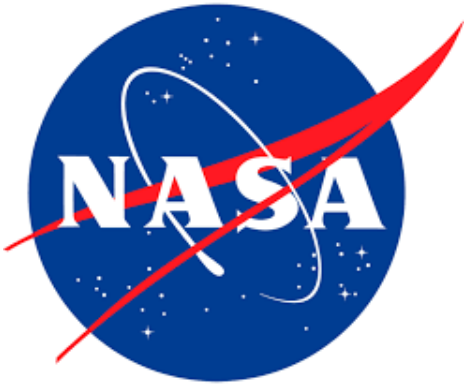


NASA EZ-RASSOR
UCF Fall 2018 Team

Group Members:
Sean Rapp
Tiger Sachse
Ron Marrero
Chris Taliaferro
Cameron Taylor
Harrison Black
Lucas Gonzalez
Camilo Lozano
Tyler Duncan
Samuel Lewis

Project Sponsor:
Mike Conroy

Project Stakeholders:
Jason Schuler
Kurt Leucht

Last Revision:
15 April 2019

Table of Contents

Introduction

The EZ-RASSOR (Regolith Advanced Surface Systems Operations Robot) is an inexpensive, autonomous regolith mining robot designed to mimic the look and functionality of NASA's RASSOR, but on a smaller scale. The primary goal of the EZ-RASSOR is to provide a functional demo for visitors at the Kennedy Space Center. Business and international visitors will be able to interact with a scaled-down rover that functions and demonstrates the main RASSOR's capabilities. Some of the EZ-RASSOR's fundamental capabilities include:

- Roving across light-to-moderate terrain
- Collecting regolith in rotating, digging drums
- Returning regolith to hoppers located away from dig sites
- Autonomously navigating possible obstructions
- Cooperating in a swarm of other EZ-RASSORs

Development Team Narrative

When this project was first presented to the students at UCF, it was pitched as a two-team project with the "red" and "blue" teams being responsible for accomplishing the same goals. The focus was to have two five-person teams working separately on accomplishing the requirements based on their own interpretations. Since five was the maximum number of computer science students that could participate in on a given team, this was agreed to by both teams as well as the UCF staff.

After the red and blue team's first requirements meeting with the sponsor, Mike Conroy, it became clear that the requirements would be much better achieved with sub-teams designed to handle specific aspects such as ROS, AI, and the simulation. Thus, it began to make sense that the two teams should merge, although the blue team was hesitant on the struggles that a 10-person team would bring. At this point, the teams were still split.

It was not until the first requirements meeting with Jason and Kurt at NASA that both teams agreed that the requirements were outside of the scope of a single

5-person team given our time constraints. The design ideas that were discussed on-site, such as the implementation of GPS-Denied Navigation, were bigger in scope than what we had planned. We were certainly ready to tackle the new challenges, but understood that we would need to do it as a 10-person team. It was also at this on-site meeting that our team learned that the project would be renamed from the E-RASSOR to the EZ-RASSOR, with the objective no longer being in education but instead in demonstrations for Kennedy Space Center (described in more detail later in this document).

After meeting with Professor Heinrich to discuss the merger, he approved us to be the first 10-person Senior Design team comprised of Computer Science majors that would work on a single project. Following this approval, our team immediately went to work on merging documentation and the implementation ideas that each team had produced until that point. Ron Marrero agreed to serve as Project Manager for this transition in order to facilitate the document and resource merge.

Since the merger, the team has been just as diligent on completing the project requirements, and the team has a clear breakdown of which areas each team member is responsible for (listed in the work assignment document found later in this paper). The only constant issue since the merger is assembling the entire team for a group meeting, given everyone's varying schedules. Thankfully, this has easily been mitigated through online meeting tools like Slack and Skype, as well as the capturing of meeting notes for future reviews.

Broader Impacts

Based off of feedback from Mike Conroy, NASA may use contributions from our development team to assist their own production projects. Efficiently collecting regolith from a planet's soil is already a major challenge and while NASA has a RASSOR unit that is already being improved upon, having a scaled down version of the same robot could allow for them to solve digging challenges in an inexpensive manner.

Our code is also designed with open-sourceness in mind. None of the simulations that the development team ultimately produces will have export control data involved. In fact, this entire project utilizes open source code and

frameworks and we are forbidden from including any code that NASA has produced. The broader impact of this is that sharing our efforts with the world allows others to contribute to this project. Since NASA may use *our* efforts towards improving their own projects, they may end up with even more value from seeing community contributions to this project.

Initial Objectives

This project began as the E-RASSOR Project, with the “E” standing for Education. The initial goal was to create a robot (rover) that was designed as a low-cost open-source solution that could be shared with schools across the world. Below are the original objectives for the rover:

- Successfully deploying from the lander.
- Have the ability to self-right itself incase of flipping.
- Drive 100 meters from the lander to an excavation site to decrease the possibility of dust making contact with the lander’s solar panels.
- Utilize the on-board rotating drums to excavate the top 5 cm of surface regolith to test mining conditions.
- If conditions are adequate, mine at least 1 meter deep using a slot dozing trench method.
- Successfully mine to full drum capacity (700 kg) of regolith within 24 hours.
- Transport and deposit collected regolith to the processing plant on the lander.
- Continually repeat this mining process as long as the battery allows and dock to the charging station at the lander to recharge and perform the tasks again the next day.

Updated Objectives

Although progress was made to accomplish the above objectives, midway through the Fall semester, NASA leadership decided against the goal of this project being for education in such a short time span. Instead, the project requirements were changed to design the EZ-RASSOR, the final product of this

project. This rover will be used at Kennedy Space Center for demos concerning the production RASSOR. Additionally, this project is now designed to be a multi-year project, changing as the main rover changes, as well as to prototype new features before they are implemented on the main RASSOR. Below are the updated objectives for the rover:

- Fully controlled movement from a gamepad
- Obstacle detection through robot vision
- Self-right controls
- Automated movement to home base and from home base to a specified destination
- Digging controls through device arms that will also balance the rover

These objectives represent the topics that we constitute our work in this project. A complete breakdown of these objectives can be found in our Requirements section and a complete plan for how we will achieve these requirements can be found in the Implementation Plan section.

Since all hardware decisions will be made by the team at Swamp Works at Kennedy Space Center, our responsibility will be to implement the software architecture and create an all-new code base for the robot. The current iteration of the RASSOR's code will not be a resource in this project due to security concerns, so all software will have to be built from the ground up.

Motivations

Sean Rapp

As humanity envisions its future among the stars, especially in this exciting period in which it seems a new race to accelerate development is unfolding, the first thing that comes to the minds of many is setting foot onto the soil of another planet. Necessary for this process to begin, however, is the vital development of a practical and robust infrastructure. With the goal of all robotic systems currently populating outer space and other worlds being strictly scientific, a system geared towards civilization development and progression is both necessary and novel. As the RASSOR fills a crucial role in the ascension of our species to a space-faring civilization, it will be a machine written about and remembered.

Since the space race that brought humans to the moon, it has been far too long that we have ventured out, so it is critical that not only do we reignite the mission to advance humanity, but to keep that engine running by inspiring and educating future generations. This is where the EZ-RASSOR finds its incredible significance, and where I find inspiration.

Tiger Sachse

After European societies discovered the New World, they colonized both continents in 3 phases. First, these societies sent exploratory missions to learn more about the potential of the New World. Next, they created small, dependent colonies of adventurers on the frontier who extracted resources from the land and invented new ways to survive in harsh, foreign places. Finally, rugged colonies blossomed into fully independent societies where people could specialize and live full, safe lives. Today, civilization finds itself in the middle of this same process, but with a new frontier. We once dreamed of gold and spices in the New World; now we dream of helium-3, iron, and alien life in space.

Civilization is moving out of the first phase of Moon colonization (early exploratory missions) and into the second phase (dependent colonies). Mars has been extensively researched using robots and satellites, and so is not far behind the Moon in terms of colonial feasibility. This project, the EZ-RASSOR, is a preliminary step in moving civilization into the second phase of colonization on both the Moon and Mars.

My personal life goal is to assist civilization in colonizing the Moon, Mars, and other bodies in the Solar System. Although the EZ-RASSOR is intended for an educational audience and not a Lunar or Martian one, this project serves as a major step for me to learn about what goes into building autonomous machines for other celestial bodies. Also, the demo aspect of this project will hopefully inspire more children in the next generation to care about space and space colonization as deeply as I do.

Ron Marrero

Out of all the qualities I would like to emulate from a successful developer, I believe giving back to the community is the most important. This project fully embodies that quality in that we have been given the chance to take an existing proprietary project, and help re-imagine it as a potentially open-source project. This means no red tape and no waiting on funding, simply a project that we can

start sharing with all who are interested. I look forward to expanding my own knowledge and experience with programming in this project. Our team will be responsible for making hardware drivers and enabling artificial intelligence capabilities to allow multiple rovers to act as a swarm, both being areas I do not have too much experience in. This chance to develop my programming skills combined with the goal to make it open source will motivate me into doing the very best that I can in this project.

Cameron Taylor

I am interested in this project for two major reasons; the potential for developing an advanced reinforcement learning agent to control the system autonomously and the contribution to space exploration that this project will represent. I have always been fascinated with space and space travel and am a firm believer in the value of exploration and pushing for human progress simply for the sake of progress. My main goal for this project is to focus on building a strong autonomous control system capable of adapting to a changing environment with the relatively limited amount of environmental feedback available through the sensors onboard. I would like to learn about practical development of machine learning algorithms and hopefully contribute something exciting and powerful to the project. I hope that while I am on this project I can not only contribute to the RASSOR project in a scientific capacity, but that I might also be able to contribute something useful for the full size RASSOR on its missions to Mars.

Chris Taliaferro

This project is an incredible opportunity to learn new technologies and be a part of NASA's effort for space exploration. I look forward to take part in cutting edge studies that will achieve milestones in the scientific community. My experience as a student at UCF has proven my willingness to go above and beyond and I welcome the opportunity to work with NASA and provide a different outlook to the issues faced in their ongoing missions and studies. I am confident I will benefit and learn from this project in many ways, granting me with new skills that I can take with me moving forward in my career. I envision a career for myself in innovation where I can combine my interests and passions into something that the world can benefit from.

Sam Lewis

I have been interested in space since I was a little kid. The idea of exploring planets nobody else has ever set foot on is breathtakingly exciting. What

resources will you find? What if we find a cave? What if there's an underground lake in that cave? What if there's life in that lake? What if that lakes not even made of water? The universe is a very large place, and the possibilities for what we find are infinite as we explore it. As we explore and discover things aren't quite like what we believed, our views as a society change. So building a robot capable of making people able to explore different planets would be a meaningful project to society, and a dream come true. On top of that, this project contains aspects of computer vision, robotics and autonomous functions using swarm AI. I've been interested in these topics for a while now but I have never known where to start. This project will give me an opportunity to learn about these topics while working on something meaningful.

Tyler Duncan

In 1995, the film Apollo 13 was released and that summer after watching the movie an 8-year-old boy's obsession with space began. I remember being fascinated with how creative both the crew, as well as the scientists and engineers on the ground, had to be to solve major, time-sensitive problems with limited resources and expert knowledge of the vessel and the technology therein. I remember feeling inspired by both the bravery of that crew and how the country was united in bringing those men home safely. It was only two years later that the news broke that we had landed a rover named Pathfinder on Mars and the images it sent back captured my imagination.

When the announcement and commitments had been made to get men to Mars and a sort of competition between SpaceX and NASA began, I knew I wanted in. I see this project as a way for me to get my foot in the door. Although this project does not directly contribute to a Mars mission, there's an opportunity for the UCF development team to make breakthroughs that perhaps the Swamp Works Team possibly have not considered which in itself is exciting. This project has the power to completely alter the direction of our species. From being stuck on a lonely rock out in space, to being a multi-planetary species, and continue our deepest desires to explore the unknown, conquer nature, and elevate our understanding of our place in the universe.

Harrison Black

This project encompasses many aspects that bring forth motivation. For one, I thoroughly enjoy robotics and it is already a current hobby of mine. I take pleasure in working with embedded systems and coding microcontrollers. I enjoy

the process of creating a physical device, designing its behavior, and watching it move the way I intended it too. Right off the bat, this project provides fun as a motivation to see it through to completion and do a good job. Secondly, this project is an amazing learning experience. Working with NASA on their EZ-RASSOR will allow me to further improve my understanding and skills related to the realm of robotics. Additionally, it provides the opportunity to design and implement swarm robotics, a subject I am very interested in but have not yet had the chance to learn. Furthermore, I am captivated by the unknown. This project is a step towards discovering new things about the universe. It has the potential to aid in the progression of humans becoming an interplanetary species, which brings forth infinitely new unknowns to explore.

Camilo Lozano

Since I was a child, I was always fascinated with the marvels of space. Growing up I always dreamed of traveling to space and exploring the great beyond. As I grew, I came to realize that such conditions required to become an astronaut were not for me, but my marvel for it still remained. When I heard there was a chance to work and develop a project for NASA, I was immediately drawn to the idea. Working on such an incredible project to be able to develop a technology that is to be shared and used to help spread my same enthusiasm for space.

Moreover, for my senior design project, I wanted to work on something that would have meaning, that others would use and develop. I was at a crossroads choosing because I wanted to work for a 3rd party, but at the same time, I wanted to work on an open source project. Having been raised using Linux and other open source software I have always wanted to give back to the community that made me want to pursue my career. With this project I got the best of both worlds, working for a company for the experience, and getting to give back to the community.

I am excited for all there is to learn, develop, and grow. I have high hopes for the project which will continue after we finish and hopefully still be further developed.

Lucas Gonzalez

I have always had a major fascination with robotics and space so as soon as NASA began presenting a project with a robot, they had my interest. My interest continued to grow as the presenter continued to talk about their goals for deploying the RASSOR to Mars and the Moon to assist with their colonization.

Once they mentioned their intentions to have the EZ-RASSOR project be open-source, it immediately became my number one choice. The progress of space exploration and humanity will be dependent on space colonization, so the field will continue to grow at an exponential rate. I can imagine a student somewhere around the world stumbling upon our project, sparking curiosity until they themselves make even bigger contributions to the field. While many other projects had interesting ideas, none of them seemed to have such potential for true progress. With its combination of fields I'm truly passionate about and its future potential, I'm dedicated to creating the best project possible.

Backgrounds and Assignments

EZ-RASSOR Team Assignments										
	Harrison	Tyler	Sam	Camilo	Lucas	Ron	Tiger	Sean	Chris	Cameron
Gazebo Sim	X					X			X	
Phone App				X					X	
Swarm AI	X	X								X
Control Interface					X					
Robot Vision		X	X							X
Basic Motor Functions		X					X			
ROS	X						X	X		
General Auto Functions					X					
RC Car				X			X	X		
SLAM / Cosmic GPS	X		X					X		X

Scripts / Backend				X			X			
Documentation Management						X				
Mesh Network				X				X	X	
SEE Integration		X				X				

Figure 1 | This table shows the formal assignments for everyone in the development team for the remainder the this project.

Sean Rapp

While pursuing my degree in Computer Science, I have been exposed to various academic fields within computer science, which, like any driven computer science student, provides a well-rounded foundation to pursue specific interests within the field. One thing that has always stood out to me in terms of software development are physical results, or in other words, developing software to control moving parts. Often in our studies, our assignments and learning are kept strictly within the computer we are developing on: writing code to output some number, perform a certain simulation, or execute some purely software-based procedure.

My first encounter with programming anything physical was when helping a mechanical engineering student write code for a microcontroller to run a set of wheels, and that experience sparked an interest in lower level programming with physical parts. Creating a bridge between the purely-software world to something that the physical world can interact with seemed to open up a new realm of possibilities. Thus, I endeavored to commit to the development of the lower level systems of this project, working with ROS on the EZ-RASSOR.

While my only experiences with hardware have been with simple microcontrollers and microcomputers, flashing LEDs or turning a motor or two, I am excited to get the opportunity to work with a more complex system to develop a robust software solution for the robotic mechanisms of this project.

Additionally, the working on the mapping system that will be created for use with the EZ-RASSOR system will be an incredible opportunity. Extrapolating data from satellite images for use in map construction is beyond intriguing to me - working with real data from space is one of the most exciting ideas. In the past, I have experimented with data provided by the Mikulski Archive for Space Telescopes (MAST), particularly that related to the Kepler and K2 missions that searched for exoplanets around stars using the transit method for detection. Also, I have worked somewhat with data related to pulsars, doing image processing to detect them. While not directly translatable, I am looking forward to carry on my interest in working with real astronomical data into the mapping aspects of this project.

Tiger Sachse

My journey to Computer Science started four years ago on a bus in Gainesville. I was studying environmental sustainability at the time and I wanted to make my daily commutes more productive. I'd always had an affinity for computers, and so I thought that I ought to try my hand at programming. After some searching I found an app that let me learn Python concepts through quizzes and little challenges. A summer of practicing later I was placed with a new roommate and he helped convince me to change my major and pursue computer science. Once the year ended, I consolidated my credits into an Associate's Degree and transferred to UCF.

My style and specialties in CS really started to take shape in the last year or so. Professor Szumlanski helped shape my opinion on most of what is important in CS today, and my style continued to sharpen and become more concise each time I sat down to create something. I'd tried front-end development, back-end development, and low-level programming, but my preferences did not become perfectly obvious to me until Computer Logic and Organization, a class that revolved around Boolean logic, the central processor, and assembly. I absolutely fell for assembly and developed an intense appreciation for C. After that class I knew: low level, close to hardware development was where I wanted to be.

More recently, I'd come to greatly appreciate scripting, particularly shell scripting for projects, writing generation scripts, and writing SQL scripts. This extended my admiration for Python, my first love. In short, I've realized over the years that I am a systems developer, a toolchain developer, and a low level developer.

This skill set prepares me fully for the task at hand, namely programming a robot in a way that is simple and straightforward for people to understand. I'm excited to use ROS, which will allow me to work near the hardware, but still write in Python, my favorite language. My scripting abilities will be best applied once we begin to decide how to package and distribute the software. I want to write scripts that take the confusion and work out of deployment, so that students can simply install Raspbian, execute a command or two and have the codebase downloaded, compiled, and ready to go.

Ron Marrero

As a senior in Computer Science, I have had the opportunity to take classes that require a range of skills and programming languages. One class I took that will benefit me in this project is Robot Vision. I learned how to implement concepts such as Convolutional Neural Networks that will be crucial to the success of this project. While I am not the only team member with experience in Robot Vision, I will certainly be able to contribute the knowledge I have.

My main interest in this project has been for the simulation portion. While I will not be the only team member working in this area, I am interested in the conditions necessary to make a successful rover. This will stretch beyond simply making 3D models for the rover and will also include the physics behind the movement of the rover itself.

This task involves researching simulation techniques in Gazebo and investigating on similar work that others have done in the field. Meeting the Swamp Works team has proved to be invaluable as well since I learned from their pitfalls in designing the models for their RASSOR simulations.

I will also work on testing the simulations and working with the hardware team to ensure accurate simulation conditions. As hardware specifications change, I will be responsible for ensuring that the Gazebo model specifications change as well. The most important deliverable for this project is definitely the EZ-RASSOR software, but if done correctly, I believe the simulation models will be just as important as it will be another piece of this open-source project.

Cameron Taylor

Computer Science has been my dream career field since early high school. I was always talented with mathematics and logic based subjects, but I didn't really

enjoy the idea of doing something so abstract that it lacks immediate impacts. I was also fascinated with intelligence, consciousness, and learning which led me to the field of artificial intelligence. This project goes hand in hand with that interest because it not only requires a form of machine intelligence, it also involves a system dealing with real world input and is fairly open ended in terms of the potential goals.

I currently have a small amount of experience with machine learning and AI through projects that I have done in class and on my own. I am familiar with the methods and the math that goes into the various forms of machine learning and I was really excited to finally be able to get some practical experience in this field.

Because of my previous experience and desire to attend graduate school for AI/Machine learning, I will be leading the development of the autonomous control system for the EZ-RASSOR. I will also be assisted by Chris because of his interest in the subject. The skill set that I have will be a great help in developing this system but there is still so much for me to learn. I look forward to enjoying the experience and also growing my skills in the AI field.

Chris Taliaferro

Throughout my time studying Computer Science at the University of Central Florida, I have honed many skills that will help me and the team in our plight to develop the EZ-RASSOR. I am able to implement web-applications and mobile apps with emphasis pristine design. I have not been taught web development in my coursework, although my interest in the subject has taken over my independent studies and has made me a competent web-developer. This is why I have chosen to take the route to develop the mobile app controller for the EZ-RASSOR.

I have successfully completed CAP 4453 which is UCF's course in Robot Vision. I have examined some of the elementary concepts in machine vision such as edge-detection, methods for obtaining shape information from images, object detection, and motion analysis. I have been exposed to several unsolved problems in the subject as well as the latest and greatest from that field. My primary focus for this project is not vision but I will be able to offer my insight to any problems we might encounter during development.

Sam Lewis

When I first started college I had no idea what I wanted to do with the degree, I just thought having the computer do something for me was fun. As I got closer and closer to graduation though, I realized if I wanted a job, I needed to figure what I actually wanted out of my computer science degree. Do I want a job in front-end development? Back-end? Security? I started asking myself these questions and realized I had no idea what I wanted.

So I started trying everything that I could. I joined the Hack@UCF club to learn more about security, I took a databases class to learn about databases and built the front end of my project in the software engineering class. I am currently taking the graduate version of the computer architecture class to learn about hardware, and in the past have taken a class on robot vision. Unfortunately, I didn't feel like I learned enough in that class to make a decent enough decision about that topic. So for this project, I am heading up computer vision aspect of the robot, specifically the object detection section. I am hoping to make some connections to the theory taught in class, and look forward to all the things I will be learning.

Tyler Duncan

Over the course of the last 4 years my interest in developing video games, while still present, has began to fade as I have fallen head over heels in love with the entire spectrum of Computer Science. Recently the areas of Artificial Intelligence, Computer Vision, and Machine Learning have peaked my interest. Being on the cutting edge of technology and pushing humanity forward into new exciting frontiers is where I hope to make a name for myself as I start my career in this exciting field.

When the news broke that Elon Musk and SpaceX were planning on getting a man on Mars in my lifetime I knew that I somehow needed to be a part of it. Naturally, hearing NASA offer an internship to work on a robot that will be mining the surface of Mars peaked my interest. Up to this point, with the exception of my limited experience with Unity, most of my programming experience has been manipulating data on a computer internally and getting boring command line output to confirm a programs success or failure. While those programs have all been rewarding, I am beyond excited to work on a project where my code will actually move physical objects in the real world. It presents all new challenges and problems that will be fun to solve.

As an added bonus this will be the first time I will be working with a large group of talented individuals. Although I imagine there will be many parallels with writing songs in a band, it will be interesting to see how we are able to manage our time and resources, stay organized, and work as a team to complete such an exciting project. I have the utmost confidence that this team will be successful in its pursuits and that we will gain invaluable knowledge along the way as we get first hand experience working on a large project with a large group and a finite amount of time.

In this project, my responsibilities will include helping to design the swarming behaviors of the EZ-RASSOR and to also contribute the the EZ-RASSOR's vision system. As these are my current interests in the field, I look forward to applying my knowledge and having my confidence tested as I am welcomed by new challenges, sleepless nights, and opportunities to learn from my mistakes as we attempt to solve these problems. I have completed courses in Robot Vision and Artificial Intelligence and plan to attend courses in Robotics and Machine Learning in the spring when the execution of this team's design is underway. These courses have and will give me a foundation and knowledge base to inform my methods as the task at hand moves forward.

Harrison Black

A few semesters into my major at UCF, I felt I already gained a very strong grasp on the development of software and I was eager to learn something new and exciting. This lead me to the decision of joining the UCF Robotics Club. It was there, at the robotics club, that I gained a vast amount of knowledge from my new found friends and peers. I developed an understanding of, and skills pertaining to single board computers, servos, IMUs, computer vision, the Robot Operating System (ROS), robotics decision making, and much more. The club exposed me to the physical side of computer science which, sadly, I believe most graduates don't get the chance to experience.

Due to my prior history working with ROS, I will be leading the design of the ROS hierarchy. It is my job to develop an efficient way to send data throughout ROS and translate input from the AI or a physical controller into a command for a corresponding robotic action. Although I will likely not develop each individual ROS node, it is my responsibility to ensure all nodes function correctly pertaining to all things ROS. Furthermore, during my time at UCF I have taken courses

about AI and machine learning. Due to this and my experience at the robotics club, I will also be assisting in the development of the EZ-RASSOR's AI decision making and the corresponding physical actions.

Camilo Lozano

I've been programming since since my Junior year of high school. I participated in programming competitions since then. In fact what drove me to pick UCF over other schools was that experience of coming to a competition here. Since then I've been programming in a multitude of languages ranging from Python, C, C++, Java and Javascript. I had always been passionate about computers since I was young, but what really settled what I was gonna do was a combination of Process of Object Oriented Programming, and writing some software for an embedded system for a company. I realized what I wanted to do was software development, full stack in fact. The way how a back-end database interacts with some API endpoints to get a front end result for a user is fascinating to me and it makes sense.

Due to this background in my field, I am assigned to help and develop the whole app side of the project mainly. From the backend development that will be needed for the on board hardware to the actual front end interface of the application that will be written. I've had previous experience in developing cross platform applications through React Native. This experience along with Chris' same experience in React Native will lead us to have a successful end product. On of the main challenges that we will be facing is writing the API endpoints for the hardware. We have a couple of considerations from the limited processing power to the lack of a central communications for each robot. Issues that will have solutions but will require a lot of planning to not only implement successfully, but to implement with a proper design that can be modular and easily built upon.

Besides the application, I will be aiding in the design of the actual software on board of the hardware, from OS level operations to scripts that will be need run, to simply just system administration. I have had an extensive Linux based background that will aid in such assignments. As well as I would taking care of how the team develops the ROS modules through Docker, which should make the development easier for everyone to have one common system to be able to run all the applications. I hope to be able to learn about some new tools to use and gain lots of experience in working in a team together.

Lucas Gonzalez
For my entire four years of high school I had been a prominent member of the robotics club. Within it I built many types robots for all sorts of competitions such as racing, collecting objects, or destroying other robots. With most of these competitions using remote controlled robots, I learned how to optimize the operator's ability to manage their robot and complete their goals as efficiently as possible. Within my experience of robotics, I have also had to deal with many frustrating situations of trying to figure out why a robot was not behaving in the way I wanted to. Because of these experiences I am developing the driver station of the EZ-RASSOR. The driver station is what users will be directly interacting with whether it be for monitoring an autonomously running robot or having full control of it. I will be using my experience to ensure that the user can monitor and operate the robot as efficiently and effectively as possible.

I have also always had an interest in automation and find nothing more satisfying than having what was once a complicated task be accomplished by the press of a single button. Because of this interest, I volunteered myself to develop the autonomous and semi-autonomous functions of the EZ-RASSOR. These will be the predefined procedures that the user and the AI will be able to deploy in order to accomplish complicated tasks with ease.

Function

As the hardware for the device is still being constructed, we have a very high-level overview of the design criteria and constraints. Probably chief among the constraints in this project is the restriction on finances. With a main goal being cost minimization, we are not allowed to use any expensive proprietary software or any grandiose hardware to complete this project. Instead we will rely on open-source software such as ROS, an open-source middleware software for robot applications, to accomplish the goals of this project.

Another design constraint will be size. While we have not yet seen the final product, the development team has been given direction from our sponsor as to approximately how large they want the device to be. From our estimations, it should be no taller or longer than around 8 inches, which will mean a constraint on computer resources available. Since we will be designing the drivers and will

be responsible for creating some “swarm” functionality for these rovers, we will have to efficiently make use of the resources available, likely having to account for every single block of memory in the code.

The EZ-RASSOR must be a scaled down version of the original RASSOR. This scale stretches from size to price of the final deliverable. Like it’s big brother, the EZ-RASSOR will need to be able to perform activities on it’s own. It will need to be artificially intelligent in performing its responsibilities, and it must be able to solve any problems it encounters along the way. It will also need to be available for purchase at a reasonable cost. This is because the EZ-RASSOR project strives to promote innovation and give the general public access to cutting-edge technology. The lowered price point makes the package available for the everyday classroom environment We will also need to implement a simulation environment to test the EZ-RASSOR’s capabilities. In turn, this means our code needs to utilize some form of modularity to switch between simulation and hardware seamlessly.

Additional Design Ideas from the Team

Cameron Taylor

- Host simulation on AWS
- Use OpenCV for computer vision tasks
- Use ROS to control main operations of EZ-RASSOR
- Combine routines like path finding with true AI for high level decision making

Tiger Sachse

- Include maps in the control apps for coordinate selection
- Use an optional Xbox controller for manual movement and drum control
- Use a ROS graph to really take advantage of ROS’s portability
- Create a sort of scripting language to write robot routines in, or use a pre existing one
- Create a Python API to write scripts in for movement routines
- Include installation and setup scripts in Shell for easy configuration for students

- Create a swarm of RC cars using a Raspberry PI before programming the actual robot
- Use WiFi to connect the robots instead of Bluetooth for better signal reliability
- Have an access point or a broadcasting robot that creates and manages the WiFi network
- Use the RC cars to demonstrate wheel and swarm functionality
- Create maps of terrain using swarm data as the rover travels
- This map can then be used by the swarm for better/easier obstacle avoidance and detection
- Use a standardized message format for maximum portability

Sean Rapp

- Develop a robust, all-encompassing testing routine suite to enable quick testing of entire routine pipeline, including software and hardware cooperation
- Include an on-board feedback system for the purpose of ensuring congruence between simulated EZ-RASSOR and real-world EZ-RASSOR function

Ron Marrero

- Create an educational video to demonstrate proper use of our software and features
- Create an external API document that details the operations of the EZ-RASSOR from a development standpoint. More than just a GitHub repo link, this document will have the same level of detail as a Programming Language User Guide

Chris Taliaferro

- Implement a mobile app as the controller. This will help make the the entire package more distributable and help lower cost
- Utilize easy to obtain hardware components to give the user the option to build it, then program it all at an affordable price point
- Allow the EZ-RASSOR to connect to a computer via USB and load in pre-programmed routines to perform

Sam Lewis

- Odroid as the main microcontroller
- Arduino for the motor controller

- Remove stereo cameras, replace with a regular camera and a range detector
- Lidar for range

Tyler Duncan

- Sonar for range
- Start in python transfer to C++ for increased efficiency
- The coordinate system for robot location tracking distributed through the robot network
- Infrared I/O for peer to peer communication between robots
- Position triangulation between robots for precise positioning

Harrison Black

- We will have to make a simulation for ROS and everything will be split up and should not care if the commands are coming from the autonomous functionality or the driver. Split into Drums, Camera, and Motors files
- Everything should be mostly servo controls
- Need to make an interface for driver control

Camilo Lozano

- Separate files for Autonomous and driver functionality
- Control the speed of the drum motors while digging depending on the material being excavated to prevent damage to drum motors
- The depth of the wheel could be used to determine how dense the regolith material is and therefore how fast to spin the drum motors

Lucas Gonzalez

- General-purpose track following programs and general purpose wheel slippage corrections.
- A virtual map in the robot's memory
- Implementation of alert system

Budget

Software

The majority of the project will likely be done using open source/free software such as ROS, OpenCV, and tensorflow. Part of the project will be to implement a

simulation to test the software before it is loaded onto the physical RASSOR and this simulation could potentially be hosted on Amazon Web Services (AWS). The project may also involve some reinforcement learning and object detection, which could potentially involve the need for some additional hardware also provided by AWS through their sagemaker tool. Amazon provides a free tier of services that includes up to 250 hours per month of a t2.medium instance for designing and testing models with an additional 50 hours of m4.xlarge for training the models. The cost for any additional training time will depend on the instance chosen, but the p2.xlarge is a reasonable choice at $1.26 / hr. The potential training needs can not be fully estimated until the exact onboard hardware has been determined, however an upper estimate would be an additional 100 hours coming out to $126. The Phase 2 requirements also include creating an iOS/Android app as the controller for the system. This would introduce a fee for publishing the app; $25 for the Google Play Store and $99 for the iTunes App Store.

