Development of Robot Swarm Algorithms on An Extensible Framework

Michael Backus James Fisher Samuel Ndiwe Nathaniel Spindler Bruce Osnoe Tyrone Thomas-Wesley
Department of Mathematics and Computer Science, College of Arts and Sciences
Fayetteville State University, Fayetteville, NC 28301
Research Advisor & Reviewer: Dr. Sambit Bhattacharya
Email: sbhattac@uncfsu.edu

***Abstract*— Swarm intelligence for robots is inspired by observation of how homogenous collections of animals behave in nature to succeed in finding food and avoiding predators. Swarm robots usually lack centralized control to determine each robots individual behavior, however global behaviors can emerge through many local interactions, which are simple in nature. Studies show that simple rules executed on the individual robot can explain complex group behaviors and it is sufficient to support only local sensing and communication. The advantages of swarm intelligence are robustness at the level of the group where individual failure is not a significant problem; individual behaviors are easy to implement, and the approaches are scalable since the control mechanisms do not depend on the number of individuals in the swarm. We provide background information on swarm intelligence and describe an ongoing project at Fayetteville State University (FSU) that leverages hardware and extends software, developed at the University of New Mexico (UNM), USA and the National Aeronautics and Space Administration (NASA) agency of USA. The hardware consists of robots called swarmies. A swarmie is a small robotic vehicle with a webcam, a GPS system, sensors like IMU, ultrasonic obstacle detector; a Wi-Fi antenna for wireless communication and an on-board computer. The objective of the project is to program the Swarmies so they can communicate and thus execute cooperative april-tag collection autonomously. The software is a ROS (Robot Operating System) controller framework for the Swarmie robots. We were able to improve the swarmie's behavior to search a space more effectively and utilize computer vision methods.**

Keywords- swarm; robotics; artificial intelligence; search; ROS; swarmies;

I. INTRODUCTION

Each kilogram launched into Earth orbit or beyond costs fuel which in turn has made space exploration very expensive. Robots, such as the autonomous rovers *Spirit* and *Opportunity* that were launched in 2004 to explore the surface of Mars, employ a host of expensive hardware and sensors. There are many ways to lower the cost and risk of an extended, human stay on a planet or celestial body such as Mars. One method is to gather materials in the environment that can be converted to water and fuel. A cost effective, safe way to do this is to utilize swarm robotics. In this way we use many robots instead of just one or two, thus, improving our chances for success and lowering risk in such missions. As a benefit, the problem becomes easier and quicker to solve as we add more swarm robots, or swarmies. These and other reasons are what stands behind the motivation to study and improve swarm robotics from a space exploration point of view. Global powers in space exploration along with corporations such as are both cooperating and competing in advancing the tools and technology for the next century of space exploration. In the coming decades a focus has been set with a return to and permanent establishment on Earth's moon along with manned missions to Mars. Our goals in this project were to implement a swarming algorithm and use different search strategies.

The "NASA Swarmathon" is sponsored by NASA's MUREP Program

A. Background

Ever since the 1980s, swarm robotics has become a large research field. As it becomes more developed, the advantages of using a swarm robotics solution become more clear. Swarm robots offer a cheaper solution because each agent only needs to be able to do a specific task or so according to Yogeswaran [8]. Utilizing a swarm approach, we get additional features such as:

- *Parallelism*, which allows robots to divide tasks.
- *Robustness*, because if one robot or many fail, progress can still continue.
- *Scalability*, because as the number of swarm robots increase, problems are solved faster.

B. Swarm and Search Techniques

This work is a follow up to the University of New Mexico's extensive development and application of swarm robotics. Their iAnt project used algorithms that mimic the process of natural selection to adaptively produce efficient search behaviors. Their research served as a project template for what would later become the NASA Swarmathon. The NASA Swarmathon is a US initiative to expand knowledge on cooperate robotics and help revolutionize space exploration.

