NASA EZ-RASSOR 2.0

UCF Fall 2019 Black Team

Team 12

Team Members:

John Albury

Shelby Basco

John Hacker

Michael Jimenez

Scott Scalera

Project Sponsor:

Michael Conroy

Table of Contents

Executive Summary

Description

This project aims to build on the foundation provided by the previous iteration of the EZ-RASSOR project. EZ-RASSOR is a software suite designed for use on NASA's Mini-RASSOR rover, a scaled-down version of the RASSOR rover. The first iteration of the project, completed by a Computer Science Senior Design team in Spring 2019, involved creating computer models of the Mini-RASSOR and a moon-like environment, building out the controls and communications functionality of the rover using ROS, and implementing basic autonomy. We aim to further develop the project by adding GPS-denied navigation and improving the autonomy of the rover. Essentially, the goal of our work is for the rover to be able to safely navigate from a starting location to a given target location without the aid of GPS in a simulated moon-like environment.

Objectives

The main objectives of our project are as follows. First, the rover should be able to determine its current location. We can assume the rover knows its initial location, since the rover will typically be deployed from a base that is at a known location. Second, the rover should be able to detect obstacles in front of it that are blocking its current trajectory. Third, the rover should be able to navigate to a given target location using a path that is both efficient and avoids colliding with any obstacles. All of the listed functionality should be tested on a simulated rover in a virtual, moon-like environment.

Approach

While we will discuss our technical approach more in-depth later, here we will give a high-level overview of our technical approach to give context for future sections.

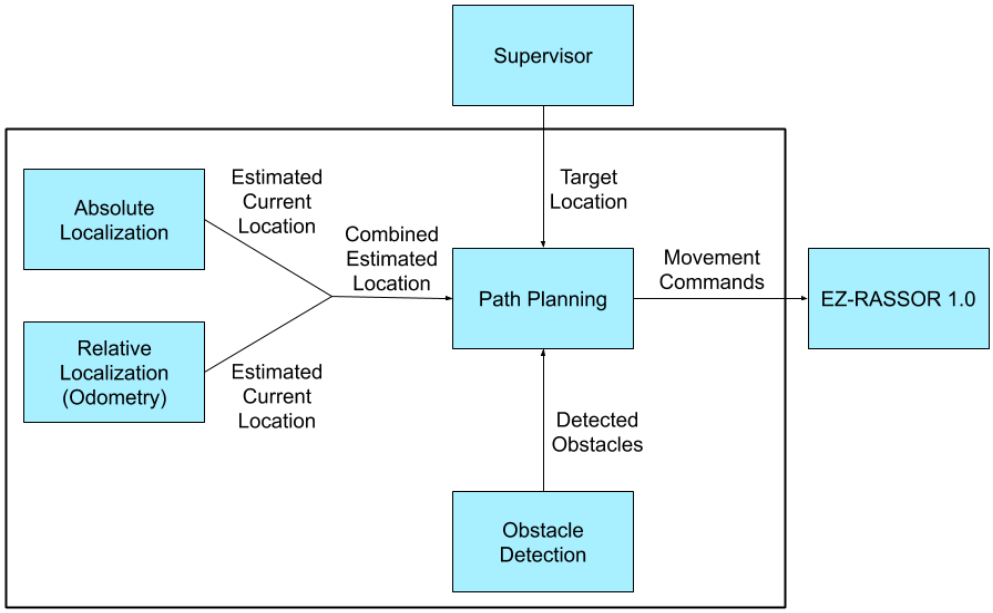


Figure 1: A high-level overview of our technical approach to the project. The part of the diagram inside the box is what we will be implementing.

In order to navigate to a desired location, the rover obviously needs to know where it is. Determining location is also known as localization. Without the aid of GPS, there is not a simple solution to this problem. Thus, we have divided the localization aspect into two parts: absolute localization and relative localization. Absolute localization is determining the location of the rover without using a starting location. With this type of localization, the rover should have the ability to be placed at any location with no knowledge of where it is and be able to determine its location. GPS is a common method for absolute localization on Earth. Relative localization is determining the location of the rover using a starting location and involves measuring how far the rover has moved from the starting location. An example of relative localization would be using the motion sensor data from your phone to determine how far it's moved from some initial location. The location estimates from absolute localization and relative localization will be combined to create one estimation for the current location of the rover.

While the rover is navigating, it will need to safely avoid obstacles along the way. In order to do that, there must be some method of detecting obstacles in the rover's path. The obstacle detection component will be responsible for detecting obstacles in front of the rover based on the sensor data available, then determining the locations of these obstacles relative to the rover.

The end goal of the RASSOR is for groups of RASSORs to autonomously complete objectives in a coordinated way. Thus, the logic of determining where a rover should go next requires swarm logic and knowledge of the current objective of the group. Our project focuses on navigation logic at the level of individual rovers, so determining the target location of the rover is out of the scope of our project. We will assume that the target location is provided by some "supervisor." The logic for this supervisor is actually the task of the other Senior Design group working on EZ-RASSOR, but for the sake of keeping the projects separate since we are in separate teams, we will be entering target locations manually.

Once the rover knows approximately where it is, where it should go, and what obstacles are ahead, this information can be combined to determine the direction and speed the rover should move. The path planning aspect of the project will aim to minimize the distance traveled to get to a target location while safely avoiding obstacles along the way. The output will be in the form of movement commands compatible with the code that was written in the previous iteration of the EZ-RASSOR project: "EZ-RASSOR 1.0."

Project Narrative

In the previous iteration of this project (EZ-RASSOR 1.0), two groups of five-person teams banded together to tackle a project with requirements that would've been impossible for a single team. Their task: create a software architecture for the scaled-down version of NASA's RASSOR, whose hardware counterpart was being built by NASA's Swamp Works at Kennedy Space Center in tandem. Due to the nature of this timeline, there was a need for a way to test the software without the final hardware. The team created a 3D computer model of the Mini-RASSOR and used a simulation environment to test the software. Even though they could develop the software for the robot using only the simulation, they also built a physical model they could test with, since what

happens in a simulation is not always what happens when the software is tested in the real world.

The features the team developed can be grouped by the packages they developed and are fully independent, making it relatively easy for the developer to upgrade or tailor the project to their needs. The components can be grouped as follows: hardware (for controlling the physical hardware of the robot), communication (for routing messages from controllers to the hardware), simulation (for running the robot in a virtual environment instead of on physical hardware), controllers (the mobile app that controls the robot), and autonomy (for controlling the robot without the need for human interaction). For our project, the main relevant component is the autonomy component. We are not the only team working on EZ-RASSOR 2.0, and we may need to communicate closely with the other EZ-RASSOR team because they will also be working on functionality related to autonomy.

Our main task is as follows: the EZ-RASSOR should be able to pinpoint where it is (in terms of latitude and longitude) without the aid of GPS and navigate to a desired location. Setting up and maintaining satellites around a non-earth planet or moon would be an unnecessary expense at this point in time, so the EZ-RASSOR should test the software capabilities of what a fully autonomous robot can do without humans around. In addition, we will also be working on path planning: once we have the coordinates of the EZ-RASSOR and the coordinates of a location we want to navigate to, the robot should be able to navigate itself to that point safely. Along the way, it should detect and avoid obstacles. Finally, we will be re-implementing the robot's navigation stack to use twist messages instead of bit string messages. Although this is not necessary for GPS-denied navigation, this is a change that the previous team recommended in order to comply with ROS standards and to make communicating with the Mini-RASSOR hardware simpler (currently, the bit string messages are translated to twist messages to communicate with the hardware; generating twist messages directly would eliminate this overhead).

Motivations

John Albury

The EZ-RASSOR project appealed to me for several reasons. First, because of its relation to space exploration. It's exciting to be part of a project that can potentially contribute to the future of space exploration, and working with NASA engineers it will be interesting to learn how the development process is for software that could potentially go to space, where errors and inefficiencies have much larger consequences than in typical software applications. Second, I'm interested in the robotics nature of the project. Having taken related courses such as Robot Vision at UCF, I'm excited to apply the theory I've learned in my computer science courses to add functionality to the rover. Third, the open-source property of the project appealed to me. I believe open-source is important to the advancement of technology, so working on an open-source project for Senior Design seemed ideal to me.

Shelby Basco

When someone asks me what companies I'd like to work for, the first one that pops into my head is NASA. They have captivated the imaginations of children across the nation for generations, and I was no different. When the project was pitched to the class, the first thing that caught my eye, besides it being supported by NASA, was the Denied GPS feature. As the sponsor, Micheal Conroy further explained, it seemed most of the computational work was relying heavily on vision. Currently, when I go into the workforce, I want to work more closely with either computer vision, artificial intelligence, machine learning, or a combination of the three. With all those factors taken into consideration, it seemed like the most logical decision was to place it as my number one choice for a senior design project.

John Hacker

To enhance my education at the University of Central Florida, I have decided to pursue a minor in Intelligent Robotic Systems. This has taught me the concepts and skills involved in artificial intelligence, computer vision, and robotics. I see this project as an opportunity to implement and develop my skills using ROS and Gazebo to solve complicated problems. This is also a very exciting opportunity to

work with a world-renowned organization such as NASA. The work at the National Aeronautics and Space Administration inspires people to reach for the stars and provides thousands of employees the opportunity to fulfill their childhood dreams. I may never get to go to space; but, developing software for a robot that may go to space is almost equally as rewarding. Even though the Mini-RASSOR is not designed to go to space, it is still neat that the EZ-RASSOR software may be scaled for a larger RASSOR to one day visit the moon and beyond.

Michael Jimenez

My personal motivation for this project stems from a desire to learn more about NASA along with the technologies and algorithms that go into EZ-RASSOR. My main goal for the project is to have fun, learn how GPS navigation works and successfully advance the EZ-RASSOR project into a better and more functional state. Despite having not taken robot/computer vision classes or working with ROS before, I am more than willing and motivated to learn all the required concepts to be able to make a positive impact. Learning and working with all of these new concepts and tools is partially why this project was so appealing to me. I hope that through the obstacles that this project presents, I will not only be able to grow as an engineer but as a person.

Scott Scalera

My personal motivation for this project is in getting the opportunity to work on a challenging project sponsored by NASA. Although the lunar rover software we are developing will most likely never be used in its intended environment, it is well worth it because our rover may inspire those who see it in demonstrations or be of use to others' research. Finally, there is no better motivation than being able to work on a difficult problem like building a completely autonomous planetary rover, a goal that has yet to be achieved in practice.

Backgrounds

John Albury

I started programming in college as a way to supplement my studies and research in Mechanical Engineering. However, once I started programming I

quickly realized that my true passion was in software development and I changed my major to Computer Science. Since then, I've gained experience in scripting, application development, and software design through internships, coursework, and personal projects.

Having previously worked on analyzing sensor data in a mechanical engineering lab, I'm excited for my first opportunity to work with data from sensors since changing my major to Computer Science. Also, I am interested in the computer vision components of the project. I have completed the Robot Vision course at UCF, but I want to learn more about applying computer vision techniques in real-world scenarios. Thus, I'll be assisting with the obstacle detection component of the project, where we'll likely use images from an onboard camera to detect obstacles.

Although I do not have much prior experience in robotics, I do have prior experience in software engineering and plan to use the knowledge I've gained to make the development process of this project as smooth as possible. For example, by using a modular design and creating clearly-defined interfaces for each of the components of the project, we can allow different team members to work on different components of the project concurrently without having to stress about how those components will be integrated later.

Shelby Basco

If you told younger me that I'd ultimately be looking for a cubicle job, my whole career path would've changed. This is because, for the longest time, I wanted to be some sort of artist and specifically, being a video game artist seemed to call me. I realized that even though I liked to draw, I didn't like the idea of my income solely relying on someone's opinion. Life will always be inherently biased but I wanted something a little bit more objective yet could be applied to many fields. So then I found programming, I could still be involved with video games but I'd be making the nuts and bolts. The only reason I decided to major in Computer Science was that having "Game Development" on my degree seemed to narrow my opportunities in the job market. Prior to UCF, I had tried to learn web development at the end of my senior year in high school. I did it to sort of get a head start on coding but it didn't seem like something that would be taught in class. I end up not quite retaining what I had learned since I simply followed exercises and never built anything with it.

It’s now my senior year and my most recent group project was BLE-Attendance, which consisted of web and mobile app development. I end up working on the database, so it was a part of the web development online course that I never got to in high school. In terms of knowledge related to the project, I have no experience with robotics whatsoever. I do however have some understanding of basic computer vision concepts because I am currently in the Robot Vision class. I don’t have any work experience or projects outside of class at the moment so I’m absolutely in shock and thankful to be working on this project. I’m also thankful for my team making me project manager and for being so involved without having to say anything.

John Hacker

My first experience with programming came during my senior year of high school when I switched out of a graphic arts class into AP Computer Science, on the second day of the year. It was a class I always was interested in taking but seemed to never make the cut of fitting into my schedule. I sure am happy I landed in this course because it was the first time in my school career that I felt like I did not want to leave the classroom most days when the bell rang. Solving the logical puzzles and learning a new language (Java) was fun and exciting. I originally applied to UCF to study emerging media in order to become an animator but changed my major to computer science during orientation. I decided my logical skills were far stronger than my drawing.

Fast forward to my first semester of Senior Design, I am taking my final course to finish a minor in Intelligent Robotic Systems. Classes I have taken to complete the minor include Robot Vision, Artificial Intelligence, and Introduction to Robotics. Each of these courses has taught me skills that are applicable to this project. In Robot Vision, I learned about many techniques to manipulate and analyze images, such as edge detection, convolution, segmentation, and machine-learning classification. Artificial Intelligence taught me several path planning algorithms and what makes them optimal. Finally, Introduction to Robotics taught localization techniques and gave me lab experience using ROS and simulation in Gazebo through a cascade of turtle-based tutorials. These classes have well equipped me to be familiar with the techniques to be successful in this project. Conversely, the EZ-RASSOR project is giving me the opportunity to practice algorithms I only learned about before.

Michael Jimenez

Computer science has always been something that has interested me. As a kid, I used to play a lot of video games and always wondered how everything worked. This desire and curiosity for knowledge has led me to pursue a degree in computer science and even a minor in mathematics. My experience with computer programming started in college where I learned: how to program in C and Java, how to construct and use data structures, how compilers and memory work, basic computer security and discrete mathematics. After learning these foundational components I began to take an interest in computer security/cryptography and have even worked two separate security related internships. During my time at Citrix as a security engineering intern, I worked with SSL/TLS and certificates (PEM and DER encoded), CSR's and x509 CRL's, RSA, EC, ECDSA, DH public and private key pairs and wrote multiple scripts to standardize, visualize and renew the certificates found in a repository.

Despite not having much experience with path planning, robotics and computer vision, I am more than willing and excited to learn about all of these fields of computer science. I am very excited to start working on this project and honored to have been chosen to work with NASA.

Scott Scalera

My journey formally into computer science, unfortunately, started only last fall when I decided I wanted a career that is in demand and requires problem-solving and critical thinking. Up to this point, I have gained a strong appreciation for algorithmic thinking (from classes such as COP3502, COP3503, COT 4200) as well as system design thinking (COP3225, COP4331, and EEL4710). And, I hope I can be of use in these areas as well as others, to the team.

Division of Labor

Going forward, we have divided the tasks necessary to meet our requirements into four distinct categories: Absolute Localization, Path Planning, Obstacle Detection, and Odometry & Twist Message Support. Each member of the team will work on tasks related to two to three of these categories. Each category will

have two to four members working on it, depending on the estimated workload of the tasks related to the category. Additionally, each category has been assigned a lead. The lead for a category will be responsible for ensuring consistent progress for tasks related to that category and will be the main point of contact for that aspect of the project.

	John A.	Shelby	John H.	Michael	Scott
Absolute Localization					*
Path Planning				*	
Obstacle Detection			*		
Odometry & Twist Message Support	*				

Figure 2: The formal assignments for the project team. An asterisk indicates that the team member is the lead for that component of the project.

Absolute Localization

The Absolute Localization component of the project (determining the location of the rover without the use of a known initial location) is a complex and important part of the project. Thus, it has been allocated the most team members of any of the categories. The members of this subteam will be responsible for designing a method for determining the current location of the rover and will determine the hardware necessary to fulfill their design. This component will output an estimation for the current location of the rover (in the form of lunar coordinates) so that this information can be utilized in other components of the project.

Path Planning

The Path Planning subteam will be responsible for planning the route to the desired location. Because there will potentially be many other rovers in an area, swarm logic must be utilized to ensure that rovers are not taking paths that will lead to collisions between rovers. The other EZ-RASSOR 2.0 team will be implementing swarm logic as part of their requirements, while we are responsible for navigation at the level of an individual rover. Thus, the Path Planning

component of our project will assume that it is given a coarse path to a destination (in the form of waypoints) that avoids predictable collisions with other rovers. Our Path Planning subteam will be responsible for designing a method to safely and efficiently route a rover to its next waypoint. This component will output movement commands for the rover.

Obstacle Detection

In order to safely navigate to the desired location, the rover will need to have the ability to detect and avoid unforeseen obstacles along the way. The members of this subteam will be responsible for all tasks related to detecting the obstacles being faced by the robot's camera. This subteam will design a method for detecting obstacles and determine the hardware necessary to fulfill their design. The Obstacle Detection component will output an object representing a detected obstacle, which can include information such as the estimated distance between the obstacle and the rover, the estimated size of the obstacle, and possibly a classification of the obstacle.

Odometry & Twist Message Support

The Odometry & Twist Message Support subteam of the project will be responsible for implementing the Odometry (relative localization) component of the project. Using knowledge of the rover's starting location, the Odometry component will aim to determine how far the rover has moved so that its current location can be estimated. This component will output an estimation for the current location of the rover (in the form of lunar coordinates) so that this information can be utilized in other components of the project. This subteam is also responsible for transitioning the current EZ-RASSOR codebase from using bitstring messages to using twist messages.

Objectives

The overarching goal of the EZ-RASSOR project is to have a robot to: present to the public at the Kennedy Space Center, teach students about how some of the technology that NASA uses works, and provide a foundation for university groups to research problems that the RASSOR faces. The EZ-RASSOR software controls a robot, such as the Mini-RASSOR or a simulated version of it, to

demonstrate what is possible for surface regolith mining. Below are the current objectives of our contributions to the multi-year EZ-RASSOR project:

- Absolute localization of the robot on a known coordinate system (such as the lunar coordinate system)
- Relative localization of the robot using a known starting location
- Obstacle detection through cost-effective hardware and software techniques
- Efficient path planning for traversing to a desired location
- Preservation of the safety of the robot by safely avoiding obstacles along paths
- Twist message support to match desired open-source standards

Goals

The goal of this project is to provide autonomous navigation capabilities to the EZ-RASSOR. For our purpose, autonomous navigation is defined as the rover being able to move from its current position to some arbitrary position given to the rover without the need for human intervention. The processes needed for this are a method of localization, obstacle detection and avoidance, and path-finding.

One of the major aspects of this project is determining the rover's current coordinates. Many of the other aspects of the project, such as path planning, will rely on the accuracy of the estimation location. Accordingly, one of the main goals we will focus on is making the location estimation as accurate as possible.

Since this software is meant to be used on a demonstration/research robot, the rover platform we build for testing should be relatively low-cost. Thus, another goal will be determining the necessary hardware (cutting out any sensors or other hardware that are not absolutely necessary) for the rover and finding cost-effective, high-quality parts.

Since the rover is being designed for use in a moon-like environment, we will also test the robot in a simulated lunar environment. The previous EZ-RASSOR team developed a 3D computer model of the Mini-RASSOR and developed simulated Mars and moon environments to test it in, but we will need to add to

these environments in order to fully test the navigation capabilities we are developing.

Function

This section will provide a high-level overview of the design criteria and foreseeable constraints. The design criteria were derived from the goals and specifications & requirements sections.

To start off, EZ-RASSOR should be able to use GPS-denied navigation. GPS-denied environments, such as the moon or other foreign celestial bodies, make the use of GPS not currently possible due to the lack of orbiting GPS satellites. We will develop an autonomous vision-based navigation algorithm that can be used to accurately specify EZ-RASSOR's position within 25 meters. At the moment, 25 meters is our goal; this distance was given to us by our sponsors. Due to the uncertainty in the feasibility of this goal, our sponsor has informed us this number may change throughout the course of the project as we progress through different obstacles and do more research. We will be able to test this functionality through our virtualized environment.

Another important function of EZ-RASSOR is obstacle avoidance and detection. As our robot traverses the unknown, it should be able to utilize optics/sensors to prevent collisions with other objects.

Since we are the navigation team for the EZ-RASSOR project, we will also deal with pathfinding for simple navigation. This will essentially come down to how we are able to determine and use the data the robot has about the surrounding environment. We will be able to control our robot through the use of ROS, an open-source middleware software for robot applications, to accomplish the goals of this project. Our robot should also be able to measure how far it has traveled. This will give us an additional estimate of the robot's current location based on an initial known location and how far the robot has traveled from that location.

Finally, the main foreseeable constraints are those affiliated with the cost of the hardware necessary to implement our solutions. Our goal is to minimize cost at every step but to also consider performance and stability. Cheaper hardware

may come at the cost of the quality of the end project. Knowing this information, it will be a challenge to recommend hardware to the Swamp Works team that is both high-quality and cost-effective.

Broader Impacts

The impacts of this project could potentially be global (and beyond) in the future. The excavator robot the EZ-RASSOR is based on, the RASSOR may be vital to future efforts to build bases on and colonize the moon and Mars. Regolith, the material that the RASSOR is designed to efficiently excavate, has many purposes on the moon and Mars. First, water can be mined from regolith. For humans to survive long-term on another planet, they need a local source of water, and mining water from regolith is a promising option. Second, regolith can be mixed with plastic to be used in 3D printing. Due to the high costs of transportation to other planets, 3D printing will likely play a vital role in future colonization missions; instead of shipping some structure produced on Earth to the moon or Mars, the structure can be produced relatively quickly and cheaply on the planet itself through 3D printing. Finally, regolith can be used as an efficient radiation shield for bases. Radiation protection is another obvious necessity for human survival on other planets, and covering human bases in the regolith material provides a relatively low-cost solution. In all of these cases, the RASSOR could play a crucial role by collecting and delivering the regolith to be processed.

Although the software we are writing is not meant for the RASSOR itself, the EZ-RASSOR could play an important role in the development of the RASSOR. While the RASSOR is currently being developed by the NASA SwampWorks team at the Kennedy Space Center, breakthrough innovations in the project can be accelerated by crowd-sourcing the development to universities. As an open-source, smaller, and lower-cost version of the RASSOR, EZ-RASSOR can easily be distributed to universities to allow experts in their respective fields to tackle specific problems. The EZ-RASSOR can also serve as a safe testing platform for prototyping new ideas due to its low cost relative to the RASSOR.

Specifically, our work in GPS-denied Navigation will provide an important component for future teams working on the autonomy of the robot. Location

information can be used in areas such as swarm control (knowing the current locations of the robots is an important aspect of efficiently orchestrating the robots to perform a task) and autonomous navigation (in order to navigate to a specific location, the robot needs to know its current location relative to its target location). Additionally, we will explore many different approaches while developing this project. The research and experimentation we perform can provide a starting point for other projects that are looking to perform navigation in the absence of satellite GPS (i.e., other robots that are meant to go to the moon or Mars).

Legal, Ethical, and Privacy Issues

Since the EZ-RASSOR software may influence the research and development of the software that will eventually be deployed on RASSORs on the moon, we must ensure that the GPS-denied navigation we are adding is of high quality and has been tested thoroughly. Faulty autonomous navigation software on the RASSOR could endanger human lives and mission-critical infrastructure on the moon.

In particular, obstacle detection and avoidance must be reliable, since faulty obstacle detection and avoidance could cause the rover to crash into and damage astronauts, mission-critical infrastructure, and other rovers. Even if the rover crashes into a rock and only damages itself, this could endanger the overall mission since the RASSORs are expected to be important for in-situ resource utilization on the moon. Absolute localization and odometry must also be reliable and reasonably accurate, since these components are how the rover determines its current location. If these components malfunction or are highly inaccurate, the rover could get lost. Again, the RASSORs are expected to play an important role in resource utilization, so losing one or multiple RASSORs could lead to resource depletion, which could then risk the lives of the astronauts. Thus, the ethical duty of our team is to prioritize accuracy and reliability when designing and implementing GPS-denied navigation, then we will need to test and evaluate our solutions thoroughly when we reach the testing phase.

Specifications & Requirements

The following list of requirements was created to define the goals our team. Our Florida Space Institute sponsor, Mike Conroy, agreed for us to accomplish these tasks through research and development.

- The EZ-RASSOR software shall be delivered at the end of April 2020.
- The EZ-RASSOR software shall follow standards of open source code.
- The EZ-RASSOR software shall have well-documented research and development to be used in conjunction with past and future documentation to create a thorough understanding of what is a part of the software and why it is so.
- The EZ-RASSOR software shall provide software that continues to be presentable for demonstrations via NASA employees.
- The EZ-RASSOR software shall be developed in ROS.
- The EZ-RASSOR software shall be developed in Gazebo.
- The EZ-RASSOR software shall have a defined set of newly required hardware to fulfill its requirements.
- The EZ-RASSOR software shall use localization to locate the lunar coordinates of the robot within 25 meters of its true position (stretch goals are for better precision).
- The EZ-RASSOR software shall have a pathfinding ability for simple navigation.
- The EZ-RASSOR software shall not traverse the same path repeatedly to avoid digging itself into the soft surfaces on the moon.
- The EZ-RASSOR software shall be able to measure how far it has traveled.
- The EZ-RASSOR software shall use twist commands to control the robot’s linear and angular velocity vectors.
- The EZ-RASSOR software shall perform obstacle detection.
- The EZ-RASSOR software shall perform obstacle avoidance.
- The EZ-RASSOR software shall be tested in a simulated moon-like environment.

Additional Ideas from the Team

John Albury

- Utilize the A* algorithm in path planning.
- Utilize YOLO for object detection.
- Integrate the location of the rover (latitude and longitude) in the mobile app and allow users to select a latitude and longitude for the rover to autonomously navigate to.
- Add object classification functionality to obstacle detection (i.e., is obstacle another rover or just a rock?).

Shelby Basco

- If the rover is rebooted, check if there was previously saved data from the last iteration.
- Add Bluetooth in-app for rovers without wifi.

John Hacker

- One-direction LiDAR to detect objects close and in front of the robot.
- Use the camera on top of a base station and each robot to triangulate the relative position of each robot to the base station with consideration to stars, planets, height, level, and curvature of the surface.
- Use accelerometer and gyroscope/level to calculate the distance traveled and assist in refining position precision.
- Use an optical flow sensor and light source to track the movement of the surface from the point of view of the robot to estimate the distance traveled by the robot.

Michael Jimenez

- Use an optical odometer to determine velocity and displacement.
- Use 1-dimensional LiDAR to cut costs and to work with obstacle avoidance.
- Use a path planning algorithm that works well in unknown environments and doesn't waste time on excessive computation.

Scott Scalera

- Use a dome camera, inclinometer, and clock to both determine the rover's position (longitude/latitude) and orientation using celestial bodies, and to classify and range objects distant from the rover.
- Use the forward-looking vision sensor, lights, and orientation sensors in the vision odometer for dead reckoning.
- Absolute localization can use pattern matching to find the important stars in the stitched image of the sky.
- Absolute localization can be tested by using it (a version that works on the Earth instead of the moon) to determine our position on Earth. This would help us reason about the kind of capabilities it could achieve on the moon.

Research

Lunar Environment

In this section, we describe the lunar environment with an emphasis on factors significant to the operation of the rover.

The moon's tidally locked orbit around the earth has had an immense impact on the moon's development over the last 4 billion years. With the near side of the moon permanently fixed towards earth, it has largely been spared by large meteor impacts in comparison to the far side. This has shaped the terrain of both sides with the nearside highlands (regions of ancient crust) being void of many large impact craters and therefore smoother. While, on the far side relentless impacts over the last billions of years have shaped a jagged and rocky terrain. Further, the terrain has been affected by space weathering and micrometeor impacts. Due to the moon's lack of atmosphere, highly energetic particles have been able to bombard the surface, breaking rocks down into fine particles. While micro meteors create fine particles during the impact process. Over billions of years the continuous formation of these fine particles have blanketed the surface forming the thin regolith layer. This coating, with thicknesses greater in the highlands and thinner in young impact basins, has properties of fine sticky powder, creating the moon's smooth appearance.

The moon’s orbit also impacts the visibility on the surface. With earth always in view on the near side of the moon, earth's irradiance brightens the landscape at all times, a likely help to the rover’s visuals sensors. Though, with a downside of drowning out the light of stars. On the far side of the moon, apart from the edges, the earth is never visible with the only irradiance from starlight during night. This provides advantages in being better able to view a great number of stars for localization but requires the rover to operate in low light conditions. During the day in the equatorial regions, the sun's irradiance is greater than on earth, in comparison to the polar regions where the sun is hardly ever visible.

Figure 3: An image taken during the 11th Apollo mission in the Sea of Tranquility. [1]

The tidally locked orbit of the moon around the earth has also had an impact on the pattern of solar irradiation the moon experiences. With a slow orbit around the earth, the moon has a synodic period of almost 30 days. This means a lunar day lasts for 30 days with full irradiation during the middle of this period. During these long periods of irradiation, temperatures can reach several hundred degrees kelvin which pose a danger to the rover. Other hazards posed to the rover are micrometeor strikes and highly energetic particles that overtime degrade the rover’s physical health and electronics respectively.

Although our rover could potentially be used on any part of the moon, it has been indicated to us that certain parts of the moon are more significant than others. In

particular the southern polar regions. Over the last few decades numerous space agencies have probed the moon, either using remote sensors or directly by smashing probes to detect the presence of ice water. These experiments have indicated the presence of ice water likely does exist, to some capacity, in the permanently shadowed craters of the south pole. This is significant, as water can be split to form fuel precursors in the form of O_2 and H_2, which would decrease the cost of space missions to the rest of our solar system.

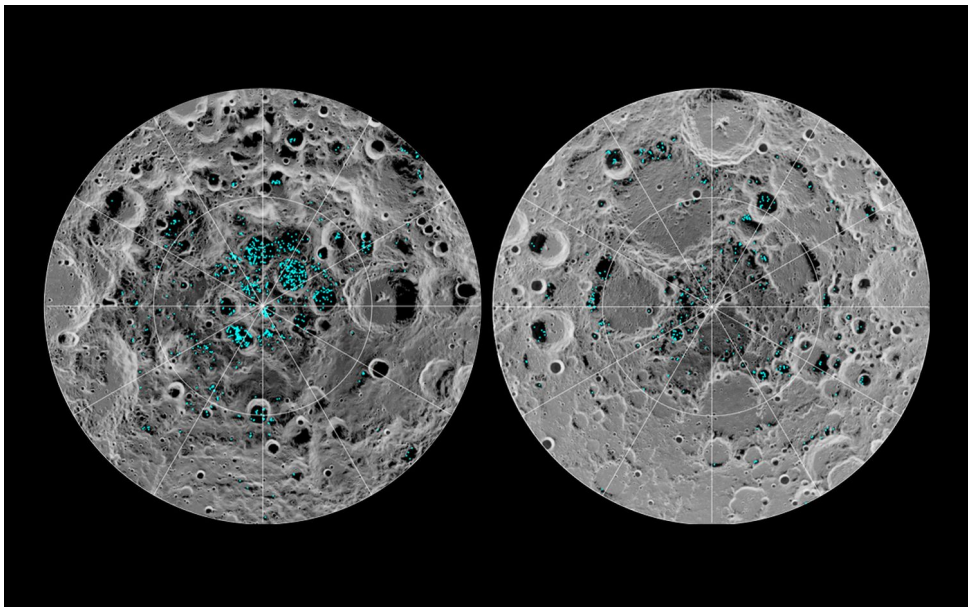


Figure 4: Left, south pole, right, north pole. Blue dots are likely locations for the presence of ice water based on remote sensor data. [2]

As the missions for our rover, if it were to be sent, would likely take place at the rim and center of one of the southern craters, a more detailed description of this environment is as follows. With the sun occuring at a low angle in the sky, the maximum temperature the robot would experience will be rather low and will only swing further lower when entering a crater. Further with the sun and earth at such a low angle, the stars will be a more important factor for navigation. Further along the line of lighting conditions, the low lighting may hinder our ability to detect objects, measure distant objects for localization, and other visual tasks also increasing the importance of stars for navigation. Finally, due to the lack of ground truth regarding the location of minable ice water, it is unknown how far exactly the robot will need to travel from the base to the mining site. In the likely situation that the base is located on one of the southern craters rim, our rover may only need to travel as short as one kilometer since, in this case, ice water is

present on the inner rim. It's also possible that the trip could be a couple tens of kilometers if the ice water is present in minable quantities only in the craters center. In either of these cases, we should be prepared for the varying trip distances.

Map and Path Planning

Function

Mapping and path planning are both fundamental subsystems of the rover. Three use cases are given below showing how mapping and path planning fit into navigation.

In the case of long range dead reckoning, an evolving short range map could be maintained for immediate path planning and obstacle avoidance. This is especially suited for long distance journeys where the rover's main objective is to get from point A to B rather than documenting the area it's traversing.

In the case of short-medium range navigation, our rover might have a detailed map of an area. The function of the map, in this case, is to act as a reference to localize from, which is done by comparing its sensor data to the created map. After obtaining possible matches, the best match or matches correspond to the robot's likely position. With this location and the location of surrounding obstacles in the map, an efficient path can be determined.

In the case of short range navigation, where a map is unavailable but the rover has a known starting position, the rover will need to perform simultaneous localization and mappings. In this case, SLAM utilizes this starting location, the motion of the robot, and what it sees to build a map of the environment while localizing the robot within this map. With the map updated and position estimated, this map can be utilized by a pathfinding algorithm to determine the most efficient path from the current estimated position to the destination.

Background

Navigation is the process of controlling the path a mobile platform takes from one location to another. In a rudimentary system, this could be performed by placing

predefined markers or lines along the mobile platforms intended paths and having the platform travel over the markers to the destination much like how bin robots move about warehouses.

For mobile platforms that must be able to operate with greater freedom but not necessarily with care about its current position, in this situation, dead reckoning is a very good candidate. It works by creating a temporary map from which it can do path planning.

For mobile platforms that are less constrained and must be able to operate with greater freedom with precise movements, the problem of navigation becomes more tricky but can be solved one of three ways. The first way is to treat navigation as a purely localization problem, where the robot has a highly detailed map with which it can localize itself and plan paths through. The second way is to treat navigation as a purely map building problem, where the robot has a method to accurately determine its absolute position absent the map. Which, knowing its position, allows the robot to create an accurate map with which it can use to plan paths. The third way is to treat navigation as both a localization and mapping problem (SLAM), in which neither a highly detailed map nor a method for accurately determining absolute position is available. SLAM does exactly as it says, it creates a map while allowing the robot to localize itself in that map. With this map it can then plan a path. In the case of our system, the first option is not possible as a highly detailed map of the lunar surface is not available therefore requiring the rover to map it first. Also, for the second option, our cosmic GPS system is likely to only give a rough absolute position, with relatively large errors. This would make our map highly erroneous if we treated this as only a mapping problem. Because of these limitations, simultaneous localization and mapping is the only method that would give us the ability to precisely determine the rover's location and the spatial positions of objects around the rover. From which, we could perform the most robust path planning (fastest and safest passage).

Map

How we describe or map the environment will have a large impact on how the path planning algorithm is implemented and how we obtain the resources required. So in this section we will discuss different types of map structures.

Fixed Cell

Fixed cell is a map type commonly used to represent a metric space. It is essentially a discrete metric map where each arbitrarily small point in metric space is assigned to the respective cell. These cells are identically shaped and fixed in size composing a grid of cells. A key reason fixed cell maps are used is that they can be naturally implemented using arrays. Therefore, this representation inherits all of the properties of the arrays such as constant time manipulation of individual cells and spatial qualities. However, this comes with the disadvantage of requiring extra memory for spaces that are unimportant. A common technique used for decreasing memory usage without impacting map resolution is to implement a composite map, where multiple maps are used each with a different fixed cell size. In the highest resolution map, it is composed of the metric space immediately surrounding the area of interest. With each level up, the cell size of the map increases with the total map size increasing as well. In this approach, the map is able to cover a very large space while still allowing details.

Quadtree

Quadtree is another metric mapping type, less commonly used than the fixed map type above. It seeks to solve the problem mentioned in the previous section, that being, fixed cell maps waste a lot of memory when representing large and identical regions. In the case of quadtrees, the map is divided into a recursive tree structure where each branch represents a quadrant of the map. This representations advantage is in being a single structure capable of representing maps in a very memory efficient manner. It does not require keeping track of multiple map layers, each of a different cell size, as is required with a composite map. The quadtree also has an advantage in its dynamic ability to change the size or detail of the map without having to recreate its whole structure, only branches need to be added or removed, as is needed with array based fixed cell representations. A downside of this representation though versus array based fixed cell maps, is the non constant runtime for accessing cells in the map. Depending on how large and detailed the map is, the average height of the tree may require many node traversals. Further, checking neighboring cells may require many tree traversals which are slower than the fixed cell representation. Also, for dense maps, quad trees may not only be slower than array based fixed cell maps, but also less memory efficient as many pointers are used.

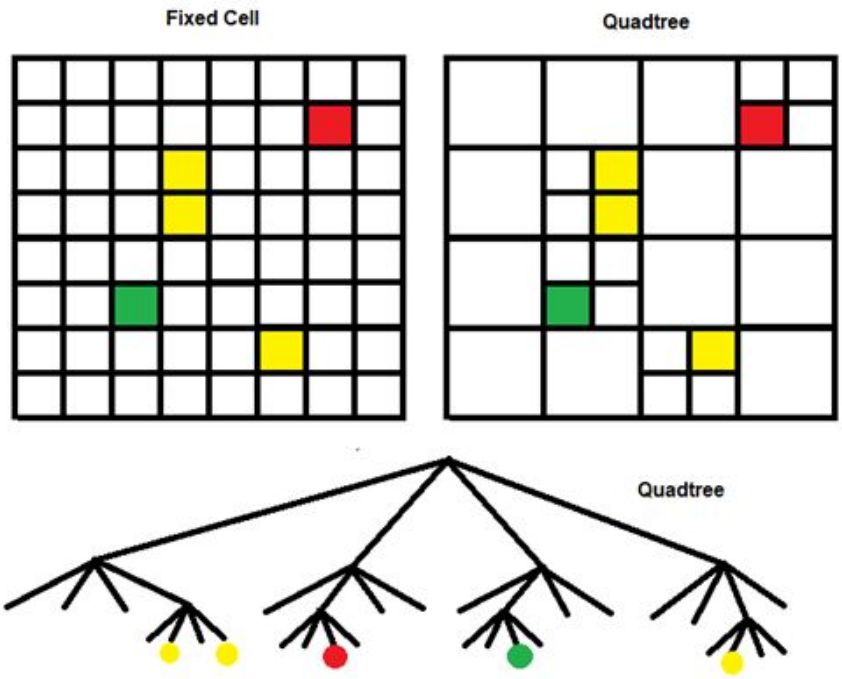


Figure 5: Fixed Cell vs Quadtree Map.

Initializing these maps from sensor data varies. In our case, a stereo-vision sensor is used, which can first find locations of the image that are apparently discontinuous from the background and treat these as obstacles. From measuring these obstacles width, height and distance and from knowing our position on the map, we can use triangulation to determine where the objects should be placed on the map. This is expanded on more in the object detection section. For our purposes we will assume the obstacles are given to us in terms of coordinates and then we will fill these cells with a high cost.

Path Planning

Now that we have discussed how we could represent EZ-RASSOR's environment into a cost map that we can use to determine good and bad paths; we will discuss some of the path planning algorithms that are applicable.

A*

A* (pronounced 'A star') is a popular path planning algorithm that is able to find the most optimal path given your starting location and the destination. A* is very

flexible and can be used in a wide range of contexts. A* is similar to a greedy, best-first search in that it can use a heuristic to guide itself through many situations. A* can be compared to Dijkstra's algorithm in that its performance is very similar. The reason that A* works so well that it combines different pieces of information that Dijkstra's algorithm uses like favoring vertices that are close to the starting point and information that greedy, best-first search uses like favoring vertices that are close to the destination.

