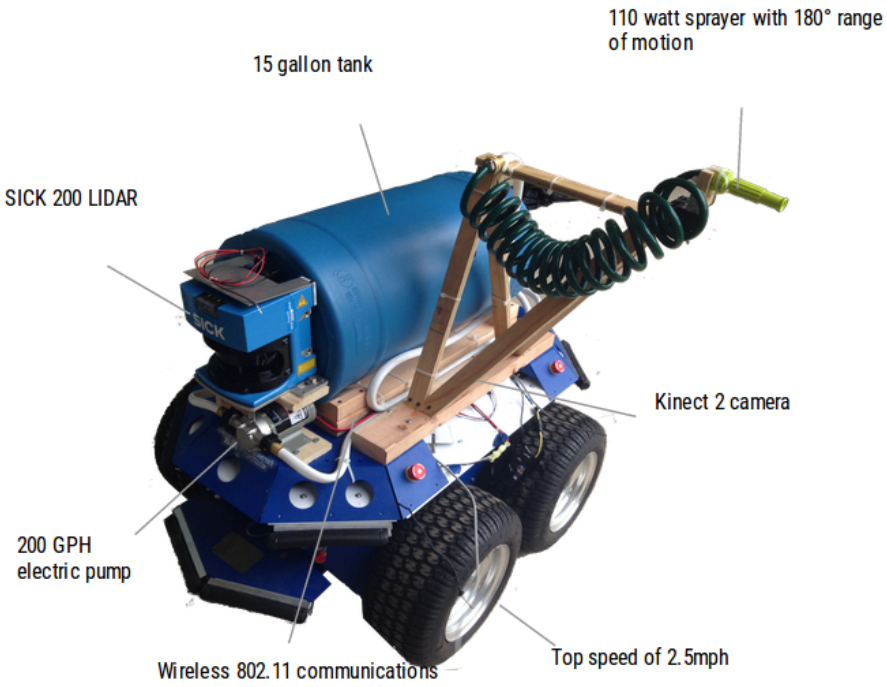
ECE 4560 Final Project Write Up

Andrey Kurenkov, Pavel Komarov, Troy O'Neal



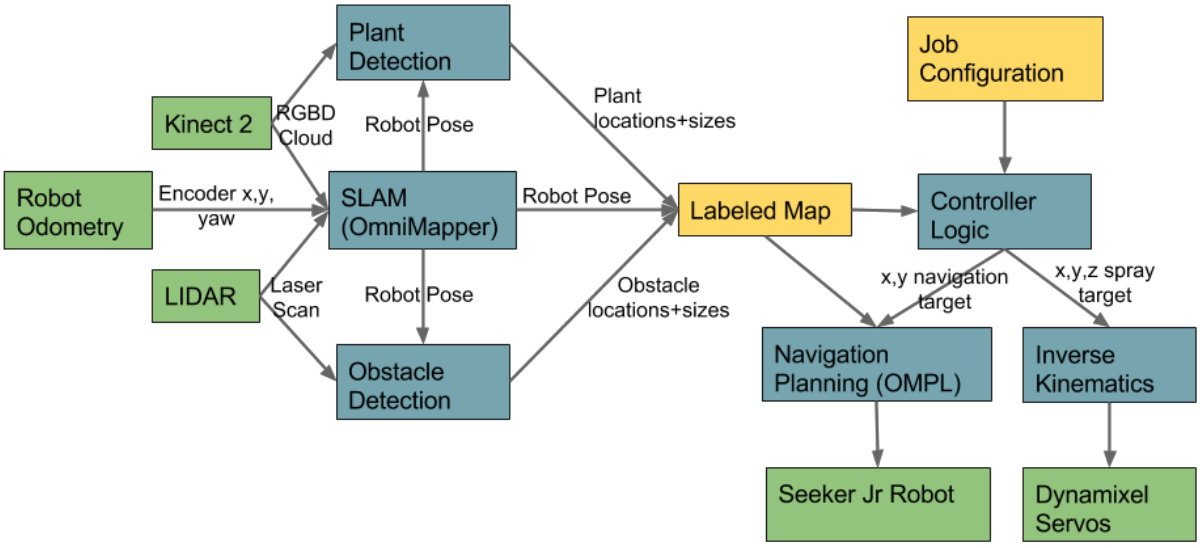
Problem Statement

The goal of the larger project was to create a robot capable of navigating through a field of crops and dynamically spraying targets.The effort was multifaceted and complicated, so we separated out a few parts for the context of this class project:

1. SLAM (localization, mapping, and plant detection).
2. Design of a robotic manipulator to direct payload fluid.
3. Path planning to a goal location and obstacle avoidance.

Solution Strategy

Like the problem, the solution is complicated. On the next page is a diagram of our larger solution pipeline, which looks less like a pipe than a graph. Note that not all systems are fully functional, but all have been extensively researched and labored upon. Following the graphic are descriptions of subsystems related to this class.