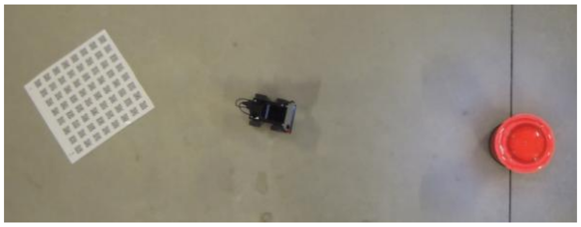


Figure 1. An iAnt returns to the base after scanning a tag. The University of New Mexico's Swarm Robotic Research Group collaborted with NASA to create the Swarmathon, the first ever national swarm robotics competition.

II. Related Work

Many algorithms exist to replicate the natural, social behavior of flocks, herds, and swarms. Understanding the unity and flow of movement that natural swarms of ants, birds, bees, butterflies, bats, fish- and many more organisms exhibit is a goal of swarm intelligence [8]. Search techniques tend to use a random, biased search as a starting point but from using heuristic functions, A* search, or with evolutionary and other learning algorithms, a better search technique can be acquired.

A. *Biological Inspiration*

Because of their limited ability to communicate with each other, ants and other social arthropods/insects have become a focus of swarm robotics. Their tasks usually involve simple goals such as searching, retrieving, dropping off, and so on. Ants have trail pheromones that directly attract and determine the behavior of other ants nearby, such as that achieved by Cazangi et al. and other groups of researchers [9].

B. *Particle Swarm and Ant Colony Optimization*

The Particle Swarm Optimization behavior belongs in the broader field of computational intelligence, and the more specific Ant Colony Optimization serves as a base algorithm that of which includes many different variations [4]. Members in the particle swarm bias their interest toward areas in the environment that have yielded more success. They do this organized by only a few different motion or abilities usually programmed into a state machine. After each update, the goal function is sampled. The number of particles or agents used is usually kept low, around 20 to 40. Upon each update, an agent's velocity is updated using:

$$v_i(t+1) = v_i(t) + (c_i \times rand() \times (p_i^{best} - p_i(t))) + (c_2 \times rand() \times (P_{gbest} - p_i(t))) \tag{1}$$

where $v_i(t+1)$ is the velocity for the ith particle, the weight coefficients are represented by the c values, c_1 and c_2, and the best positions is given by $p_i(t)$, which is the ith particle's position at *t*, for time. P_{gbest} is the best position known to all particles. The particle's position is updated by:

$$p_i(t+1) = p_i(t) + v_i(t) \tag{2}$$

C. *Heuristic and A* Search*

A heuristic function attempts to solve a problem quicker than the usual methods such as brute force. A heuristic function defines the cost of solving a problem, such as finding the shortest path to a city by which roads to travel. By sorting different alternatives among search algorithms and using a branching structure based on currently available information, it is then decided which pathway to continue down to. A heuristic function may not always give the best answer, but approximate it. Therefore a heuristic function may not find all solutions as they only find one, while some may only find a solution that is slightly better than traditional methods.

A * or "A star" search is a special algorithm that uses tree or graph traversals to find an efficient path between many nodes. A star search qualifies as an informed search algorithm because it finds a solution by checking possible paths to the goal state that produces the least cost. A tree of potential "paths" are created and each path is expanded every step, with the end goal being eventually reached. Usually a priority queue known as a *fringe* is used to determine which cost nodes to continue down the path. “A star” search algorithms can be utilized in many ways, such as determining the best path to a goal while including known obstacles in an agent's path.

III. Methods

A. *Quadtree Search*

There are many strategies to create an informed search algorithm and one of them is Quadtree search. This spatial indexing technique uses a collection of nodes that represent the different spaces of the area being indexed[12]. Each quadrant has exactly four children. We can recursively determine how small a space we want to sample from a larger initial space.