A* can be written out algebraically as f(n) = g(n) + h(n), where g(n) represents the exact cost of the path from the starting location to any vertex n, and h(n) represents the heuristic or hypothesized cost from the vertex n to the goal. A* recalibrates g(n) and h(n) as it traverses towards the goal. Throughout each iteration of the algorithm, the lowest vertex n is traversed with the value g(n) + h(n). As you can already tell, the heuristic function h(n) is very important in the use of the A* algorithm so it's imperative that we choose a good heuristic function for EZ-RASSOR to use.

Heuristics

Before discussing some possible heuristics that EZ-RASSOR can use, we will first discuss more of the A* algorithm's functional properties through different heuristic values.

- If h(n) is 0, then only g(n) is used to decide the next vertex, and A* turns into Dijkstra's Algorithm, which is guaranteed to find an optimal path.
- If h(n) is <= the cost of traversing from vertex n to the destination, then A* is guaranteed to find the shortest path. The lower h(n) is the more vertices that A* expands, making it slower.
- If h(n) = the cost of moving from vertex n to the destination, then A* will only follow the best path and never expand anything else, making it very fast. Despite this performance benefit, finding perfect heuristics is not always feasible.
- If h(n) > the cost of moving from vertex n to the destination, then A* is not guaranteed to find the shortest path, but it can run faster.
- If h(n) is very high relative to g(n), then only h(n) plays a role, and A* turns into Greedy Best-First-Search.

Knowing all of the ways that our heuristics can affect EZ-RASSOR's path planning can help us better specify our heuristic function. Throughout the many different heuristics that we can give to EZ-RASSOR, we can fine-tune our algorithm to perform slow/fast or optimal/suboptimal depending on the given task. This flexibility is important for a space rover with many different missions.

Speed vs. Accuracy

Now we will discuss the trade-off between speed and accuracy with regard to EZ-RASSOR. In a perfect world, EZ-RASSOR would be able to run as fast as possible with perfect accuracy. Unfortunately, speed and accuracy usually have trade-offs in real-world scenarios which can be exploited to prioritize speed or accuracy. For most real-world situations, you don't always need the most optimal path between two vertices, you only need a path that's pretty close and that runs relatively fast. This ideology can be extended to some real-world situations involving EZ-RASSOR. One such example would be EZ-RASSOR attempting to reach a waypoint to mine regolith. If the algorithm to find the most optimal path takes a lot of CPU power and time, we will essentially be wasting time calculating the fastest path from each waypoint. On the other hand, if we have a slightly less optimal algorithm that still takes us to the final destination but computes the path in half the time, we will find ourselves saving a lot of time over the course of calculating many different paths.

Another real-world trade-off between speed and accuracy for EZ-RASSOR is path planning using different types of terrain. In an environment with different types of real-world obstacles like holes, mountains and low friction areas (ice, thin sediment) we must also consider if the risk of traversing these types of terrain is worth the increase in speed. Suppose EZ-RASSOR's environment has flat and mountainous terrains and suppose that the cost of traversing flat terrain is 1 while the cost of traversing mountainous terrains is 3. If we use A* in this environment, we will be searching 3 times as far along flat terrains as opposed to mountainous terrains. This is because there is more likely a path around the mountain that we would prefer. Thankfully, as previously discussed, we can adjust this 3:1 ratio by simply making the heuristic cost of traversing flat terrains 1.5. This will make searching the mountainous areas more likely. Having this kind of flexibility in an algorithm is definitely a plus.

Before discussing some possible heuristics we can use for this project, lets trace over a few example cases of EZ-RASSOR's pathfinding using the A* algorithm.

	F			
	X	X	X	
		14	10	14
		10	S	10
		14	10	14

Figure 6.

In the example above, let's suppose that S represents the starting position of our EZ-RASSOR robot and F represents our final position. Let's also suppose that X represents an obstacle that we cannot traverse like a large hole or steep-sloped area. Finally, let's say that each adjacent move for our robot is a cost of 10 and each diagonal move is a cost of 14. This logic is used considering that the length of the hypotenuse ($(\sqrt{a^2 + b^2}$) is smaller than the sum of the two other edges in a right triangle.

Let's also note that we do not know the values of each vertex until we traverse it as we would expect on the moon or other foreign celestial bodies. Since we are using A* in this example, we can rewrite each vertex as a function of h(n) and g(n) as we saw above.

Then we can write each vertex as the following:

	F			
	X	X	X	
		48 = 14 + 34	**48** = 10 + 38	**56** = 14 + 42
		54 = 10 + 44	S	**62** = 10 + 52
		68 = 14 +54	**68** = 10 + 58	**76** = 14 + 62

Figure 7.

each cell takes the form f(n) = g(n) + h(n), where g(n) represents the total cost from the start position to the current vertex, h(n) represents our heuristic which in this case is the cost of the current vertex to the final position and f(n) represents the sum of h(n) + g(n) which is the number we are using to choose our path at each step. Since we are trying to maximize our speed and accuracy, we will always choose the vertex with the smallest f(n) value.

Please note, in this simple example, EZ-RASSOR only has 8 possible places to move. In the real world, our robot will have many different directions to choose from and to avoid checking overlapping paths, we will attempt to have a similar amount of directions to choose from to limit excessive computation.

After choosing the smallest unvisited vertex, all of the adjacent vertices that can be reached from the new vertex will be assessed and given values based on their heuristics as seen below. Since we have multiple vertices with the lowest value, we simply pick one and move to that vertex.

	F			
	X	X	X	
	48 = 28 + 20	**48** = 24 + 24	**48** = 20 + 28	
	58 = 28 + 30	**48** = 14 + 34	**48** = 10 + 38	**56** = 14 + 42
	60 = 20 + 40	**54** = 10 + 44	S	**62** = 10 + 52
		68 = 14 +54	**68** = 10 + 58	**76** = 14 + 62

Figure 8: ___ - represents a node that we have visited.

Finally, looping through this process a few more times gives us the following:

62 = 52 + 10	**48** = 48 + 0 F			
56 = 42 + 14	X	X	X	**66** = 34 + 34
62 = 38 + 24	**48** = 28 + 20	**48** = 24 + 24	**48** = 20 + 28	**58** = 24 + 34
66 = 34 + 34	**58** = 28 + 30	**48** = 14 + 34	**48** = 10 + 38	**56** = 14 + 42
	60 = 20 + 40	**54** = 10 + 44	S	**62** = 10 + 52
	74 = 24 + 50	**68** = 14 +54	**68** = 10 + 58	**76** = 14 + 62

Figure 9 Most optimal path using A*.

This yields the optimal path for EZ-RASSOR to take to get to the waypoint. Despite this example's simplicity, it shows a lot of the important concepts that are involved with A*. One thing that should be noted is that in this example we didn't show any values for the obstacles. Behind the scenes, these obstacles will be assigned very large values to deter our robot from thinking about traversing them. For example, on top of the heuristics that we already used in the demonstration above, we would have an additional added cost for obstacles. We

will leave the explanation of how EZ-RASSOR will perform obstacle detection for a later section.

If holes get an extra added cost of 120, mountains and sloped terrains get an extra added cost of 50 + the angle of incline (0-90 degrees) and areas that we already visited get an extra added cost of 10 to reduce the possibility of EZ-RASSOR digging itself into the sand (areas of thin sediment). All of these extra obstacle heuristics will help our robot to safely and quickly get from start to destination.

Now that we have seen a simple example of how EZ-RASSOR will use A* we can finally talk about how a possible heuristic function will work. As mentioned above, EZ-RASSOR will have to take into account obstacles when deciding an optimal path to a given waypoint. When looking at obstacles, we should assess the following costs:

- Vertices that EZ-RASSOR has already visited will receive an additional cost of 10 to deter our robot from digging itself into the ground.
- Holes in the ground will receive an additional cost of 120 + the angle of decline (0 - 90 degrees).
- Mountainous or sloped terrains will receive an additional cost of 50 plus the angle of incline (0 - 90 degrees).
- Non-traversable obstacles like boulders, other EZ-RASSOR robots or our lunar base will receive a special heuristic that will add a very large cost that will essentially not allow our robot to traverse these objects.

Obviously this model and its costs will need to be tested and tweaked to better fit EZ-RASSOR's current objective. Now that we have seen A* work in a 2D environment, the question arises, does A* work in a 3D environment? The answer is yes, A* actually is not limited to 2 dimensions meaning that we technically are able to use it in our 3D virtual simulation. The 3D version of A* is called navigation mesh or NavMesh for short. Navmesh works very similarly to A* but instead of representing the map in a 2D grid, we now represent everything using polygons to cover the different topography of the map that an entity can traverse. If an obstacle should not be traversed by any means, it is possible by simply making sure that no navmesh is generated on those types of obstacles.

Essentially, think of the navmesh as every traversable vertex, thus EZ-RASSOR cannot explore areas not generated on the map.

Here is an example of what a navmesh looks like:

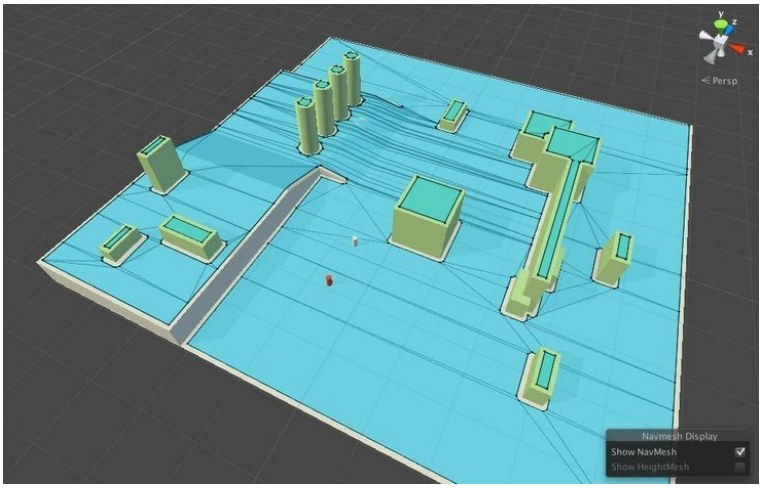


Figure 10: Map of 3D navmesh. [3]

Unfortunately, navmeshes are computationally heavy when dealing with other moving objects in the environment. If we decided to use a navmesh with a swarm of 10 EZ-RASSOR's then we would have to calculate an optimal path with respect to each other robots movement for each move that we take. This would slow down EZ-RASSOR's path planning tremendously which is not a feasible option. The other downside of navmeshes is the large memory footprint that is required while building up an optimal path. This is because we have to visit many different vertices with appealing f(n) values before reaching our final waypoint. Despite A* and navmeshes having large memory overhead and somewhat high levels of required computation when dealing with swarms of robots, there are still ways that these algorithms can be used in our project. These algorithms' main downfall is that they do not function quickly in an unknown local environment with a global objective. Regardless, there may be a use for these algorithms later on after our EZ-RASSOR robots have established localization.

We will now explore some other path planning algorithms to determine if we can find something more optimal to help EZ-RASSOR on its global objective in an unknown local environment.

D* Lite

A variant of A*, known as D* Lite was first described by Sven Koenig and Maxim Likhachev as an improved incremental heuristic search algorithm combining ideas from LPA* and Dynamic SWSF-FP. Like the original A*, D* works with a cost map, and using a heuristic, finds the most optimal path from start to goal. But it varies in a very fundamental way. In A*, the map is assumed to be known in its entirety even before the search has commenced. That means after a path is found and the robot is traversing the space, if a new obstacle is found on the map, the path A* had previously generated is void. There is no way for a robot utilizing A* to handle this situation other than to recalculate the path from the current position to the goal position. So, using A* with a map that is largely unknown with a short line of sight could carry large computational costs as the path would have to be continuously recalculated. D* Lite, on the other hand, is designed for these situations in mind where the map is largely unknown and for recalculating the path from its current position to the goal position.

In D* Lite the path is originally generated with the cells immediately surrounding the goal positions having a low cost. The search initializes the cells spreading from the goal position to find the path to start. From here, the robot follows these cells with each motion moving to a cell with a lower value than its previous cell. When an obstacle is found, the cells between the goal and the new obstacle do not need to change, only the costs associated with the cells after the obstacle need to change, as shown in the example below.

12	13	X	Goal - 0		**12**	**13**	X	Goal - 0
10	X	X	2		**10**	X	X	2
9	7	5	4		**9**	**7**	new - X	4
X	8	7	6		X	**8**	7	6
11	10	9	8	**->**	**11**	cur - 10	9	8
13	12	X	10		13	12	X	10
15	14	13	12		15	14	13	12
Start - 17	16	15	X		Start - 17	16	15	X

Figure 11: The highlighted values need to be updated.

Notice how in this example, an obstacle was found and it caused 7 of the cell values to be updated. This indicates that only a relatively small part of the map needs to be updated every time a new obstacle is found. Further, notice that the obstacle was not in the current path of the robot, which means the path of the robot does not actually need to change.

15	16	X	Goal - 0		**15**	**16**	X	Goal - 0
13	X	X	2		**13**	X	X	2
11	10	X	4		**11**	**10**	X	4
X	9	7	6		X	**9**	new - X	6
12	cur - 10	9	8	**->**	**12**	**cur - 10**	9	8
13	12	X	10		**13**	12	X	10
15	14	13	12		15	14	13	12
Start - 17	16	15	X		Start - 17	16	15	X

Figure 12: The highlighted values need to change.

In the case the obstacle is placed in the lowest cost path between the robot's current position and the goal, a new path will have to be generated to connect the current position with the goal. These two previous examples indicate that the purpose of D* Lite is to efficiently locate the cells, after a map change, that have changed and are relevant to finding the shortest path [4].

FlowField

Since we just saw an algorithm that had difficulty computing optimal paths with swarms and had issues with memory, let's talk about flowfield. Flowfield is an algorithm that works with multiple fields or layers of information forming a set. Flowfield is similar to A* and navmesh, in that, it uses costs at each vertex to determine an optimal path among waypoints. The main advantage of flowfield over A* and navmesh is that flowfield can handle large groups of swarm robots without large amounts of computation. Let's define the two flowfield layers as the cost field and the vector field. The cost field can be thought of as the cost or distance from the current vertex to the destination of each traversable node (similar to h(n) from A*).

Here is an example of what the cost field might look like for flowfield:

Figure 13: Example of a cost field in the flow field algorithm. [3]

Notice that in the example above, the cost from the pink point to the red point is a cost of 9 (yellow path) and the cost of the blue path is approximately 4.12; but, since it's going through an obstacle as represented in green, the yellow path must be taken. Next, the vector field can be defined as, for each node, the

direction to the closest adjacent node having a lower cost than its own and other adjacent ones. This is essentially just a vector pointing to every node's cheapest neighboring vertex. Each vector field is computed one tile at a time by looking at the cost field. The x and y components of the vectors could be computed using the following:

$$V_x = \textit{left tile distance} - \textit{right tile distance}$$

$$V_y = \textit{up tile distance} - \textit{down tile distance}$$

Here is an example of what the vector field might look like for flowfield:

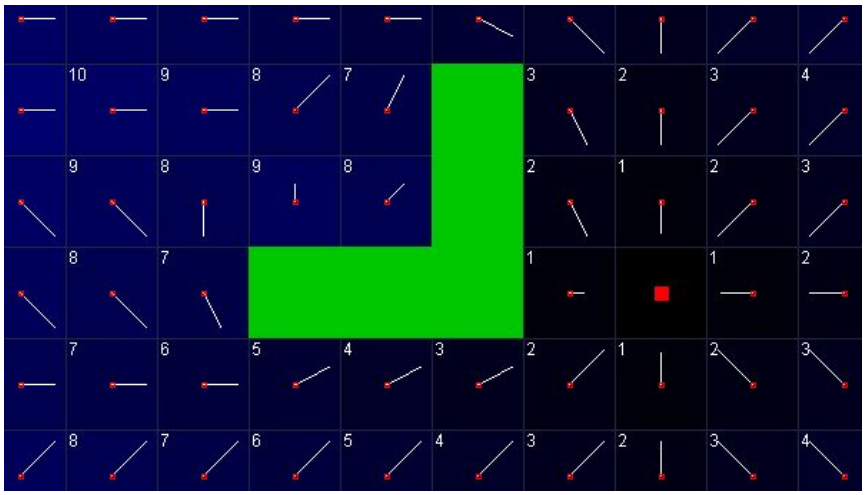


Figure 14: Example of a vector field in the flow field algorithm. [3]

This layer of flowfield is how the algorithm is able to handle large amounts of moving entities. Each EZ-RASSOR swarm that will be moving using this algorithm will simply follow the vector field to each waypoint, which allows for a lot less computation worrying about where each robot moves with respect to each other. Despite this benefit, flowfield requires a large amount of memory overhead for large areas and is not feasible when dealing with a single EZ-RASSOR system. Flowfield also requires us to have some prior knowledge of the terrain/obstacles. For this reason, the flowfield is still an unlikely choice as EZ-RASSOR's initial path planning algorithm as it will not scale well in an unknown environment.

Bug Algorithms

We will now discuss a family of algorithms known as the bug algorithms. These algorithms use similar logic to ants, bees and other insect-like creatures as they

traverse unknown local areas seeking a globally known objective like food or other resources. These algorithms seem like a good initial choice for EZ-RASSOR as our robot will be placed in an unknown environment with only its global waypoint initially known.

The first bug algorithm that we will discuss is called bug0. Bug0, although not given the most intuitive name, provides an interesting and safe solution to our problem. Things that we can assume with the bug0 algorithm:

- The direction in which our waypoint/goal is located is known
- We are able to compute the distance between some arbitrary points x and y
- We are able to sense and gather information from our surroundings

Now that we know the basic assumption for how bug0 works, we can talk about its path planning. The three main steps for bug0 are:

1. Head towards the goal
2. Follow obstacles until you can head towards the goal again
3. Continue this process until you reach your goal

An example of the bug0 algorithm process can be seen below:

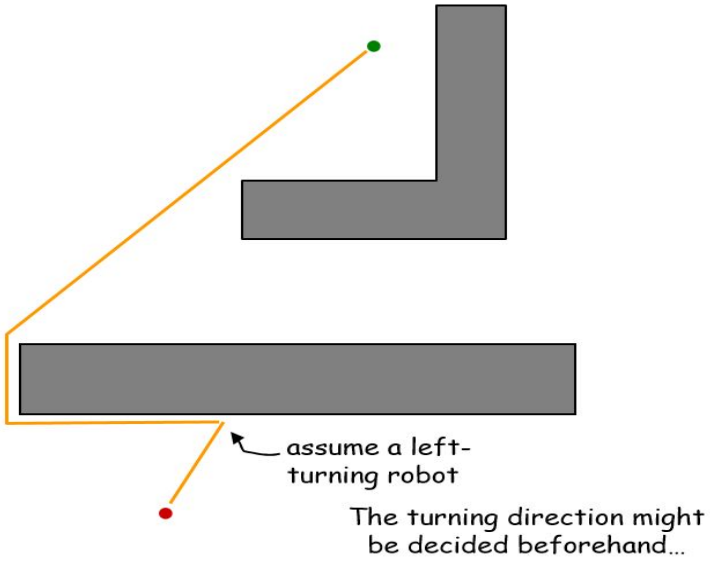


Figure 15: Example of the bug0 algorithm. [5]

Despite how simple this algorithm is, it performs relatively quickly in unknown environments as a result of low amounts of precomputation. It also will work in swarms if we ensure that we don't use the same path every time and we have proper protocols for avoiding other EZ-RASSOR robots. With an algorithm so basic like bug0, there will be unknown obstacles where we will endure the worst-case runtime. This can be seen in the map below, bug0 will continue to wrap around the obstacle in the left direction as a result of our protocol.

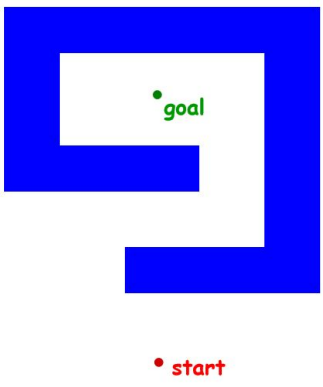


Figure 16: Worst case map for bug0 algorithm. [5]

This situation is one that we would like to avoid at all costs. We next discuss the bug1 algorithm, which adds more computation to avoid these types of scenarios.

The bug1 algorithm adds to the bug0 algorithm by adding some space for memory to things like sensing the environment and known direction to the goal. The bug1 algorithms process can be described as the following:

1. Head towards the goal
2. If you sense an obstacle, wrap around it and remember how close you get to the goal
3. Return to that closest point and continue

An example of the bug1 algorithm going from the red dot to the green dot can be seen below:

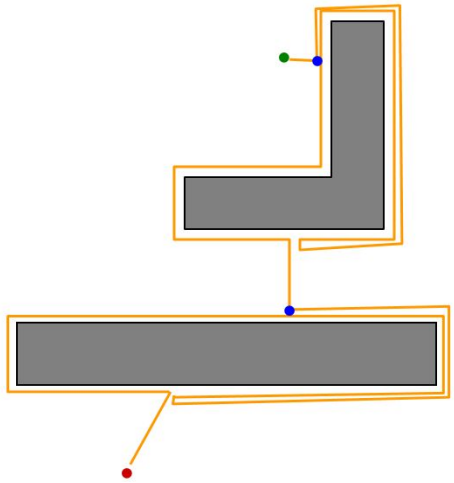


Figure 17: Traversal using bug1 algorithm. [5]

As you can see, bug1 is essentially an exhaustive search algorithm as it explores all options before committing. Bug1 has very predictable performance on average. The best case runtime for bug1 can be thought of as simply taking the straight line distance between the start and destination. In contrast, the worst case runtime can be thought of as the same distance in the best case plus the distance of each encountered obstacle's perimeter. In many cases, this will be an expensive algorithm to use with EZ-RASSOR, as we must consider fuel/battery consumption and time as a valuable commodity.

Now that we have seen how EZ-RASSOR would perform using the bug0/bug1 algorithms, let's discuss another approach and call it the bug2 algorithm. The bug2 algorithm now defines the straight line from start to waypoint as the m-line.

Instead of exhaustingly searching around each obstacle that we encounter, this time EZ-RASSOR will wrap around an obstacle until it reaches a point on the m-line and follow the m-line either until it reaches the waypoint or has to continue wrapping around another obstacle. The bug2 algorithm follows these guidelines:

1. Head towards the waypoint on the m-line
2. If EZ-RASSOR encounters an obstacle, wrap around it until it intersects with the m-line **closer to the goal**
3. Leave the obstacle and continue following the m-line towards the waypoint

Some examples of how EZ-RASSOR would traverse from the red point to the green point using the bug2 algorithm can be seen below:

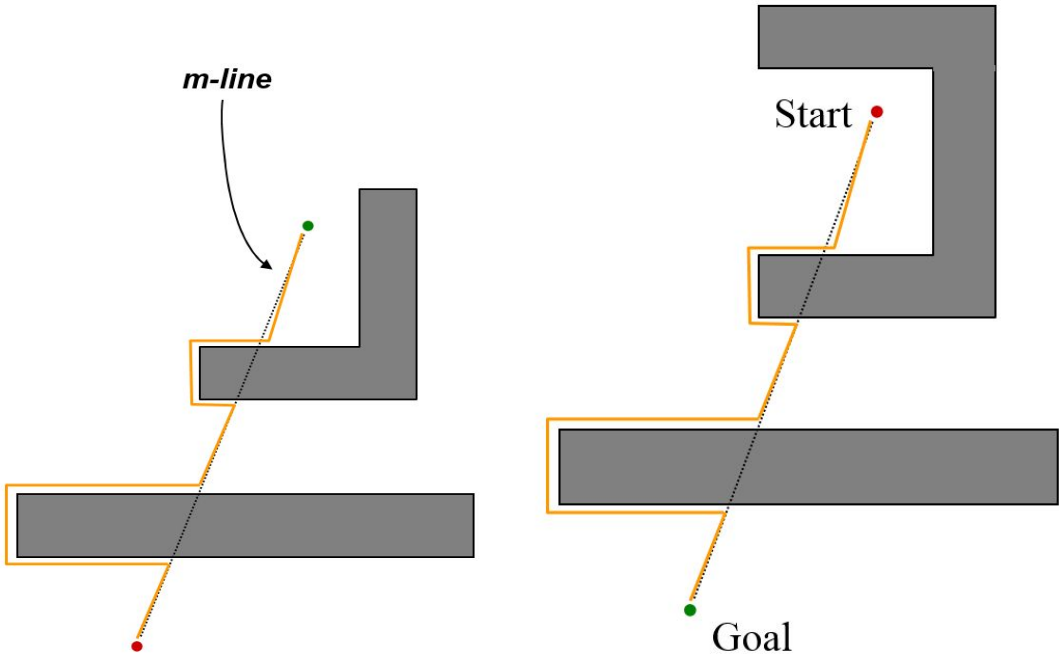


Figure 18: Examples of the bug2 algorithm. [5]

In the second example, if EZ-RASSOR did not check that the m-line was closer to the goal, we may end up repeatedly circling the starting obstacle an indefinite number of times. Bug2 is considered a greedy algorithm in that it will always take the cheaper option. In many real world cases, bug2 outperforms bug1.

After discussing the bug0, bug1 and bug2 algorithms, it's time to talk about a more realistic version of bug that utilizes a range sensing device. This stands out

for EZ-RASSOR because our robot is expected to have sensors like Lidar that will sense our environment and be able to provide us with valuable path finding data.

The final bug algorithm that we are thinking about using with EZ-RASSOR is the tangent bug algorithm. Tangent bug stands out from the other bug algorithms through the use of its range finding sensor comparisons. Tangent bug relies on finding endpoints of finite, continuous segments. At each step of the tangent bug algorithm, our EZ-RASSOR robot will sense all of the terrain 360 degrees around itself to determine the closest point towards the destination. In the image below, assume that x is the current location of our robot and qGoal is our final destination.

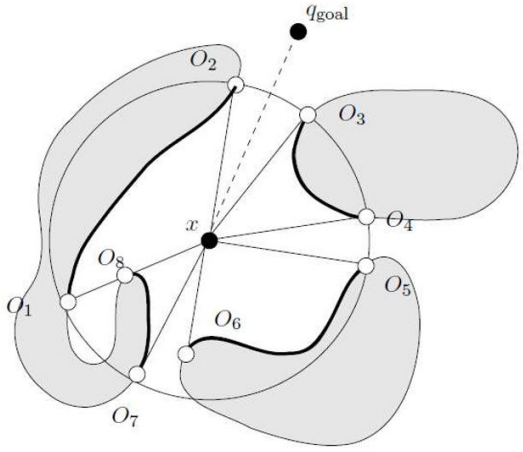


Figure 19: Example of how tangent bug finds its next vertex. [5]

In this situation, EZ-RASSOR would be able to simply traverse to the goal without having to avoid obstacles. If the path to the goal was blocked, the tangent bug algorithm would pick the O_i with the shortest distance to the goal. More formally, for any O_i such that the distance(O_i,destination) < distance(current location, destination), choose the vertex O_i that minimizes distance(current location, O_i) + distance(O_i,destination). Recall that at any point x, our EZ-RASSOR only knows what it sees through its sensors and where the final waypoint is. Below is an image where the tangent bug algorithm is used to get from start to destination using some finite range sensor.

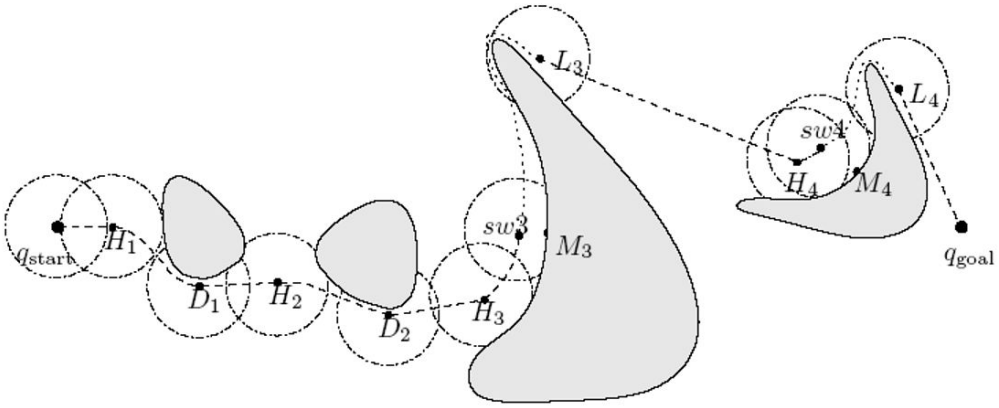


Figure 20: Tangent bug path planning. [5]

The tangent bug algorithm is a good fit for EZ-RASSOR because it would enable our EZ-RASSOR robot to path find in an unknown environment like the moon or mars and not suffer from tons of computational delay due to obstacles or other robots. This algorithm is also beneficial in that we can acquire environmental data from our surroundings through our sensors and send this data back to a global controller entity to aid in future path planning through heuristics/cost maps. This algorithm performs better on average than bug 0, bug 1 and bug2 with the only downsides being that it's more complex and requires sensors.

To conclude our discussion of the bug algorithms, we will discuss the pros and cons of each process. Bug0 is a very fast and simple algorithm but does not work well in many tricky environments. Bug1 is safe and reliable but exhaustive and slow. Bug2 is a greedy version of bug1 that avoids a lot of excessive travel but fails in some complex situations. Finally, tangent bug outperforms bugs 0 - 2 in most situations and uses information from sensors. Tangent bug is a great algorithm for exploration but uses more computation than bugs 0 - 2.

Now that we have fully discussed some of the possible path planning algorithms that EZ-RASSOR can utilize on the moon or on other environments, we will briefly discuss our thoughts on how we could use these algorithms with EZ-RASSOR in mind.

As mentioned above, A*, navmesh and flowfield all are very smart and safe algorithms but they fall short when it comes to computational overhead. When on the moon, time and energy are valuable and with these algorithms we will be squandering a good amount of these resources. These algorithms also require a

good amount of prior map knowledge which we can assume we do not yet have. With this in mind, these algorithms will likely not be used for initial exploration but have use in forming optimal paths once large areas have been traversed. On the other hand, the bug algorithms do not always find the most optimal path, but do not suffer from excessive computation when exploring the unknown and are relatively safe. With this information, there is a possibility of EZ-RASSOR using different types of path planning algorithms in different situations. For example, when exploring an area for the first time, EZ-RASSOR swarms will use something like tangent bug which will allow them to quickly and safely traverse the environment. Once a large amount of area has been recorded and applied to a cost map, other algorithms like flow field or A*/navmesh can be used to repetitively take the most optimal and safe paths to and from multiple points. As more and more data is collected, our maps will eventually turn into a highway like system that will speed up traversal.

Our overall goal for EZ-RASSOR when working on path finding is to find a fine tuned balance between speed, accuracy and safety when traversing.

Localization

In the next section we will discuss a method we can potentially use for better estimating our position, based on our sensor data and our understanding of the surrounding environment.

Monte-Carlo Localization (MCL)

MCL is a probabilistic method used for determining distribution likely positions based on sensor reading and how well those readings agree with the map. The process first starts, with an initial distribution of possible locations, represented by particles. After a time step with an associated movement, this movement is performed on each particle shifting the distribution. This new distribution is then filtered based on the sensor reading. For particles that do not agree with the movement, they are thrown out leaving only those that could plausibly exist. With each time step, the distribution changes accordingly, with either tightening or loosening of the distribution. A problem that can occur is when all the particles are filtered, when this happens, it cannot be reversed and localization is completely lost, which means the rover has to assume it is positioned anywhere. A frequently used technique to prevent this is to scatter a light cloud of extra

particles around the currently estimated distribution. These particles add some extra overhead but counteract the method's tendency to overestimate confidence in its location. These points also don't have any effect on the results as they are likely to be filtered in the next iteration. In the figure below shows MCL filtering conceptually, the blue particles are the final distribution of possible states after one iteration (not including extra added particles for the next iteration).

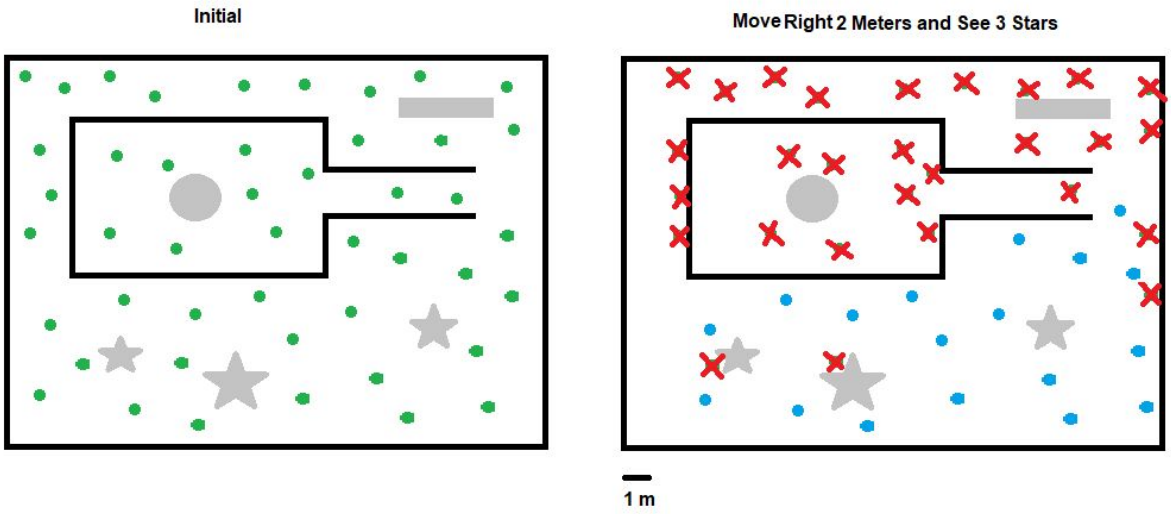


Figure 21: Green indicates initial distribution, Blue indicates final distribution.

In our system, we could use MCL two different ways. In this case, we have a map of an area in which we are trying to localize. And, for combining GPS-denied navigation and dead reckoning systems to better estimate our positions over a medium to long range journey [6].

Simultaneous Localization and Mapping (SLAM)

SLAM is a family of algorithms that seek to solve the problem of how to determine the robot's current position without having a map and how to create a map without the robot knowing its exact position. This problem is classically formulated using a two step recursive Bayes filter, where matrices and sets of the following properties are maintained: a vector **x**, holding the current location and orientation of the rover, a vector **u**, holding the information on the movement to vector **x**, a vector **m**, holding the time invariant position of an observed landmark, a vector **z**, holding the location of observed landmarks at different times, a set of

vectors **X**, holding the history of robot positions **x**, a set of vectors **U**, holding the history of robot movements **u**, a set of vectors **M**, holding the landmarks **m**, and a set of vectors **Z**, holding the history of observed landmarks **z**. From these variables the probability distribution of the current position and map can be described by $P(\mathbf{x}_k, \mathbf{M} \mid \mathbf{Z}, \mathbf{U}, \mathbf{x}_0)$, which can be solved in two steps (a time and measurement step) where:

Time-step:

$P(\mathbf{x}_k, \mathbf{M} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k}, \mathbf{x}_0) = \text{integral}(P(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{u}_k) P(\mathbf{x}_{k-1}, \mathbf{M} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k-1}, \mathbf{x}_0) d\mathbf{x}_{k-1})$

, and then a,

Measurement Step:

$P(\mathbf{x}_k, \mathbf{M} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0) = P(\mathbf{z}_k \mid \mathbf{x}_k, \mathbf{M}) P(\mathbf{x}_k, \mathbf{M} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k}, \mathbf{x}_0) / P(\mathbf{z}_k \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k})$

, where $P(\mathbf{z}_k \mid \mathbf{x}_k, \mathbf{M})$ and $P(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{u}_k)$ are the observation and motion models respectively. The observation states that the determined probability distribution of the observational model is dependent only on the current position and the landmark locations. In contrast, the motion model states the probability distribution of the current position depends only on the most recent position and the most recent movement from that position. This can be better understood conceptually in the following figure.

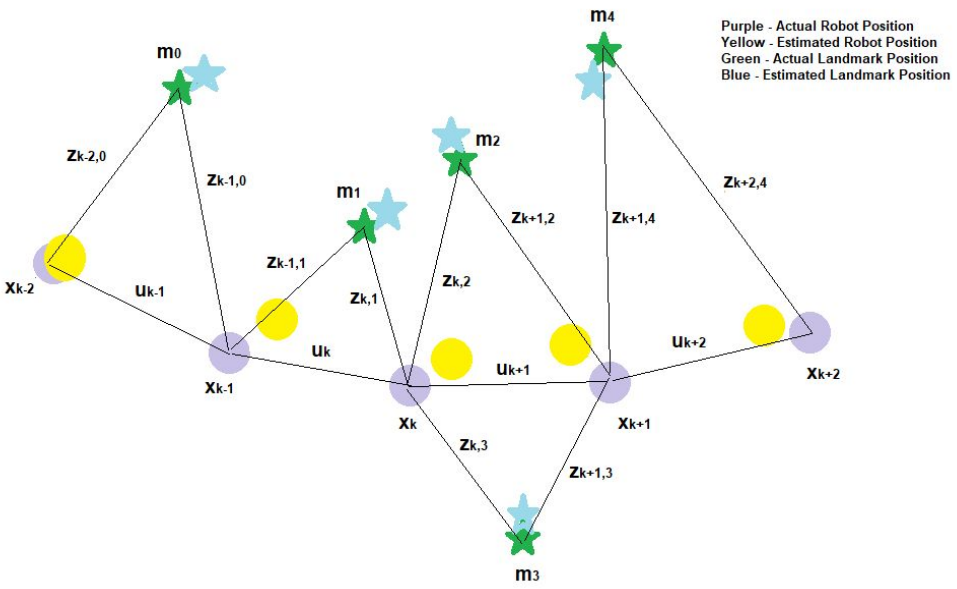


Figure 22: Conceptual SLAM diagram.

Because the same landmarks are measures between different positions, the estimated landmark positions become highly correlated over time. This ultimately

allows for the construction of a map that the robot can use to accurately localize itself, even though estimated positions of the landmarks are different from their actual positions. Now having discussed the problem and form of SLAM we will discuss two algorithms for solving the recursive Bayes filter above and their pros and cons [7].

EKF-SLAM

EKF-SLAM was one of the first methods researched in the 1990s that was developed to solve the problem of simultaneous localization and mapping. It is a feature-based method utilizing an extended Kalman filter and the highest probability for determining the estimated location of individual data (features). It most notably varies from FastSLAM in that it uses a Kalman filter. Something to note about the Kalman filter are two aspects: (1.) it assumes a gaussian distribution. (2.) it only works with linear models. These raise problems with utilizing this method in SLAM problems as the motion model is most often not a linear process [7].

FastSLAM

FastSLAM on the other hand also a feature based simultaneous localization and mapping algorithm was developed in the early 2000s. It is characterised by the use of a particle filter most commonly the Rao-Blackwellized Filter. FastSLAM is now the defacto standard of SLAM and has an advantage of EKF-SLAM in that it is capable of (1) assuming non-gaussian distributions and (2) it can work with non-linear models (although it still regularly linearizes them). FastSLAM when compared to EKF-SLAM is seen to give better probability distributions but for a slightly higher computational cost (though there are several optimization methods) [7].

Frontend SLAM

In the previous sections titled EKF-SLAM and FastSLAM, we discussed what is known as the backend of a SLAM system, which calculates probability distributions given sensor measurements. The frontend involves the sensor itself, which could be anyone of the sensors described in the object detection section below. The landmarks recorded by the sensors could likely be of point cloud form or in the form of features such as boulded, craters, etc. In the case of point cloud form, it would have the advantage of increasing the detail of the map therefore

decreasing the possibility of false data associations while also tightening the probability distribution. A disadvantage, however, would come in the form of higher computational costs, particularly in the case of the EKF-Filter as it requires every landmark (point in the point cloud) to be kept. With the landmarks being macro objects in the environment, it will have almost the exact opposite advantages and disadvantages. The computational costs would be much lower but at the cost of a less robust map with greater ambiguities [7].

Obstacle Detection

Function

The obstacle detection subsystem will work as a correctional assistant to the path planning subsystem. The sensor chosen for this task is to collect information on the local environment. The new information will inform how the EZ-RASSOR plans to move. This section will focus on what possible hardware the robot can use to collect data and how different options would affect the ability to avoid any obstacles.

