A Swarm Control System for the EZ-RASSOR

Daniel Silva[1]

Abstract— **In this project, we create a system for controlling a swarm of EZ-RASSOR mining rovers. The system is responsible for dividing the task of collecting regolith among a group of EZ-RASSOR rovers and handles the management of each rover's battery level, their current objective, and the paths which they follow. The system utilizes a custom scheduling algorithm to coordinate rover tasking and local repair A* path planning to generate energy-efficient paths while re-planning when necessary.The swarm controller is built using the Robot Operating System (ROS), simulated in Gazebo, and functions seamlessly with the open source EZ-RASSOR platform. The source code for this project is publicly available at `https://github.com/FlaSpaceInst/EZ-RASSOR`.**

I. INTRODUCTION

The EZ-RASSOR is a modular version of NASA's full scale RASSOR [1], an autonomous regolith mining rover being developed by NASA's SwampWorks group for deployment on the moon. The primary goal of this smaller scale rover, the EZ-RASSOR, is to recreate the original RASSOR using solely open source technologies and grant public access to its software. In doing so, NASA hopes to enable educational and research institutes around the world to expand upon this work and aid in solving the difficult problem that is autonomous excavation of resources from other celestial bodies.

Prior to this project's expansion of its functionality, the EZ-RASSOR was capable of the following, either autonomously or while being controlled remotely:

1) Roving across slightly treacherous terrain
2) Mining, storing, and dumping regolith with rotating drums
3) Autonomously detecting and avoiding obstacles
4) Localization within an environment using odometry techniques

While these capabilities allow an individual EZ-RASSOR to perform its basic tasks, a large group of rovers had no ability to collaborate efficiently or intelligently in order to achieve a high-level goal. Additionally, things such as rover battery constraints and a varying number of designated dig sites were not taken into account. Therefore, this project aimed to build upon the existing EZ-RASSOR platform by implementing a swarm control and management system. This system enables a swarm of rovers to cooperate in mining regolith from the lunar surface by breaking up tasks into assignments and waypoints for each rover in the swarm. Individual EZ-RASSORs are then able to navigate towards waypoints and accomplish tasks using their on-board autonomous capabilities. The high-level swarm controller also maintains constant communication with each rover, monitoring metrics such as battery levels and current

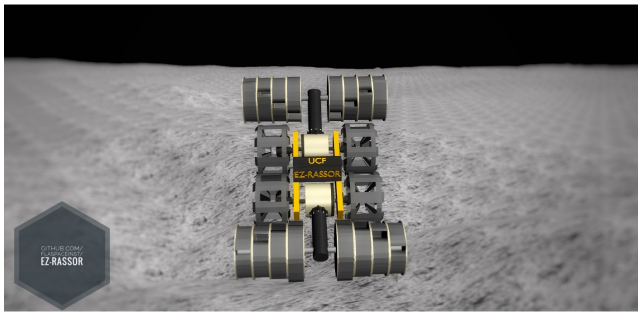


Fig. 1: The EZ-RASSOR rover model in Gazebo.

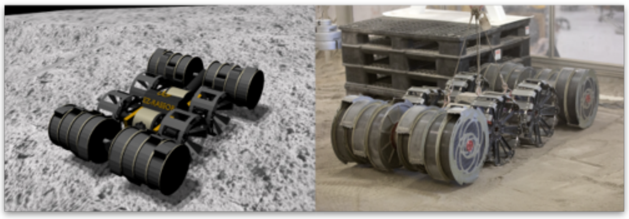


Fig. 2: Side by side of NASA's RASSOR (right) and the EZ-RASSOR in Gazebo (left).

positioning. This allows the system to make informed decisions with regard to rover tasking and routing. For example, the swarm controller is able to recall rovers to charging stations or re-plan rover's paths when necessary. Details regarding the implementation of this system using the Robot Operating System (ROS) are provided in the following section.

Definition 1: RASSOR: Regolith Advanced Surface Systems Operations Robot

II. METHOD

In order to create our swarm control and management system, several subsystems were developed and interconnected. Each of these subsystems make up a small part of the overall swarm controller and rely on one another in order to properly manage the rover swarm. A small description of each of these subsystems are provided below.

Environment Map: The environment map is simply an elevation map of the lunar surface surrounding the rover's initial deployment location, which will be used during path planning. It should be noted that for this project, the lunar environment is assumed to be static. Therefore, this elevation map will not be updated as rover's explore the environment.