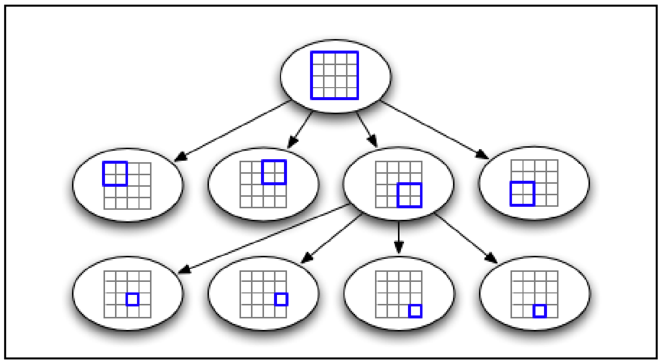
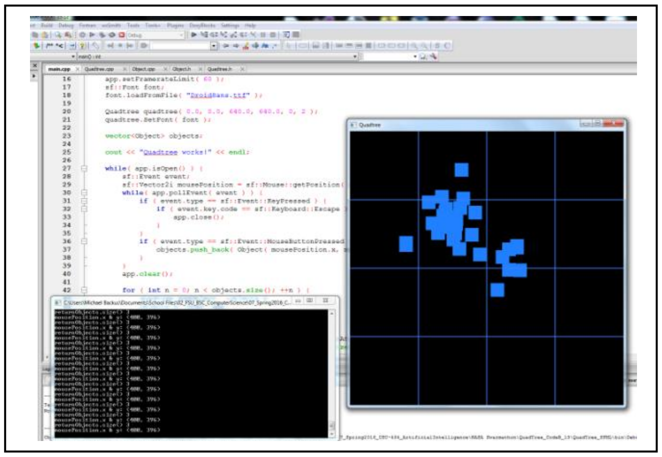


Figure 2. The tree structure that Quadtree utilizes.



```
app.setFramerateLimit( 60 );
sf::Font font;
font.loadFromFile( "DroidSans.ttf" );

Quadtree quadtree( 0.0, 0.0, 640.0, 640.0, 0, 2 );
quadtree.SetFont( font );

vector<Object> objects;

cout << "Quadtree works!" << endl;

while( app.isOpen() ) {
    sf::Event event;
    sf::Vector2i mousePosition = sf::Mouse::getPosition(
    while( app.pollEvent( event ) ) {
        if ( event.type == sf::Event::KeyPressed ) {
            if ( event.key.code == sf::Keyboard::Escape
                app.close();
            }
        }
        if ( event.type == sf::Event::MouseButtonPressed
            objects.push_back( Object( mousePosition.x,
        }
    }
    app.clear();

    for ( int n = 0; n < objects.size(); ++n ) {
```

```
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
returnObjects.size() 3
mousePosition.x & y: (400, 396)
```

Figure 3. Quadtree implimentation and testing with the SFML GUI.

This example implementation of Quadtree using the above GUI has two subdivisions. The objects placed represent places that the rovers have been. Each quad stores this information which can then be used for more intelligent searching. Quadtree will be put into mobility and used to improve the overall random searching by creating a bias to visit less visited areas, which will provide an improved coverage over time.

IV. Hardware and Software

The hardware used for this project were the three swarmie robots provided by NASA and the Robot Operating System, or ROS, which runs on the Ubuntu Linux version 14.04 open source operating system distribution. The swarmie robots, or rovers, are hand crafted robots of regularly-available materials and electronics or devices.

The Robot Operating System, or ROS, is a collection of software frameworks for robot software development. ROS is an open-source yet robust robotics platform that has existing support for a wide range of robot designs. The user can extend it to support other designs and additional features when necessary. ROS uses a system of topics that publish messages that can then be listened to by subscribers. The open nature of ROS makes it a perfect platform for robotics, whether here on Earth or to utilize in future space exploration and missions to other worlds.

A. *Anatomy of a Swarmie*

The body of a swarmie is assembled from a collection of 3D printed and laser cut parts. While these parts are custom made for the swarmies, the rest are not. The rovers were shipped to us already assembled by the team at the University of New Mexico and NASA. Other parts include:

- Four DC motors to power the wheels.
- A high capacity 3 cell lithium polymer (LIPO) battery to power all of the onboard electronics and computers.
- Three ultrasonic sensors near the front.
- A standard webcam and a GPS unit.
- An inertial measurement unit or IMU.
- An Intel NUC PC.
- A WIFI adapter.

The Intel NUC PC is running Ubuntu Linux 14.04 with ROS installed. Once the system is powered on and hooked up to a monitor, one can acquire the IP addresses needed to connect to the rover, load code onto it, and begin running search programs. C++ is the programming language we used.