Background

EZ-RASSOR will be following a path given a global map attempting to accomplish a global goal. In short, the goal of path planning is to find a path that gets the robot to its goal destination and does not waste much time or movement. This typically needs to be accomplished by looking at a cost map of the entire environment and predetermining an optimal route that minimizes cost. In some situations, it is conceivable that unforeseen costs exist in the environment which will hinder the robot from achieving its goal. Additionally, some path planning algorithms may incorporate or use only local data to find an admissible route. Clearly, collecting local data is an important task for EZ-RASSOR software. The key role in collecting this data is to detect obstacles in front of the robot and avoid them. Avoiding obstacles will overlap a lot with the path planning section, so it will not be the focus of this section.

For use on the Mini-RASSOR, there will need to be a variety of characteristics of a sensor to be considered. Possibly the most important factor is the price tag. Given this is an educational and developmental program, there will not be much

profit being made from the robot; thus, maintaining a manageable budget for the hardware is a high priority. Some sensors may be able to do everything and more that is needed for EZ-RASSOR, but it can not be seriously considered if it costs considerably more than other sensors that will get the basic jobs done with some extra work.

Another very important factor is the capability of the sensor. A sensor will need to collect enough data for EZ-RASSOR to infer meaningful conclusions. If the sensor does not inherently find the depth of objects in front of it, there needs to be a way to calculate it given the collected datum. The third must-have is that the sensor must physically fit on the Mini-RASSOR. Since that is the primary robot, this package is being developed for at the moment, any required hardware needs to be attachable to the Mini-RASSOR and not interfere with its function.

Ideally, the sensor should collect depth information in two dimensions. Knowing if there is an obstacle straight ahead of the rover is helpful, however, having a map of depths within a field of view would give us much more data to use. To a lesser extent, the range of how far the sensor can collect accurate data will be in question for all of the prospective sensors.

Data Collection

In order to add obstacle detection functionality to the rover, we'll likely need to add one or more sensors to its current design. Based on the data these sensors collect, we should be able to (1) detect obstacles along the path of the rover, and (2) determine the locations of the detected obstacles relative to the rover. In this section, we will give a summary of the sensors we researched and their advantages and disadvantages.

Sonar

Sonar is one of the oldest forms of distance-sensing that is still widely used today. Animals like dolphins and bats have used sound for object detection for millions of years, but the basis for the sonar we know and use today was developed during the early 1900s. Sonar for object detection works by sending out a sound, then listening for the echo. Based on how long it takes to hear the echo, the distance to an object in the direction the sound was sent can be calculated. The calculation a sonar performs to determine the relative position of

an object is given by the following formula, where *d* is the distance to the object, *V* is the speed of sound, and *t* is the time it took for the sound wave to reflect back to the sonar:

$$d = \frac{V * t}{2}$$

The equation is divided by 2 since the sound wave must make a roundtrip to the detected object and back (the wave travels *2d* in time *t*).

Figure 23: An ultrasonic sensor. Due to its small size and low cost, it is popular for obstacle detection in small-scale or low-cost robotics projects. [8]

For robotics, sonar typically takes the form of an ultrasonic sensor, shown above. Using ultrasonic sound rather than lower-frequency sound waves results in greater precision but a shorter range of detection. Compared to other forms of object detection that utilize light waves instead of sound waves, sonar is less precise, has a shorter range of detection, and has a shorter refresh rate [9]. Due to its short range, sonar is typically used for immediate obstacle detection (objects a few meters away) in robotics. However, because it has been around and studied for so long, it is able to be fabricated into relatively cheap and small sensors.

Besides its small size and low cost, sonar provides the benefit of being relatively unaffected by the conditions of the environment surrounding it, such as light or dust [10]. This would be a great benefit on the moon since dust will be kicked up constantly, but there is one problem: sonar would not actually work on the moon.

Sound cannot travel in a vacuum, so sonar would not work on a planet with very little atmosphere like the moon. While the Mini-RASSOR is designed as a research/demonstration rover and will never actually go to space, it would be ideal to keep the project in line with the spirit of the RASSOR, which could leave Earth one day.

Radar

Radar distance sensors use radio waves to measure the distance to an object. There are two main ways that a radar distance sensor can be implemented. The first is similar to sonar, except that radio waves are used instead of sound. Radio waves are sent out in a direction and the time it takes for the radio waves to be reflected back can be used to determine the distance to the object that reflected it. The second approach measures how much the frequency of the radio wave has been shifted after being reflected off an object. The sensor compares the frequency of the transmitted radio wave to the frequency of the reflected radio wave in order to measure the distance to the object that the radio wave was reflected off of (the more the frequency has shifted, the larger the distance) [11].

Figure 24: Tesla vehicles use a combination of cameras, ultrasonic sensors, and radar to provide a comprehensive view of the area surrounding the vehicle. [12]

Compared to other distance-sensing technologies, radar has a long range of detection and a wide field of view, allowing it to detect multiple objects at once [11]. It also is insensitive to surrounding conditions such as light, dust, or fog, making it ideal for robots that need to operate in a variety of environments. For these reasons, it is useful as part of a system that primarily relies on distance-sensing technologies that are more sensitive to surrounding conditions. For example, Tesla introduced radar as a supplement to its camera-based system for autonomous driving. Camera-based systems rely on computer vision to detect and avoid obstacles. If Tesla vehicles only used data from cameras to make driving decisions, how would they be able to operate in heavy rain or fog? Thus, radar is crucial to the system because it will still operate reliably in those conditions. For these reasons, Tesla has actually transitioned to using radar as the primary technology of detecting the vehicle's surroundings [13].

The main problem with using radar for our project is the price point. Distance sensors that use radar are more expensive than any of the other technologies described in this section (other than LiDAR). They are also not as widely available on most sites that sell robotics sensors, making it more difficult to select a model that is appropriate for our project. Thus, radar likely won't be of use for us.

LiDAR

The most accurate and popular sensor in obstacle detection today is LiDAR (Light Detection and Ranging). LiDAR is a distance-sensing technology that works by sending out laser beam pulses, then measuring the reflected light using a sensor. Depending on the implementation, the distance to the object will be determined using either the time it took for the light emitted from the laser to be returned (similar to sonar, except that it uses light instead of sound) or how the light wave was affected by the reflection [14]. Doing this for many different angles allows for a distance map of the surrounding area to be constructed; the distance of objects can then be determined based on that distance map. LiDAR works at long ranges and has high precision and a high refresh rate, which is why it is used extensively in the autonomous driving industry [9]. In the autonomous driving industry, LiDAR is typically used in the form of a spinning laser, so that a 360-degree view of the surrounding area can be formed.

Figure 25: Google's self-driving car, Waymo, uses LiDAR technology, which can be seen on top of the vehicle shown. [15]

For EZ-RASSOR, integrating LiDAR would make obstacle detection straightforward since it directly measures the distance of surrounding obstacles. However, due to its high cost, it likely is not feasible for our project. The Mini-RASSOR is meant to be a low-cost version of the RASSOR. The idea of adding a several-hundred-dollar sensor for obstacle detection would violate that design principle. However, there are cheaper LiDAR options available. One way to reduce cost is to use a unidirectional LiDAR instead of a LiDAR that spins to form a 360-degree view. The obvious disadvantage is that we no longer have a 360-degree view of the surrounding area, but if the LiDAR is facing the direction the rover is heading, then obstacle detection could still be performed.

Infrared Sensor

Infrared distance sensors, like LiDAR, emit light in order to measure the distance to an object. However, unlike LiDAR, infrared distance sensors usually use triangulation to calculate distance.

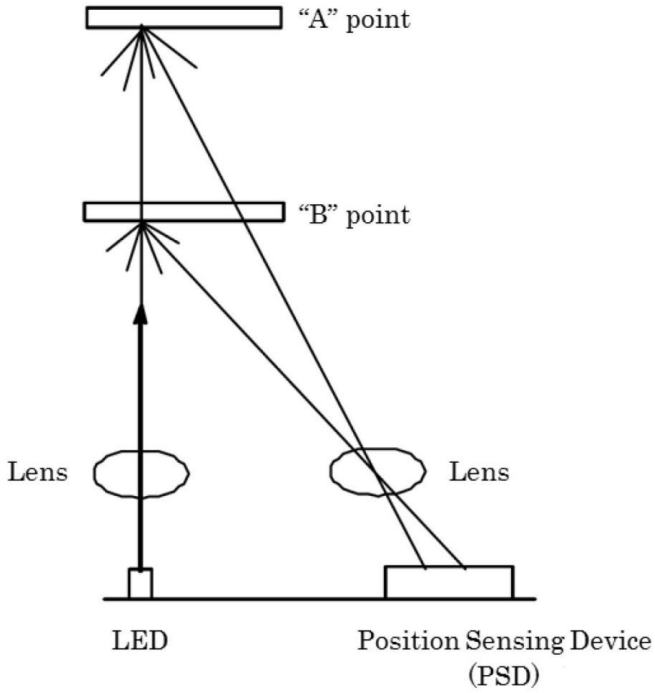


Figure 26: How infrared distance sensors work. [16]

As shown in the diagram above, light is emitted from an infrared LED in the direction that we want to measure the distance of an object. The light is reflected off of the object and onto the Position Sensing Device. The Position Sensing Device precisely measures the spot on the device where the light was reflected onto. Based on this position, the distance of the object that the light was reflected off of can be calculated [16]. Essentially, the PSD forms a right triangle between the point the light was emitted from (the LED), the point where the light was reflected (the object), and the point on the PSD that the light was reflected onto, then calculates the length of the side of the triangle that connects the LED and the object; this gives the distance from the infrared sensor to the object.

Compared to other distance sensors, these infrared sensors are extremely cost-effective, fairly precise, and have a decent refresh rate [9]. However, they have a short range of detection and are sensitive to the reflectivity of the object and light conditions (such as interference from sunlight). Due to their sensitivity, they are mainly recommended for robots that will be operating indoors. Our rover obviously is not meant for indoor use, so this type of sensor is not ideal for our use case.

However, there is a more recently developed implementation of infrared sensors that do not face the same type of sensitivity problems. These sensors are often marketed as “LiDAR” (even though they use LEDs, not lasers), and they use the time it takes for the photons from the emitted light to reflect off the object and return to the sensor (similar to sonar, except using light instead of sound) instead of triangulation [17]. These have a longer range of detection and are more reliable than the infrared sensors discussed previously but are more expensive. Compared to LiDAR that uses lasers, they have a shorter range of detection and a slower refresh rate but are significantly more cost-effective [9]. Since LiDAR that uses lasers is likely too expensive for our project, this “LiDAR” that uses infrared LEDs could provide a cost-effective alternative, although it is still more expensive than most of the other options listed.

VCSEL Sensor

A vertical-cavity surface-emitting laser (VCSEL) sensor is a distance-sensing technology that, similar to LiDAR, sends out laser beams in order to measure the distance to an object in a specific direction [18]. To calculate the distance to the object, it measures the time it took for the light emitted from the laser to be reflected back, then performs the same calculation as shown earlier for sonar, except that VCSEL sensors use the speed of light instead of the speed of sound in their calculations. Compared to other distance-sensing technologies, sensors that use VCSEL have high precision and are low-cost, but they have a much shorter range of detection [9]. Due to its extremely high precision at close ranges, VCSEL is often used in computer mice and for face identification.

For obstacle avoidance, VCSEL is not ideal since it will only give the rover a couple of meters to make adjustments to avoid the obstacle (however, testing will need to be done with the Mini-RASSOR to determine how much of a warning it needs in order to reliably avoid obstacles; due to its slow speed, it may not need much warning). The main benefit this sensor would provide is that it could assist in map creation and path planning in addition to obstacle detection. Since VCSEL has such high precision, it could be used to determine a precise relative location for detected obstacles. Due to its low cost, it may be worth exploring VCSEL as an addition to other obstacle-avoidance sensors. The other sensor(s) can be used for intermediate or long-range obstacle detection so that an alternative path for the rover can be planned. Then, once the rover gets closer to the obstacle,

the VCSEL sensor can be used to report a precise relative location of the obstacle.

Microsoft Kinect

Microsoft's Kinect is a motion-sensing technology that was originally developed for gaming applications (first for Xbox 360, then for Xbox One). It uses time-of-flight (ToF) to calculate the distance to many points in the surrounding environment and then forms a 3D map of the environment based on the collected distance data [19]. Essentially, the Kinect sends out beams of infrared light in many directions and measures the time it takes for the light to be reflected back to the Kinect sensors; based on the time it took for the light to be reflected back, the distance to the object that the light was reflected off of can be calculated.

It turned out that the technology was not as popular for gaming as Microsoft had hoped, but instead, the Kinect gained popularity with researchers and roboticists. The Kinect generates 3D maps of the environment around it; using these generated maps instead of building their own reduces the amount of work for researchers and roboticists. Microsoft has since discontinued its gaming Kinect line and has now released Azure Kinect, which is meant specifically for artificial intelligence applications.

Figure 27: Microsoft's Kinect, originally designed for gaming applications, has found popularity in computer vision research and robotics due to its ease of use. [20]

We were initially attracted to the Kinect due to the ease of use of the 3D maps it generates, as well as the resources available online about how to develop computer vision applications (such as obstacle detection) with it. It would allow us to quickly prototype obstacle detection and avoidance methods without having to focus on the hardware or the signal data that is being collected. For a group of software developers, this is ideal. However, the Azure Kinect is well out of our price range. By using a used Kinect for Xbox 360 or Xbox One, we can reduce the cost significantly. It would still be more expensive than many of the other technologies discussed here, but the benefits it adds could make it worth the cost. The main problem with using a gaming Kinect on the Mini-RASSOR is the size of the Kinect. The Kinect is too heavy and likely would not even fit onto the Mini-RASSOR.

Stereo Camera

Cameras are a popular data collection medium because it provides a two-dimensional array of one or three-channel datum. Unlike some other data collectors, this means that the cameras can find dangerous drops in topography along with imposing obstacles in front of the robot. Cameras are also the only collectors discussed here that do not inherently provide distance values for the objects they detect; so, a process to find the distance of obstacles will be detailed in length. The information collected by cameras can be used to segment, categorize, and track objects in its view. This means that a camera can find a bounding box for an object, figure out what kind of object is in the bounding box, and follow the movement of the object across the screen as either the object, robot, or both move. The same strategies can be used to find obstacles in a distance and calculate how far away they are from the robot/camera. The following paragraphs will detail how such an algorithm would work to find the distance from objects the Mini-RASSOR is using stereo cameras.

Today’s cameras capture images with millions of pixels to provide clear and precise visuals that are easy for humans to see fine details in the field of view. These images are pleasant to look at and provide tons of data to analyze, but going through so much data can be computationally expensive and slow. Having so much information may not be necessary for computer calculations. A solution to reduce the time to compute is to downsample the images. This means that the resolution of the images will be reduced in order to perform fewer calculations across the image. This will also result in decreased position variance, which will

hurt the accuracy of the distance calculations; so, the number of images are downsampled needs to be no more than a moderate amount. To perform downsampling, convolutions are performed to the image. This means that local sets of pixels are represented in a new image via the weighted average value of the group. The weights are determined by a small, two-dimensional kernel that slides across the image. The values in each part of the kernel represent the weight of the pixel it is overlapping in the sum value. If the sum of the weights is 1, then the weighted sum will be a weighted average of the pixels. The value is assigned to the position in the new image which is aligned with the center of the kernel. Increasing the size of the kernel will result in decreased resolution. Performing multiple convolutions creates a pyramid, where the original image is the base and every layer up is the same image with a smaller resolution.

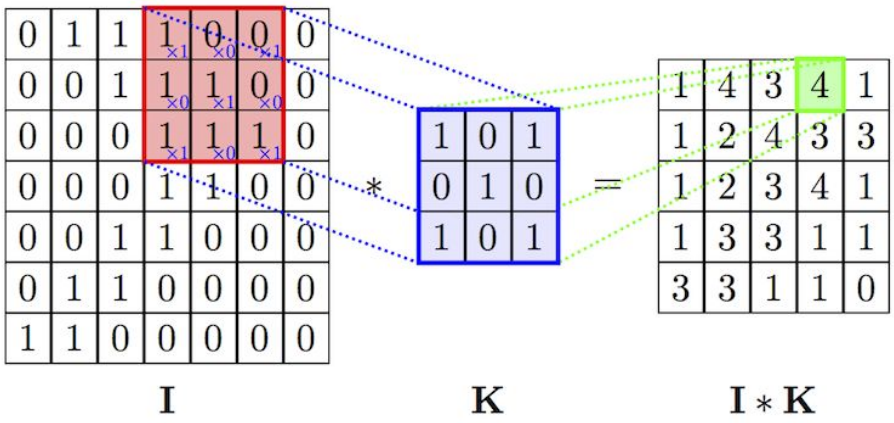


Figure 28: Convolution with 3x3 Kernel. [21]

The stereo cameras will need to segment objects in its view in order to calculate its distance from the robot. The first step to finding objects in an image is to find where there are edges in the image. Edges can result from objects overlapping each other or an object partially covering a background. A fast and reliable method to create an edge image is called the Difference of Gaussian (DoG). Before explaining this operator, it is important to know what a Gaussian blur is. This can be applied to an image by convolving the image with a kernel of values determined by Gauss' equation $\frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$ where x and y are the coordinates of the value in the kernel and σ is a desired standard deviation. The resulting kernel will emphasize values closest to the middle of the kernel and exponentially decrease the weight of values farther away. Blurring an image with this kernel is often advantageous because the original value of the pixel has the

most weight in the convolution, so the resulting image will be more accurate to the original than a flat kernel. The DoG operation is done by twice applying a Gaussian convolution on the image with two different σ-values and subtracting the result of one from the other. If done on a gray-scale image, the result of DoG will produce an image where pixels with higher intensities signal a higher chance of an edge existing there [22].

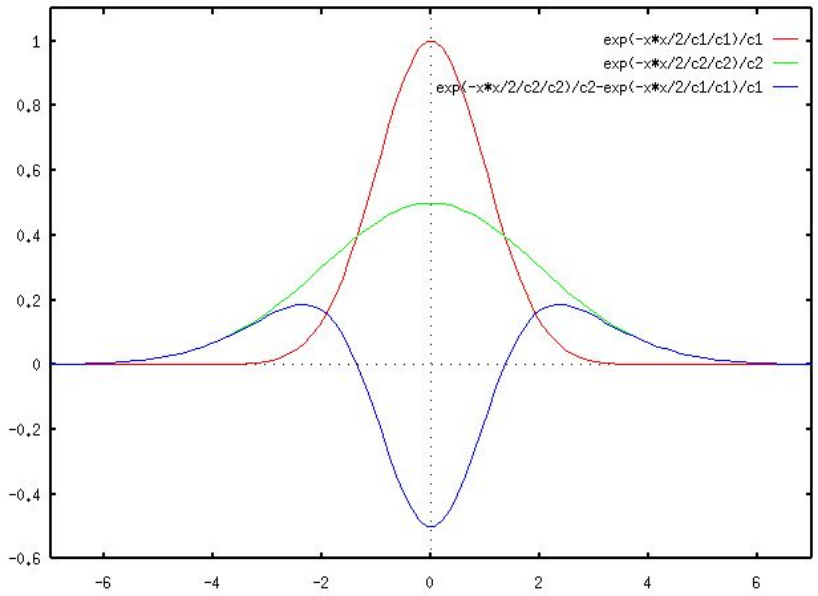


Figure 29: The Difference of Gaussian with Different σ-values. [22]

There are several methods to increase the precision of edges in the images, including non-max suppression, thresholds, and double thresholds. The use of these methods may be unnecessary for this project because edge detection is not the final goal and they will increase the computation time. For the sake of considering all options, it is still appropriate to quickly summarize these methods. Non-max suppression is accomplished by finding x and y derivatives of the image and using arctangent to find the direction of edges in the image. By using the direction of edges, edges found using the Difference of Gaussian will be removed if they are not the strongest part of the edge perpendicular to its direction. This results in a single-response (one-pixel length) only where the edge is most likely to exist. A threshold can then be used to only keep edges above a certain intensity. Another option is to use a double threshold. In this case, an edge will be considered strong if its intensity is above the higher threshold, weak if it is between the thresholds, and not an edge if it is below the lower threshold. Using a blob filter, weak edges will be suppressed—along with non-edges—if

they are not connected to a strong edge. Both thresholding methods can be adjusted to result in either a binary or grayscale edge image.

The next step in distance calculation is matching segments in the images captured by the two cameras. One approach is to find good features to track in one image (ie. the right image) and find them in the other image (ie. the left image). This can be done in a similar fashion to convolutions. A correlator function can be used over a sliding window of a set size (ie. 7x7 pixels) to calculate a score for how likely the key features are within the window. The result will be a two-dimensional array of matching scores. The highest score is most likely to be the location of the object trying to be found from the first image. If there are several scores close to being the highest and one score is significantly closer to the position of the object in the first image, that one is probably the most likely to match the query object. Since the cameras are so close to each other, it is a good option to suppress scores that are on the opposite side of the middle of the image. Also because the cameras' position relative to each other only in the x-direction, the objects should only differ in the x-position as well. So, a good way to prioritize high scores is to only consider ones in a relatively same y-position. Another way to prune the scores is to use non-max suppression. This will only keep values that are stronger than all the scores adjacent to it. One way to optimize the correlator is to use a sliding sum. This is implemented by saving the matching scores in a window to a two-dimensional array and reusing all the values in it except for the far left column (for a window sliding to the right). Consequently, the correlation calculation is only performed for a one-dimensional array of size *n* instead of an *n* x *n* two-dimensional array for each window. The sliding sum does not help for the first time calculating a window in its row because the previous calculation does not overlap it. This could be helped by moving the window in a snake pattern, instead of a typewriter pattern.

Once the position of an object is found on both cameras, its distance from the robot can be triangulated. When objects are farther from the cameras, the displacement between cameras will be small because the relative position of the object will not change much. Conversely, an object up close to the cameras will produce a large disparity. The basis of this idea will be used in how distance will be calculated. An accurate formula to estimate the distance (z) of an object is to use the formula $z = \frac{b \times f}{x_2 - x_1}$ where b is the baseline distance between the cameras, f is the focal length of the cameras, and x_1 and x_2 are the x-positions of the objects

in each image [23]. This formula is consistent with the inverse relation of distance and difference in position between images. After the distance has been found for the most prominent features, a three-dimensional map of objects at their average x-y position and z distance from the robot can be made. An object which is far away will have a greater z-value than those which are close. This means that the resulting depth image will be using a left-handed coordinate system.

Another option when using two cameras is to interpret a point cloud which describes points via an XYZ-Cartesian coordinate system. Using the Pythagorean Theorem, depth for each point can be calculated, similarly to pixels on a depth image. These points can be grouped by their angle to camera. Combining the angle grouping and distance calculations essentially converts the Cartesian coordinate system on a polar one. Research shows that these points can be used to detect cliffs, holes, and above-ground obstacles [24]. Cliffs can be simply found by creating a laser scan for the farthest point in each direction of the point cloud. A laser scan is a one-dimensional array of distances. In said array, each index represents a consecutive list of angles that the robot may turn and travel toward. Holes are detected by projecting points beneath the ground to the floor-level height. Then, a laserscan is created by taking the minimum distance values from all of the floor-projected points. Next, above-ground obstacles can be detected by creating a laserscan based on the closest distance in each direction before a point is taller than an arbitrary amount of the ground. Finally, a complete laser scan may be created by taking the minimum values in every direction from each of the three prior laser scans.

Single Camera

As demonstrated above, calculating the distance of obstacles using a stereo camera is a complicated process. The same objective is considerably harder using a monocular camera. The human brain is able to perceive distance in a single picture; however, computers do not have the same training by default to interpret the point of view and distance based on the expected size of an object and perceived size in the image.

There are two main approaches to estimate the distance of objects in view of a camera. One is to use machine learning to learn to recognize objects and use their expected size to calculate how far away it is. An example of this is training a computer to analyze how far apart the two pupils on a person’s face are and

correlate that with the ground truth of how far away the face is from the camera. After learning from a sufficiently large dataset, the computer had developed a model that takes input of an image with a face, determines the distance in pixels between the pupils, and outputs an estimated distance that the face is from the camera. This can be a very useful and accurate way to use a single camera to calculate the distance of a specific kind of target.

The drawbacks of this method are plentiful. First, there needs to be an abundance of data with ground truths to allow the machine to learn from. Since the EZ-RASSOR software will be employed in unexplored environments, there are not many pictures with objects that the robot will encounter as it traverses the moon and other rare environments. Also, there needs to be a common feature of the objects in the dataset to focus on. This means that the machine can not dynamically find the distance of unseen, unusual objects (ie. training on faces does not allow the machine to perform the calculation on walls, cups, or other objects). The EZ-RASSOR will need to be able to calculate the distance for several kinds of objects (ie. rocks, craters, hills) and this method will make it difficult to recognize unseen kinds of obstacles. Training for new objects on-board the robot is not an option due to constraints and priorities for the lightweight hardware.

The second and better option for a single camera in this application is using the assumption that the ground is flat and calibrations of the camera to measure the distance of an obstacle from the camera. Autonomous cars sometimes use a method like this if they are trying to avoid adding additional software to collect distance data. To make this method possible, the camera must be calibrated to an appropriate height and angle toward the ground. These values must be known by the software and constant in order to accurately estimate the displacement. This is done by counting the pixels between the bottom border and the bottom of the object [25]. This method still requires the segmentation portion done in the stereo camera method described above.

There are some pretty big concerns with this approach. First, is that the camera on the robot might not be configured correctly. A slight difference in the expected height or angle of the camera will throw off all the calculations and the software cannot dynamically attempt to correct that because the values are hardcoded. A similar problem is that the camera may be correctly configured, but the camera

can be jarred from its intended position from an extraneous action through usage in the field. It is possible to fortify the position of the camera, but impossible to promise that the camera will definitely never move. A third problem is that the assumption of a flat ground could be false in some areas. If the ground begins to curve up or down, then calculations will be off based on the angle of the curve. Detecting this curve will be impossible to do using a single camera because the ground does not typically look any different than a flat ground in an image when looking down at it. A final pitfall for using a single camera in this way is that the range is limited. In order to find objects up close to the rover, the camera will need to be tilted far enough down that it cannot see the skyline if the field of view is not wide enough. This could also mean that the camera needs to be placed awkwardly forward on the robot, which could affect the balance of the robot.

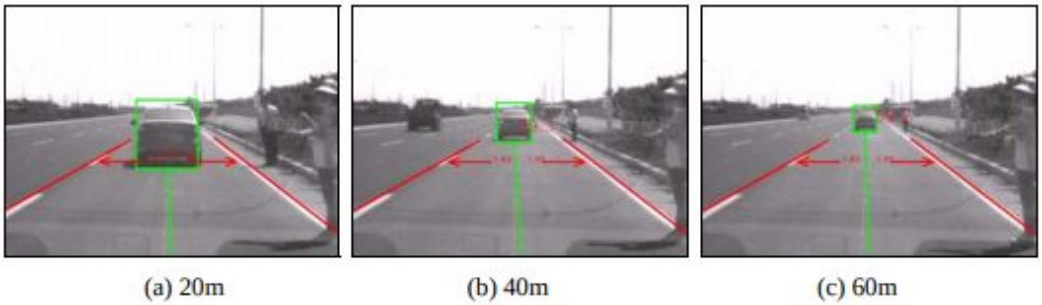


Figure 30: Distance from car on a single camera. [25]

A single-camera has the benefit of being cheaper than a stereo camera because it is less of the same hardware. There are also a lot more options for single cameras that means the required specs will be easier to find at a reasonable price. However, the implementation of distance calculation is significantly harder for a single camera and the difference in price is likely worth it for ease and accuracy.

Obstacle Reporting

The previous section on obstacle detection focused on studying different ways to measure the distance of objects in front of the robot. In the proposed design subsection, a detailed explanation explains why a stereo camera is best for this project. The result of using a stereo camera is a depth image can be created,

given three-dimensional information on the environment the robot is facing. This map does not provide any further observations on the information it contains.

There are a couple of reasons why analyzing the depth image to identify obstacles could be helpful to this and other projects. Primarily, the reason is that finding objects within the depth image could potentially be used for a visual odometry system, aiding in the localization subsystem. The benefits of different tracking methods are discussed in the Odomerty subsystem section; but, one idea involves finding objects in the view of the stereo camera and following where it goes in the depth image. The change in the three-dimensional position of an object in the depth image, which should be stationary, can be used to imply movement by the robot.

Less related to this project, a separate group working parallel to this one is working to create a cost map to inform coarse path planning for groups of RASSOR rovers. In practice, the cost map will be instantiated using vague information. This is why obstacle detection is important in the first place because there can be unexpected obstacles along a decided path. The cost map can be updated with more precise and accurate information if obstacles are identified, measured, and reported to the other group's system.

The process to find objects in the depth image to track and report is not simple. The first step is to find where the obstacle is segmented in the x and y direction in the image. Then, the distance from the camera of each obstacle is easily found. The width in pixels can be used with the distance from the camera to formulate the actual size of the object. Next, classification can be used to identify what kind of object is found. Finally, the position of the obstacle can be found, relative to the robot, that can be reported to identify where it should be mapped to in lunar coordinates, given the robot's lunar coordinates.

Noise Reduction

The first step to finding objects in the depth image is to refine the depth image. The most important reason to refine the depth image is that the calculations will result in noise in the map that will make analysis difficult. Some popular noise reduction techniques include filtering using a statistical average: mean, median, or mode. This means that a sliding window will be used to reassign the value of the center pixel at each position of the window as it moves along the entire

image. For instance, a mean filter would use n x n window to slide across the image, centering on every pixel possible without moving the window out of bounds of the image, and set a new value of the center pixel as the mean average of every pixel in the window.

In this practice, mean filtering is likely not a good option because edges would blur too much since a set containing some high values and low values with a gap in between would result in the center pixel being set to a value in the gap of values that represent neither the object nor its background. Instead, a median or mode filtering system is more appropriate. Both would do a better job of maintaining strong edges; however, each still has its own pitfalls. The median filter could be centered on a pixel, which should be on part of the edge of an object but will be assigned as the distance of the background because the edge is convex. This happens because more pixels in the image will be a part of the background than the object, so the middle value will belong to the background. Alternatively, the mode filter will struggle if most of the values are similar, but none of them are the exact same. This could result in a center pixel, which was incorrectly calculated as distant, not being corrected to close because none of the pixels around it matched values, even though they were all around the same value.

Recall that the depth image is created using the observation that displacement between features in the two images has an inverse relationship with the distance of said feature from the stereoscopic camera. Since digital cameras use discrete values for the resolution of their images, there comes a point when far away objects become rapidly increasingly more imprecise with the distance calculation. This is obvious once it is considered that a change over a single pixel could actually represent a change within a range of continuous true movement values. Due to this fact, distances calculated as far away may be unreliable to detect real objects at their real sizes. So, another way to refine the depth image is to calibrate to only consider objects within a certain range. All values above a certain threshold will be set to NaN (not a number) to essentially say that any object in that direction can be considered infinitely far away and should be ignored for the purpose of reporting and tracking obstacles [26]. The value for the threshold has some interesting ways to be calculated. This threshold will likely be determined by a set distance from the camera that is known to be inaccurate for its model, or only the closest k-percentage of the values will be considered. The

value of k can be adjusted based on what seems to be working or is important. It might be that the camera is very accurate at a distance due to high resolution, so 90% of the values may be acceptable. Conversely, maybe the stereo camera's resulting depth image has poor reliability, so only the closest 50% of obstacles will be considered. This value depends a lot on the quality of the hardware being used.

Bounding

Initially, the depth image holds a z-value (depth) at each discrete x-y value (width and height) in the stereo image. Higher z-values signify farther away objects in that direction. There is no inherent grouping of pixels happening to describe where the edges of distinct objects in view begin and end. So, the first step of obstacle reporting is to bound the positions of objects based on similar z-values near each other. The result of this function will be sets of limits in the x and y directions that hold an entire object within them.

One may assume that the segmentation methods discussed in the Absolute Localization section can apply to the segmentation of obstacles in a depth image. However, this is a much more complex problem than what is being accomplished in the other subsystem. The Cosmic GPS subsystem is using a monocular camera oriented toward the sky to capture images of celestial bodies (stars). The stars will appear relatively flat upon a relatively flat background, all of the stars will look relatively similar in intensity, and the intensities of stars are distinct from the intensity of its background. The problem with segmenting a depth image created by a camera which looks upon a landscape is that the objects will not always be flat, the background will never be flat, objects will be represented by many unique intensities (distances), and the distance of objects will often have a smooth transition where it meets the ground. This problem requires a more robust solution to account for all these challenges.

The first step to segmenting obstacles in the depth image is to find the edges of obstacles in the depth image. One way to find edges is to use an edge detection filter. One such filter is called the Canny edge detector. This operator was developed in 1986 but is still a very powerful tool that can sometimes be used for real-time edge detection. The technique works by first applying the Gaussian filter, discussed in Initial Image Processing, to smooth the image of any remaining noise. Then, the x and y derivatives of the image are taken and

calculate the magnitude and orientation of edges at every pixel in the image. Next, non-maximum suppression is applied to make the edges in the image a single-pixel wide at the strong part of the edge. Finally, edges are tracked via hysteresis after a double threshold is used to identify weak, strong, and non-edges. Only the strong edges and weak edges connected to strong edges are kept.

Figure 31: Canny Filtered Image. [27]

Once edges are detected within the depth image, the objects they surround can begin to be bounded within the image. One possible way to accomplish this starts by checking each pixel (or window of pixels) in the edge image to see if it contains an edge. If the pixel is an edge and has not been visited before, a bounding box will start to be created for it. Every pixel within a window centered on the original pixel will be checked to find connected or nearby edges. Any edges found in the window will be added to a queue where new edges will be polled to find more connected edges. Once the queue is empty, the maximum and minimum values of both x and y for the connected edges will be used to define the corners of a box ((minX, minY), (minX, maxY), (maxX, maxY), (maxX, minY)) which bounds the object described by its edges. Bounding boxes will be created until every edge in the image is inside a bounding box. Some boxes may be too small to hold a meaningful obstacle. So, a threshold may be used to ensure that the area of every bounding box holds a reasonably sized obstacle.

Ranging

The goal of ranging an obstacle for reporting and tracking is to get a crude understanding of how far away the obstacles are from the camera. Given the depth image with successful segmentation, ranging will be simple because only a general estimation is desired. Conversely, one could consider creating a three-dimensional model of the obstacles; but, the robot will only see the obstacle from one point of view, so the exact depth behind where it can see is impossible. Therefore, a complex shape will not be considered for the obstacles. They will be computed as a cube or cylinder shape. Note that both of these shapes appear the same when looking at them from the side.

Within the bounding box of the obstacle, there will likely be values that do not represent the obstacle being bounded. Given that edges are unlikely to form closed loops, it is not obvious which pixels are a part of the obstacle and which are a part of the background. A simple solution to this would be creating a threshold to distinguish where the important values are in the given segment of the depth image. There is a Thresholding subsection elsewhere in the paper which can describe some different approaches to decide the best value to threshold on. The basic idea is that background values will mostly be on one side of some value and foreground values should be on the other side.

Because the robot will be operating to minimize the likelihood of running into obstacles, the lower values will be used as the foreground of the obstacle because it is most concerned with how to avoid objects. For convex obstacles (ie. rocks), it is obvious that the smallest values are in the foreground because the obstacle is closer than the ground or sky behind it. But for concave obstacles (ie. craters), it would make common sense to suggest that the larger values represent where the hole is. However, the distances looking into the hole do not represent how far away the hole is because the robot would enter the hole via the brim. This means that to best represent the distance to the hole, the distance to the ground around it is best to be used. Refer to the image below to visualize why looking into the hole will give an overestimate of the distance to a concave obstacle. Hence, the lower side of the threshold is best for both convex and concave obstacles.

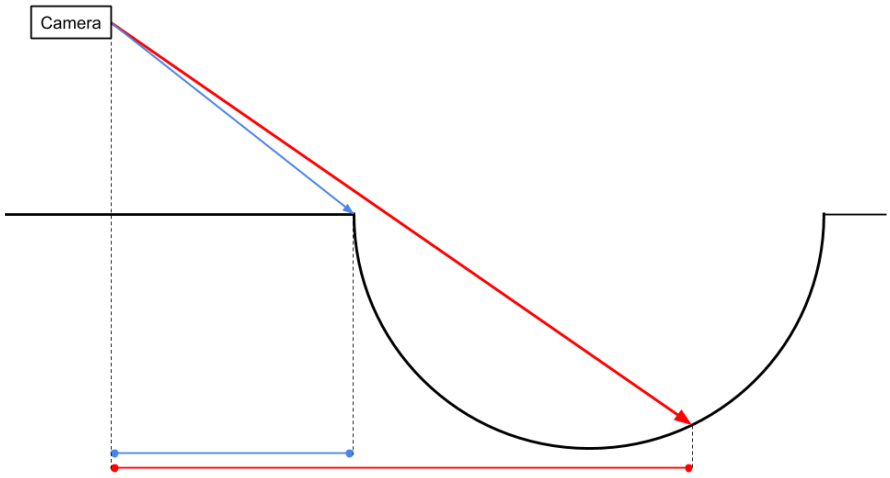


Figure 32: Distance From the Camera to the Brim Around a Hole Versus into the Hole.

As mentioned above, the obstacle representation will be a rough estimation. So, the distance of the object from the camera will have to summarize all the distances in each pixel using an average. Therefore, using the median value of below-threshold depth values will give a reasonable estimation of the distance from the camera to whatever obstacle is held within the bounding box. The median calculation provides the benefit of not allowing outliers to influence the average much.

Since the camera is above the ground, holding onto the robot at some positive, non-zero height, the distance that the robot needs to travel to reach the object is not equal to the depth calculated in the depth image captured by the camera. If it is assumed that the ground and robot are perpendicular to each other, the distance to travel is, in fact, less than the distance from the camera to the obstacle because a right triangle is formed. The distance from the camera to the object is the hypotenuse. The known height of the camera is the adjacent side to the angle between a line normal to the ground and line of sight with the obstacle. Finally, the unknown distance between the rover and the obstacle is the opposite side in the triangle. According to the Pythagorean Theorem ($a^2 + b^2 = c^2$), the distance to the obstacle can be calculated with the formula $Opposite = \sqrt{Hypotenuse^2 - Adjacent^2}$. Since the camera is not all the way on the front of the robot, the amount disparity between the position of the camera and

the front of the robot must be subtracted from the distance calculation to get the true distance from the robot.

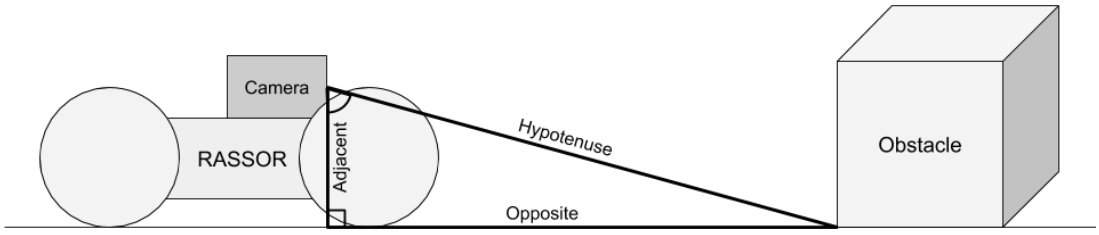


Figure 33: Right Triangle Between the Camera, Ground, and Obstacle.

Classification

To report obstacles, the size of the object will need to be calculated. Before a size can be calculated, the type of obstacle needs to be determined. This is mainly because the height of an obstacle may be either positive (convex) or negative (concave).

To make a very effective classification tool, it could be ideal to make a machine learning model and train it to classify segments of depth images for holes and rocks. The training would likely be done off-board the RASSOR to save computation power for more primary tasks, such as movement, localization, and obstacle avoidance. This training, however, would require a large swath of diverse and labeled data to learn from. Such data does not exist at the moment, but could be acquired for future implementations of the software after some navigation of the moon has occurred.

A simpler solution is more appropriate for this generation of EZ-RASSOR. Such a method could include looking at a window in the center of the image and use the thresholded values to vote for whether the center is closer or farther than its environment. This invites some obvious criticisms for accuracy and adaptability. For instance, the center of the segmented area may not actually be part of the obstacles for a myriad of reasons: such as multiple obstacles in one segment or inaccurate segmentation.

