STUDENT SUMMER INTERNSHIP TECHNICAL REPORT

Development of Semi-Autonomous Robotic Platform for Mapping Radioactive Hanford Farms

DOE-FIU SCIENCE & TECHNOLOGY
WORKFORCE DEVELOPMENT PROGRAM

Date submitted:

December 10, 2021

Principal Investigators:

Joel Adams (DOE Fellow Student)
Florida International University

Alexander Pappas (Mentor)
Washington River Protection Solutions

Ravi Gudavalli, Ph.D. (Program Manager)
Florida International University

Leonel Lagos, Ph.D., PMP® (Program Director)
Florida International University

Submitted to:

U.S. Department of Energy
Office of Environmental Management
Under Cooperative Agreement # DE-EM0005213

Applied Research Center
FLORIDA INTERNATIONAL UNIVERSITY

TABLE OF CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

This research work has been supported by the DOE-FIU Science & Technology Workforce Initiative, an innovative program developed by the U.S. Department of Energy's Office of Environmental Management (DOE-EM) and Florida International University's Applied Research Center (FIU-ARC). During the summer of 2021, a DOE Fellow intern, Joel Adams, spent 10 weeks doing a summer internship at Washington River Protection Solutions (WRPS) under the supervision and guidance of Technology Integration Project Manager, Alexander Pappas. The intern's project was initiated on May 17, 2021, and continued through August 5, 2021 with the objective of developing a semi-autonomous robotics platform to perform radiation mapping at facilities by integrating numerous sensors and performing field testing of created algorithms.

1. INTRODUCTION

The cleanup mission at Hanford of radioactive contamination is aided by Washington River Protection Solutions' Chief Technology Office (CTO) which is responsible for the development of technology that increases the safety and efficiency of the mission. Semi-autonomous robotic platforms provide a promising step forward for the maturation of this technology as it allows for greater efficiency and safety of contamination surveillance.

These platforms include the Husky by ClearPath Robotics which was developed in order to navigate Handford sites while measuring radiation levels. This involved integrating numerous sensors such as LiDARs, cameras, and Geiger sensors, developing software for modeling robot behavior and using the gathered sensor data, and performing field testing with the platform to test the robustness of the development. Software development was done using the Robotic Operation System (ROS) which enabled modeling of complex robot behavior using a state machine and also allowed for the implementation of a simultaneous localization and mapping (SLAM) algorithm for exploration. Radiation maps were constructed from sparse points collected from a Geiger sensor using a cubic interpolation technique in order to create a curvature-minimizing polynomial surface.

2. SENSOR SETUP AND INTEGRATION

In order for robots to interact with the world by both taking in information and outputting actions, sensors and actuators are necessary. It is therefore essential that the setup of these items are appropriate for the task at hand. The platform used for completing the tasks is the Husky platform by Clearpath Robotics, which is shown in Figure 1. The platform provides actuators in the form of motors for the wheels, and has software available online for basic functionality, such as interfacing with the motors.

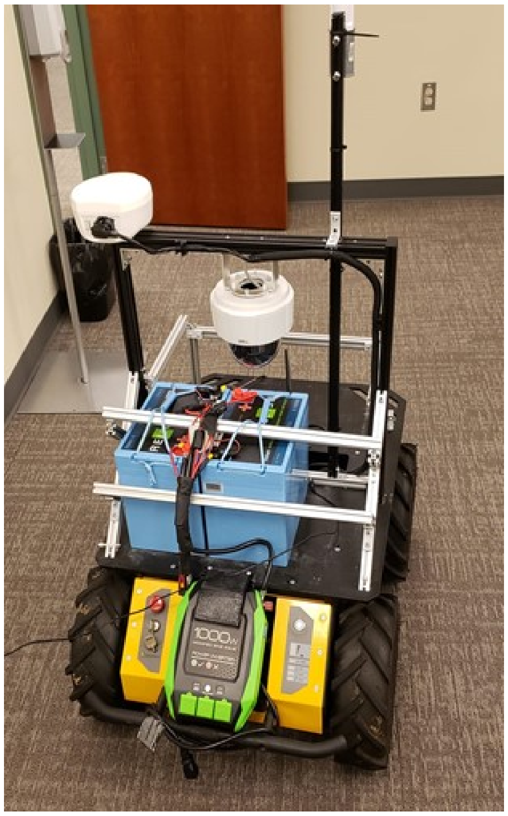
Figure 1. Clearpath Robotics Husky platform before development.

The planned setup was to utilize the onboard computer and, add two additional smaller computers, as well as three depth cameras for localization, obstacle detection, and mapping. For the existing onboard computer, a clean install of an operating system image was performed which came preinstalled with the Robotic Operating System (ROS). ROS provides the user with a framework for developing software for robots by having a messaging system that standardizes numerous types of sensor data and robot information, makes working with a distributed computing network seamless, and allows integration with a global community of robot software developers. The additional computers added, the Nvidia Xavier and a Raspberry Pi computer allow for the offloading of computational load so that the system can handle the work.

The three depth cameras are all Intel RealSense cameras that use a stereo camera setup in order to derive a disparity image with depth information of the nearby environment. Using this information can assist the robot in estimating its position in space as it traverses the environment, as well as concatenating a point cloud to form a colored map. Additionally, obstacle detection and avoidance can be done with the assistance of these sensors.

The three camera sensors were placed 45 degrees off from each other in order to cover a wide space in front of the robot and ensure no obstacles are missed, shown in Figure 2.