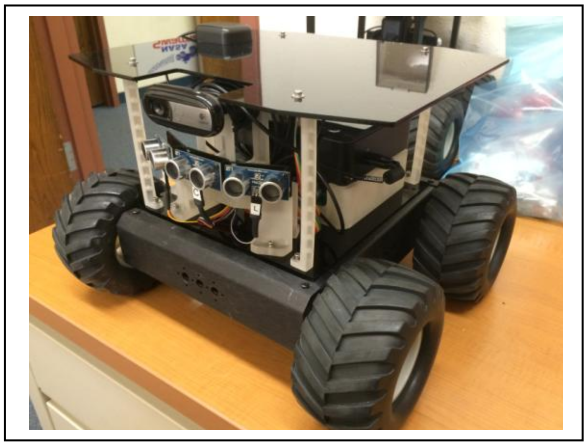


Figure 4. An assembled swarmie.

B. *Visualization with Gazebo and the Rover GUI*

Gazebo is a program that allows the construction and simulation of scenes within a 3d, graphical user interface (GUI). Within the GUI, we can tweak the physics simulation rate in order to speed up or slow down the simulation, as well as perform the basic 3d transformations: move, rotate, or scale on objects within the scene. The scene can be paused or resumed. The simulator provides quick turn around on code testing and allowed us to iterate through versions and tweaks to our code rather efficiently. The "Abridge" module, which controls the lower level hardware on the physical swarmies, is not included in the simulation.

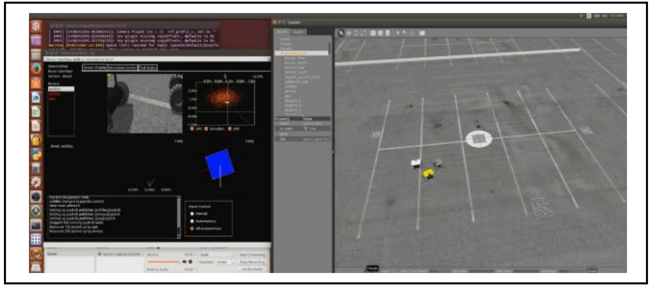


Figure 5. Overview of Rover GUI and Simulation

The Rover GUI was built by the University of New Mexico organizers of the swarmathon and is a central control program for the rovers. We can select the simulation of a preliminary or final round, select a ground texture, and change the rovers to manual or autonomous. Once the simulation is running, we can see the rovers moving and collecting tags. IN addition to this, ROS provides many ways of debugging and looking at data. Topics can be seen by the RQT viewer's Topic Publisher, and the equivalent of "print" statements can be created by using the ROS_INFO_STREAM command, along with previewing these statements in a new terminal by typing in *rostopic echo rosout/msg*. This functionality of both ROS and the Rover GUI helped us dissect how the code was working and better yet, where and how to edit, modify, and to add our own code.

V. PROJECT DEVELOPMENT AND EXPERIMENTS

The first goal was to solve or improve the target detection issues and to create the general swarm behavior of the swarmies. We wanted them to all report in and swarm on a cluster of discovered targets, and to return those closest to the base first and work their way outward. The second main goal was to improve the overall efficiency in searching and obstacle avoidance. When left to their default random search, the swarmies may not even look at an entire quadrant of searchable area, and there were problems with the rovers coming into contact with each other especially around the collection zone. Work and testing on this project first began in the simulator, with a gradual improvement and completion of the swarm code before we did any real testing on the rovers. We then tested the rovers in real life, which lead to the issues present with localization and calibrating the sensors, along with improvements and ideas in searching for tags when none have been currently found.

A. *Mobility Package Modifications*

The mobility package was the heart of where most of our work was done. Within mobility, nearly everything when it comes to the rover's movement resides. Separate modules such as Target Detection sends data on whether a target has been collected to mobility, then a callback function within mobility handles the next step. Moving and rotating were handled by the use of state machines which utilizes a switch statement. The three states given to us were *transforming*, or deciding on what to do next, *translating*, and *rotating*. When the rovers are in autonomous mode, they perform a random search by selecting new goals 50 centimeters in front of them and continue searching for tags. After finding a tag, the goal is set back to the center, and the rovers continue to the new goal. If the angle becomes too large at any given moment that the transforming state is current- which happens multiple times a second, then the state machine falls into the rotate state until that angle is reduced. We modified and created more state machines to add more features as well as more control over the different tasks we wanted the rovers to do.

