Performance Optimization of Swarm Algorithm and Sensor Data for NASA Swarmathon Competition

Juan Aguilar, Ann Blanchard, Adam Sibiski, Jason Soto, Scott Jagolinzer, Sabri Tosunoglu

Department of Mechanical and Materials Engineering
Florida International University
10555 W Flagler St
Miami, FL 33174

jagui080@fiu.edu, ablan115@fiu.edu, asibi003@fiu.edu, jsoto103@fiu.edu, sjago001@fiu.edu, tosun@fiu.edu

ABSTRACT

This paper contains the methods used by students of Florida International University to optimize an autonomous robotic code to ensure the maximum amount of resource retrieval. Specifically, this paper deals with the ROS operating system and how each of the robot's components can most effectively communicate with each other as well as streamlining communications between robots. The process was carried out by analyzing various portions of the code to obtain marginal increases in performance. This resulted in a faster and more effective overall search and retrieval functioning of the swarm robots.

Keywords

Swarm Robots, Swarmathon, Robot Operating System, Performance Optimization, Sensor Data, Accuracy, Algorithm.

I. INTRODUCTION

To make advancements in the field of space exploration using cooperative robotics, the National Aeronautics and Space administration (NASA) has launched the second annual Swarmathon. It is a competition in which universities nationwide are challenged to use a set of autonomous rovers to explore and collect resources randomly distributed in a large area with possible obstacles. This collaboration allows much quicker resource retrieval rates, ultimately increasing the opportunity for large scale exploration. The competition simulates exploration of an unknown area where humans either cannot reach or cannot survive. It requires each team to develop a unique search algorithm for the rovers to assess the area and collect possible useful resources. For the competition's preliminary stages, teams are allowed three robots to search the area for resources which are represented by April Tags placed on cubes. Because the location of the cubes is unknown, the robots must work together to gather the most possible resources in the least amount of time. Robots are tasked to autonomously scan the area and utilize the onboard camera to detect these tags and a sonar detection systems to detect upcoming obstacles in the robot's path. A claw is then used to pick up the tag and return it to a central collection area in the quickest possible manner.

1.1 The Robot

Designed and provided by the Moses Biological Computation Lab at the University of New Mexico, the rovers, or Swarmies, are small robotic vehicles measuring approximately 30 cm x 20 cm x 20 cm. [1] As seen in figure 1 below, they are equipped with claws for the collection of resources. They also include ultrasound distance sensors, a webcam, a WiFi antenna, and a GPS system to allow the robot to navigate, search for the cubes, and locate the base area.

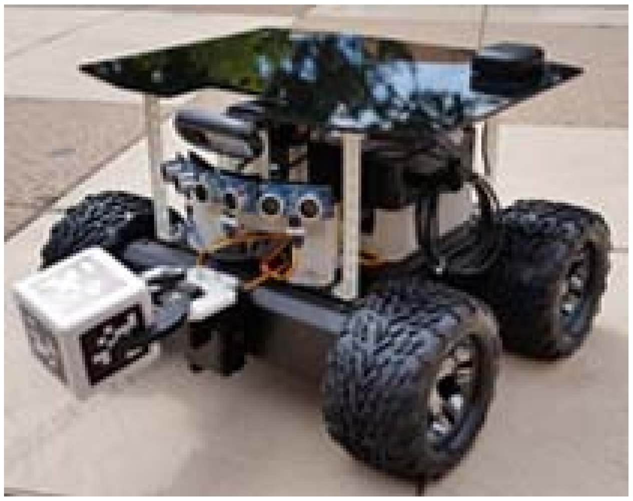


Figure 1. Swarmie 2017 with an Added Claw Feature to Collect the Resources [1]

1.2 Robot Operating System

Inspired by the biological conduct of group functioning insects such as ants and termites, swarm robots utilize algorithms that adjust these behaviors into error tolerant, scalable, and flexible robot foraging strategies in varied and complex conditions. [1] Since the robots' hardware cannot be changed, the optimization of resource collection must be carried out through manipulation of the code. The robots rely on the Robot Operating System (ROS) to operate and execute all the independent functions of the robots. The purpose of the ROS operating system is essentially to compartmentalize the various tasks to be completed by the robots. These tasks are known as packages which control different components of the robot. ROS also allows for the packages to communicate with each other using libraries known as 'topics'. These topics store information that is published to them from packages. Then, a package can subscribe to the topic to receive information from said topic. Each of the robot's functions are separated into their own code but rely on ROS to communicate the information from package to package. For instance, in case one of the robot's sensors detects an obstacle, this is communicated to the mobility package, in charge of the robots' movement, which then uses this information to avoid the obstacle. Editing these packages allows the robot to operate using different logic. The logic used in the packages determines the efficiency at which the robot operates.