Hardware

Our team is using an RC car kit from Sunfounder, with pre-existing Raspberry Pi, to create a demonstration RC car for the EZ-RASSOR. The kit is priced around $90, and some additional parts bring the cost of the demonstration to around $110. Our team received a lot of help and free printing/acrylic cuts from the Innovation Lab on campus. *A very important thanks goes to Elizabeth Nogues for all of her assistance in building and re-engineering the RC car kit to fit our team's needs.*

Budget Estimation Line Items:

Item	**Price**
IOS Store App Publishing	$99
Google Store App Publishing	$25
AWS SageMaker	$50-100
Sunfounder kit, Power Supply & SD Card	$120
Batteries/misc.	$20

Administrative Printing Costs	$150
Total	$464-514

Figure 2

Development Phases

For the benefit of those who will read this document, we have cataloged all of our planning *and* work across the entire Fall 2018 - Spring 2019 semesters. All our efforts that will be discussed are broken up into two different phases: requirements gathering phase and implementation phase. Each phase highlights the work done in the Fall and Spring semesters, respectively. Variances between planning and execution are highlighted and will be beneficial to the reader, as the best of planning does not always lend itself well to execution.

Requirements Gathering Phase

This development phase took place strictly during the Fall 2018 semester. In this phase, the UCF development team was responsible for researching the technologies that we would be using in this project and also documenting the reasons why we chose not to use other technologies.

Additionally, this phase initially began as an effort between two competing five-person teams: the red and blue teams. This team split was done by design from the project sponsor as a means to produce higher quality work. However, after both teams first visited the Swamp Works lab and spoke to the NASA engineers, we determined that this project would actually be more successful if we were to work together as a single ten-person team. This decision was approved by the Senior Design professor, Dr. Mark Heinrich, and our FSI sponsor, Mike Conroy.

Initial Requirements

When this project first began, the development team had defined an initial set of requirements necessary for the completion of this project. This requirements list was created after discussing the project only with our FSI sponsor and without having spoken to Swamp Works. It is shown below and is listed for historical reference:

- The EZ-RASSOR software shall be delivered at the end of April 2019.
- The EZ-RASSOR software shall be open source.
- The EZ-RASSOR software shall be adequately documented to facilitate maintenance and continued development.
- The EZ-RASSOR software shall adhere to proper object-oriented and SOLID design principles.
- The EZ-RASSOR software shall provide a solution that is presentable for demos around the Kennedy Space Center.
- The EZ-RASSOR software shall be parameterized to facilitate transfer to other similar devices.
- The EZ-RASSOR software shall be designed to allow for a smooth migration of the simulation environment to another environment.
- The EZ-RASSOR software shall be designed to allow for a smooth migration of the autonomous control system to another system.
- The EZ-RASSOR software shall be designed to allow for the straightforward inclusion of additional remote control devices.
- The EZ-RASSOR software shall contain a simulation.
- The simulation shall be developed as an open source model.
- The simulation shall be developed in Gazebo.
- The simulation shall contain tests of the desired EZ-RASSOR features.
- The simulation shall contain several test environments to ensure the system is robust.
- The simulation shall be to scale in terms of measurements and physics of the environment.
- The EZ-RASSOR software shall receive input from an external controller.
- The controller shall be in the form of an iOS and Android app.
- The controller shall contain movement controls in the form of “forward”, “backward”, “left”, “right”.

- The controller shall contain live video feeds directly from the cameras located on the EZ-RASSOR body.
- The controller shall contain a panel that displays status information about the EZ-RASSOR.
    - Health
    - Control Status
    - Current Objective (AI)
    - Battery
    - Regolith Quality
    - Position
    - Distance Traveled
    - Bucket Drum Capacities
    - Arm Angles
- The controller shall be designed with tablets in mind, but will also provide a simplified smartphone version.
- The EZ-RASSOR software shall contain a central middleware system to run the robot's external hardware.
- The central middleware system shall be ROS.
- The central middleware system shall manage all relay between the simulation, autonomous control, and remote control modules.
- The central middleware system shall be made up of a network of ROS nodes connected in a ROS graph.
- There shall be ROS nodes for the following hardware:
    - Wheels
    - Cameras
    - Arms
    - WiFi receiver
- There shall also be several ROS nodes for data logic and processing.
- These logic and processing nodes shall be written in Python.
- Logic and processing nodes written in Python shall include unit tests written using the Unittest framework to ensure computational integrity in normal and unexpected conditions. These tests shall be easily runnable from the Bash/ZSH command line and will give students a functional example of unit testing.
- There shall be an automated system to configure and manipulate the ROS graph in an easy way from the Bash/ZSH command line.
- ROS shall enable multiple robots to communicate over a network using WiFi nodes (for swarm functionality).

- There shall not be separate ROS graphs for individual robots, but rather a single ROS graph. This simplifies communication between nodes.
- All messages sent between nodes shall be formatted in a standardized way.
- Message formatting standards shall either be adapted from built-in message formats or be completely custom and tailored to the exact needs of the communicating nodes.
- Any ROS nodes that have been created by external developers shall be either shipped with this software or be automatically downloadable for easy deployment by students.
- All necessary software for the EZ-RASSOR shall be easily deployable via an installation script or another packaging mechanism.
- The EZ-RASSOR shall gather materials autonomously.
- The EZ-RASSOR shall deposit materials into processing unit on lander autonomously.
- The EZ-RASSOR shall navigate 3D space autonomously.
- The EZ-RASSOR shall exhibit obstacle detection.
- The EZ-RASSOR shall exhibit obstacle avoidance.
- The EZ-RASSOR shall exhibit detection of neighboring EZ-RASSORs.
- The EZ-RASSOR shall exhibit tracking of neighboring EZ-RASSORs.
- The EZ-RASSOR shall calculate the distance to objects in its view.
- The EZ-RASSOR software shall contain an autonomous control option.
- The autonomous control option shall include auto-dig capabilities.
    - The auto-dig capabilities shall include options to dig to a specified fill capacity.
    - The auto-dig capabilities shall include a mobile dig option.
    - The auto-dig capabilities shall include a stationary dig option.
- The autonomous control option shall include exploration capabilities.
    - The exploration capabilities shall include environment map building.
    - The exploration capabilities shall be limited within a limited range.
    - The exploration capabilities shall be collaborative with other EZ-RASSOR’s.
    - The exploration capabilities shall be capable of marking resources and hazards within a given search area.
- The autonomous control option shall include two modes of operation; “Fully Autonomous” and “Semi-Autonomous”.

    - The fully autonomous mode should be capable of utilizing swarm and AI to explore the environment and mine/return resources collaboratively with other EZ-RASSOR systems.
    - The fully autonomous mode should be capable of operating continuously without human input.
- The autonomous control option shall provide environmental awareness.
    - The environmental awareness shall develop a 3d map of the environment
    - The environmental awareness shall contain subroutines capable of performing specific actions.

Defined Requirements

Since the merger of this project, the development team went hard at work to define the requirements for this project in conjunction with Mike Conroy and the Swamp Works team. These requirements explain *what* we are going to do in this project and the subsequent section on execution plans explains *how* we are going to accomplish the requirements.

In discussing this project with our sponsors and mentors, we referred to the main requirements and stretch goals as Phase 1 Requirements and Phase 2 Requirements, respectively. This is due to the fact that another UCF development team will work on this project in a later semester. Therefore, it made sense to also identify requirements that could be solved at a later date. From these requirements that we defined internally, we crafted the statement of work document that was sent to the mentors and sponsors for approval.

These requirements are repeated on the attached Statement of Work Document and are as follows:

Phase 1 Requirements

Hardware

- The team shall create a functional demonstration RC car to test the team's ROS graph. This car shall include a Raspberry Pi, WiFi antenna, and four wheels.
- The team shall create an application for communication with and manipulation of the EZ-RASSOR and the RC car. The application shall allow for input from either a keyboard/mouse or a gamepad.

Robot Vision

- The team shall create software to detect obstacles in the EZ-RASSOR's path, using data provided from an onboard webcam.
- The team shall create software to identify obstacles according to approximate height and width.

ROS

- The team shall create ROS nodes that allow for multiple EZ-RASSORs to share data.
- The team shall create startup scripts for the EZ-RASSOR for easy deployment on both the EZ-RASSOR and the RC car.
- The team shall create autonomous functions for the rover that are either user-initiated or rover-initiated. They are defined as follows:

User-Initiated Functions	**Rover-Initiated Functions**
Movement along a fixed path	Obstacle avoidance
Raise/lower arms	Self-right if fallen
Dig/spin drums	Deploy from base
	Return to base
	Dump drum contents
	Adjust arms to balance

Gazebo

- The team shall create a flat-surface environment in Gazebo with obstacles for the purposes of testing
- The team shall create a functional EZ-RASSOR model in Gazebo.
    - The model may or may not accurately reflect the final hardware product provided by NASA
- The functional Gazebo model shall be capable of movement given input movement commands
- The functional Gazebo model shall be capable of obstacle detection within the simulation, to demonstrate the functionality of robot vision.

Phase 2 Requirements

Hardware

- The team shall create a mobile application as an interface for the EZ-RASSOR to control movement and to initiate pre-defined functions such as “dig”.
- The team shall include movable arms on the RC car, as well as simulated drums on the ends of the arms. These simulated drums shall consist of LEDs to demonstrate a rotational effect.
- The team shall enable the RC car to communicate with other RC cars (or simulated RC cars on a laptop). This enables the RC cars to demonstrate any swarm technology that the team creates.

ROS

- The team shall design all ROS nodes in such a way that they can be easily packaged and deployed by other people. This means that all ROS nodes will be distributable and standardized.
- Any message formats used will either be built-in formats or will be custom but distributable.

Interface

- The team shall design an Android and/or iPhone application to control the EZ-RASSOR.

Swarm AI

- The team shall design ROS nodes capable of Swarm operations. These operations are defined as autonomous functions that a group of EZ-RASSORs can collaborate on.

GPS Denied Navigation

- The team shall design and implement functionality for the EZ-RASSOR to determine its relative location on a planet given pre-loaded orbital data and the current video feed from the onboard rover camera(s).
- The team shall take this combination of data to create Terrain Relative Navigation, assisting in planned trips in relation to the terrain.

Statement of Work Document

As stated, the below document is the statement of work document (SOW) that our team has created to sign-off on requirements with the sponsor and

stakeholder. It is slightly visually different from the rest of the document as this section will be printed out for confirmation and signing. While a SOW was not a requirement of this project, we found that having it allowed to complete our commitments as well as to have those commitment agreements documented.

NASA EZ-RASSOR Project

Audience:
NASA Swamp Works
Florida Space Institute

Development Team:
UCF - NASA EZ-RASSOR Team

Purpose

This Statement of Work (SOW) is an established agreement between the Client ("NASA Swamp Works" and "Florida Space Institute") and the Development Team ("UCF - NASA EZ-RASSOR Team"). This document will serve to outline the project requirements, milestones, resources, assumptions, and procedures that are necessary for the completion of this project. By signing this document, the client and development team agree that the project is considered "successful" once the below requirements are met. After the agreement is made and signed, the client understands that any additional requirements will require more time to accommodate the work. If the time requirement surpasses the project deadline set by UCF, the development team will be unable to complete the requirement. This is to ensure that the development team successfully passes the course.

The client can reconfigure this document to suit their needs before signing. Any reworking of this document after signing from either party will require both parties to sign a new version of this document. The latest signed document is the one that will be considered current.

Phase 1 Period of Performance

The development shall commence on October 15, 2018, and shall continue through April 5, 2019. After this, an integration test shall be performed with the client to demonstrate the functionality of the project. Final delivery of the project with all necessary documentation shall be provided on April 19th, 2019.

Project Members

Project Sponsor:	Mike Conroy	(mike.conroy@ucf.edu)
Stakeholders/Mentors:	Kurt W. Leucht	(kurt.leucht@nasa.gov)
	Jason M. Schuler	(jason.m.schuler@nasa.gov)
Development Team:	**Ronald Marrero (Project Lead)**	(ronald.marrero@knights.ucf.edu)
	Cell: 407.558.1386	
	Chris Taliaferro (Communications)	(guitartaliaferro@knights.ucf.edu)
	Tiger Sachse	(tgsachse@gmail.com)
	Camilo Lozano	(clozano@knights.ucf.edu)
	Lucas Gonzales	(lcgonzalez@knights.ucf.edu)
	Cameron Taylor	(cetaylor758903@knights.ucf.edu)
	Harrison Black	(harrison.w.black@knights.ucf.edu)
	Samuel Lewis	(samuel.lewis@knights.ucf.edu)
	Tyler Duncan	(tduncan13@knights.ucf.edu)
	Sean Rapp	(seanprapp@gmail.com)
Senior Design Professor:	Dr. Mark Heinrich	(heinrich@cs.ucf.edu)

Scope of Work

After a change in project scope, it was determined by FSI and NASA that the EZ-RASSOR project would be a long-running project to be completed by multiple teams at UCF. The UCF development team listed above is responsible for the completion of the hard-requirements that have been presented, and all remaining requirements will be left to be worked on by a future team. These requirement sets are split into Phase 1 and Phase 2.

Completion of the Phase 1 requirements by the final date mentioned above will satisfy the completion of work from the current UCF development team. The UCF

development team may work on Phase 2 Requirements if time allows. Additionally, at the end of the Phase 1 deadline, the current UCF development team will produce a report with the status of the below Phase 2 requirements for future work and will include this report in the GitHub repository.

Phase 1 Requirements

Hardware

- The team shall create a functional demonstration RC car to test the team's ROS graph. This car shall include a Raspberry Pi, WiFi antenna, and four wheels.
- The team shall create an application for communication with and manipulation of the EZ-RASSOR and the RC car. The application shall allow for input from either a keyboard/mouse or a gamepad.

Robot Vision

- The team shall create software to detect obstacles in the EZ-RASSOR's path, using data provided from an onboard webcam.
- The team shall create software to identify obstacles according to approximate height and width.

ROS

- The team shall create ROS nodes that allow for multiple EZ-RASSORs to share data.
- The team shall create startup scripts for the EZ-RASSOR for easy deployment on both the EZ-RASSOR and the RC car.
- The team shall create autonomous functions for the rover that are either user-initiated or rover-initiated. They are defined as follows:

User-Initiated Functions	Rover-Initiated Functions
Movement along a fixed path	Obstacle avoidance
Raise/lower arms	Self-right if fallen
Dig/spin drums	Deploy from base
	Return to base

	Dump drum contents
	Adjust arms to balance

Gazebo

- The team shall create a flat-surface environment in Gazebo with obstacles for the purposes of testing
- The team shall create a functional EZ-RASSOR model in Gazebo.
    - The model may or may not accurately reflect the final hardware product provided by NASA
- The functional Gazebo model shall be capable of movement given input movement commands
- The functional Gazebo model shall be capable of obstacle detection within the simulation, to demonstrate the functionality of robot vision.

Phase 2 Requirements

Hardware

- The team shall create a mobile application as an interface for the EZ-RASSOR to control movement and to initiate pre-defined functions such as “dig”.
- The team shall include movable arms on the RC car, as well as simulated drums on the ends of the arms. These simulated drums shall consist of LEDs to demonstrate a rotational effect.
- The team shall enable the RC car to communicate with other RC cars (or simulated RC cars on a laptop). This enables the RC cars to demonstrate any swarm technology that the team creates.

ROS

- The team shall design all ROS nodes in such a way that they can be easily packaged and deployed by other people. This means that all ROS nodes will be distributable and standardized.
- Any message formats used will either be built-in formats or will be custom but distributable.

Interface

- The team shall design an Android and/or iPhone application to control the EZ-RASSOR.

Swarm AI

- The team shall design ROS nodes capable of Swarm operations. These operations are defined as autonomous functions that a group of EZ-RASSORs can collaborate on.

GPS Denied Navigation

- The team shall design and implement functionality for the EZ-RASSOR to determine its relative location on a planet given pre-loaded orbital data and the current video feed from the onboard rover camera(s).
- The team shall take this combination of data to create Terrain Relative Navigation, assisting in planned trips in relation to the terrain.

Deliverable Materials

The EZ-RASSOR Team shall produce the following deliverables:

- A complete GitHub repository with all related software pertaining to this project including ROS nodes, Gazebo simulations, and any code related to stretch goals
    - All code produced will contain accurate comments to describe processes being implemented. At a minimum, each method and code file created will receive a comment to describe it's workflow.
- A functional RC car that demonstrates the functionality of the EZ-RASSOR code
    - The team reserves the right to keep this RC car
- A PowerPoint presentation summarizing the team's work on the project
- An on-site NASA demonstration of the successful completion of project requirements
- An on-site UCF demonstration of the successful completion of project requirements
- An in-depth Final Design Document that fully describes the journey taken by the development team included all of the steps needed to reproduce their efforts in this project

Student Responsibilities

The students on the NASA EZ-RASSOR team have agreed to the following responsibilities:

- Will maintain open communication with the project sponsor on our project
- Will complete the agreed upon requirements
- Will honor the goal of open source, and not using any unlicensed or copyrighted software to achieve the goals of this project

Client Responsibilities

The sponsor and stakeholders for this project have the following responsibilities:

- Will be available via GitHub communication or email to answer questions related to the project
- Will maintain a line of communication with the Swamp Works team to gather information that we can apply to our project or to provide feedback on our progress
- Will maintain a line of communication with Dr. Heinrich and our team to keep Dr. Heinrich and our team updated on any new requirements that may arise

Out-of-Pocket Expenses

Mike Conroy has provided the UCF Development Team with instructions on applying for a project grant through the NASA Florida Space Grant Consortium which can be found here: https://floridaspacegrant.org/programs/senior-design-projects/.

It will be the UCF Development Team's responsibility to assess whether or not resources will need to be purchased to complete the project and to properly submit a grant request as early as possible.

Assumptions

The UCF Development team has made the following assumptions about the project:

- The Swamp Works team will complete a stable version of the EZ-RASSOR hardware on or before March 2019 so that we can begin implementing our ROS nodes on the hardware
- Any open source technologies or languages our team uses are per our discretion as we have not been required by the sponsor to use any particular language(s)
- We will be able to engage in frequent (weekly) communication with SMEs at NASA to discuss implementing project requirements as well as to provide feedback on our progress

Requirements Change Procedure

The following process will be followed if a change to this SOW is required:

1) Changes to the SOW must be communicated to the team via email for documentation purposes.
2) The team will evaluate the time effort required and ensure that the time needed to add/modify any requirements does not exceed the total time available for this project.
3) If there are any disagreements on modifying requirements, both parties should contact Dr. Heinrich to make a final decision as the team wishes to successfully graduate while also fulfilling the requirements we have agreed to.

IN WITNESS WHEREOF, the parties hereto have caused this SOW to be effective as of the day, month and year first written above.

Student	Sponsor
Name:	*Name:*
Signature:	*Signature:*

Stakeholder	Professor
Name:	*Name:*
Signature:	*Signature:*

Research Design

These research design sections reflect the countless hours that we spent in the requirements gathering phase to investigate the technologies we would be using. A deep dive into multiple ways to approach our project is listed here which was crucial in determining the pros and cons between various technology options in the next section.

Backend Systems Communication - (REST)

REST (Representational State Transfer) is a set architectural guidelines to follow when creating web services. Web services that follow these guidelines are considered RESTful web services. The criteria a RESTful system must follow begins with a client-server architecture. Separating the UI from data storage enhances the modularibility of the UI across different platforms and allows for simplifies scalability to server components. RESTful systems also must adhere to statelessness and cacheability. The client-server communication must be constrained by no client context being stored on the server between requests. Clients and intermediates can also cache responses, so responses must define themselves as cacheable or not to prevent garbage data in response to later requests.RESTful systems must be layered as well. A client cannot tell whether they are connected directly to the end server, or to an intermediary. Finally, RESTful systems must utilize a uniform interface. The constraints of the interface are as follows.

- Resource identification in requests.
- Resource manipulation through representations

- Self-descriptive messages
- Hypermedia as the engine of the application state.

If all of the above points are followed, then a web service can be categorized as RESTful. In our case, web service APIs that adhere to the REST architectural constraints are called RESTful APIs.

Simulation Software

Due to the nature of this project, our team needs to design simulations based on the EZ-RASSOR and the environmental conditions that the rover will face. In researching how we plan to implement this, we decided to focus on simulation technologies that were completely open source, with a strong backing from the community.

Our team also needs software that would integrate well with ROS and any simulator we decide on would need to integrate well with that technology. Apart from what was researched, we also spoke with Mike Conroy about this subject and he informed us that the Swamp Works team uses Gazebo as their main simulation software. Given this information, we compared Gazebo against similar software and decided that Gazebo would be the best software for this project.

For the implementation of the software, our team has been looking at the various documents available on Gazebo's website as well as several tutorial videos that are available online. Our plan for the simulation software is to develop a functional EZ-RASSOR robot that we can test on and design a set of environments onto which we can test the rover's operations. Using these configurations, we will be able to fully test out the ROS software before having to rely on a hardware test which would take significantly more time.

Robot Operating System (ROS)

The complexity of our project prevents us from using a simple robot with a raspberry pi (for example) where we can hardcode all of our software. While having all of our software and functionality being made specific to one physical robot might make sense, it would end up jeopardizing the modularity that we hoped to achieve in this project. Therefore, we found that our project worked best in a messaging environment. Specifically, an environment where set of robots could execute commands based on instructions it received from a central

processing module. This becomes extremely apparent when discussing Swarm functionality; even if all the robots did all of their calculations locally they would still need to communicate with each other. Additionally, using a messaging system allows us to even create future versions of the EZ-RASSOR without having to change the core of our project.

A standout choice for this endeavor was the Robot Operating System (ROS). The reasoning behind this technology is listed in the next section titled “Technologies Considered and Technologies Used”. This current section simply highlights our need for a robust messaging platform, a need which was identified practically from day one of this project.

Unified Robot Description Format (URDF)

The Unified Robot Description Format is a time-tested way to describe robots in ROS. It is a xml formatted file which contains robot related information like Links – geometry figures such as boxes, cylinders or sketchup models, and Joints – the information how the links are connected to each other, e.g. if they are fixed or continuous and in which state they are. There also is a companion macro language called xacro which reduces xml code length, and enables the usage of variables (for scaling a whole bot).

It may look simple, but this robot took a bit of maneuvering. It has 4 movable wheels, 2 movable arms, and 2 rotating drums and with soon to have custom colors and will drive around in simulation The following render was created with Rviz. Rviz (ROS visualization) is a 3D visualizer for displaying sensor data and state information from ROS. Using Rviz, we can visualize the current configuration on a virtual model of the robot.

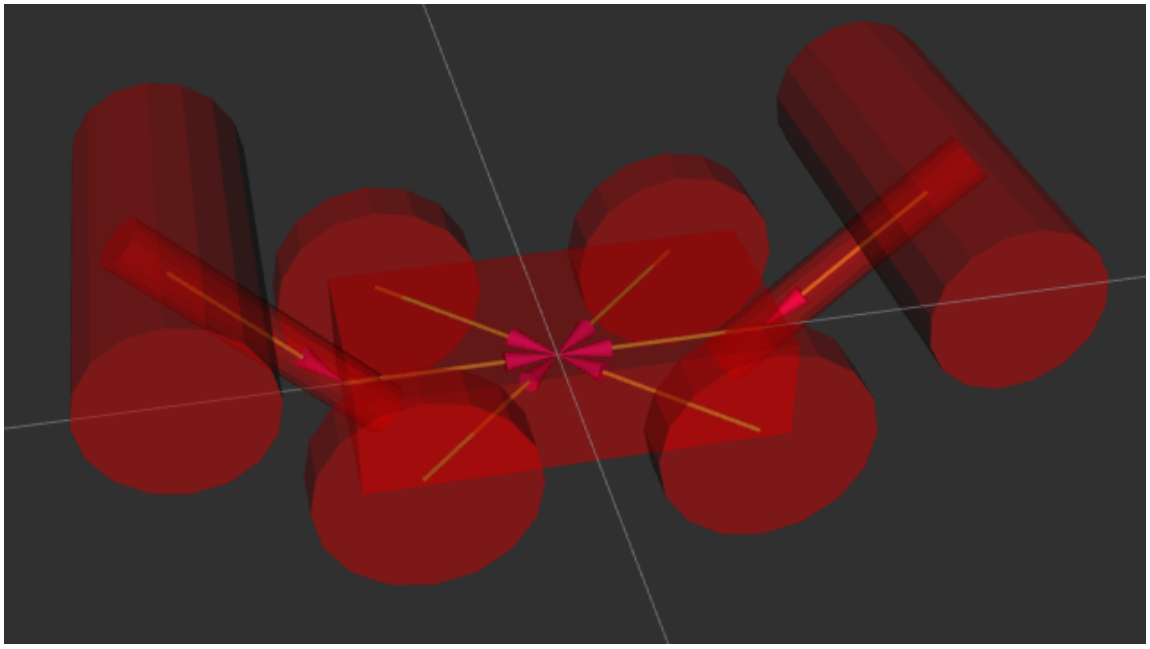


Figure 3 | Our latest URDF model of the EZ-RASSOR

While URDFs are a useful and standardized format in ROS, they are lacking many features and have not been updated to deal with the evolving needs of robotics. URDF can only specify the kinematic and dynamic properties of a single robot in isolation. URDF can not specify the pose of the robot itself within a world. It is also not a universal description format since it cannot specify joint loops (parallel linkages), and it lacks friction and other properties. Additionally, it cannot specify things that are not robots, such as lights, heightmaps, etc.

On the implementation side, the URDF syntax breaks proper formatting with heavy use of XML attributes, which in turn makes URDF more inflexible. There is also no mechanism for backward compatibility.

To use a URDF file in Gazebo, some additional simulation-specific tags must be added. The next steps in making our EZ-RASOR model will be to add the correct sensors needed to compute our obstacle detection and orbital map. A Gazebo tag similar to the following example will be needed to implement this.

```
<gazebo reference="EZ-RASSOR">
  <sensor type="ray" name="velodyne_sensor">
    <pose>0.0 0.0 0.0 0.0 0 0</pose>
```

```
    <visualize>false</visualize>
    <update_rate>10</update_rate>
    <plugin name="gazebo_ros_velodyne_controller_${lidarName}"
filename="libblock_laser_plugin.so">
      <topicName>/velodyne_${lidarName}</topicName>
      <frameName>velodyne_${lidarName}</frameName>
      <alwaysOn>true</alwaysOn>
      <gaussianNoise>0.00</gaussianNoise>
      <updateRate>10.0</updateRate>
    </plugin>
  </sensor>
</gazebo>
```

Figure 4

Swarm Intelligence

Swarm Robotics is a field that is just beginning to bloom and is challenging the minds of some brilliant men and women. The idea is a simple one, finding a way to have relatively simple robots organize, inform, and solve complex problems as a decentralized group rather than an individual and without some God-like overlord. These goals come directly from nature. Colonies of ants, schools of fish and flocks of birds are all capable of what seem like complex organization and problem-solving strategies despite the simple cognitive ability of each individual within the group. An individual ant for example, when considered by itself, is relatively uninteresting, arguably stupid and ill-equipped for most tasks. However, a colony of ants is somehow able to organize into war parties to defeat a common enemy, can build complex structures, can use their combined bodies to create life rafts during flooding or bridges to cross large distances, and can move objects 100's of times their size by executing a common task.

Figure 5 | Ants building a bridge out of their bodies to cross a body of water.

These behaviors in insects, birds, fish, and other social organisms are the result of millions of years of evolution and biological and environmental change. So it is challenging to try to perfect these behaviors in robots in such a short time. However, for our purposes, there are some steps we can take.

First, we must define the problem. Manual control of a robot which collects materials on a planet where radio signals, traveling at the speed of light can take up to an hour to send and receive which is not the most desirable approach. Therefore, an automated solution is the preferable path. However, it is the nature of this robots mission which is the cause for concern. This robot is built to be tough, durable, and dirty and will no doubt be faced with more perilous circumstances than other robots who have traversed the same path. Should the robot become stuck, buried, or broken then the entire mission of mining these materials is at stake. Swarm Robotics is a solution to this problem. Swarm Robotics focuses on large numbers of robots working together toward a common goal, and because of this, the system requires relatively simple robotic systems to be scalable. Having large numbers of robots working on the same problem allows for small failures within the system. Should an individual robot happen to experience a system failure, the overall system can self-correct and remains unaffected. The benefits of having multiple robots executing a common goal are obvious. However, the implementation of such a system is one that is challenging

and has been the subject of countless research papers. Many of which are published by a man by the name of James McLurkin [1].

McLurkin is a well-known researcher in this field and has found some interesting solutions to many of the problems we may face as the project unfolds. For example, how to get the robots to communicate with each other given restricted computational power and capabilities. His approach utilizes research in Distributed Algorithms and Graph Theory.

Basically, every robot in the system is thought of as a node in a graph with a finite edge distance between them. This allows the group to remain relatively close together and discourages individuals to wander off. Communication from one robot to any arbitrary robot is achieved using basic searching algorithms and spanning tree concepts taught in most second-year computer science courses. This means that communication across large distances can be passed through what amounts to a network of robots. This network of robots will make mapping an environment much more efficient as they can cover vastly larger areas as a group then a single robot can on its own. Once the area is mapped, a constantly updating ledger of robot positions can be updated through the network to allow robots to avoid collisions, and more efficiently mine and offload their contents.

Figure 6 | Image of robots building a structure. These robots passively communicate to each other through the environment. Only recognizing the changes the other has made.

Individually, each robot must be informed mainly by the surrounding environment and execute its tasks in response. Returning to the ant example, the individual robot could be seen as one that is simple, and relatively uninteresting just like an ant, but the entire system must be decentralized. Although there is still much research to be done in this field, it would seem that the foundations are set such that steps can be taken to implement a swarm intelligence into the EZ-RASSOR.