Scheduler: The scheduler serves as the swarm controller's decision making unit and is responsible for dispatching and commanding each EZ-RASSOR. This subsystem takes into account each rover's current task, battery level, and position, as well as the distance to each dig site and whether the site is occupied, in order to make intelligent decisions with regard to rover tasking and routing. A diagram depicting the scheduler's basic decision making flow is shown in figure 5.

Path Planner: This subsystem is used to generate energy-efficient paths for rovers when they must navigate to a dig site, charging station, dump location, etc. Implemented using the A* search algorithm, the paths returned from this path planner should also avoid any large craters and other treacherous terrain present in the lunar environment. As mentioned earlier, this subsystem relies on the elevation map described above to create accurate paths.

Waypoint Client-Server API: Serves as a communication layer between the central swarm controller and each individual rover. This sub-system is responsible for sending waypoints and actions to rovers while monitoring their feedback, recalling rovers for charging when low on battery, and re-planning rover paths when unexpected obstacles are encountered.

A diagram depicting the flow of information between each of the subsystems is provided in figure 3. At a high level, the central swarm controller is comprised of the scheduler and the path planner. Based on the current status of each rover and dig site, the scheduler will make decisions regarding which rover should do what. When it is time for a rover to move somewhere, the path planner is called to generate an efficient path from the rover's current position to a goal coordinate. Once the path is generated, it is sent to the rover's corresponding waypoint client, which will repeatedly send each waypoint in the path to the rover. During navigation, a rover will send feedback to its waypoint client, allowing the client to alter the rover's current task or path when necessary. This feedback is also used to update the swarm controller's knowledge of each rover, allowing the system to make decisions which account for a rover's current battery or position.

It should be noted that figure 3 visualizes only the swarm controller's decision making process and functionality. Each individual rover is responsible for its own autonomous navigation and localization. This functionality was developed by previous EZ-RASSOR developers and therefore will not be discussed in this paper. Detailed information regarding the implementation of each subsystem will be provided in the following sections.

A. MAPPING

Arguably the most important step in designing the swarm controller was to build a map of the lunar environment. To do so, NASA's Lunar Surface Model [2] was leveraged in order to instantiate an elevation map. This surface model is a high-resolution topographical map of the lunar surface, created by NASA using their Lunar Reconnaissance Orbiter. This surface model comes in the form of a black and white image, depicted in figure 4, where each pixel value

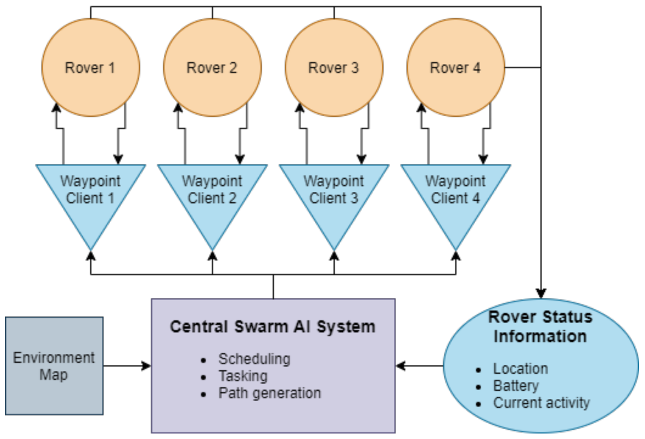


Fig. 3: High level diagram of the complete swarm control system and each of its interconnected subsystems.

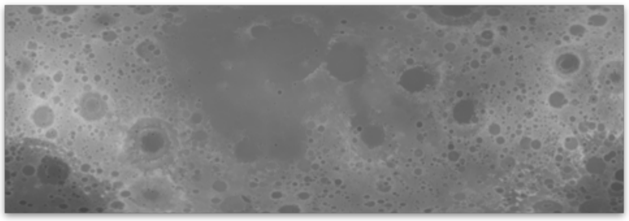


Fig. 4: Image of the lunar surface using NASA's elevation data [2]

represents the elevation at a coordinate with respect to the maximum lunar elevation. By leveraging the metrics provided alongside this surface model, namely the maximum and minimum elevation, we were able to transform these pixel values to true elevation values. This one-channel elevation map can now be utilized by the path planner described below, and was also used to create realistic Gazebo worlds.

The central goal of this project was to simply get the swarm management system up and running, complete with informed scheduling and accurate path planning. Therefore, this pre-built map served as a great way to create our environment map without requiring the use of SLAM algorithms. Allowing the system to update this map using SLAM as rovers explore the environment would be an excellent expansion of this project, and would enable the map to become increasingly accurate over time. However, for this project, we assume the lunar environment is static and we will not be updating this map. It should also be noted that even the highest resolution surface model has roughly 3 to 5 meters of physical distance between each pixel. Therefore, in practice, one must assume this map contains some degree of error and ensure that individual rovers are able to autonomously avoid any small obstacles not present in the map.