SLAM

The entire component of the graph leading up to the Labeled Map structure made up the SLAM component of the this project. The problem in more detail was to combine multiple sensors (Kinect 2 so as to create a high resolution map of the farm), use those sensors to locate plants to spray as well as obstacles to avoid, and to keep track of the location of the robot so as to navigate it for spraying and obstacle avoidance.

In solving the problem, several ROS-compatible SLAM packages were investigated and evaluated for our purpose. All the listed ROS packages were found to be unsuitable for our purpose, because they only worked with monocular data or several sensors. However, the original proposal of using OmniMapper was still promising, so the next step was to explore using that package.

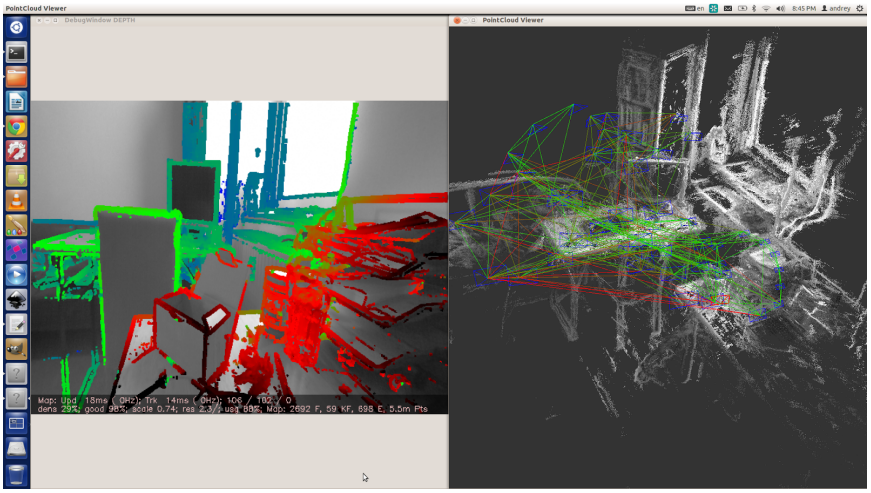
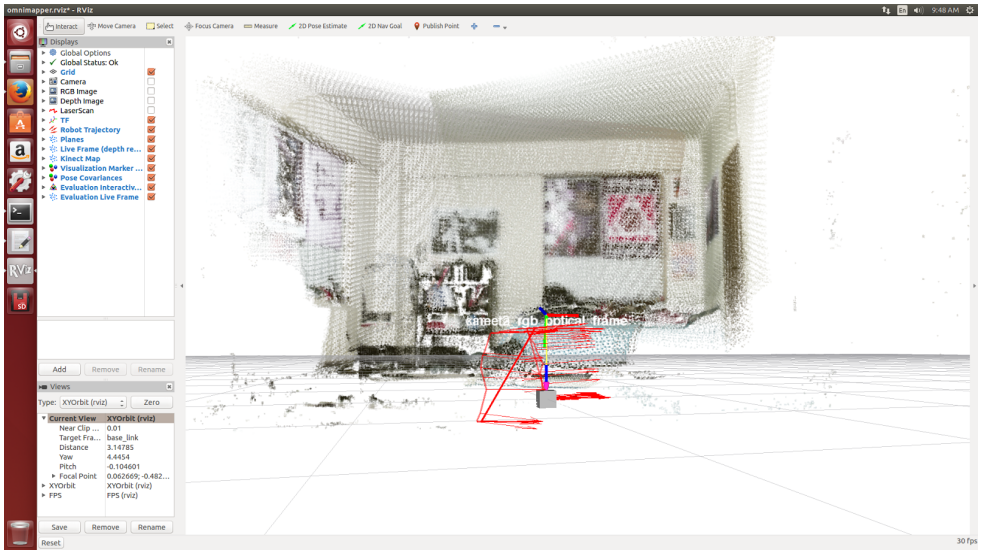