1.3 Project Overview

Taking advantage of ROS's compartmentalization is key to optimizing the search for resources. Because all tasks are compartmentalized into packages, these packages can function independently of each other. Optimizing ROS and editing the packages, written in C++, can have a beneficial effect on the performance of the search algorithm of the robots. It is important to streamline the exchange of information between packages as well as between robots. This streamlining process is carried out by editing the topics that the packages publish and subscribe to. Ideally, there is a minimum number of topics that should be used by packages in the operating system. Finding a balance between reliability and time efficiency will be key to successfully competing in NASA's Swarmathon competition.

In summary, this work makes the following contributions: (1) optimize the code for a more efficient performance of the robot; (2) locate useful sensor data and compile it into a single location for ease of use; and (3) improve resource retrieval rate by ensuring precise positional awareness, proper object identification, and successful communication.

II. LITERATURE SURVEY

Autonomous robots can be used to explore a variety of unfamiliar environments. Information can be obtained depending on the functions and components of the robot. These robots are necessary in exploring areas that cannot be accessed by humans, such as caves, deep oceans, and even other planets. NASA

Figure 2. Spirit MER-A Rover

developed two rovers for exploration on Mars. One of the rovers, named Spirit, was launched from Cape Canaveral in 2003. Unfortunately, the rover ceased mobility after entering soft soil. Data was still collected for about a year from the stationary position; however, the rover was determined to be irrecoverable. In 2010, the rover ceased communication and the mission was declared complete. Figure 2 shows the rover being tested inside a lab at NASA.

One month later, NASA launched a second rover which has experienced a much higher level of success. The second rover, named Opportunity, can be found below in Figure 3. The rover has been exploring mars since January of 2004 and is currently still functioning to share data with scientists and engineers on Earth. The rover has traveled about 24 miles and has proven to be very effective. Although the rover has accomplished much, the use of multiple robots can accelerate the rate of exploration and significantly increase the amount of area covered.

Figure 3. Opportunity MER-B Rover

Swarm robots have proved to be a more efficient and inexpensive alternative to solitary space robots for In-situ resource utilization (ISRU) efforts. For instance, "20 Swarmies can travel and search 42 km of linear distance in 8 hours without recharging which is the distance covered in a marathon and the same distance traveled by the Mars exploration Rover Opportunity in 11 years." [1] This is largely due to the cooperative aspect of their foraging algorithm. Though inexpensive, they are more robust, flexible, and scalable than monolithic robots operating alone. [1]

This project builds on important prior works done by the FIU Panther Swarm Team for the first annual NASA Swarmathon. The first design of the Swarmie included sensors, a camera, a GPS system, and a Wi-Fi antenna to communicate. The robots would scan an area in search of tags which represent a resource or object. After observing a tag, the robot would return then to a central location to virtually deliver the "resource." A search algorithm that relies heavily on the compass and ultrasound data was vital for the robot to keep track of the nest position. To allow the rovers to successfully collect and return the tags, four main mode were implemented. As shown in figure 4, they are broken down as runway (green), position (blue), sweep (orange), and return (red) [2]. "The runway is an east to west path along the x-axis that passes through the nest. The position mode assigns a y-coordinate value to the rover and has the rover either moving north or south to get to the desired y-coordinate. The sweep mode is a west to east path that has the rover looking for tags as it moves back across the arena. The return mode has the rover moving north or south to return back to the x-axis so that it can go back to the runway mode." [2] However, the algorithm was not proved to be successful during the physical testing because it made use of the "sonar reading in conjunction with the walls to track and reset the positional data of the rover." [2]