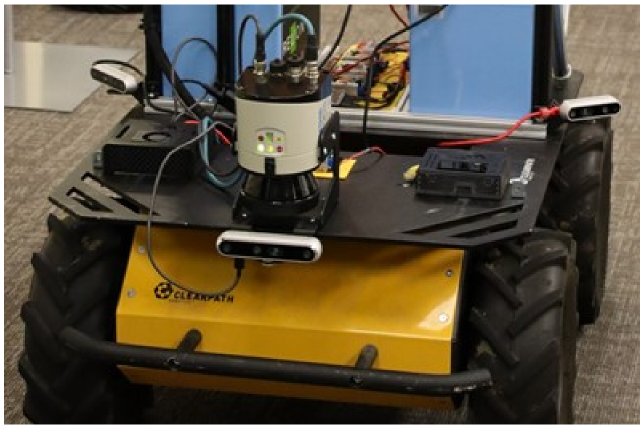
Figure 2. Setup of three depth cameras and onboard computers.

Based on the mapping sessions completed, the setup performed adequately in indoor environments with one such example shown in Figure 3 below. The top image in the figure shows a concatenated 3D colored point cloud map of a conference room with minor imperfections visible. The bottom image shows both the path traversed by the platform when performing the mapping session, as well as boxes surrounding individual point cloud sections that were stitched together using SLAM, an algorithm that will be addressed in the robot localization and navigation section of this report.

Tests performed outside with this sensor setup proved to be inadequate for the purposes of the task. This is because of the combination of both the vastness of outdoor environments and the relatively short range of the depth cameras' point clouds. The resulting point clouds only captured pavement underneath the robot in a bubble of a few meters. This also inhibited the robot's ability to localize itself in the environment since the ground did not have an abundance of geometric features.

For these reasons, a new sensor setup was arranged for the Husky mobile platform which included a Velodyne 3D LiDAR. The advantage of using this sensor is it can reach points up to 100 meters away. The LiDAR sensor has a full 360° range to gather points, resulting in a greater number of points than the previous setup could provide. The sacrifice made by switching setups was a larger margin of error for each point in space, forfeiting some of the accuracy of smaller details in the map. The sensor also carries a larger computational load, although this can be offset by leaving much of the post-processing of the point cloud data to be performed after the robot is finished mapping instead of live during data collection. A voxel filter is also utilized to reduce the number of redundant points in the cloud.

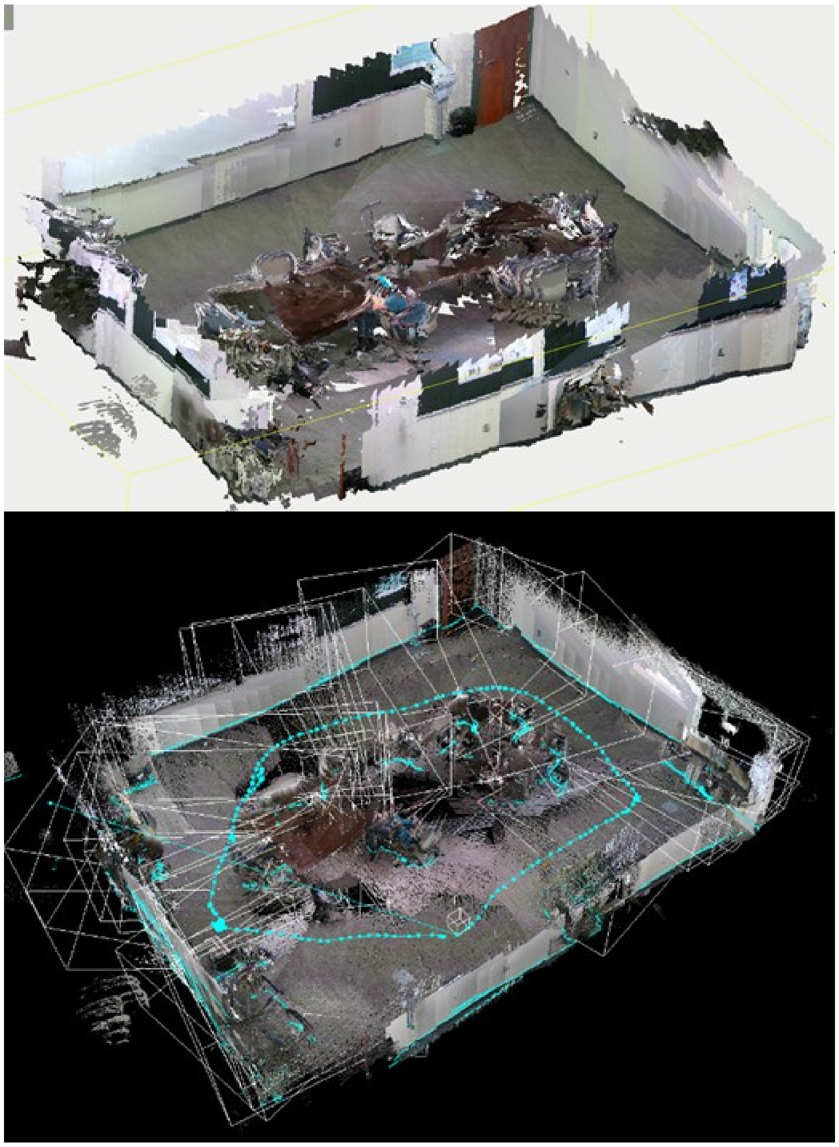

Figure 3. 3D maps of conference room (top) and map with path traveled shown (bottom).

Live data readings from the 3D LiDAR sensor are visualized in Figure 4, and the complete map formed using the sensor is shown in Figure 5. These maps have no color as the sensor is not used in combination with RGB cameras as with the depth cameras. The environment is still able to be seen with ease by people viewing the map.

The localization was estimated using ICP odometry on the LiDAR data and numerous other setups were tested using a combination of IMU and wheel encoder sensors, which will be discussed in the next section. This is important to note because the accuracy of the map depends not only on the quality of the LiDAR sensor used, but also on the robustness of the localization techniques used on the robot platform.