Some of the custom state machines included "START", "SCANNING" and "BLOBTRACK." The "START" state machine is the initial state machine, and simply makes the rovers turn 180 degrees and travel for a predetermined distance, thereby getting them further from the base instead of having them start their random search right away. "SCANNING" is a hard coded state machine that makes the rovers search for targets or the base and will be discussed further later. "BLOBTRACK" is the state machine that controls the blob detection, which is a computer vision technique we utilized that will also be discussed further later.

We created numerous ROS topics to meet our needs. One of them obtained the number of rovers spawned. This allowed us to determine whether we were in a preliminary round or a final round, thereby determining the amount to scale our searching or other hard coded values. Additional functions such as bubble sort proved very useful.

B. *Target Detection*

Initial work began in target detection and mobility by creating a struct that would store the values of targets detected, their x and y positions and distance from the base. Once target(s) are detected by a rover's camera, one target would be picked up by the rover while the others are stored in a priority queue data structure. The queue in general is an abstract data type is a first-in first-out (FIFO) structure while a priority queue assigns priority to items, which for us was the distance from the base. This information was then broadcasted on a ROS message for other rovers to see. At this point, rovers that haven't found a tag yet would immediately stop their search and go exhaust the source of detected tags by dequeue'ing them until there were no more tags left in that area.

C. *Blob Detection and Training the Dataset*

Computer vision is a field of study that includes methods for acquiring, processing, analyzing and understanding images. One of the best if not the best sensors we had on our rovers were the camera. While only a webcam capable of capturing small videos at a 320 by 240 resolution, this is plenty of data to work with. Blob detection is a computer vision technique that takes a section from an image- usually first converted to grayscale or sampled by luminance values- and flag it as a "blob." For our work, we wanted blobs to be anywhere in the image that could be a cluster of tags, particularly beyond the point of where the camera could not identify a specific tag or pick it up, and use this "blob" as a point of interest for the rover to rotate toward and move closer to. If harnessed to its full potential, this method could make the camera one of the most intelligent sensors we worked with. We would also want to be careful with this, as parking lot lines would also show up as potential blobs.

Binary blobs are those white pixels (or pixels based on a RBG/scalar threshold value) that are grouped together. We can calculate, in pixel space, the area of these blobs and even filter them by color, size, and shape including concave, convex, and by using inertia ratio, whether the shape is more oblong or circular. The options and abilities of blob detection is beyond the scope of this paper, and judging by the number of possible parameters to tweak, we knew that we would have to do a different method other than manual adjustments. This is where training the data set came into play. The images included blank asphalt, clusters of tags near or further away from the camera, parking lot lines, and some with both. We were able to a success rate of 74% with 4 false positives and 300 false negatives out of 941 images. It is a very conservative approach to reduce wasted time chasing pebbles. The accuracy of blob detection can be thrown off by camera exposure changes due to cloud cover, shadows, and specular highlights and reflections from the sun.