Obstacle Detection and Mapping Overview

The EZ-RASSOR robot must be able to successfully navigate through terrain with the limited resources it has. Since the robot will be a simulation of the real one that is to be used in outer space, there will not be any onboard GPS system to track where it is or where it has been. This will mean the robot will need to rely on its other sensors to be able to navigate the terrain and keep track of what it has navigated.

To accomplish this there are a couple of options. Ideally, the easiest way would be to use a type of radar that is used in modern-day autonomous cars, called LiDAR. High-end LiDAR uses a laser light to map out its 360-degree surrounding area. It is advantageous because the results are in real time, and they accurately produce an image that can be processed to map the terrain in 3D and take measurements of spatial features while delivering a decent amount of range. Additionally, it would be the easiest to code and work with as the mapping would be done easily, without needing to work on any other separate algorithms for stitching together separate data chunks. Unfortunately, this technology is quite a bit more expensive than other options and would require a decent amount of onboard processing power. After some research, we discovered there are drastically cheaper versions of LiDAR that only test in one direction with a limited range. It does give an accurate measure of depth though, so we could implement a program to have the robot do a 360-degree turn and get similar data, though more limited, to the more expensive models. The only problem is that the robot is slow and all the spins required to properly map out terrain would eat up a large amount of time it needs to spend digging.

If LiDAR is not possible in this implementation of the EZ-RASSOR another option is some form of traditional sonar. Sonar is a much older technology, allowing it to

be much cheaper than LiDAR. Unfortunately, sonar has much more shortcomings, beginning with its short range of detection, it does not create a 3d image, rather it is more useful for just immediate object detection. Moreover, depending on the implementation used, the sonar could have a limited angle of range, which might require more than one module of sonar. Much like the cheaper version of LiDAR, this data would only really be helpful for immediate detection of objects and would have to be used in conjunction with a camera, to fully realize the objects around it. The camera would be trained through OpenCV to be able to detect landmarks in its track and use that as points of reference to calculate distance traveled by the robot. Lastly, as a backup system because of its unreliability, an algorithm will be implemented on board that keeps track of the number of rotations done by the wheels. Knowing the wheel circumference, the distance traveled can be tracked if no wheel slippage occurs. When used in conjunction with the onboard accelerometer, we can further increase the accuracy of this approximation. Having all this data, a pretty good approximation of the robot's location can be used to internally map the unknown surrounding terrain. Software wise the map would be generated and stored in an appropriate data structure that would be easy to export and import as well as mark areas on the map with their elevation. This would allow for robots to share their own mapping of the area with other robots in the area to work together and create a more complete mapping.

Obstacle detection would go hand in hand with map generation. For immediate threats, the onboard sonar or LiDar would be used to detect rapid changes in elevation such as big hills or holes. Immediate action could be taken with this data to decide to move. For more complicated obstacles, such as being closed in on an area a map of the area would be used to retrace its steps. A common data structure used to help in decision making is a quadtree in which it would recursively be able to find what direction is best for the robot to travel in. Additionally, the camera, as previously mentioned would be used to train the robot of what obstacles to avoid. The advantage of using the camera would be that it allows for detection of an obstacle at a longer distance than the sonar would allow for. The problem for this is that it would require lots of training though OpenCV as obstacles can present themselves visually in many different forms.

What still remains to be determined is how the robot would be able to map and detect changes in the physical terrain. This would be terrain that is not possible to traverse but from it either forcing the robot to be stuck on such or from it being

extremely rocky terrain. Currently, with the sensors on the robot, it is not really possible to accurately predict for that obstacle. So one of our goals is to determine a hardware or software configuration that can handle this type of obstacle.

Hazard Detection Concepts

Solving the hazard detection problem has two main challenges associated with it: we will not have a robot while we are developing it to test our models, and we will need a wide variety of specific terrains to train the robot on if we want it to stand a chance of being able to roam successfully. Gazebo solves both of our issues by allowing us to design our own robot from scratch and allowing us to build or import custom terrain for our robot to traverse. It even allows us to select and use a stereo camera to interact with its surroundings in the simulation.

This brings up an interesting concept. Normally people will take pictures of the real world and train a model. If we took this approach, we would have to track down the different terrains that could be construed as a hazard, photograph them, and them hope the training is effective enough for the robot to navigate in both the simulation, and then the real world. This would be a problem because the past models had specific cutoffs for the slope of a hill that it can make it up. We want our robot to be able to tell when a slope is too steep, which means our computer vision model will have to be trained on slopes that steep. With Gazebo though, we are able to develop the exact terrain we need. This means that we could develop our own custom terrains we want to train the robot to view as hazards and then immediately see how the robot responds to similar terrains in the simulation. So instead of taking real pictures we could use the 3D generated terrain to train the robot in the simulation, and then have what it learns in the simulation be used to navigate in real life.

These robots are meant to function in various terrains that we don’t have access to. This means that how steep the robots are able to traverse completely depends on the terrain of the environment in the landing zone, which could easily be different depending on where the robots are being used and the soil conditions present. This means the goal of our robots hazard detection should be versatility.

To accomplish this, we plan on training the robots to react extreme obstacles such as walls and sudden drops in the simulation, because these should translate over to the real world well. To have the robots become fine tuned to their landing zones and surrounding dig sites, we will have the robots train in real time at the site. We will have them determine which slopes are accessible by giving them a range of slopes that could possibly be accessible, then when they run into one in their new environment, they will take a picture of it, and then attempt to travel over the obstacle. If they fail, they can use their acrobatics to correct themselves and label that data as failed. If they succeed without tipping over, then they can label that obstacle as accessible. Once the data is labelled, it can be sent over to the central processing station for all of the robots. It will then take all the labelled information it collects from all of the robots and train a model all the robots in that environment can use to navigate more successfully in the future. This model, combined with the model generated in the simulation will allow the robot to avoid major obstacles like cliffs, and to learn which of the minor obstacles it should avoid depending on its new environment.

These robots are meant to be mining for resources though, thus their main functions are not mapping and terrain navigation. If the robots are constantly gathering data, they will not be as efficient in their mining efforts, and this could reduce how many resources are collected per robot on average, which is not ideal. The best case would be to initially generate a map and hazard detection model the robots can use for their specific environment the rest of the time they are in operation at that location. This would allow them to find the most efficient paths to their destination and mine at one hundred percent efficiency for the remainder of their time in operation.

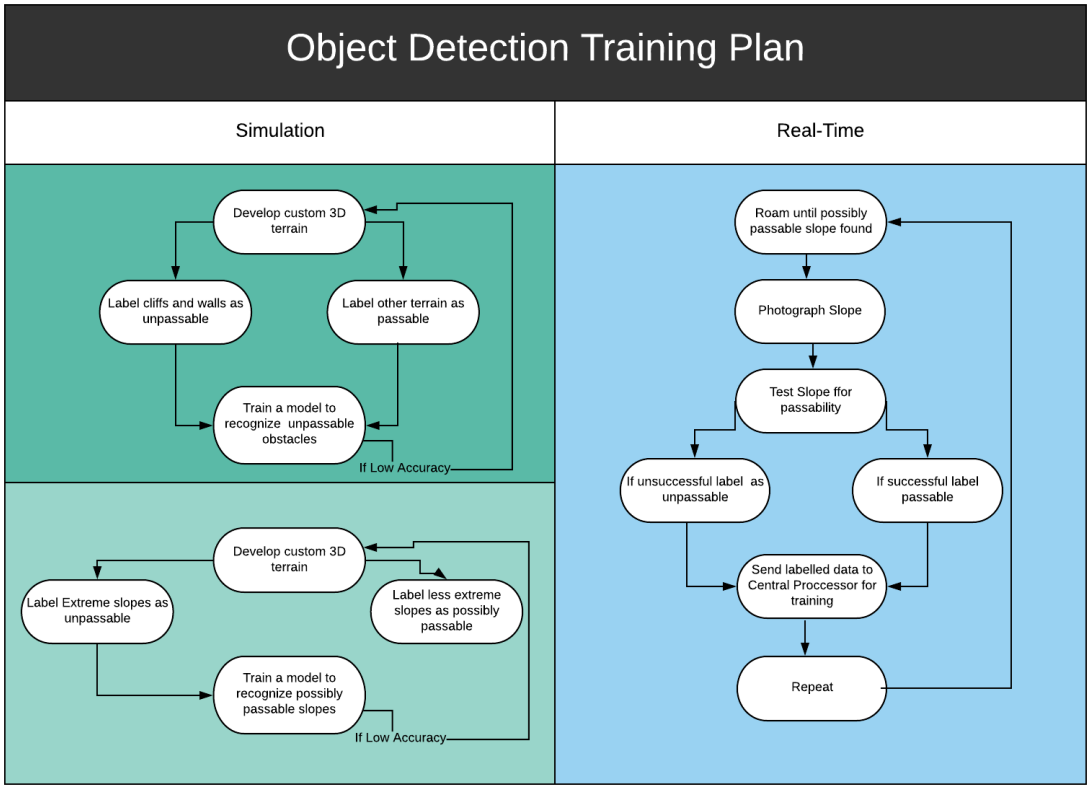


Figure 7 | Chart Representing Planned Course of Training the Computer Vision Models

YOLO Algorithm: Possible Algorithm for Hazard Detection

We plan on using the YOLO algorithm for object detection [2]. It stands for “you only look once”. This is because it looks at the entire image once while detecting objects, as opposed to breaking it down into a large amount of smaller images.

There were several ways of object detection before the YOLO algorithm came about. The main point of them was all the same though: to create a large amount of bounding boxes that can be used and checked to see if an object is present within it. These bounding boxes could either contain an object or not. The rest of the process involved figuring out if any of the possible bounding boxes actually do contain an object. This is very inefficient and time consuming. It will also lead to errors as each bounding box is being solved as its own problem, without taking into account the boxes around it.

The YOLO approach is much faster than the majority of the other methods for detecting objects that can be used. It begins by splitting the image into 7x7 grid squares. Then it uses each of those grid squares as a root and determines if that grid is the center of an object. It can generate up to two bounding boxes per grid square created, and the bounding boxes are not limited to just the grid. They can expand as far as they need to in order to encompass the object it thinks it is, if the algorithm thinks the grid square actually contains something.

It then gives everything a confidence rating to determine how likely it is that each grid box actually represents that specific object. It also compares the grids with other local grids to help determine which grid box is the center of the object. Because the algorithm bases its findings on its surrounding, this means that it can sometimes miss items if there are several of the same thing very close together.

Class confidence score = Box confidence score x Conditional Class Probability

The Equation used to find the confidence score

The main advantage of this algorithm is speed however. Fortunately, the robots are not going to be moving very fast, which means that is not a huge concern for us. A large concern then becomes much processing power we will actually have for image detection. This is a concern because in order to reduce the cost of the machine we will have to buy cheaper (and less robust) microprocessors.

The current version of YOLO uses 27 layers in its neural net to figure out what an object is. There is another version of YOLO though that uses only 9 layers and is faster than the original version. The only problem with the faster version of the algorithm is that it is less accurate. This may not be an issue though.

Some testing will have to be done to determine if the faster version of YOLO is capable of being run on a low end microprocessor, possibly turning down the framerate to make it more feasible, since their slow movement will allow for intermittent object detection instead constant detection, and if the error rate is too high to reasonably have the robots navigate without mistakes being made. If the algorithm is found to be too slow, then we will have to explore possibly making the algorithm run concurrently or finding another algorithm meant to run on low end processors.

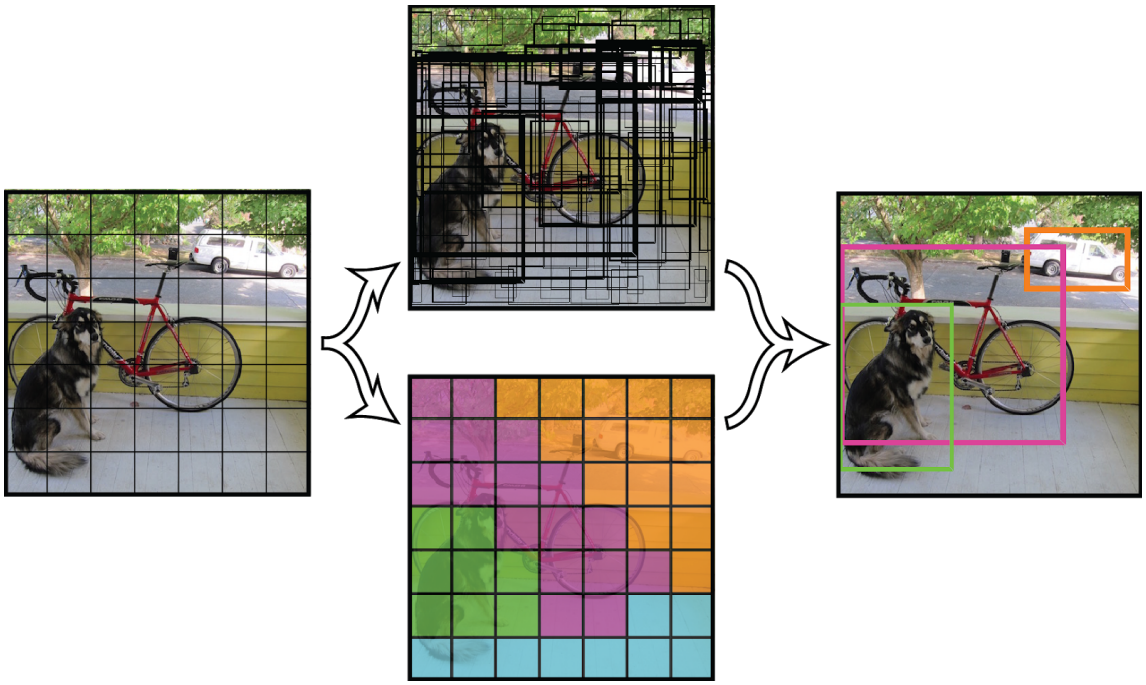


Figure 8 | The YOLO algorithm bounding boxes [2]

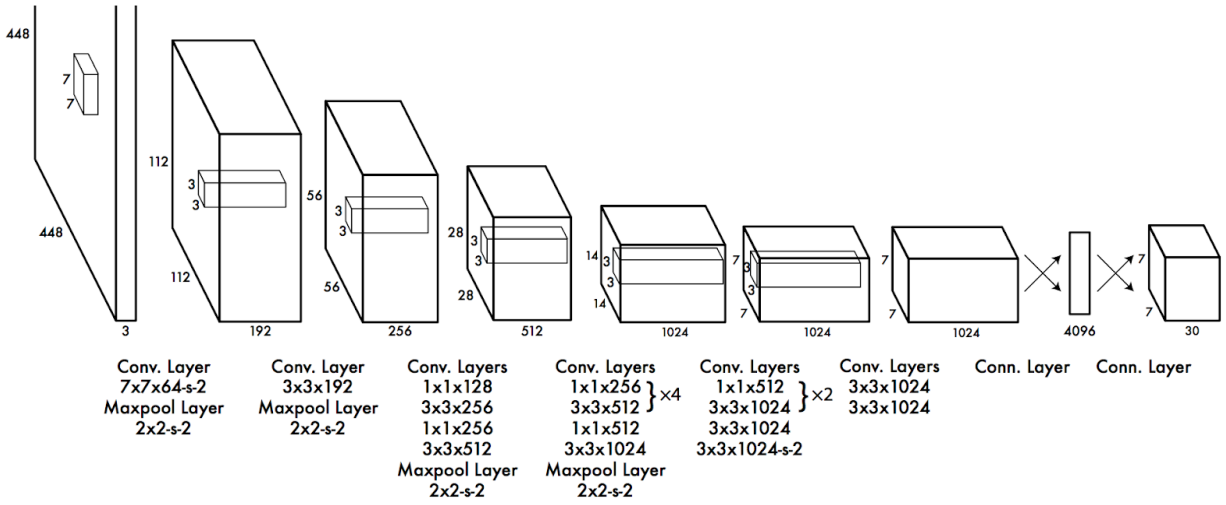


Figure 9 | The YOLO Algorithm Neural Net Layers [2]

Slope Detection: Possible Algorithm for Hazard Detection

While complete object detection would be a nice way to have our robots interact with the world, we may run into the problem that the microprocessors on the EZ-RASSOR will not be powerful enough to handle navigation, mapping and object detection all at the same time. This means we must figure out a way to reduce the processing load when it comes to hazard detection.

This is where Slope Detection comes into play [3]. On an environment like Mars, there are not going to be very many types of obstacles; it will mostly just be craters and hills made out of martian regolith. This means that most obstacles can be completely determined by their slope.

We have discovered a research paper outlining the process for finding the slope of a terrain from just a single camera. This is important, because as we lower the budget for the robot, we may have to lower its complexity, and this could be one way of handling that.

The process goes through three main stages:

1. Region Extraction
2. Texture Filtering
3. Hough Transform

In the region extraction section, the goal is to remove anything from the image that could interfere with finding the slope of the terrain. This means getting rid of far away mountains and clouds. The paper uses histogram thresholding to accomplish this. The main problem with this part of the process is that it may ignore other robots and some of the terrain because it is not all completely snow like in the paper. Some tests will have to be done to see how this algorithm handles in non-arctic environments and moving robots.

In the texture filtering section, the paper references a bandpass filter that was used to determine where all of the little differences in height are. They then run a Canny edge detector on the filtered image. The images are now prepared for the slope detection stage.

The Hough transform is where the slope is actually detected. It transforms each pixel in the image into a sinusoidal line. If these lines intersect then the area there is very likely sloped the same. The more lines that intersect in that one place, the more likely there is to be a line there. The problem is a Hough transform will often times only generate one slope per image given to it. Because of this, we can separate the image into several smaller chunks. Once we separate it, we only need one slope line per image subset, this means we can use the fast Hough transform. This version only ever returns one slope line and uses a lot less processing power.

Because the robots move slowly, taking periodic pictures of their environment and generating a slope map would be a non-intensive way of navigating. The processing problems arise because this approach isn't capable of track detection or detecting.

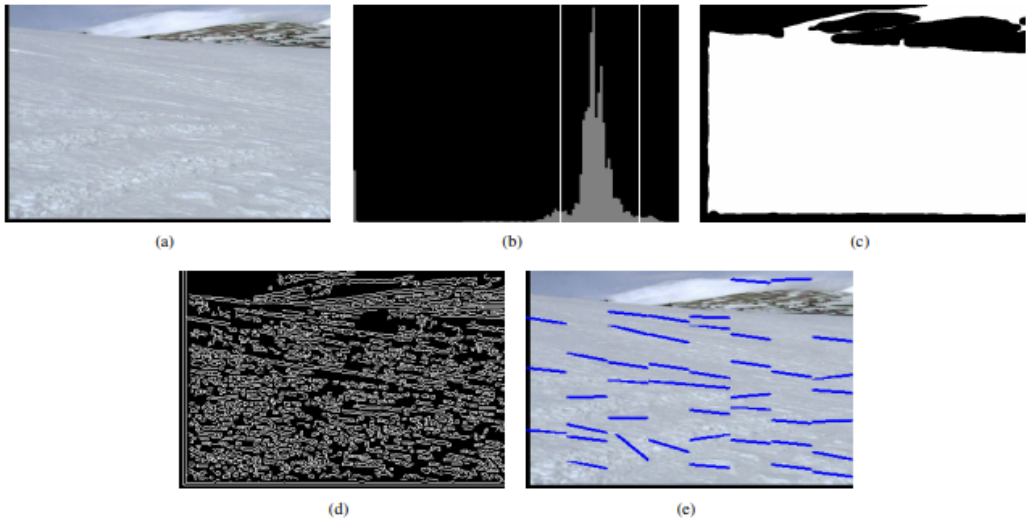


Figure 10 | *Slope map of an arctic environment* [3]

SLAM Overview

Throughout our research so far, we believe that SLAM will be the best mapping method to use for the EZ-RASSOR. SLAM stands for "Simultaneous Localization And Mapping." Using this method the EZ-RASSOR will be able to create a map for its environment and determine its location in that environment at the same time. There are multiple ways to implement this algorithm, and the way in which we implement it depends heavily on how NASA designs the EZ-RASSOR. SLAM

can be implemented with the use of LiDAR, sonar, stereo-cameras, and even mono-cameras. Whichever data sensor is chosen strongly influences how we construct our code base for SLAM, but the basic idea around SLAM remains the same.

The bare bones of SLAM consist of observing the environment, extracting landmarks, associating data, estimating the robots state in the environment, and updating the state and landmarks. SLAM uses whatever data sensor it has to observe its surroundings and determine what significant features can be used as a landmark. As the robot moves, it keeps track of its odometer, where the landmark is currently, and where the landmark is expected to be. SLAM takes into account the fact that all these sensors on their own would quickly become decalibrated from the true position of the robot. With the conjunction of all of the sensors, SLAM can use data from each one of these sensors and constantly estimate and update its position in its environment fairly accurately. (The process is comparable to a blind individual using their walking cane to determine where they are in the world.)

SLAM Landmarking

Landmarks are chosen by taking note of various features in the environment that are easily observable so they can be re-observed later. SLAM can be tweaked to look for specific objects as landmarks, such as large rocks. It is important to note that landmarking should be geared towards objects that are prevalent in the environment and objects that are stationary. Landmarking objects that move over time will cause the robot to falsely assume its own position. Though the landmarks need to prevalent, they also need to be somewhat unique from each other so two landmarks can’t be mistaken for one another. Mistaking one landmark for a different landmark will also cause the robot to incorrectly determine its location.

Visual SLAM

As mentioned previously, there are various sensors that can be used to implement SLAM. When SLAM is implemented using cameras, it is often referred to as “Visual SLAM.” For this project we will likely either use a stereo-camera or a mono-camera. Since these two types of cameras interpret data in differently, the way we implement SLAM will be different depending on which type of camera we are supplied.

A stereo-camera is really just two cameras calibrated to behave as one. The concept is similar to how humans see. Since there are two points of view, the camera system as a whole is able to estimate distance and depth the same way the human eyes can. This allows SLAM to roughly calculate the scale of its close range environment at any moment. This calculation of scale helps significantly with the localization of the robot. If supplied with a stereo camera, we will either implement an offshoot of SLAM known as “ORB-SLAM,” or another version we are currently researching called “RTAB.”

When referring to a “mono-camera” we are really talking about a standard camera with one single lense; hence “mono.” However, it is important to distinguish it as a mono-camera to avoid any confusion. Sense there is only one lens, there is no way to accurately estimate depth or distance in the environment, especially if the camera is stationary. Distance between objects can be very roughly calculated while the camera is in motion by compare the current keyframe with the previous keyframe. Though this is not very accurate and does not tell us much about the actual scale of the environment either. If we are required to use a mono-camera, the best course of action is to use “ORB-SLAM.”

ORB-SLAM

ORB-SLAM is an offshoot of SLAM that works by looking at each keyframe of the video feed and placing markers or “orbs” on all the noteworthy points in the environment. It’s similar to landmarking in regular SLAM, except we don’t keep track of specific objects in the environment, but instead an abundance of points. ORB-SLAM can be implemented with both mono-cameras and stereo-cameras [4]. The process is nearly identical, and the only difference is that with a stereo-camera, results will be much more accurate do to the fact we can calculate environmental depth and scale at any moment, regardless of if the EZ-RASSOR is moving. However when restricted to a single lens, there is no way to keep a constant awareness of the environment's scale from the mono-camera alone. In order to localize our position in the map with a single lens, we must perform some serious image processing. Luckily, ORB-SLAM is fully capable of circumventing this issue. ORB-SLAM will look at each keyframe from the video feed and distribute points or “orbs” onto the image. Each keyframe will contain a significant amount of points in it, sometimes hundreds. By taking all the keyframes and all the points within those keyframes, ORB-SLAM can create

what is know as a “point cloud map” [4]. With the use of the point cloud map, the EZ-RASSOR can now localize itself inside the environment by identifying locations it has already been by comparing distance between the points in the point cloud. ORB-SLAM is able to keep track of the same point when it moves from keyframe to keyframe using a method often referred to as loop detection and closure. Through loop detection and closure, ORB-SLAM is able to detect when a point from one keyframe is the same point in another. This keeps ORB-SLAM from mistaking the same point in two different frames as two different points. Loop detection and closure plays a vital role in the localization of the EZ-RASSOR. Since multiple points are experiencing loop closure from keyframe to keyframe, ORB-SLAM is able to localize the EZ-RASSOR with strong confidence and accuracy and help the EZ-RASSOR understand how it is progressing through its surroundings.

Figure 11 | ORB-SLAM orb/point assignment through mono-camera

Pictured above is an example of how ORB-SLAM examines an image and places points around the environment [5]. These points are often placed at places with a sharp change in color.

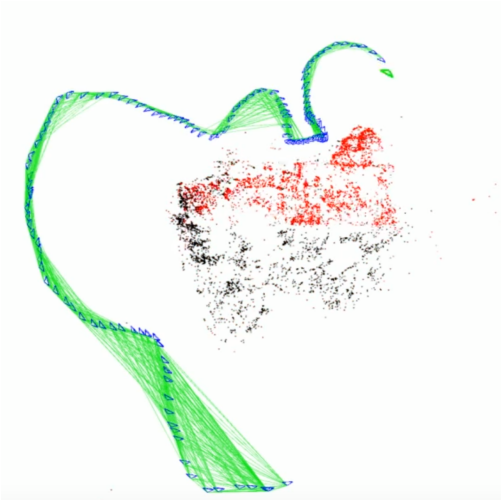


Figure 12 | Digital map of orbs/points created by ORB-SLAM

Shown above is the map ORB-SLAM constructs as it moves throughout its environment [5]. Red dots represent points that are currently in view and black dots represent points that cannot be seen. The blue triangles are past keyframes and the green lines between them represent where a loop closure has occurred. The green triangle in the top right is the current keyframe. The map above was created by walking around and office and is of the same environment as the picture from the previous page.

Figure 13 | Dense point cloud map

To help further define what it is ORB-SLAM does, we have included a picture of the dense point cloud map created when mapping the office setting [5]. If you

look closely, the picture above is comprised entirely by placing the points that were marked by ORB-SLAM into a 3D environment with the points original color. The overall process of orb slam can be shown in the following image [4]:

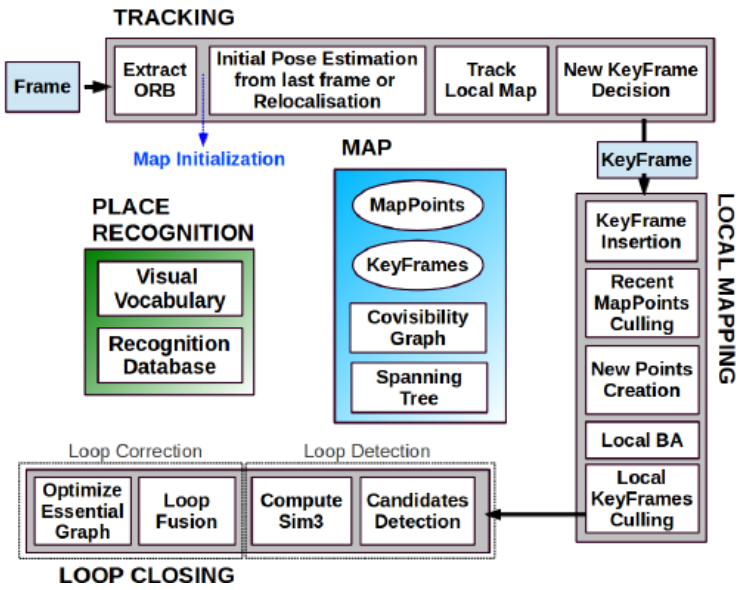


Figure 14 | High level overview of the ORB-SLAM system

LSD-SLAM

LSD stands for Large-Scale Direct SLAM. It is designed to be able to run in real time at a high frame rate. One of the interesting features about LSD-SLAM, is that it actually uses the entire image, instead of solely relying on points of interests found by looking at all the corners[6]. This way of handling the data tends to come with problems though, mostly being that it can be hard to determine the scale of the picture in terms of the world.

LSD-SLAM solves this problem by utilizing multiple types of camera data, static stereo and temporal stereo. Static stereo is where the same image is taken on two different cameras at the same time. Because the images are always going to be the same distance apart, this method is a good baseline, allowing the scale of the world to be estimated very nicely. The problem created though, is that the baseline is constant and has a specific range, which means it will not be as effective in different environments, such as extremely large or small enclosures. This problem is solved by using the temporal stereo approach. The temporal

approach takes two images on the same camera at different times, and then compares those two images to determine how far it has gone. By combining these two approaches, static stereo cameras are able to determine the scale of the world to what it is looking at and temporal stereo cameras are able to fill in the cracks in the depth map.

Keyframes are how most of the geometry is calculated in the photos that allows us to infer the depth. They are defined as:

$$\mathcal{K}_i = \{I_i(Left), I_i(Right), D_i, V_i)$$

$I_i(Left)$, $I_i(Right)$ stands for the left and right image, and D_i, V_i stands for the inverse depth map and variance map, respectively. Keyframes are a function of all of these things.

We begin with the static stereo camera data. Initially, the variance between the two cameras is calculated. Then, once a new keyframe is determined, the depth map is “pruned”. Because this system is in real-time, the keyframe will most likely be the same from frame to frame, which means that only minor changes are needed. So the depth map is “pruned” by making these minor adjustments, such as things that are no longer on camera, or that have moved.

Once we have the keyframe pruned, we take a look at the temporal stereo data. It looks at the current frame and then compares it to the current keyframe. It then fuses the frame into the keyframe, as long as the inverse depth error is small enough. Once that is done, we will want to determine the direction and distance that the camera has moved. This is done by calculating the photo residuals, where *u* is the coordinates of a pixel and *p’* is a 3D position that has been specifically modified for this equation.

$$r_{\boldsymbol{u}}^{I}(\boldsymbol{\xi}) := I_1^l(\boldsymbol{u}) - I_2^l(\pi(\boldsymbol{p}'))$$

Once this is done we will want to make sure that the images can be compared regardless of their lighting situation. For this, LSD-SLAM uses an affine lighting correction technique. It modifies the residuals to look like this:

$$r_{\boldsymbol{u}}^{I}(\boldsymbol{\xi}) := aI_1^l(\boldsymbol{u}) + b - I_2^l(\boldsymbol{p}').$$

Where a and b are found through the following minimization equation:

$$a^* = \frac{\sum_{\boldsymbol{u}\in\Omega_L} I_1^l(\boldsymbol{u}) I_2^l(\boldsymbol{u}')}{\sum_{\boldsymbol{u}\in\Omega_L} I_2^l(\boldsymbol{u}') I_2^l(\boldsymbol{u}')}$$

$$b^* = \frac{1}{|\Omega_L|} \sum_i \left(I_1^l(\boldsymbol{u}') - a^* I_2^l(\boldsymbol{u}) \right)$$

These values are then used to check the depth values to make sure that they are consistent. If everything checks out then the keyframe can be finalized. Once it is finalized it is added to the pose graph, which keeps track of all of the keyframes that have been generated and uses it to perform tracking.