B. SCHEDULING

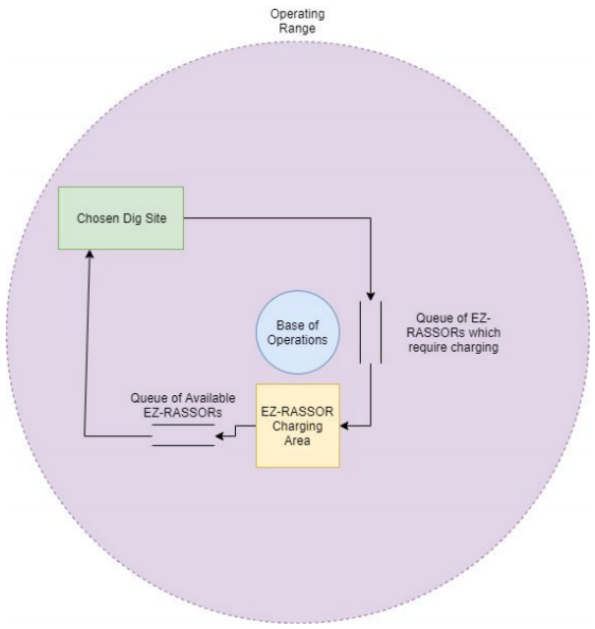
As described above, the scheduler is the central decision maker for our swarm control system. This subsystem will constantly loop through each rover, checking whether a given rover's current activity should be updated. The decision making algorithm implemented in this subsystem is described in algorithm 1. Note that this decision loop for each rover is always running, so long as the swarm controller is active. A visualization of this decision loop is provided in figure 5.

Algorithm 1 Rover Scheduling Algorithm

```
$S$: the set of all rovers in the swarm
$D$: the set of all dig sites
$C$: the set of all charging stations
procedure ScheduleRovers($S$, $D$)
    for each rover $r$ in $S$ do
        if $r$ is idle and $r.battery > 75$ then
            $D_n \leftarrow$ dig site with fewest rovers currently there
            send rover $r$ to dig site $D_n$
        else if $r$ is digging $r.battery < 35$ then
            $C_n \leftarrow$ charging station with shortest queue
            send rover $r$ to charging station $C_n$
        else if $r$ is driving to dig site then
            if $r.battery < 35$ then
                $C_n \leftarrow$ charging station with shortest queue
                send rover $r$ to charging station $C_n$
            else if $r$ is at designated dig site then
                command rover $r$ to dig for set period of time
            end if
        else if $r$ is driving to charging station then
            if $r$ is near designated charging station then
                if charging station is unoccupied then
                    command rover $r$ to charge
                else
                    set rover to idle nearby station
                end if
            end if
        else if $r$ is charging then
            if $r.battery > 95$ then
                $D_n \leftarrow$ dig site with fewest rovers currently there
                send rover $r$ to dig site $D_n$
            end if
        end if
    end for
end procedure
```

C. PATH PLANNING

The path planning subsystem is tasked with the creation of energy-efficient paths for rover navigation. Due to the computational restraints of the individual rovers, path planning is done on the more powerful swarm controller node and then paths are sent to individual rovers to follow. Our path planning subsystem leverages the Local Repair A* algorithm, which is a decentralized multi-agent path planning algorithm that is highly scalable to systems with large numbers of agents. This algorithm works by naively planning an agent's path using the A* search algorithm, while ignoring the other agents in the environment, and then solves conflicts or collisions at a local level.



Operating Range
Chosen Dig Site
Base of Operations
Queue of EZ-RASSORs which require charging
Queue of Available EZ-RASSORs
EZ-RASSOR Charging Area

Fig. 5: Diagram depicting the flow of rovers, commanded by the scheduler, between the important areas on the lunar surface.

The A* search algorithm itself is a guided best-first search algorithm that guarantees finding the shortest path between two nodes. The algorithm leverages a heuristic, usually the Euclidean or Manhattan distance, to guide its search and determine which nodes are likely to form the shortest path. We modify this algorithm to function on our lunar elevation map and provide the algorithm a few additional constraints to ensure generated paths are safe for rovers. The major constraint we introduce is to ensure that two connected nodes in a path do not form a slope greater than the maximum slope the EZ-RASSOR can climb. This ensures that each EZ-RASSOR can physically follow paths generated by A*. Psuedocode for this algorithm is provided in algorithm 2.