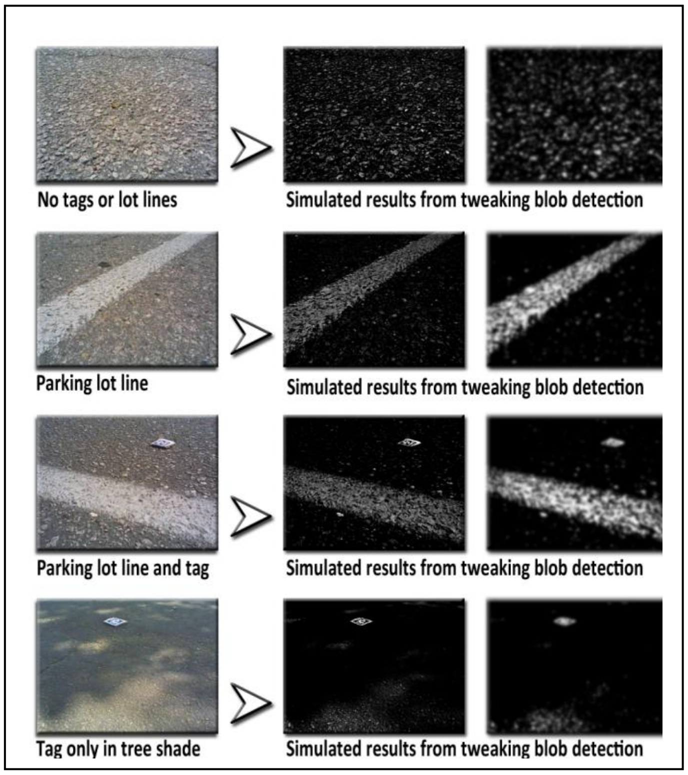

No tags or lot lines
Simulated results from tweaking blob detection
Parking lot line
Simulated results from tweaking blob detection
Parking lot line and tag
Simulated results from tweaking blob detection
Tag only in tree shade
Simulated results from tweaking blob detection

Figure 6. *Digital simulation* of what blob detector should theoretically do if possible to tweak the values perfectly. The white areas are flagged as blobs.

D. *The Scanning and Holding Pattern Concept*

The scanning idea came from looking at some of the work previously done on the iAnts. Swarmies can "look" around once at a goal to see if they have missed a tag within their proximity. Scanning works as a separate state machine that recursively calls itself, while incrementing a variable, then selecting if the rover should rotate right, left, or finish and exit the state. Scanning in this way would prove even more useful if the rover's camera were able to look at the horizon and see changes in the usual terrain pattern, such as asphalt, and tag these as places of interest for other swarmies to look at. We later converted scanning to more of a "spiral pattern" where the rovers went into a pre-determined spiral pattern to both help them find the base or targets they think should be there.

The holding pattern idea is similar to what's used at airports to avoid congestion and collisions. The idea was to have rovers who are all near a tag cluster to take turns going to the base instead of all going at the same time and generating more obstacle calls. Functions were created that *pick the second closest point in an array of four from the collection disk*, which means that larger triangles were formed and less collisions were possible. Rovers that were closest to each other paused and waited briefly for others to enter the holding pattern.

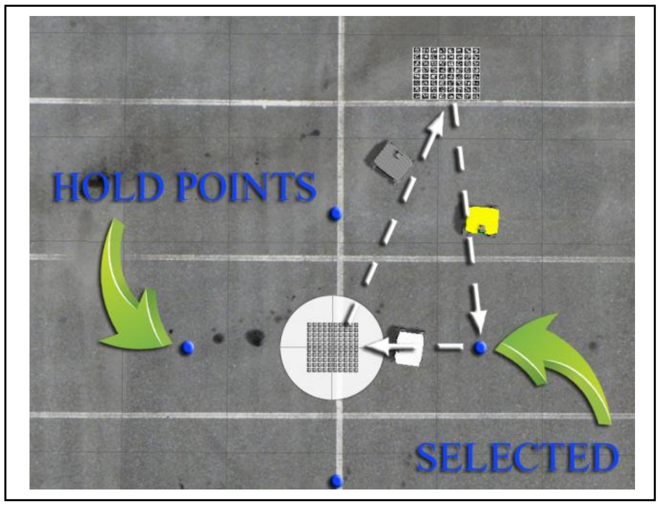


Figure 7. A swarmie holding pattern.

Upon further testing, we realized that the time gained was minimal if not detrimental because in real life the rovers did not have the same localized precision, and therefore it would just create more goals that they would have to follow and more potential ways for them to get lost.

E. *Improving Obstacle Detection*

The obstacle detection package was the least concerning and least problematic out of the code that we began working on. After subscribing to the sonar sensor's range data, the sonar handler function sends an integer value to another callback handler in the mobility package, which in turn decides if the rover will turn left or right. The sonar sensors work better the lower the speed is kept, however, increasing the rover's speed and bumping up the obstacle flagging parameter from the default of 0.4 meters to a higher value seemed to work. The sonar sensors in real life however were very noisy so ended up adding a kalman filter to smooth out the noise, with mixed results.

One potential improvement was to speed the rovers up if they were returning to the base with a tag or if they were returning to pick up a known tag's location in the queue. The only obstacles between the tag collection disk in the center or other tag's location were other swarmies. This means that careful attention and testing had to be done to ensure that rovers would not hit each other at higher speeds. The holding pattern concept, along with checking the distance between rovers at higher speeds is what helped this method work. However, the holding pattern idea was ditched toward the end of development, because we found out that over time, the improvements gained in the simulator did not translate well to real life.