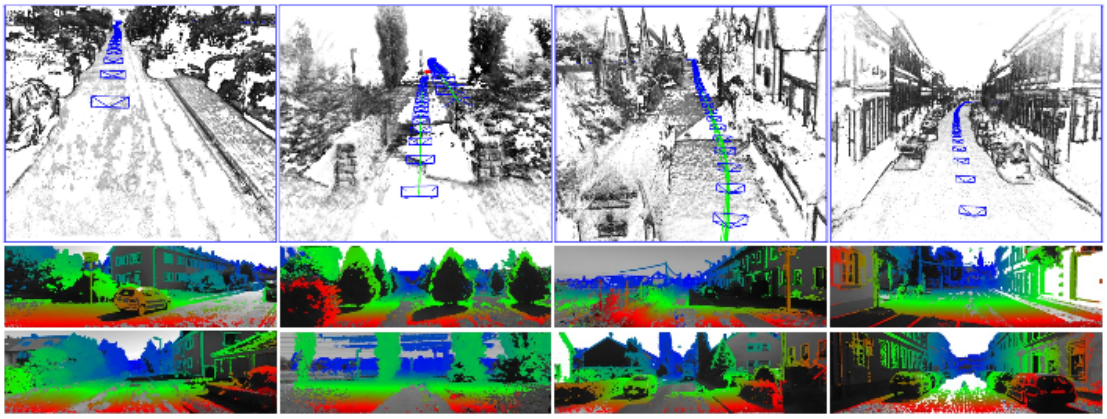


Figure 15 | The resulting point clouds and depth maps from running LSD-SLAM

The above image represents the main points of the LSD-SLAM approach. The colored pictures represent the depth map generated by the stereo cameras, with red points being the closest and blue points being further away. The point clouds above them show how the tracking works, along with the keyframe generation and storage. The blue boxes are where the keyframes are generated and the green lines are the estimated path the object took to connect the multiple keyframes.

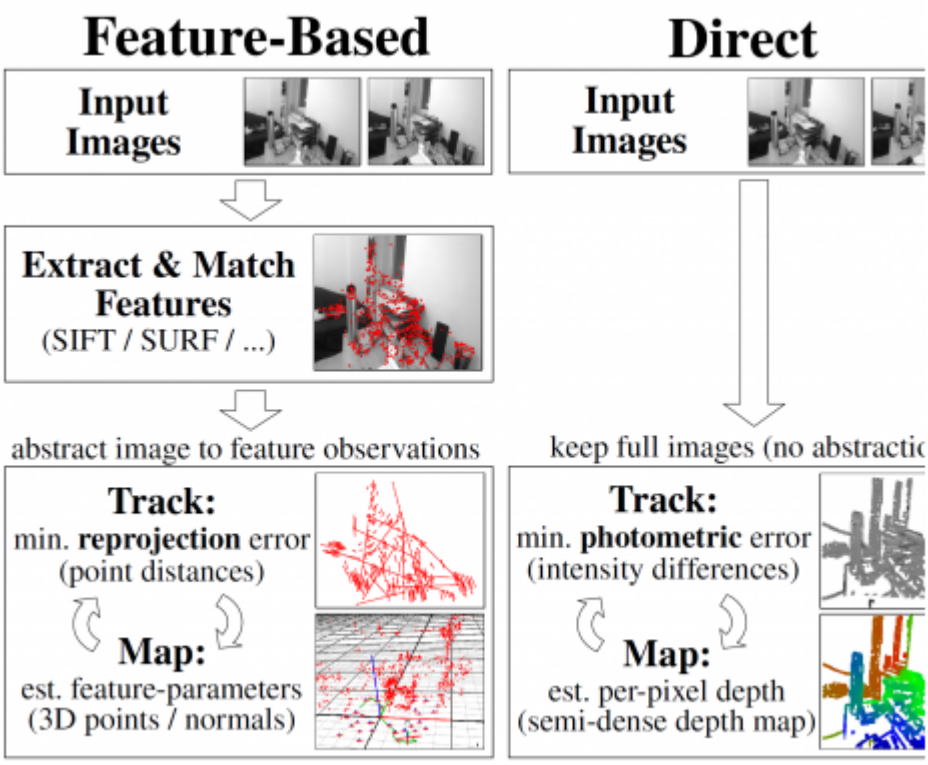


Figure 16 | Shows the difference between the way images are mapped using the ORB-SLAM approach (Left) and the LSD-SLAM approach (Right) [6]

System Visualization

By default when using ROS, the user is restricted to using only the terminal for sending commands to the robot and receiving data from the robot. However, this requires a very good understanding of ROS and the robot itself to send it the commands correctly. Even if very familiar with both, it would still be a challenge to understand what the robot is seeing and thinking strictly through reading sensor data. Because of this, the use of efficient and effective visualization can be extremely valuable for all stages of the code development cycle and during normal runtime once fully developed. During the development cycle, it can assist the debugging and testing process by efficiently providing developers the information that they need in a more intuitive way. While during the normal runtime, users operating the system can be quickly informed of valuable state information in regards of the robot and its environment.

ROS provides a software framework called “rqt” that implements GUI tools in the form of widgets. These widgets come in the form of standalone windows that

enable the user to graphically access and manipulate data from the robot and/or its environment. Some examples of the basic ways that rqt allows the user to obtain visual data is using rqt_plot which represents specified data (such as the change in a robot's position over time) with a 2D plot and rqt_image_view which can display images and videos from the robot's camera. Both of these tools can be useful for the end user to quickly recognize and understand what is happening with the robot and its environment. An example of the more advanced ways that rqt allows the user to obtain visual data is by using rqt_dep which provides a graphical representation of the dependencies within the ROS project. The use of this tool is valuable for the debugging and testing process as it allows developers to quickly see how ROS nodes are interacting with each other by simply hovering the cursor over them.

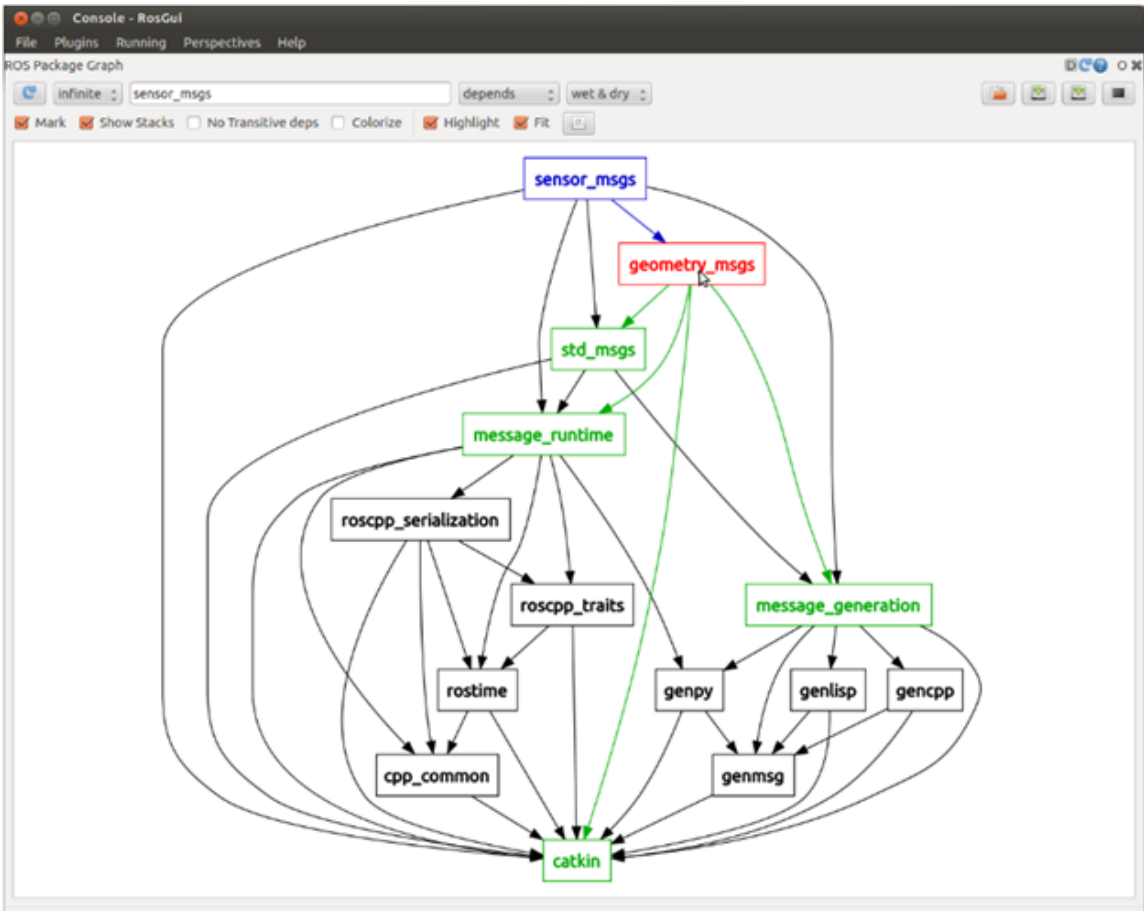


Figure 17 | Shows an example of a graph created by rqt_dep [7]

An example of how rqt allows users to manipulate data visually is using rqt_reconfigure which allows the user to view and edit parameters that have been determined to be accessible with dynamic_reconfigure. The use of this tool is

valuable for the debugging and testing process as it allows developers to quickly view and adjust the parameters of selected nodes. In addition to it speeding up the process, it allows for safer testing as it does not require the developer to directly manipulate code and potentially cause other issues throughout the project.

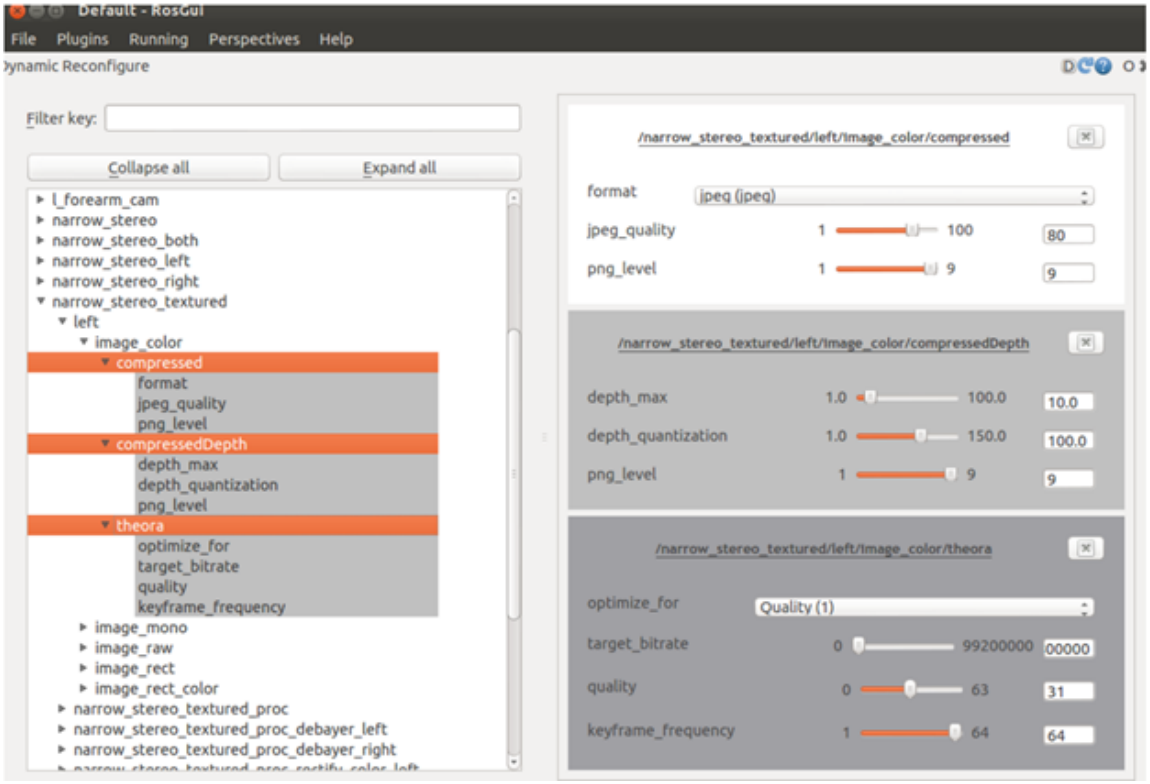


Figure 18 | Shows an example of the rqt_reconfigure tool being used [7]

Another powerful tool that ROS provides is “rviz”, a plugin that takes sensor data and translates it into a highly customizable 3D visualization environment within which the robot can be displayed and manipulated. The tool simplifies the debugging process by giving developers a much better understanding of what the robot is seeing, thinking, and doing as the 3D visualization provides a much more intuitive insight than simply reading the direct sensor data. There are two main methods to using rviz, both of which allow developers to quickly detect any obvious problems or discrepancies. The first method is motion planning, which compares the planned movement paths versus the actual movement paths. Developers can manipulate the robot into a desired pose and location and compare it to the pose and location that the robot thinks it is at.

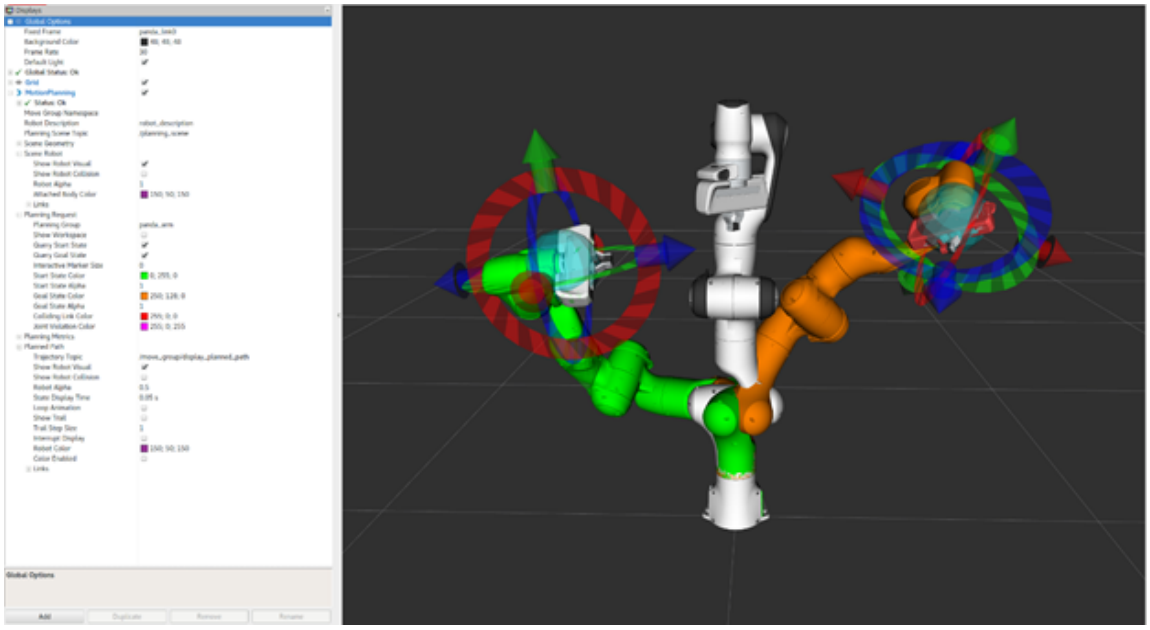


Figure 19 | An example of the rviz tool being used for motion tracking [8]

The second method is environment visualization, which combines all sensor data to create a 3D environment representing what the robot sees. This in combination with visualization markers, allow the developer to better understand the robots mapping, pathing, and obstacle detection.

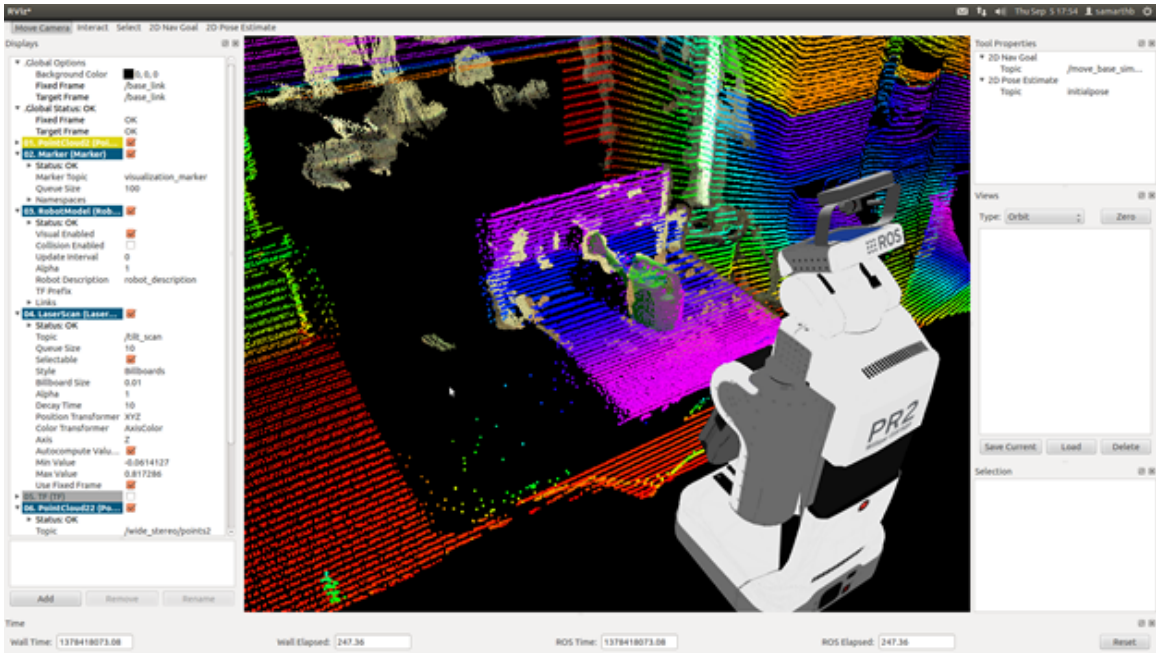


Figure 20 | An example of the rviz tool being used for environment visualization [9]

When a developer that has familiarized themselves with the ROS system and the rqt framework wants to access a specific rqt widget, they can easily do so through normal ROS commands. However, end users operating the robot are likely to not be as familiar with the system and would be unable to access these widgets. Additionally, even a more seasoned developer may find it difficult or frustrating to manage multiple widget windows at the same while working effectively if they want to see more than one type of information at the same time. To solve both of these issues, ROS implemented rqt_gui which allows developers to integrate a graphical user interface with ROS. The integrated GUI allows the user to have multiple active rqt widgets docked into a single window in a customized layout. Gathering all of the desired widgets into a single window makes the use of rqt tools easily accessible and manageable for all users.

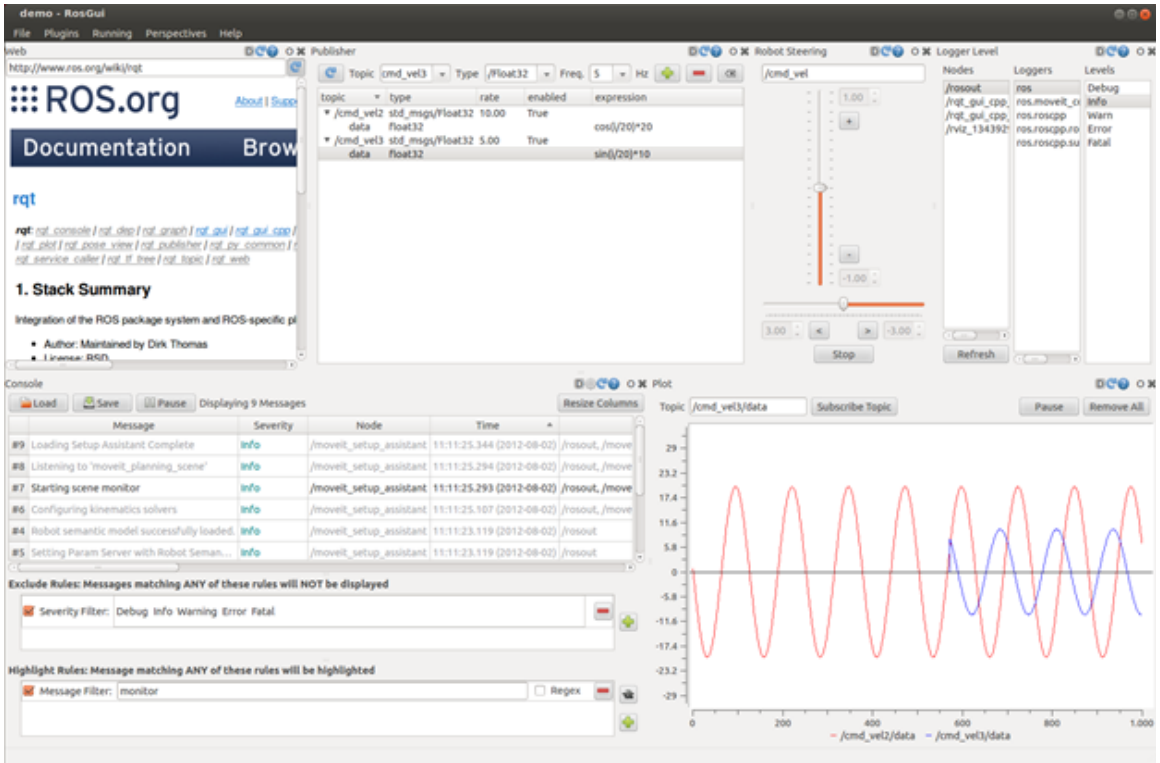


Figure 21 | Shows an example of a GUI combined using rqt_gui [10]

In order to implement rqt_gui, we must first create a GUI for it to use. In order to develop the GUI, our team will be using the open source version of Qt Creator. (Qt Creator was selected because it allows us to quickly and easily design a user-friendly GUI through the use of its drag and drop style design tools.) We expect to be adding and removing features from the GUI many times throughout

the development cycle. Because of this we wanted to be able to quickly create and edit the visual design of our GUI so that more time can be allocated to the functionality of our system. With the use of Qt Creator's drag and drop style design tools we are able to accomplish this while not needing to sacrifice the creation of a well-made user-friendly GUI.

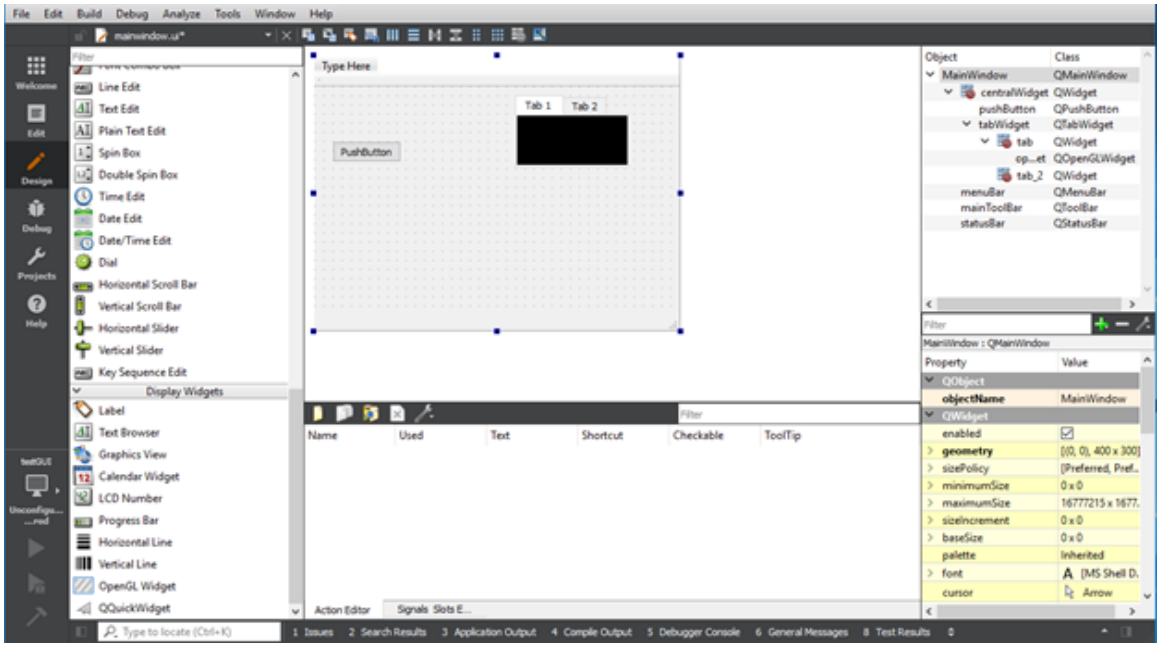


Figure 22 | Shows how a GUI can be easily designed using QT Creator

Once the GUI has been designed, it must be given the necessary functionality through code. While Qt by default uses C++, for this project we will be using Python to stay consistent with the rest of the project and because the ROS features and documentation for integrating GUIs support Python more. With Python selected as the language to use, it needed to be decided whether to use the PyQt binding or the PySide binding. According to our research, both PyQt and PySide are supported by Qt Creator and ROS with no major differences in the implementation apart from naming conventions. However, through online tutorials and communities we found the use of PyQt to be significantly more popular than PySide. Because of this we decided to go with PyQt so that we are more likely to find a solution if issues occur during the development process.

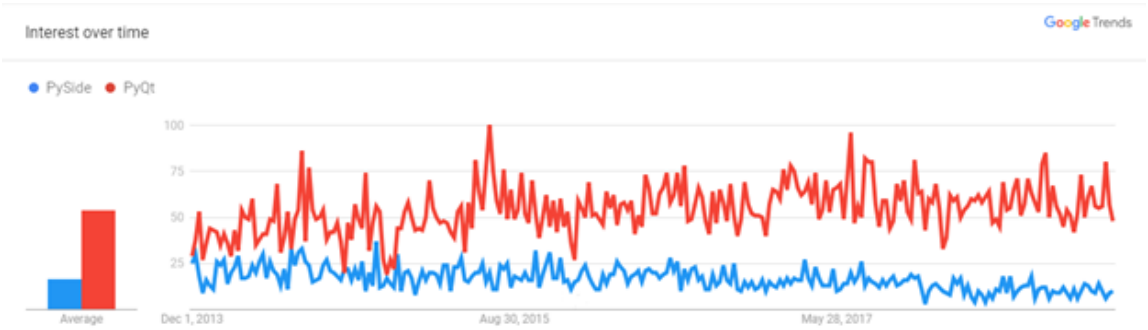


Figure 23 | Shows the popularity of PyQt over PySide using google trends [11]

GPS-Denied Navigation

A problem that arises from running multiple autonomous machines on a celestial body is management and understanding of their positions, both absolute and relative to one another. In the EZ-RASSOR system, there are multiple efforts described to handle this problem, such as on-board sensors that calculate positions based off of information available to just the EZ-RASSOR, such as the distance travelled, rotation of the wheels, and so on. Additionally, with the map generation processes that will be involved in the system, the EZ-RASSORs will be capable of maintaining position knowledge based off of points of interest that they catalog during exploration routines.

Due to the natures of these techniques, barring any other effort in determining and maintaining a rigid scheme of location knowledge, the systems depend entirely on themselves for their own certainty of position. This results in a critically high probability of experiencing drifts away from data validity. In other words, an outside source of position information gathering and confirmation is essential to ensuring the integrity of the system's positioning information. Our solution to this problem takes advantage of the satellite data, specifically imagery of multiple kinds of the target celestial body, be that the Moon or Mars. On Earth, we have a highly developed GPS system that is built upon a constellation of 31 operational GPS satellites that provide our devices with a means to determine their position using outside systems. This results in incredible information integrity - if our mobile device's positional information was determined using merely their accelerometer and had an origin point marked at the location of manufacturing for example, one would be naive to expect the location reported by the device to be reliable at any point in time whatsoever. This GPS

constellation is not yet available on other celestial bodies, but through using information from past and current orbiters, we can create a higher-level GPS implementation that results in an acceptable abstract substitute for an intended GPS constellation. This is built off of lower-level concrete data sets that were not initially intended for GPS purposes.

By taking advantage of all the data that has been acquired and repurposing it to create new systems, we will be able to provide a level of efficiency that can only be seen when creating new information systems based entirely off of already-deployed machinery and science systems. This is the intention of the GPS-Denied Navigation module of this project: to make use of what is available in terms of orbiter information that can be used for location knowledge without necessitating the deployment of further GPS-dedicated satellites to form a constellation. This may be necessary in the far future, but can have its purpose fulfilled by this general system in the meantime.

The GPS-Denied Navigation module will be implemented to take advantage of multiple leagues of datasets made available by past and ongoing lunar and martian missions, such as elevation maps, temperature maps provided by infrared observations, and visible-light maps. Each type of dataset will be used in a different manner and in a different purpose to create a high-confidence level in location knowledge integrity, by coming to conclusions on positions using multiple techniques from different perspectives rather than relying on a single technique on all datasets that would result in high confidence in low-integrity conclusions.

Visible-Light Image-Based

On the surface of the celestial body will exist the EZ-RASSOR system with its various imaging sensors, which will not only be used for object detection and avoidance procedures but also for mapping purposes. An additional pair of eyes will be provided by the images the have been created by the various orbiters around the celestial body, namely the maps that have been generated from their datasets.

Elevation-Based

By using on-board cameras and the techniques for determining elevations from a first-person perspective that will be employed on the actual system itself, information about the elevation of the surface around the robot itself can be

determined. This information can be paired up with information that comes from missions that have provided elevation data.

Heat-Based

An additional form of data that can provide information on body geography which will lead to a further-enhanced confidence level in location determination is that which is provided by orbiters equipped with infrared sensors. Given the availability of heat sensors on the robots, relationships of the temperatures between different surface features can be used as well to facilitate location determination. Similar to the use of elevation, changes and patterns in temperature over the landscape can be used and is similar to the use of visible-light information since it details the temperatures of specific surface features which can be used as a factor.

Pink's Algorithm

Oliver Pink [12] authored an algorithm designed for use by self-driving cars. It matches first-person views of roads with satellite imagery in order to determine a localization for the vehicle. In his paper, lane markings in roads are used as notable features which were previously extracted from satellite imagery and combined with live stereo automobile imagery. While the algorithm is intended for use with automobiles on developed roads with distinct lane markings, Pink notes that the algorithm can be adapted for use with any landscape that has features which can be distinguishable from both aerial and first-person imagery. For lunar purposes, craters, small hills, and small valleys can fill this role. As there is no atmosphere on the Moon, weathering effects are not present, and thus such surface features are persistent and can be extracted from satellite imagery and associated with first-person imagery even if the former far precedes the latter.