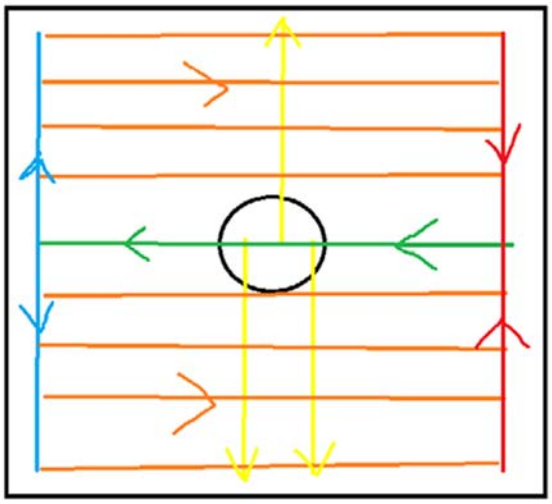


Figure 4. Diagram of Search Algorithm used for NASA's Swarmathon Competition 2016 [2]

The robots used in this competition implement the ROS operating system. This system allows for the compartmentalization of the various tasks the robot must perform. The robots various task, like moving and detecting tags for example, will all have individual codes that are then compiled through ROS and can function independently of each other. To perform all tasks correctly and in a timely manner however, the individual codes, known as packages, must be able to communicate between each other and between the robots. This communication is carried out through a messaging system comprising of topics that can be published to or subscribed to. [6] These topics work as a message board where pertinent information is published for all packages to use. The mobility package, for example, is responsible for the movements and exploration of the robot and depends on the obstacle detection package to alert it of possible obstacles in the robots path. The mobility package then uses this information to take the necessary steps to avoid a collision. For this scenario, the obstacle detection package will detect an obstacle and publish this information to the pertinent topic. The mobility package, who is subscribed to this same topic, notices that there is an obstacle and avoids it. It is in this same manner that all packages communicate to allow the robot to function effectively.

When the code has been compiled and the robot is operational it is important to verify that the code is working in the manner it was designed to. For this, Gazebo is used to simulate the competition. The Gazebo simulator allows the user to simulate both stages of competition with options for 3 robots or 6 as well as variable field sizes. The simulator also has options for changing the distribution of tags dispersed throughout the competition field. These simulations are crucial for understanding how the code will function during competition and how to make adjustments to optimize the robots' functioning. It is also important to experiment with the introduced error in simulation. The larger the error the more faithful to the real world the simulation will be so experimenting with this variable can be very beneficial for competition. It is also possible to adjust the speed of the simulation. This is a very useful tool because you can increase the number of simulations carried out in each time period. Simulations however are no substitute for a real-life test. These tests will help to finalize the code and ensure that it works as desired.

III. MOBILITY OPTIMIZATION

The optimization process was carried out with the intent to increase performance without significantly altering the code. Various aspects of the program such as obstacle detection and mobility were critically analyzed to determine the most efficient method in which the swarm robots can explore the unfamiliar environment. [5] Marginal improvements in different algorithms within the program established faster overall functionality which in turn results in a quicker resource retrieval rate.

The search method implemented for the code is a preset pattern in which the swarm robots move in 1 meter increments in the X direction and then after in the Y direction. A representation of this search is shown in Figure 5, which also displays the competition area. This pattern allows for the rovers to quickly return to home base after locating a resource. After each step, the rover completes a 360 degree scan in search of the tags. If one of the tags are located, the robot will collect the tag using a mechanical claw. The swarmie will then return to the home location in one motion which will sum the displacement in the X direction and Y direction to create a single step. There are tradeoffs when designing a search pattern. This structured code will cover the entire area, but unfortunately, a raking pattern will increase the time it takes to complete scanning the area [3].