Size Estimation

Once the type of object is known, a rough estimation of the size of the obstacle can be calculated. To simplify the size-estimation process, a rectangular prism-shaped obstacle where one side is perpendicular to a line between it and the point of view of the camera. This means the object will appear as a square to the camera. Since the stereo camera will not have images surrounding the entire object, some interpolation of volume but me done through some assumptions. In some instances, properties of the object may be known that allow for a more precise and accurate estimation if the type of object is known [28].

For each side of the square seen by the camera, a triangle can be made with that side and line segments between the camera and each side. The sides created by the corners of the square and the camera can be represented by two-dimensional vectors ($a,\ b$) where one dimension is the depth of the object from the camera and the second is the x or y position values, depending on if the side of the square is vertical or horizontal. For example, the vertical side would only consider the y-position value because the angle will only exist for the difference on that axis. The angle between these vectors (θ) can be determined using the definition of a Dot Product. Hence, we can find the angle between the vectors using $\theta = cos^{-1}\left(\frac{a \cdot b}{||a||\ ||b||}\right)$.

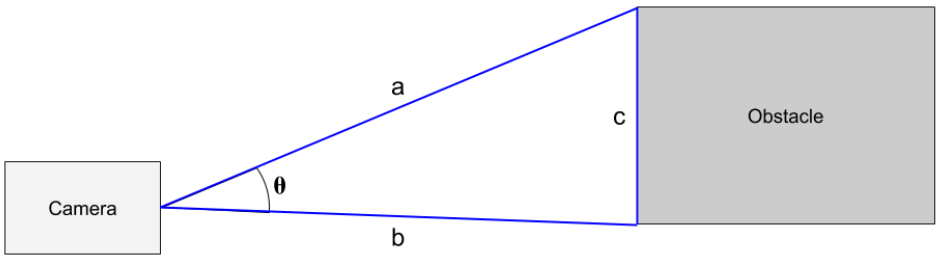


Figure 34: Side-Angle-Side Triangle Between the Camera and an Obstacle.

In order to report the size of obstacles in view of the camera, the size will need to be estimated using real-world units instead of pixels. Because different cameras may be used, objects may be at varying distances, and a depth image is supplied, the estimation will be calculated using the depth values and not pixels. For each side of the bounding box, real-world measurements for the length of the side (c) may be calculated using the triangle described above. Since one angle and the lengths of the two sides adjacent to it are known, the length of the

opposite side may be calculated using the Law of Cosines: $||c|| = \sqrt{||a|| + ||b|| - 2\,||a||\,||b||\,cos\,(\theta)}$. Once this formula is applied to one vertical side and one horizontal side, the product of the lengths is the area of the bounding box.

The final problem with this estimation is estimating the depth of the obstacle. As noted before, the camera can not see behind the face of the obstacle facing the camera. Even given the classification, properties of common obstacles do not give enough information to give a truly accurate estimation. Therefore, the depth of the rectangular prism representation of the obstacle will be arbitrarily calculated using either the average, minimum, or maximum values of the two sides that were calculated using the Law of Cosines.

Localizing

The final step before reporting an obstacle is determining the global position of the obstacle. This is a relatively easy estimation given all the work done in previous sections.

The first necessary piece of information is the distance between the obstacle and the robot. This is calculated in the Ranging subsection. Next, the angle between the orientation of the robot and the line to the obstacle is needed. In the Size Estimation subsection, angles are found between the corners of the bounding box. To find the necessary angle for localization of the obstacle, the Dot Product definition will be used with vectors going straight down the center of the field of view and the midpoint of one horizontal side of the bounding box.

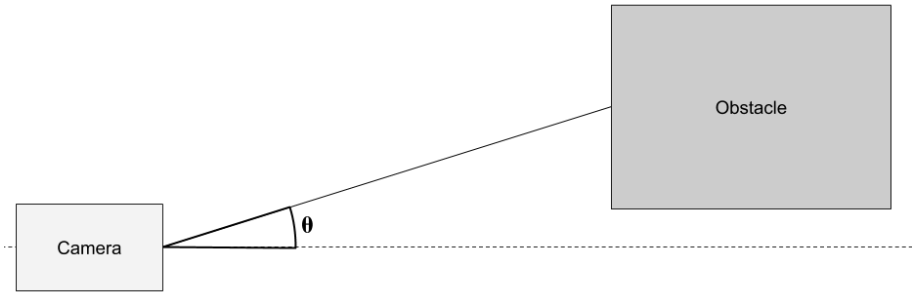


Figure 35: Angle Between Straightforward and the Obstacle.

The final bit of important information is the lunar coordinates and orientation of the rover. The distance and the difference between the robot's orientation and the angle between the center of the robot and the obstacle can be considered as a vector originating from the robot's known position. The head of said vector is the estimated location of the obstacle in lunar coordinates.

Dynamic Environment

So far, the discussion of obstacle detection and reporting has been based on the assumption of a static world. In practice, the RASSOR robots will be used in a swarm guided by a supervisor. If there is a swarm of RASSOR robots on the moon or any other place, the robots will inevitably face some obstacles which are capable of or actively moving, including each other. To avoid robot's colliding into each other, or infinitely displacing each other from their desired route, some precautions must be taken.

As briefly discussed in the path planning section, after EZ-RASSOR has decided on a path to take it will commit to moving there. As our robot is moving towards its local destination, we must check that EZ-RASSOR's local environment does not change in this time period. Some examples of reasons why this could happen are: another EZ-RASSOR is moving in front of us, an astronaut is walking in front of our robot or a rock or other obstacle has fallen in front of us. These examples are just view changes that we can encounter while traversing. Realizing this, we must implement a protocol that will account for these changes. Although this is a stretch goal for our project, we will list a possible protocol that we think could work.

Looking at the nautical rules for the road for inspiration, we have come up with a potential protocol that can work with multiple EZ-RASSOR's in the same area. Instead of having a sensor that will be rapidly scanning for changes in distance, we will simply take a snapshot of EZ-RASSOR's current view as it is moving. This will be done multiple times as we move. Similarly to boats, we were thinking that on each side of EZ-RASSOR we will have a different bright color that when seen through a camera can be easily detected. For example, let's say that in front of EZ-RASSOR is neon blue, on the right is neon green, on the left is neon orange and on the back is neon yellow. If we are to see any of these specialized colors, by simply comparing RGB values taken in our proposed snapshot, we not only can tell that we are looking at an EZ-RASSOR, but in which orientation that

it is relative to us. Knowing this, there are a few main situations that must be accounted for. If we see neon blue, green, orange, yellow or any combination of these we must know how to safely navigate. If we are seeing another EZ-RASSOR from behind, we will simply treat it as an obstacle and plan around it. If we see an EZ-RASSOR from the left, we can assume that it will be traveling towards the left and know that this area is most likely blocked for the time being. The same goes if we see an EZ-RASSOR robot from the right. If we see another robot from in front, we will simply use the protocol that each robot should proceed by taking a rightward path around each other. This works because each EZ-RASSOR robot will be seeing the same color if they are head on. Since this is a stretch goal, we will only be able to progress on this once other objectives are completed.

Localization

Localization refers to the process of determining a robot's position and bearing in some environment. There are two ways to approach this problem: relative and absolute localization. Relative localization refers to tracking your motion, rotation, etc. to determine how far and which direction you have moved relative to some starting position. Absolute localization on the other hand, uses the known positions of "landmarks" to pinpoint your location (i.e. GPS).

GPS-Denied Environment

GPS-denied refers to the ability to localize within an environment without the aid of a GPS. Because the rover should, hypothetically, be able to go to the moon, we need to be able to localize without GPS since we don't have a satellite constellation orbiting the moon. It might be possible to ping the satellites on Earth from the moon but our approach should minimize the dependencies to work specifically only on the moon for future developers to expand upon.

Relative Localization

One way to localize a robot is by determining a relative location. Because we have a starting position stored in memory, we can keep track of our movement relative to "home" and give an estimated current location. We can relatively

localize a rover by using odometry, which refers to using onboard sensors that output data related to movement over some duration of time.

Odometry

One way to determine the relative location of the robot is to use an Inertial Measurement Unit (IMU) to track the movement of the robot. This type of hardware contains an accelerometer, gyroscope, and magnetometer. The former two are the most important to the application of positioning. By using the information of both the accelerometer and gyroscope, the robot will get information from a fused sensor with six degrees of freedom.

The accelerometer will give information on how fast the robot is moving and in what direction. The given speed will be independent of wheel spin and consider the unexpected movements like falling or sliding. Using the gyroscope to know the yaw rotations of the robot along with the velocities, a fairly accurate calculation of the robot's relative position from an origin can be made in two-dimensions. Another benefit of an IMU is that it provides roll and pitch information so that when the robot is moving on uneven ground, three-dimensional displacement is still able to be calculated.

Another way to determine motion is through visual odometry. This refers to the apparent motion between time t and time t+1 that is determined by either the difference in value intensities or the displacement created by tracking "interesting" points. This approach isn't subjected to an IMU's "drift" error but is prone to the trade off of a probabilistic error that comes to matching between frames.

For more information regarding odometry, please refer to the other much larger Odometry section in the paper. The reason we mention odometry is because our localization architecture plans to use the motion estimate vector(s) that the Odometry will produce to get a combined better location estimate than if we were to use only an absolute or relative localization.

Absolute Localization

Despite knowing what location home is and a map of the terrain, it's important to still have a way to determine an absolute position. Odometry is helpful but the

error compounds the further you get away from the starting position due to the measurements consisting of mostly movement vectors. Absolute localization determines your location based on known “landmarks” you see rather than saying you travelled 34 miles north west of the start. GPS is a form of absolute localization because it determines a location on Earth by using the closest 4 satellites to a GPS receiver and triangulates its location. But because there is no GPS on the moon, we must devise a way to determine our location without it.

Absolute Localization consists of two parts: cosmic GPS and Park Ranger. cosmic GPS refers to obtaining an estimated location by identifying stars or celestial bodies. Park Ranger is used to help verify the location produced by the cosmic GPS if the ground surroundings, within the error radius of cosmic GPS estimate, can be matched to a map of the terrain.

Cosmic GPS

Function

The cosmic GPS subsystem uses an assortment of celestial bodies to determine its selenographic coordinates. From a black-box perspective, the cosmic GPS can be viewed as capturing a time-stamped full or partial image of the sky and outputting the coordinates from which the image was taken.

Background

For an autonomous rover operating in a sparsely mapped and ambiguously featured environment, such as the moon, a method for determining or estimating absolute position is necessary for the rover to successfully navigate across tens of kilometers without getting lost. This is due to the eventual large errors in position that generate after prolonged periods in navigational systems using dead reckoning without absolute references.

The standard way of determining the absolute position is to use fixed references with known position. These fixed references, also called landmarks, can be any feature that is characteristic of an area, preferably distinct, deterministic and occurring over long periods of time. Natural candidates to use as landmarks are large masses of the lunar landscape. The use of these features has the

advantage of having permanently fixed coordinates but with the downside of being difficult to distinguish. This problem is compounded by the lack of detailed global terrain maps of the lunar surface. The current global terrain map resolution is close to 300 meters with only some small subregions of the lunar surface detailed enough to create accurate three-dimensional models of landmarks the rover could be trained to identify. With this limitation and other limitations such as the amount of memory required to store landmark representations, it is probably best not to use natural features of the lunar surface for determining absolute reference on a global scale. Though for polar regions, the resolution of the digital elevation maps are detailed enough to use terrain features for absolute location.

A similar method to using natural terrain features as fixed references is to use artificial objects, possibly optical markers. These landmarks can be made unambiguously distinguishable and can be optimally placed for maximum coverage. By having these markers broadcast their coordinates (longitude, latitude, and altitude), a rover in line of sight would be able to triangulate its position. Although this would be a robust method and provide the rover very accurate position estimations, it would also necessitate placing the markers there in the first place providing new challenges and dampening the viability of this approach.

Landmarks external to the moon such as celestial bodies provide the best-fixed references. It follows a method humans have used for hundreds of years on earth, using the position of stars and constellations to determine their geocentric coordinates. With accurate measurements, a sextant can be used for estimating latitude and longitude to within a nautical mile. The currently established methods combined with the favorable environment of the moon for viewing celestial bodies, such as a thin atmosphere and no light pollution (in the case of the far side of the moon), makes celestial observation even more accurate and robust.

Data Collection

In this section, we discuss the camera models and configurations we could use to capture an image of the sky.

Multi-Camera Hemispherical Configuration

This configuration has four cameras each with a wide-angle lens fixed at 45-degree angles from one another. The orientation of each camera is further fixed at a 45-degree angle from the plane of the rover. This system of cameras would give a full combined view of the entire sky including some overlap and subhorizon views. The advantages of this configuration are in providing the possibility of capturing the images of the sky in one capture (decreasing the amount of time the rover would need to remain stationary if long exposure times are needed) with only some post-capture processing. A negative of this configuration is the potential size of the system's hardware (housing to hold four cameras) and the possible difficulty and time needed to construct it.

Single-Camera Hemispherical Configuration

This configuration utilizes a circular fisheye lens camera fixed perpendicular to the rover's plane. This configuration would simplify the design of the hardware as it would require only one camera and be permanently fixed, not needing any servos and actuators to rotate it around to view the whole sky. One of the negatives of this configuration is that, on angled surfaces, the camera would capture less of the sky than with other configurations. Another negative would be that the image distortion is greater for circular fisheye lens cameras than with the wide-angle lens cameras used in the other configurations.

Rotatable Single-Camera Configuration

This configuration has a single wide-angle lens camera fixed on a rotatable platform at a 45-degree angle. The platform would have a servo motor that rotates the platform in angular steps of 45 degrees. This configuration would have the added benefit of increasing the view of the whole sky at times the rover is on sloped surfaces when compared with the previous configuration. But would come with the limitations of adding mechanical complexity to the cosmic GPS hardware, as well as, adding an extra image processing step for stitching the images together into a full-sky view.

Stationary Single-Camera Configuration

This configuration has the camera fixed at a 45-degree angle directed in the heading of the rover. This configuration has an advantage over the previous

configuration in that the hardware complexity of rotating the camera relative to the rover is removed. But in case it is necessary to capture a full-sky view for accurate localization, the rover would need to rotate itself in 45-degree increments. The previous drawback, along with the need for an extra image processing step for stitching the images together like in the previous configuration are the negatives of this configuration.

Circular Fisheye Lens

Fisheye lens cameras allow for images to be captured with a very large angle of view of up to 180 degrees. They are commonly used for numerous scientific purposes, notably for taking measurements (photogrammetry) of solar irradiation in forest ecosystems. Due to its current scientific use, camera models and calibration methods for this type of lens are well known and developed.

Wide Angle Lens

Wide angle lens cameras allow for images to be captured with a large angle less than 180 degrees but in most cases, more than 120 degrees. They are commonly used for recreational purposes as they can capture a large range of view without much distortion. A common use significant for our project is capturing frames to be stitched into for panoramic images. Like fisheye lenses, camera models and calibration methods for wide angle lenses are well known and developed.

Initial Image Processing

Regardless of the camera setup, we need to make sure the images follow a workable format. The possible image related issues are: lens distortion, multiple images, and noise. Lens distortion can be intentional to obtain a larger viewing angle such as when a fisheye lens, but it makes working with the image less than ideal. If we have a multiple camera setup, we need to stitch the image together to maximize our field of view. And the last issue can be a non-related camera issue: noise. Noise can manifest due to poor lighting conditions, an imperfect sensor, extreme temperatures, etc.

Lens Distortion

As mentioned before, lens distortion can be intentional to obtain a certain effect for aesthetics. But there can also be unintentional distortion that occurs with human error or less than perfect equipment. The most common types of distortion are a type of optical or radial distortion. This refers to the bending or curving of expected straight lines that has to do with lens error. Other types of distortion exist, but we focus on radial distortion since it's the most common.

Types of Radial Distortion

Barrel Distortion occurs usually on wide angle lenses such as the aforementioned. The reason that this happens is because the lens captures a larger field of view than the image sensor expects, which causes it to "squeeze" as much info it can into the same resolution it's designed to output. This produces an image with the lines in the center are straight but those along the edges appear to curve away from the center [29].

Pincushion Distortion has the opposite effect of Barrel Distortion. This refers to a lens having a smaller field of view than the image sensor expects, giving it a "stretched" appearance to match the expected resolution size. This distortion can be seen on many zoom lenses. So the final image also has straight lines in the center but the outward edges appear to curve towards the center [29].

The final type of radial distortion is Mustache/Wave Distortion. This is essentially a combination of the previous two. It gives the image an appearance of the center bulging out towards the viewer and starts to flatten out towards the edges, making it difficult to fix. In photo editing applications, such as Photoshop, they don't usually have built in tools to automatically fix this. Simply applying tools to fix the Pincushion or Barrel Distortion in the image will only exaggerate the opposite type of distortion [29].

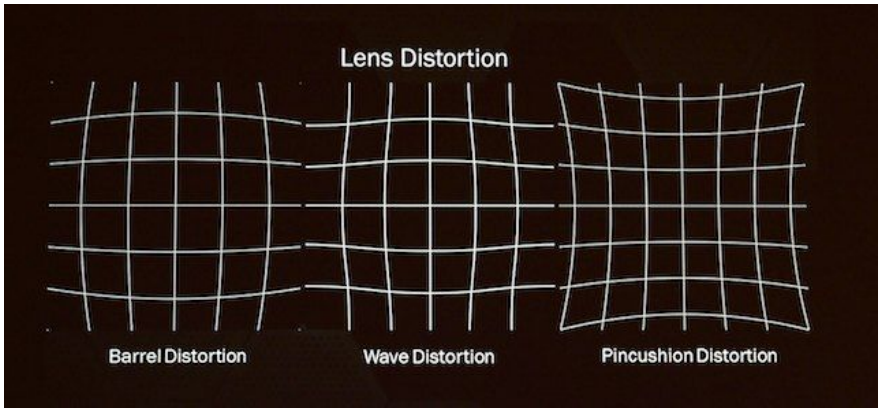


Figure 36: Shows the effects of the different distortions. [30]

Image Stitching

If we were to go with a multi-camera approach, we need to be able to stitch these views together. Although image stitching is mostly used for panoramas, this could be employed for our application as we will talk about later. But for now, we will illustrate the general overview and pipeline of image stitching. Image stitching can be broken down into two methods: direct and feature-based.

Direct

Direct techniques consist of comparing all image intensities or pixels against each other. These techniques are subjected to scale and rotation of the images but wholestically take into account the role of each part of the image [31]. Images are shifted or warped to determine how similar and where they can overlap. The main problem with these approaches is that they tend to reach higher computation times so are avoided in most applications.

Feature-Based

By stitching images along “interesting” points, this allows for the correspondence of these feature vectors to be robust against scale, rotation, lighting, translation, and noise [31]. These approaches allow for calculating the motion of the camera between captures and are less computation heavy than Direct methods. The main problem with this approach is that it's inherently dependent on unique features that can be matched across frames. Often to combat this problem, some probabilistic function that matches features while disregarding outliers is implemented.

Pipeline

According to [31], the Image Stitching pipeline can be broken down into 3 steps: Calibration, Registration, and Blending. [32] follows this structure but splits up the steps into smaller pieces in regards to panoramas specifically.

Calibration refers to helping correct the imperfect nature of a camera lens and the possibility of different camera lenses used to capture the different frames. They state that “intrinsic and extrinsic camera parameters are improved” which help with figuring out the 3D points of the scene. Intrinsic refers to associating a 3D point relative to the camera while extrinsic refers to inferring the position and direction of the camera when a frame was captured by knowing where another frame was taken.

Registration refers to figuring out how to align two images taken at different views. There are various ways to handle this but feature based registration models can be broken down into 4 steps: (1) feature detection, (2) feature matching, (3) determine the transformations needed to align images, and (4) align the images. For (1), the popular feature detection algorithms are descendants of SIFT, Scale Invariant Feature Transform, in which feature vectors describe parts of the image that are robust against scale, translation, rotation, and illumination [33]. To match these features for (2), [32] specifically employs RANSAC as way to match neighborhoods of pixels but there are other methods such as brute force, least square fit, and the Hough transform. Just like the previous step, [32] uses a specific method. Initially for (3) and (4), the 3D estimated positions of the features and the camera are calculated. Then, an alignment algorithm such as the popular bundle adjustment is used to figure out the most optimal position of the 3D points. When the images are joined, the transition between the images can be quite harsh so then blending along the seams is needed.

Blending can be broken into two different approaches: alpha “feathering” and Gaussian pyramid. Alpha “feathering” uses the weighted average of two images and is most useful in well aligned images that are different only in the shift in intensity. The Gaussian pyramid sorts out the images along a spectrum, the lower on the scale, the higher amount of blurring along the seams while not affecting the non-boundary values.

Figure 37: From left to right: 1 is the initial images, 2 is feature detection, 3 is RANSAC matching, 4 is the final image. [32]

For panoramas, a surface is needed to achieve a certain effect like in Google street view. Some of the potential surfaces to project the stitched image on are Rectilinear, Cylindrical, Spherical, and Stereographic [31]. Rectilinear has the appearance of converging two planes at a “corner”, which is often used when mapping to cube faces. Cylindrical is the popular type of panorama where the view is within a cylinder, giving a 360 degree field of view but looking right above it is distorted. Spherical isn’t subjected to the distortion that Cylindrical experiences, giving the appearance of being encased in a bubble. The final surface is Stereographic, which gives the illusion of looking down on a small planet.

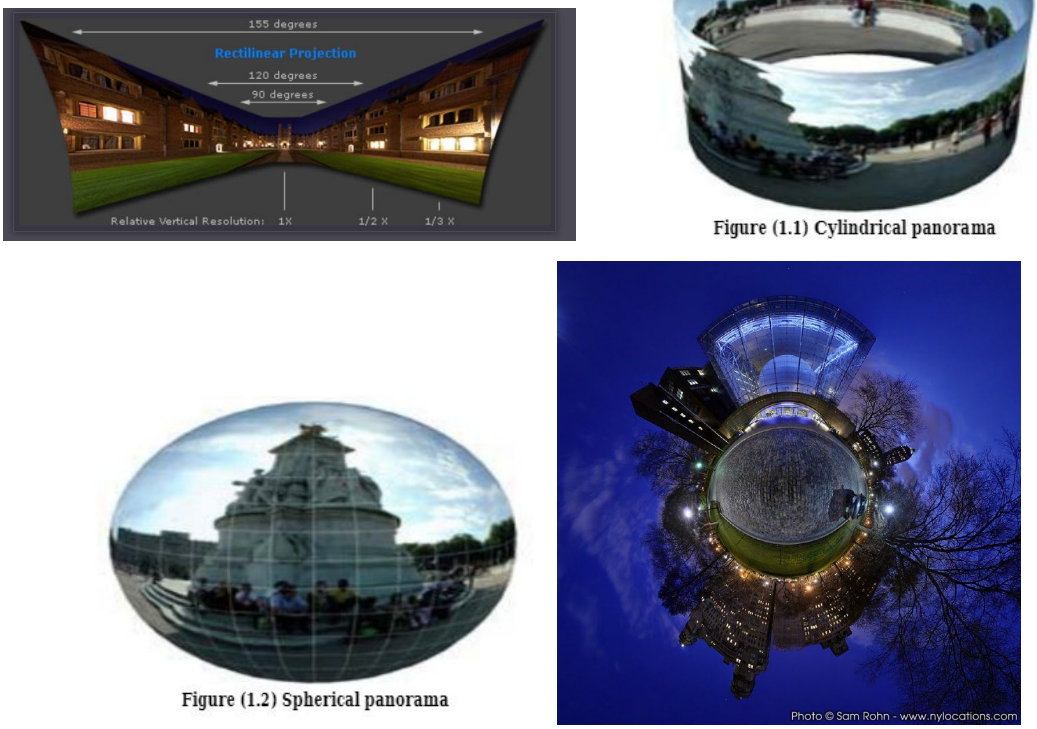


Figure 38: Clockwise starting from the top-left, the panorama types are: rectilinear [34], cylindrical [35], stereographic [36], spherical [35].

For our particular application, the Spherical surface would be best but we would only use the top half of the sphere to illustrate the stars hitting the moon surface as a sphere rather than a plane. The problem with using multiple cameras and stitching the images is that footage would have to not only be synched to capture at the same time but also there is always some possibility of misalignment or “ghost” objects. For simplicity and affordability for future hardware compatibility, we forego the multi-camera approach in favor of a singular camera.

Noise Reduction

When recording data, whether video or audio, it’s important to reduce noise that is due to factors such as movement, imperfect camera, etc. Because the data for each star may occupy a small radius, it’s important to minimize false negatives (i.e. a star is classified as a non-star). There may be false positives but with each step in the process (our approach), the possibility of them affecting the results should decrease.

Types of Noise

There are several classifications of noise in images that are caused by a variety of imperfections within the data collection, processing, and analysis processes. One common type of noise is Gaussian noise, which refers to sensor related noise caused during capture by non-ideal lighting or temperature or other such conditions. The disruptions from these conditions cause points of inconsistencies within the image where the center is strongest and the noise gradually, radially dissipates away from the center of the point. Another form of noise due to a flaw in data collection is Salt and Pepper noise, which refers to disturbances in transforming analog signals to digital, resulting in the image to be speckled in white and or black pixels or “salt and pepper”. Unlike Gaussian noise, Salt and Pepper noise causes a constant value of noise for each segment of noise.

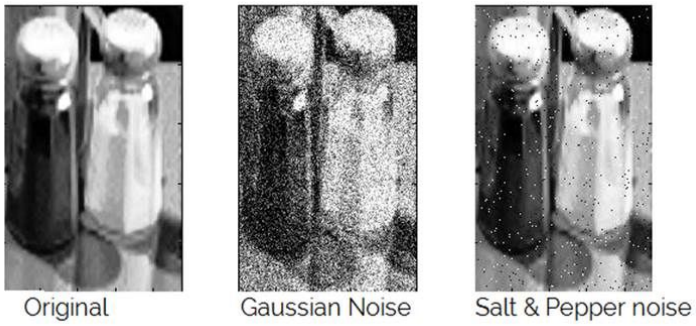


Figure 39: Gaussian and Salt & Pepper Noise. [37]

Since a camera will be used to capture images of stars in the sky, another important type of noise is Shot noise, which relates to the imaging of light sources. Shot noise refers to noise due to statistical variance in how many photons can be collected per time interval of an exposed frame in a camera when capturing electromagnetic waves such as x-rays, gamma rays, and visible light. In other words, a camera capturing video will likely record different amounts of discrete protons in each image because there are random fluctuations (in the form of the Poisson distribution) in how light moves [38].

Noise is not only caused by how data is collected. It can also be caused by poor design of processing and analysis of images. Oftentimes, computer vision researchers will apply filters to reduce the resolution of an image. They do this for

several possible reasons, including speed of analysis, resolution consistency, and noise reduction. If done for the latter purpose, it is rather ironic that such filters can actually cause a new type of noise.

Noise Reduction Algorithms

There are various algorithms to reduce noise, one way would be to apply a linear smoothing filter. This simply entails convolving the image with a mask, usually Gaussian or flat, to “smoothen” or average the pixels to be “in harmony” with its neighbors. A Gaussian mask will emphasize the original pixel and those closest to it, while a flat mask will equally weigh all of the pixels within the window. The issue with this approach is that it can fail to preserve true edges. This happens when a blur is needed to be too strong to reduce the noise appropriately and edges that used to exist become unidentifiable. Most applications of noise reduction are some form of nonlinear smoothing.

Gaussian blurring is considered a type of isotropic diffusion, which allows for blurring from sections to “bleed” over edges. A solution to this is to not allow blurring to reach over edges. This is called Anisotropic diffusion, where blurring is only allowed to permeate within its own section. Using this kind of method will make noise reduction possible without harming the ability to identify segments of the image that contain foreground images. This is the best option for reducing Gaussian and Salt-and-Pepper noise mentioned above.

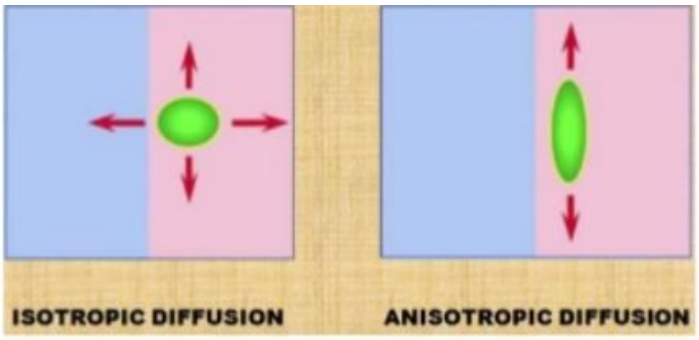


Figure 40: Isotropic vs. Anisotropic Diffusion. [39]

Figure 41: Anisotropic Diffusion vs. Gaussian Blurring. [40]

Another goal for noise reduction is sometimes sharpening the image because there is a noisy blur that disrupts edges. A common algorithm to sharpen in an image is the Laplacian of Gaussian (LoG) operator. To start, the image needs to be blurred using a Gaussian mask. This may seem counterintuitive to sharpening, but it is necessary to reduce the other types of noise because edges will be found in the next step. Then, the second derivative of the Gaussian-blurred image is calculated. In this derivative image, a constant intensity will be zero, a consistent change in intensity will be a positive or negative value, and crossing zero signals an edge. Edges exist at a change in sign because that means the intensities were increasing or decreasing and then flipped. When the direction of change flips, there exists a local maximum or minimum of intensity. The final result of the LoG operator is the edge image. Finally, a sharpened image can be made by adding the original image to the LoG edge image [41].

Figure 42: Laplacian of Gaussian Sharpened Image. [42]

For “reducing” noise by improving the signal to noise ratio the technique used will affect certain types of noise more than others. For this reason we will state the different types of noise and ways to improve this signal to noise ratio.

For shot noise (the incident of random photons hitting the sensor) this noise can be corrected in mainly two different ways. The first method we will mention for reducing the impact of shot noise is to increase the exposure of the camera sensor.This is because the signal increases more quickly than shot noise thus giving a stronger signal to noise. A caveat to remember though is that if the sensor is exposed for too long the individual cells of the sensor will become saturated truncating the signal measurement. The second method mentioned for reducing the significance of shot noise is to stack frames. In many ways stacking frames is equivalent to increasing the exposure. In stack framing although for each frame the signal of each star is indistinguishable from noise, because the noise is randomly distributed across a frame and over many frames, while the signal from stars are fixed to particular cells, when the intensities of multiple of these frames are added together the signal from the stars become distinguishable.

For dark current noise, a randomly distributed noise, stack framing can be used as the noise behaves similarly to shot noise. Another technique commonly used involves subtracting the baseline signal (also called a dark frame) generated by

the dark current. This technique doesn't actually reduce the noise itself, only the bias. Finally on the topic of dark current noise, due to the temperature on the moon, absent from the sun, dark noise should be less of a factor than the other sources of noise as it is temperature dependent and becomes negligible at very low temperatures.

For read noise, caused by the variations in electrons randomly added or lost in the circuit when transferring data from the CCD to the chip, this noise is random though fixed and dependent on the electronics. In cases when the read noise is greater than the signal measured The best technique is to increase the exposure. Although stack framing can also be used the need for potentially many frames and computational cost of this make increasing exposure a better option.

For fixed pattern noises such as dark signal non-uniformity and photo response non-uniformity noise, these sources of noise can not be removed only the bias toward the patterns can be subtracted. In the case of dark signal non-uniformity a dark frame can be used to retrieve a bias frame much like with the dark random current above that is subtracted from the measurements. In the case of photo response non-uniformity, the camera can be calibrated for different exposures to measure the sensitivity of each pixel. The differences in the sensitivities can be caught during the calibration and used to construct a special frame to correct it.

Now having discussed the different noises and the techniques to reduce or decrease their significance, we will discuss the contrasting qualities of increasing exposure and stack framing. As previously indicated, they are roughly equivalent and interchangeable. For increasing exposure time, as already mentioned, the primary negative of using this technique is the limit on the signal that can be acquired. A positive of exposure time is that the improved signal to noise ratio comes free in that it is improved without image processing. The pros and cons for frame stacking is the exact opposite. With frame stacking, the primary advantage is the lack of limit on the strength of the signal measured, while the primary negative is in having to combine these frames which involves image processing. For these reasons neither one is better than the other, they can be used together in a complementary fashion to achieve results better than if only one of them was used.

Azimuthal Projection

An azimuthal projection uses a curved object such as the Earth surface and projects it onto a flat plane [43]. This is important in this application because when an image of the sky is captured, the Earth's surface is treated as a plane, which misrepresents how the stars hit the surface of Earth or any celestial body you're on. The most common types of Azimuthal Projection are Gnomic, stereographic, and orthographic projection.

Common Types

An orthographic projection map is often what we usually see if it's a map of the world so it looks the most natural to us. The projection point is placed at infinity and would only appear like this if you were thousands of miles away. This causes a distortion in the image at the edges and the shapes deviate from their actual appearance.

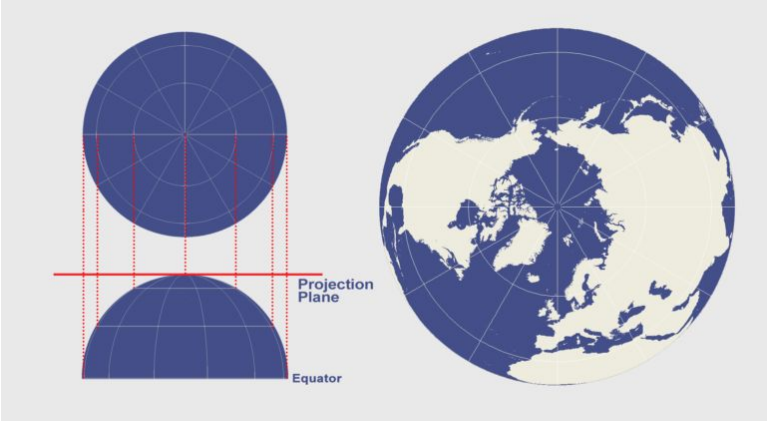


Figure 43: Orthographic projection. [43]

A stereographic projection places the light source at the opposite end of the object, creating an appearance of everything being tugged at that point as well. This type of map is used commonly for navigation maps because of how it conserves shape, despite the "stretched" scale of objects.

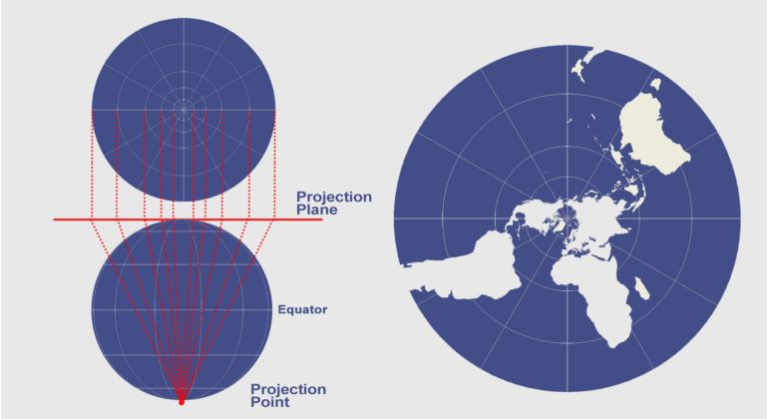


Figure 44: Stereographic projection. [43]

A gnomonic projection places the light source at the center of the Earth or body, which doesn't appear as stretched as stereographic, but contains less than a hemisphere of information. These are often easiest for planning the shortest route between points but don't preserve shape or scale the distance accordingly.

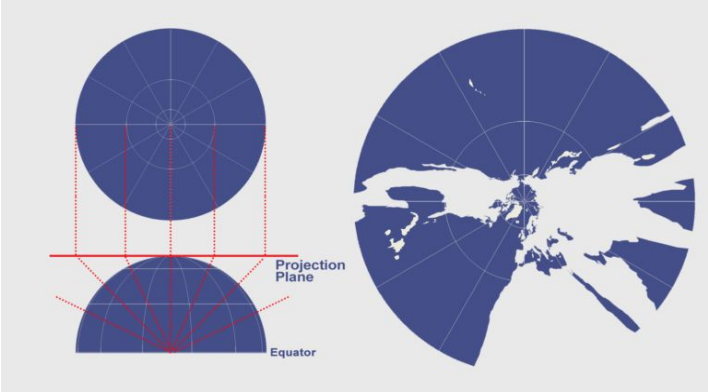


Figure 45: Gnomonic projection. [43]

Image Segmentation

In order to use stars to calculate the position of the robot, identifying where there are stars in the images captured by the camera is necessary. The first step to tracking stars is separating where, in the image, there are likely to be stars. This is accomplished by finding a threshold to set a binary value for every pixel, whether it's a star or background. Second, stars will need to be classified/identified so that sight reduction can be performed.

Thresholding

To classify pixels, they can be turned “on” or “off” based on their value relative to a defined threshold. The purpose of creating such a binary image is to segment where foreground objects are over the background. This application will look to segment stars in images of the sky so that analysis can be done to determine the absolute position of the camera on the moon. Typically in thresholding, the foreground is shown in white and the background is displayed as black. Some popular types of thresholding are histogram shape, entropy, and local.

A histogram is a graph comparing the number of pixels at each level of intensity (typically for grayscale images). Thresholding based on the shape of a histogram means that there is a point on the x-axis where clusters of intensities can be differentiated and classified. A common way to accomplish this is to set a single threshold at the local minimum that is closest to the median value of the intensities [44]. The point of this is to create a binary distinction between a group of high-intensity pixels and low-intensity pixels.

Entropy is the measure of disorder/uncertainty of a system. The value of entropy in an image increases if its values are well distributed across the entire range, and low if values are well compartmentalized [45]. Entropy can be calculated using the relationship between the histogram of an image and the probability density function. The probability distribution function describes the probability that any given pixel will have a certain intensity, given by $p_i = \frac{n_i}{N}$ where n_i is the number of pixels with a given intensity and N is the total number of pixels. The total value of entropy is calculated by $H = \Sigma^{N}_{i=1} \left[p_i \times log\left(p_i\right) \right]$. Since the value of entropy and the intensities in the image follow the same range (ie. 0 to 1), the value of entropy can be used to threshold the image. Pixels above the value of entropy will be white and should be the stars in this application.

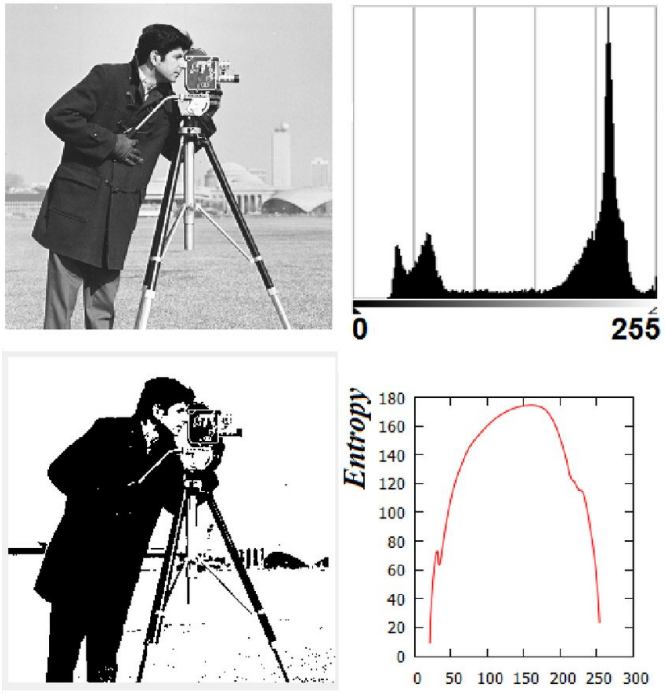


Figure 46: Entropy Thresholding of an Image. [46]

Both of the previous thresholding techniques use global data of the image to create a single threshold. In contrast, local thresholding decides a new threshold for each pixel, considering the characteristics of itself and adjacent pixels. The way the threshold is calculated can consider factors such as mean, standard deviation, maximum, and minimum intensity values of pixels within a window centered on the pixel in question [47]. Therefore, local thresholding can be a more computationally intensive technique but has the benefit of allowing different filters to be applied throughout an image and not affect the threshold of the entire image.