In the algorithm, certain inputs are required. These are as follows:

1. Stereo imagery from mounted stereo camera
2. Aerial imagery from orbiter
3. Initial GPS location
4. Initial heading

While the first two are easily available for this project, there is no GPS information available. Pink details that the final two items are only needed to provide a base from which to derive localization. In the case of deployed rover

systems on celestial bodies, this initial position and heading can be gathered from the mechanism from which it is deployed onto the surface, thus satisfying the data requirements and nullifying the need for GPS information.

Based on this information, it can be seen that despite the algorithm's initial intended use was for roads with clear lane markings, it is a plausible route for determining location on other surfaces, as long as there are distinct features available for extraction and matching.

Control App

To further ensure that the entire package is cost effective and distributable, we will be implementing a mobile app to control the robot when it is not in autonomous mode. To achieve this, the mobile app will need to connect the the robot via bluetooth, or WIFI. We would like to implement a mobile app controller instead of a traditional wired controller because traditional controllers can easily add unnecessary costs to the deliverable package. The main focus of the EZ-RASSOR is to be highly distributive at an affordable price point. Implementing this form of technology will make the EZ-RASSOR package cost less, and allow future developers to get experience with mobile app development if they choose to further expand on the mobile app controller. Some may argue that a mobile app controller can effectively *raise* the cost of the package because the user would already have to have access to a smartphone to use the application, but this would be nothing but a mere addition to the package. USB controllers will still be supported.

In our approach, we will be focusing on connectivity over WIFI. We are choosing this approach because of our plan to implement a mesh network. Utilizing WIFI instead of bluetooth will enable multiple machines running ROS to connect to the same network, instead of individual connections. Based on ROS' specifications, each robot will have a unique name that can be called individually or grouped together to perform swarm tasks.

To control the robot, we will be implementing customized control software using React Native.

React Native is a JavaScript framework for writing real, natively rendering mobile applications for iOS and Android. It is based on React, Facebook's JavaScript

library for building user interfaces, but instead of targeting the browser, it targets mobile platforms. In other words: web developers can write mobile applications that look and feel truly "native," all while using the JavaScript libraries already known. Plus, because most of the code written can be shared between platforms, React Native is the optimal choice when simultaneously developing for both Android and iOS. To allow the app to communicate to the robot, we will utilize the open source software called NodeOPC-UA. To allow the app to run NodeOPC-UA, the app will need NodeJS Mobile encapsulated within the app. [13]

React Native was the best choice for this development as the app in itself will not use much of the native hardware's technology. Meaning that cross platform development should be much simpler. For the most part the app will just consist of API calls which thanks to the framework will for the most part be platform agnostic. Testing will be done on both platforms as much as can be done, since iOS has certain limitations when it comes to developing for it, requiring a license to test applications. In the future we will consider the possibility of publishing this application to their respective app stores. This should not be a problem for the Google Play store, but for the Apple App Store, would require a developer account, which costs around $100 per year. This is all considering we get approval to do this from the sponsor, as this code might remain open source.

Due to previous experience, and the cumbersome nature of React Native dependencies, and compilation issues, the team will try to take the Docker route for software compilation. Doing this through a container will solve multiple issues that can often be had when compiling the React Native application in a new environment. This would make the process easier for the developers as we would be able to easily run each others changes to the application without much issue. As well as more importantly being able to allow the future developers that come into this project to be able to run and compile the code through the same environment we had. More of this will be explained and talked about in the Docker portion of the document. The amount of third party packages that are to be used in this development will be kept at a minimal. Often times in React Native many of these packages are third-party supported so it is unknown for how long they will be supported, and how well they would be documented. Docker would also help with issues in compiling that occur when dependencies fail either due to them breaking support or just not working anymore.

Another consideration that must be taken into account while developing this application is how would the connection is to be established between the mobile application and the the EZ-RAZOR hardware. It will be Wifi but how the connection is to be established needs to be determined. The end user must know what the access point looks like, and how to differentiate them, given that the situattion where many EZ-RASSORs are present. They should either be numbered or some other form. Also, once the connection is established, we need to find or identify the IP address to which the connection is going to be made as it could change depending on the situation. Having some form of dynamic addressing will be useful in this case.

Control App UML Diagram

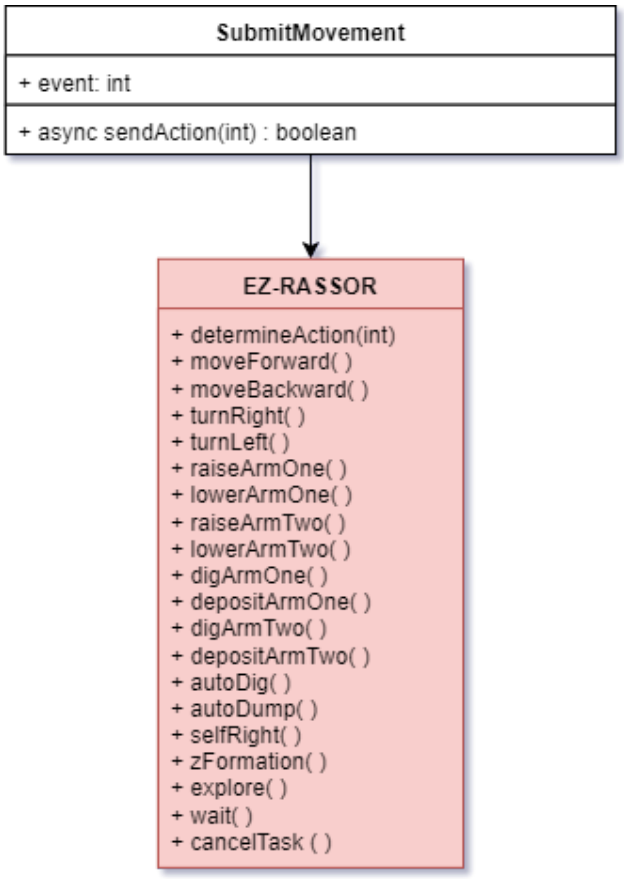


Figure 24 | Control App UML Diagram

The control app's design is fairly simple for Phase 2. When a button is pressed, *Submit Movement* is evoked and an API call is made to the EZ-RASSOR. Once the robot receives this call, It will perform this action indefinitely or until a threshold is reached (i.e the arms cannot be raised any higher). Once the button is released, the event's negative value is sent. This will let the EZ-RASSOR know when to stop performing the requested task. By doing this we allow the user to hold down a button to, for example, move forward for as long as a button is pressed. The sendAction() method returns a boolean if the request was successful or not. If not, the last made request will be repeatedly sent until success is achieved. A limitation to this design is that only one action can be performed at any given time. Meaning, the user won't be able to move and dig at the same time. We work around this issue by implementing the autonomous functions which perform complex tasks with the press of a single button.

Control App Design 1

Below we have a design prototype for the controller application. The phone would be connected to a computer hosting the ROS Master node via WIFI. Where it design currently stands, controller input will not be supported when swarm activities are active. Each button input would be translated to the following commands.

1. Turn Left
2. Move Forward
3. Turn Right
4. Move Backwards
5. Raise (Left) arms
6. Lower (Left) arms
7. Raise (Right) arms
8. Lower (Right) arms
9. Rotate Drums to excavate
10. Rotate Drums to deposit
11. Toggles Left and Right arm linking. (5 & 7 will raise both arms, 6 & 8 will lower both) Pressing this button would set both arms to the highest raised position before any additional input is registered.

AI. Toggles automated control

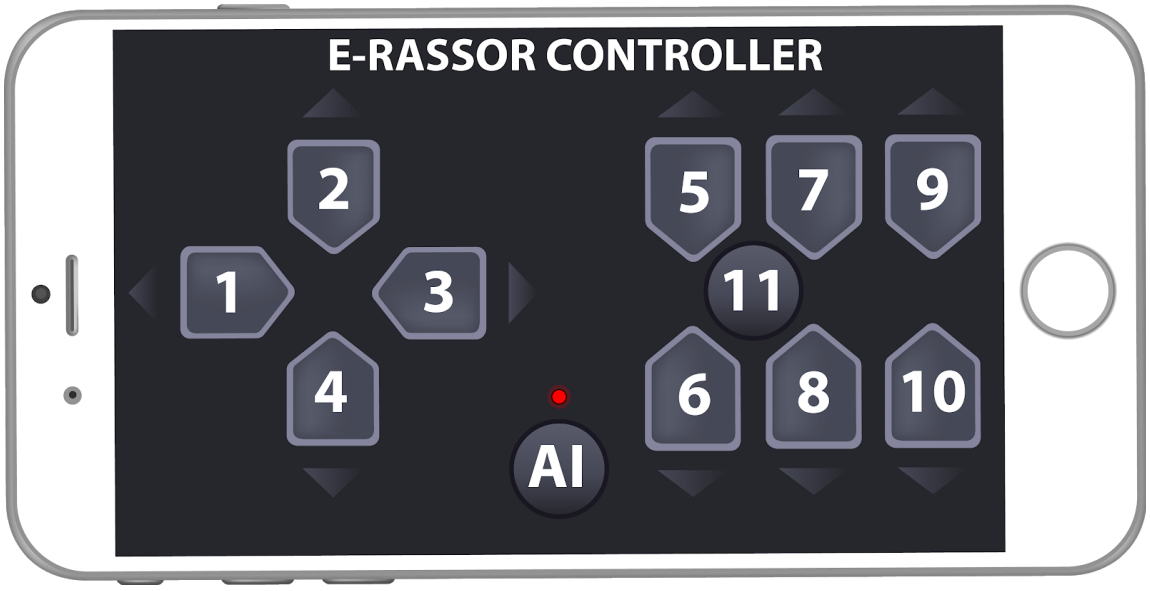


Figure 25 | Initial design for a mobile-phone sized controller application

Control App Design 2

This is our second design for a controller application. This design is meant for a tablet-sized device. In addition to all functionality in Design 1, this version would display all vital information of the EZ-RASSOR. This vital information would include, but is not limited to, EZ-RASSOR Health, EZ-RASSOR Status, if a hazard has been detected or not, regolith quality, battery level, distance traveled, the angles of the drum arms, as well as the current capacity of each rotating drum. This version would also include a live video feed of all the attached cameras as well as as a console log.

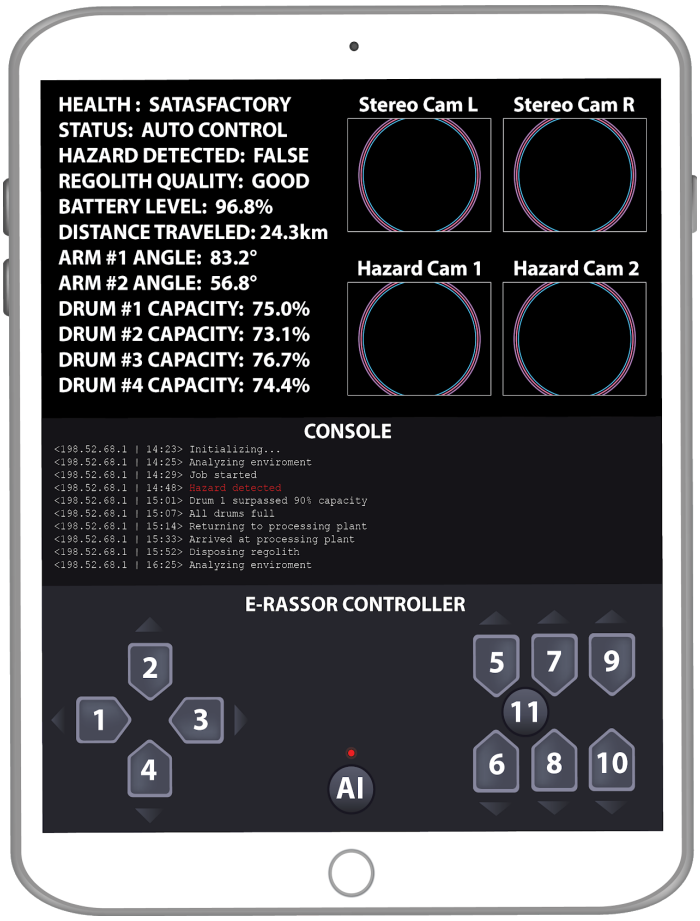


Figure 26 | Initial design of a tablet-sized controller application

Autonomous Control

The autonomous control system will encompass one of two potential control systems for the EZ-RASSOR. This system will be broken down into two separate parts: one will provide environmental awareness, bringing visual and sensor data to the system in the desired format (only used outside simulation) and the other will be the central decision making system that will build the model of the environment which will also make decisions on what actions to take given the environment. The first phase of development will focus on the central decision making system because the availability of data and resources for training and testing in a real world environment is currently limited. The simulation we will create will allow us to assume the environmental awareness is functioning and feed correctly identified environment data directly to the decision making core.

The decision-making core will have details provided about objects in the environment, the world position of the system, and the relative location and status of various components of the system. We considered the different approaches to machine learning with this type of environment and found that reinforcement learning is the best suited for training this system. We considered several different approaches for state features and action choices. One approach would be to simply pass the raw visual data and positional/velocity data from the sensors as the state features. Another possibility would be to use the stereo camera view to create a 3d representation of the environment, labeling the objects and calculating the position relative to the EZ-RASSOR body. There are also several possibilities for the states associated with these features. We could consider a highly granular approach, where each action is a single value sent to a particular moving component, or we could abstract the actions like simply choosing routines like “move to location” or “dig until full”.

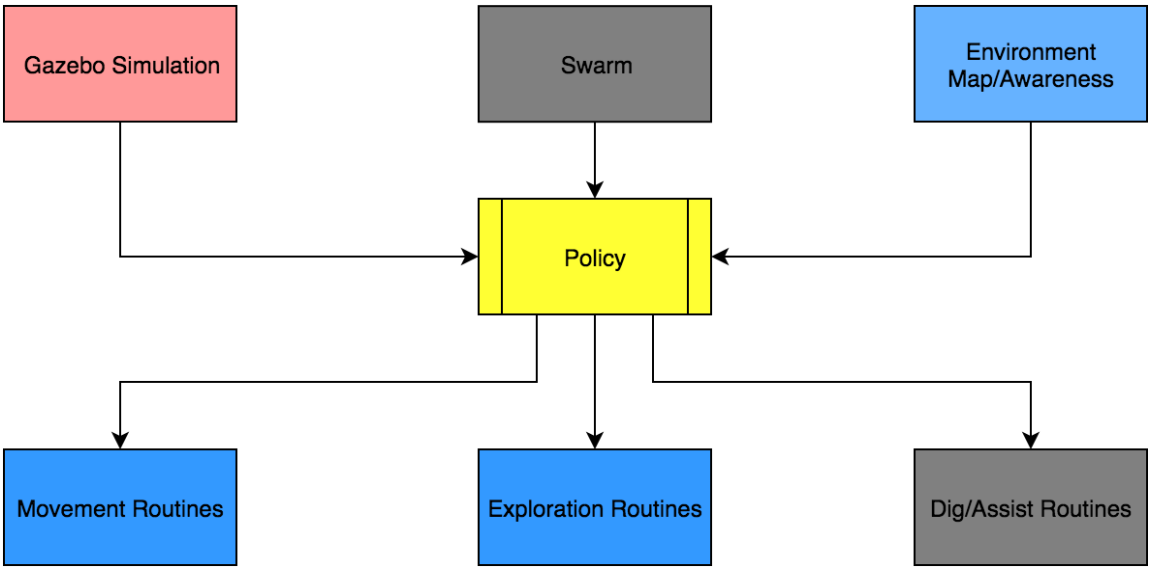


Figure 27

The utilization of SLAM (implementation and researched discussed elsewhere) has many potential implementations. We are currently considering LSD slam as the most likely version for our system, however there are some other considerations. First of which is the ability to navigate given the environmental mapping provided by the LSD-SLAM. Because of the potential for the shared SLAM mapping through swarm collaboration, we need a more robust and less noise sensitive system. Several forms of smoothing exist for this type of system such as Gaussian filtering [14] and bilateral filtering [15]. After these images have been sorted we looked into occupancy grid mapping techniques that can work in tandem with SLAM to produce a more suitable large scale navigation representation of the data.

The first approach to solving this problem would simply be to take the first and second gradient of the point cloud and attempt to determine edges in 3D that are too steep to be climbed by the EZ-RASSOR. This approach seems feasible, however it would only be computationally feasible if the EZ-RASSOR has a GPU onboard. The other possibility is to hand off the point-clouds to the homebase and have this gradient calculation performed there. While this should be possible after smoothing the data, an even better approach would be an entire object separation of the original 3d environment. Supporting research [16] discusses exactly this type of solution and proposes solutions for various densities of point clouds. Once the Gazebo test environment is developed, we will be able to

experiment with these potential solutions and provide the best possible navigation map for the EZ-RASSOR.

Sensor Fusion

Evolution has gifted us with a phenomenal computational apparatus in our heads which is capable of computing massive amounts of input from different sources and sending signals to different parts of the body to react to those inputs. Each of our five senses is capable of informing our actions in a way that can be interpreted as intelligent or rational. However, when these senses are combined, a much more accurate model of our environment is produced and our ability to act in that environment is optimized. This same idea will need to be implemented in the EZ-RASSOR.

As the robot navigates its environment it will need to know when to begin digging, when to stop digging, when to dump its contents, and when to initiate a specific maneuver such as flipping itself back over. These actions will depend on its ability to sense its environment and make informed decisions based on the data it receives from those senses. The robot is already equipped with an array of sensors, most of which are monitoring motor functions and vision. The EZ-RASSOR will need a way of using all of these sensors to monitor its status and allow it to intelligently navigate the environment and understand its current state. However, sensors are subject to variance and noise and the EZ-RASSOR is meant to be a less expensive robot, so the same cost-consciousness can be expected from its hardware. This implies that the sensors used will be less precise and subject to even greater variance and noise. Understanding this is an issue, steps must be taken to allow the robot to have the most optimal estimations of its current state as possible. The leading solution to this problem is to use an algorithm using Kalman Filters. Kalman Filters allow for indirect measurements and generate optimal estimations.

In an algorithm that uses a Kalman Filter, each state is estimated using both a state matrix and a process covariance matrix. Which represents the error in the estimate. The initial state is then processed as the previous state and for every iteration, a new state is predicted based on the physical model and the previous state. Using the predicted model and the measurements from the sensors, a Kalman Gain value is calculated and the state and covariance are updated and outputted. The algorithm then uses the newly estimated model to help predict the next state and so on. A benefit to this approach is that there are

redundancies built into the equations. Should the sensors which are helping to generate the output model of the robots environment experience unusually large amounts of noise, the values generated by the predicted model can be trusted over the sensor readings. Conversely, if the predicted model is way off due to large disturbances such as wheel slippage or obstruction, the sensors can be trusted over the predicted model.

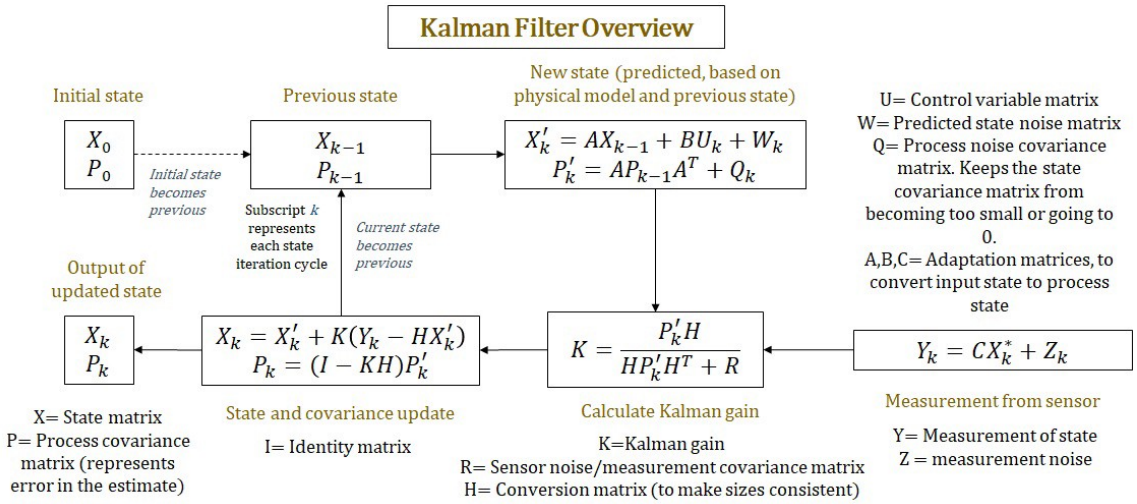


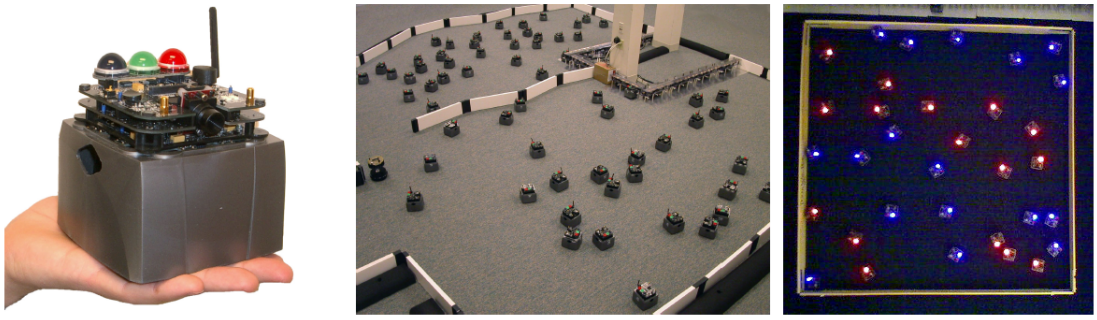
Figure 28 | The approach on implementing Kalman Filters

This approach has been proven to generate an optimal estimation of an agent's current state should system inputs be linear. However, Nature is rarely linear and it should be assumed she will not be so kind to us on another planet. Although the Kalman Filter approach is mainly assuming a linear system, there are alternatives which allow for non-linearity in the system. There are two approaches for this non-linearity. The first approach is to locally linearize the system at each iteration around the mean of the current state estimation. This produces Jacobian matrices which are used in the prediction and update states of the Kalman Filter algorithm. This approach is known as the Extended Kalman Filter Algorithm. However, this approach can be computationally expensive at times due to the calculation of some Jacobian matrices and also does not perform well with systems which are highly non-linear. The second approach, which goes by the unique name of an Unscented Kalman Filter selects a minimal set of sample points in the algorithms Gaussian distribution such that their mean and covariance are the same as the distribution. These points are referred to as sigma points. These sigma points are then propagated through the nonlinear system model. The mean and covariance of the nonlinearly transformed points

are calculated and used to create a new Gaussian distribution which is then used to predict the new state estimate.

Robot Mesh Network

In order to implement a swarming behavior in multiple robots, each robot will need a way to communicate with the other robots in the group. In most swarm robotics experiments this is generally done using an array of IR transmitters and receivers which are placed around the outside of the robot. These sensors transmit and receive data to each robot which help inform that robots behavior when performing tasks such as mapping an area, getting in specific formations, or executing a simple robot sorting algorithm. The EZ-RASSOR is not equipped with such sensors. Instead the EZ-RASSOR will have to rely on its stereo vision system and as of yet unknown system in which the robot receives remote control commands to communicate with the group. One problem that arises is one of noise in the network since all robots will be transmitting at once and any valuable information such as hazards, area topography, or important terrain information will compete and be lost in the noise. Unless new sensors are considered, any notions of a viable network of robots executing behaviors of an intelligent swarm are challenging. Assuming hardware requirements met, distributed algorithms seem to be the best approach for executing a robot mesh network. James McLurkin [1] a leader in the field of Swarm Robotics has cleverly utilized such algorithms which attempt to solve some of the problems. Each robot is represented as a node in a broadcast tree which messages can propagate with the source of the message being the root of the tree. This information can be used to give good estimates of the distance between a robot and its neighbors which allow for the topology of an environment to be closely mapped. In **figure 29** below, each robot has a combination of LEDs located at the top of its body as seen in figure 29.a. The propagation of a message through the broadcast tree can be observed in figure 29.c with the robot in the lower left-hand corner as the root of the tree and the red lights being the message sent through the network.



(a) A SwarmBot (b) The Swarm. (c) Hop distribution on robots

Figure 29

Using a structure like this will allow communication of robots who may not be near each other and will reduce the need for a computational overlord or high bandwidth communication. **Figure 30**, below is an example of a minimum spanning tree.

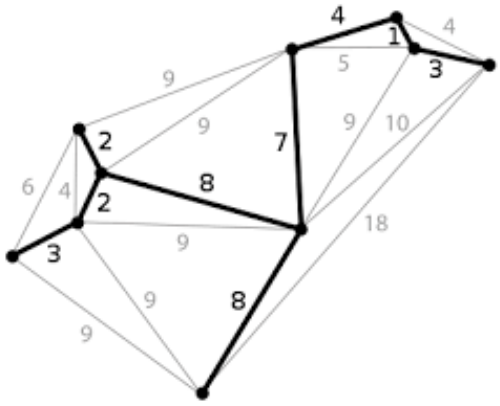


Figure 30 | Illustration of a minimum spanning tree structure propagating through a graph.

For our purposes, resource prioritization can be executed. For example, as the RASSOR gathers resources over time and the production of materials continues, certain materials may require to be prioritized. Water may be running low, metals may be needed in surplus or Oxygen for fuel may be needed immediately. Being able to send a signal to the swarm to inform them that a surplus of a specific material has been found can prove vital to rapid collection of such a material. However, this approach is not without its challenges. One behavior McLurkin found was that as the robots individual speed increased the quality of the network degrades. This would imply a need for a max speed for the robots which

may prove to be challenging considering each robots target collection rate per day.

Stereo Vision vs. Monocular Vision

Our ability to perceive depth is a gift afforded to us thanks to a set of two light detecting sensory organs known as our eyes. Each of our two eyes is effectively a camera which receives light rays which are projected from an object as photons are reflected off of that object. These light rays enter our eyes through our iris and interact with sensors in the back of our eyes.

Our brains then receive each image captured by both eyes and fuse the information in both images together to create a single perceived image with perceived depth information. This process can be recreated using a camera system. This system is known as Stereo Vision shown in Figure 31 below:

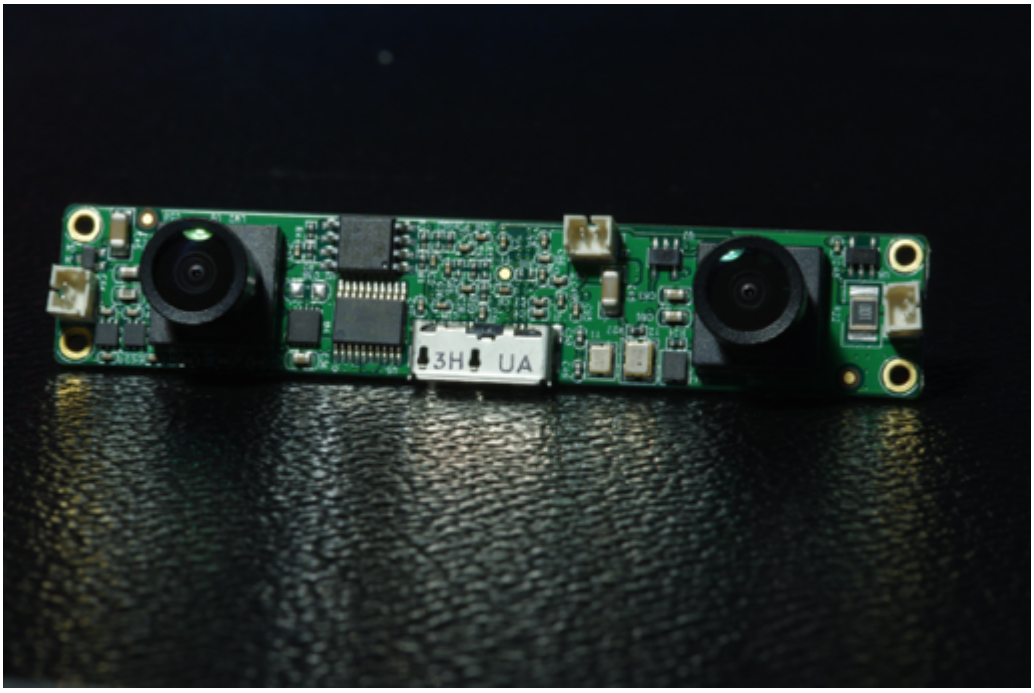


Figure 31 | Stereo camera system

To understand how such a system works we must first understand the geometry behind the theory.

In **Figure 32** below, *X1* and *X2* represent the sensors of the vision system which are receiving the light rays being projected of object *W.* *C1* and *C2* represent the focal point or iris of the vision system and *f* represents the focal distance between the cameras sensor and the focal point on the camera which the light enters. *B* represents the fixed distance or *baseline* distance between the two cameras and *Z* represents the distance to the object from the vision system.

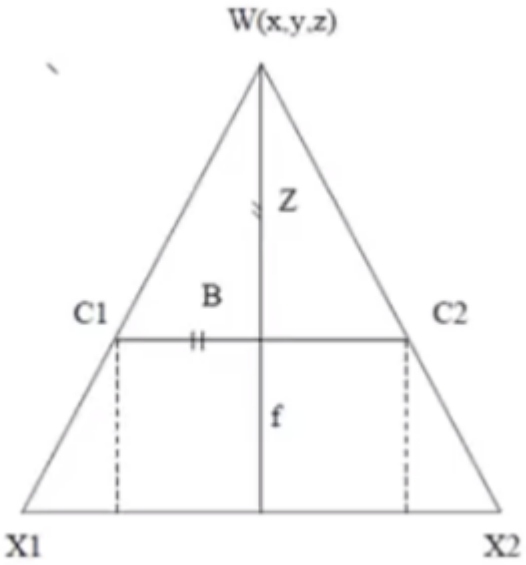


Figure 32

B and *f* can be known values as they are fixed so the relationship of such geometry can be expressed formally as follows:

$$Z = \frac{fB}{x_1 + x_2}$$

Where $x_1 + x_2 = disparity$. This disparity value represents the difference in an object's position from the left image verses the right image. This effect can be observed by anyone with two eyes if they hold their finger out in front of their head and alternate between opening and closing either eye. **Figure 33** illustrates this effect below.