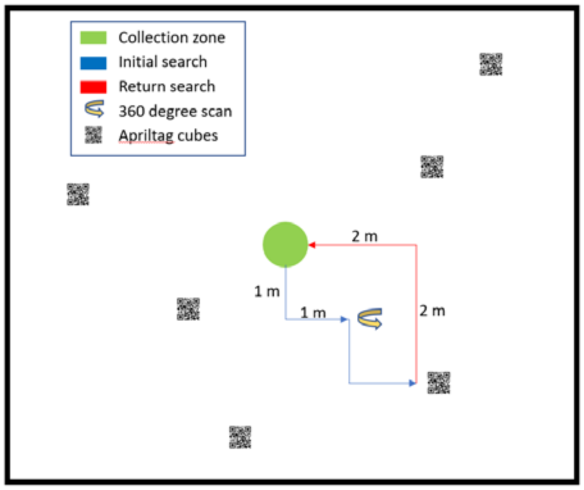


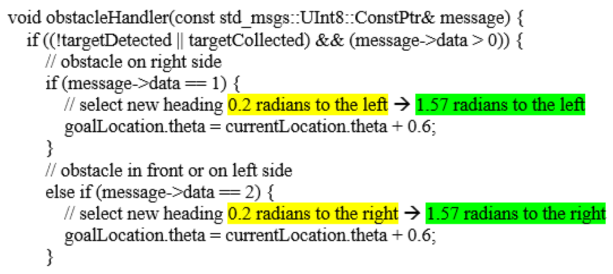
Figure 5. Diagram of Incremental Search Pattern

Originally, the code called for the rover to orient itself using local positioning and a compass; however, a preset command proved to be much more effective. When the swarm robot then returns to search again, rather than trying to determine which direction it is facing, it will rotate 180 degrees and then proceed to search. It will follow the same initial path without stopping to scan areas that have already been covered. Once the swarmie reaches an unexplored location, it will then continue to scan and proceed in

1 meter increments until another tag is located. A diagram of the search pattern can be found below.

IV. OBJECT DETECTION

Due to the frequent collisions experienced by the swarm robots, it was necessary to focus on improving the robots' response to walls and other swarm robots. The code initially prompted 0.2 radian (~11.45 degree) turning increments when an object was detected by the ultrasound sensors. This turning increment was increased to 1.57 radians (90 degrees) to decrease the time it takes the rover to return to its search for resources or tags. This change within the code is show in Figure 6. Utilizing 90 and 180 degree turns proved to be very effective because they complement the square, stepping motion of the search pattern.



```
void obstacleHandler(const std_msgs::UInt8::ConstPtr& message) {
  if ((!targetDetected || targetCollected) && (message->data > 0)) {
    // obstacle on right side
    if (message->data == 1) {
      // select new heading 0.2 radians to the left → 1.57 radians to the left
      goalLocation.theta = currentLocation.theta + 0.6;
    }
    // obstacle in front or on left side
    else if (message->data == 2) {
      // select new heading 0.2 radians to the right → 1.57 radians to the right
      goalLocation.theta = currentLocation.theta + 0.6;
    }
```

Figure 6. Obstacle Avoidance Algorithm Improvement

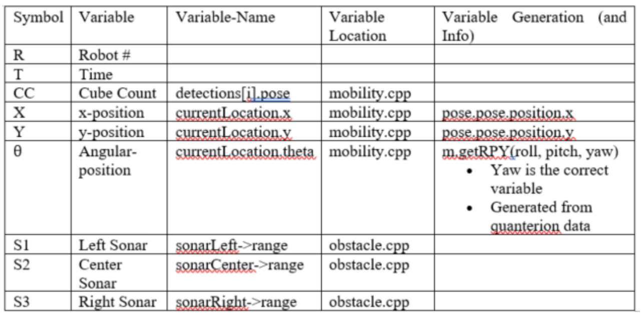
A goal of proper object identification by distinguishing the difference between robots and walls was not achieved. Due to the sensors being placed on the same axis, the robot is unable to distinguish between objects based on height. To compensate for this, future experimentation and competition could implement modular sensor placement. This would provide the ability to place one or multiple sensors on a higher or lower axis. This would allow for one of the sensors to pass over any rover but not a wall. Distinguishing between a robot and wall is important because the response of a rover may not be the same for both. Optimizing the response based on which object is interfering with the robot's motion could result in a decreased search time. For example, if both swarm robots approach each other and turn 90 degrees to the left and right when facing each other, a large portion of the grid may go unexplored for some time.