Local Repair A* runs online, meaning that once a path is generated using A*, it is immediately sent to a rover for navigation. Local Repair A* then monitors each rover's progress along their path, re-planning a rover's path when a collision is imminent. This approach does require accurate obstacle detection and avoidance, however, we know that individual EZ-RASSORs are capable of this. This flexible re-planning approach was optimal for a number of reasons. Firstly, when compared to other multi-agent path planning approaches, many of which plan in the time dimension to avoid rover collisions altogether, Local Repair A* has a significantly lower time complexity. This low time complexity was a must-have for this system, due to the large lunar environment that this path planner must function in. Additionally, we found that the treacherous lunar environment frequently forced rovers to autonomously avoid small obstacles. Once a rover begins to autonomously veer from its pre-planned path, the swarm control system can no longer guarantee the rover is following a safe or efficient path. Due to this, the path must be re-planned from the rover's current location after avoiding an obstacle. Local Repair A* is able to handle this situation seamlessly, and due to the speed of this algorithm, without a significant time overhead. In order to understand how the swarm controller monitors rover progress and re-plans when necessary, see the following section regarding the Waypoint Client-Server API.

D. Waypoint Client-Server API

Implemented as a ROS Action Client-Server using the actionlib library, the swarm controller's Waypoint API handles the sending of individual waypoints or actions to rovers. This subsystem also monitors rover progress along a path and re-plans when necessary, as described in the above path planning section. Therefore, the swarm controller maintains a waypoint client and server for each rover in the swarm.

The flow of information from the central swarm controller, to a rover's client, and ultimately to the rover itself is depicted in figure 3. Essentially, once a path is generated, it is sent to a rover's corresponding waypoint client. The rover's waypoint client will then repeatedly send the next waypoint in the path to the rover, wait until the rover reaches it current waypoint, and then send the following waypoint. During navigation, the rover's waypoint server will constantly send the rover's current battery level and position back to its client. The waypoint client is then able to leverage this feedback in order to implement the re-planning aspect of Local Repair A*. In practice, the client monitors a rover's distance from its current waypoint. If the distance is found to be greater than some threshold, the client will simply preempt the current path, plan a new path using A*, and send the rover along this new path.

It should also be noted that each waypoint client is an individual ROS node, the central swarm controller is also a single ROS node, and finally each rover is comprised of a family of nodes which handle autonomous navigation. By making each waypoint client a separate ROS node, this enables re-planning to occur on threads outside of the swarm controller's main thread, allowing the overall system to run smoothly without being frequently interrupted for re-planning.

III. DISCUSSION

A demonstration of our swarm management and control system can be found at https://www.youtube.com/watch?v=vFvvzjbMjeQ. This video will demonstrate a swarm of 3 EZ-RASSOR rovers mining regolith from 3 unique dig sites. It should be noted that our swarm simulation can be ran using a single ROS launch file, and parameters such as swarm size, rover spawn locations, dig site locations, rover battery life, and more, can all be easily passed to the launch file in order to create varying simulations. As the video begins, the swarm controller will task each rover to a certain dig site, plan paths for each rover, and finally send each rover along their path. One should notice how the rovers are capable of autonomously navigating along these paths. When the rovers reach their given dig sites, each rover's bucket drums will lower and the rover will begin digging in a back and forth fashion. Unfortunately, we were not able to simulate the rover's bucket drums becoming full with regolith, which is one shortcoming of this project. Therefore, the rovers will dig until their battery levels deplete below a certain threshold. At this point, the swarm controller plans a path to the nearest unoccupied charging station and signals the rover to attempt to charge. In our demonstration video, there is a single charging station located at the center of the map. Notice that when rovers reach the charging station, if the station is occupied, rovers will wait nearby until it is their turn to charge. Another shortcoming of this project is that the system does not account for a rover's battery level depleting to zero while waiting in line. While this is unlikely to occur, it is certainly a situation which should be accounted for.

The following segments of our demonstration video display the benefits of our path planner. The segment that initially follows the overall swarm control segment will demonstrate a rover attempting to cross treacherous terrain without a pre-planned path. The following two segments will then demonstrate the swarm controller generating paths around the same treacherous terrain in an attempt to visualize the benefits of A* path planning. As shown in the video, our path planner is able to generate safe paths around terrain which the rover certainly can not cross. However, a clear disadvantage of A* path planning is that while paths generated with A* are efficient, they are not necessarily natural. For example, paths will avoid steep terrain by as little room as possible, rather than giving the rover some comfortable margin for error. This is portrayed in the second path planning example, which shows a rover navigating precariously around a deep crater.