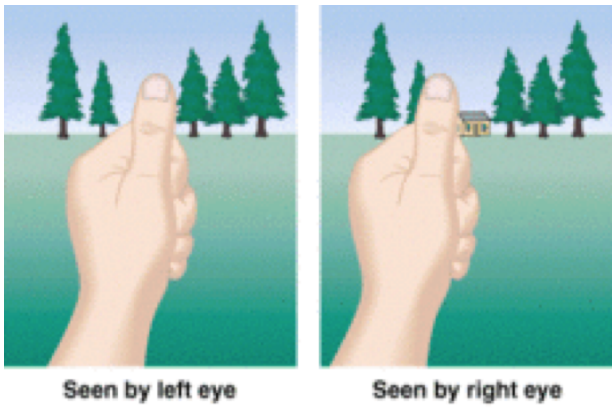


Figure 33

As the viewers thumb approaches the viewer, the difference in position or disparity value increases which informs us that objects with a higher disparity value are closer and objects with a lower disparity value are further away. Using this information computers can generate disparity maps which are an array of disparity values represented by pixel brightness in an image. **Figure 34** below is an example of such a disparity map:

Figure 34

However, before a disparity map can be produced each pixel of an object in a left image must be matched in the right image. Given that objects can exist anywhere in three-dimensional space and therefore and have a range of disparity values, searching for corresponding pixels in a second image can present many challenges. Iteratively searching through each pixel in an image can achieve successful results, however, it is computationally expensive and when this strategy is applied to a video feed, the results is less than optimal. In **Figure 35** below we see two blue squares:

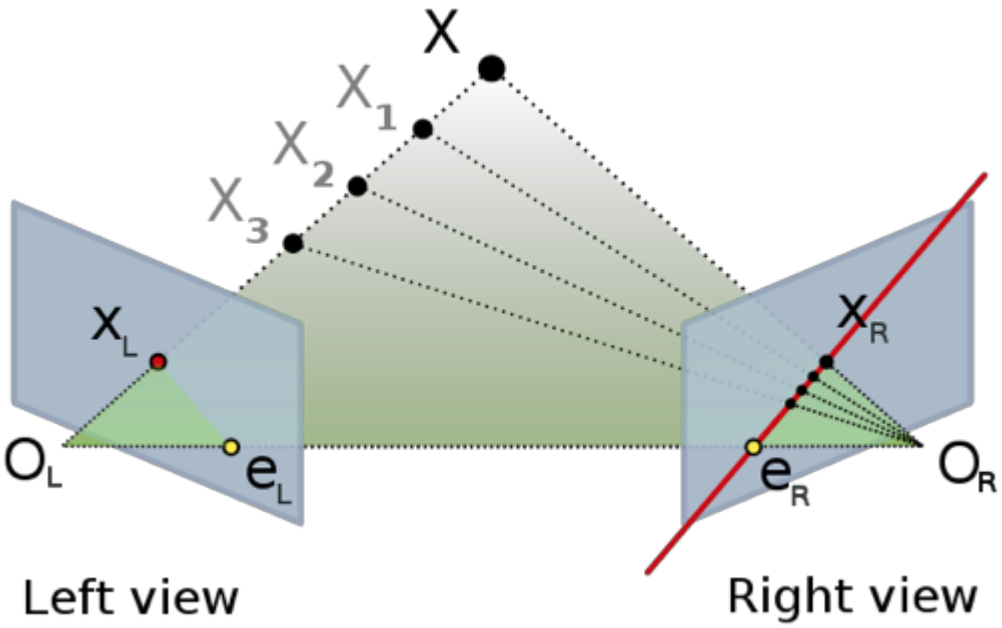


Figure 35

These squares represent the image plane each camera can perceive. *X* represents the object we are attempting to find in our right view. The light coming from object *X* passes through our left image plane and interacts with our camera sensor O_L . Since we know the distance between O_L and O_R and we know the distance from our sensors to our focal plane we can create what is known as an epipolar plane which contains the three points O_L , O_R , and *X,* represented by the green triangle. This plane will intersect with our image planes and form what is known as an epipolar line which can be seen as the red line on the right view in the figure above. If our pixel, which is part of object *X,* exists in our right image, it must exist on that epipolar line. This cuts our search space down significantly and allows us to optimize our disparity map generation.

The current version of the RASSOR is equipped with a stereo vision system on both the front and back of the robot which is used to view and understand the surrounding environment, detect obstacles, track neighboring RASSORs, and navigate both to and from the dig site. The benefits of a stereo vision system come mainly from depth perception. Giving the robot the ability to understand its environment in a 3-dimensional way allows for a better world model to be generated and more intelligent operations to be carried out. However, stereo camera systems can be challenging to work with due to synchronization problems, calibration issues, computational expense, the price of each camera, and receiving the signal feed from multiple robots each equipped with over 8 cameras can pose its own challenges. Cutting the number of cameras needed on the robot by half would present benefits in both the total cost of the robot as well as the cost computationally.

Deep3D [18] is an open source software which is capable of taking video from a single view camera and create a artificial stereo view. This technique has led to an efficient algorithm which can take both the original view (left) as well as the artificial view (right) and allow for a simple stereo matching algorithm to generate a 3-dimensional disparity map. **Figure 36** represents this approach:

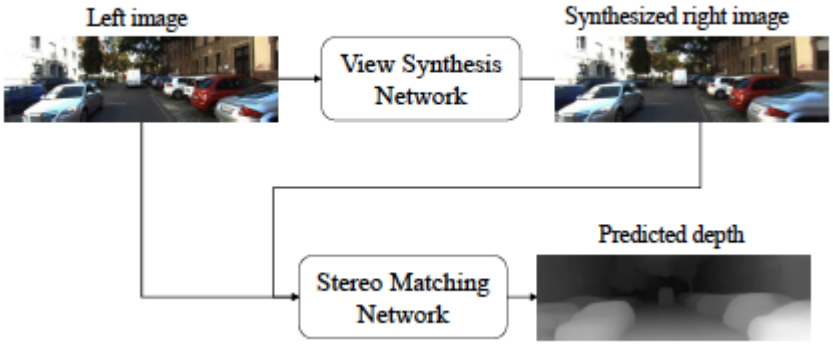


Figure 36

Using these techniques could possibly reduce the cost of the robot significantly by reducing the number of cameras needed. More testing will need to be done to determine which method best suits our goals.

Driver Station

Past Iterations

From the documentation for the RASSOR [19] and RASSOR 2.0 [20] we have been able to get a good grasp on the general goals and functions of the driver station that would be desired for the EZ-RASSOR. From the documentation of the original RASSOR, it appears that the robot was fully controlled by a human user through a remote driver station. This driver station was a software application that would stream the camera feed from the robot and read in the inputs from a gamepad controller, translating them to high level control commands that the robot would execute. The team had faced several issues with using pure manual control, especially when the bucket drums had different amounts of regolith that required constant adjustments to the arms to keep the robot balanced.

RASSOR 2.0 had many improvements to the software and was able to implement the use of an entirely autonomous mode for the robot. However, they wanted to still leave the possibility for manual control. Because of this, the driver station was separated into two primary modes of operation: supervisor and tele-operation. The supervisor mode was used while the robot was in autonomous mode and the driver station would receive the health, status, and camera stream from the onboard computer. If desired, the driver station could always be switched to tele-operation mode, leaving the robot under full manual control. Just like the initial version, the robot was controlled using a gamepad controller though more details of the robot's condition were provided to the laptop software. In addition, the user had the option to place the robot into semi-autonomous modes. These modes had the robot use its sensors to complete a small task, if this task was completed or stopped it would return to tele-operation mode. The team had reported no issues with the software, so it is assumed that there were no major issues with the driver station in this iteration of RASSOR.

Implementation

Just like the rest of the software, we intend to improve upon the driver station as much as possible for the EZ-RASSOR. The general structure of splitting the driver station into a supervisor mode and tele-operation mode will remain the

same between the RASSOR 2.0 and the EZ-RASSOR and some of the features that will be mentioned in this section may already be a part of the RASSOR 2.0. Notably, we will not have access to the software, so it is still important for them to be established. For the supervisor mode, our goal is to deliver the user with all the necessary information in the most convenient and effective way possible. Ideally, we would want to view the status of multiple robots at a single time when implementing swarm AI. However, this may not be possible without the user interface being too cluttered and unreadable. We will be providing the stream of the camera along with user friendly, graphically represented analytical data that will allow the user to quickly recognize the condition of the robot. We have already decided many features that we may display within our GUI, though even more may be added during the development process.

The GUI will be displaying the live camera footage from the EZ-RASSOR. By displaying the camera footage, the user is able to quickly see what the robot is seeing, allowing them to ensure that the robot is successfully transmitting data and may show some obvious problems occurring with the robot. The EZ-RASSOR has multiple cameras; however, we do not want to devote too much screen space to the camera footage since there is much other important information that we want to show as well. To solve this, we will be displaying the footage from one camera at a time and allow the user to change which is displayed. If given enough time in this project, we will also allow the user to display all cameras at once in a grid pattern. Another option that we would like to add if given enough time is to allow the user to have the display pop out into its own expandable window.

The GUI will also be displaying the map generated by the EZ-RASSOR robots. By displaying the generated map, the user can ensure that the robot is generating it correctly and that the robot is travelling in the right direction. The user would have the option to display the map generated by just the selected robot or a combination of the maps generated by all of the robots. If possible, we would also like to display the positions of the other online robots within the map. Just like the camera footage, we would like to give the option to have them in their own expandable window.

The GUI will also display some basic information to represent the condition of the robot. Some examples of these conditions would be battery percentage, bucket drum capacity, and distance from the central hub. These conditions would be

initially be displayed in the form of text but may be translated into visual representations if it would be beneficial to the end user.

The GUI would also need to display a list of all robots currently online. This is what the user would interact with to switch the driver station to the other robots. We would also like to show the user the general condition of these robots without devoting too much screen space to it. In order to accomplish this, we would use a three-tier color coordinated system that designates the condition of the robot as fully operational, operational, and non-operational.

The GUI will also display an information terminal that will be outputting any and all state changes of the selected EZ-RASSOR. It is likely that these messages will constantly be outputted and be difficult for the user to read all messages as they are outputted. This will be fine as it is not intended for the user to read all messages but instead scroll through the info terminal to get a detailed explanation of how the ROS nodes are interacting with each other whenever the robot is behaving in an unexpected or undesired way,

The GUI could display both implementations of the rviz render. Just like the camera footage we will avoid devoting too much space by allowing the user to switch between the motion planning and the environment visualization.

Depending on how the autonomous operation of the E-RASSOR is handled, the GUI will display a task list of what the robot has completed, is currently completing, and is planning on completing. Due to these tasks only being assigned during autonomous mode, the task list would only be displayed while in supervisor mode.

Lastly, the GUI can also show an IMU sensor display. This display would represent the orientation of the selected robot at the current moment. The display would consist of a cube with the top, bottom, and sides being different colors to make the orientation of the robot immediately clear to the user.

For the tele-operation mode, our goal is to hide the system complexity from the user, while not restricting their control over the robot and improving the overall usability. It is important to keep in mind that our user interface must be able to give an overall vision of the operations, full control of the system, handle and acquire data, and accomplish all this while being as intuitive as possible. At this

moment we don’t see any problems with the use of both a gamepad controller and an application running on a laptop, so that will remain to be our primary implementation of the driver station. During tele-operation mode, the user will be using a gamepad controller to have full manual control of a single robot at a time.

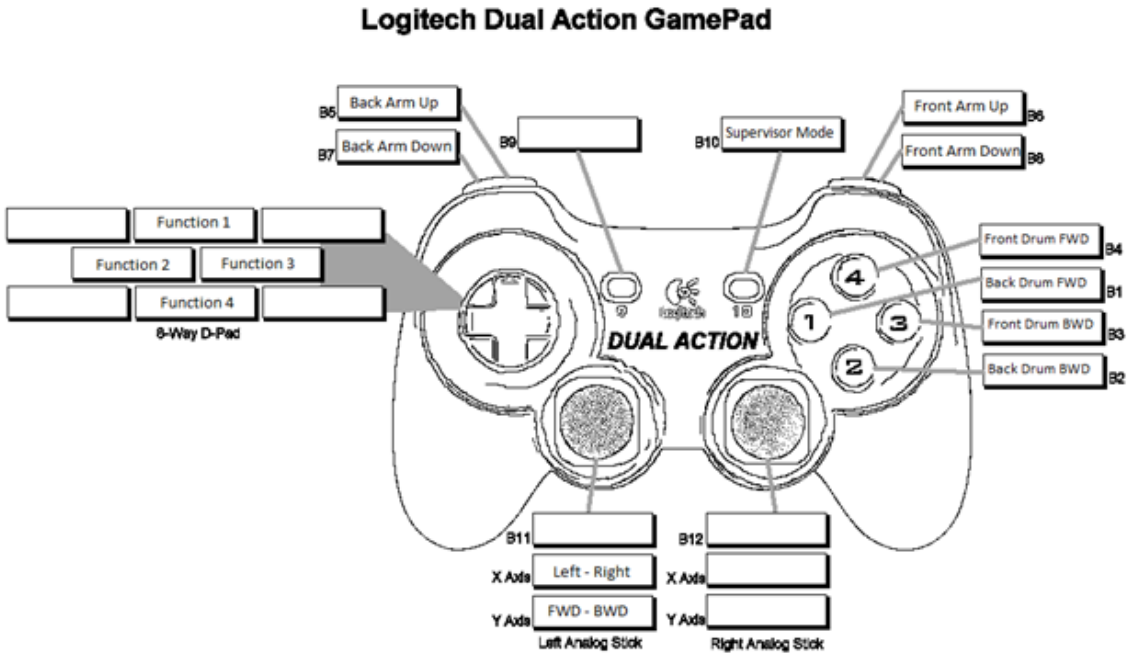


Figure 37 | Proposed gamepad controller layout for the EZ-RASSOR

The button mapping of the controller will likely go through multiple design phases as we try to make it as intuitive and user friendly as possible. Due to the amount of inputs required and the number of buttons limited by the controller, this has posed a difficult challenge. At this time, we are quite satisfied with the current layout for the controller, though it is not perfect. We believe that most of the button mapping is in ideal locations that will be easily learned by new users. A beneficial feature of this layout is the ability to support up to four different semi-autonomous functions in an ideal location. By having these functions mapped directly to the controller, we can save valuable space within our GUI and give the user full control through just the controller instead of requiring them to interact with both the laptop and controller. Additionally, we still have one button, one analog stick, and both analog buttons open for any functionality we may want to add later. Our only concern with the layout is the control of the drums for

each arm. The driver needs to be able to independently turn the drums of both arms forwards and backwards. Because this requires four inputs we decided to map them out to the four buttons on the right side of the controller. An alternative approach was to have the drive use a combination of inputs to select the drum and to then select the direction that it would turn. However, we believe that separating them into four separate inputs would be more intuitive to new users. While this is not necessarily ideal, we believe the user should still be able to get comfortable with it quite quickly. We may consider experimenting further with the button layout but for now are quite satisfied with how it is. If given enough time, we may even allow users to change the button layout to what suits them best.

In robotics, the two most popular driving layouts are arcade and tank. With arcade drive, all movement is controlled with a single analog stick; moving the stick along the y-axis will make the robot drive forward or backwards and moving the stick along the x-axis will make the robot turn clockwise or counterclockwise in place. With tank drive, movement is split between two analog sticks; the y-axis of the left analog stick will control the left-side drive train and the y-axis of the right analog stick will control the right-side drive train. To move the robot forward and backward with this layout the user must push both analog sticks in the same direction on the y-axis; to turn the robot in place the user must push both analog sticks in opposite directions. Arcade drive is seen to be more intuitive to most, but tank drive allows for faster precise movement of the robot. We had originally been leaning towards using tank drive because of its speed but decided to go with arcade drive. This decision was made in order to simplify the input and keep it consistent through the driver station, mobile application, and autonomous operation. In order to further simplify the input, we will be limiting the analog stick to only detect forward, backward, turn clockwise, turn counterclockwise. To make it easier for the user, these inputs will have a range such that 46°-134° will be considered move forward, 136°-224° will be considered turn clockwise, 226°-314° will be considered move backward, and 316°-404° will be considered turn counterclockwise. If the analog stick is at the angle of 45°, 135°, 225°, or 315° it will be ignored. Additionally, we decided to have the input only take Boolean values such that the motors go at full power when input is detected and stop once no input is detected. During testing we will consider shrinking down the ranges and decreasing the sensitivity to see if it is more user-friendly.

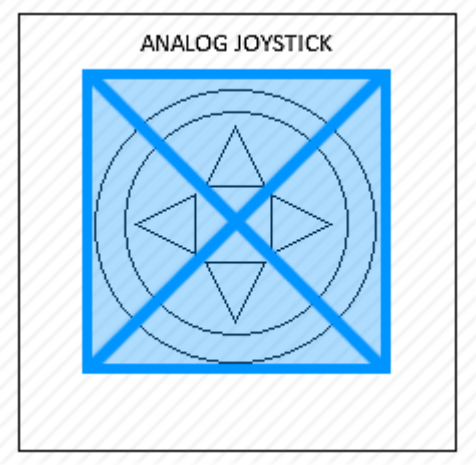


Figure 38 | Shows a visual representation of the areas that the analog stick detects input

Alert System

While in supervisor mode, we hope to be able to display the necessary information of multiple robots at a time, however, depending on the number of robots implemented in the swarm AI, it might not be possible to display them all in an effective way. Additionally, when a robot is in tele-operation mode, the driver station will only display the information from that specific robot. This can potentially be a major issue because a robot not on display may be malfunctioning without the user's knowledge. If the malfunction needs immediate attention, the situation can get severely worse if not noticed quickly enough. Therefore, we need to figure out a way to immediately notice potential malfunctions from any robot without being able to view all of the robots at once.

We've come up with a solution of implementing an alert system though it is currently deemed to be a stretch goal if given enough time due to its complexity. The system will attempt to predict what the robot should be doing or seeing and checking if that is what it is actually doing or seeing. If an unexpected event is detected, it will alert the user which robot it occurred to and a quick message of what the event is. The methods used to detect these events would be completely different depending on the event so there is no general implementation that we can apply. For example, detecting an ideal excavation site requires advanced robot vision while detecting malfunctioning motors requires checking their encoder. We would plan on implementing three types of alerts: notifications, warnings, and critical errors. Notifications are successful events: the user does not need to know these events as the swarm AI would be able to handle them on its own, but we think it would be useful for the user to still be made aware of them

occurring. Warnings are unexpected events that may or may not be an issue; it is suggested for users to quickly check these events to ensure that the robot is performing properly. Critical errors are unexpected events that would require immediate attention by the user; when alerted the robot may need tele-operation, maintenance, or retrieval. If we are able to implement the alert system, we would most likely use rqt_console to display them. Though, we would need to conduct further tests to see if it can monitor multiple robots at the same time.

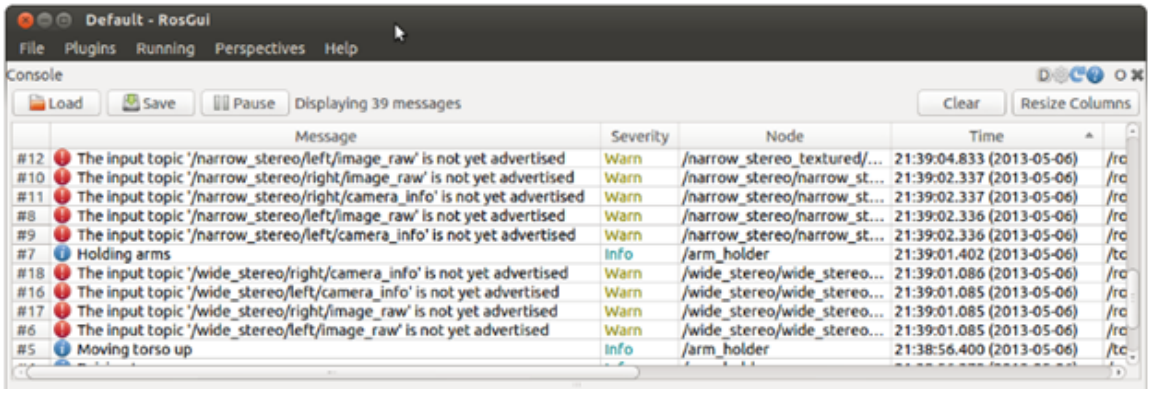


Figure 39 | Shows an example of a console window that was generated by rqt_console [21]

Potential Notifications:

- Ideal excavation site detected
- Robot fully recharged and redeployed

Potential Warnings:

- Camera not seeing what is expected
- Deviation from the task at hand
- Robot farther from the central hub than expected

Potential Critical Errors:

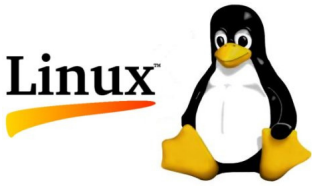
- Battery dead
- Battery too low to return to base
- Motor malfunctioning
- Unable to self-right
- Unable to move

Technologies Considered and Technologies Used

Operating Systems

This project is not intended to be cross-platform. Instead, it is being built on a particular stack of software, and each layer of the stack is significant. Perhaps the most important layer is the operating system. This project needs an operating system that is fast and efficient but is also usable, familiar to students, and ideally free. The OS must run on lite hardware like the Raspberry PI, but it must be capable of interfacing with external hardware such as wheels and cameras. The OS cannot be confusing to future contributors, as one goal of this project is to span across multiple years. For this reason, Linux and ROS will be used in combination to run the EZ-RASSOR.

Why Linux?



Linux has been chosen as the primary operating system for this project. Specifically, this project will run Raspbian on the RC car and Ubuntu Server 18.04 on the main board of the EZ-RASSOR. Why Linux instead of Windows Server or something else?

Linux has a lot of amazing features that make it the operating system of choice in this situation. First and foremost, Linux is extremely user friendly, as both a desktop operating system and in server settings. Linux is used in the vast majority of servers around the world because of its suite of server tools, incredibly strong scripting and terminal environment, and its developer friendliness.

Second, the team is quite familiar with Linux, which makes it easier to use than a different server/embedded system. Using Linux will also be good for students, as it is much more likely that computer science/engineering students have some experience in a Linux environment than in some other custom/lesser known

environment.

Third, ROS runs natively on Ubuntu and struggles to perform on other distributions, let alone other operating systems. This makes Ubuntu the go-to choice for an on-board device, as ROS is crucial and central to the design of this system. Additionally, Linux is free and open source. A free, open source environment is important for this project's success as this environment will avoid extra fees and potential legal implications for students who wish to learn about the EZ-RASSOR. The easiest way to disseminate information is to make it free and open source to the public.

Finally, using Linux presents a teaching opportunity to educators to help students learn more about this operating system. Linux is incredibly useful for development and is one of the most popular operating systems in the world, yet, due to low desktop adoption, many potential software developers and engineers do not experience Linux before getting to college. Knowledge of Linux and Bash Shell are useful and marketable skills, so more exposure to these tools in high school is beneficial to students.

Why ROS?

ROS stands for the Robot Operating System and is an open source, middleware robotics system created in 2007 to make robot programming easier and more standardized. It works by creating a "ROS graph" of "ROS nodes," individual programs written in C++ or Python that control and interface with an individual piece of hardware, or that contain a piece of software logic. Each ROS node can function independently of any other nodes. Nodes can communicate using standardized or custom "messages" that are sent between different nodes using the ROS graph. These messages contain data such as movement commands, strings, logs, etc.

ROS is a perfect fit for this project for a few reasons. First, it is free and open source like the rest of this project. This project will use Linux, which is free and open source. Students and schools tend not to like to spend large amounts of money on software (nobody does, really) and open source code is necessary to learn how a system works from the code, up. By using ROS, any of the core

functionality of ROS can be read and understood by students, as well as all external nodes that may be included to run actuators/cameras/arms/etc. Additionally, as mentioned before, ROS runs natively on Ubuntu/Raspbian, the two operating system choices for this project.

Second, ROS has a large, active community and user base. This has several major benefits. A healthy community means vast forums of knowledge where users have asked and answered questions, found solutions to known problems, and brainstormed on potential fixes together. This amounts to a vast support network that the team can use to troubleshoot any issues that may arise during development. An extensive user pool also means that most major hardware already have clean, efficient ROS nodes that merely need to be installed and turned on to use. This means that the logic for driving, arm movement, camera operation and even much of the necessary robot vision for this project has already been written and extensively tested on various systems. It essentially gives the team a huge tool chest to pull from. Not all the tools in the chest are needed, but chances are that if we need a tool, we can find it with a quick search.

ROS is extremely portable, which is a major consequence of its nodes-in-a-graph architecture. ROS nodes can be dropped in or removed at will with only minimal configuration changes to the graph. This means that preexisting ROS nodes for one type of wheel or one type of camera can be swapped with other nodes for different types of hardware, so any hardware changes do not require large code refactoring. This allows for the logic behind the RC car demonstration software to port directly to the EZ-RASSOR. The only required changes will be that the correct nodes for the EZ-RASSOR's wheels and cameras are "dropped in" and the RC car's wheels and cameras are removed. This malleability is also valuable to students who want to take the EZ-RASSOR code and port it over to their own creations.

ROS integrates with Gazebo, our simulation software, perfectly. In fact, Gazebo and ROS were created in tandem by the same group of developers, and so are a match made in Heaven. This is a crucial advantage for ROS, because it allows our team to easily construct a simulation environment for our code to be tested in. Environments like the Moon and Mars are hard to replicate on Earth, but anything is possible inside of a simulator.

Finally, ROS nodes can be written in Python. Python is an excellent language that excels at enabling the rapid development of technologies. The language is not the most efficient language ever created, but as a tradeoff it is extremely easy to use, and is exceptionally capable at making complicated logic clear and concise.

Why Not MIRA?

MIRA (Middleware for Robotic Applications) is an open-source C++ framework that provides inter-process communication capabilities and tools to develop and test applications of a distributed nature. It is primarily used for robotics applications, but may find use in other fields as well for the functionality it provides developers. As a framework for distributed software projects, it itself is not a centralized process, which serves to give MIRA a better case for use in activities that involve multiple robots operating together, due to a lack of a central communications process.

As it is a C++ framework, and even includes the addition of new language structures and features to C++ without the need of a metacompiler or additional pre-processor, the time frame for completing the EZ-RASSOR project would be greatly hindered by the learning curve that is involved with committing to develop under the MIRA framework. However, it is understood that what should be chosen to be used for the EZ-RASSOR project as a middleware package should be chosen based on how it will provide ease in meeting the requirements, and while MIRA was a strong candidate to be chosen to serve as middleware to run on the EZ-RASSOR, ultimately the lack of employment of MIRA in the industry lead to our decision against it. While being cross-platform, it still has only found use in a small handful of projects in its history, telling us that it may not be fully matured, both in terms of functionality and community strength, which are two major factors that the timely and complete fulfillment of ROS requirements are based on.

Why Not an RT-Middleware Implementation?

OpenRTM consists of two separate middlewares: OpenRTM-aist and OpenRTM.NET. Both of these packages are implementations of the RT-Middleware standard, a set of specifications devised by the Object Management Group. Similar to ROS, the RT-Middleware architecture is designed in a way that involves associating hardware components with a concept called RTC, or RT-Components, which is analogous to the Node concept found in the ROS specification.

While OpenRTM-aist and OpenRTM.NET do not have tightly-linked simulation environments akin to the relationship between ROS and Gazebo, there have been efforts to create interface layers between OpenRTM-aist and Gazebo in order to facilitate simulation development and testing. Despite this, the relationship between the implementations and simulation suites is not seen as matured, and compared to what the ROS environment has to offer, lead us to decide against moving forward with an RT-Middleware implementation.

Simulation Software

As we considered various simulation software, we also wanted to ensure that we had a fully open source solution if possible without sacrificing capability. While there are not many software solutions that would fit this category, the ones listed below could have all been used for this project.

Why Gazebo?

Gazebo is a robotics simulator designed in C++ that is fully integrated with ROS. It is an open source project that has been in development since 2003 and

provides us with a playground to test and modify various aspects of our robot without risk of damage to the robot or any need for actual physical hardware pertaining to the robot. We will be able to implement all necessary robotic actions within this environment. We will use Gazebo to refine both user-controlled operations as well as autonomous and swarm AI operations. With the use of Gazebo, we will extensively test our ROS architecture and determine the best way to pass data between nodes. We will determine how well various autonomous functions perform and have a better idea of which practices are more viable before testing on the physical robot. The area where Gazebo will prove most useful is with the implementation for swarm robotics. We will be able to fine-tune the different ways a group of EZ-RASSORs can and will behave together without any financial risk. Although the transition from simulation to the real world is never perfect, we will gain much more ground testing in this environment and working out the kinks of the transition when the time comes. For the first semester of Senior Design, we intend to finish a software version of EZ-RASSOR that runs inside of Gazebo, both autonomously and user controlled.

Gazebo uses a URDF file type to create ROS compatible robot models. URDF stands for “Unified Robot Description Format.” The file is written in XML and revolves around creating different components of the robots, defining the size of a given component, giving labels to the components, and how all the components link to each other. It is important to note that XML is not a programming language, but a markup language. No functionality of the robot can be defined inside the XML. All XML can do is tell Gazebo how the the different individual parts of the robot look, how big the individual parts are, and how all the parts piece together to create the whole robot. The individual parts range from the chassis, wheels, and even the various sensors that reside on the modeled robot.

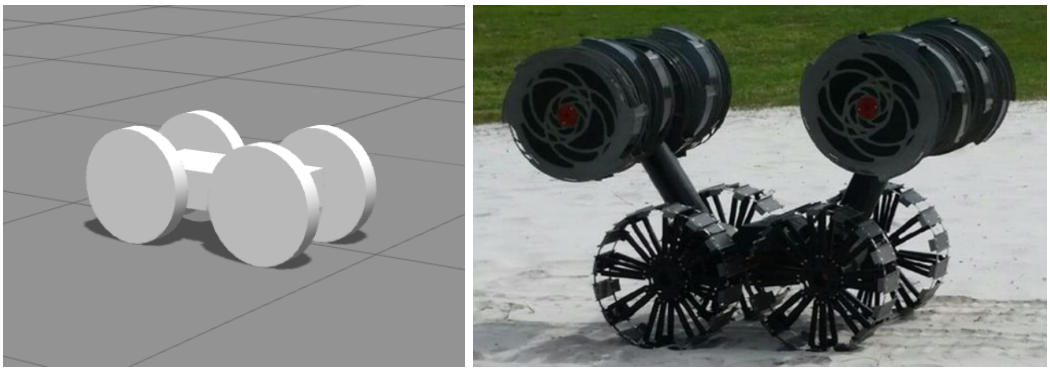


Figure 40 | Side by side comparison of a prototype model and the RASSOR [20]

Gazebo also provides us the ability create various environments to test the EZ-RASSOR in. This will allow us to insure that the robot works properly on flat land and hilly terrains. To create a custom environment, we must make “height maps” for Gazebo to interpret. A height map is a black and white image where black represents lower altitude and white represents higher altitude [22].

Part of the resources of the Gazebo community is the ability to share models and simulation backdrops for use in other projects. As such, we intend on shaping the background for our simulation with models found in the community. Since designs in Gazebo can be made with intense granularity, like defining how an object reacts to sunlight, we can spend more time on the functionality of the Rover by utilizing community resources.

In designing the EZ-RASSOR rover for the simulator, we will also be using open source models as a foundation. This will be helpful to get functional tires, as well as proper weights. However, we will still be designing the rest of the rover to match our hardware as well as to accommodate the specific requirements for this project such as a camera that we will use to monitor the environment.

Why Not V-REP?