Image from SLAM package testing

OmniMapper was then researched and the operation behind it was summarized. It is a plugin-based architecture which allows different sensor types to be combined for SLAM, and uses the backend of GTSAM for actually executing SLAM. In order to test it, data was extracted from the Kinect 2 using the libreenect2 linux drivers and the IAI_Kinect2 ROS package was used to publish the point cloud from it as a ROS topic. Additionally, a static transform publisher was created in a launch file to specify where the Kinect is relative to the global frame.

View of Kinect 2 data in rViz

Then, OmniMapper SLAM was tested by creating a launch file with the appropriate parameters and launch both the Kinect and OmniMapper nodes.



SLAM results from just the Kinect 2

As seen above, both the location trajectory and combined point cloud map could be visualized in rViz. However, the results were noisy and the trajectory was sensitive to fast movements. This was in significant part due to the low frequency (~1 Hz) of the point clouds and the possibility of ICP not converging on subsequent iterations. Therefore, the next step was to integrate the high-frequency odometry from the robot to make the localization more accurate.

OmniMapper is also configured to use a transform from Odometry as an input to SLAM:

```
<param name="odom_frame_name" value="/odom"/>
 <param name="use_tf" value="true"/>
 <param name="tf_trans_noise" value="0.05"/>
 <param name="tf_roll_noise" value="10000"/>
 <param name="tf_pitch_noise" value="10000"/>
 <param name="tf_yaw_noise" value="0.2"/>
```