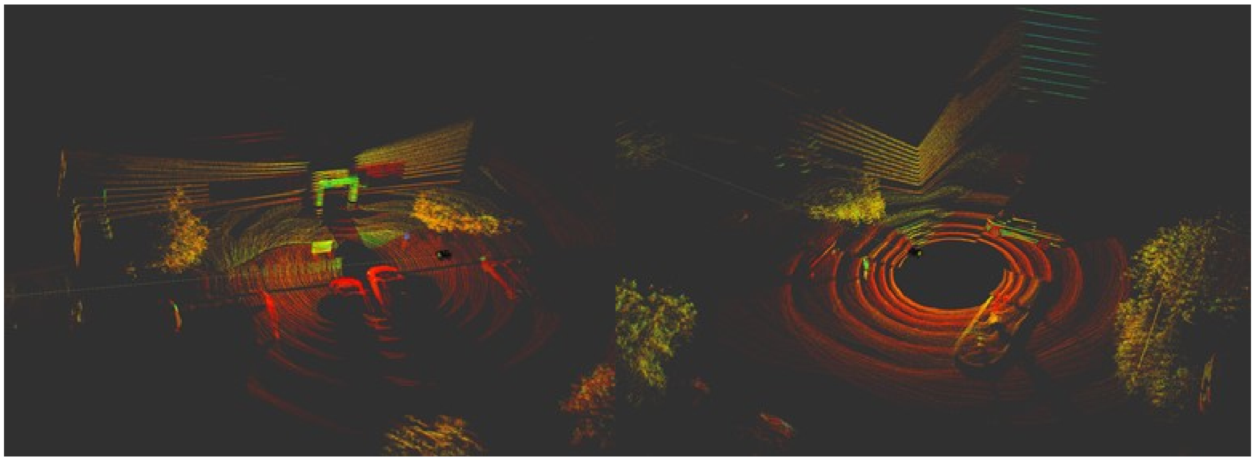
Figure 4. Live data view of Velodyne 3D LiDAR of outside environment.

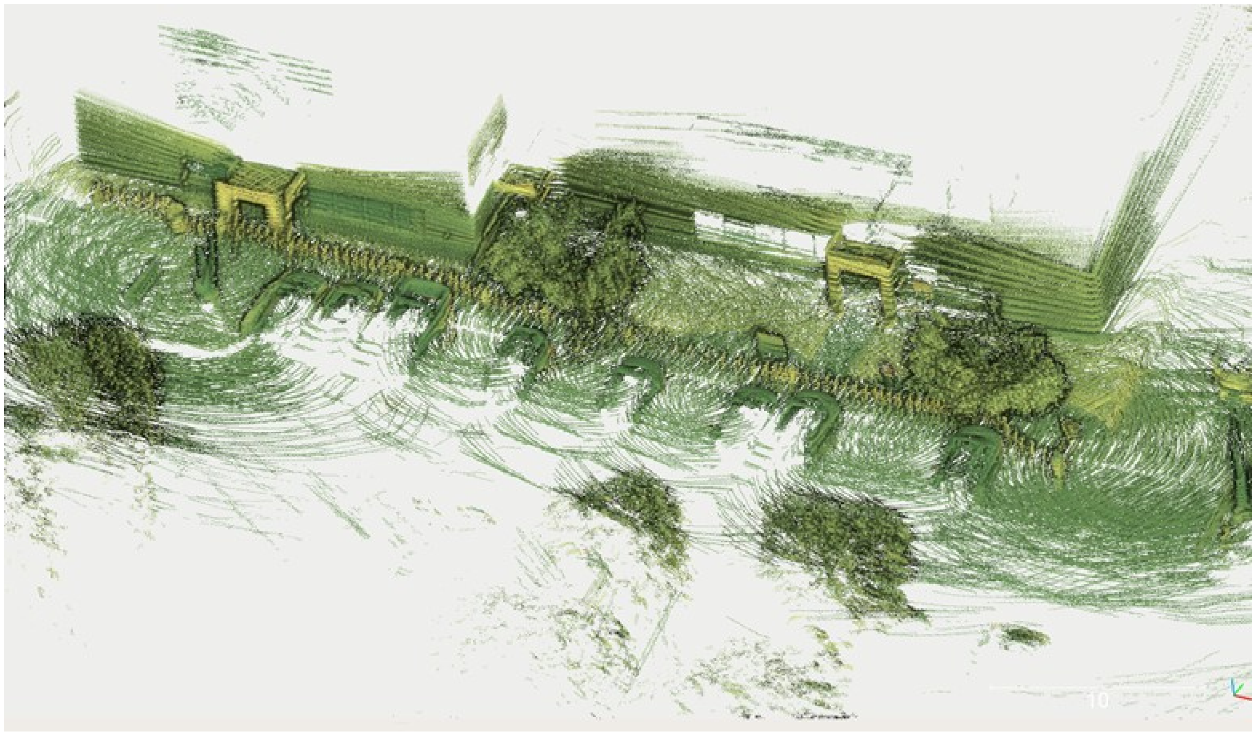

Figure 5. 3D map of north side of building made using Velodyne 3D LiDAR with visual filters applied for enhanced appearance.

3. ROBOT LOCALIZATION AND NAVIGATION

Robot localization refers to a robot's ability to estimate its position and orientation in space at any given time. This is a challenge for a number of reasons, including dead reckoning error which refers to the accumulation of small errors over a larger distance. One way to counter this issue is using a simultaneous localization and mapping (SLAM) technique which attempts to use information from mapping to correct the localization and vice versa. The SLAM algorithm used is RTAB-Map's graph-based approach. The correction of a map section from a SLAM algorithm is referred to as loop-closure. In order to get an accurate map of an environment, the ideal is to have small dead reckoning error and frequent loop-closure occurrences.