V. VARIABLE ANALYSIS

The goal of this section was to make the sensor data easy to manipulate and store. To accomplish this, the data must be organized into an array format which updates at regular intervals or continuously. These arrays can then be exported to a text file where they can be saved as a matrix for further analysis. A theoretical array containing the desired raw data is shown in Equation.1 below. Table 1 contains the definitions for the data stored in the array along with information concerning the variables name, location, and how it's generated. After searching the code it was observed that locating the source of the data was much more difficult than observing where it was used. Due to this it is assumed to be easier to extract the sensor data from the current variable locations by publishing the data straight out of the .cpp files.

$$(R\#, T, CC, X, Y, \theta, S1, S2, S3)$$

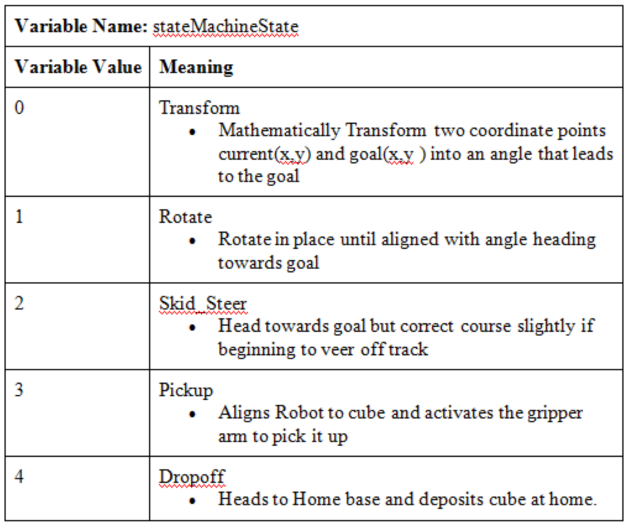
Table 1. Identified Sensor Data



Symbol	Variable	Variable-Name	Variable Location	Variable Generation (and Info)
R	Robot #			
T	Time			
CC	Cube Count	detections[i].pose	mobility.cpp	
X	x-position	currentLocation.x	mobility.cpp	pose.pose.position.x
Y	y-position	currentLocation.y	mobility.cpp	pose.pose.position.y
θ	Angular-position	currentLocation.theta	mobility.cpp	m.getRPY(roll, pitch, yaw) • Yaw is the correct variable • Generated from quanterion data
S1	Left Sonar	sonarLeft->range	obstacle.cpp	
S2	Center Sonar	sonarCenter->range	obstacle.cpp	
S3	Right Sonar	sonarRight->range	obstacle.cpp	

Differing from the Sensor data is the Robot number (R#), a value used to identify data generated by a specific robot. According to previous Swarmathon competitors there does not exist a way differentiate the robots without pulling the IP address, which is not allowed. Due to this a method is being developed to assign a ranking to each robot which would serve as an identification number. While the communication of the data has not been worked out the logic portion of the code is currently under development. To distinguish the robots each shall generate a random number, save the variable, and proceed publish the result to the message board. All of the robots would be subscribed to this data and receive an array of random variables when the announcing process is completed. The robots would then compare their individual saved values with the list and determine their rankings correspondingly. The robot with the largest number will assign itself the number "1"and subsequently each robot will number itself depending on the "rank" of their number. If two randomly generated numbers are identical the robots will be directed to generate a fresh set of random numbers and begin the comparison process again. This data can theoretically be used for robot specialization and task delegation, however its primary function will be allowing operators and the robots themselves distinguish individual robot data from the memory matrix.

Table 2. Description of State_Machine States in Mobility.cpp



Variable Name: stateMachineState	
Variable Value	**Meaning**
0	Transform • Mathematically Transform two coordinate points current(x,y) and goal(x,y) into an angle that leads to the goal
1	Rotate • Rotate in place until aligned with angle heading towards goal
2	Skid_Steer • Head towards goal but correct course slightly if beginning to veer off track
3	Pickup • Aligns Robot to cube and activates the gripper arm to pick it up
4	Dropoff • Heads to Home base and deposits cube at home.

To obtain sensor data we began by looking into the “mobility.cpp” file as this code can be considered the main program for the swarm robots. After analyzing the code the base behavior of the swarm bots was identified. The robots operate on 5 basic modes called “State_Machines” where the variables dictate which mode (specified behavior) to follow.