F. *Final Searching: Spiral Scan, Quadsearch and Quadtree*

After our initial 180 degree turn and movement away from the center of the base and a spiral scan for five minutes, we break from that motion and go into a random walk motion. The goal with spiral scan is to cover some space and drop down objects with quadtree. If we find any targets during this time, then we break from spiral scan and random walk and proceed to collect targets until that area has been exhausted.

Quadsearch is a simpler version of quadtree and works in a similar way. Every time the rovers move forward, we increment an integer variable. These numbers can become large after awhile, but not larger than a few thousand. After awhile of not finding a tag, the rovers sort the quadrants they've been in and select the one with the smallest number. They then make a move to go to a random location but within that quadrant, then continue their random search. Given enough time, this method can produce a fairly decent coverage of the map.

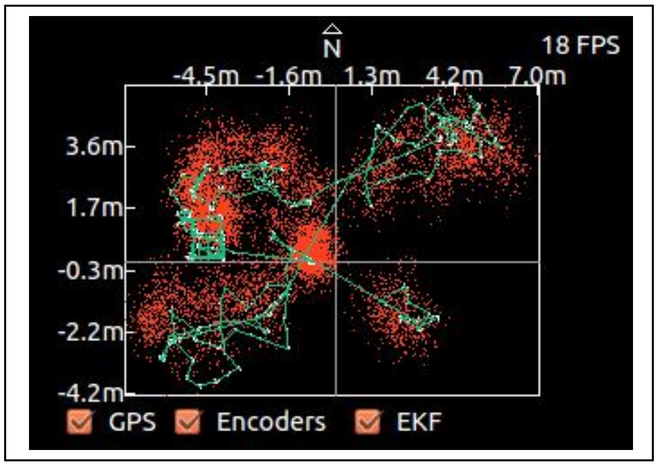


Figure 8. Preliminary round coverage of one rover after 22 minutes in the worst case scenario of not finding a single tag. Each rover would do the same.

For our quadtree search, we simply dropped these imaginary objects down every time the rover translated forward. These are stored in a dynamic array, or vector. Then once the timer got to a certain point where the rovers didn't find anything interesting, we queried the locations two meters to the north, south, east and west. Those areas are then sorted and the one with the smallest number of imaginary objects is then set as the new goal. We then proceed with random search. Had we taken more time to properly weight the timers on these methods, we could have achieved more coverage.

G. *Localization: Wheel Encoders and GPS*

The rovers have an ability to use a mix of both and dead reckoning calculations via the odometry package and data received from the motor or wheel speed. The wheel encoders work by approximating the position in world space where individual rovers have been based on the revolutions of their wheels. While it may seem simple, this is precisely how the odometer works on millions of vehicles and can be quite accurate. Accuracy varies depending on the friction being used and attributes of the encoders. The idea behind localization is simple: For the rover to know where it is at in real life as it does so well in the simulator. This part of the project proved to be the most difficult.

VI. Results and Conclusion

Most of the outcomes that we set out to do were solved or at least tested, but not in the best ways. At first, we started out by doing all of our work in the simulator. We used a queue and ROS topics to broadcast tags already picked up by the rovers and to also let other rovers know where the tags detected were. After much work with the scorekeeper to ensure tags were not picked up again by other rovers, the performance of the swarming rovers were very impressive. The only problem is that the rovers were flagging each other as obstacles too often so that's where the holding pattern came into play. Either way, it seemed that at this point, all we had to do was to create and finalize our better search algorithm to make that aspect of our rover's work a little more intelligent and to see what we could improve with obstacle detection and rover speeds. We were collecting over 200 tags within an hour of simulation time, which wasn't bad. The simulation we had gave us great confidence up to this point in our development.