Modifications to OmniMapper launch file to use Odometry

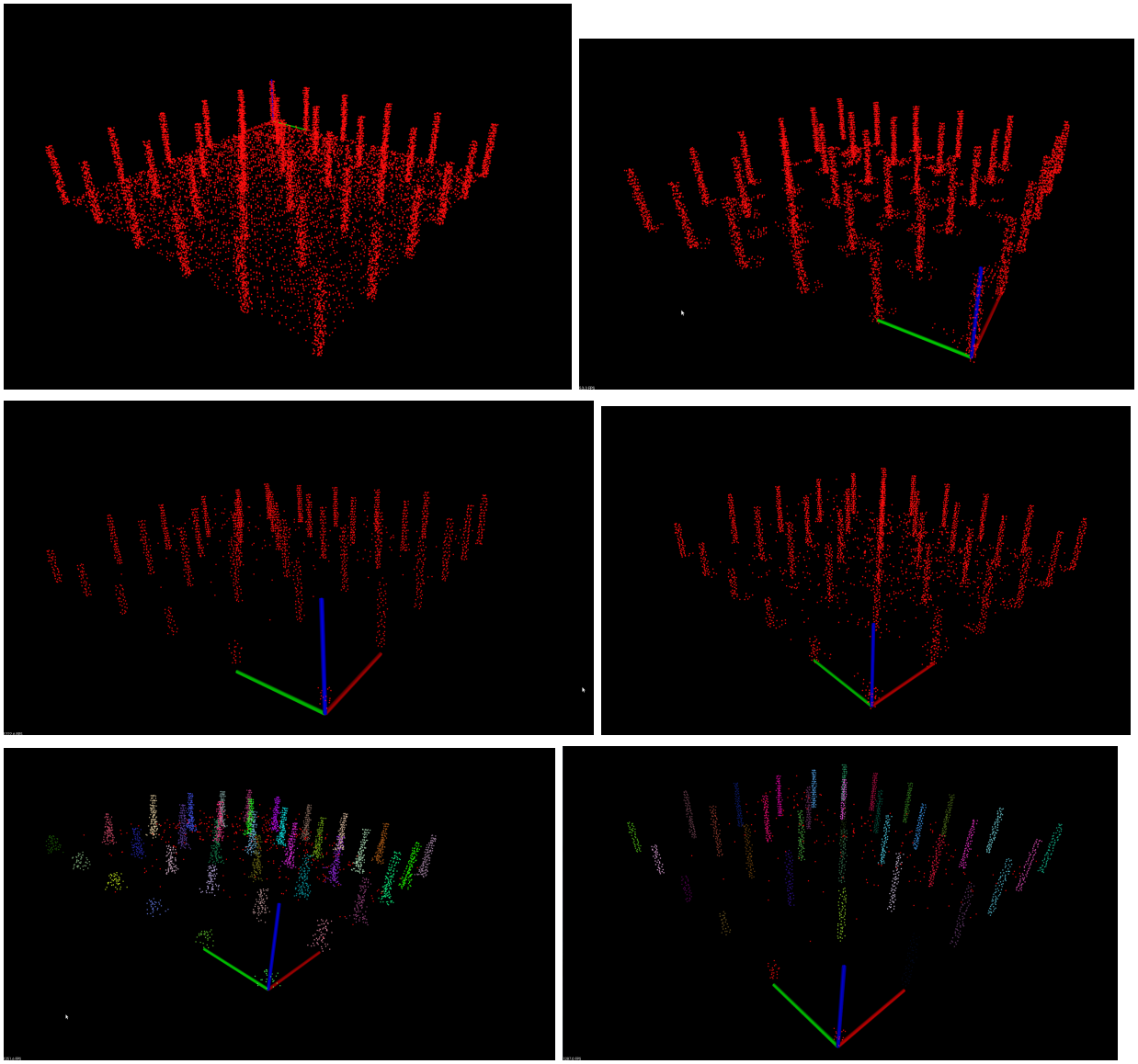
```
tfBroadcaster = new tf::TransformBroadcaster();
...
void odometry_update(){
double ex = robot->getEncoderX() / MM_IN_M;
double ey = robot->getEncoderY() / MM_IN_M;
double et = to_radians(robot->getEncoderTh());
    tf::Transform transform;
    transform.setOrigin( tf::Vector3(ex, ey, 0.0) );
    tf::Quaternion q;
    q.setRPY(0, 0, msg->et);
    transform.setRotation(q);
    tfBroadcaster.sendTransform(tf::StampedTransform(transform,
    ros::Time::now(), "odom",  "agribot"));
}
```

Code to send the Odometry based TF messages to OmniMapper

Though this code was written and compiled, there was no way to easily test the combined SLAM since it would require the Kinect to be mounted on the robot, which at that point was broken. Therefore, with the implementation in place the next step was to implement the code for locating plants within point clouds, as a key purpose of having the point cloud sensor was for finding plants to spray dynamically. To address this problem, we made the assumption that plants will be reasonably far from each other and so will result in clusters of point clouds denser than their surroundings. The Point Cloud Library (PCL) was used to implement the filtering and clustering steps needed to find plants with this assumption:

1. Filter out noise by removing statistical outliers
2. Downsample to simplify cloud
3. Filter out points below some threshold (remove ground)
4. Build KDTree on this Point Cloud

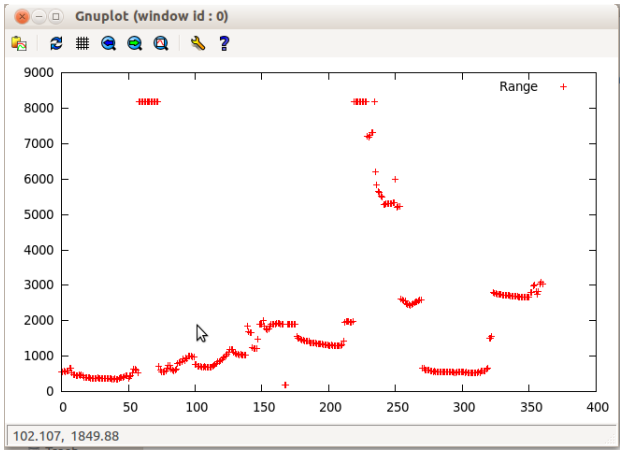
5. Perform Euclidean Clustering to find plants

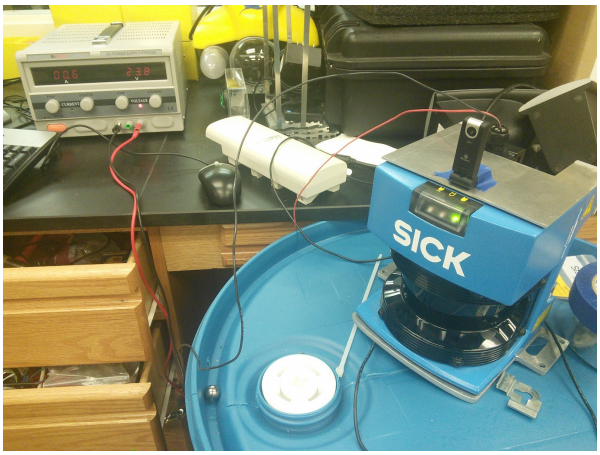


The code for this was completed first, and the next step was to test it. The plan was to test it with actual point cloud data from our test plant structures, but this proved to be impossible since the Jetson TK1 computer we were working with unexpectedly broke before this was undertaken. In order to complete the testing regardless, a synthetic point cloud was created and used for testing; above are images for the filtering and final clustering steps, from top left to bottom right. The bottom right displays clustering results for two different point clouds, showing that the approach works even despite some noise.

In parallel with this synthetic testing, we pressed on with the goal of integrating the LIDAR laser sensor for SLAM as well as obstacle detection. First, we worked to just get the SICK sensor

working. We wired the necessary connector based on the SICK LMS 200 documentation, and used the sicktoolbox to successfully communicate with it and extract data from it.





As a new Jetson was not received until later and integration on the robot could not start until then, the goal of actually integrating the LIDAR data into SLAM and obstacle detection was started on. The sicktoolbox_wrapper package was installed and confirmed to be working. After this, the OmniMapper parameters were again modified to include the SICK as a sensor. Although the integrated hardware was not ready to test and therefore we could not confirm this worked, we also wrote code for detecting obstacles in front of the robot based on the SICK data.

```
<node name="sick_node" pkg="sicktoolbox_wrapper" type="sicklms"
respawn="false" output="screen">
    <param name="port" value="/dev/ttyUSB0"/>
</node>
```

Then, in the omnimapper launch parameters, this is straighforwardly added:

```
<param name="use_csm" value="true"/>
```

As can be seen in the OmniMapper source code, this makes the package subscribe to the output of the the created sicklms node (which outputs to the topic "scan"):

```
if (use_csm_)
 {
   // Subscribe to laser scan
   laserScan_sub_ = n_.subscribe ("/scan", 1,
    &OmniMapperROS::laserScanCallback, this);
```

