` in function `publishChar()`. In `cylinder_segment.cpp`, a subscriber `choice_sub` to topic `/choices` is setup and a callback function `choiceCallback()` is triggered whenever a key press is published from `cw3.cpp`. Then, depending on the key pressed, `choiceCallback()` toggles the boolean variables `filter` and `fast_filter` that control different filtering functions. Also, a boolean variables `replace` is set to true when either 'f' or 'p' is pressed in order to trigger the replacement of cylinder in the planning scene (see section 1.3).

Publishing filtered point cloud

This is not part of question description and it is only used for visualisation purposes. In order to publish point cloud and visuaise it in rviz, a publisher and a helper function is setup. The publisher is initialised as follows:

```
pc2_pub = nh.advertise<sensor_msgs::PointCloud2>("/pc2", 1);
```

The function is implemented as follows:

```
void publishPointCloud2(pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud,
                        const sensor_msgs::PointCloud2ConstPtr& input)
{
    sensor_msgs::PointCloud2 msg = *input;
    pcl::toROSMsg(*cloud, msg);
    pc2_pub.publish(msg);
}
```

This function can be placed anywhere in function `CloudCB()` with `cloud` being the filtered pointcloud and `input` being the input of `CloudCB()`.

Publishing cylinder size

Along with the pose of the cylinder, the size of the cylinder is also required for grasping tasks in q2 and q3. In order to do this, a publisher of `std_msgs::Float64MultiArray` is setup and initialised with the following:

```
c_size_pub = nh.advertise<std_msgs::Float64MultiArray>("/cylinder_size", 1);
```

A helper function is implemented to extract information from `cylinder_params` and publish this information. The format of the data is an array of length two with the first element being radius of the cylinder and the second element being the height of the cylinder.

```
void publishCylinderSize ()
{
    std_msgs::Float64MultiArray c_size;
    c_size.data.resize(2);
    c_size.data[0] = cylinder_params->radius;
    c_size.data[1] = cylinder_params->height;
    c_size_pub.publish(c_size);
}
```

Timer

A timer is setup to record the time used for filtering and segmentation in `CloudCB()` using the following:

```
pcl::StopWatch stopwatch;
// reset the timer at the beginning the segment you want to time
stopwatch.reset();
// get time at the end the segment you want to time
stopwatch.getTime();
```

Question 2

In order to solve the following tasks, the pose of the cylinder and the dimensions of the cylinder are needed. These values are determined and published as described previously, only two subscribers needed to be written to get these values. cyl_pos_sub and cyl_dim subscribers were written with $cylSubCb$() and $cylDimSubCb$() callback functions for getting the cylinder pose and dimensions respectively.

Next, as the pose is published with respect to the camera frame; therefore, it had to be transformed to the world frame (panda_link0). $convertCToW$() was created to convert from the camera frame to the world frame with the following equation.

$$^0T_{cyl} = {}^0T_c \, {}^cT_{cyl} \tag{0.1}$$

Where $^0T_{cyl}$ is the homogeneous transformation from the cylinder frame to the world frame, ^0T_c transformation from the camera frame to the world frame and $^cT_{cyl}$ is the homogeneous transformation between the cylinder frame and the camera frame.

To solve this task, 3 helper functions were used which were responsible for type conversions. Namely: $transformToHom$() converts from StampedTransform to homogeneous matrix, $transPoseToHom$() make a conversion from PoseStamped to homogeneous matrix; finally, $transHomToPose$() converts back the homogeneous matrix to geometry_msgs::Pose

2.1)

In order to create 3 bumper sensors at the centre of each table, $cw3q2AddBumper$() and $cw3q2BumperTouched$() functions were implemented.

First, the purpose of $cw3q2AddBumper$() is presented. This function is called at the beginning of test_cw3.cpp and initialise the positions of the bumpers with the positions of the centre point of each table and store them in a class vector variable. The table positions are get with the help of the $planning\_scene\_interface.getObjectPoses(table\_names)$ where table_names is a vector of strings with the name of the tables. As a result of this, the code can handles any modifications and changes accordingly the positions of the bumpers if the tables are moved to an other position. The bumper sensor values are stored in a class vector as std_msgs::Bool values. False value refers to no object is detected at the centre of the table; while, true means object is located on the table. Initially, the bumpers are initialised with false values, but after this, $cw3q2BumperTouched$() function is called where the bumpers are updated with the real values.