Localization is a requirement for both mapping and navigation of an area because it is desired to set goals for the platform to travel to and to calculate trajectories that avoid obstacle collision. The real-time avoidance of obstacles is made possible using the depth cameras and the 3D LiDAR that was later installed. For the localization part of the platform, a number of different combinations was tested involving an inertial measuring unit (IMU), the wheel encoders, the depth cameras, and the 3D LiDAR. Kalman filtering techniques allow for the combining of multiple sources of odometry by taking into account the uncertainty and covariance of each source. The implementation of this technique is achieved using the ROS package "Robot Localization". The final setup used was ICP odometry alone from the 3D LiDAR sensor, which was fed into the SLAM algorithm to construct the geometric map of the environment.

Figure 6 below shows an obstacle map of an indoor room visualized on a ROS software package called RVIZ. The pink color indicates an obstacle, whereas the cyan color indicates a buffer zone set around this layer. The software stack used for this application is the ROS Navigation Stack, which constructs the obstacle layer from the sensor data and calculates trajectories in real-time in order to navigate to set points in space.

The usefulness for all of this functionality is not only for mapping, but also for navigating to set points in space called waypoints. These can be set using 3D coordinates and orientation values and combined in a list that can be fed to the navigation stack sequentially. This was achieved by writing a new ROS program that takes a simple text file with waypoint information as input and publishes the information by responding to a ROS service. A ROS service is an implementation in the Robotic Operating System of a client-server paradigm so that information can be requested instead of continuously published. An additional program was written that writes this text file using a remote controller that is setup with the robot platform. The source code for these programs is listed in the appendix of this report.

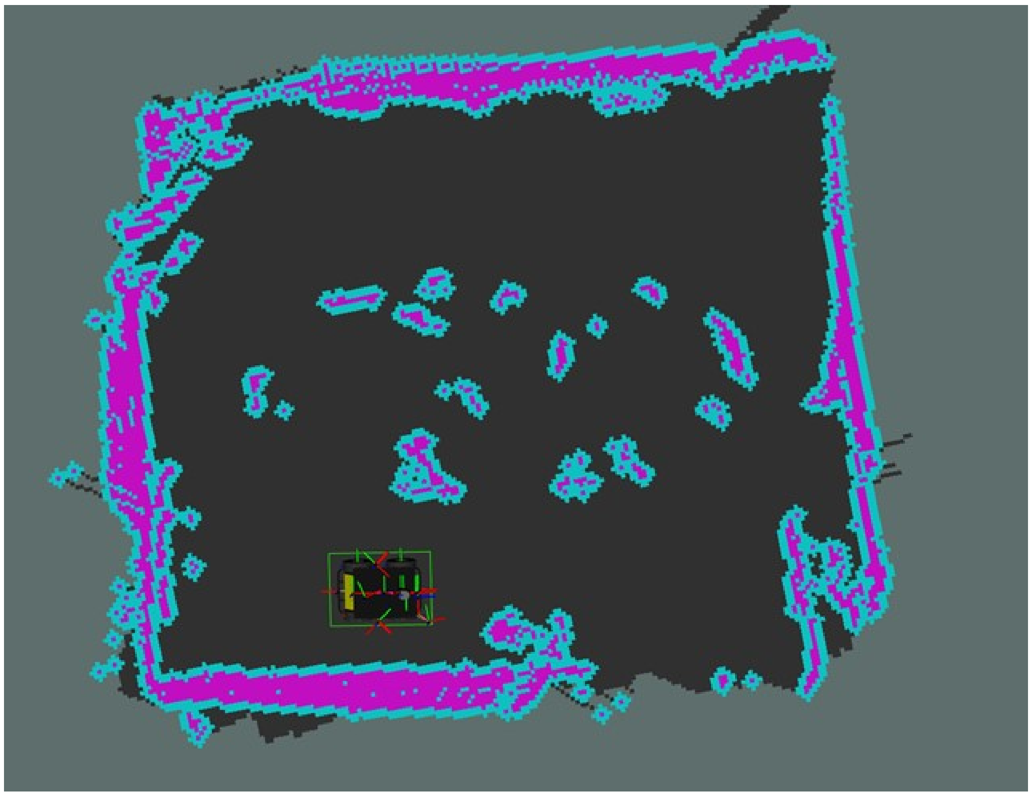

Figure 6. Visualization of obstacle layer used for navigation derived from robot's sensor data.