V-REP is perhaps the most popular robotics simulation software available. It was developed and is maintained by a company in Switzerland known as Coppelia Robotics. Strengths of the software include a strong set of physics engines, a large library of default robots, and is very well documented due to the company behind the product. It is also widely regarded in the robotics community to be a complete simulation environment, providing the user with a multitude of resources that ship with the software to help empower developers.

However, the program itself is not open source. The software can very easily be used by educational institutions under the GNU GPL, but requires a commercial license otherwise. We believed this to be a main drawback based on the spirit of this project. Since our EZ-RASSOR will be open source from the ground up, we also want to use a simulation software that has a strong community of open source contributors.

Why Not ARGoS?

ARGoS is a simulator that is completely open source with the code still being actively maintained on GitHub. The simulator is available for MacOS and Linux and is designed for large-scale robotics simulations. By leveraging the power of C++, users can extend their simulations beyond what is available through the UI. Researching what others have been able to achieve in ARGoS was very promising as well; there are many projects which utilize the software for what it is known for by making massive scenes and massive sets of robots that traverse the area.

Although an important use of ARGoS is in the case of massive robotics, as could end up being the case for our EZ-RASSOR project, we ultimately decided against it for its lack of support. ARGoS was created by Carlo Pinciroli alone and although there have been others since inception who have contributed to the project, that number only amounts to 11 other people. Given the time constraints on this project, we needed an open source simulator that was well-documented and well-established. Additionally, the UI for the program was fairly simple in nature when compared to other simulators. We would end up with a dependence on writing C++ code specific to the simulator to accomplish basic goals that other simulators have achieved through a WYSIWYG (what you see is what you get) editor.

Machine Learning Software

One goal of the EZ-RASSOR project is to create a tool that can replicate the behaviors of the RASSOR robot developed by NASA. Because of the nature of the environment, the system will interact with and the requirements that it be capable of functioning fully without human input, there is a need for some type of artificial intelligence and machine learning. The software needs to be capable of implementing neural networks and reinforcement learning. It should also be efficient, easy to train and flexible.

Why Tensorflow?

Tensorflow is an open source software package, developed by Google, that functions as one of the most widely used machine learning and artificial intelligence toolkits available. Tensorflow contains a powerful computational core that is based on the concept of tensors. is an open source machine learning framework that provides implementations in several languages (C++, Python, Javascript) and is meant for multiple environments (AWS, Mobile, Desktop). The software provides a useful abstraction around some of the more complex implementation details of machine learning and provides higher level tools (Keras) to help rapidly prototype models and network architectures.

One of Tensorflow main strengths is the graph based approach to its computation. This structure allows for easy analysis of the algorithms the user has created and also provides easy portability to other platforms. Tensorflow also provides extremely well documented, reliable implementations of a wide variety of machine learning tools. Having a reliable system that can implement Convolutional Neural Networks, Deep Neural Networks, Reinforcement Learning and Natural languages processing all together is incredibly powerful and something that will benefit us a lot in terms of exploring our options for the autonomous control system.

Another great benefit of Tensorflow is it's support for tensorboard. Tensorboard is a browser based visual analytics tool built into tensorflow that allows for visualizations of the graph and its execution. Tensorboard also allows for measurement and metric visualization of the training process and the final results. This will be particularly useful for our project because we can produce visualizations for our final presentation. The graph execution is also useful for debugging and fine tuning the system, something our project will likely use quite extensively.

Why Not Caffe?

Caffe

Caffe is a machine learning framework that was developed at the University of California Berkeley at the Berkeley Artificial Intelligence Research (BAIR) Lab. It is an open source solution that provides an easy-to-use design that allows developers with less experience to get something up and running fast. While the platform is rapidly increasing in popularity, it is not widely considered useful for production code.

Why Not PyTorch?

PyTorch

PyTorch is a newer machine learning framework started by Facebook's AI research group that debuted this year (2018). While documentation is very accessible and the platform in itself is stable, it does not have the long term stability that TensorFlow does. Moreover, it is more difficult to deploy than TensorFlow. While PyTorch requires the use of a REST API that must be placed a layer above the model for access, TensorFlow provides its own framework for deploying models and accessing them without special configurations.

Computer Vision Software

Why OpenCV?

When developing an intelligent system, interacting with a real environment as opposed to a simulation can lead to many potential issues. For our project, a large portion of the input data will be provided by cameras. Our autonomous control system will need to be capable of developing an understanding of its environment from largely 2d image data. OpenCV is an open source computer

vision library that works well in collaboration with tensorflow. OpenCV will provide state of the art vision algorithms that will provide object detection/classification functionality, as well as, object tracking and distance measures to assist in building a model of the environment.

Mesh Network Connectivity

When Implementing a mesh network there are numerous things one must consider before implementation. We must think about how many devices we would like connected, what is the maximum distance we plan to allow between the robots. Do we want many robots to talk to a single host, or many robots talking to many other robots. Different technologies offer unique strengths and weaknesses, but none would make the perfect candidate. This section will describe the pros and cons of those technologies in detail and determine which one will suit us best for this application.

Why Wi-Fi?

Widespread-available Wi-Fi has been around since the turn of the century and has proven time and time again to be a reliable, secure, and high-speed method to communicate amongst paired devices. Wi-Fi is high in consideration for this project due to the fact that ROS natively supports Wi-Fi connectivity. This makes implementation of our mesh network much more streamlined and simple.

Wi-Fi provides service in private homes, businesses, as well as public spaces with Wi-Fi hotspots that are free-of-charge or commercial.

> The 802.11 standard provides several distinct radio frequencies to use in Wi-Fi communications. Each range is divided into multiple channels and regions provide rules and regulations on channels which set maximum power levels within these frequency ranges. We are lucky enough in Orlando to have premium frequency ranges which would offer us higher-than-average Wifi speeds. [23]

Wi-Fi provides us with a stable connection among the entire swarm and allows interconnectivity between the robots. Sadly, there are still some drawbacks to using Wi-Fi that could present problems in the future. If used as transceivers onboard the EZ-RASSOR, it could consume an excessive amount of battery. Also, the microcontrollers that represent the brain of the robot might not be able to process the high data rate and cause unexpected or unforeseeable errors. The team currently has some plans in place that could possibly avoid the on-board processing issue, but we are still unsure of the battery capacity of the hardware provided.

Why Not Bluetooth?

Bluetooth has been around for quite some time now and has proven to be a reliable method to easily connect devices such as speakers or headsets. When taking Bluetooth into consideration with regard to our mesh network, several issues arise.

The hardware incorporated into Bluetooth consists of two components. First of which is a radio device. The radio device is responsible for transmitting and modulating the Bluetooth signal. The other component is a digital controller. This is a CPU whose primary responsibility is to run a “Link Controller”. The Link Controller processes the baseband and manages FEC protocols. It also takes care of asynchronous and synchronous transfer functions. The CPU also attends to all instructions related to the host device.

The first issue that arises with the use of bluetooth is its lack of security.

> Bluetooth 2.1 and earlier did not require any form of data encryption. Additionally, if encryption was implemented, the encryption key is valid for only about twenty-four hours. This is because if more time is given, an attacker could apply an XOR attack to retrieve the encryption key. Later versions of Bluetooth address this security flaw, but even in recent years, Bluetooth has been criticized for severe security flaws. In April 2017 a set of Bluetooth vulnerabilities called “BlueBorne” have arose. These

vulnerabilities would allow attackers to connect to devices without authentication and have full control over the device. [24]

Another major issue with Bluetooth is its support of mesh networks. For nearly the entire time Bluetooth has been in service, it has solely supported one-to-one device communications. This means they there can only be a single host device and a single child device. This clearly set bluetooth out of consideration for this project because we need multiple devices to communicate with each other as well as communicate with a single predetermined host device. Very recently Bluetooth technology now supports mesh networking. Although, this technology is in very early stages and could potentially be a liability when further in development. We have chosen to pursue time-tested technology that we know will securely execute the job we need done.

Programming Languages for On-Board Processes

Being an embedded system, a large factor in the efficiency and practicality of the running software is the means by which it is implemented. Considering this, it is easy to quickly find oneself desiring to stick to bare-bones languages and toolings for all assets that are held and ran locally on the EZ-RASSOR. When determining the appropriate tools to employ on on systems with dedicated, domain-specific usages, it is often wise to keep the usage of heavy libraries, runtimes, and computationally expensive processes to a minimum.

The ROS architecture supports hardware programming in both the C++ and Python programming languages, so due to this mindset of focusing on low-level efficiency, we initially leaned toward keeping on-board EZ-RASSOR software written in relatively lower level and compiled languages, particularly C++ for the ROS module. However, after further research into what may bottleneck a system of already high complexity that must employ several toolings, and both the objective of the project and the nature of its development life-cycle, the decision to employ Python as our primary language to program the EZ-RASSOR was an easy one to make.

Why Python?

Much of the software will be written in Python, and this decision has less to do with the mechanics of the language, such as its grammar and features, and more to do with the ease with which one can develop a full system from many complex modules, which is provided by the vast number of easily-usable tools and components made available by developer communities from all fields. In other words, if in order to meet a specific requirement an additional complex functionality must be used that may constitute an entire project on its own, the probability of finding a neatly packaged, easily usable, open source Python library is high. This is likely due to the nature of Python both as a language and its history; its ease of use and generous learning curve have facilitated rapid development of a plethora of usable plug-and-play components by both professionals and hobbyists as the language soared in popularity over the years.

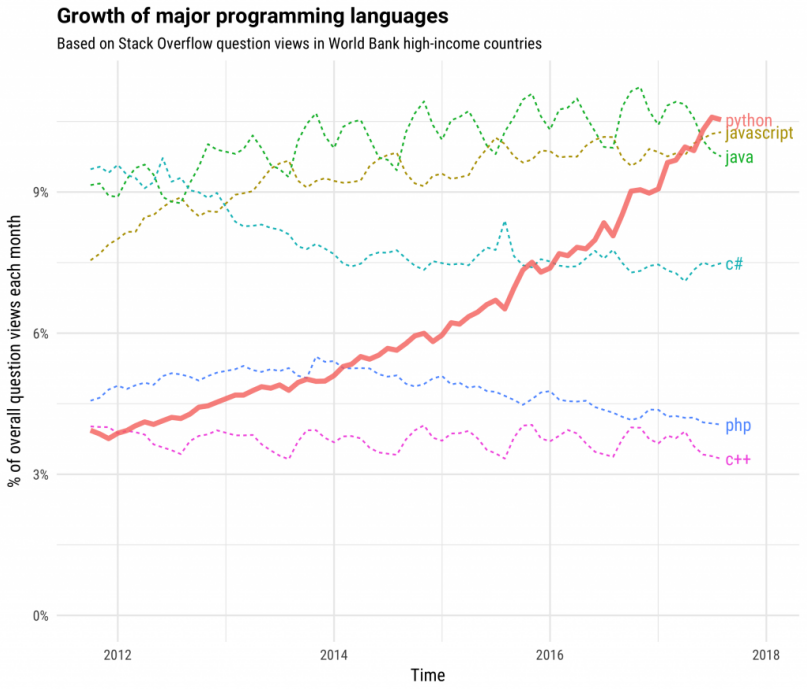


Figure 41 | Python's growth as shown via views on Stack Overflow questions, compared to other major languages. (Stack Overflow)

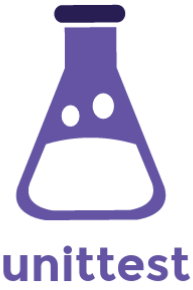
As Python is a general-purpose language, it lends itself to all fields, thus providing the force that creates this growth. As the EZ-RASSOR itself is largely a multi-faceted project, with components pulling from multiple diverse fields in computer science, employing a general purpose programming language with extensive support for dedicated uses facilitates group unity and minimizes middlemen needed for communicating between components.

A strong developer community surrounding a programming language also expedites the development of this project, as an incredible breadth of resources made freely available by a growing, passionate community greatly reduces time spent resolving development issues, developing custom modules from scratch, and determining the best courses of action to take when deciding on solutions to specific problems. The Python community is one of the fastest-growing communities for a programming language in history as well, which may also be attributed to its ease of use and accessibility. According to the TIOBE Index, which analyzes programming language usage and popularity, Python surpassed C++ and ascended to the top three languages in popularity in their report from September of 2018. Stack Overflow also supported the notion that Python may be considered the fastest growing major programming language.

While many new technologies are being rapidly introduced to the development world, Python has shown more than promising strength in its capabilities by setting a new precedent for what it means for a programming language to grow. Compared to other technologies that have been introduced relatively recently, Python's growth far surpasses those.

Additionally, the development life-cycle of our software is one that will involve rapid prototyping, and our team's close familiarity with Python and development methods that go hand-in-hand with the language enforce the idea that a more sustainable and organized development process will result from choosing this language. Less time will be spent double-checking language documentation, questioning ideas and practices in development methodology, and refactoring code, due to our experience with Python outweighing our experience with C++. Furthermore, a consensus on a consistent, team-wide programming methodology is more easily attainable when committing to a toolset the entire team knows well.

Why Unittest?



Unit testing for all Python ROS nodes will be done using the Python unittest framework. This framework is the main framework for unit testing in Python. Why are unit tests necessary? Does unittest do everything the team needs for testing?

Unit testing provides an excellent way for developers to automatically check their code for errors, edge cases, and unknowns (edge testing). Unit testing also provides a way to create acceptance tests that confirm that core functionality is operational. After every new codebase release, unit tests should be run to confirm that all core functions still act the way they are expected to act.

For an example of edge case testing, it is useful to test each function by passing various parameters into the function and monitoring the function's behavior. It is important to pass edge case parameters like NULL, 0, negative numbers, and max/min numbers into functions to ensure they work for an appropriate range. In Python, types are inferred and are less critical than in other languages. This opens up the opportunity for unit tests that could pass weird/foreign/unknown types into functions to check the functions' reactions.

Unit tests should also be used to write acceptance tests. Acceptance tests are procedures that confirm that a piece of the software is working correctly in normal use (as opposed to edge use). For example, acceptance tests will be necessary to confirm that the ROS nodes controlling the wheels and arms on the RC car respond correctly when told to move in some certain direction.

It is still up for debate whether edge testing or acceptance testing is more valuable. Many argue that excessive edge testing wastes huge amounts of time and is still incapable of detecting all edge cases. Others argue that acceptance tests are too obvious and are a waste of time to write. Both test types do provide some value and will be used, however measures will be taken to prevent the team from spending too much time writing tests and not enough time writing code.

Unittest is a capable framework that provides all the tools necessary to create proper edge tests and acceptance tests. It allows multiple tests in one file, automatic execution, mocking, and setup/teardown functionality. These are powerful abilities that make it easy to create scenarios and situations that adequately test this software's functions.

Why Bash Shell?

The Bourne Again Shell (Bash) will be used as the primary shell for this project. It will be employed in the Linux environment on the Raspberry Pi for the RC car,

and also will be present in the Linux environment on the main EZ-RASSOR. Bash has a lot of features that make it necessary for this project. But first, what does a shell do?

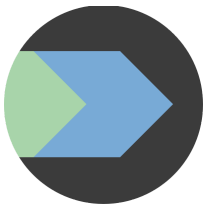
An operating system's shell interfaces directly with the kernel of the operating system (the code that runs all the hardware on the computer). The shell acts like an API to the kernel and provides many system calls that are useful to programmers (like printing to stdout and writing to files). The shell also provides a huge array of tools to write scripts that can "glue" programs together. Examples of these tools include pipes, loops, environment variables, and more.

This project needs a shell that can wrap together ROS and any other external Python or C++ programs that may be necessary. More importantly, this project must be easily installable and configurable by end-users. This is especially true for the open-source code used on the RC car, and the team wants this code to be deployable with a minimal number of instructions. Does Bash provide these functions?

Absolutely! Bash is able to glue programs together easily using Bash scripts. These scripts exist in many, many projects and are very familiar to Linux developers. Bash scripts are capable of doing anything that a programmer might want to do, from downloading an entire project, to creating a file system, to spinning up an entire ROS graph.

Bash scripts will be used to provide an easy way to install and configure this project, especially the RC car portion. These scripts will handle dependencies, files, and everything else that often makes installing new software a huge pain.

Why Not Fish Shell or ZSH?



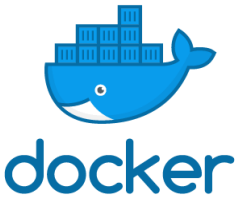
Other available shells on Linux include ZSH and Fish Shell. Both are well supported shells with large communities, however both are inappropriate for this

project. Bash is better than both of these shells for this specific application (not necessarily in general).

ZSH is extremely similar to Bash shell, but with a few added features. ZSH is honestly a better version of Bash for general use, and frequently Linux developers will switch from the default Bash to the improved ZSH as they become better at working in the Linux environment. Why use Bash then? Bash is the default shell on most Linux distributions, and for this reason it is more attractive than ZSH. The team wants this software to be friendly and easy to install. We can reasonably assume that anyone installing this software will have Bash available, because Bash is the default shell for Ubuntu, Linux Mint, and Debian, to name a few. If we required users to install ZSH, that would make the installation process for this program more complicated. Also, requiring ZSH as a dependency would only improve the installation experience, which is a rather small portion of what this software is going to do.

Fish suffers from a similar problem, and has its own major issue as well. Fish is, of course, not default on any Linux distribution and is rather different than Bash and ZSH. In fact, Fish is extremely different and Fish syntax is completely incompatible with Bash and ZSH syntax. This means that any scripts written for installation and configuration would have to be written in Fish script, which includes a significant learning curve for people only familiar with Bash. Familiarity is the key and, despite Fish's attempts at being more usable and friendly than Bash or ZSH, its wildly different syntax makes it undesirable for this project.

Why Docker?



Docker is an open-source project that automates the deployment of software applications inside containers by providing an additional layer of abstraction and automation of OS-level virtualization. Docker does not emulate any hardware. It also doesn't require preliminary allocation of memory or disc space either, so it is very quick to get running. A container like Docker utilizes the computer's actual hardware and isolates the embedded application, so we can work with software

and libraries that are specific to the EZ-RASSOR but not to the host operating system. The key benefit of Docker is that it allows us to package the application with all of its dependencies into a standardized unit for software development. This container can then be shared amongst the team for a streamlined development experience.

> Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop. This gives developers the ability to create predictable environments that are isolated from rest of the applications and can be run anywhere. [25]

From an operations standpoint, apart from portability, containers also give more granular control over resources giving your infrastructure improved efficiency which can result in better utilization of your compute resources.

Due to these benefits, containers have seen widespread adoption. Companies like Google, Facebook, Netflix and Salesforce leverage containers to make large engineering teams more productive and to improve utilization of compute resources. In fact, Google credited containers for eliminating the need for an entire data center.

Setting Up Docker

After installation we must check `docker --version` to ensure that we have a supported version of Docker.

```
docker --version

Docker version 17.12.0-ce, build c97c6d6
```

Figure 42

Next we can test our installation to ensure It is working properly. We can test this using a simple hello world Docker image.

```
docker run hello-world
```

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest:
sha256:ca0eeb6fb05351dfc8759c20733c91def84cb8007aa89a5bf606bc8b315b9fc7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

Figure 43

Here we have a list of commands that will be helpful:

```
## List Docker CLI commands
docker
docker container --help

## Display Docker version and info
docker --version
docker version
docker info

## Execute Docker image
docker run hello-world

## List Docker images
docker image ls

## List Docker containers (running, all, all in quiet mode)
docker container ls
docker container ls --all
docker container ls -aq
```

Figure 44

Mobile Application Frameworks

When determining which mobile framework to use there is a wide array of options to choose from. The main three I have chosen to look further into is NativeScript, React Native and Unity. I have chosen these three frameworks due to their modularibility and simplicity of porting between iOS and Android operating systems. Each one of these frameworks offer WIFI features which helps us implement our mesh framework.

Why React Native?

React Native is a mobile application framework that is very similar to NativeScript, due to it's JavaScript based framework for writing natively rendering apps and components for iOS and Android. Instead of using Angular like NativeScript, React Native is based on React, Facebook's Javascript library for developing web-applications and user interfaces. Although, this application targets mobile devices rather than web browsers.

The question may arise, "What is the reason for this focus on *native* applications?". The simplest answer is so the application stays acceptionaly responsive and prevents the app from dating as quickly. In 2012 Mark Zuckerberg commented,

"The biggest mistake we made as a company was betting too much on HTML5 as opposed to native". [26]

Zuckerberg had this regret because his companies apps were written using existing methods which would use a combination of JavaScript, HTML, and CSS to render a webview. This is usually suitable, and is the norm for websites nowadays, although when implementing in a mobile environment, this approach suffers from performance issues.

React Native solves performance issues by incorporating React, which works separately from the main user interface thread, so the application is able to maintain high performance without sacrificing capability. This is done by leveraging the phone's user interface libraries rather than using HTML and CSS as an intermediary.

React Native has a lot of things going for it, although nothing is perfect. Like NativeScript, React Native is relatively young being released in 2015. Although, It has a significantly larger community due so several mainstream applications such as Facebook, Instagram, Skype, Pinterest and Uber all adopting its use. This means React Native has many active developers that are always looking to

better the technology and assist others in achieving its full potential. The other pitfall of React Native is that because it adds another layer to the project, it can make debugging a bit more difficult. This is due to the intersection of React with the host platform.

React Native is a very promising solution to our problem, and with its cross-platform capabilities, it achieves the degree of accessibility we are looking for. React Native is still in its early stages, and the typical pitfalls of new technologies are still present, but we believe the pros will outweigh the cons in this situation.

Why Not NativeScript?

Nativescript is an open source framework to develop apps on iOS as well as Android. This framework was initially designed by the company Progress. Nativescript builds apps solely through the use of Angular and Javascript (or any language that compiles to Javascript, such as Typescript). Any app developed in Nativescript is completely native, meaning it has access to the same APIs as if the app was developed in Android Studio or Xcode. Also, developers can incorporate third-party libraries like npm.js, CocoaPods and many more without the use of wrappers.

Nativescript is available on all major platforms (Windows, Linux, and MacOS) which would suit well working with ROS and Gazebo. It offers a “Hot Reload” feature for instantaneous visual changes when code is updated, making development very quick. Nativescript seems like a great option although it does has its shortcomings. It’s greatest weakness is its age. Nativescript was initially released late 2014 and is still building traction. This means more time is needed to grow to possess a decent 3rd party community and the current number of available components is limited. Another side effect of NativeScript’s lackluster community is its *lack* of a community! There are a number if open issues on NativeScripts GitHub and until this technology gains more traction, It would be quite risty to determine if this approach is going to work.

Why Not Unity?

Unity is a cross-platform game engine created by a company called Unity Technologies. Unity's first debut was in 2005 as a MacOS exclusive game engine. Fast forward to today and Unity now supports twenty-seven different platforms! This engine can be used to create 2D and 3D games as well as simulators.

After reading Unity's introduction, you may be thinking "Why is a game engine in consideration for a mobile app?". Well, one must first consider what *is* a mobile app? In our case (referring to mobile app controller 1) the mobile app is a single screen with several buttons that achieve certain functionality. One could also see this as a main-menu within a 2D game. Unfortunately we would would create the most boring game in existence because the "main menu" is all we would need in our application.

To implement the mobile app controller using Unity would be fairly straightforward. Unity also as a large community so we would expect to have very few unresolved questions. As stated previously, Unity supports an abundance of platforms which would make the application even more accessible due to the fact that it could run on phone as well as game systems and mainstream operating systems. This makes it incredibly easy to test the basic functionality of our app without the need of having it loaded on our intended device.

Unity seemed like the optimal choice that hit all the marks we needed for our application. As we traversed deeper into our research we found some major performance pitfalls that could cause unnecessary headaches later in development. The first major flaw is the entire concept of using a game engine for a standard graphical user interface application. Unity's core game engine runs in the background at all times rather than being event driven. This means even a simple "main menu" app would be considerably CPU and GPU intensive even if nothing is actually happening. This would entail a significant hit to battery performance. In contrast, React Native and NativeScript use the host's native UI

components and launcher, so if no events are detected, then no resources are used. The other major issue with unity is the necessity of a license. Unity does have a personal version which is free but the point of EZ-RASSOR is to make a widely available application. Deploying this app would invoke monthly licencing fees which could be covered initially by the project budget, but would soon require more outside funding which we do not have available to us.

Backend

There are many different web servers that you can install on the Raspberry Pi. Traditional web servers, like Apache or lighttpd, serve the files from your board to clients. Most of the time, servers like these are sending HTML files and images to make web pages, but they can also serve sound, video, executable programs, and much more.

However, there's a new breed of tools that extend programming languages like Python, Ruby, and JavaScript to create web servers that dynamically generate the HTML when they receive HTTP requests from a web browser. This is a great way to trigger physical events, store data, or check the value of a sensor remotely via a web browser.

Why FLASK?

Flask is a Python framework that allows developers to easily create a database without having to worry about the intricacies of database schema. Instead users can create classes in Python and the Flask framework will create a database based on that information, even with foreign key constraints. For the most part the core of the software being written for the logic of the EZ-RASSOR software is already going to be written in Python. This an advantage to use because many of the functions that are going to be written, such as moving the arms or moving the EZ-RASSOR is already going to be written in Python. As you can easily import other Python files as module into others. This would save us in the implementation writing the API, instead of having to rewrite some of the core

functions, or just simply call the other program from a terminal command, this allows for the importing of other already written functions in Python. There is also much support available online for this framework which highlighted it as being easier to use than our other considerations.

Why Not Express?

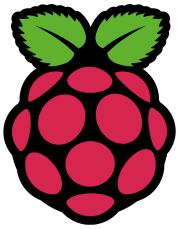
Express is a JavaScript library that allows for easy communication with a defined backed. Unfortunately the main reason that we did not pursue this avenue is due to the lack of cohesiveness with the rest of the project. Since our core development will be written in Python, we wanted to choose a development platform that would consist of the same language.

Why Not Django?

Django is another Python framework that is powerful enough to host an entire web application backend. However, when we looked at our hardware limitations, it became clear why Django would not be the best solution. The core of the EZ-RASSOR will already be overloaded with the other processes running such as the computer vision which means that resources will already be scarce. Django is a much larger framework that comes with a lot more features packed in. In fact one of the initial considerations was instead of making a frontend GUI application, it was to run a web server that can directly control the EZ-RASSOR without any installation. Django in that use case would have been perfect as it includes a solid backend for front end website development. The problem was hardware, doing with such an implementation would cause a significant lag to the CPU, which as it is could be a little underpowered.

Hardware

Why Raspberry Pi?



The Raspberry Pi is a credit-card-sized computer capable of running full desktop operating systems. It is often used in conjunction with Raspbian, a spin off of Debian, which itself is a Linux distribution. The Raspberry Pi supports USB connections, WiFi, HDMI, and connections to microcontrollers and other pin-based interfaces, making it the perfect brain for this project's demonstration, the RC car.

The Raspberry Pi will be used for the demonstration RC car because it interfaces well with all external hardware required for this project. Because it supports a fully fledged Linux operating system, it can run ROS and handle the ROS graph with ease. It is also rather cost-effective and its on-board WiFi means that this project can employ a mesh network and other swarm technologies without the addition of more hardware. It also does not consume a huge amount of power and requires only an SD card for storage, further simplifying the hardware requirements of the demonstration RC car.

Perhaps the Raspberry Pi's greatest strength is its familiarity. Students who have dabbled in hardware projects or robotics likely have seen or worked with Raspberry Pis in the past. We want students to be able to focus on the code and not worry about learning how to operate in a foreign environment or use archaic/obtuse hardware. The comfort that the Raspberry Pi provides is invaluable in this respect.

Why a Demonstration RC Car?

We plan to build and operate our ROS graph on a demonstration car for this project before porting the graph to the main EZ-RASSOR. The design of our car will be simple: we will take an RC car kit, build it, and add some moving pieces to represent the EZ-RASSOR better. Microcontrollers will interface with a Raspberry Pi that has been mounted to the top of the car.

Why create a demonstration car in the first place?

Primarily, the car will give us an opportunity to demonstrate ROS code in the real world, outside of a simulation. This is useful if the EZ-RASSOR is not completed in time or cannot be present for project presentations in April. It is also possible that our access to the EZ-RASSOR will be restricted to certain times or days, but if we have a demonstration car we can work on it at will.

Building the car will also give the team early experience with deploying ROS in a Raspbian environment. Gaining this knowledge early will make the initial setup and configuration of the EZ-RASSOR easier, and will help us iron out bugs in our process before we get the real hardware in our hands. If something is going to go horribly wrong, it is better if it goes wrong on the demonstration car instead of on a very expensive piece of NASA hardware.

Finally, because ROS is portable, any and all of the logical ROS nodes created for the car will be integrated instantly into the EZ-RASSOR once it is ready. The nodes used for the hardware (wheels, camera, etc) will surely be different; however these nodes will be third-party nodes and will be available on the Internet, pre-made, for both devices. By deploying our logic nodes first on a demonstration car and then on the EZ-RASSOR we will prove that our nodes are well written and portable.

Why the Sunfounder Model Pi Car?

This project will be using the Sunfounder Model Pi Car as the primary demonstration car to show off our ROS graph and navigation app functionality. This kit is perfect to build an RC car that runs ROS nodes for several reasons.

First, this kit is built for the Raspberry Pi, and so all included microcontrollers and servos are guaranteed to work with the Raspberry Pi and with Raspbian. The Raspberry Pi is guaranteed to fit on the car base and will be provided adequate power from the batteries. This compatibility takes a huge design concern out of building the RC car and allows the team to focus more on writing code, instead of on picking hardware and figuring out wiring and electronics.

Next, this kit comes with many of the same tools that will be present on the EZ-RASSOR. It has 4 wheels and a camera, and it also has additional servos that can be configured into arms. These arms will be important later, and will be discussed in the execution plan. The camera likely will not be used, but it will be salvaged for additional wires and servos. The kit also has great wire management tools such as wire wraps and ribbons to make the finished product look professionally made. Ideally, the RC car will look very similar to the EZ-RASSOR, except on a much smaller scale.

Finally, this kits provides an excellent foundation to work from. All of the pieces of this kit are movable and customizable. The acrylic base can be replaced and will be remade to support certain planned modifications. This new base will be bigger, thicker, and will support custom made arms that will simulate the arms and drums of the EZ-RASSOR. All of this is possible because of the equipment and instructions provided in this kit. This kit takes much of the work out of actually building a programmable robot.

Why Not the Sunfounder Model Pi Car Open Source Code?

Sunfounder has written a large program to operate its Model Pi Car. This code is open source and is available for free on Github. Our project will not be using any of this available code for a few reasons.

The provided code is “open-source” but the repository does not actually include a license. This means that anyone who wishes to work with or build upon the code does not have a solid legal foundation to do so. Without a license, code is assumed to be non-extendable, and so it is legally safer to avoid this code.