This function is called simultaneously in the main while loop of the test_cw3.cpp file; it checks whether the bottom centre points of the cylinder object and the cubic object are in an epsilon range of the position of the bumpers and updates the bumper Boolean values accordingly. Moreover, each bumper value is published separately as a std_msgs::Bool message on separate topics, namely: bumper/1, bumper/2 and bumper/3.

2.2)

The explanation of this question can be found in Question 3.

Question 3

Task 3 is carried out using the MoveIt library. The pose and dimension of the cylinder is taken from the published values from question 1. The pose of the cuboid object and the tables are taken from the simulation using the PlanningSceneInterface class instance created in the test node.

The PlanningSceneInterface and MoveGroupInterface class instances are used extensively and throughout almost every part of this question. Before any of this happens however, the end effector link of the MoveGroupInterface is set in the test node.

As the PlanningSceneInterface and the MoveGroupInterface class instances are used so extensively throughout the member functions of the CW3 class, it is necessary to be conservative with memory used. Therefore, these class instances are always passed by reference rather than passed by value into the CW3 class methods, otherwise there would be a huge waste of memory on the stack at runtime and this would most likely affect performance.

An options menu is periodically printed to the console using the ***ROS_INFO_STREAM*** functionality from ROS and a ***std*** : : ***string*** created in the ***get_options*** function.

The character inputting into the console at runtime is handled using a switch case statement. Dependant on the input, different functions are executed and in the case of invalid (default) choice, a warning is printed to the console using ***ROS_INFO_STREAM***. If there is no choice or null char nothing happens and the switch case statement is broken.

The rest of the functions simply execute picking and placing the relevant objects.

```
////////////////////////////////////////////////////////////////////////////////
void
CW3::cw3q3(int chah, moveit::planning_interface::PlanningSceneInterface& p_s_i,
                moveit::planning_interface::MoveGroupInterface& m_g_i)
{
        if (this->bumpers.at((chah - '0' - 1)).data == true)
        {
                this->move_object(chah, p_s_i, m_g_i);
        }
        this->pick_up_cylinder(p_s_i, m_g_i, "cylinder");
        this->place_cylinder(chah, p_s_i, m_g_i);
        ros::Duration(2).sleep();
        return;
}
```

chah is the input 'int' (or char) from the test node. This function uses the ASCII value of the input char (in this case it is definitely 1, 2 or 3, as this functionality is handled in the ***make_choice*** function) to determine whether the input is 1, 2 or 3.

The rest of the functions for this task simply fill out the functionality of picking and placing objects. The workflow is always to move to a position above the object to be picked and placed first. Then the manipulator executes the pick using the MoveIt library. The manipulator then moves back to the pose it was in before the pick. Then the manipulator moves to a pose above the desired place location. The manipulator then places the object and retreats. This operation is repeatable.

All repeated code is placed in helper functions and all complex and hard to read code is also placed into helper functions. No lines exceed 80 characters and everything is clearly named. No values are hard coded and class variables are used in every instance possible. No variables are reinitialised where possible.

It is **very important** to note that there is some functionality lacking in the final submission. When using the hard coded values for the original location of the cylinder, all the functionality worked fine. However, after implementing the integration of question 1 and questions 2 and 3, the limitations are as follows:

1, and then 3 can be pressed, and vice versa. The cylinder can be moved from floor to free table, and free table to free table indefinitely. However, if 2 is pressed, the cuboid will be moved from table 2 to a free table without trouble. But, when it comes to moving the cylinder from *anywhere* to table 2, path planning cannot be completed and collisions are somehow found. Great lengths were gone to to try and fix this and work out where the bug came from after integrating question 1, but eventually no solution was found.

References

[1] F. Dellaert and M. Kaess, "Factor Graphs for Robot Perception", Foundations and Trends in Robotics, vol. 6, no. 1-2, pp. 1-139, 2017. Available: 10.1561/2300000043 [Accessed 21 March 2021].