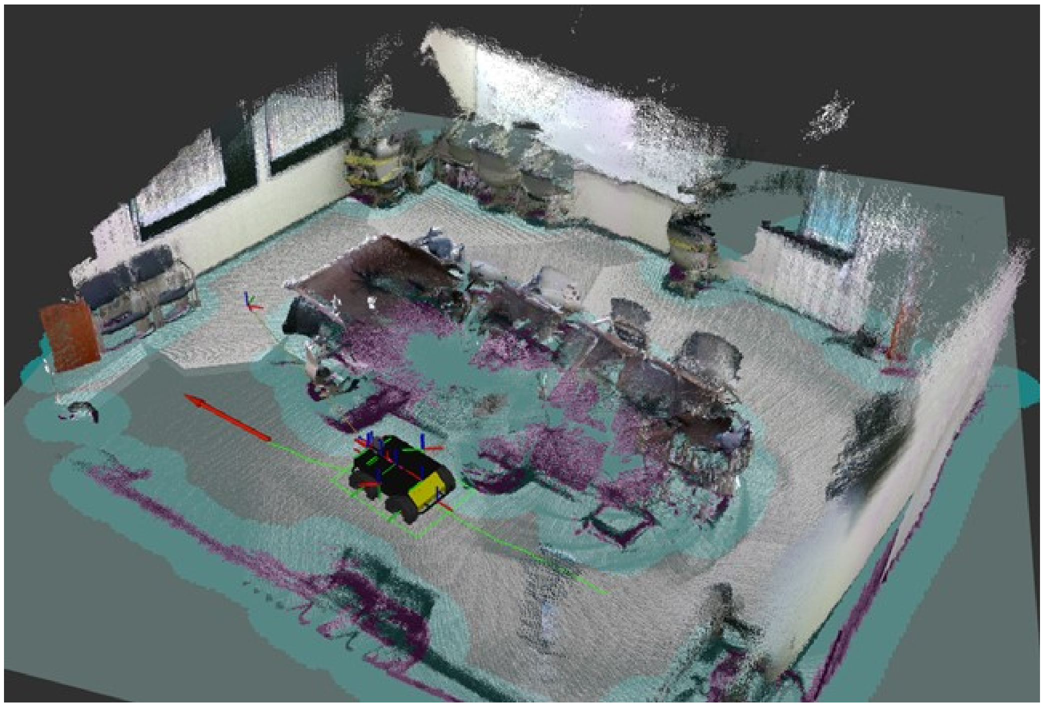

Figure 7. View inside RVIZ software of robot mapping and navigating to a specified point.

4. RADIATION MAP

To generate rough radiation maps of the environment, a simple Geiger sensor is used which is hooked up to the Raspberry Pi computer. A program was written utilizing ROS action client library, which is similar to ROS services except it also allows for live feedback upon a request from an action client. The error for the measurements made by this sensor decreases with time, so the robot stays still while a measurement is being taken. These measurements are of a single point in space and thus not an adequate map of radiation in the environment. For this reason, a surface interpolation technique is needed in order create an acceptable map. Figure 8 below shows an example of 2D maps made using randomized data points generated in Python.

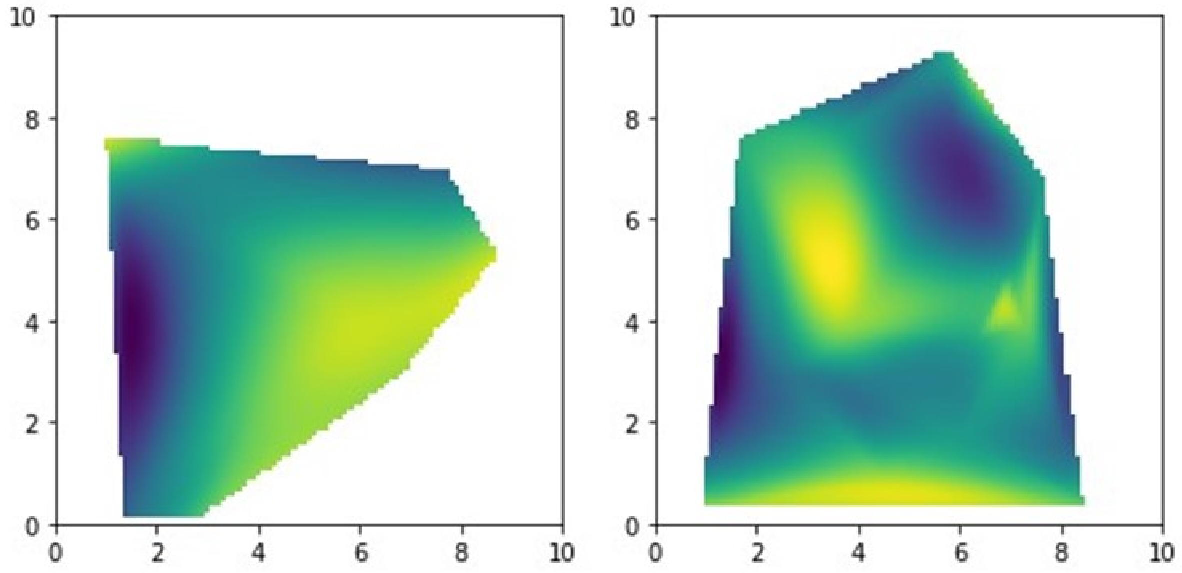

Figure 8. Interpolated surfaces using randomly generated samples.

Numerous different surface interpolation techniques were researched including Kriging, a geostatistical technique, and Natural Neighbor, a technique using Voronoi tessellations. Ultimately the SciPy Interpolate function from the python library SciPy was used which has a collection of scientific tools.

The software written allows for requests to take a radiation measurement using the Geiger sensor. The measurement is then taken until the error is below a specified range and then a final value is provided to the ROS system to be used for surface interpolation. As will be described in more detail in the next section, the radiation map is constructed after the mapping session is finished as a part of post-processing. The values measured from the Geiger sensor are stored in a vector as well as the coordinates of each measurement. These pieces of information are used in a Python script after to construct the final radiation map.

5. STATE MACHINE

State machines allow for robots to engage in more complex behavior involving decision making. Each state has its own set of code to follow and specified criteria allows for additional states to be activated. The Python library used is called Smach, which is well integrated with ROS. Figure 9 below shows the state machine developed for the Husky's radiation mapping procedure.