A good local thresholding technique for night sky images is automatic adaptive local thresholding. It involves comparing arbitrary pixel values with a “special value” (the threshold) to determine if a pixel is “significant” or part of the background. The first part of this technique involves finding the average intensity, W(x,y), for an S x S window centered on the arbitrary pixel (x, y) in question. Then the ratio of the pixel intensity to average is determined

$$R(x,y) = I(x,y) / W(x,y),$$

where I(x,y) is the intensity of the pixel centered in the S x S window. Then, after the ratios for all of the pixels have been calculated, the final step is to take the max and min ratio values, Rmin and Rmax, and calculate the threshold T via

$$T = Rmax + Rmin / 2.$$

This T is then used and compared to each I(x,y). In the case T is less than I(x,y), it is assigned 1 else it is assigned 0. One caveat of this method is the S x S window impacts the results to a significant degree. Therefore, based on the camera used and the ultimate size of the stars in terms of pixels, it is best to tweak this window to determine its best size empirically [48].

Once each pixel is categorized as the sky (black) or starlight (white), the stars will be identified and used in calculations to define the position of the robot taking the image. To find an entire star, every white pixel will need to be grouped together with its directly touching neighboring white pixels forming what can be thought of as islands which correspond to stars. A method for doing this clustering follows as such. Starting with a two dimensional array. Of light pixels and black pixels and an array with the coordinates of each light pixel. Take an arbitrary pixel and place it into a new cluster, then check for its neighboring pixels to see if they are light or black. For those neighboring pixels that are white, place them into the cluster and further check their neighbors. Do this until none of these neighbors have a light pixel or an unvisited one. Then, pick a new pixel from the array and start a new cluster. Continue this until all of the pixels in the array have been accounted for. Finally, we will have a number of clusters, composed of a contiguous blob of pixels, corresponding to individual stars [48].

The final step before we can take measurements involves finding the center of these stars. Although this step may seem unimportant, it greatly impacts the accuracy of later calculations. The algorithms used for finding these centers are called centroiding algorithms. Below describes a centroiding algorithm by M. V. Arbabmir et al. In an arbitrary cluster first the marginal distribution of the point spread function is found via,

$$Ix = SUM\ L,\ y=1\ \{\ I(x,y)\ \},$$
$$Jy = SUM\ L,\ x=1\ \{\ I(x,y)\ \},$$

where Ix is the sum of the pixel intensities for an arbitrary row of the cluster and Jx the sum of pixel intensities for an arbitrary column of the cluster. As an example where L= 3, we are finding I_1, I_2, I_3, J_1, J_2, and J_3. Then from these the mean intensity of the rows and columns are found via,

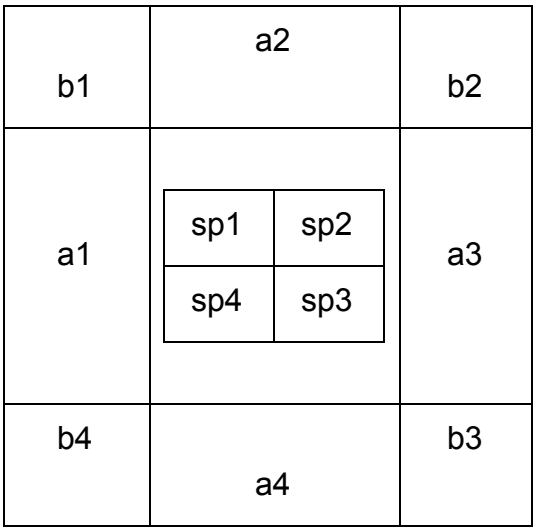
$$I_{avg} = (1 / 2L) \text{ SUM } L, x=1 \{ Ix \},$$
$$J_{avg} = (1 / 2L) \text{ SUM } L, y=1 \{ Jy \}.$$

With these averages, the IWC is computed via the following,

$$x_c = \text{SUM } L, x=1 \{ (Ix - I_{avg})x \} / \text{SUM } L, x=1 \{ (Ix - I_{avg}) \}, Ix - I_{avg} > 0,$$
$$y_c = \text{SUM } L, y=1 \{ (Jy - J_{avg})y \} / \text{SUM } L, y=1 \{ (Jy - J_{avg}) \}, Jy - J_{avg} > 0,$$

where x_c and y_c are the initial starting values for the next step. The previous step is useful compared to choosing the pixel with the highest intensity, as the most intense pixel may be due to noise. Using this initial centroid, we calculate the subpixels of the centroid pixel O_i, using the neighboring values. As follows:

$$sp1 = a1 + a2 + 2Oi / 4,$$
$$sp2 = a2 + a3 + 2Oi / 4.$$
$$sp3 = a3 + a4 + 2Oi / 4,$$
$$sp4 = a4 + a1 + 2Oi / 4,$$



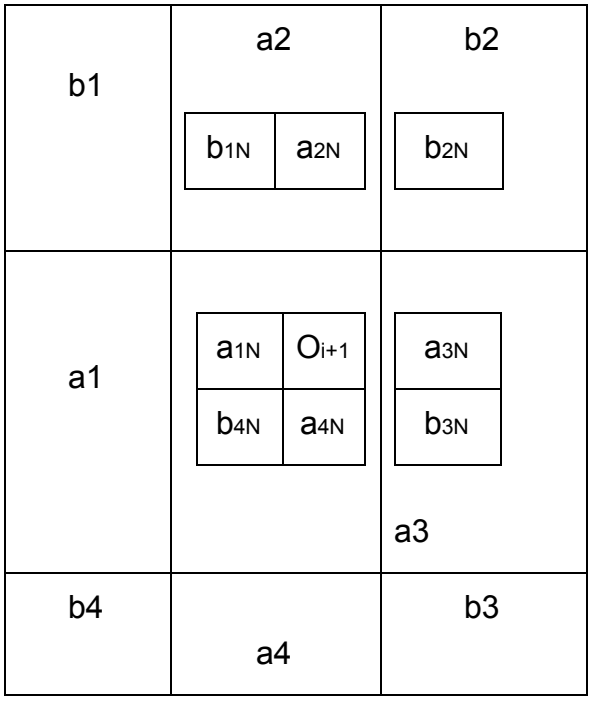
Using these subpixels with the highest intensity we update the centroid values x_c and y_c.

$$x_c = x_c +- ad,$$
$$y_c = y_c +- ad,$$
$$ad = 2 ^ (-i + 1).$$

Then to set up the new iteration to find the subpixel use the following formulas,

$$a1N = sp1,$$
$$a2N = b2N + Oi + 2a2 / 4,$$
$$a3N = b2N + Oi + 2a3 / 4,$$
$$a4N = sp3,$$
$$b1N = O_i + b1 + 2a2 / 4,$$
$$b2N = a2 + a3 + 2b2 / 4,$$
$$b3N = b3 + O_i + 2a3 / 4,$$

b4N = sp4,



From here the subpixels of the centroid O_{i+1} is found following the above steps. And, after a number of iterations (say 5) the procedure is complete and the coordinate of the star is ready for use in later measurements [48].

Celestial Body Classifier

The common approach to identify objects of interest in an image is usually to use a neural network and increase the complexity depending on how hard a problem is. For stars and celestial bodies, because they stand out against the blackness of space, it's much easier to extract the shapes than in a typical vision identification problem. A unique problem, however, is if you were to attempt to identify one specific star, it's nearly impossible to determine without using its neighboring stars. This is because a single star will be indistinguishable from any other distant light source. But since the stars are so far away, we at least know the same stars seen on Earth would be the same on the moon. We do have to account for how what we can see will be different on the moon compared to here on Earth. The moon sky will not only be affected by the lack of an atmosphere but the Earth will become almost a fixed pseudo "moon" on the side facing the Earth [49]. This allows for the Earth to be a constant source of light except during the waxing and waning when it darkens during a "new Earth" phase, which would be an observed full moon on Earth.

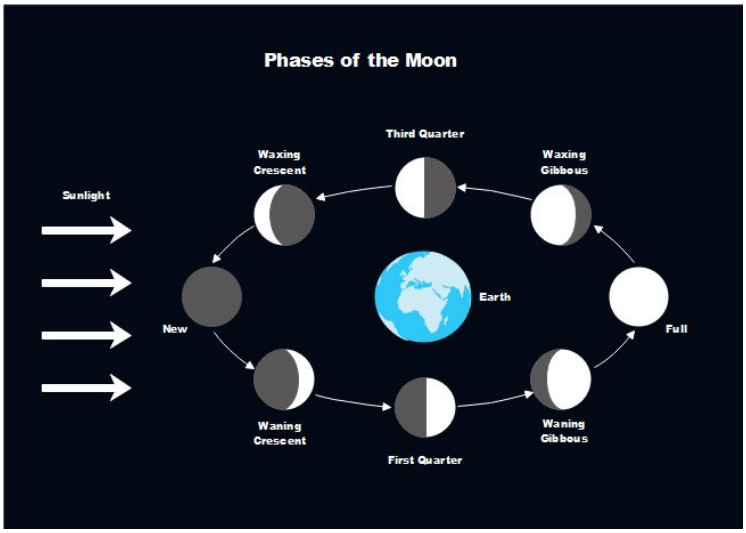


Figure 47: Phases of the moon. [50]

Lunar “Day”

During a lunar day, it’s possible that we would see the Earth, the Sun and stars. However, due to the light intensity of the Earth and the Sun, it’s possible that the stars can’t be captured during the day even with a long exposure. So the things we could use to localize during the day are the Earth, the Sun, and what we can see directly in front of us such as craters, rocks, other rovers, sparse man-made “landmarks”, etc.

Sun Navigation

The most prominent star for our solar system is the Sun, which allows for easier recognition than some stars at night amongst the hundreds that can be seen from the Earth’s surface. [51] tests sun sensors for navigating for rovers, the area relevant to this project is their test for determining direction and position from the Sun and gravity. According to their results, it gives a rough estimate within 200 km, which makes it less than ideal to use independently to determine the pose of the rover. [52] analyzes the use of a sun compass to help reduce the error from the localization estimates from the odometry. Although the Sun can certainly help localize, it would be ideal to have more sights to help triangulate a location. The Sun would also be useless if the rover was on a portion of the moon currently experiencing lunar night.

Earth Detection

Although there is a chance that Earth can be seen during the day or night, the Earth and the Sun are likely the only celestial bodies that can be seen for the lunar day through a camera. This is because the Sun and Earth light would overpower the camera sensor, obscuring the stars that could otherwise be seen with the naked eye. A simple method we could use to find the Earth is to employ a sort of edge detection and identify the earth by the colors blue and green. It's unlikely we would see a blue and green object in the sky that wasn't Earth but it's not as usable if this software was modified for Mars or on other non gaseous planets/celestial bodies. Another approach could be a recognition software for whatever expected celestial body. But, this would likely involve pre-training a neural network on images of the celestial body, whose actual view in real world application may vary enough from training data to incur a high enough error rate to not be reliable enough for navigation.

Park Ranger

Because we can't see the stars during the day through a camera, another way to use terrain based landmarks to localize the rover. This can also be utilized in the dark, provided that lights are supplemented to illuminate the scene captured by the camera. Park Ranger works off the idea of having an overhead image of the terrain that the current surroundings can be compared to. The specific details of how it works are described in the respective section.

Lunar "Night"

When the Sun is not in view on the moon, it can be categorized as night with or without the Earth looming overhead, depending on which side, just as the moon during the night on Earth. This allows for the stars and Earth to be used to localize along with Park Ranger if the rover is fitted with lights to help the camera capture the terrain.

Star Recognition Algorithms

Junkins and his team proposed an early star recognition algorithm that consisted of forming star "triplets" to determine the angles between the stars to identify them [53]. The main problem with this approach was that it tried every possible combination of three stars. It wasn't until Liebe's suggestion of using the two

closest stars to a reference star as a pattern and the angle they formed that the time complexity dropped [53]. This accounted for predicted stars too dim to appear and for when the two closest stars were at a similar distance. Approaches after this either focused on organizing the database of reference stars or extracting stars efficiently. Quine illustrates the former by placing database stars into binary trees depending upon the angles between the stars. Padget demonstrates the latter by using a grid to rotate stars around a reference star until aligned to the origin [53]. Then the cells that contain stars are used as features to match to the database, which reduces star extraction complexity. [43] builds off of this idea and the idea to store the database into a binary tree. It extends Padget's grid approach but does this to the database images to extract the stars by using reference stars as the parent nodes and the children nodes being neighboring stars sorted by distance. The main issue overall with these approaches is that star identification is dependent on its relationship to other stars so why not extract stars from constellations.

Constellation Detection

[54] is a recent paper from Stanford that covers star identification, but rather than matching a specific star, it detects constellations. The way they create the database is slightly different than the previous one mentioned. In their initial image processing, they use different colors to differentiate between the stars and their connecting edges. They also filter unrelated non-star data and then start to record the template. To organize the data further, they use the brightest star in the field of view as a reference star set at (0,0) and the second brightest star set at (1,0). The distance between these stars is then normalized, creating the scale for the template as the distance between the two brightest stars. A constellation entry consists of the previously mentioned stars but also the third brightest star and defines an area formula that describes the visibility magnitude of these stars, which allows the template to be able to match with queries of varying scales and brightness. For optimization, the distance between constellations themselves isn't important which means they can simply keep track of the 5 nearest constellation neighbors to the current constellation. For a query image, first they rank stars based on brightness and then iterate through the database based on how many stars are in a particular constellation. By picking the brightest two stars in the query constellation, the scale can be calculated, which can be used to check the templates. If the stars in the query constellation match some part of the template, then the number of matching stars determines if it's a possible

match or not. The process is then repeated on other potential constellations in the query image. After all the possible constellations in the query image are decided, then mismatched constellations are removed. In the case there are overlapping edges between the constellations, the conflict resolution states that the first constellation with the most detected stars is considered the correct version. After removing those edges, the neighbor constellations are then checked for overlap as well.

After reviewing these approaches to star detection, approaches that aim to identify a specific star are still dependent on the nearby stars to create a pseudo constellation. Although the constellation detection algorithm doesn't identify specific stars, it could be used to extract stars to take many nautical based measurements. This approach would take longer than identifying one specific star but many stars could allow for a more accurate triangulated location. Depending how often the robot localizes it, one approach would be more optimal than the other. If the time between estimated coordinates is short, the single star identification would be less computationally complex. But if the interval in between captured frames of the sky is relatively long, then shelling out the time for constellation extraction to provide accuracy would prove invaluable.

A method developed for identifying stars during this semester is described below. After finding the center of the stars and their relative angles from each other as well as their angular distances from the robot, we can use the law of cosines on a sphere to find the distances between each of the pairs of stars (triangulation).

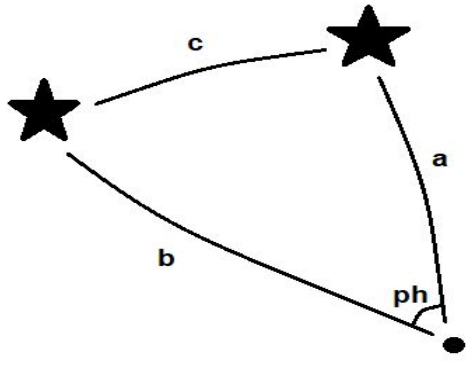


Figure 48: Law of Cosines on a sphere.

With the ability to determine the distances between the stars we see, our method can determine the identities of these stars. We do so by taking the 500 or so stars with the largest apparent magnitude from a standard star catalogue [55] like Hippocus. And we calculate the distances for each pair of these stars (those pairs of stars with an angular distance less than 120 degrees), which we can represent as a graph where the vertices are stars (with known identities) and the edges are the distances between a pair of stars.

Having this graph, we can use it as a reference to match with what we see. We start our procedure by using the stars with the brightest intensity as they are most likely to be seen and found in the reference graph we precomputed. With the brightest star and the second brightest stars measured, we use the equation above to calculate the distance between them. Based on this distance, we add marks to the counting arrays of the brightest and the second brightest stars, where each element corresponds to a star in the reference graph. The stars marked indicate a possible identity based on a distance observed. Then, for the third brightest star, we compare it to the first and second brightest stars like we compared the brightest and the second brightest. Further we do this for fourth using the first, second, and third, etc. Once the nth star iteration has been completed, we grab the element in each array with the largest number of counts as these elements correspond to the respective identities. Below shows an example:

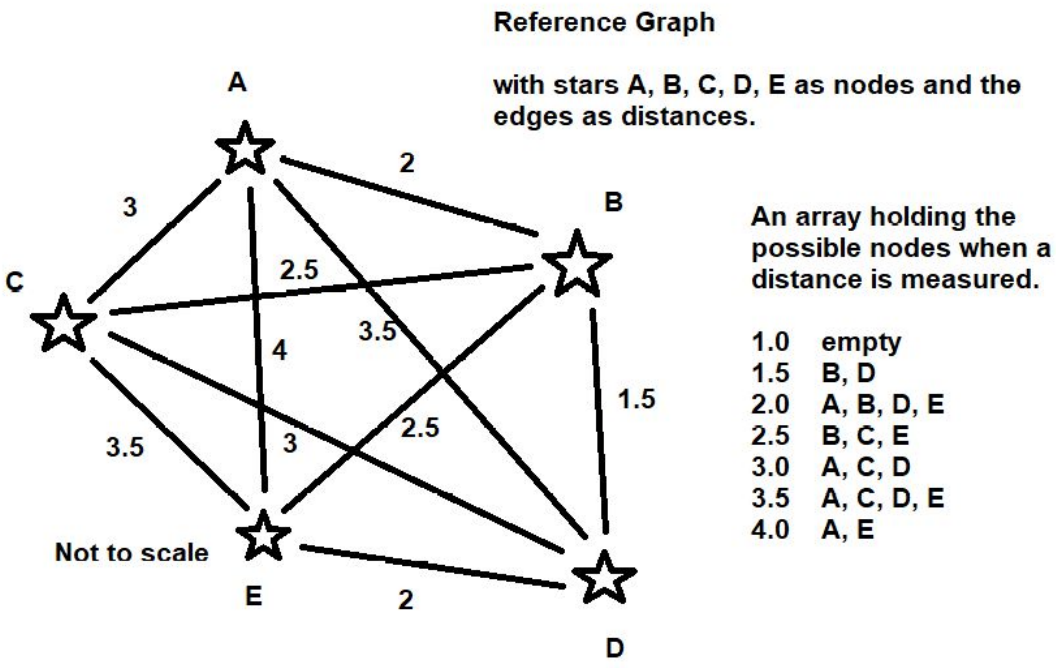


Figure 49: Shows the reference graph used by the example below.

Example: Using the reference graph above, lets say we measured three stars. The brightest star denoted 1st, the second brightest star 2nd and the third denoted 3rd. The calculated distance between the 1st and 2nd is 2.5. The distance between 1st and 3rd is 1.5. The distance between the 2nd and 3rd is 3.

Based on these measurements we prepare the counting array for each star and the fill in the counting arrays based on the measurements:

	A	B	C	D	E
1st:		II	I	I	I
2nd:	I	I	II	I	I
3rd:	I	I	I	II	

Results: the 1st star is star B, the 2nd star is star C. the 3rd star is star D.

Figure 50: Shows an example of how the proposed method can identify stars based on the reference graph.

A potential problem associated with this method is in trying to identify stars that are not in the reference graph, which would lead to star misidentification. A potential method of solving this, is by choosing the brightest stars as they are more likely to be present in the reference graph. This is why starting with the brightest star and then choosing in order of decreasing intensity is done. Also, when for example the nth star is located in the reference graph, then the stars brighter than it are in the reference graph. The resulting counting array will have only one element with n-1 marks unless there are two or more stars with identical connections in which case there are multiple elements with n - 1 marks. In this approach, we can correctly determine the correct candidate or possible candidates. When an nth star is encountered that has less than n - 1 marks in its counting array, we stop the process as this star was not in the reference graph. In the case there are multiple candidates, we traverse the reference graph to determine which is correct.

Capturing Stars

Regardless of the star identification algorithm, capturing an image of the starry night isn't so simple. Even though stars can be seen with the naked eye, because they're so far away a camera's shutter needs to remain open long enough for star light to hit the sensor. Most shutters used in various applications have rolling shutter speeds or the hardware was made in mind to limit on how long you're allowed to keep the shutter open. Since most affordable cameras for robotics purposes don't have much of a manual shutter option, it was a question of if this can be accounted for on the software side. Stumbling across [56], Star Stacker takes multiple pictures and "stacks" them together to create an image of a starry night. Unsure of what kind of camera took the raw photos and if some of the "stars" are noise, this approach is a possibility for our application rather than a solution. If it doesn't work, we could always suppose we have a good enough camera that either allows for manual shutter time or is specifically designed to handle star light conditions.

Because we could see both stars and the earth in one frame, it could be beneficial to use all the info we can. A simple way we could do this is by checking if we find that some object that is blue and green that we say is "earth", but it restricts the application to be on the moon or within range of the earth. Most approaches that seek to solve this problem work off the idea of two different sensors for stars and celestial bodies. One approach [57], referenced by [58],

checks for stars and celestial bodies by alternating between the two by fiddling with the exposure. [58] states that this method could result in false matching and inaccurate results. [58]’s method aimed to extract both the centroids of stars and the edges of celestial bodies at the same time to determine the attitude (orientation) of a spacecraft and the line of sight vector that it makes with the celestial body in view. As stated in their approach, thresholding the image is a bit more difficult because the celestial body can appear much brighter than some stars. They combatted this by iterating over the image with a double window of variance that is used to determine if something is a star, edge of celestial body, or something else.

Geometric Image Transformation

With the image of the sky captured and the positions of the stars determined to a sub pixel level. The values of these pixels are transformed into spherical coordinates as determined by the distortion of the lens determined during calibration. With the lens distortion corrected, by using the measured tilt of the robot by an inclinometer or IMU measurement we can determine the zenith of the robot. Further, the distance of all of the significant stars relative to this zenith can be determined as well as the angles of these stars relative to the robot's bearing. Giving the following:

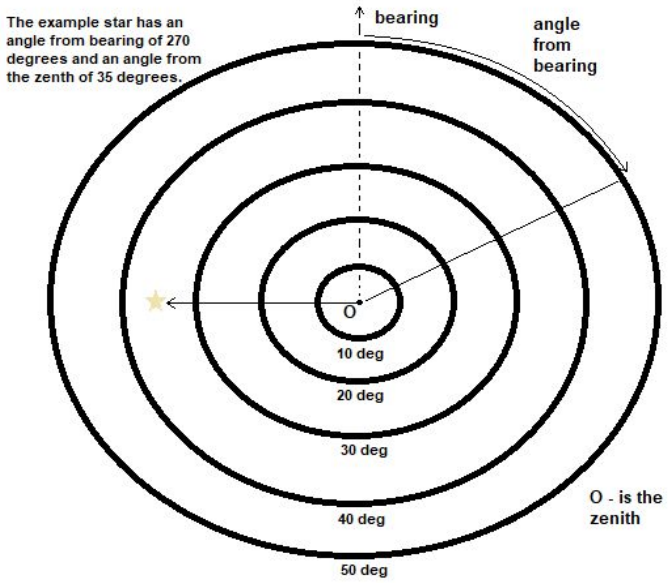


Figure 51: Note, this image will be represented using a data structure describing this map. Possibly, a sorted array holding a reference to each star instance, where each star instance holds detailed data about it such as its angle to the robot's bearing, its distance from the robot, and so on.

Sight Reduction (Nautical Navigation)

Prior to the invention of the GPS (Global Positioning System), people had to navigate with often a known map of the roads or terrain. Even with a map, the environment can start to look pretty similar after a while like when sailing in the ocean. Some people can't imagine living without a GPS but it's actually not that much different from navigating by the stars.

GPS

The Global Positioning System is much older than you'd think. Invented by the US Naval Research Laboratory, it was to be the successor to the first satellite-based navigation system called Transit. Developed by the Applied Physics Lab at John Hopkins University and fully launched in 1968, Transit consisted of 36 satellites orbiting the earth to provide accurate navigation for the military and commercial ships [59]. The main concerns with Transit were that a location was available after an interval of a few hours, the coordinates could only calculate two dimensions (latitude, longitude, but not altitude), and could not track fast-moving ships. GPS wasn't initially planned to ever be given to the

public until a commercial airline was shot down after drifting off course into restricted airspace and President Reagan declared access to be an issue of safety, it would still be some time until this policy was enacted [59]. Transit wasn't fully replaced until GPS was fully functional in 1995(6), which was also around the time that the public was allowed to also use the system albeit at poor accuracy. In 2000, President Bill Clinton made the high accuracy available for people regardless of nationality across the globe to use [59]. In terms of war or countries under scrutiny, a GPS blocking device can be released into the area when deemed necessary.

Originally called Navstar GPS, the system consists of three main components: the satellites, control stations to monitor them, and receivers in the devices requesting a location. Each satellite in the constellation (group) is constantly broadcasting where it is and the satellite's atomic clock time at the speed of light, receivers calculating the distance from the satellite (distance traveled = velocity (speed of light) * time travelled (time received - time sent)) [59]. This means that their clocks must be synchronized. Although the difference can be counted in microseconds, it can cause a difference of several kilometers. There are a few additional things that have to be considered. First, the satellites are subjected to less gravity, allowing them to move much faster than the rotation of the earth and thus the surface time appears to be "faster" than the satellite. Second, the receivers are not equipped with atomic clocks due to their costly nature and size, which also causes a difference in time [59]. With this in mind, there are at least four satellites in the range of every position on earth at all times. The first three satellites are used to triangulate the location of the receiver, and the fourth satellite is designed to synchronize the time differences. The control stations consist of receivers to ensure that a satellite is where it should be by predicting the orbital path of the satellite given conditions such as gasses at high altitudes. The predictions sometimes deviate from the actual path so the updated predictions are sent to the satellite to correct it along with clock synchronization.

The main reason that this system can be compared to nautical navigation is because of how it can be used to trilaterate a receiver's location. Not to be confused with triangulation, trilateration describes the method of using the intersections of spheres to determine a shared point on the circumferences. This allows for obtaining three-dimensional coordinates rather than two-dimensional coordinates. A basic nautical navigation technique triangulates a location based

on the celestial bodies that can be observed from that location. The satellites then become comparable to close proximity stars that we can communicate with and manipulate. In theory, trilateration with stars or celestial bodies seems possible but impractical since it's difficult to confirm in real time that a non man-made satellite is at where it's predicted to be.

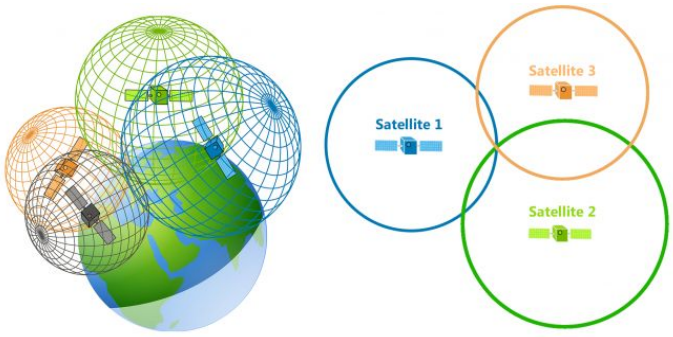


Figure 52: Trilateration vs Triangulation. [60]

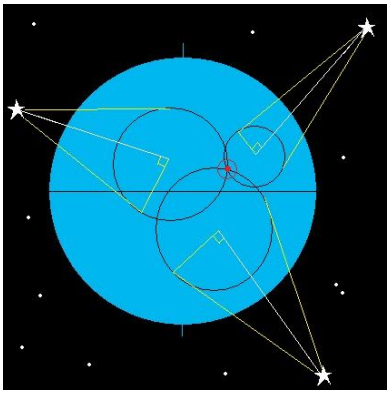


Figure 53: Basic Nautical Navigation. [61]

Nautical Navigation

Prior to the advent of publically available real time location tracking, sailors determined where they could be using a sextant, nautical almanac, clock, and a map. Even with a GPS, often sailors and even naval officers are taught nautical or celestial navigation in the case that the onboard GPS fails for whatever the reason. By night or day with decent visibility of the stars, moon or sun, an estimated location can be derived by using trigonometry or by pairing a sextant with a nautical almanac and clock, it can be equated to simple geometry equations.

Basic Nautical Navigation

A sextant can be equated to a giant compass to measure the angle that the celestial body makes with the apparent horizon. This measurement allows you to determine how far you are from the GP of that celestial body by calculating the product of the complement of this angle and the distance that 1 degree is equated to. The GP (ground point) refers to where a celestial body is directly above a point on Earth, which can be looked up in the almanac, given that you know the date and time. Knowing how far you are from some GP only gives you so much information. It's essentially drawing a circle on your map with the radius being the distance you found, but you could be on any point on the line the circle makes. A simple way to refine this is to take different measurements of different celestial bodies and find the intersection of these circles to find a more educated guess of where you could be. The problem with this approach is that you can be off by 20-30 nautical miles and up, which makes this less than ideal for navigation.

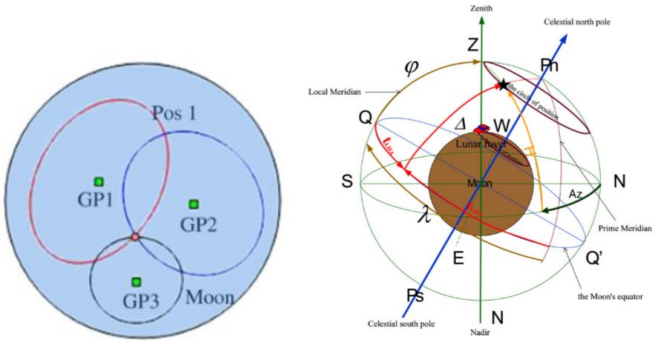


Figure 54: Left is Basic Nautical Navigation and right is Intercept Method [62]

Intercept Method

Another way to calculate where you are is called the intercept method. Although [62] devises a new way of celestial navigation for rovers, it does a good job of explaining this method with regards to the moon. The method starts by picking a point near where you think you are. First, the Local Hour Angle is calculated, where AP is your assumed position, t_GHA is the angle between the Prime Meridian and the GP, and λ_AP is the latitude of the AP. The GP of the celestial body is taken directly from an Astronomical Almanac.

$$t_{\mathrm{LHA_{AP}}} = t_{\mathrm{GHA}} + \lambda_{\mathrm{AP}},$$

Figure 55: Local Hour Angle of the AP. [62]

Next, the angle from the horizon to the celestial body can be calculated with the longitude, ψ_AP, and latitude of the AP, as well as the Local Hour Angle and Declination of the celestial body.

$$Hc = \arcsin(\sin\varphi_{\mathrm{AP}} \cdot \sin\Delta + \cos\varphi_{\mathrm{AP}} \cdot \cos\Delta \cdot \cos t_{\mathrm{LHA_{AP}}})$$

Figure 56: Computed altitude of the observed celestial body. [62]

We can then calculate the azimuth, which is the angle from North and to the H_c.

$$Az_1 = \arccos\frac{\sin\Delta - \sin Hc \cdot \sin\varphi_{\mathrm{AP}}}{\cos Hc \cdot \cos\varphi_{\mathrm{AP}}}$$
$$Az = \begin{cases} Az_1 & \text{if } 180^\circ < t_{\mathrm{LHA}} < 360^\circ \\ 360^\circ - Az_1 & \text{if } 0^\circ < t_{\mathrm{LHA}} < 180^\circ \end{cases}$$

Figure 57: Compute the azimuth at AP. [62]

This allows us to calculate the parameters A, B, C, D, and E, which are parameters that can estimate the displacement from our assumed position to an estimated position rather than redrawing circles to find where the intercept is. ΔH = H_o - H_c, where H_o = the actual measured angle from the celestial body to the horizon at the rover's position rather than the AP:

$$\begin{cases} A = \sum_{i=1}^{n} \cos^2 Az_i, & B = \sum_{i=1}^{n} \sin Az_i \cdot \cos Az_i, \\ C = \sum_{i=1}^{n} \sin^2 Az_i, & \\ D = \sum_{i=1}^{n} (\Delta H)_i \cdot \cos Az_i, & E = \sum_{i=1}^{n} (\Delta H)_i \cdot \sin Az_i, \\ G = A \cdot C - B^2 & \end{cases}$$

Figure 58: Variables for an approximation without drawing lines. [62]

The rover's position can be then estimated by:

$$d\lambda = \frac{A \cdot E - B \cdot D}{G \cdot \cos\varphi_{\mathrm{AP}}}, \qquad d\varphi = \frac{C \cdot D - B \cdot E}{G},$$
$$\lambda = \lambda_{\mathrm{AP}} + d\lambda, \qquad \varphi = \varphi_{\mathrm{AP}} + d\varphi.$$

Figure 59: Calculate latitude and longitude. [62]

Where λ is the estimated latitude and ψ is the estimated longitude. The estimated position can be used as a new AP and the steps can be repeated at each iteration for each new captured image or new angle measurement of a celestial body.

Nautical Navigation without a Sextant

To calculate the angle from the celestial body to the horizon, we need to know where the horizon is. If we employ some sort of horizon detection and we know the fov angle for the camera facing the horizon, we can figure out the angle that the sextant is used to find. There is a version of the Astronomical Almanac online but their website is currently under renovation rendering some parts unusable but you can request for certain information. The only dependence relying on this is figuring out the GP on some point on the moon for a celestial body given a date and time. It could be possible to do without this and figure out the GP by calculating the distance you are from where the celestial body would appear to be directly overhead ((90 - angle measured to horizon) * (moon's surface area/360)), but this would only give you latitude. To calculate the longitude, it's not as intuitive because unlike latitude, the distance between these lines varies depending on how close you are to one of the poles. According to [63], the longitude can be calculated by knowing the time and how many hours in a day (360 / Number of hours in a day = number degrees per hour). So if you observe the sun at the highest point at say 4:00, then your longitude is (4 * degrees per hour). The time at which the sun is at the highest point may not necessarily be at noon and will depend on where you are.

Park Ranger

The common approach to localize the robot is by using some sort of SLAM. Used in an unknown environment, SLAM creates a 3D map of the environment and localizes the robot within it. Due to the nature of space exploration, NASA often seems to send out some sort of surveyor satellite to gather information before they send expensive equipment out on the surface of a celestial body such as Mars they wish to study. Working off this assumption, it would be beneficial to utilize the overhead imagery akin to a map of the area. If we have a map of the

area, this makes SLAM less desirable. SLAM would be useful in creating an accurate 3D map but would need to be done by a team of rovers and take quite a long time to cover all of the moon's surface. By providing more map information, it comes at a higher cost of memory and computation time, not to mention a strain on the battery of the rover this software is meant for. SLAM could prove promising in the future if there was a type of rover dedicated to mapping the surface but EZ-RASSOR was built for mini-RASSOR, an educational, scaled down, mining rover.

Although we can get a location from the Cosmic GPS to around within 1 nautical mile, it's not as accurate as the requirements would need. We do plan to combine it with odometry readings but it would help if we could perform dead reckoning with something else to help with accuracy. An idea we came up with to suit this need is called Park Ranger.

Park Ranger, named after those who look after our national parks, refers to using things such as terrain, landmarks, etc. and a map to figure out where you are. This approach is inspired by a couple different papers all with different things that were used to help localize the robot. The different objects used to match to the map are listed in order of least to most feasible in the scope of our project: Man-made lines, boulder, background, and horizon.

Man-Made Lines

For any kind of image based matching, the best way to improve accuracy is by having something "interesting" in an image. In approaches like [64], [65], and [66], man-made structures are used to localize. Buildings and roads are distinct enough, often being quite harsh against their surroundings particularly in rural areas. Rather than using whole buildings, [65] uses the vertical edges of the corners of buildings to match to a satellite image of the area. In cases where there are sparse buildings and the roads are more dirt paths, [66] uses a 360 panorama and transforms it into a top view of the scene for matching. The problem with these approaches is the fact that there will probably be very sparse man-made buildings and not much that could be distinguished as a road, at least in the beginning of a non-Earth colony.

Tracks of a rover could be used as a type of “road” due to the packing nature of lunar soil as a sort of path following but would be useless when matching to satellite data and especially when navigating to never visited destinations.

Boulders

Since we can’t rely on types of man-made structures or even tracks of a rover when navigating to a brand new location, we need to rely on the natural terrain. [67] describes determining 3D points of clusters of boulders, translating these points to absolute points using visual odometry, and then projecting these points vertically to obtain 2D points to match to an overhead image of the area. The problem with this approach is that these boulders could look quite small and our map of the moon wouldn’t give us enough to match them efficiently enough to be useful.

Background

Originally the idea of Park Ranger was to use the boulders for finer localization. After realizing that it would only be feasible if we had a high quality overhead view of the area, which meant it would also be a cost of memory. The next idea was then to take all the 3D points taken from the scene and minus the boulders, essentially using the background to match to orbital data. This idea was inspired by [68], which talks about how different flavors of Visual SLAM work and how they compare to each other. The matching of the background comes mainly from the Robust Visual SLAM section, which refers to separating the background objects from the dynamic objects and then not throwing the dynamic ones out due to their ability to disappear and reappear across frames. A dynamic object doesn’t refer to a moving object but rather one that appears to move when the robot moves. It wouldn’t be as necessary to have a much higher quality map and the background should be more visible in an overhead image. The main problem, even mentioned in [68], is that there’s a case where the dynamic objects can obscure the view such that there is less background to reliably track.

Horizon Detection

Even though matching a set of 3D points to a satellite image is certainly feasible, it’s not robust to certain lighting conditions it could experience such as being at the bottom of the crater. Because of how the Sun hits the moon, the surface can be subjected to harsh shadows, which could cause poor illumination in craters.

We won't see the stars if it's daytime on camera due to the Sun and the Earth so the only other thing we could use is the edge that the top of the crater makes against the sky. In other situations, the horizon is also less subjected to poor illumination or occlusion than our direct surroundings. It would also be useful to localize with the horizon since the cosmic GPS requires it to determine a non-sextant measurement.

DEM vs. Overhead Image

To localize with the horizon, it's necessary to compare the query horizon to a digital elevation map (DEM), which could be stamped of where it was obtained from this "map". The problem with DEMs is that they are generally quite large files when compared to a normal image. This means it would likely be faster to attempt to match something like boulders if you attempt to go through the whole file than to exhaustively search a DEM. In practice, the computation to search through the DEM is reduced by either sampling it way ahead of time of localization usage or it's broken up into chunks, only loading a specific chunk to compare to. You could attempt to match a horizon with a normal overhead image but when map the query horizon to the overhead, there's a chance it will just look like a line and won't be distinct enough to narrow down the possible locations, rendering Park Ranger useless.

Match Query Horizon to DEM

Since [69]'s method focuses on localizing with mountain peak horizons, it samples the DEM for the horizons with the highest maxima. The peaks in the query image are reconstructed by determining 3D points, with more emphasis on determining the correct scale of points on the horizon than the scale from camera to horizon. Another method is to load up part of the DEM that you want to compare to if you know a certain area you could be and not have to load up the whole DEM. [70] splits the DEM into slightly lower quality tiles that are loaded into memory according to an estimated location to match the query extracted horizon but their approach assumes that there is a GPU that can offload this workload from the CPU. Some approaches focus on narrowing down potential matches. [71]'s novelty consists of developing a robust representation of the horizon as well as using two different voting schemes to determine the position from the horizon. First type of voting determines where in the map the query horizon is most likely to contain it by use of the horizon descriptor. The second

type of voting utilizes the descriptor comparison and the rotation of the query horizon.

Match DEM to Query Horizon

Most approaches compare the image to DEM data but [72] instead compares the DEM to the query horizon. [72] first throws the 3 different color channels (RGB) of the query picture into two neural networks and resulting pixels that were classified as part of the horizon are obtained from the Canny filtered version of the original image. Two databases are created from the DEM: map peak database and map feature point database. The map peak database contains the highest peaks while the map feature database contains all peaks. Assuming that a query image has at least one or two peaks, a potential match is selected by choosing a peak from the map peak database and two physically closest peaks to it are obtained from the map feature database. These peaks are then projected onto the query image, if the horizon generated by them is higher than the query horizon, it's eliminated. Otherwise, it's kept as a potential match and these potential matches are ranked to how much they line up to the actual horizon.

Problems with Park Ranger

Regardless of our approach, there is always some potential error due to the nature of extracting information from an image and matching this to another set of data. If we use this with the Cosmic GPS, it could be used to refine the estimated location from the stars but it's possible that the matching produces a result that neither agrees with Odometry nor Cosmic GPS. This means a set of rules would need to be implemented to determine which estimate takes precedence or an estimate might need to be thrown out if an estimate doesn't line up with the others at all.