The final two segments portray our Local Repair A* algorithm aiding a rover which has encountered an unexpected obstacle. In the first of these segments, re-planning has been disabled and a wall is spawned between the rover and its goal. With re-planning disabled, the rover will continuously attempt to reach a waypoint that is blocked by the wall. This demonstrates the need to preempt a rover's path and plan a new path after an obstacle has been avoided. In the final clip, Local Repair A* re-planning has now been enabled, demonstrating the full capability of our system. Once the waypoint client receives feedback that our rover has autonomously drifted off course, a new path will be planned and sent to the rover. This will occur once the rover has gotten around the artificial wall. The viewer should note that once this new path is planned, the rover is able to reorient itself and leverage the new path to successfully reach its goal. There is, however, a small lag period during the time the waypoint client is re-planning the rover's path. This causes the rover to repeat the last ROS movement command it received until it receives a new path. This lag period can be seen in the video right after the rover gets around the wall. Rather than stopping rapidly, the rover slowly drifts forward executing its last received ROS command until the new path is received. While this doesn't affect the rover's ability to reach the goal in this scenario, it's certainly possible that this could cause rovers to drift even further into treacherous terrain. Given more time, we would have liked to have addressed this issue, as well as the issues mentioned above with regard to A* path finding, bucket drum capacity, and rovers queuing for charging.

IV. CONCLUSION

There remain several interesting directions for future expansion and improvement upon this project. Firstly, small fixes to the problems described in the above section would be a great addition. These include adjusting the A* path planner to create safer paths around terrain, simulating rover bucket drum capacity, and improving the behavior of rovers waiting for charging stations. Additionally, this project makes two strong assumptions; that NASA's Lunar Surface Model is accurate and the lunar environment is static. However, if this system were to truly be deployed on the moon, we could certainly not rely on both of these holding true. Therefore, leveraging multi-agent SLAM approaches to update the system's environment map as rovers explore the lunar surface is an important expansion of this project. I believe this is the most interesting, difficult, and useful problem a swarm controller would need to solve in order to be truly intelligent. Another interesting improvement would be to enable the system to cache previously computed paths and leverage these when appropriate. This could save large amounts of time that would otherwise be spent computing paths which have previously been computed. This approach also makes sense in practice, being that rovers would likely be utilizing the same dig sites for extremely long periods of time. Intuitively, this would allow the swarm of rovers to create their own highway network connecting dig sites, dumping sites, and charging stations on the moon. Lastly, this work focused solely on the high-level control and management of a swarm of EZ-RASSORs. However, much work could still be done in order to improve the autonomous functionality of each individual rover. Examples include improving the low-level motion control, obstacle detection and avoidance, and localization capabilities of the EZ-RASSOR.

REFERENCES

[1] NASA, "Regolith advanced surface systems operations robot." https://technology.nasa.gov/patent/KSC-TOPS-7.

[2] NASA, "High resolution topographic map of the moon." https://moon.nasa.gov/resources/87/high-resolution-topographic-map-of-the-moon/.

Algorithm 2 A* Path Finding Algorithm

```
$M$: elevation map of the environment
$S$: start coordinate
$G$: goal coordinate
procedure FindPath($M$, $S$, $G$)
    # initialize the open set
    $open \leftarrow$ list containing solely the start coordinate
    # For node n, previous[n] is the node immediately preceding it on the generated path
    $previous \leftarrow$ empty map or dictionary
    # For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    $gScore \leftarrow$ empty map or dictionary
    # g score of the start node is always 0
    $gScore[S] \leftarrow 0$
    # For node n, fScore[n] := gScore[n] + h(n, G) where h() is a chosen heuristic function. fScore[n] represents our current best guess as to how short a path from start to finish can be if it goes through n.
    $fScore \leftarrow$ empty map or dictionary
    $fScore[S] \leftarrow h(S, G)$
    while $open$ is not empty do
        $current \leftarrow$ the node in $open$ having the lowest fScore[] value
        if $current$ is $G$ then
            return backtrack($previous$, $current$)
        end if
        $open \leftarrow$ open.remove(cur)
        for each $neighbor$ of $current$ do
            $tentative_gScore \leftarrow gScore[current] + h(current, neighbor)$
            if $tentative_gScore < gScore[neighbor]$ then
                # Found new, more efficient path to neighbor so record it.
                $cameFrom[neighbor] \leftarrow current$
                $gScore[neighbor] \leftarrow tentative_gScore$
                $fScore[neighbor] \leftarrow gScore[neighbor] + h(neighbor)$
                if $neighbor$ not in $open$ then
                    $open \leftarrow open.add(neighbor)$
                end if
            end if
        end for
    end while
    # No path was found between S and G
    return None
end procedure
```