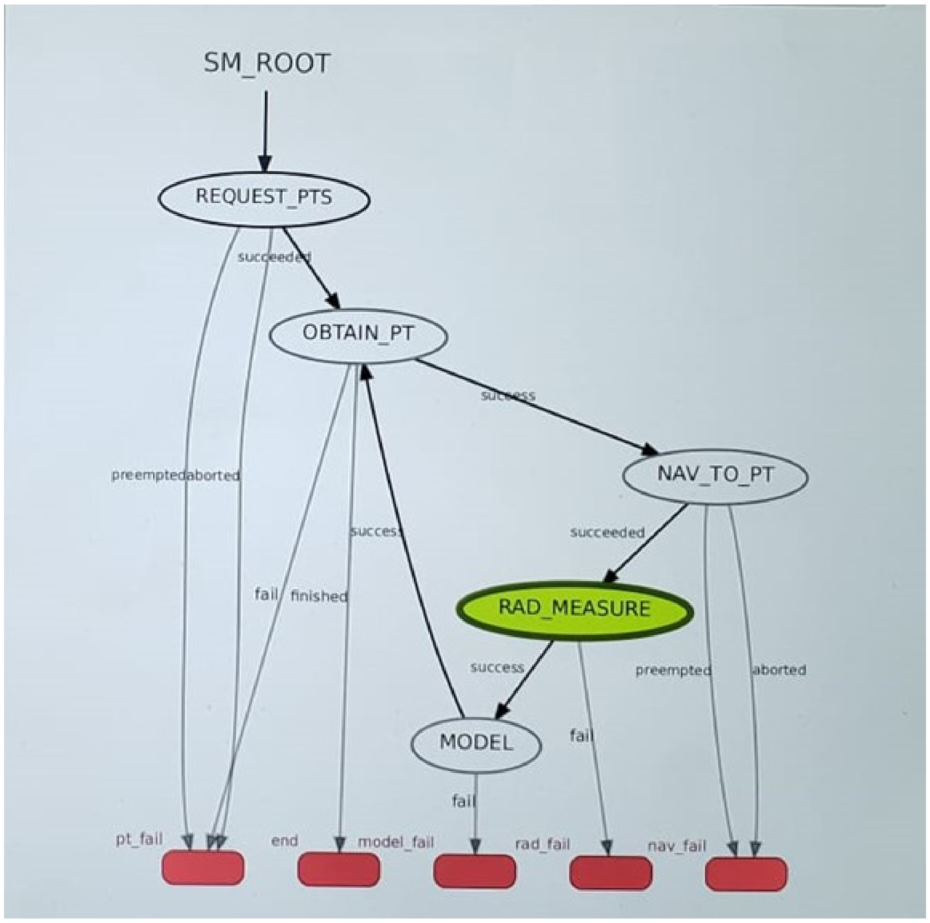

Figure 9. State machine for Husky robot used to perform radiation mapping.

The robot is first manually driven using a remote controller and buttons allow the user to specify points in space to later travel to in sequence. These points are saved into a text file which the state machine requests upon launching. Once the points are retrieved from the list, the machine iterates through each point individually and then travels to that point and takes a radiation measurement at that location. The robot stays still while measurements are taken and then the point is saved in the modeling state before further iterating through the point list.

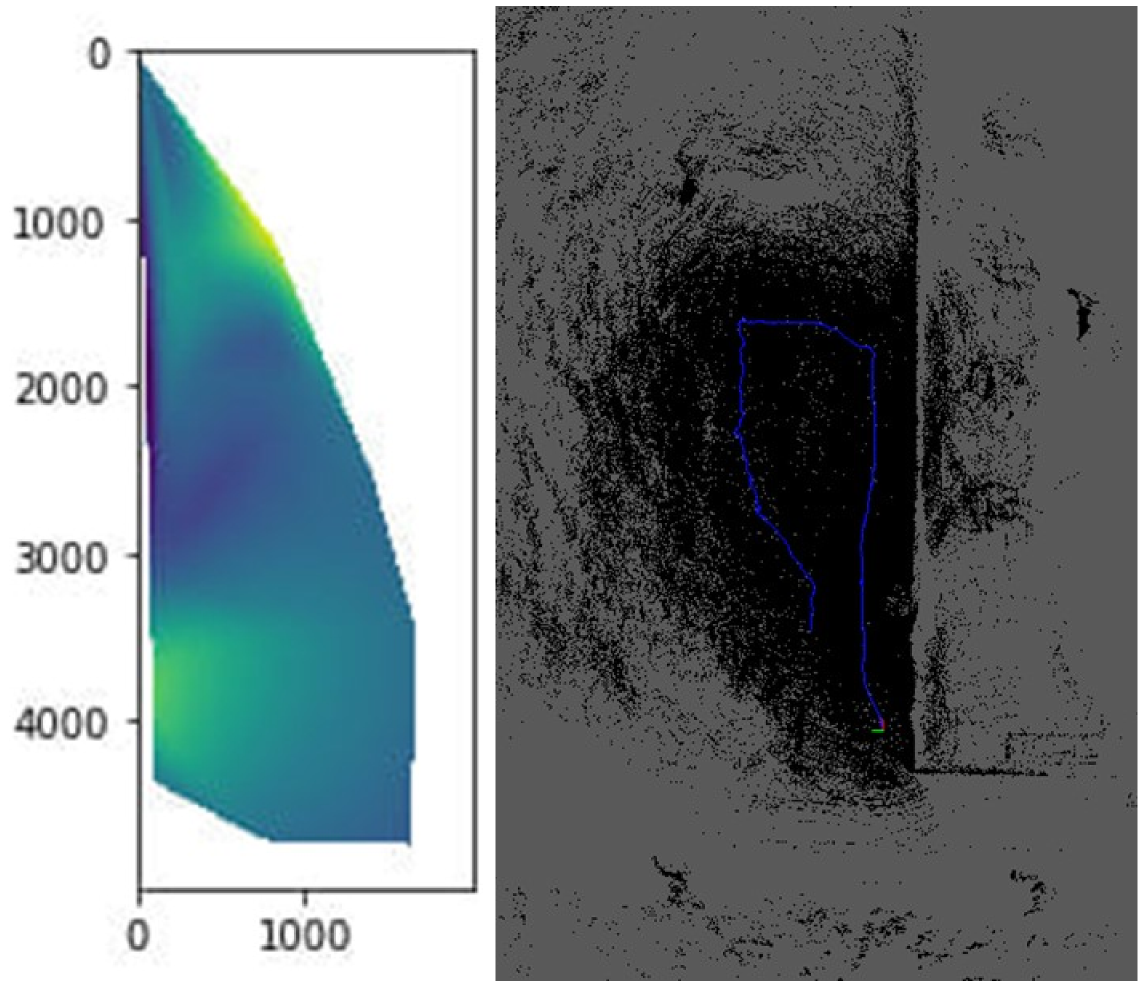

Figure 10. Radiation map of farm location (left) with units in millimeters and aerial view of point cloud for same location (right).