We figured that once we loaded the code on the physical rovers, then the code would work just fine at first but after awhile drift from the wheel encoders and GPS would occur. Then we would simply have to calibrate the hardware and tweak the values of each hardware device to provide a better match to the simulation, but it was not as easy as we first thought. The one piece of hardware that worked very well was the camera. In daylight conditions, the camera had little to no motion blur present, which allowed the rovers to detect targets with relative ease even at default to moderate speeds. Indoor testing on the camera proved difficult simply because indoor lighting is less intense than outdoor lighting, which forced the camera's sensor to expose for more time each frame, thereby creating more motion blur within the images. But this was understood and expected.

The next issues we had came from the sonar sensors and obstacle detection. The problem here was that these sensors were getting a lot of noise thereby triggering false readings of obstacles closer to the flagging distance, and the rovers were stopping more than usual to correct themselves. Applying a filter to smooth out this noise helped some, but not enough. We had a similar issue with the IMU. In the simulator, the IMU performed perfectly, but in real life, the IMU was among the noisiest of all the sensors. This was throwing off the angle to the goal measurement, thereby also triggering random corrections even when none were needed.

These issues, as annoying as they were would not prevent the rovers from eventually doing what they were supposed to do, they were just being slowed down. It was the localization which comprised of wheel encoders and GPS data that really gave us problems. Without decent localization, it wouldn't matter how good our code worked in the simulator, so toward the end of development we focused more on this. We were able to calibrate the IMU but even so it still gave us trouble. Varying attempts of either merging or using the wheel encoders or GPS to update the home position proved difficult as well. However, in the end, our improvements to localization did work out okay, but one thing is for certain: future improvements while working with the physical rovers must solve the localization issues first, before anything else.

Before continuing on to create better search or swarm algorithms, small teams should focus on completely solving the IMU, Sonar, GPS and odometry issues. A small amount of drift is inevitable from the differing surfaces and level of asphalt quality, but their end results should definitely perform better than ours. Overall our team learned a lot about robotics and some of the issues pertaining to them, but our greatest lesson was that there is much room for improvement, especially in our underestimation of working with the real rovers.

References

[1] A. Engelbrecht, Computational Intelligence: An Introduction, 2nd ed. "Particle Swarm Optimization" University of Pretoria, South Africa. 2007, pp. 322-326

[2] A. Jevtic, D. Andina, Swarm Intelligence and Its Applications in Swarm Robotics. Madrid, Spain: University of Madrid, E.T.S.I. Telecommunication. WSEAS Int. 2007

[3] D. Kampally, R. Annae, S. Kumar. Analysis of Algorithms to Implement Swarm Bots for Effective Swarm Robotics. Hyderabad, A.P, India: Department of Computer Science and Engineering. IJCSET. August 2011, Vol 1, Issue 7, pp 407-414.

[4] J. Brownlee, Clever Algorithms: Nature Inspired Programming Recipes. "Swarm Algorithms" CC/Non-commercial, 2011, pp. 237-274

[5] J. Hecker, "Swarmie User Manual", Quick Start Guide for Physical Robots. University of New Mexico. Github.com, 2015

[6] J.M. O'Kane, A Gentle Introduction to ROS. Columbia, SC: Department of Computer Science and Engineering, 2014.

[7] M. Quigley, B. Gerkey, W. Smart, Programming Robots with ROS, O'Reilly Media, Inc: Sebastopol, CA, 2015

[8] M. Yogeswaran, S.G. Ponnambalam, Swarm Robotics: An Extensive Research Review. Selangor, Malaysia: School of Engineering, Monash University, Sunway Campus, 2010

[9] R. Cazangi, V. Zuben, F. Figueiredo, Autonomous Navigatoin System Applied to the Collective Robots with Ant-inspired Communication. Proceedings of the 2005 conference on Genetic and Evolutionary Computation, ACM, pp 128, 2005.

[10] R. Eberhart, Y. Shi, J. Kennedy, Swarm Intelligence, "The Particle Swarm" The Morgan Kaufmann Series in Evolutionary Computation. Academic Press: London, United Kingdom. 2001, pp. 287-324

[11] S. Russel, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed. "Solving Problems by Searching" Pearson Education, Inc: New Jersey. 1995-2010, pp. 64-108

[12] H. Samet, R. Webber. "Storing a Collection of Polygons Using Quadtree" *ACM Transactions on Graphics.* InfoLAB: July 1985 182-222