Odometry

Background

Odometry is the estimation of the change in position of an object over time. In robotics, it is typically used to estimate how much a robot has moved from a starting location. This can involve the use of data from onboard sensors and/or

cameras, as well as calculations based on how many times the wheels of the robot have been spun. However, for the EZ-RASSOR, using the amount of times the wheels have been spun would likely be highly inaccurate due to the amount of wheel slippage that will occur in environments like the surface of the moon or Mars. Thus, we will utilize data from the onboard inertial measurement unit (IMU) and possibly from onboard cameras, such as the forward-facing stereo camera.

After speaking with the SwampWorks team, we learned that it is safe to assume that the rover will know its initial location (because it will typically be deployed from a base or lander that is at a known location). Thus, we can combine absolute localization and odometry to estimate the coordinates of the rover. Using odometry in addition to absolute localization is important because the absolute localization techniques we are researching are relatively slow, and it would not be feasible to be updating our location constantly using only these absolute localization techniques. Odometry using sensor and/or camera data will serve as a fast, constantly-running technique for estimating the rover's current location. However, odometry techniques typically have drift error (the farther away you move from the starting location, the less accurate the estimation of your current location becomes), so localization techniques are still necessary. The absolute localization system will periodically "reset" the starting location of the rover that the odometry system uses. This is done in the hopes that it will reduce the amount of drift error that occurs.

Wheel Odometry

The simplest way to measure how far a robot has traveled is to perform wheel odometry, which uses the number of times each of the wheels has been rotated to calculate the distance and direction the robot has moved. In ROS, the nodes that control the hardware for the wheels report the total number of "ticks" (the number of incremental rotations) of each wheel. The number of ticks by itself does not help us, but it does if we know the distance that the wheels travels each tick.

There are two approaches to measuring the distance that the wheels travel each tick. The first is to determine it experimentally. This is done by having the robot move in a straight line for a set distance. While the robot is moving, the number of ticks over that distance, which can be observed in ROS, is measured. Then, the distance traveled per tick can easily be calculated. For example, if there are

20 ticks when the robot moves 1 meter, that tells us that the robot moves 1/20 = 0.05 meters each tick. The other way to determine distance traveled per tick is to calculate it based on the attributes of the wheel and motor. Knowing the diameter of the wheel, we can calculate its circumference by multiplying by pi. The number of ticks in a complete revolution of the wheel must also be found. This can be found in the specifications for the motor you are using; in the specifications, it may be referred to as the encoder resolution. The distance traveled per tick can then be found using the following formula:

$$Distance\ Traveled\ Per\ Tick\ =\ \frac{Wheel\ Diameter \times \pi}{Ticks\ per\ Revolution}$$

Both approaches have advantages and disadvantages. Calculating the distance traveled per tick experimentally will likely give you an accurate result for the surface you tested on, but if the robot were placed on a different surface, then the true value would probably be much different due to factors such as slippage. Calculating the distance traveled per tick based on the attributes of the wheel and motor is free from experimental error and will likely give you a good general value, but it does not take into account slippage in any way. Therefore, it will likely be erroneous in environments with high slippage. The SwampWorks team has a bin full of regolith-like material that they use to test the RASSOR and Mini-RASSOR, so if we are able to use that environment we could determine the distance traveled per tick experimentally. Otherwise, we could just calculate the theoretical value with the approach shown earlier.

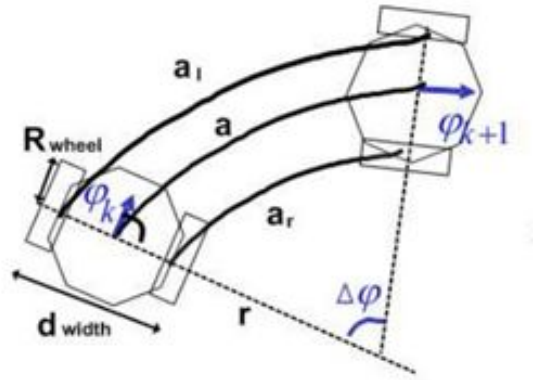


Figure 60: How a differential drive robot moves based on the velocity of its left and right wheels. [73]

Once we have determined the value for distance per tick, we can determine the distance the wheel has moved simply by multiplying the distance traveled per tick by the number of ticks since the last odometry update. Since the nodes controlling the wheel motors post just the total number of ticks, at each odometry update the number of ticks must be stored so that in the next update the number of ticks since the last update can be calculated by subtracting the previous total number of ticks from the most recent total number of ticks.

In a differential drive, the wheels on either side of the robot should have the same velocity, so the wheels on each side can be viewed as one wheel for the purpose of calculations, as shown in the previous figure. To determine the overall distance that the robot has traveled, calculate the average of the distances traveled by the left and right wheels. This will give us the distance the robot has moved but not the direction. To compute the angle that the robot rotated, the distance between the left and right wheels must be measured. Once that distance is known, the angle (in radians) that the robot rotates is given by the following formula:

$$Rotated\ Angle\ =\ \frac{Left\ Wheel\ Distance\ Traveled - Right\ Wheel\ Distance\ Traveled}{Distance\ Between\ Wheels}$$

For the distances and angles discussed previously, velocity can be estimated by dividing the change in distance or angle since the last odometry update by the time since the last update.

While wheel odometry is fairly straightforward to implement compared to other forms of odometry, it is the least accurate in most situations. It is extremely susceptible to errors due to wheel slippage. As an extreme example, if a robot is stuck in mud and spins its wheels while not actually moving, wheel odometry will report that the robot has been moving the whole time. The surface of the moon has high slippage, so we will not be able to rely heavily on the readings from wheel odometry. However, wheel odometry is useful when used in conjunction with other forms of odometry. IMU odometry, which uses sensor data to estimate how much the robot has moved independent of the movement of the wheels, can help account for the error due to slippage. Wheel odometry can be used as a general guide for the direction the rover should be moving, while other forms of odometry can be used to determine how much the rover is actually moving. Additionally, wheel odometry can be used to detect slippage when used in conjunction with other forms of odometry. For example, if there are large

discrepancies between the wheel odometry readings and other odometry readings, the rover is most likely in a high-slippage area. This can be reported so that in future traversals the rover avoids this area. Also, if the robot is supposed to be moving according to wheel odometry but is not moving according to other forms of odometry, the robot is likely stuck, and it can send out a distress signal.

IMU Odometry

Since we know that the Mini-RASSOR is already equipped with an onboard IMU (the BMI055 IMU from Bosch Sensortec), we should utilize IMU data to assist in odometry. The BMI055 IMU consists of an accelerometer and a gyroscope. The accelerometer measures the linear acceleration of the rover in three directions: X, Y, and Z. The gyroscope measures the angular velocity of the rover in three directions: roll, pitch, and yaw. Based on the data that is collected, the change in position of the rover can be calculated through integration.

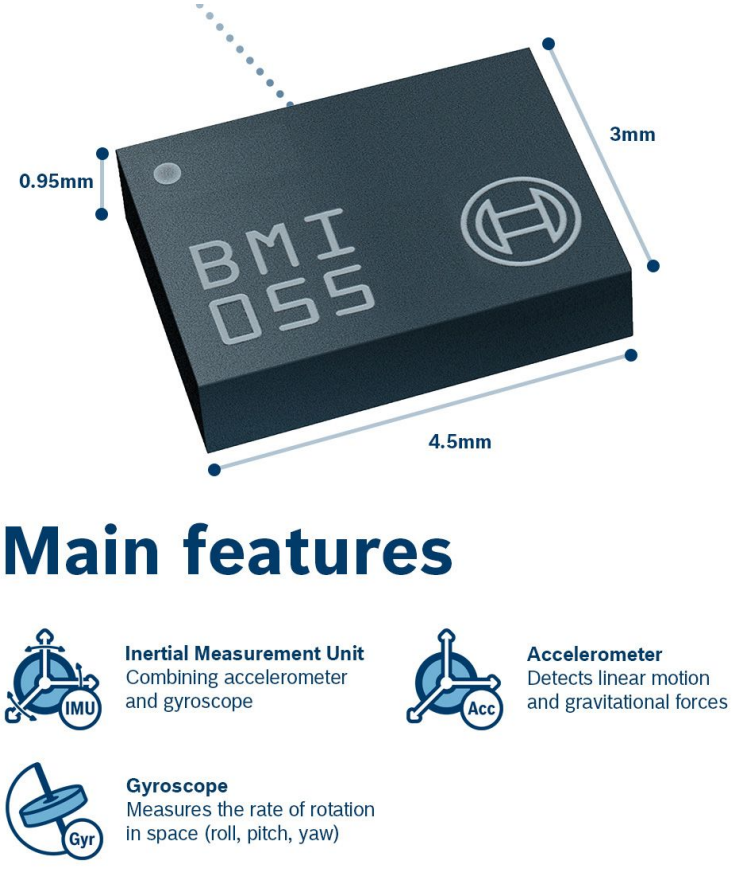


Figure 61: The BMI055 Inertial Measurement Unit. [74]

The main issue with using IMUs for odometry is that the odometry measurements are extremely susceptible to drift error. The IMU odometry system starts by the position of the system being initialized from a known location (for EZ-RASSOR, this could be a lander or base that the rover is deployed from). Then, the change in position, which is calculated based on the acceleration measured by the IMU, is added to the initial location to determine a new location for the system. This process is repeated continually: estimated changes in position are used to determine the new location of the system. The problem with this is that errors are carried over at each step. Even if the errors are tiny, they accumulate as the system moves and lead to inaccuracy. Over large distances, this leads to large discrepancies between where the system thinks it is and where it actually is. Thus, using IMU data alone is not a reliable way to perform odometry.

Visual Odometry

Visual odometry is a cheap type of odometry that can be used for multiple tasks that a robot may need. Cameras can be used for both relatively and absolutely localizing a robot. Cameras can be used to perform relative localization by tracking objects over time to estimate the motion of the robot; absolute localization can be performed by identifying or triangulating objects in the surrounding environment. Here, we will focus on relative localization using cameras in the form of visual odometry. Despite being able to be used for a variety of tasks, localization techniques using cameras are sensitive to environmental factors such as lighting and terrain. We will go over the content in [75] for most of this section, which is a review of visual odometry papers.

Types of Visual Odometry

Monocular camera

A single camera is easy to set up and is cost-effective for odometry. Without having to calibrate multiple cameras, it does come with trade-offs. Because there is only a single perspective of the scene, determining the distances of objects from the camera and estimating motion in two directions (such as northeast or southwest) can be difficult.

Stereo camera

Adding an additional camera helps determine the depth of objects, which in turn helps with determining the size of objects. Knowing the position of each camera

and focal length, we can determine the location of an object by triangulation. This is why stereo cameras are typically preferred over single cameras for tasks like localization. The problems with stereo cameras are that they are more expensive, need more calibration, and need a way to sync the footage across cameras if it's not built in. If the distance from the object to the cameras is vastly larger than the distance between the cameras, it's no better than the monocular camera at determining scale; this is similar to trying to determine someone's height from far away.

Omnidirectional

By having an omnidirectional setup, the field of view is quite large and can be up to 360 degrees. Because of this, a well defined 3D model of the world can be constructed and is independent of rotation. Like with the stereo camera, there is the cost of calibration and synchronization across the system as well as a monetary cost.

Visual Odometry Based Localization Approaches

Feature-based

Feature-based refers to extracting features across sequential frames and estimating motion by comparing the apparent movement of distinct features. If using stereo cameras, 3D points can be constructed and feature points can be extracted to determine motion between frames t and t+1.

Appearance-based

Appearance-based refers to change in appearance and intensity values of pixels across frames. Optical Flow (OF) is used to determine motion and speed. OF algorithms use the intensity of neighborhood pixels to determine differences in brightness across images. Dense OF algorithms consider the difference in all image pixels while Sparse OF algorithms focus on the difference of a certain number of pixels across frames. Sparse OF algorithms are used more than Dense OF due to the fact that Dense OFs don't hold as well against noise.

A common appearance-based method is template matching. This method is also applied to other problems such as object detection and video compression. Template matching extracts a subimage from the main image and uses it to compare with areas of another image to determine how similar a section is.

When applied to localization, this can be used to estimate the displacement of the robot and the rotation angle.

Hybrid

Hybrid approaches are developed to get the best of both worlds since one works better than the other depending on the case. If the environment is heavy with texture, it's more optimal to go with feature-based. In contrast, if it's low-texture, appearance-based will outperform feature-based.

Positioning of Camera(s)

Downward

A downward-facing camera setup would be less sensitive to dynamic environment factors such as change in light and wind. It would also be less subjected to dirt or other things getting on the lens. Even though a camera in this setup would have more constant lighting conditions, it is likely to be in constant shadow and have lots of blur because it would observe faster motion (since the ground appears to move much faster than, for example, a rock you're passing from a distance). This camera could be a feasible approach for aerial type vehicles/robots but not for ground-based vehicles/robots.

Front-Facing

The front-facing camera setup has the significant advantage of obtaining more information than a downward-facing camera for most cases. Although it's more sensitive to environment changes, it also is more versatile.

Problems with Visual Odometry

The main problems with visual odometry are the computation power needed and sensitivity to changes in lighting and shadows from objects in the environment or even the robot itself. The other factor intrinsic to any camera system is the scale uncertainty. Although stereo cameras allow for the scale and distance of objects to be determined through triangulation, if the observed object or scene is far enough away, the scale ability of the stereo cameras can be equated to a monocular camera. When determining scale with a monocular camera, an absolute scale can only be determined with motion, compared with a known object scale in the scene, or additional sensors. Additional sensors can also

account for things such as wheel slippage on uneven terrain, which would cause some apparent motion but the actual displacement of the vehicle is less than observed.

Rocky 8

[76] entails a rover prototype called Rocky 8, which was the predecessor to the Mars Exploration Rovers Spirit and Opportunity. The general overview of the Rocky 8 system comprises visual odometry, full vehicle kinematics, and an IMU. The combination of these allow for a relative localization that is built to compensate for wheel slippage and drift from the planned path. First, the visual odometry determines motion from apparent visual motion and is combined with IMU data in a Kalman Filter to give a better estimated motion. Then, this is compared to the data from the vehicle kinematics to determine how much slippage has occurred. If the slippage is below a threshold, they are used to complement each other. Otherwise, a slippage vector is obtained by calculating the difference between the results of the Kalman Filter and the vehicle kinematics. This slippage vector is then used with the planned path vector to calculate quantities such as wheel rotation to adjust the rover's trajectory to align with the correct path.

Mars Exploration Rovers

Rocky 8 served as a prototype rover for the Mars Exploration Rovers, and some of the features made it to the Mars Exploration Rover (MER) system. [77] focuses mainly on how the visual odometry system performed outside the testing environment on Mars. Because each step in the system required almost 3 minutes of CPU time, the visual odometry was only incurred in 3 situations: short drives up steep slopes, possible wheel draggage like in digging trenches, or driving through sand. Sometimes the system could not determine a solution due to either large apparent motion or the images didn't have enough "interesting" points to track. The highlight was the ability to promote rover safety by correctly identifying wheel slippage and stopping a task if it hasn't been making significant progress due to slippage.

ROS Packages for Visual Odometry

We will most likely be using an open-source ROS package to assist us in performing visual odometry instead of implementing all of it ourselves. There are

several reasons for this. First, the stereo camera we will be using will be outputting images with RGB-D data (RGB values and depth values) rather than just RGB data. The extra depth data available can make visual odometry more accurate, but writing real-time algorithms that utilize all of this data is extremely complex and is an active topic of research. Second, we're not sure if visual odometry will even be useful for the EZ-RASSOR. There are not as many visual features when traversing the moon as, for example, when traversing the inside of a building or room, so visual odometry may perform poorly. Thus, we think it would be best not to spend a significant portion of our time implementing a complex component of the project from scratch when that component may not even be used in the end.

When researching ROS visual odometry packages, there were several things we were looking for. First, the package must be able to process RGB-D data. We want to be sure that we're utilizing the depth data we have because RGB data on its own might not be of much use (since the moon is mostly the same color). Second, the package must be updated to the same version of ROS that we are using; currently, this is ROS Kinetic. Third, we would like the package to implement visual odometry frame-to-frame instead of using a persistent model. Frame-to-frame implementations only compare visual features between the most recent frames, but persistent models store visual features so that you can compare the visual features in the current frame with visual features from previous frames. Persistent models work well for enclosed environments where you expect the robot to pass by the same area multiple times; when the robot passes by an area it has seen before, this approach allows the robot to quickly determine where it is. In a large, moon-like environment, we are not expecting the rover to pass by the same areas often. Storing the visual features seen would be a waste of memory and would likely make the algorithm slower.

RTAB-Map

Real-Time Appearance-Based Mapping (RTAB-Map) is a ROS package to facilitate SLAM [78]. While we do not plan to use SLAM on the EZ-RASSOR, this package provides a node that performs visual odometry from RGB-D images. The node provides both a Frame-to-Frame mode and a Frame-to-Map mode. Frame-to-Frame mode matches visual features between consecutive frames, while Frame-to-Map mode uses a persistent model for storing visual features. As discussed earlier in this section, we believe a frame-to-frame approach would be

a better fit for EZ-RASSOR, so the following information pertains only to Frame-to-Frame mode. GoodFeaturesToTrack (GFTT) is the default method for feature detection [79]. GFTT detects strong corners in an image that are most likely to be able to be matched across different frames. However, the following feature detection methods are also supported: SURF, SIFT, ORB, FAST, BRISK, and KAZE. For feature tracking, optical flow is performed on the features extracted by the feature detection method. For motion estimation, Perspective-n-Point (PnP) random sample consensus (RANSAC) is used to determine the new pose (position and orientation) of the camera.

Fovis

Fovis is a ROS package that provides tools for visual odometry [80]. Included in the package is a node that performs visual odometry from RGB-D images. The node uses a frame-to-frame approach for visual odometry. For feature detection, Features from Accelerated Segment Test (FAST) is used [81]. FAST is a computationally-efficient algorithm for detecting corners in an image. Feature matching is performed by forming 80-byte descriptors (the brightness values of the 9x9 pixel patch around the feature, except for the bottom-rightmost pixel in the patch), then sum of absolute differences (SAD) is used to compare the similarity of features across different frames for matching. For motion estimation, Horn's absolute orientation method is used, followed by several refinings. Horn's absolute orientation method provides an estimate for motion by minimizing the Euclidean distance between feature matches.

The main drawback of Fovis is the lack of documentation and active development compared to RTAB-Map. There is a version of Fovis made for ROS Kinetic available on the Github repository for the package, but the last commit was made in December 2017. Future teams will eventually need to move EZ-RASSOR to a newer distribution of ROS or even to ROS 2. With the current state of Fovis development, it's likely that the package will not be updated to newer versions of ROS and will be unusable once the transition of EZ-RASSOR to a newer version of ROS is made.

Combining Multiple Measurements for Localization

As discussed earlier, using wheel or IMU data alone for odometry is not reliable. Incorporating visual odometry could increase accuracy, but it suffers similar problems to the other forms of odometry in that it is always calculating the new position of the system relative to the previous position, so any errors in the calculation carry over to future position estimates. Thus, even with wheel, IMU, and visual odometry, the rover will still likely encounter significant drift error when traveling over large distances. The most straight-forward way to fix the problem of drift error is to also incorporate data from an independent measurement of position, such as GPS. Since GPS does not take into account previous positions of the robot when estimating the current position, each position measurement is independent and drift error is not a problem. For this reason, GPS is extremely useful for bounding the amount of drift error present in a localization system.

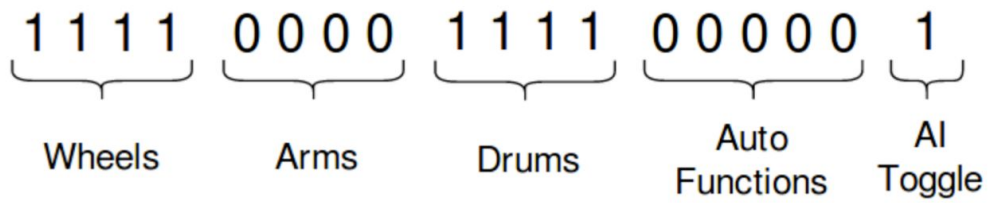
Synchronizing GPS and odometry data is not a simple task and is typically accomplished using a Kalman filter. Kalman filtering is a mathematical technique for combining different measurements of the same variable, where each measurement has its own inaccuracies, to produce an estimate of the variable that tends to be more accurate than any single input measurement. For our use case, the variable being measured and estimated would be the rover's pose: its position and orientation. Also, since GPS will not be available for the EZ-RASSOR, the coordinates generated by absolute localization techniques will be used as a substitute for the coordinates that would typically be produced by GPS.

Luckily, using multiple measurements of a robot's pose to produce an estimation of the robot's true pose is a common task for ROS users, so the robot_localization package has been developed to assist with this task. The package allows for multiple partial pose measurements to be supplied and an estimated pose is outputted [82]. Since some forms of measurement only measure some components of a robot's pose (for example, GPS would measure the robot's 3D position but not its orientation), supplied pose measurements are allowed to include only what components of the pose they measure. The robot_localization package offers both an extended Kalman filter and an unscented Kalman filter to combine the pose measurements appropriately.

Extended and unscented Kalman filters perform the same task, but the unscented filter offers higher accuracy in some scenarios at the expense of higher computational cost. By combining the different measurements of the robot's pose, the filter will output its own estimation of the robot's pose; this estimation should be more accurate than any of the inputs. For EZ-RASSOR, the sources of pose measurements can include IMU odometry, wheel odometry, visual odometry, and absolute localization.

Twist Message Support

To issue movement commands and toggle system modes, the previous team used a bitstring representation of the commands and toggles. The movement commands for the wheels, arms, and drums of the rover each have their own bits. For example, to control the arm, there are four-bit values, where 1 represents "do the action" and 0 represents "don't do the action": front arm up, front arm down, back arm up, back arm down. If conflicting commands are given (i.e., a command is given for front arm up and for front arm down at the same time), all conflicting commands are ignored. For the autonomous functionality of the rover, there are five-bit values to represent the different modes of autonomy: auto-drive, auto-dig, auto-dump, self-right, and full autonomous mode. There is also an AI "kill" bit. This is a safety feature that allows for autonomous functionality to be quickly turned off so that control is given back to a user (to prevent the rover from crashing, for example).



Wheel bits

Left side forward	1000
Left side back	0100
Right side forward	0010
Right side back	0001

Arm bits

Front arm up	1000
Front arm down	0100
Back arm up	0010
Back arm down	0001

Drum bits

Front arm dig	1000
Front arm dump	0100
Back arm dig	0010
Back arm down	0001

Autonomous bits

Auto-Drive	10000
Auto-Dig	01000
Auto-Dump	00100
Self-Right	00010
Full Autonomous	00001

Figure 62: The bitstring architecture for the current EZ-RASSOR codebase. [83]

While this is an efficient way to pass commands, it is highly specific to the EZ-RASSOR project. Since EZ-RASSOR is designed to be an open-source project developed by multiple, changing teams, the project should adhere to open-source standards as much as possible. In ROS, movement commands are typically issued using Twist messages. Twist messages represent two vectors in 3D space: one for linear velocity and another for angular velocity. From these two vectors, an accurate representation of a robot's desired movement can be determined. Replacing the bitstring representation with Twist messages would make EZ-RASSOR development easier for those already familiar with ROS and would open up opportunities for EZ-RASSOR to utilize open-source ROS packages. For example, the ROS navigation stack requires that the robot using the navigation stack uses Twist messages.

Additionally, the drivers for the motors that the SwampWorks team uses on the Mini-RASSOR require that Twist messages are used. The previous team did not find out about this until towards the end of the project, so they simply added a conversion of bitstring messages to Twist messages just before the messages

are posted to the topic that will issue the movement command for the hardware. While this works, it is not a permanent solution. Thus, our team will be responsible for rewriting the EZ-RASSOR architecture to use Twist messages instead of bitstring messages.

Mini-RASSOR

Physical Rover

The Mini-RASSOR, the rover that the EZ-RASSOR software is designed for, is actively being developed by engineers at Swamp Works. It consists of a chassis, four wheels, and two drums. Most of these parts are 3D-printable, making the rover relatively cheap and simple to construct. The main novelty of the RASSOR and the EZ-RASSOR rovers are their drums. On a planet with low gravity like the moon, digging is a difficult task. On Earth, heavy excavators use their weight to ensure that they can dig without themselves being moved. However, on a surface with low gravity like the moon, this becomes infeasible; the excavators would simply push themselves, without digging up much regolith in the process. Even if the force of gravity were the same on the moon as it is on Earth, it is extremely expensive to send heavy objects to space, so it would still be infeasible cost-wise. The RASSOR and EZ-RASSOR take care of this problem with their dual-drum system. The drums spin in opposite directions, canceling out the forces that would normally push the rover away before it could dig a large amount of regolith. The result is a rover that can dig and hold more regolith at a time than it weighs. These drums can be raised and lowered as needed, so that during travel the drums will be out of the way of the rover.

Figure 63: The Mini-RASSOR.

In terms of computing hardware, the Mini-RASSOR has an Intel Atom x5-Z8360 processor, and it has 4 gigabytes of RAM. Since the Intel Atom processor uses the x86 instruction set architecture (the most popular instruction set among personal computers and servers), software designed to run on most personal computers or servers can be run on the Mini-RASSOR. This will be helpful because it will allow us to develop without having to worry about software compatibility issues. For communication, the Mini-RASSOR uses standard wifi.

Currently, the Mini-RASSOR lacks sensors and cameras. However, the Swamp Works team is currently in the process of adding a depth camera (the Intel RealSense D435i), which has an onboard Inertial Measurement Unit (IMU). The depth camera images will provide a means for us to perform obstacle detection, and the IMU data will allow us to determine information about the movement of the rover. The depth camera will be placed on the bottom of the body of the rover and will face forward, gathering data on what is in front of the rover. Due to the positioning of the camera, it will be blocked by the drums when the drums are lowered (i.e., when the rover is digging). For navigation, this is not a problem as long as we program the rover to raise its drums whenever it is traveling. The Swamp Works team is also considering adding a single camera to the rover, which would be placed on the body of the rover and would face behind it. This would allow for the rover to form a more complete view of its surrounding environment.

Simulated Rover

To facilitate testing of the EZ-RASSOR software, a digital model of the Mini-RASSOR was created by the EZ-RASSOR 1.0 team The team originally created the digital model because they had no physical rover to test on: the development of the physical Mini-RASSOR was still in progress while they were developing the EZ-RASSOR software. Thus, creating a prototype to test on was crucial to the success of their project. The simulated Mini-RASSOR will serve as our testing platform as well. The Swamp Works team recommended we focus on the simulation, with running and testing EZ-RASSOR on the physical Mini-RASSOR being a stretch goal.

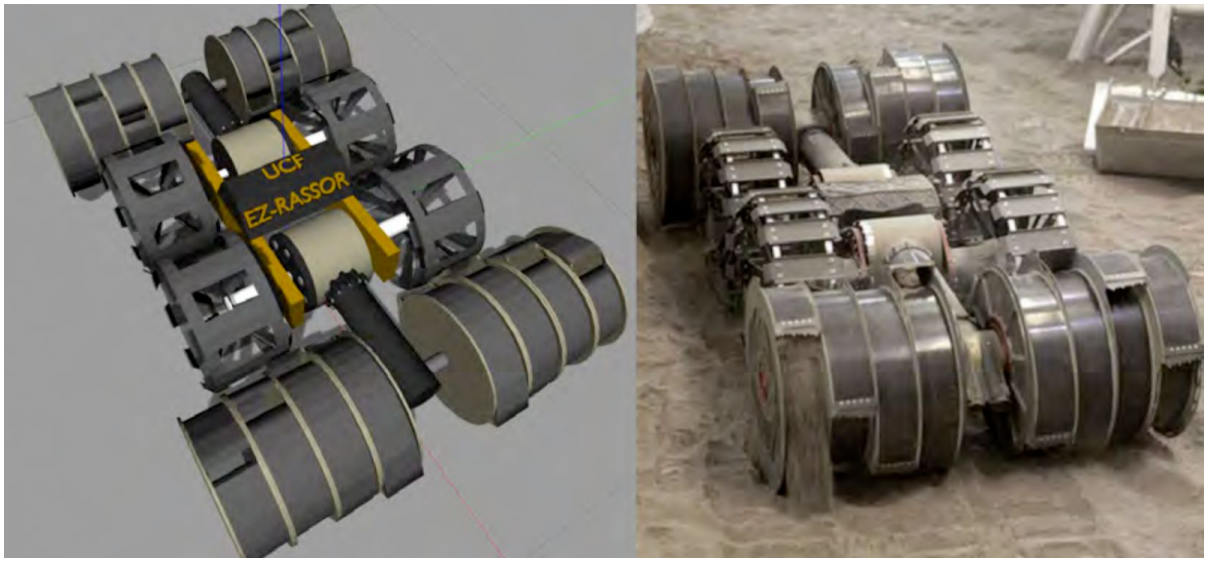


Figure 64: The digital model of the Mini-RASSOR (left) and the physical RASSOR (right). [83]

The digital model of the Mini-RASSOR was developed iteratively by the previous EZ-RASSOR team. Initially, a very simple prototype was made so that EZ-RASSOR code could be tested while the development of a more sophisticated model was in progress. As the project progressed, more accurate models were created, resulting in the most recent version that is shown above. All of these models were developed using Blender, an open-source application that can be used to develop models of 3D objects. By exporting a Blender model as COLLADA files, the model can be used in Gazebo. Thus, Blender is easy to integrate with the Gazebo simulation environment.

Adding Hardware

Since the Mini-RASSOR does not currently have many set plans for sensors, this gives us the flexibility to recommend hardware that will suit our needs to fulfill our requirements. We will be testing EZ-RASSOR in the simulated environment, so adding and testing hardware on the simulated Mini-RASSOR is essentially free. For this reason, we have agreed upon the following plans with the Swamp Works team. When determining extra hardware to add to the simulation, we will first seek approval from the Swamp Works team, as they may deem the hardware would never feasibly be added to the physical Mini-RASSOR due to size, cost, or other reasons. Once the simulated hardware additions have been approved, we will add the extra hardware to the simulated rover. The Swamp Works team will then use the results of our project to determine whether the hardware will ultimately be added to the physical Mini-RASSOR.

Gazebo Simulation

Integrating ROS and Gazebo

To simulate the EZ-RASSOR code as accurately as possible, the ROS nodes that would be running on the physical hardware need to be simulated in a virtual environment. Thus, ROS nodes are spawned and run in a Gazebo simulation. Gazebo is an open source 3D robotics simulator that was used by the EZ-RASSOR 1.0 team to test their code in a simulated environment. Luckily, using Gazebo to test ROS code is a common task and there are a standard set of steps to make ROS and Gazebo work together smoothly.

First, a ROS launch file must be defined. This specifies which ROS nodes to spawn; it is not inherently tied to running a simulation. On a physical rover, a launch file would be used when the robot is starting up. A launch file written for a Gazebo simulation, however, contains extra information to pass to the Gazebo client, such as which robots to spawn (in our case, just one or multiple Mini-RASSORs) and the simulation environment to use. When defining a robot to be used in a simulation, information like the geometry, surface, and mass of its components must be defined. Both ROS and Gazebo have standard ways to define robots, but they use different formats (ROS uses the XML-based Universal

Robot Description Format, while Gazebo uses the Simulation Description Format). However, the package ROS-Gazebo takes care of the conversion automatically, so it is safe to use the ROS-standard Universal Robot Description Format (URDF). These robot definitions are referenced by the launch files when specifying which robots to spawn in the virtual environment. For the current EZ-RASSOR codebase, the launch files are located at packages/extras/ezrassor_launcher and the URDF robot definitions are located at packages/simulation/ezrassor_sim_description/urdf.

Next, ROS plugins must be used in Gazebo. While using a launch file to initialize the Gazebo simulation will successfully set up the virtual environment, additional functionality will likely be needed. Examples of uses for ROS plugins include simulated sensor data and external control of objects in the environment. Currently, ROS plugins are being used in EZ-RASSOR for a simulated stereo camera, a simulated inertial measurement unit (IMU), and for controlling the rover from a gamepad. The simulated sensor and camera collect data based on the simulated environment and post the data to ROS topics that our code can subscribe to. Similarly, the ROS plugin for controlling the rover from a gamepad works by posting the movement commands to a ROS topic. If additional cameras or sensors are needed, we can simply add more plugins to the project. As long as we follow the same naming scheme and data format, it makes no difference to the rest of the code whether it is being run on a physical rover or a simulated one.

Simulated Environments

Since EZ-RASSOR is designed for robots that operate on other planets, mainly the moon, the simulation environment should mimic those environments as much as possible. Additionally, objects should be able to be placed in the environment to test the rover's obstacle detection and avoidance functionality. To facilitate this, Gazebo allows for users to create and edit "worlds": 3D environments that a robot can be simulated in. These worlds can be stored to world files so that the specific simulated world to spawn a robot in can be specified in ROS launch files.

For testing the EZ-RASSOR project, the previous team developed three worlds. The first was an Earth-like environment with various obstacles placed in the environment. This type of world is simple to make because it uses the default

configurations in Gazebo. The other environments were a moon-like environment and a Mars-like environment. These are more complicated to make because the gravity, slippage, and terrain must be changed from the standard configuration. The gravity and slippage to use can be determined from online sources, but the terrain is not as simple. For terrain, Digital Elevation Models (DEMs) were used. DEMs are 3D maps of a planet's surface and are typically created using data from spacecraft that have orbited the planet. Gazebo has a built-in DEM for the moon, so the terrain in the simulated environment could be initialized based on that data. For the Mars world, however, a DEM available online was imported into Gazebo. Originally the DEM was 11GB large, so the team used only a 129m x 129m section of the map to reduce the size. All three of these worlds can now easily be specified when launching the Gazebo simulation.

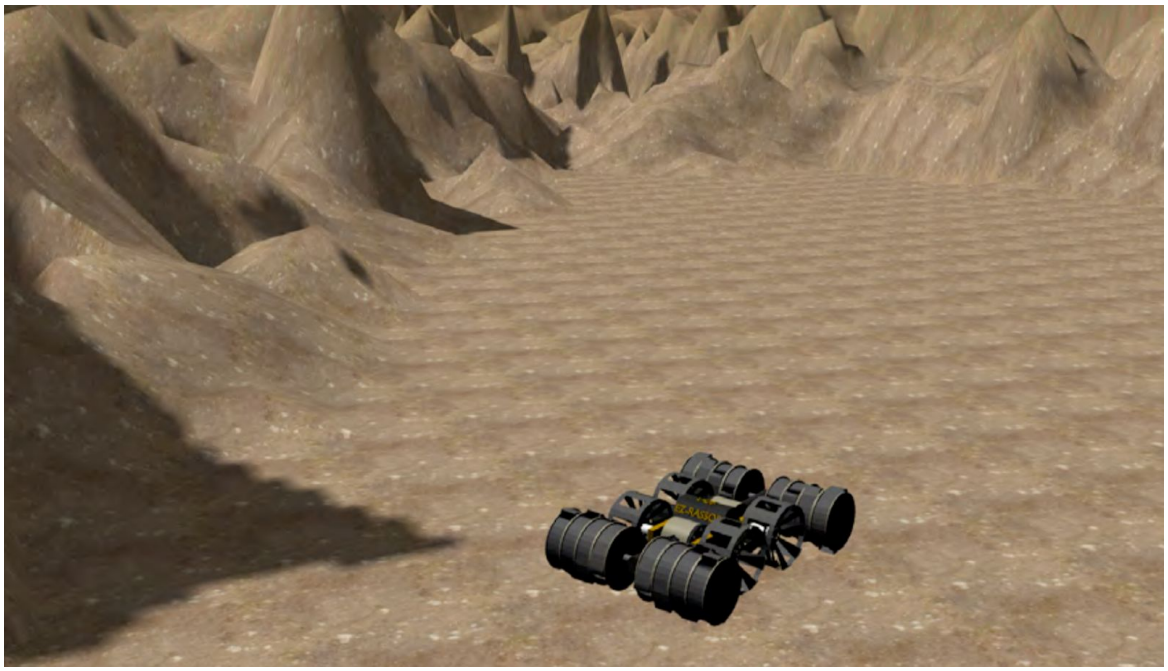


Figure 65: The Mini-RASSOR in a simulated Mars environment. [83]

For our team's purposes, these simulated environments will serve as a good starting point for when we are testing the navigation functionality. To test the obstacle detection and avoidance functionality, we will likely be adding more obstacles to the environments. Additionally, we plan to use the current configuration of the stars to help determine the location of the rover (referred to as celestial GPS), so an accurate representation of the stars should be added to the simulated environments. Gazebo has a starfield that can be enabled in a

scene via the built-in SkyX package, but it is just a picture of a starfield projected in the sky of the simulated environment. One solution could be to replace the default picture of the starfield with one taken from a known location, then have our rover's simulated camera pointed at this image in the sky. Gazebo is open-source, so if necessary we can simply place our own starfield picture in the location of the default starfield picture. Another solution could be to use the ROS Video Plugin for Gazebo. This allows for video to be shown on a texture in the world. We could place an image of the starfield directly in front of a simulated camera, then shift the perspective of the image as the rover moves to simulate how the rover's perspective of the stars would actually shift as the rover traverses the moon. If neither of these solutions work, we can test the accuracy of celestial GPS separately from the rest of the project. In the simulation, we could assume the error of celestial GPS is some constant (which would be determined through tests of our celestial GPS system outside of the simulation) and use the simulation to test the other components of the project.

Design Summary

First, the robot receives the coordinates of a destination that it must travel to. Before moving to that destination, the robot must determine its own location and orientation. If the robot knows its starting location and orientation, odometry can be used to provide an estimate of the robot's current location and orientation. Otherwise, the robot can use a picture of the sky and horizon to estimate its location and then refine this estimate by comparing its surroundings to a digital elevation model.

Then, the robot uses its depth camera to create a point cloud that is used to identify the directions and distances to the nearest obstacles the robot can see. The robot then determines how it must move to get closer to its destination while avoiding obstacles. This movement is communicated via Twist messages (rotational and linear velocity values). As the robot moves, a combination of wheel, IMU, and visual odometry is used to measure its movement. The robot uses the point cloud produced by the depth camera to further narrow down its location by comparing what it sees to a digital elevation model.

This design is summarized below. The only difference between the design and our implementation is that the Cosmic GPS (using a picture of the sky and horizon to derive the position of the robot) was implemented outside of the simulation environment. Because it was infeasible to accurately model the stars in a simulated environment, the Cosmic GPS was implemented outside of the simulation using real pictures of the night sky taken from Earth.

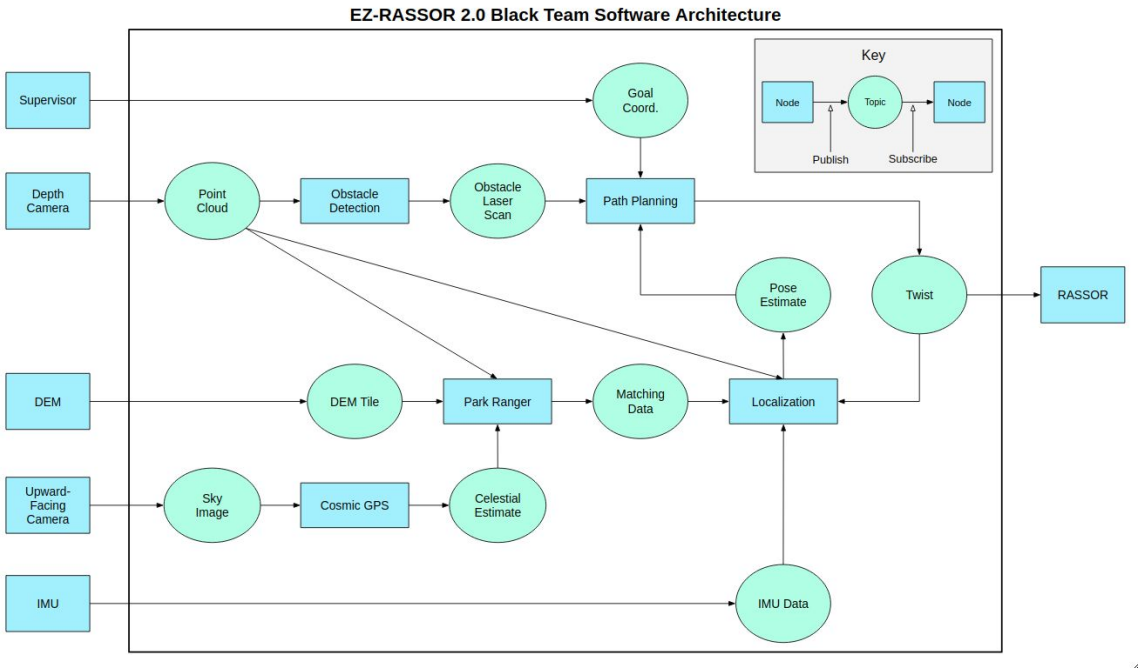


Figure 66: Software Architecture Diagram.