Within this file the usage of multiple desired data points was discovered, these being X-position, y-position, and orientation. The default swarm program uses these variables to direct its movement. By polling its current position (Point A) and setting a goal (point B) the robot determines two points on the coordinate plane. The robot then performs some simple trigonometry to find the angle of the line connecting the two points. By reading its current orientation the robot determines whether it needs to rotate to match its orientation to the direction of the goal. Once the deviation in angles falls below a certain threshold the robot breaks out of the aligning program and enters the driving program where it heads to the destination at a fixed velocity. As long as the angle remains below the threshold the robot will not break into the aligning program, but instead use minor adjustments to maintain on course.

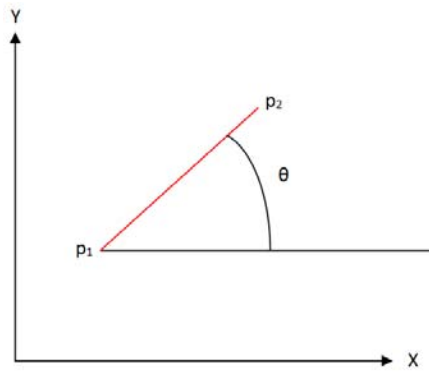


Figure 7. Swarm Bot Alignment Geometry. (Credit David Heffernan)

The next piece of code reviewed was the “Obstacle.cpp” file. The code operates by subscribing to data generated by the sonar sensors, analyzing it, and determining an operating state. By checking which sonar’s are detecting obstacles it specifies one of four obstacle modes as the current state for the robot. Once the correct mode has been determined (See Table 3) the data is published and used by the “mobility.cpp” program. . The format in which data is published follows the format “obstaclePublish.publish(obstacleMode)” while “Obstacle.cpp” operates is that it takes in sonar data then analyzes it to make a decision between four states. It then publishes the corresponding state to be used by other programs. The format in which data is published appears to be as follows.

“obstaclePublish.publish(obstacleMode)”.

To subscribe to the data, “mobility.cpp” first announces

“ros::Subscriber obstacleSubscriber;”

And then later in the code appears to set a variable equal to the subscribed data.

“obstacleSubscriber = mNH.subscribe((publishedName + "/obstacle"), 10, obstacleHandler);”

Table 3. Description of Obstacle Modes in Obstacle.cpp

Variable Name: obstaclemode.data	
Variable Value	**Meaning**
0	No collision
1	Collision on Right Side
2	Collision in front or on Left side
4	Block lifted in front of center ultrasound

“Obstacle.cpp” operates is that it takes in sonar data then analyzes it to make a decision between four states. It then publishes the corresponding state to be used by other programs. The format in which data is published appears to be as follows.

“obstaclePublish.publish(obstacleMode)”.

To subscribe to the data, “mobility.cpp” first announces

“ros::Subscriber obstacleSubscriber;”

And then later in the code appears to set a variable equal to the subscribed data.

“obstacleSubscriber = mNH.subscribe((publishedName + "/obstacle"), 10, obstacleHandler);”

VI. CONCLUSION

It was necessary to compare various search patterns to determine which would be the most effective. Although the rake covered the entire area, it was important to experiment with a spiral and stepping search pattern to determine which would cover the most area in the shortest period of time. This would allow for the maximum number of theoretical resources retrieved by the autonomous swarm robots.

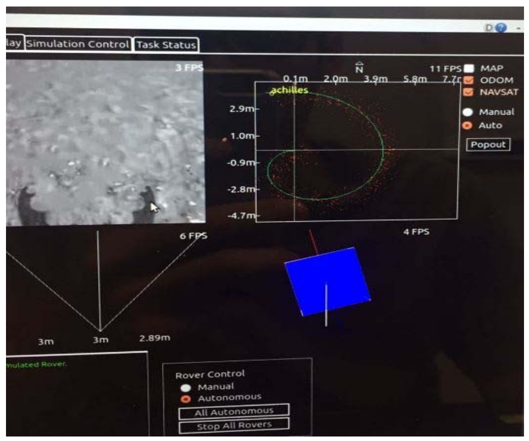


Figure 8. Spiral Search Simulation

The spiral method (shown in figure 8 and 9) proved to be effective in scanning the area; however, it was very difficult to orient the robots. Due to noise and low accuracy GPS tracking, the robots would often misalign themselves making it difficult to return to the goal zone. Using higher quality sensors and hardware could provide much higher performance and programmability.