The Husky platform was taken to a farm location where this procedure was tested. Figure 10 above shows both the radiation map constructed from the trip as well as the corresponding point cloud generated using SLAM. The actual radiation values are not shown as the sensor accuracy of the sensor is not high. For this reason, the usefulness of the radiation map is in finding trends and specific locations of hot spots. In the future, more expensive sensors can be integrated for a more robust map.

6. CONCLUSIONS

A semi-autonomous robotic platform was developed for the Chief Technology Office at Washington River Protection Solutions in order to aid in the cleanup mission at Hanford's radioactive sites. The goal is to develop technology that will increase the overall safety and efficiency of this Hanford mission. The Husky platform was developed starting with the installation of a new operating system and a collection of sensors which were integrated into the system. After further refinement of this setup, the localization and navigation of the robot was further developed in order to better equip it for field testing. Software was written to collect data from a Geiger sensor, which was then used in combination with a state machine to create an algorithm for mapping an environment and producing a 2D radiation map alongside its geometric twin. This setup was tested at a Hanford site and the resulting maps demonstrated that the robot is capable of leveraging all of its data from its sensors in order to make complex decisions and construct a geometric and radiation map of its environment.

APPENDIX A.

State Machine Code:

```
#!/usr/bin/env python
import rospy
import smach
from smach_ros import ServiceState, SimpleActionState, IntrospectionServer
from waypoint_list_server.srv import *
import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from radiation.msg import rad_testAction, rad_testFeedback, rad_testResult
import numpy as np
import scipy.interpolate
import matplotlib.pyplot as plt
import pathlib
import os

class Obtain_pt(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=['success','fail','finished'],
                             input_keys=['obt_input'],
                             output_keys=['obt_output','counter'])
        self.counter = 0

    def execute(self, userdata):
        rospy.loginfo('Executing state Obtain_pt')
        if (self.counter >= len(userdata.obt_input.poses)):
            rospy.loginfo("Reached end of point list!")
            return 'finished'
        else:
            my_path = os.path.abspath(__file__)
            print(my_path)
            rospy.loginfo('Moving to the following point:')
            list = userdata.obt_input
            goal = MoveBaseGoal()
            goal.target_pose.header.frame_id = "map"
            goal.target_pose.header.stamp = rospy.Time.now()
            goal.target_pose.pose = list.poses[self.counter]
            userdata.obt_output = goal.target_pose
            rospy.loginfo(goal.target_pose)
            userdata.counter = self.counter
            self.counter += 1
            return 'success'

class Model(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=['success','fail','end'],
                             input_keys=['mod_input','location','size','counter']),

    def execute(self, userdata):
        rospy.loginfo('Executing state MODEL')
        location = np.array([userdata.location.pose.position.x,userdata.location.pose.position.y])
        size = userdata.size
        if(userdata.counter == 0):
            self.values = np.zeros([userdata.size])
            self.points = np.zeros([userdata.size,2])
        #np.append(self.values,location)
        #np.append(self.points,[userdata.mod_input])

        self.values[userdata.counter] = userdata.mod_input
        self.points[userdata.counter] = location
        rospy.loginfo("Radiation Measurement recieved is: %f", userdata.mod_input)
        grids = tuple(np.mgrid[0:10:0.01,0:10:0.01])
        rospy.loginfo(self.values)
        rospy.loginfo(self.points)
        if (userdata.counter == userdata.size):
            scipy_interpolated_values =
scipy.interpolate.griddata(self.values,self.points,grids,method='cubic')
            plt.imshow(scipy_interpolated_values, extent(0,10,0,10),origin='lower')
            my_path = os.path.abspath(__file__)
            print(my_path)
            plt.savefig('rad_map.png')
        return 'success'

# main
def main():
    rospy.init_node('smach_example_state_machine')

    # Create a SMACH state machine
    sm = smach.StateMachine(outcomes=['end', 'nav_fail', 'pt_fail',
                                      'rad_fail', 'model_fail'])

    # Open the container
    with sm:

        # Add states to the container
        smach.StateMachine.add('REQUEST_PTS',
                    ServiceState('waypoint_list_server',
                             ReadWaypointList,
                             request = "waypoints.txt",
                             response_slots = ['list','size']),
                             #output_keys = ['waypoint_output']),
                    transitions={'succeeded':'OBTAIN_PT',
                                 'preempted':'pt_fail',
                                 'aborted':'pt_fail'},
                    remapping={'list':'points',
                                 'size':'list_size'})
        smach.StateMachine.add('OBTAIN_PT', Obtain_pt(),
                              transitions={'success':'NAV_TO_PT',
                                           'fail':'pt_fail',
                                           'finished': 'end'},
                              remapping={'obt_input':'points',
                                           'obt_output':'point',
                                           'counter':'global_counter'})
        smach.StateMachine.add('NAV_TO_PT',
                         SimpleActionState('move_base',
                                           MoveBaseAction,
                                           goal_slots=['target_pose']),
                         transitions={'succeeded':'RAD_MEASURE',
                                      'preempted':'nav_fail',
                                      'aborted':'nav_fail'},
                         remapping={'target_pose':'point'})
        smach.StateMachine.add('RAD_MEASURE',
                         SimpleActionState('rad_action_server',
                                           rad_testAction,
                                           result_slots=['radiation']),
                                           transitions={'succeeded':'MODEL',
                                                        'preempted':'rad_fail',
                                                        'aborted':'rad_fail'},
                              remapping={'radiation':'data_pt'})
        smach.StateMachine.add('MODEL', Model(),
                              transitions={'success':'OBTAIN_PT',
                                           'fail':'model_fail'},
                              remapping={'mod_input':'data_pt',
                                           'location':'point',
                                           'size':'list_size',
                                           'counter':'global_counter'})

    sis = IntrospectionServer('server_name', sm, '/SM_ROOT')
    sis.start()
    outcome = sm.execute()

    rospy.spin()
    sis.stop()

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

Waypoint List Creator Node