Design & Implementation

The Research section describes all the possibilities of how to implement solutions to the problems faced in this project. The Design & Implementation section describes the technologies, algorithms, and strategies that are used in the final product. Each section below describes what was done and why we did what we did. Then, the timeline for the development of each component of the project is outlined.

Obstacle Detection

Many hardware options were considered for collecting data on the obstacles facing the rover. The final decision is to use a depth camera that will produce a point cloud without extra software being written. The recommended camera to be used for a real-world application of this software is the Intel Depth Camera D435i. In our testing, this is simulated in Gazebo by a Kinect driver, which should give similar results. This camera creates a point cloud and colored depth (RGB-D) image. Additionally, the camera has an embedded IMU to measure movements and rotations of the robot that it is attached to [84]. Using a camera which automatically makes a point cloud is profoundly beneficial to the efficiency and effectiveness of the obstacle detection subsystem. The suggested camera can produce the point cloud faster than our software likely would because it is being calculated on a separate computer from the robot's motherboard, and the point cloud is likely more accurate than a custom one built by our team.

For obstacle detection, the NumPy library is heavily utilized in order to best optimize many of the operations performed on large arrays. First, the point cloud is read in using a NumPy method which directly accesses the buffer in the point cloud message. The point cloud is reshaped to easily be read using three-dimensional Cartesian coordinates. It is likely that there are not-a-numbers (nan) in the point cloud, so those are removed because they cannot be determined to be obstacles.

To begin processing the data in the point cloud, the *forward*, *right*, and *down* directional values are each sliced from the point cloud. This means that there is an array which gives how far in each direction the points are from the camera where each value at the same index of the arrays is describing the same point. Using the *forward* and *right* values, the points can be given two additional values: *step* and *distance*. The *step* is the discrete angle at which the point is closest to, from the camera's line of sight. The *distance* is the length across a flat floor which would need to be traveled before reaching the described point.

Next, all the points are grouped together by their *step* values and then sorted by *distance*. This sorting is not necessarily in-place. That means if multiple points have the same *step* and *distance*, they are randomly sorted amongst each other.

Each group is then iterated through to find the first obstacle that appears in them. By comparing the points at indices *i* and *i+1*, the *hike* and *slope* are calculated for each point. The *hike* is the change in *distance* between two points and the *slope* is calculated by taking the absolute value of the quotient of the differences between *distance* and *down* values of two points. Then, both the *slope* and *hike* are compared to a configurable threshold, to find if an obstacle exists.

The *slope* is good for finding above-ground obstacles. The slope threshold is the steepest angle which the robot is able to climb or descend into. The slope assumes that there is continuous, flat ground between the points that the robot can travel on. Hence, it is not good for detecting holes. In order to best detect holes in the ground, the *hike* value is used. The *hike* assumes that no such ground between the points exists and that there is an obstacle when two consecutive points are far away from each other. The threshold compared against *hike* represents the smallest diameter of a hole that the robot cannot travel over. Since the gap between points in the point cloud naturally increases as they get farther from the camera, the *hike* detection is unreliable above a certain distance. Thus, the *hike* comparisons halt after it reaches points beyond the configurable maximum distance that the Path Planning subsystem checks.

Then, the minimum *distance* of a detected obstacle between using *hike* and *slope* comparisons represents the closest point at which any sort of obstacle exists. Finally, a laser scan is created from the determined obstacle distances at each step. This laser scan is published for the use of the Path Planning subsystem.

Obstacle reporting was previously considered for this project, but has been left to be done as a possible future work. Given the limitations of what the depth camera can see and the requirements of EZ-RASSOR 2.0, it has been decided that obstacle reporting is not feasible or vital at this time.

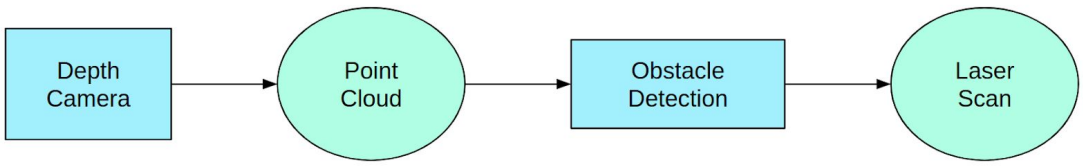


Figure 67: Design of the Object Detection subsystem

Development Timeline

Fall 2019

- Research — December 5, 2019
- Initial design — December 5, 2019

Spring 2020

- Implement point cloud collection — January 12, 2020
- Test point cloud collection — January 19, 2020
- Finalize point cloud collection — January 26, 2020
- Implement step and distance — February 2, 2020
- Test step and distance — February 9, 2020
- Implement step groups — February 23, 2020
- Test step groups — March 1, 2020
- Implement drop, hike, slope — March 15, 2020
- Test drop, hike, slope — March 22, 2020
- Finalize detection method — April 3, 2020

Odometry

We implemented odometry by combining wheel, IMU, and visual odometry using an extended Kalman filter. We chose to use an extended Kalman filter instead of an unscented Kalman filter because of the lower computational cost.

We implemented wheel odometry using the diff_drive_controller ROS package. This controller for differential-drive-based robots converts Twist messages to floating-point values to send to each motor to tell it how fast and in what direction to spin, and it performs wheel odometry automatically based on the Twist messages it receives. Previously, the floating-point values for controlling the motors were calculated manually, so we replaced that portion of the code with the diff_drive_controller package. An added benefit of using the controller instead of calculating the values manually is that the controller allowed us to easily specify constraints for the movement of the robot, such as minimum and maximum velocity and acceleration.

For IMU odometry, we were able to send IMU messages directly to the Kalman filter because the Kalman filter implementation we used supports both the IMU and Odometry ROS message types. We used the GazeboRosImu plugin to

simulate the IMU readings, with some noise added to the measurement to more accurately simulate real-world performance.

We implemented visual odometry using the RGB-D visual odometry node of the RTAB-Map ROS package. This node takes RGB-D images as input and outputs odometry messages. We chose to use this node from this package because it is highly customizable and well-documented.

For the extended Kalman filter, we used the ekf_localization_node node in the robot_localization ROS package. This package is the current ROS standard for Kalman filter implementations, and its Kalman filter node has many features that other ROS Kalman filter implementations do not have, such as support for an unlimited number of odometry sources and for several different ROS message types.

When configuring the extended Kalman filter, we needed to define what specific values each odometry source measures about the motion of the robot. Our definitions were as follows: IMU odometry measures forward linear acceleration and yaw, wheel odometry measures forward linear velocity and yaw angular velocity, and visual odometry measures forward linear velocity, side-to-side linear velocity, and yaw angular velocity. Since our robot uses a differential drive, it can only move forward/backward and rotate along the yaw axis. However, we also added side-to-side linear velocity for visual odometry so that slippage in that direction (i.e., if the robot is sliding sideways down a hill) can be measured.

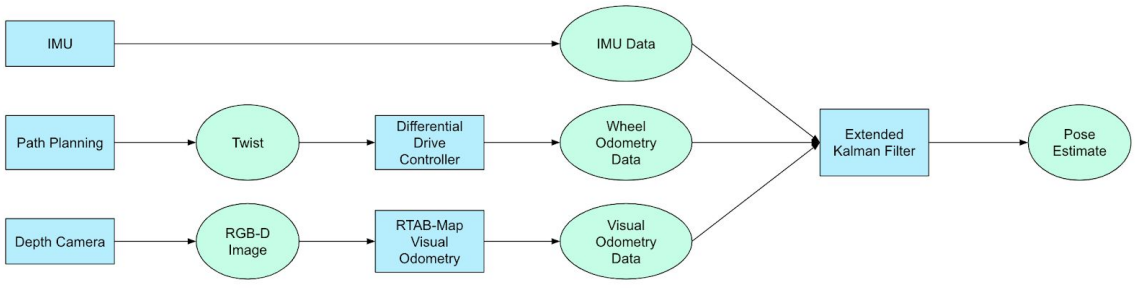


Figure 68: Design of the odometry component of the project.

Development Timeline

Fall 2019

- Research odometry — December 5, 2019
- Initial design of odometry — December 5, 2019

Spring 2020

- Add IMU odometry — January 15, 2020
- Test IMU odometry — January 31, 2020
- Add extended Kalman filter — January 31, 2020
- Test extended Kalman filter — February 15, 2020
- Add wheel odometry — February 15, 2020
- Test wheel odometry — February 29, 2020
- Add visual odometry — February 29, 2020
- Test visual odometry — March 15, 2020
- Testing and performance evaluation — March 31, 2020

Absolute Localization

For localization, we use two complementary methods, cosmic GPS and Park Ranger. With cosmic GPS we infer our absolute position by way of the stars. With park ranger we infer our absolute position using the immediate terrain features. These positions are represented by a lunar coordinate or distribution of them.

Cosmic GPS

For the cosmic GPS we have two hardware assumptions, a dedicated camera for capturing images of the sky and a mechanism that originates the face of the camera towards the robot's zenith. We used a samsung S7 phone camera, as the camera itself doesn’t matter, and gives excellent control over such aspects as the ISO and exposure time. As for the mechanism that orientates the face of the camera towards the robot’s zenith, we simulated this by placing the camera on a tripod and carefully pointing the camera towards the zenith.

For the cosmic GPS software pipeline, from the image capture to lunar coordinate, it first involved an image processing step of noise reduction via anisotropic diffusion. This allows for the background and the stars to be

smoothed without rounding the edges of the stars, thus decreasing the false positive rate while maintaining the accuracy of the center of each star. Then, after the noise of the image is reduced, automatic adaptive local thresholding or global thresholding is performed to find the “important” pixels of the image. These “important” pixels are then clustered into “blobs” that correspond to individual stars. With the stars recognized, the next step involves a centroiding algorithm to determine the subpixel central coordinate of each individual star, followed by the calculation of their respective angles (angular distance from center and from the forward direction). With the angles and distances, the five brightest stars are matched to five stars in a reference star graph using the method proposed in the constellation section. Once the stars have been identified, the three brightest stars are used to derive the robot's celestial position by scanning for a celestial coordinate which minimizes the difference from the measured distance to each of the three stars. Then, both the celestial position and the time the image was taken were used to derive the geographic position.

Park Ranger

In order for Park Ranger to work, there were some prerequisites. We needed to create a world that also matched the terrain from a digital elevation model (DEM), which we could read and use as the known map of the area. We created an auxiliary package called *dem_scripts*, which houses all the scripts that are directly related to the processing of a DEM.

Because DEM readers are not built into Ubuntu, we needed to use either an application, driver, or a library that can be used to read them in. In most applications and libraries, it is standard to use a library called GDAL [85] as a base library. GDAL is "a translator library for raster and vector geospatial data formats" of which includes support for PDS (NASA's Planetary Data System format) and GeoTiff (.tiff) [85]. Although Gazebo depends on a GDAL library, those dependencies only let Gazebo read DEMs and are not persistent outside of Gazebo. A search for GDAL libraries on a system with Gazebo installed will yield a positive match, but attempting to execute operations such as *gdal_translate* invokes an error. By consequence, attempting to install additional libraries for development with GDAL causes dependency conflicts between Gazebo and GDAL development libraries. So in order to read DEMs without breaking the environment to run the EZ-RASSOR simulation, a method to isolate the dependencies was necessary.

Given the three known choices at the time: VirtualMachine, Docker, or Anaconda. Docker was chosen for being not only lightweight compared to the other two, but also its modularity. In terms of implementation, each program has their own docker image associated with it. In our implementation, there is a *Dockerfile.base* and a *Dockerfile.child* file rather than the standard one Dockerfile per directory. Both *Dockerfile.base* and a *entrypoint.sh* are used for setting up the program to run as a local user rather than as root. Even though a docker container is used for isolation, it is insecure to run as root [86].

Another part of our implementation that is not standard is the fact that we use bind mounts to mount a host directory inside a docker container [87]. The most popular way to store data with Docker is through volumes but was not the most efficient solution for our application. Volumes are usually recommended for better security and efficient transfer of data from container to container [87]. Although we could store our data in volumes and pass it to other containers when dynamically created in the script, we ultimately need to get the results back to the host. This point alone makes bind mounts the best choice in this case.

In *dem_scripts*, we created *run_programs.sh*, of which takes in a keyword as an argument. When executing the *run* keyword, the user is asked how many subprograms to run and then the current subprogram they want to run. The user has four subprograms to choose at each iteration: *get_elev, mk_gaz_wrld*, *extract_tile*, or *convert2tif*.

- Get_elev - Given a GeoTiff (.tif), a text file is outputted with the corner geographic coordinates of the GeoTiff's (.tif) corners, dimensions of the GeoTiff (.tif), and all the elevation data. Also another text file is outputted with all the local maximas.
- Mk_gaz_wrld - Given a GeoTiff (.tif), a copy is created of the GeoTiff (.tif) in the form of a .jpg of a compatible size with Gazebo.
- Extract_tile - Given a GeoTiff (.tif), smaller GeoTiffs (.tif) or "tiles" are created.
- Convert2tif - Given a PDS (Planetary Data System) file of either a set of .lbl and .img or a set of .lbl, aux.xml, and .jp2, a copy is created of the PDS in the form a GeoTiff (.tif).

There is another program called *auto_gen_wrld,* which was not implemented in *run_programs.sh* due to time constraints. This program takes in an image (.jpg), which is used to generate a model package and world file. These are then piped to the corresponding directories to run a world based off of the image as the terrain or heightmap object.

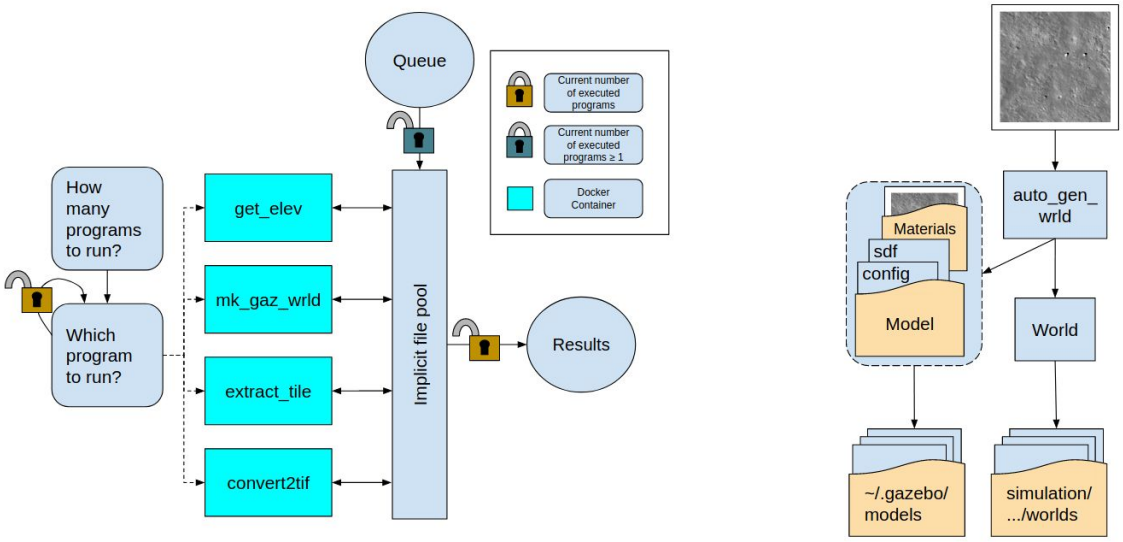


Figure 69: Diagram of programs in dem_scripts. *run_programs.sh* is on the left and *auto_gen_wrld* on the right.

Ultimately, in the final implementation of Park Ranger, we created a Particle Filter based algorithm to localize the rover based on its surroundings. Particle filters are a family of algorithms, where each particle represents a position that the robot could be in. Each particle contains a weight that represents how likely it is the robot is at that position, with higher weights representing a higher likelihood. The particles are initialized such that each position on the map is equally likely, then the weights and positions of the particles are updated according to sensor measurements until the positions of the particles converge. The weights of the particles are updated by comparing a point cloud representing the surrounding environment to a known DEM. The DEM is a grid of values, where the value in each cell represents the height of the ground at the area that cell represents. Park Ranger aims to find the cell in the grid where the robot is most likely to be, so a discrete particle filter is used [88]. A discrete particle filter is different from the standard continuous particle filter in that it uses discrete integer coordinates instead of continuous floating-point coordinates.

The main steps of the particle filter implementation are as follows: initialize, predict, update, estimate, and resample.

First, the initialization step generates particles at each position of the DEM, and each particle's weight is set to $\frac{1}{n^2}$, where *n* is both the height and width of an $n \times n$ DEM. The likelihood function is then called between the initialize and predict steps.

The likelihood function compares a *local DEM* and a *predicted DEM* for each particle to obtain a score; the particle's weight is then updated by multiplying it by the score. The most recent point cloud is converted to a top-down view to obtain the *local DEM*, and the *predicted DEM* is a section of the DEM corresponding to what the robot would expect to see if it were at the position of the particle. The facing angle used when determining this section is the most recent heading estimate from the odometry subsystem. To obtain a score for each particle, the histogram comparison technique is used to measure the similarity between the *local DEM* and *predicted DEM.*

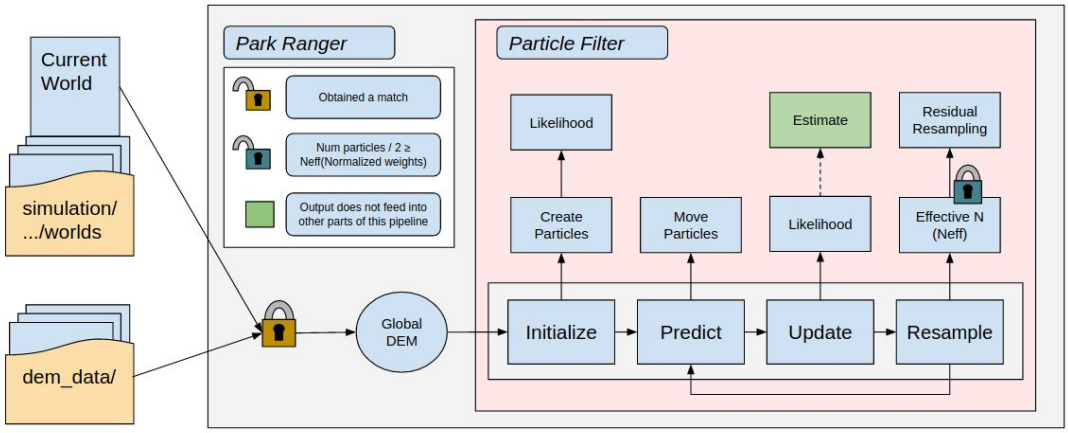


Figure 70: An overview of the Park Ranger pipeline.

The second step is the prediction step, which updates the position of each particle by adding the displacement of the most recent odometry position measurement from the previous odometry position measurement. The heading of each particle is set to the most recent odometry heading estimate.

Next, the update step invokes the likelihood function to update the weights of the particles. Then, the estimation step gives an estimated cartesian position of the robot by calculating a weighted average for the *x* and *y* coordinates of the particles.

Finally, the resample step is only invoked if the *neff* (effective number of particles) is less than half of the current number of particles. The *neff* function takes in the normalized weights of the particles and returns the inverse of the summation of the normalized weights. If resampling is invoked, then residual resampling is used to determine which particles to keep and which to remove from the particle filter. Then, the weights of the particles that were kept are set to $\frac{1}{m}$, where *m* is the number particles after the resample step. Then, steps 2 through 5 are repeated. Once the particle filter converges to a single position, only the prediction step affects the estimate.

To combine the location estimate with Odometry, we sent the location estimate produced by Park Ranger to the Kalman filter used by Odometry. This was accomplished by publishing an Odometry message from Park Ranger that contained only the predicted *x* and *y* positions of the robot. The prediction was only published once the particle filter had converged or nearly converged to a single position estimate. When configuring the new Park Ranger input to the Kalman filter, we specified to only use the *x* and *y* position estimates from the Odometry message (since these are the only variables Park Ranger predicts) and specified that the estimate was absolute, not relative.

Development Timeline

Fall 2019

- Research cosmic GPS — December 5, 2019
- Research park ranger — December 5, 2019
- Initial design of localization — December 5, 2019
- Camera + Image Proc (Proto) — December 15, 2019
- Celestial Body Recognition (Proto) — December 15, 2019
- Sight Reduction (Proto) — December 15, 2019

Spring 2020

- Park Ranger noise + edge — January 1, 2019
- Camera + Image Proc (Impl) — February 1, 2019
- Celestial Body Recognition (Impl) — February 1, 2019
- Sight Reduction (Impl) — February 1, 2019
- Park Ranger DEM create — February 1, 2019
- Camera + Image Proc (Test) — March 1, 2019
- Celestial Body Recognition (Test) — March 1, 2019
- Sight Reduction (Test) — March 1, 2019
- Complete Park Ranger — March 1, 2019
- Park Ranger (Test) — March 31, 2019

Path Planning

In this section we will present our path planning algorithm and discuss how it works and functions with ROS in greater detail. After analyzing many different path planning algorithms, we have decided that the most efficient and appropriate algorithm for our EZ-RASSOR project is the wedge bug algorithm. Wedge bug builds off of the tangent bug algorithm which we discussed in the previous section and works better with the constraints that are given to us by NASA (74 degree field of view and low to the ground sensors). The wedge bug algorithm can be thought of as a more complete and optimal version of tangent bug in that it is able to perform less sensor scans and choose more optimal paths in longer trips.

Similar to tangent bug, wedge bug uses sensors to predict where the most optimal and safe local path is at each and every step towards a goal. Think of the scans that wedge bug will use as little slices that have limited range and an arc length. An example of a slice or wedge can be seen below:

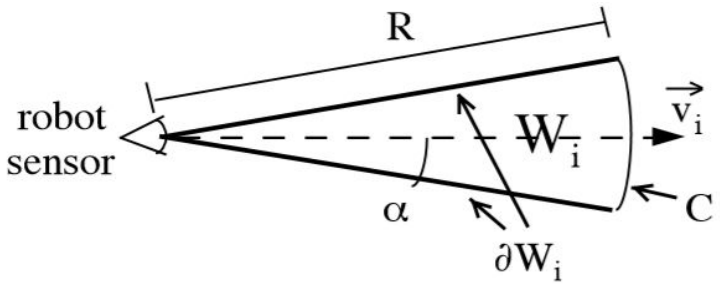


Figure 71: Example of a wedge for the wedge bug algorithm. [89]

As we can see from the image above, each wedge W_i is the field of view that EZ-RASSOR will have at each step of our path navigation. Each wedge will have a radius R, arc length C, a vector v_i that represents the distance from our robot to the final destination, an angle α between v_i and the border of our wedge ∂W_i. These wedges are primarily what wedge bug will be using to scan its environment to find the most optimal and safe path at each step of our unknown environment. Wedge bug uses two main modes: motion to goal and border following. Motion to goal can be thought of as normally how tangent bug finds and moves at each step and border following can be thought of as the way in which we look either to the left or right of our initial wedge to see if we can find the end of an obstacle to eliminate border following around longer obstacles.

The wedge bug algorithm starts out by initially calculating the vector xT where x represents the position of our EZ-RASSOR robot and T represents the position of our final destination. If there are no obstacles blocking our path to the goal, wedge bug will simply proceed to the goal. If there are obstacles blocking the goal the algorithm will then search a subgraph G1 of our initial wedge W_0. This subgraph, in which we search for a vertex V in which to move to, can be represented as G1(W_0). Before we explain what this set represents, LTG(W_0) stands for the local tangent graph of W_0. This is simply how EZ-RASSOR will be seeing the world through each wedge from outgoing tangent lines. In our case, obstacle boundaries appear as continuous contours in the range of data; the endpoints of these contours are called the “obstacle vertices”. Each endpoint e corresponds to a discontinuity in the range data or to an intersection of a contour with ∂W (border of a wedge) or C. An example of a LTG(W_0) can be seen below where F represents the focus points or point to move to at the current iteration.

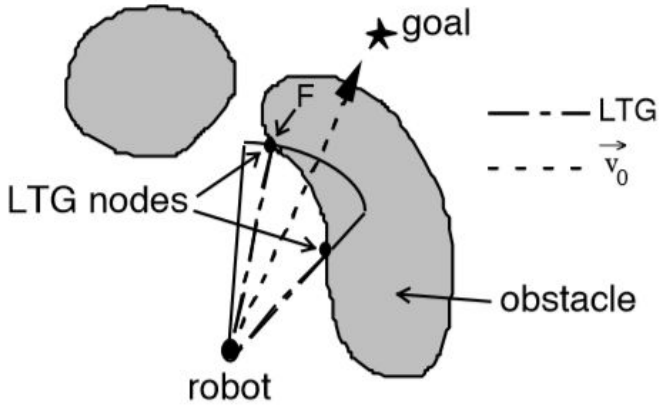


Figure 72: LTG calculated within W(x, v_0). [89]

At each step of the wedge bug algorithm, our goal is to find a vertex V such that G1(W_0) = { V∈LTG(W_0) | d(V,T) ≤ min(d(x,T), dLeave) } where dLeave is the distance from EZ-RASSOR's initial position to the goal, d(x,T) represent the distance from EZ-RASSOR to the goal and d(V,T) represent the distance from a vertex V to the goal. dLeave is used to ensure that EZ-RASSOR never strays farther than the initial distance from the goal. dLeave can be thought of as a circle of radius distance from our initial position to the goal with the goal at the center of the circle. Leaving this circle would indicate negative progress and can only be used in border following. The set G1(W_0) can be thought of as the set of vertices V such that they are elements of our local tangent graph for W_0 such that they are less than the min of (current distance of EZ-RASSOR to the goal or somewhere within the dLeave circle).

Now that we have discussed at a high level how wedge bug functions, we can now see an example of how motion to goal works.

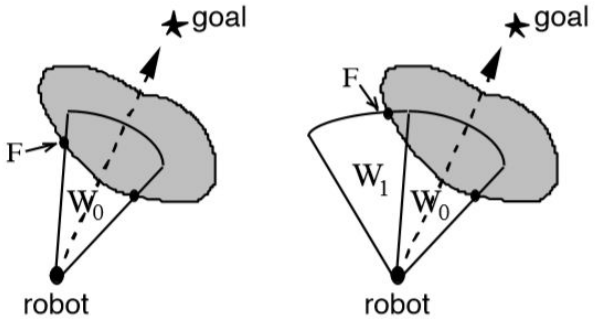


Figure 73: Example of virtual motion to goal. [89]

In this example, we can see that the left image depicts the first part of a motion to goal step. EZ-RASSOR sees W_0 and the focus points F. The nodes of LTG(W_0) are marked. Since our focus points are part of the edges of our wedge, we need to continue to scan to see if we can obtain a focus point within our wedge indicating that we can see the end of the obstacle. Since W_1, in the right image, finds a focus point within our wedges we can stop scanning. This is because formulaically, F∈int(W0 ∪W1), so motion to goal ends. The idea behind this scanning is that it is quicker and requires less traversal to scan then to traverse and recalculate.

An example of how EZ-RASSOR would use border following can be seen below.

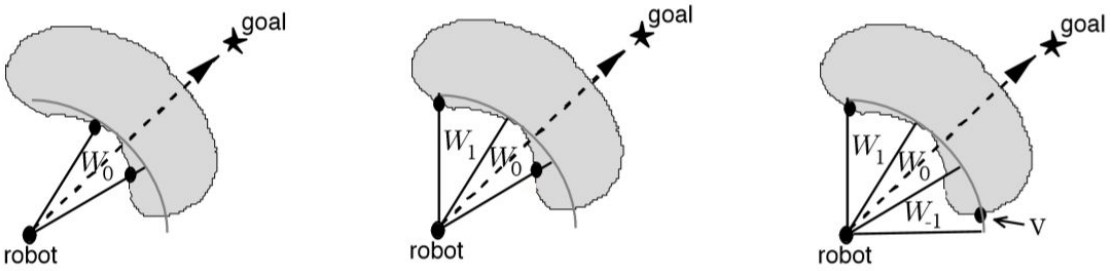


Figure 74: Example of virtual border following. [89]

The example on the left depicts the first part of a border following step. The nodes of LTG(W_0) are marked with black circles. Since there does not exist a V∈ int(LTG(W0)) (this simply states that there is not a vertex found in the interior of our wedge from the local tangent graph of wedge 0) , the robot scans W1 (center). Again, since there does not exist a V ∈int(LTG($W_0 \cup W_1$)), the robot

scans W−1 (right). Now, there exists a V ∈int($W_0 \cup W_1 \cup W_{-1}$), so the border following ends. This is because once again, we have found a vertex V that is not an element of the border of ($W_0 \cup W_1 \cup W_{-1}$). Wedge bug ensures that it is able to find a vertex that is in between its wedges borders if blocked by an obstacle. This ensures that EZ-RASSOR is able to find which path sees the end of the obstacle faster and leads to a more optimal solution. If our border following scans do not lead us to conclude a more optimal way around an obstacle, we must decide on a direction to go around an obstacle to determine if further progress can be made. If not, the algorithm will halt, if so, once we have moved around the obstacle, we will repeat this process until our final destination has been reached. Now that we have briefly discussed how the wedge bug algorithm works, we will discuss how we can implement it using ROS (robot operating system).

Fortunately, in ROS there are a lot of preexisting libraries and classes that we can use to help us accomplish our goal of path planning. Since we are not dealing with a 2D environment and a more complex 3D one, we will not be able to simply look for obstacles from within a small toy like example like the ones shown above. In this case we will have to get a bit more technical and use laser scans from EZ-RASSOR's depth camera. This depth camera will be responsible for both hole detection and detection of regular upward facing obstacles. In EZ-RASSOR's flow of data, we expect (path planning) to receive a disparity or depth image from the obstacle detection team. From this depth image, which will contain many points, each with its respective distance from our EZ-RASSOR's current position, we will be able to pass this data into the depthimage_to_laserscan function in ROS. This will be very useful for our robots navigation as it will now be able to theoretically create these laser scans as we have seen in the examples above. From this laser scan, we will be able to find and use all other components that are required for the wedge bug algorithm to function. Note that this is only a figurative laser scan and not a physical one. In this function our depth image or wedge is converted into a representation of what a laser scan would look like at each step of the way. From this point, it's simply a matter of fine tuning the laser scan to work with inclines and declines. Thankfully, we were instructed that we will not have to worry about traversing very steep terrains (angles greater than 60 degrees) as the swarm team will give us waypoints in a somewhat safe manner. However, we still have to account for the possibility of skidding down an incline/decline. Our solution to this problem is to

simply move at a slower linear and angular velocity to reduce wheel skid and other possible collisions. We also fixed the preexisting twist message implementation that the previous team created, as their commands always moved EZ-RASSOR at full linear and angular velocity. We implemented a ramping function that will gradually increase EZ-RASSOR's linear and angular velocities as it starts moving and gradually decreases these velocities as we approach our destination. This ramping function allows EZ-RASSOR to smoothly transition between movements and greatly reduces the risk of wheel skid and collision.

Once we have our laser scan, we can perform motion to goal and border following as needed per the wedge bug algorithm until we reach our goal. At each step, once we have found the most locally optimal and safe point, we will pass a twist message to the motion controls. This twist message consists of a vector which indicates the x,y,z direction and distance from EZ-RASSOR's current position and an angle theta representing the number of degrees that we must turn to in order to face our next stop. We must also account for a small distance ε from the obstacle to account for the width/length of EZ-RASSOR's large protruding drums to prevent it from colliding when moving forwards and to account for our robot turning when planning the next move.

Another thing that we must account for when traversing the unknown is the possibility of moving objects/obstacles. This can be anything from falling rocks to other EZ-RASSOR's traversing a similar area. The only time that the wedge bug algorithm does not account for moving objects or its environment to change is during the motion to goal phase as it is moving towards its target location. To help EZ-RASSOR account for some change along its path while moving, we will likely check our depth image to see if we are too close to anything as we are moving. Since we know the rate that we will be traveling at and the distances of the objects in front of us, if we sense any abrupt changes in the distances of objects in the foreground, we will immediately stop and recalculate a new path. But what if we run into multiple EZ-RASSOR's or other changes in the environment as we move? In this case, we will look for guidance from nautical navigation's rules for the road. This protocol will be discussed in greater detail in the obstacle detection and avoidance sections but will essentially cover how EZ-RASSOR will handle encounters with other robots and moving obstacles. As of right now, this will be considered a stretch goal for our team.

Now that we have discussed how EZ-RASSOR will navigate through its unknown environment, the only thing left to discuss is how we will decide on where we want to go. This question and many other questions regarding swarm tactics have been assigned to the EZ-RASSOR gold team. We have been informed that the gold team will be the mastermind behind assigning waypoints to each EZ-RASSOR robot. But what if our waypoint is an obstacle? For example, what if some of our EZ-RASSOR robots are given a waypoint that leads them into a large digsite or rock that can be misinterpreted as an obstacle? In this case, our robots will be confused in the sense that they will always be trying to avoid the current obstacles that just happen to be the goal. In this case, we have been informed that the gold team will not only assign us a waypoint but a threshold value that will allow EZ-RASSOR to easily identify when it is in the range of the waypoint. Once we enter this threshold, which can be thought of as a circle around the obstacle letting EZ-RASSOR know when it should switch its navigation mode, we will check if each obstacle in our way is the waypoint. If so, then we have reached our goal and then whatever objective EZ-RASSOR has been sent to do, it will switch into that mode. Some examples of modes that EZ-RASSOR could have are: dig mode, repair mode and threshold mode. The addition of these modes will be considered a stretch goal for our team.

Development Timeline

Fall 2019

- Research path planning algorithms — December 5, 2019
- Research the bug algorithms — December 5, 2019
- Initial design of navigation approach — December 14, 2019
- Decide on path planning algorithm — December 14, 2019
- Decide on path planning algorithm implementation — December 14, 2019
- Complete documentation — December 4, 2019

Spring 2020

- Implement path planning algorithm — March 1, 2020
- Test algorithms and protocols — March 29,2020
- Fine tune algorithm — March 29,2020

Twist Message Support

After completing the research phase and starting the design phase of adding support for Twist messages, we noticed that the EZ-RASSOR code had recently been modified to support Twist messages. It turned out that some members of the previous Senior Design team had gone back and added this functionality in October (after we determined our project requirements).

The main functionality we added in regards to Twist messages is that we implemented ramping (based on the sine function on the interval 0 to π) in linear and angular velocity. Previously, only a single speed was used when the robot was moving. Ramping linear and angular velocity resulted in much smoother driving, especially in the simulated moon environment where there was high wheel slippage and low gravity.

Testing & Evaluation

Obstacle Detection

The process of obstacle detection was tested in the Gazebo environment using a simulated Kinect sensor [90]. If the algorithms work in the simulation environment, then the concepts are proven to work.

To increase the criticality of the tests, multiple Gazebo worlds are used to check for different aspects of the expected results. One such world is the Hole world, which has a simple, rectangular hole and a cube next to each other—shown in Figure 75. This world is used to quickly and easily tell if the produced laser scan is detecting both negative and positive obstacles and if the location of the detections are accurate. Another environment with profound impact on the system is a simulated moon world, based on real topography data of the moon. This world helps test whether the obstacle detection algorithm can handle dynamic gradients in the ground around it. Also, the craters on the moon have ridges around them, which tests differently than a hole which is flat on the ground. Additionally, the above-ground obstacles on the moon have gradual and inconsistent inclines on them, unlike the cube.

The first step in evaluating the tests is to confirm the collection of a point cloud. Then, tests show whether edge detection works on the point cloud. The product of the obstacle detection subsystem should be a visually accurate laser scan. The laser scan is viewed using the data display tool RVIZ. Multiple laser scans are published at the same time, displaying the objects detected by the *hike*, *slope*, and minimum values of both. Each laser scan is colored differently, for easier distinguishing. Additionally, each laser scan is individually turned on and off in several combinations, in order to get the best idea of their accuracy. A visual comparison is conducted between the Gazebo objects and the laser scans in RVIZ to determine the accuracy of the applied obstacle detection algorithm.

Odometry

In order to test odometry, we measured the accuracy of the odometry system at various points along a path in a simulated moon environment. The environment was created to accurately represent a typical path a rover might need to take to a dig site. In addition to the high wheel slippage induced by the ground and the low gravity meant to simulate an actual moon environment, this testing environment had uneven terrain for the rover to traverse and a large boulder and a crater, both of which the rover had to maneuver around to reach its destination. The overall distance to the destination was approximately 50 meters.

To collect data on the error of the odometry system's predictions for the rover's location, we set up a ROS node specifically for testing that subscribed to both the rover's actual location (provided via the simulation environment) and odometry's predicted location. The node tracked the following values regarding the rover's position: the starting X coordinate x_{start} and Y coordinate y_{start} (these only needed to be collected once at the beginning of the test), the actual current X coordinate x_{actual} and Y coordinate y_{actual}, and the odometry system's current prediction for the X coordinate $x_{predicted}$ and Y coordinate $y_{predicted}$. We then computed the following based on this data: for every meter traveled (with distance traveled being calculated as $\sqrt{(x_{actual} - x_{start})^2 + (y_{actual} - y_{start})^2}$), the accumulated error is calculated as $\sqrt{(x_{actual} - x_{predicted})^2 + (y_{actual} - y_{predicted})^2}$. This allowed us to represent the accumulated error as a function of distance traveled, which gives us an estimate of the drift error of the odometry system.

During our performance tests, we performed the above calculations for both the combined odometry system and the wheel odometry system during the same trial. We did not include visual odometry in our tests because the visual odometry system gets lost when it can't find any features to track in the environment, and it needs another localization system present for it to use as a ground truth. This was not a problem when it was used as one component in the combined odometry system (it would use the output of the combined odometry system as the ground truth when it was lost), but it's an issue when attempting to test the accuracy of the visual odometry system in isolation.

Absolute Localization

For testing cosmic GPS and Park Ranger, we used a combination of simulated environments and the real world. For cosmic GPS the system was tested by performing absolute localization on earth to determine how accurate the system is and then infer how it would perform on the moon. This is best, as in Gazebo, it would be difficult to simulate the movement of stars in the sky. For Park Ranger the opposite is true. As park ranger involves matching terrain features over large ranges to a digital elevation map, it would be very difficult for us to accurately simulate the moon environment over this range in real life. Therefore to accurately test park ranger we used Gazebo as we already have several moon maps based on real digital elevation data to test with. With these maps we tested the accuracy of the method and reported on possible situations where the method excels or performed poorly.

Path Planning

For testing and evaluation of path planning our approach was to test the wedge bug algorithm in a simulated virtual environment until it was able to consistently navigate short to medium range distances. When we assessed how well the wedge bug algorithm worked, we tested the following:

- Can EZ-RASSOR avoid small rocks
- Can EZ-RASSOR avoid small craters
- Can EZ-RASSOR avoid steep/non-steep inclines/declines
- Can EZ-RASSOR avoid all of the above while moving towards the goal

For each case, we tested EZ-RASSOR's navigation skills in both a virtual moon environment and a flat obstacle course world using gazebo.

Performance & Results

In total, the EZ-RASSOR 2.0 Black Team's project is a success. The odometry system is accurate enough that the rover's predicted location stays well within our requirement of 25 meters on all tests we ran. The Cosmic GPS surpassed our expectations and did so in real-world tests. Park Ranger was implemented successfully, but the accuracy and consistency of the predicted location was limited by the height and field of view of the onboard camera as well as other factors.

The secondary facet of the project included traveling to a location without harming the robot. In simulation, the Mini-RASSOR is able to near-optimally travel to a destination using minimal information of its surroundings and avoids dangerous obstacles successfully.

Below are more in-depth descriptions of each subsystem's performance at the completion of this project.

Obstacle Detection

In general, the algorithm is successful in detecting both above-ground obstacles and holes where the back edge can be seen by the depth camera.