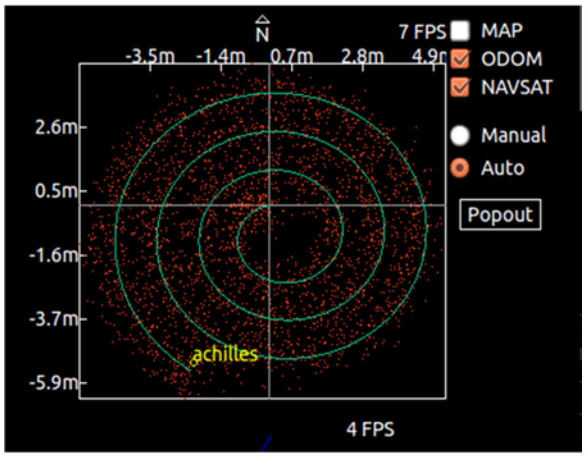


Figure 9. Map Readout of a Spiral Search Simulation

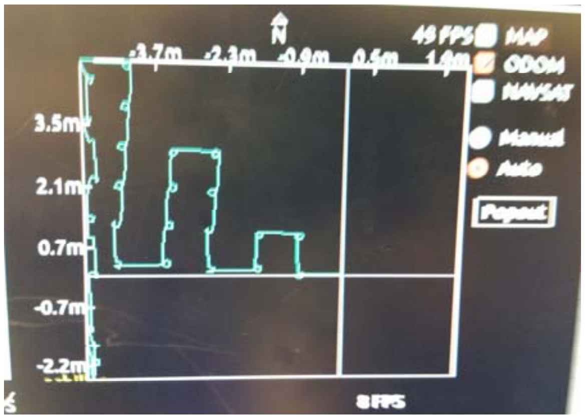


Figure 10. Simulation Result of One of the Zone's Pathing with the Little Circles Representing the 360 Sweep at Each Space

A more structured search method would then be created to eliminate any errors associated with the robots ability to position itself. A chessboard pattern was implemented to allow for the rover to return home after collecting resources by summing the steps it has moved away from home base. The 1 meter increments would allow for the area to be divided up into simple sections for the rovers to explore. After each step, the rover would complete a 360 degree scan and continue searching for resources. Figure 10 displays is an image of the simulated path.

VII. ACKNOWLEDGMENTS

A special thanks goes to the NASA Kennedy Space Center and all of the volunteers for hosting the Swarmathon competition and inviting Florida International University to participate. The authors also extend their gratitude to the Robotics and Automation Laboratory and the Department of Mechanical and Materials Engineering at Florida International University in Miami, Florida for providing guidance, lab space and equipment for testing.

VIII. REFERENCES

[1] National Aeronautics and Space Administration, "Learn more," 17 April 2017. [Online]. Available: http://nasaswarmathon.com/about/.

[2] FIU Panther Swarm Team, "Development of FIU Panther Swarm Algorith for NASA's Swarmathon Competition," Florida International University, Miami, Florida, 2016.

[3] Hoff, Nicholas, Robert Wood, and Radhika Nagpal. "Distributed colony-level algorithm switching for robot swarm foraging." Distributed Autonomous Robotic Systems: 417-430, 2013.

[4] Richard, William K., and Stephen M. Majercik. "Swarm-Based Path Creation in Dynamic Environments for Search and Rescue." Proc. of Genetic and Evolutionary Computation Conference, Philadelphia, PA, USA. N.p., 11 July 2012. Web. 12 Feb. 2016.

[5] Shoutao, Li, Lina, Li, Lee, Gordon and Zhang, Hao. "A Hybrid Search Algorithm for Swarm Robots Searching in an Unknown Environment." 11 Nov. 2014. Web. Feb. 2016.

[6] Stolleis, Karl. "The Ant and the Trap: Evolution of Ant-Inspired Obstacle Avoidance in a Multi-Agent Robotic System." 26 Jun. 2015.