```
#include <ros/ros.h>
#include <tf2_ros/transform_listener.h>
```

```
#include "tf2_geometry_msgs/tf2_geometry_msgs.h"
#include <sensor_msgs/Joy.h>
#include <fstream>
#include <iostream>

tf2_ros::Buffer tfbuffer;
std::ofstream ofile;

void joyCallback(const sensor_msgs::Joy& joy)
{
  if ((joy.buttons[0] == 1) && !ofile.is_open()){
    ofile.open("waypoints.txt");
  }
  if ((joy.buttons[1] == 1) && ofile.is_open()){
    ofile.close();
  }
  if ((joy.buttons[2] == 1) && ofile.is_open()){
    geometry_msgs::TransformStamped transform;
    try {
      transform = tfbuffer.lookupTransform("map", "base_link", ros::Time(0));
    } catch (tf2::TransformException &ex) {
      ROS_WARN("%s",ex.what());
      ros::Duration(0.1).sleep();
    }
    float x = transform.transform.translation.x;
    float y = transform.transform.translation.y;
    float z = 0.00;
    float qx = transform.transform.rotation.x;
    float qy = transform.transform.rotation.y;
    float qz = transform.transform.rotation.z;
    float qw = transform.transform.rotation.w;
    tf2::Quaternion quat(qx,qy,qz,qw);
    tf2::Matrix3x3 m(quat);
    double roll, pitch, yaw;
    m.getRPY(roll, pitch, yaw);
    std::cout << x << ", " << y << ", " << z << ", "
    << roll << ", " << pitch << ", " << yaw << std::endl;
    ofile << x << ", " << y << ", " << z << ", "
    << roll << ", " << pitch << ", " << yaw << std::endl;
    ros::Duration(1).sleep();
  }

}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "waypoint_file_creator");
```

```
 ros::NodeHandle n;
 tf2_ros::TransformListener tf_listener(tfbuffer);
 std::string joy_topic;
 if (n.getParam("joy_topic", joy_topic))
 {
   ros::Subscriber sub = n.subscribe(joy_topic, 1, joyCallback);
   ros::spin();
 }
 else {
   ros::Subscriber sub = n.subscribe("/joy_teleop/joy", 1, joyCallback);
   ros::spin();
   }
 return 0;
}
```

Waypoint List Program

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "geometry_msgs/PoseStamped.h"
#include "geometry_msgs/PoseArray.h"
#include <tf2/LinearMath/Quaternion.h>
#include <sstream>
#include <list>
#include <iostream>
#include <fstream>
#include <vector>
#include <waypoint_list_server/ReadWaypointList.h>

bool readFile(waypoint_list_server::ReadWaypointList::Request &req,
              waypoint_list_server::ReadWaypointList::Response &res){
    std::vector<std::string> lines;
    std::string line;
    int size;
    std::cout << req.filename << std::endl;
    std::ifstream input_file(req.filename);
    std::list<geometry_msgs::PoseStamped> waypoint_list;
    geometry_msgs::PoseStamped waypoint;
    int seq = 0;
    if (!input_file.is_open()) {
        std::cerr << "Failed to open file" << std::endl;
        return false;
    }
    while (getline(input_file, line)) {
        lines.push_back(line);
    }
    input_file.close();

    size = lines.size();
```

```
for (auto l : lines) {
    int i = 0;
    int j = l.find(',');
    auto x = l.substr(i,j);
    i = j;
    j = l.find(',',j+1);
    auto y = l.substr(i+1,j-i-1);
    i = j+1;
    j = l.find(',',j+1);
    auto z = l.substr(i,j-i);
    i = j+1;
    j = l.find(',',j+1);
    auto roll = l.substr(i,j-i);
    i = j+1;
    j = l.find(',',j+1);
    auto pitch = l.substr(i,j-i);
    i = j+1;
    j = l.find(',',j+1);
    auto yaw = l.substr(i,j-i);
    waypoint.header.seq = seq;
    waypoint.header.frame_id = "map";
    //std::cout << x << " and " << y << " and " << z << " and " << roll <<
    // " and " << pitch << " and " << yaw << std::endl << std::endl;
    waypoint.pose.position.x = std::stof(x);
    waypoint.pose.position.y = std::stof(y);
    waypoint.pose.position.z = std::stof(z);
    tf2::Quaternion rot;
    rot.setEuler(std::stof(roll),std::stof(pitch),std::stof(yaw));
    waypoint.pose.orientation.w = rot.getW();
    waypoint.pose.orientation.x = rot.getX();
    waypoint.pose.orientation.y = rot.getY();
    waypoint.pose.orientation.z = rot.getZ();
    waypoint_list.push_back(waypoint);
    seq++;
    //std::cout << waypoint << std::endl;
}

//Convert from "std::list<geometry_msgs::PoseStamped>"" to geometry_msgs::PoseArray
std::list<geometry_msgs::PoseStamped>::iterator iter = waypoint_list.begin();
geometry_msgs::PoseArray array;
array.header.frame_id = "";
array.header.stamp = ros::Time::now();
array.header.seq = 1;
while(iter != waypoint_list.end()) {
  array.poses.push_back(iter->pose);
  iter++;
```

```
    }
    res.list = array;
    res.size = size;
    //std::cout << res.list << std::endl;
    ROS_INFO("Successfully sent waypoints to client");
    return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "waypoint_list");

  ros::NodeHandle n;
  ros::ServiceServer service = n.advertiseService("waypoint_list_server", readFile);
  ROS_INFO("Ready to retrieve waypoint list.");
  ros::spin();

  return 0;
}
```