The laser scan produced by the obstacle detection subsystem is acceptable for the purpose of finding a safe path if the configurable thresholds are set appropriately. There is a balance of making the thresholds for the size of holes found larger that makes less of the hole be detected, but too small would make the robot scared of traveling almost anywhere. Making a bad threshold results in the robot going nowhere in many situations or falling into sizable craters. A sweet spot has successfully been found to be reliable in conducted tests.

Holes are not able to be found after a certain distance away from the camera, but this is acceptable because the Path Planning subsystem only looks for obstacles at a distance shorter than hole-detection fails. There are also times where small

gaps of false negatives are in between true positives; but, this is okay because the Path Planning subsystem will only allow the robot to travel between large gaps of non-obstacle space. Obstacles could be better detected if the camera was higher, so it could see over small hills.

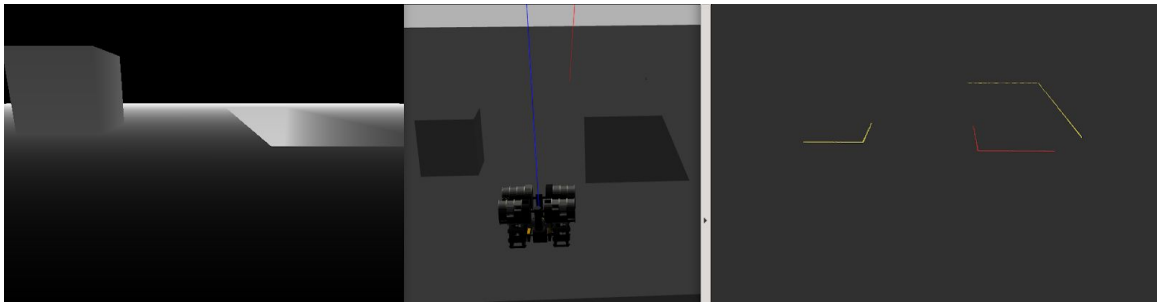


Figure 75: Depth image, Gazebo simulation, and RVIZ laser scans

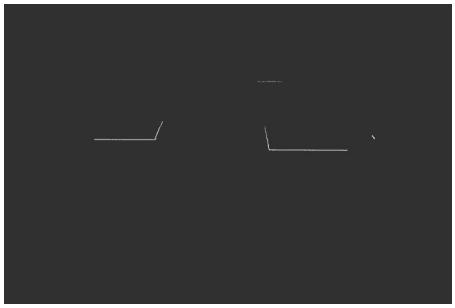


Figure 76: RVIZ combined laser scan

Figure 75 shows how obstacle detection performs in the Hole world in Gazebo. There is a cube to the left and a rectangular hole to the right. The middle image shows the Gazebo simulation, which is considered the ground truth. The left image shows the depth image that the stereo camera is able to produce and matches the data in the point cloud. The image on the right is the two laser scans produced by thresholding the points' *hike* and *slope* in a step-group. The red lines are detections of a hole, from the *hike*. The yellow lines are the detection of an above-ground or positive obstacle, from the *slope*. Figure 76 is the product of taking the minimum distance in each step-group between the laser scans produced by the *hike* and *slope*. The line going up the right side of the hole stops short because Path Planning does not look any further than where it stops, and the *hike* comparison stops being admissible after a point.

Odometry

Our method for odometry was highly effective, especially given the constraints of the environment. Each of the odometry methods had their own challenges: for wheel odometry, there was high slippage on the surface the rover traveled on and many slopes that the rover could slide down; for visual odometry, the camera had a limited field-of-view and there were essentially no visual features in the environment (since the surface was all the same color); IMU odometry in general is highly inaccurate by itself. However, by combining all of these odometry methods using an extended Kalman filter, we were able to achieve much higher accuracy than any single odometry method could have achieved on its own.

The results of our performance tests are shown below. The combined odometry system has slight error (approximately 0.16 meters) even before the rover has started moving, but the error only grows at a rate of approximately 0.01 meters per meter traveled. This means that, given a known starting location, the rover could likely travel hundreds of meters (possibly even a few kilometers, but we're not able to run the simulation for that long or test an environment that large on our personal computers) before our odometry system's accumulated error surpasses 25 meters (the requirement we set at the beginning of the project for the localization components). The error of the wheel odometry system in isolation grew at a rate of approximately 0.69 meters per meter traveled, showing that it was worth it to combine odometry methods using an extended Kalman filter.

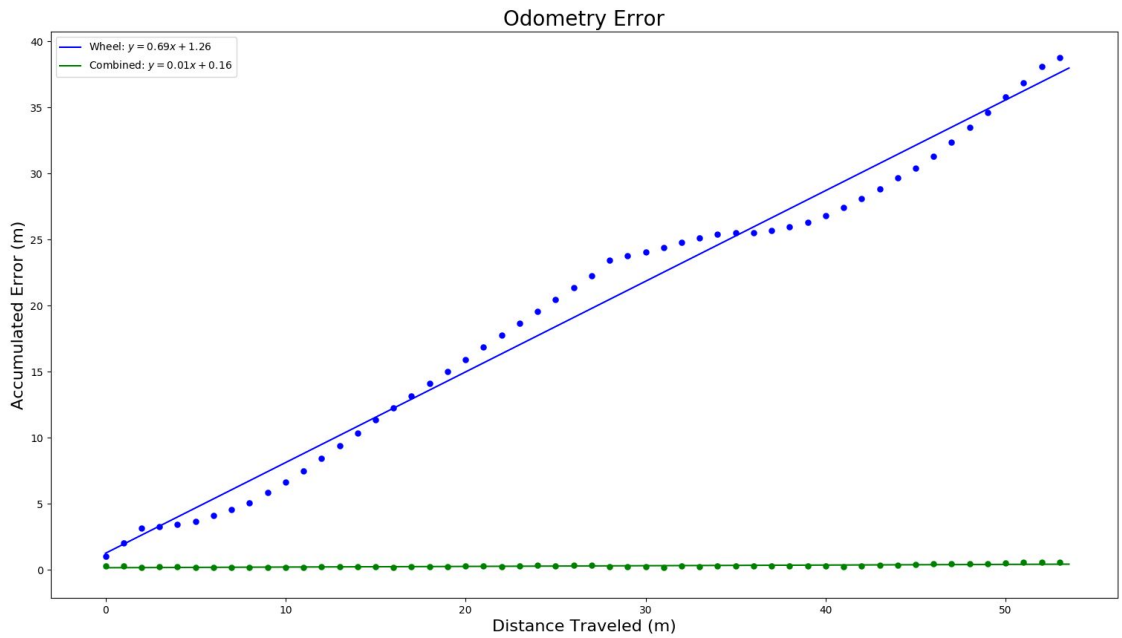


Figure 77: Accumulated odometry error as a function of distance traveled

Absolute Localization

Cosmic GPS

Our method for determining a geographic position based on the positions of celestial bodies in an image was successful. As shown below in Figure 78 we were able to determine a position about 47 miles from our actual position. Although 47 miles sounds like a large distance, it should be noted that the method works on a global scale and is highly sensitive to small errors. The two major sources of errors being deviation of the front of the camera from the zenith and the lens distortion. The first error (the greater of the two) offsets the true position by the magnitude of the deviation. In Figure 78, the green tag marks the location where the image was taken and the red tags mark different estimated positions based on different combinations of stars used. Most of the error was likely due to this deviation as a roughly 1 degree latitude and longitude difference from the zenith results in a change of 60 miles. Note, when testing the camera, it was placed on a tripod and faced towards the zenith as carefully as possible, something that is very difficult without precision equipment. The second error causes incorrect angle measurements in the image from the center (the zenith)

to individual stars. This results in variations in the determined position depending on which stars are selected. Thus, considering the importance of the hardware in governing the accuracy of the cosmic GPS, the results are decent.

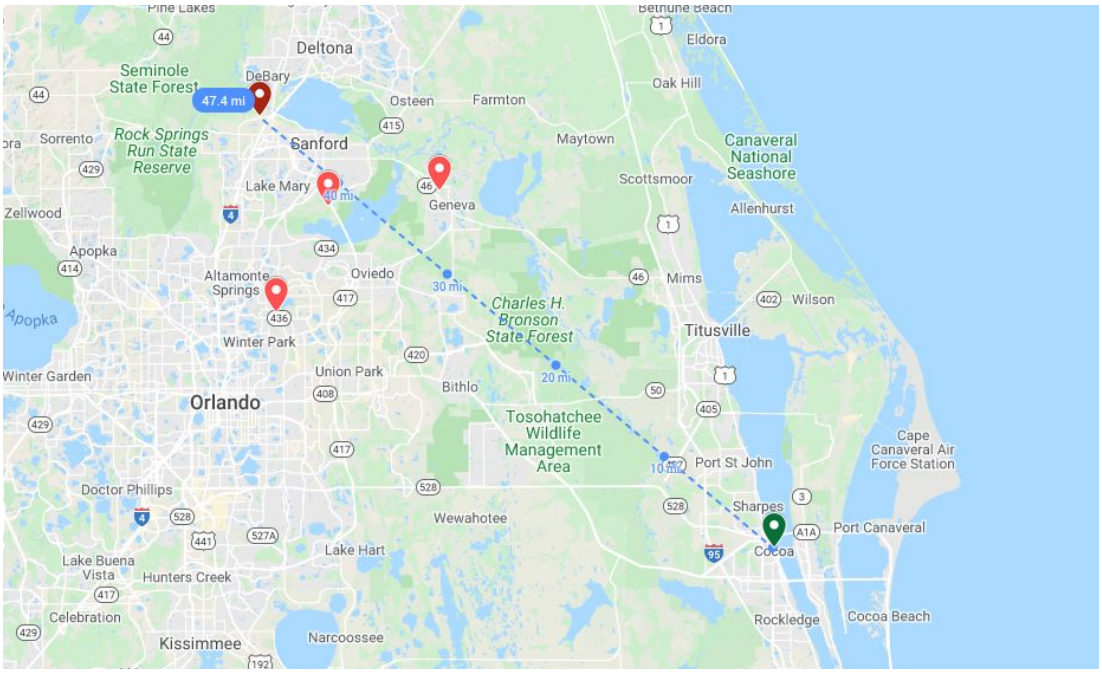


Figure 78: Actual and estimated locations, using Cosmic GPS

Park Ranger

Although a converging estimate was successfully derived, the accuracy and consistency of the estimate was less than ideal. The main factors that could be affecting the accuracy are: physical camera constraints, heightmap constraints, and simulation constraints.

The physical camera constraints adversely affect several components of EZ-RASSOR. The depth camera the Mini-RASSOR uses has a horizontal field of view of 74 degrees and is positioned low to the ground due to the rover's relatively small frame. This is likely the major contributor to the inaccuracy of Park Ranger because it bounds the number of usable points generated in the local DEM.

The heightmap constraints refer to the factors in creating the realistic moon terrain as a heightmap object in Gazebo. To create a heightmap, the choice of rasters (i.e. .jpg, .png, .tif) can affect the appearance and position of the terrain

object. The ratio of the raster dimensions to the range of elevation values can also affect the terrain.

The last factor that could be affecting the accuracy of the estimate is having to test in a simulated environment. There is a noticeable latency between estimate measurements due to computation power being dedicated to rendering the simulation.

The major ways these adverse effects could be reduced are: placing the camera higher up, increasing the field of view by adding more cameras or using a different camera, and testing Park Ranger outside of simulation.

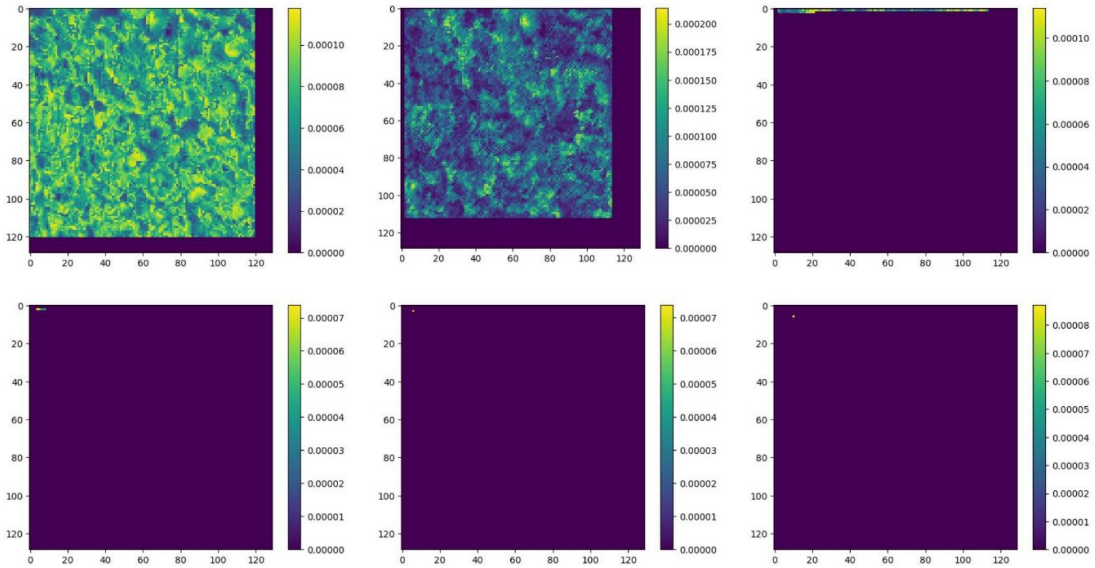


Figure 79: For five iterations of Park Ranger, this shows the weight value for each position.

This shows that the particles start converging quite fast after even three iterations, sometimes on a position that is in a different quadrant than the actual position of the rover. Although the accuracy and consistency of the estimate could be improved through the previously mentioned methods, the Park Ranger component could be scoped itself as an independent long-term research project. Implementing this type of localization in a moon-like environment and on a robot with such a limited field of view as well as a low camera height, is a relatively novel challenge.

Path Planning

In general, the wedge bug algorithm is successfully able to avoid small rocks and craters and large obstacles with steep slopes in a simulated environment. Having only 74 degrees of view and using a low to the ground sensor have been a challenge. Despite these limitations, EZ-RASSOR is still able to successfully navigate autonomously using the wedge bug algorithm. As previously mentioned, we have implemented a twist message ramping function that allows for smoother transitions in linear and angular movement. This ramping function greatly reduced the risk of wheel skid, collision with obstacles and was very successful.

When it comes to how optimized EZ-RASSOR's path planning routes are, they are pretty efficient considering our limitations (74 degree view and low to the ground sensors). We realized early on that the safety of EZ-RASSOR is more important than taking a faster route and thus shifted our focus primarily towards safety. This is because when we would try to cut corners or fit in between a narrow passage, we would find that more likely than not, EZ-RASSOR's large upper drums would collide with obstacles. This has a lot to do with the fact that we can only see 74 degrees wide in front of us. As a result, we increased the buffer size (the amount of clearance that we need to fit through obstacles) which prioritised EZ-RASSOR's safety but decreased path optimality. It was concluded by our team and sponsors that choosing the most optimal path was not as big a concern as obstacle avoidance and thus we needed to shift our focus and prior implementation.

Another thing to note, when working with the wedge bug algorithm and floating point numbers, it was very difficult to determine after turning if we have already looked at a certain point or not. This is because even the slightest change in view or movement would change a laser scan and its data. In our earlier implementations of wedge bug, we attempted to turn towards areas of uncertainty to check if they were safe and then turn back. It was near impossible to turn back to the exact point since there are so many margins of error and doing this multiple times would only lead to a greater propagation of uncertainty. We later concluded that we would only do the computation in a wedge while viewing it and always find the best point at hand that we can see. If we can't see something in our current wedge don't worry about it, just find the best path at each step of the way as this reduces the need to constantly turn back and forth. What we found in doing this was a much less complex algorithm and one that

worked better and was more accurate. Once this was completed, all we had to do was update our current laser scan to the most up-to-date obstacle detection laser scan that also detects holes. Once this was completed, we found that EZ-RASSOR was able to successfully navigate to distances of up to 50 meters where it encountered rocks and craters.

Facilities & Equipment

In order to discuss our requirements and see the actual Mini-RASSOR and RASSOR rovers, we met with the Swamp Works team at the Swamp Works facility in Kennedy Space Center on October 18th. There, we met our mentor, Kurt Leucht, the software lead for the Swamp Works team. After gaining further background on the RASSOR and EZ-RASSOR projects, we discussed our requirements with Kurt. We then took a tour around the facility, where we were able to see the test bin that the team uses to test the RASSOR and other projects (shown below). This 625 square foot bin is full of dirt that resembles lunar regolith. Interestingly, this regolith-like material was found by chance. NASA was originally producing regolith-like material itself based on samples of real lunar regolith. However, this was an extremely expensive process, and it would not have been feasible to manufacture enough of the material to fill up such a large test bin. Later, though, an astronaut visiting a testing site in Arizona noticed that the dirt at the site, which was waste from a previous project, was very similar to lunar regolith. After testing samples of the material, NASA confirmed that the dirt from this site was actually more similar to lunar regolith than what they were creating previously. NASA was then able to buy this dirt at a low price, and this dirt is what now fills the large testing bin in the Swamp Works facility.

Figure 80: A view of the SwampWorks facility. The large, enclosed box is a test bin whose surface mimics the moon's. [91]

Budget & Financing

Software

We do not have any software costs, as we are using only free and open-source software and tested the project on our own machines.

Hardware

An upward-facing camera is needed to take pictures of the stars in the sky. This camera is used for testing the development of the Cosmic GPS. The camera is the only cost of the project and was financed by a single team member, who owns the camera upon the project's completion.

Item	Purpose	Cost
Wide-View Camera	Upward-facing view of the sky/horizon	$55
Total		$55

Figure 81: The cost of the hardware needed to develop GPS-denied navigation

Milestones

The tasks related to the development of the project are summarized in the Gantt chart below.

Figure 82: Task Gantt Chart.

Future Works

Though this project has been successful, there are several improvements that have been discussed by the team which would improve the overall product. These ideas could not be completed in the period worked on by this team, nor were they a part of the initial requirements. Knowing that NASA and FSI would like to continue with future iterations of UCF Senior Design teams working on this project, we believe that these are good launching points for discussions to set requirements for future EZ-RASSOR projects.

Mini-RASSOR Integration

Although our software works well with the simulated Mini-RASSOR that we've tested on, the EZ-RASSOR software package hasn't been run on the physical Mini-RASSOR yet. Depending on the state of the Mini-RASSOR when a future team works on this integration, it may be useful to have an interdisciplinary team made up of Computer Science and Electrical/Computer Engineering students. The ECE students could work on sensor integration and other aspects of the hardware, while the CS students could work on processing the sensor data (we've tried to emulate as much as possible with our simulated sensors what the actual sensor data would look like, but there could be some differences) and making any changes to the algorithms necessary based on real-world tests.

SEE Integration

The Simulation Exploration Experience (SEE) project provides university teams with the challenge of developing a simulated lunar mission. The RASSOR is designed to help in future lunar missions, so it would be useful for teams participating in this challenge to have access to a simulated Mini-RASSOR running the EZ-RASSOR software package.

Mission-Specific Behavior

Currently, the path planning subsystem assumes that the destination it is trying to reach is a clear space. That means that obstacle detection is being done all the way until the destination is reached. However, it is likely that there will be instances where the destination would itself be considered an obstacle. Consider the circumstances where the robot may be traveling to a charging station above

the ground, or a dig site which is a hole. This means that it could be necessary to completely or partially turn off obstacle detection either for the entirety of the path traveled or just when the robot is within a certain distance from the destination. This would likely require cooperation between the path planner for a robot and the central system providing its commands.

Rules of the Road

Currently EZ-RASSOR is able to navigate in both simple and complex environments. However, EZ-RASSOR has not had to worry too much about moving obstacles like astronauts, falling objects or other EZ-RASSOR's in its environment. In future scenarios, we expect there to be swams of EZ-RASSOR robots all navigating the same terrain, some with different objectives. This process will lead to some robots having overlapping paths or just being in close proximity with each other. Knowing how hectic that it can get on Earth with traffic, it makes sense that we should consider implementing some rules of the road. This could be in the form of a simple protocol that all EZ-RASSOR robots follow, which will allow them to continue their objective while staying safe and avoiding collisions with other EZ-RASSOR robots and moving obstacles.

Currently, EZ-RASSOR is able to check for movements in its environment and can tell when it gets too close to other objects. Using this sense of its surroundings and the possibility of color coding different parts of EZ-RASSOR (for example, red on the back, green on the front, yellow on the left, and blue on the right), we can determine which direction an EZ-RASSOR robot is moving based on the color that it sees. If two EZ-RASSOR robots are traveling head on, both robots see the color green, and they would both know that they should keep to their right when traveling forward. If two EZ-RASSOR robots are traveling in a single file line, the back robot would see the color red, and would know that it should keep a safe distance behind the other one. If colors are not an option, this idea could simply be extended to using QR codes, where each code is placed on a unique side and tells the other robots what its objective is and where it's currently trying to go; granted, this would require connection to a database. These processes can be extended to various scenarios and would allow for better navigation with many autonomous robots all traveling at the same time.

Continuous Path Planning

The way the EZ-RASSOR approaches path planning is by moving a small, configurable distance, then stopping when it either travels this distance or encounters an obstacle in its way. Once the rover has stopped, it searches for the most optimal path that avoids any obstacles it can see. Potentially, the robot could move continuously while considering new paths, so that it only stops when necessary to avoid running into an obstacle. This would reduce the time it takes the rover to reach a destination. The obstacle detection component already outputs the detected obstacles at a fairly high rate, so this should only affect the path planning component.

Cosmic GPS Hardware and Integration

The Cosmic GPS has not been integrated with the EZ-RASSOR platform and is without a viable hardware platform. The next two steps regarding the Cosmic GPS is to first build a camera system that can adequately capture images of stars without false positives and that naturally points towards the rover's zenith. This camera would allow for easier and more accurate testing. The second step is to build a virtual model of the camera system and integrate the current software into the EZ-RASSOR platform. Note that in order for the Cosmic GPS to be useful in either Gazebo or SEE, the simulation will need to be modified to accurately simulate the night sky.

Update Mini-RASSOR Simulation Model

The Mini-RASSOR simulation model was created during the development of EZ-RASSOR 1.0. We changed some of the simulated sensors and modified their placement on the model, but the model itself has remained the same. While the model has worked great for testing our project, it was created while the physical Mini-RASSOR was still being designed at SwampWorks. Thus, the dimensions of the model are likely off. One task for a future EZ-RASSOR team could be to work closely with the engineers at SwampWorks to make the simulated Mini-RASSOR model the real Mini-RASSOR as closely as possible.

Conclusion

For the past academic year, this team has faced countless issues that we were able to overcome through perseverance and intelligence. We originally had to define the difference in roles between the Black and Gold teams and that led to a small shakeup in rosters. We started our requirements with a focus on localization, which naturally led to adding path planning and then obstacle detection. After a few small shifts in what exactly was expected of the team to complete, we were able to devise a plan to provide everyone on the team with a leadership role and multiple opportunities to work on unique aspects of the project. This division in labor was designed by keeping in mind the experiences that each member has had and what they wanted to take away from this project. The diversity in experience and yearning to learn was a major factor in the success of this project. The members of this team all had some combination of experience in robotics, computer vision, algorithm design, and academic research; however, nobody possessed great knowledge in all of these fields.

The Black team, led by Shelby Basco, was able to use a multitude of technologies to assist the development processes along the year. Regularly scheduled meetings were coordinated with Mike Conroy and Kurt Leucht to discuss progress, current hurdles, and the direction of the project. Our team met several times a week to plan and work on the implementation of our algorithms. We were able to effectively plan tasks using Trello during both semesters. A shared Discord server with the Gold team with private channels and roles made communication easy and effective. Undoubtedly, one of the largest hurdles that every senior design team this year faced was the sudden, global reaction to the COVID-19 pandemic. Nevertheless, our team was quick to adapt and overcome, using the technologies available to us. Meeting rooms in Zoom and their ability to share control of a screen, made collaboration almost as easy as working side by side. Luckily, one member had a strong enough computer to run the simulation environment while streaming their screen to the rest of the team.

The Absolute Localization team, led by Scott Scalera, accomplished using a view of the stars to determine an approximate location for the rover on Earth. Furthermore, the Park Ranger subsystem can use orbital data to match the surrounding environment and produce a location estimate.

The Odometry team, led by John Albury, accomplished combining data from the IMU, wheels, and depth camera to form an estimate for the magnitude and direction of the movement of the robot. When attached to a known starting point, this results in another approximate location for the rover.

The Obstacle Detection team, led by John Hacker, accomplished utilizing the point cloud produced by the depth camera to create a laser scan of the closest obstacles in front of the robot. The laser scan detects both above-ground obstacles and below-ground holes.

The Path Planning team, led by Michael Jimenez, accomplished moving the rover to a given destination on a near-optimal route. A modified version of the wedge bug algorithm is used to avoid needing a map of the moon. The rover will always aim to move straight towards the destination if it can, but if there are obstacles in the way, the rover will move around them to minimize the stray from trajectory while prioritizing safety.

This project has been a profoundly positive experience for everyone on the EZ-RASSOR 2.0 Black Team. The RASSOR in an amazing achievement of engineering, and we are all proud to have contributed to its narrative. The EZ-RASSOR is planned to be a longstanding, extensible project for both Senior Design projects and researchers. This team thanks FSI and NASA for allowing us this opportunity to contribute to such a meaningful project and develop our skills as software engineers.

References

[1] S. Dick et al. “Apollo 11 Image Gallery”, 2007. https://history.nasa.gov/ap11-35ann/kippsphotos/apollo.html.

[2] F. Tavares. “Ice Confirmed at the Moon’s Poles”, 2018. https://www.nasa.gov/feature/ames/ice-confirmed-at-the-moon-s-poles.

[3] IronEqual. “Pathfinding Like A King - Part 1.”, In *Medium*, IronEqual, 8 Sept. 2017, medium.com/ironequal/pathfinding-like-a-king-part-1-3013ea2c099.

[4] S. Koenig and M. Likhachev. “D* Lite”, In *AAAI*, 2002.

[5] G.D. Hager and Z. Dodds. “Robotic Motion Planning: Bug Algorithms”. www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf.

[6] D. Fox et al. "Monte carlo localization: Efficient position estimation for mobile robots." In *AAAI,* 1999.

[7] T. Bailey and H. Durrant-Whyte. “Simultaneous Localization and Mapping: Part I”, In *IEEE Robotics and Automation Magazine*, (June 2006), pg. 99.

[8] “HC-SR04 Ultrasonic Range Sensor on the Raspberry Pi”. https://thepihut.com/blogs/raspberry-pi-tutorials/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi.

[9] “Distance Sensing”. https://www.sparkfun.com/distance_sensing.

[10] R. Burnett. “Ultrasonic vs Infrared (IR) Sensors -- Which is better?”, 2017. https://www.maxbotix.com/articles/ultrasonic-or-infrared-sensors.htm.

[11] D. Kohanbash. “LIDAR vs RADAR: A Detailed Comparison”, 2017. http://robotsforroboticists.com/lidar-vs-radar.

[12] “Autopilot | Tesla”. https://www.tesla.com/autopilot.

[13] “Upgrading Autopilot: Seeing the World in Radar”, 2016. https://www.tesla.com/blog/upgrading-autopilot-seeing-world-radar.

[14] L. Wasser. “The Basics of LiDAR - Light Detection and Ranging - Remote Sensing”. https://www.neonscience.org/lidar-basics.

[15] B. Howard. “Google: Self-driving cars in 3-5 years. Feds: Not so fast”, 2013. https://www.extremetech.com/extreme/147940-google-self-driving-cars-in-3-5-years-feds-not-so-fast.

[16] B. de Bakker. “How to use a SHARP GP2Y0A710K0F IR Distance Sensor with Arduino”.

https://www.makerguides.com/sharp-gp2y0a710k0f-ir-distance-sensor-arduino-tutorial.
[17] S. Leibson. “Fundamentals of Distance Measurement and Gesture Recognition Using ToF Sensors”, 2018. https://www.digikey.com/en/articles/techzone/2018/nov/fundamentals-distance-measurement-gesture-recognition-tof-sensors.
[18] “VCSELs and Distance Sensing”. https://www.sparkfun.com/news/2796.
[19] J. Meisner. “Collaboration, expertise produce enhanced sensing in Xbox One”, 2013. https://blogs.technet.microsoft.com/microsoft_blog/2013/10/02/collaboration-expertise-produce-enhanced-sensing-in-xbox-one.
[20] D. Cardinal. “Making gesture recognition work: Lessons from Microsoft Kinect and Leap”, 2013. https://www.extremetech.com/extreme/160162-making-gesture-recognition-work-lessons-from-microsoft-kinect-and-leap.
[21] I. S. Mohamed. “Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques”, 2017.
[22] R. Wang. “Difference of Gaussian (DoG)”, 2018.
[23] K. Adi and C. E. Widodo. “Distance Measurement with a Stereo Camera”, In *International Journal of Innovative Research in Advanced Engineering (IJIRAE)*, page 25, 2017.
[24] Muhammad Ghani and Ksm Sahari. “Detecting negative obstacle using Kinect sensor”, In International Journal of Advanced Robotic Systems, 2017.
[25] G. Lei et al. “Monocular Vision Distance Measurement Method Based on Dynamic Error Compensation”, In *International Journal of Digital Content Technology and its Applications (JDCTA)*, page 236, 2013.
[26] D. Han et al. “Real-time object segmentation using disparity map of stereo matching”, In *Applied Mathematics and Computation*, page 1, 2008.
[27] BogoToBogo. “Canny_Edge_Detection”, 2014.
[28] Z. Wang et al. “On-Tree Mango Fruit Size Estimation Using RGB-D Images”, In *MDPI*, page 1, 2017.
[29] N. Mansurov, “What is Lens Distortion?” https://photographylife.com/what-is-distortion.
[30] A. Wilt, “HPA Tech Retreat Day 4”, 23 Feb 2013. https://www.provideocoalition.com/hpa-tech-retreat-day-4/.

[31] S. Pravenaa and Dr. R. Menaka. “A Methodical Review on Image Stitching and Video Stitching Techniques”, In *International Journal of Applied Engineering Research*, Vol 11, No. 5. pp 3442-3448. 2016.
[32] S. Arya, “A Review on Image Stitching and its Different Methods”, In *International Journal of Advanced Research in Computer Science and Software Engineering*, Vol 5, Issue 5. May 2015.
[33] S. Lerner. “Implementing SIFT in Python”, 26 Dec. 2018. https://medium.com/@lerner98/implementing-sift-in-python-36c619df7945.
[34] “Panoramic Image Projections” https://www.cambridgeincolour.com/tutorials/image-projections.htm.
[35] H. Abdul-Ridha Mohammed and R. A. Kamil. “Applying the Concepts of Panorama in Virtual Reality Using the Map Principle on a Library Website”, In *International Journal of Engineering Research and Development.* Vol 11. Issue 3. Pp 09-14. Mar. 2015.
[36] S. Rohn. “Stereographic-Panorama-New-York-005”, 6 Feb. 2010. https://www.samrohn.com/little-planets/stereographic-panorama-new-york-005/.
[37] “NOISE REDUCTION using Fuzzy Filtering”.
[38] HAMAMATSU. “What is photon shot noise?”.
[39] Stack Exchange. “SEfEG”.
[40] MathWorks. “Perform Edge Preserving Smoothing”.
[41] R. Fisher et al. “Laplacian/Laplacian of Gaussian”, 2003.
[42] D. W. Fanning. “Image Sharpening with a Laplacian Kernel”, 2006.
[43] “Azimuthal Projection: Orthographic, Stereographic and Gnomonic” https://gisgeography.com/azimuthal-projection-orthographic-stereographic-gnomonic/.
[44] UVIC. “Image Segmentation”, page 13, 2009.
[45] K. Meethongjan and D. Mohamad. “Maximum Entropy-based Thresholding algorithm for Face image segmentation”, page 2.
[46] A. C. Sparavigna. “Tsallis Entropy”, In *Bi-level And Multi-level Image Thresholding. International Journal of Sciences*, figure 1, 2015.
[47] Senthilkumaran N. and Vaithegi S. “IMAGE SEGMENTATION BY USING THRESHOLDING TECHNIQUES FOR MEDICAL IMAGES”, In *Computer Science & Engineering: An International Journal (CSEIJ)*, page 5, 2016.
[48] M. V. Arbabmir et al. “Improved night sky star image processing algorithm for star sensors”, *J. Optical Soc. of Am. A*, page 704, 2014.

[49] M. Mancini. “What Are Days and Nights Like on the Moon?”, 28 Sep. 2018. https://science.howstuffworks.com/what-do-day-and-night-look-like-on-moon.htm.

[50] “Moon Phases Template”. https://www.edrawsoft.com/template-moon-phases.php.

[51] J. Enright, P. Furgale, and T. Barfoot. “Sun Sensing for Planetary Rover Navigation”, In *Proceedings of the IEEE Aerospace Conference, Big Sky, MT.* 2009. https://furgalep.github.io/sbib/enright_ieeeac09.pdf.

[52] C. Boirum. “Improving Localization of Planetary Rovers with Absolute Bearing by Continuously Tracking the Sun”, 7 Jan. 2015. https://www.ri.cmu.edu/wp-content/uploads/2017/03/Absolute-Sun-Bearing-Curtis-Boirum-RI-TR-Final.pdf.

[53] B. B. Spratling, IV and D. Mortari. “A Survey on Star Identification Algorithms”, In *Algorithms*, Vol 2. Pp 93-107. 2009.

[54] S. Ji, J. Wang, and X. Liu. “Constellation Detection” https://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Ji_Liu_Wang.pdf.

[55] HIPPARCOS Catalogue (cosmos.esa.int/web/hipparcos/catalogues) --- star positioning.

[56] B. Bitterli, ”Star Stacker: Astrophotography with C++11” https://benedikt-bitterli.me/astro/.

[57] J. A. Christian and E. Glenn Lightsey. ”Onboard Image-Processing Algorithms for a Spacecraft Optical Navigation Sensor System”, In *Journal of Spacecraft and Rockets*, Vol. 49, No. 2. Apr 2012. https://arc.aiaa.org/doi/abs/10.2514/1.A32065.

[58] J. Jiang, H. Wang, and G. Zhang. “High-Accuracy Synchronous Extraction Algorithm of Star and Celestial Body Features for Optical Sensor”, In *IEEE Sensors Journal*. Vol 18. Issue 2. 15 Jan. 2018. https://ieeexplore.ieee.org/document/8119847.

[59] M. Monmonier. “Cartography in the Twentieth Century”, Vol. 6. 2015. https://history.yale.edu/sites/default/files/files/2015%20rankin%20-%20GPS.pdf.

[60] “Trilateration vs Triangulation – How GPS Receivers Work”, 4 Mar 2019. https://gisgeography.com/trilateration-triangulation-gps/.

[61] V. Moore. “Celestial Navigation – Azimuth & Sight Reduction Tables”, 1 Nov 2016.

https://astrolabesailing.com/2016/11/01/celestial-navigation-azimuth-sight-reduction-tables/.

[62] X. Ning and J. Feng. “A new autonomous celestial navigation method for the lunar rover”, In *Journal of Robotics and Autonomous Systems*, Vol. 57, Issue 1. 31 Jan 2009, pp 48-54.

[63] “Finding Latitude and Longitude” https://astro.unl.edu/naap/motion1/tc_finding.html.

[64] T. Senlet and A. Elgammal. “A Framework for Global Vehicle Localization Using Stereo Images and Road Maps”, In *IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, Nov 2011. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.592.1629&rep=rep1&type=pdf.

[65] X. Wang, S. Vozar, and E. Olson. “FLAG: Feature-based Localization between Air and Ground”, In *IEEE International Conference on Robotics and Automation (ICRA)*, Jun 2017. https://april.eecs.umich.edu/pdfs/wang2017icra.pdf.

[66] A. Viswanathan et al. “Vision based robot localization by ground to satellite matching in GPS-denied situations”, In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014. https://ieeexplore.ieee.org/document/6942560.

[67] M. Lourakis and E. Hourdakis. “Planetary Rover Absolute Localization by Combining Visual Odometry with Orbital Image Measurements”, 2015.

[68] M. R. U. Saputra et al. “Visual SLAM and Structure from Motion in Dynamic Environments: A Survey”, In *ACM Comput. Surv.* Article 51, 2, Article 37, Feb 2018. https://www.cs.ox.ac.uk/files/9926/Visual%20Slam.pdf.

[69] J. Woo, K. Son, et al. “Vision-based UAV Navigation in Mountain Area”, In *MVA2007 IAPR Conference on Machine Vision Applications,* May 2018.

[70] A. V. Nefian, X. Bouyssounouse, et al. “Planetary rover localization within orbital maps”, In *IEEE International Conference on Image Processing (ICIP)*, 2014. https://ieeexplore.ieee.org/document/7025326.

[71] G. Baatz et al. “Large Scale Visual Geo-Localization of Images in Mountainous Terrain”, In *European Conference on Computer Vision*, pages 517-530, 2012. https://cvg.ethz.ch/research/mountain-localization/BaatzECCV12.pdf.

[72] P. C. Naval Jr. et al. ”Estimating Camera Position and Orientation from Geographical Map and Mountain Image”, In *38th Research Meeting of the*

Pattern Sensing Group, Society of Instrument and Control Engineers, 1997.

[73] A. Florian et al. “asr_mild_base_driving”. http://wiki.ros.org/asr_mild_base_driving.

[74] “Inertial Measurement Unit BMI055 | Bosch Sensortec”. https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi055.html.

[75] M. O. A. Agel et al. “Review of visual odometry: types, approaches, challenges, and applications”, In *SpringerPlus*, 2016. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5084145/.

[76] D. M. Helmick and Y. Cheng. “Path Following using Visual Odometry for a Mars Rover in High-Slip Environments”, In *IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720)*, 2004. https://trs.jpl.nasa.gov/bitstream/handle/2014/38778/03-2866.pdf?sequence=1.

[77] M. Maimone, Y. Cheng, and L. Matthies. “Two Years of Visual Odometry on the Mars Exploration Rovers”, In *Journal of Field Robotics*. 2007. https://www-robotics.jpl.nasa.gov/publications/Mark_Maimone/rob-06-0081.R4.pdf.

[78] M. Labbe. “rtabmap - ROS Wiki”. http://wiki.ros.org/rtabmap.

[79] M. Labbe and F. Michaud. “RTAB-Map as an Open-Source Lidar and Visual SLAM Library for Large-Scale and Long-Term Online Operation”, In *Journal of Field Robotics*, pages 8-10, 2019. https://introlab.3it.usherbrooke.ca/mediawiki-introlab/images/7/7a/Labbe18JFR_preprint.pdf.

[80] S. Wirth. “fovis - ROS Wiki”. http://wiki.ros.org/fovis.

[81] A. Haung et al. “Visual Odometry and Mapping for Autonomous Flight Using an RGB-D Camera”, In *Proceedings, International Symposium on Robotics Research (ISRR)*, pages 5-7, 2011. https://people.csail.mit.edu/albert/pubs/2011-huang-isrr.pdf.

[82] “robot_localization wiki”. http://docs.ros.org/melodic/api/robot_localization/html/index.html.

[83] S. Rapp et al. “NASA EZ-RASSOR UCF Fall 2018 Team”, 2019.

[84] Intel. “Intel® RealSense™ Depth Camera D435i”, 2019.

[85] “GDAL”. https://gdal.org/

[86] B. Cane. “Do Not Run Dockerized Applications as Root”, American Express.io. 27 Sep 2018. https://americanexpress.io/do-not-run-dockerized-applications-as-root/

[87] “Manage data in Docker”. https://docs.docker.com/storage/

[88] B. van Pham, A. Maligo, and S. Lacroix, “Absolute Map-Based Localization for a Planetary Rover,” *12th Sympusium on Advanced Space Technologies and Automation in Robotics*, May 2013.

[89] S. L. Laubach and J. W. Burdick. “An Autonomous Sensor-Based Path-Planner for Planetary Microrovers”, In *Proceedings of the IEEE International Conference on Robotics and Automation*, May 1999.

[90] Gazebo. “ROS Depth Camera Integration”, 2014.

[91] S. Siceloff. “University Researchers Test Prototype Spacesuits at Kennedy”, 2015. https://www.nasa.gov/feature/university-researchers-test-prototype-spacesuits-at-kennedy.