New launch file for SICK and modification to OmniMapper

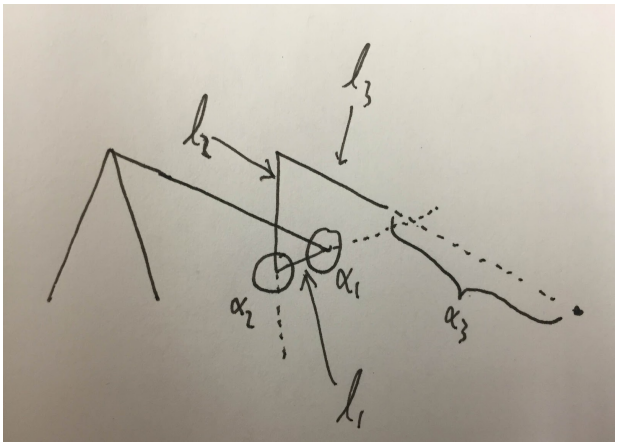
```
void lidarCallback(const sensor_msgs::LaserScan::ConstPtr& msg){
    obstacleInFront = false;
    for(int i=0; i <msg->ranges.size();i++){
        float dist = msg->ranges[i];
        float angle = msg->min_angle + i*(msg->angle_increment);
        if(angle>-20 && angle<20){
            if(range<OBSTACLE_DIST_THRESHOLD){
                obstacleInFront = true;
                break;
            }
        }
    }
}
```

Code for obstacle detection

Sprayer

To deliver water or chemical accurately, the robot needs to have some way to direct how it sprays. As pictured below, a hose extends from an electric pump to nozzle. This end-effector is mounted on two servos that together provide motion in the up-down and left-right directions.

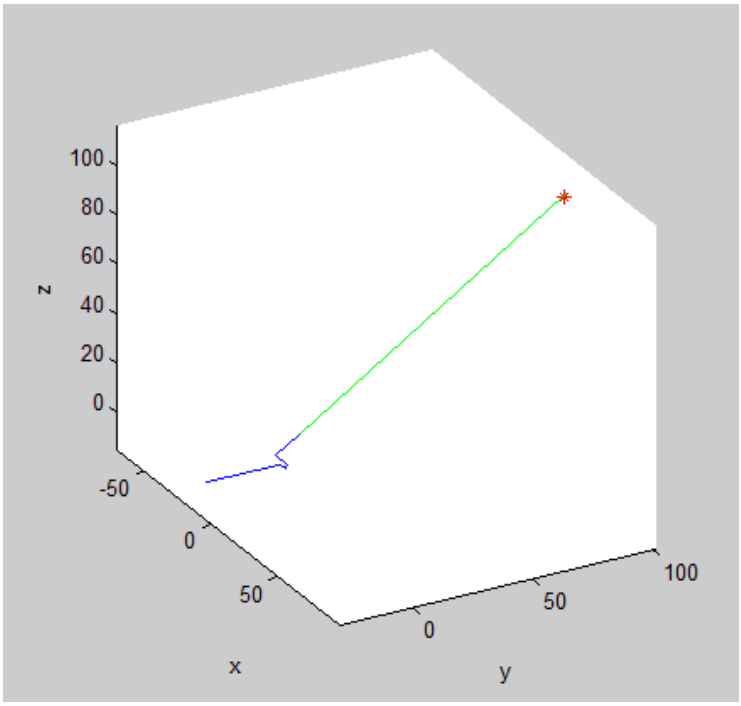
To aim the sprayer at a point, one must solve an inverse-kinematics problem. The form of this problem can be derived via symbolic forward kinematics to yield:



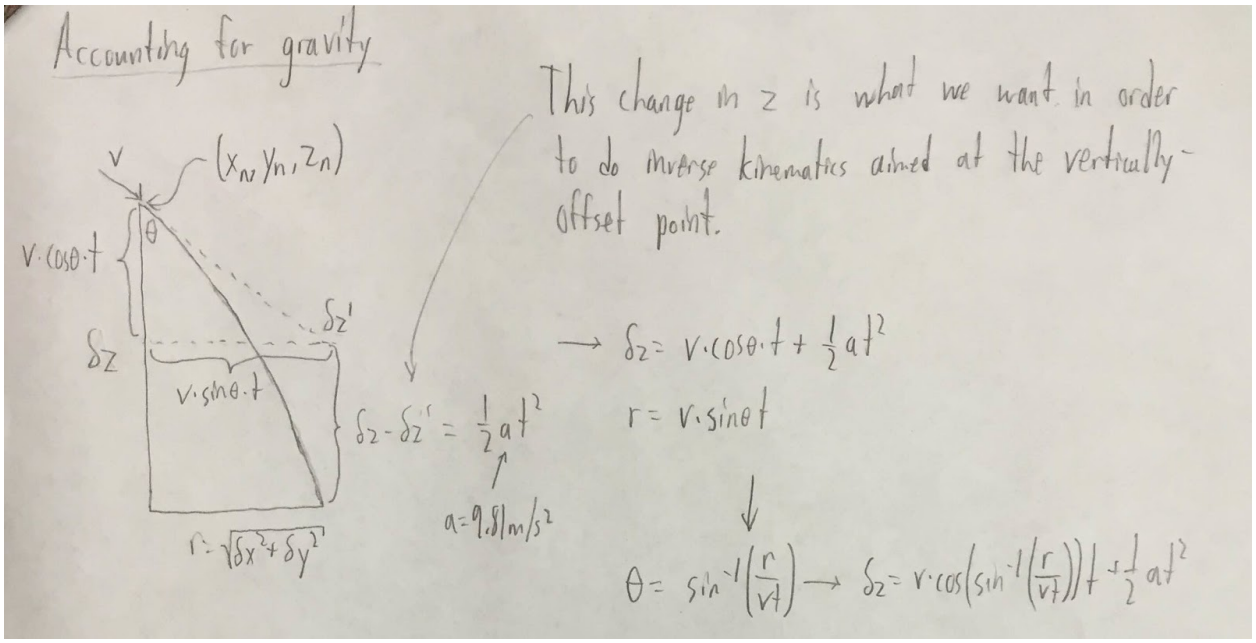
$$\begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} (\alpha_3 + l_3)(-\sin(\alpha_2)) \\ (\alpha_3 + l_3)\cos(\alpha_2)\cdot\cos(\alpha_1) - l_2\sin(\alpha_1) + l_1\cos(\alpha_1) \\ (\alpha_3 + l_3)\cos(\alpha_2)\cdot\sin(\alpha_1) + l_2\sin(\alpha_1) + l_1\sin(\alpha_1) \end{bmatrix}$$

Here the x dimension is parallel to L_1; the y dimension is parallel to L_3, and the z dimension is parallel to L_2.

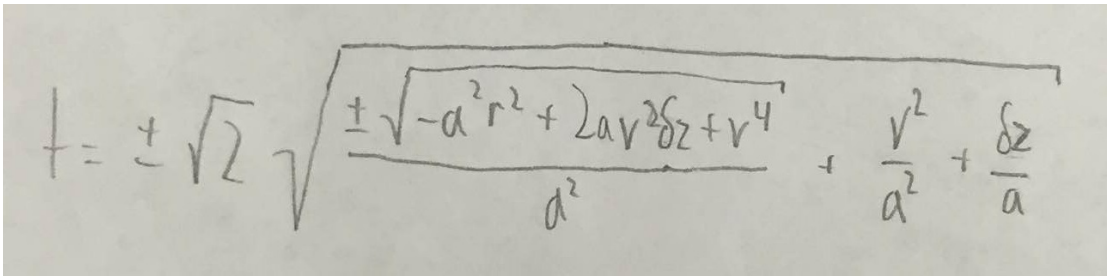
Solving this system for alphas yields complicated trigonometric expressions as codified in the Matlab and C submitted with this document. Plotting a stick-model of the manipulator aimed at a point looks something like the following.



To further improve this model, we decided to try to account for gravity. The kinematics of the liquid spray are as given.



These produce a solution for time-of-flight, t:



Unfortunately, this introduces a cycle of dependencies. We are trying to find the height difference between the point we want to hit and the point we need to aim at in order for the parabolic stream to hit said target. This quantity depends indirectly upon the current position of the end of the nozzle, which in turn is meant to be positioned according to the aforementioned height-difference.

The resolution to this dilemma is simply to solve the kinematics and inverse-kinematics equations in turn, plugging the most recent, nearer-to-correct numbers into the next step. Because this process might not always converge, we considered it too complex to be worth implementing on the physical robot.

Navigation

SLAM is a crucial part of navigation, since it constructs a map in which the robot may appropriately move to targets. The navigation subsystem consisted of the control logic that took as input the labeled map from SLAM, and appropriately controlled the wheels of the robot to move it to a desired target. This is a well-studied and nearly universal problem in autonomous robotics, and there are existing software libraries that can solve the path-planning problem. OMPL, or the Open Motion Planning Library, was researched to accomplish this for us. OMPL is a software C++ library that contains 20-30 planners which work in various control and state spaces, and it outputs paths from a initial state to a final state, avoiding obstacles along the way. OMPL was installed and used on a test maze, and various planners (KPIECE, EST, PDST, and RRT) were exercised.

The test maze

Platter Type	Planning Time (Min)	Path Time (20S planning)
KPIECE	7s	65s
EST	5s	51s
PDST	6s	44s
RRT	1s	63s

Path planning results

Although we were able to get OMPL running on a test maze, we were not far enough along in hardware development to have time to integrate it into the actual robot path planning. SLAM was also unable to be implemented in time, and this was another contributing factor to not using OMPL on the robot. Since for the project demonstration, the robot only needed to move down a single row of three plants, complex path-planning was not implemented for the robot's demo at the Senior Design Expo. Simple point-to-point navigation based on robot odometry was used.

Goal Status

With regards to SLAM, the full integration of multiple sensors for SLAM as well as plant and obstacle detection was implemented in code. However, only Kinect 2-based SLAM and plant detection with synthetic data could be tested due to hardware issues.

As far as the sprayer, the manipulator aims and works as intended. The IK works accurately and consistently.

Successful demonstration at the Senior Design Expo

OMPL

The group learned a great deal about existing motion planners (which were outside the scope of this course) while researching OMPL. The general concepts as well as idiosyncrasies in implementation were experienced (e.g., extremely long runtimes for SE(2) planning and the need for a kinematic car model). As mentioned earlier, we were unable to find the time to implement our goal of OMPL on the robot, and thus simple point-to-point navigation based on robot odometry was used (with no obstacle avoidance logic).

Conclusion

In researching different SLAM packages and how the SLAM within OmniMapper works, we learned a great deal about the variety of approaches to the problem as well as the different existing solution to it and their trade offs. We used PCL for the first time for some relatively

complex point cloud processing and visualization. Lastly, we wrote multiple ROS listeners as well as a ROS publisher in C++ and compiled the full, quite complicated package using rosmake, which none of us had done before.

In designing, assembling, and programming the physical manipulator, we learned that taking inverse kinematics can get extremely mathematically involved. Simplifying assumptions can greatly reduce the complexity of the process and may be worth making under certain conditions.

Beyond these outcomes, we learned that actual robots tend to fail alarmingly often. Loose wires, brief power outages, and faulty components conspire to make reality depart widely from ideality. Fixing these low-level issues took far more time than anticipated and contributed to contortion of attention away from complex, high-level functionality like SLAM and OMPL.