Next, the provided code subscribes to a different architecture than the architecture that will be implemented for this project. The provided code is written using a client/server architecture. The server that Sunfounder implements runs on the Raspberry Pi on the robot and takes requests from a client (typically a desktop or laptop). The software does not use ROS, instead it interacts with the GPIO pins on the Raspberry Pi directly to control the car.

This is different from our planned implementation. By using a RESTful API we will still employ the client/server architecture for the RC car and EZ-RASSOR, however we must use ROS as middleware for portability and organization. The Sunfounder code would require extensive rewriting to convert it into something that works with ROS.

Finally, the provided code simply is not of high quality or sufficient clarity. It contains several confusing and convoluted algorithms, sparse and unclear comments, and a generally messy and hacky structure. It is not up to the team’s standards. It also contains a huge amount of code that is simply not needed for the project, like code that provides a web interface for the webcam in the kit that we will not be using. The code works, but it reads and feels like it was written

very quickly by a team that was not passionate about its quality or readability. It will be easier to rewrite necessary nodes from scratch.

Main Requirements Execution Plans

These execution plans reflect the same requirements as those listed under the “Phase 1 Requirements” section in our Statement of Work document. These plans were crafted after the development team completed their research on which technologies would be best to use for this project.

Main Block Diagram

The below diagram guides how the development team is approaching the implementation of the requirements. Before any of the execution plans were defined, this diagram was created to guide our further planning.

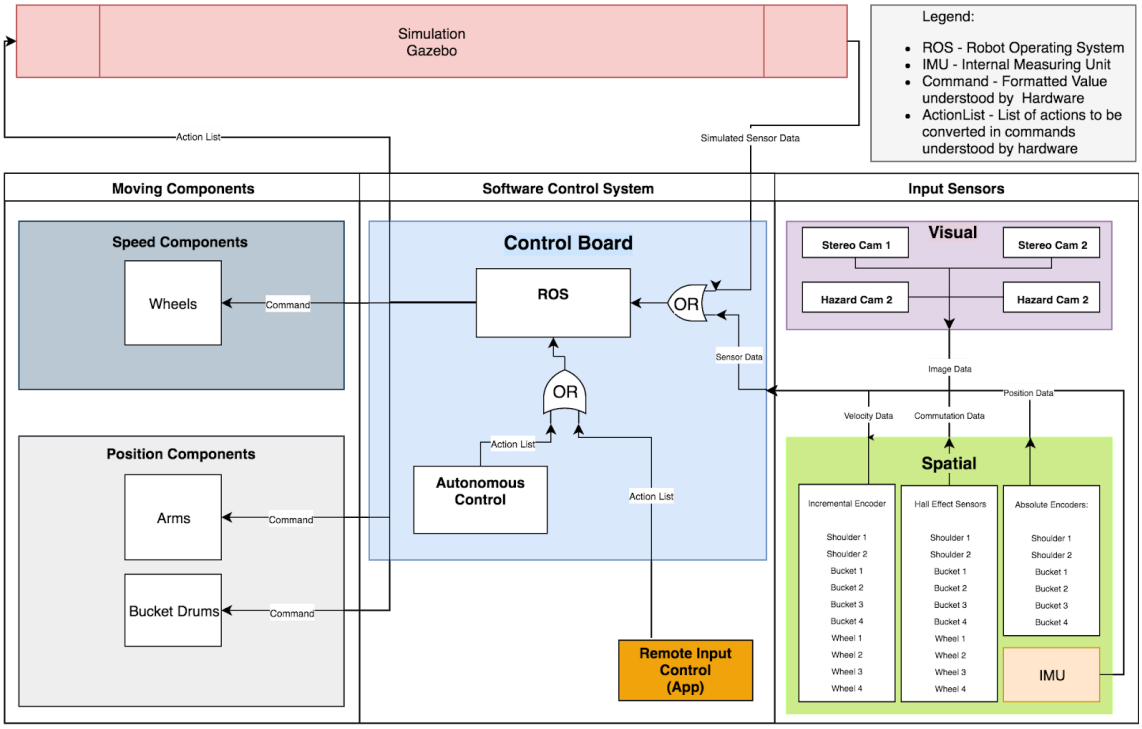


Figure 45

ROS Execution Plan

What follows is a sequence of ROS graphs that were created as we designed this project. The final graph is the current iteration of the ROS graph that we will use.

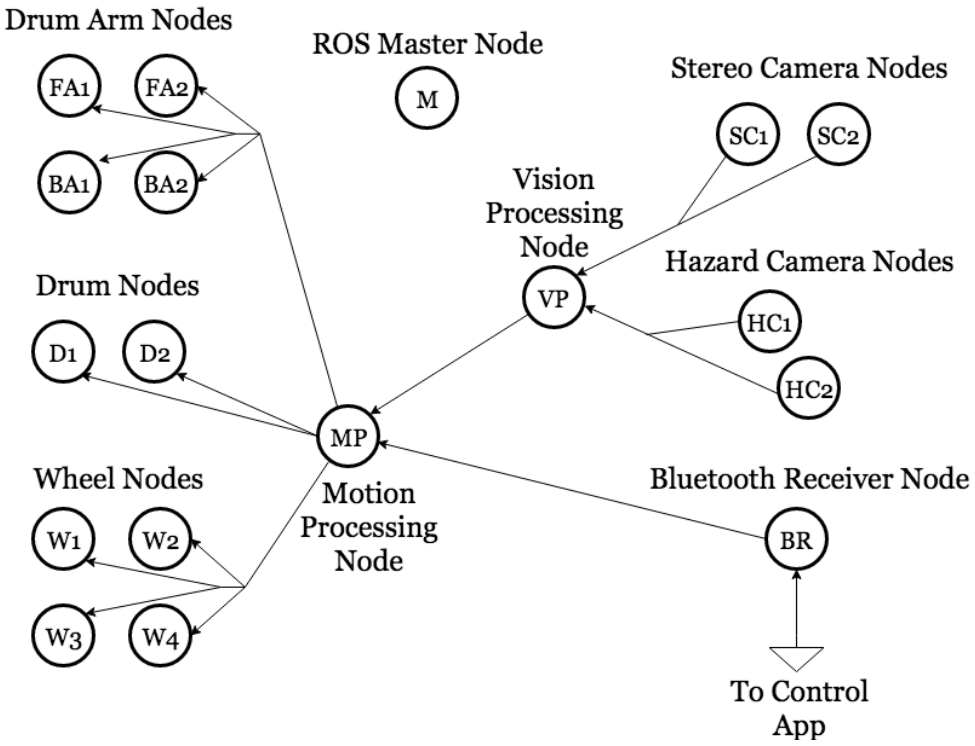


Figure 46 | This graphic shows the initial design of the main ROS graph for the robot. It is only a preliminary graph based on early brainstorming, and is not the best representation of the current ROS graph structure.

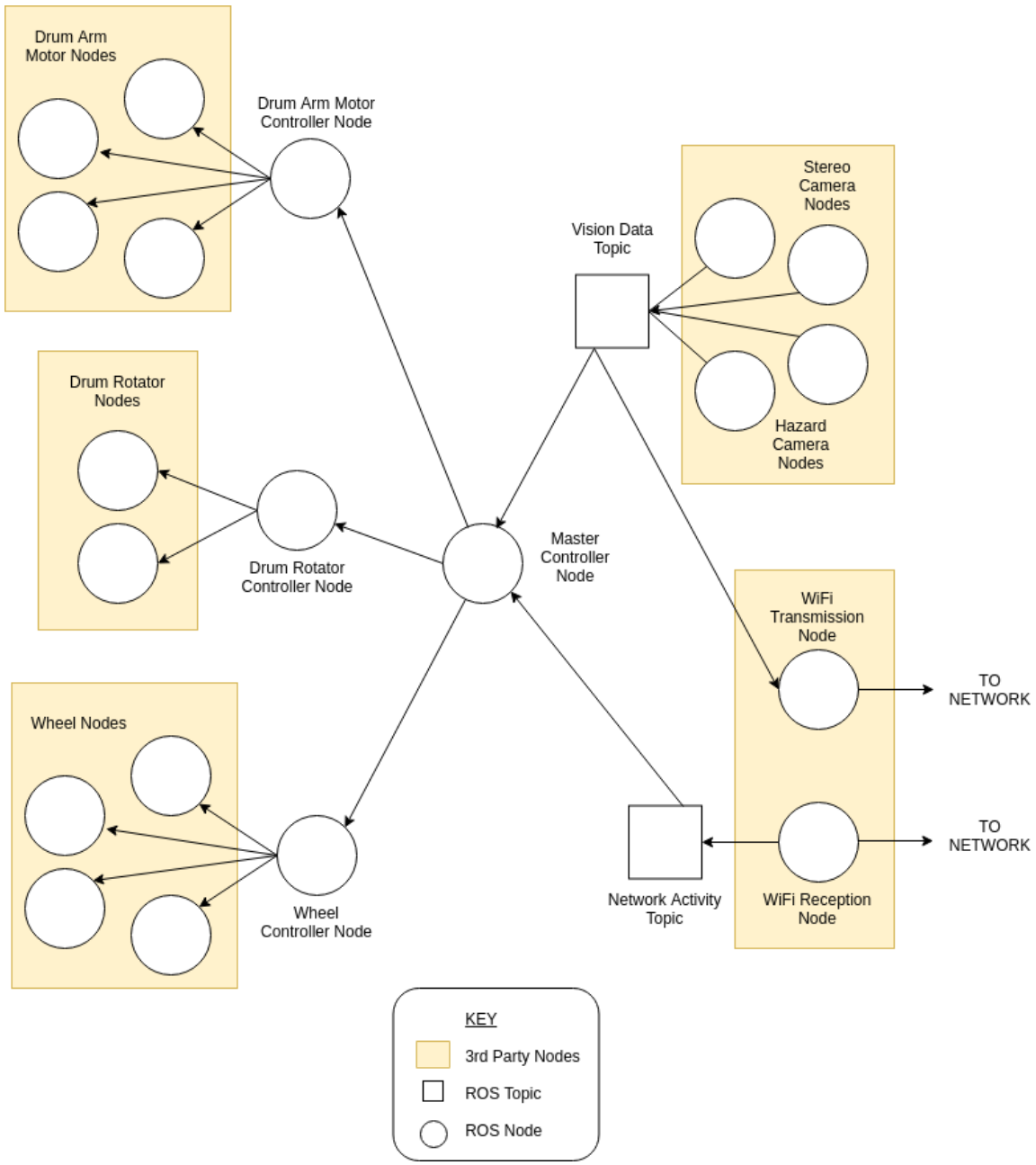


Figure 47 | This iteration shows a more refined graph that uses WiFi instead of Bluetooth for communication between the robot and control software. It assumed that 3rd party nodes would be found for many of the hardware devices present in this project.

Figure 48 | This ER diagram shows a preliminary overview of some of the node and class features in our ROS graph, however it is limited in scope and does not provide the finer details of the final graph.

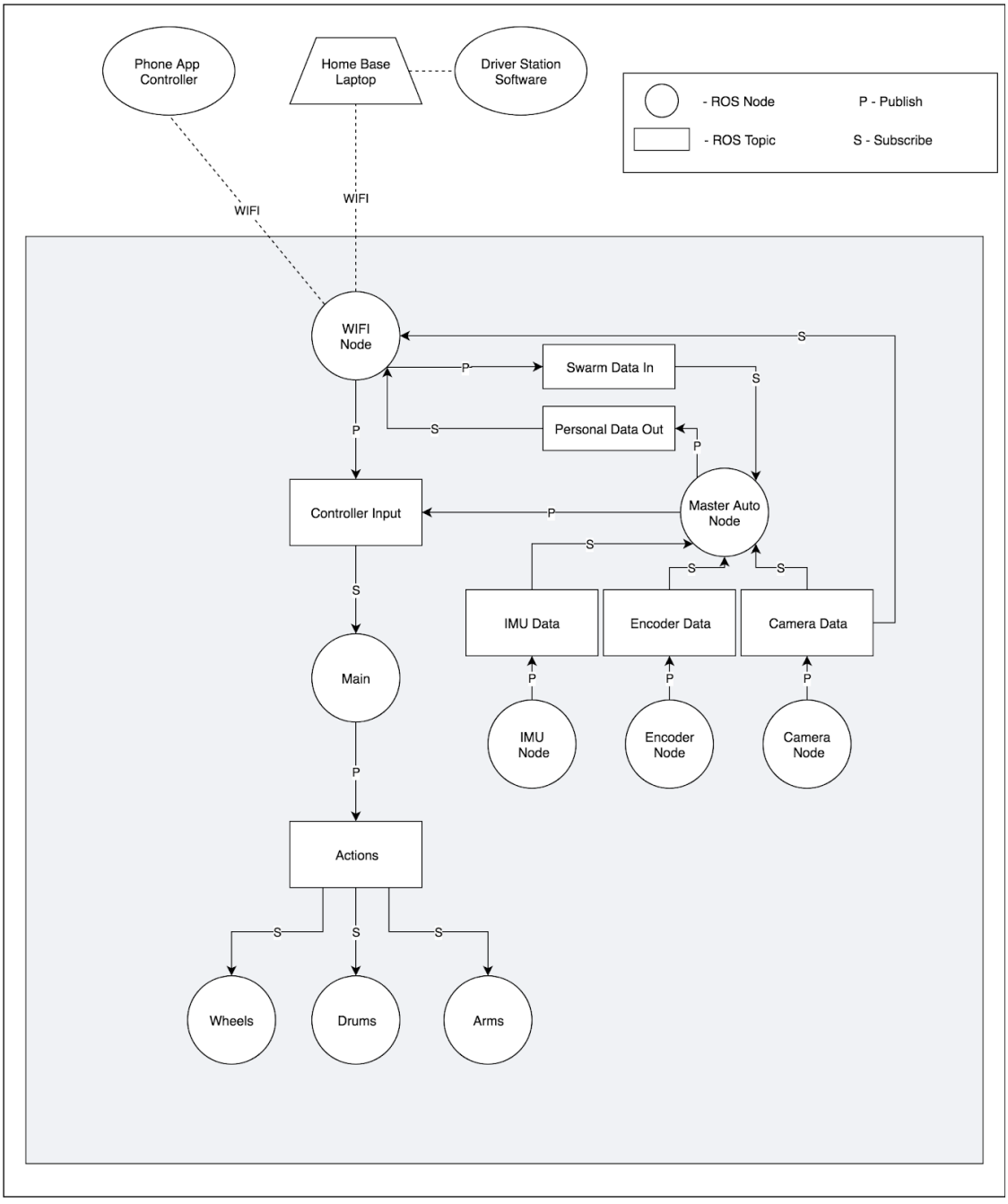


Figure 49 | The above image lays out the entirety of all the information and decision making that flows in and out of the EZ-RASSOR. Everything in the light gray pertains to the EZ-RASSOR's ROS framework, while everything outside the gray pertains to separate control systems. This is the current ROS graph that our team is working off of.

Outside of the ROS framework lies the home base, which will relay data to the EZ-RASSOR through the Wifi node. This data can either be controller data from

the driver interface, or swarm data that is calculated on the homebase itself. Also lying outside of the ROS framework is the mobile phone/tablet app. This application allows for direct control of the EZ-RASSOR with the use of an Android phone or tablet. The mobile application communicates with EZ-RASSOR's Wifi node in a similar manner as the homebase does.

The Wifi node is responsible for reading in external data, whether it be from the EZ-RASSOR's homebase, or the mobile application. The wifi node is responsible for publishing two types of data to two different ROS topics. Additionally, it is responsible for sending out camera feeds to the home base and phone app. The two types of data it publishes are controller inputs to the Controller Input topic, and data relating to swarm operations to the Swarm Data In topic.

The Encoder node represents various encoders/decoders for which we are currently unaware if they will be made available to us. Ideally, these encoders and decoders would consist of decoders to determine the current position of the EZ-RASSOR's arms at any moment, and encoders to tell whether or not the drums are currently full of mined material. If made available, we will use these sensors to publish their data to an Encoder topic to help the Master Auto Node with decision making.

The IMU node will be nothing more than an IMU sensor or a three-axis accelerometer. These two devices serve a very similar purpose so whichever one ends up being implemented is not a big game changer. The primary purpose of this node will be to get the EZ-RASSOR's current roll, pitch, and yaw data to detect if the robot has fallen onto its side or its back. Additionally, this data can be used to determine if the EZ-RASSOR is travelling up or down a hill and determine the steepness of said hill. The roll, pitch, and yaw data will be continuously published to the IMU topic. If an IMU is used, we will also be able to detect and publish the speed at which the EZ-RASSOR is moving and the cardinal direction it is facing.

The Camera Data node takes real time visual data and publishes it to the Camera Data topic. This node does not do any sort of calculations inside of it. It simply streams in a camera feed and publishes the stream. The Camera data topic is subscribed to by both the Master Auto node and the Wifi node. This topic is how the Wifi node is able to send live camera feeds back out to the homebase and phone app.

The Master Auto node is responsible for a plethora of data analyzation, including mapping, landmark identification, obstacle detection, decision making, and swarm calculations relating to its own role in the swarm. All of these responsibilities are implemented inside their own python scripts that the Master Auto node imports. This node is the entire AI back-end for the EZ-RASSOR. By not implementing each one of these tasks as a node, we increase efficiency throughout the system and avoid over complicating the ROS graph. The Master Auto node is subscribed to the Encoder topic, the IMU Data topic, the Camera Data topic, and the Swarm Data in topic. All the subscribed topics aid in various decision making abilities. Although this node is responsible for many tasks, its outputs are only published to two topics: the Swarm Data Out topic, and the Controller Input topic. The Master Auto node is constantly outputting swarm data to the Swarm Data Out topic for the Wifi node to subscribe to and relay the information back to the homebase in order to keep track of the given EZ-RASSOR's role in the swarm. On the local end, instead of creating an AI that directly controls the physical hardware of the EZ-RASSOR, we will implement an AI that acts and makes decision by relaying controller inputs to the Controller Input topic. In essence we are creating an AI user with a virtual controller pressing virtual buttons. (Note: Given the complexity and depth of these AI topics and responsibilities, we will keep the explanations simple for this section of the document and refer you to the AI section if you wish to read more in depth.)

When controller data comes in through the Wifi node or through the AI node, it is published to the ROS controller input topic as an integer. Each integer corresponds to a specific button on the controller or mobile app. The Main node is subscribed to the controller input topic and reads the controller input integer. After reading the integer, the Main node translates it into an array of integers that allows for each piece of hardware to know what action to take. The transfer and translation of information will be implemented in a way that allows for simultaneous button presses to occur and result in simultaneous actions. When the Wifi node and AI node have no output to send, they will continuously publish the integer 0 to the Controller Input topic. Integer 0 corresponds to the stop command. The Main node translate this command and sends the relevant Action array to halt all movement. The purpose of continuously sending the halt command while no buttons are being pressed is to prevent the EZ-RASSOR from indefinitely continuing its previously called action. Furthermore, this would require us to implement a button specifically for stopping movement.

Wheels	Arms	Drums
0	0	0

Figure 50

Example of the Action array for the command “Stop”

A full list of translations can be found in the section “Translations of Controller Inputs to Action Array.”

The Wheels node is responsible for controlling all actions pertaining to the wheels. This node is subscribed to the Action topic and pulls the Action array off of it. The Wheels node only takes data from the first index of the Action array. The data in the first index of the Action array tells the wheels whether or not to move forward, backwards, left, or right. We have base our movements around the concept of a tank. The EZ-RASSOR cannot move forward or backwards while turning. It must come to a stop to change its direction. Additionally, we will not be implementing variable speeds. The EZ-RASSOR moves at a fairly slow speed, only a couple of miles per hour. To keep things simple, the wheels will either be moving at a constant speed, or not moving at all.

The Arms node is in charge of moving the arms up and down. The Arms node also subscribes to the Action topic to retrieve the Action array. The second index of the Action array relays the necessary data to the Arms node so it may take the appropriate action. This data tells the arms to move up, down, or not at all, individually or at the same time.

The final node is the Drums node. It is the third node to subscribe to the Actions topic, where it also grabs the Action array. The Drums node only takes data from the third index of the Action array, ignoring all other indexes. This data informs the EZ-RASSOR whether it should spin the front drums forwards or backwards, the back drums forwards or backwards, both drums forwards and backwards, or to spend none of the drums at all.

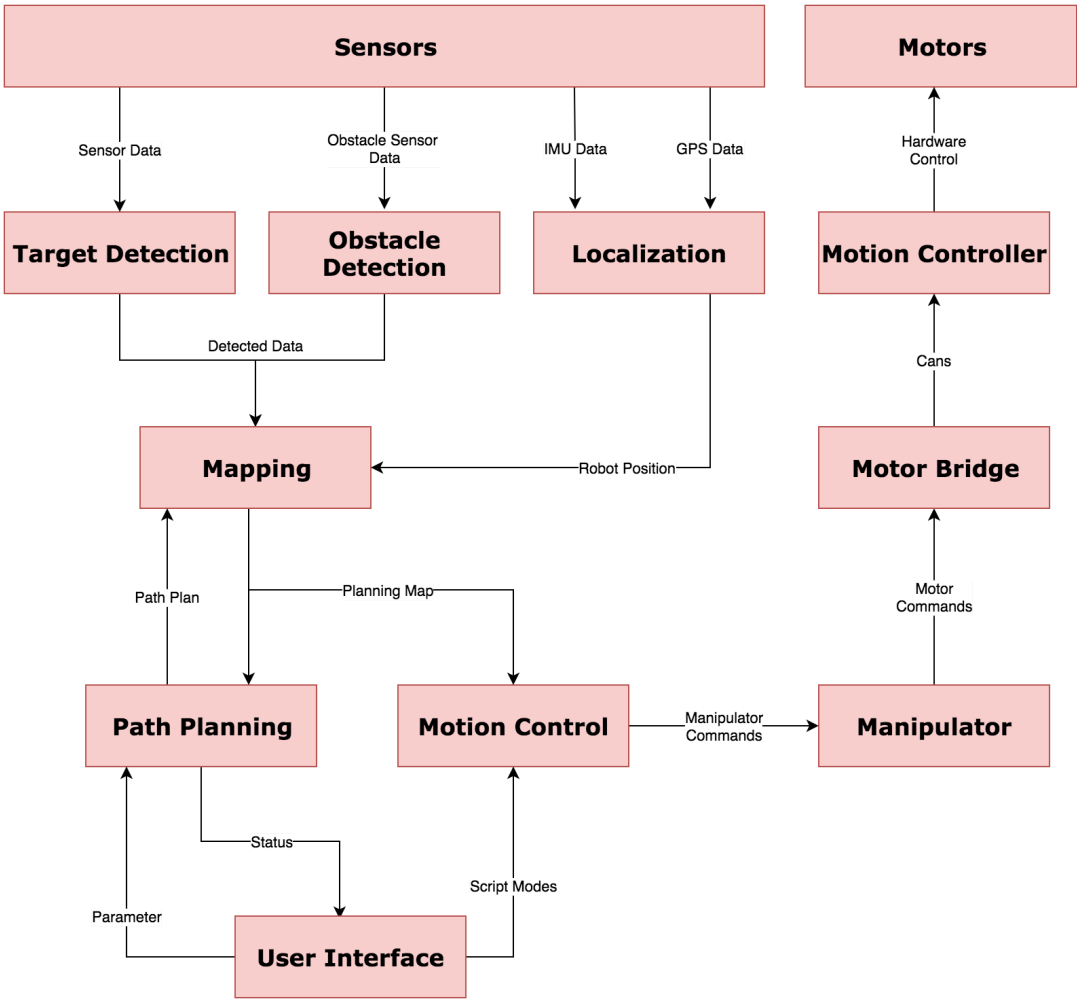


Figure 51 | This figure illustrates the ROS nodes that make up RASSOR's software. Each node is written in the C++ or Python programming language, and they communicate with each other by sending messages such as target data, obstacle data from sensors, manipulator commands, motor commands, GPS data, robot position, and Inertial Measurements Unit (IMU) data.

RC Car Plan and Execution

This project will utilize a demonstration RC car to prove the concepts of our ROS graph, as well as the abilities of the ROS graph to control wheels, movable arms, and rotating drums. The RC car is necessary for 2 reasons. First, it provides an additional testing environment besides Gazebo to test our ROS graph and its functionality. This is quite valuable, as simulations cannot possibly perfectly emulate the real world. Second, an RC car is a fun bit of physical hardware that our team can show off at the Senior Design Fair in the event that the EZ-RASSOR hardware is not completed in time.

The RC car is an assemblage of several electronics, all controlled by a central Raspberry Pi. The electronics included on the RC car are as follows: wheels with motors, servos at either end to control arms, LED rings at the end of either arm to simulate rotation, a Raspberry Pi with a battery, an accelerometer, and a servo microcontroller. The wheels and servos connect to the servo microcontroller which is wired up to the Raspberry Pi's GPIO pins. The LED rings will connect directly to the Raspberry Pi's GPIO pins. The accelerometer will also be wired directly to the GPIO pins.

The car will run Raspbian and will be fully configurable via a set of Bash scripts. Once configured, the ROS graph will be launchable via a single command. The ROS graph will include a RESTful API that will enable the RC car to act like a server. Clients (like a CLI tool, a graphical application, or the phone application) will be able to make API calls to tell the robot to move. There will be graphical and CLI tools to call some of the premade AI routines such as the self-righting routine.

RC Car Building Journal

The team has been greatly assisted by Elizabeth Nogues, an ME student and Innovation Lab employee, in the construction of the RC car. Together with Elizabeth, our team has taken an RC car kit and modified the base to support a set of movable arms, with fake spinning drums at either end. All of this is controlled by the onboard Raspberry Pi. This journal gives a walkthrough of the process, from start to finish. All references to the team working on this robot also include Elizabeth. Our team initially wanted to use a toy-engineering system to build the car from scratch, however we quickly realized that this system was not robust or customizable enough for our needs.

Figure 52

Our team decided on the Sunfounder kit described earlier in this document. Upon arrival of the kit, our team immediately realized that the acrylic base was too flimsy and would break extremely easily. The base had to be redesigned using stronger, thicker acrylic. This redesign took several attempts because of the brittle and unforgiving nature of

working with acrylic. The drawings for the base rework are shown below. As well as some of the prototype bases that either broke or had design flaws that made them unusable.

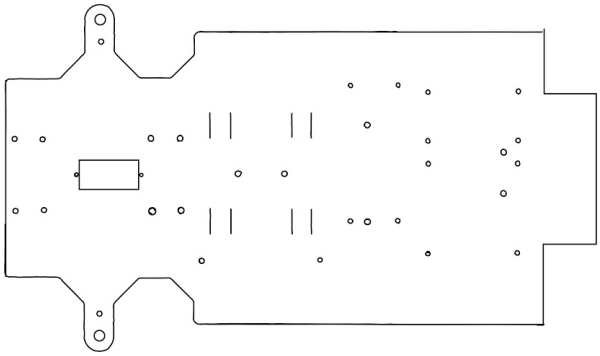
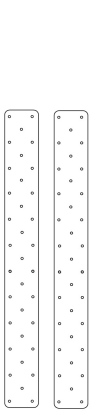
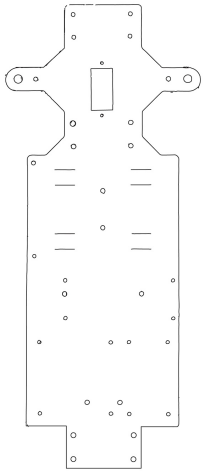


Figure 53 (top), Figure 54 (bottom-left), Figure 55 (bottom-right)





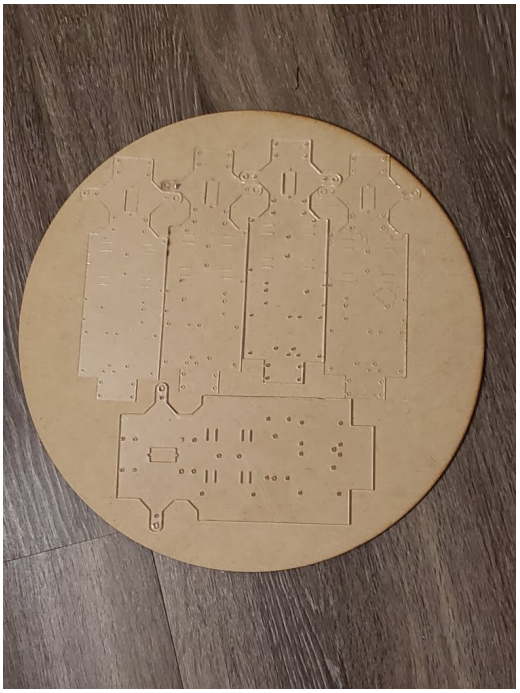
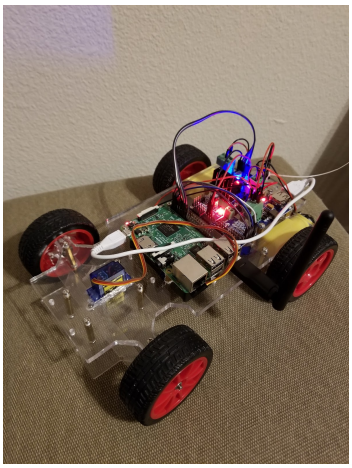


Figure 56



After the base was rebuilt from scratch, work on the arms was able to proceed. The arms were designed to use the servos in the kit originally intended for a small webcam. The arms are under construction as of this writing and have not been completed yet. The arms will also include LED lights around a drum on either end. These LEDs will be programmable to appear like rotation, so that the drums on each end of either arm look to be “spinning” without the need for more hardware or servos to spin anything.The current product, without arms, is shown in figure 56.

Scripting Details and Goals

This project will include several scripts to make it easy for end users to install and configure everything. The scripts can be broken off into different sections: installation and execution. Why is scripting necessary for this project, and what is the philosophy that will drive the design of the scripts for this project?

Scripting makes using and installing software easier for end users. This project is designed for a wide variety of people, from technical students who will be able to understand and modify the source code of this project, to children and adults with limited technical backgrounds who would just like to see the project in action. These two groups demand different things from our team, and scripts will help bridge the gap between these different types of end users to make the project more usable and comprehensible to everyone.

The scripting philosophy that this project will follow is this: write scripts and automate every process possible, especially processes that end users would otherwise have to perform to get the project to install, compile, run, or configure correctly. This philosophy is worth the time and effort to implement it because software that is easy to use is software that end users want to use. If software is exceptionally difficult to install, odds are that end users simply will not bother to install it. Software can be excellently written and function perfectly, but if it is too confusing to install or configure, nobody will ever know because that software will sit on a virtual shelf, collecting dust.

Specifically, the team desires for installation to be as easy as 1 or 2 commands, and for compilation and runtime execution to be of similar complexity. Configuration, ideally, will not be necessary at all (except for advanced users who wish to tweak our defaults).

How will this be accomplished? The main method for scripting will be using a large, central operations.sh script. This script will contain functions and links to other scripts, and will act as a single point of contact for all of the different aspects of this software. The main interaction with this operations.sh script will be through command line flags passed to the script. As a sample, here are some possible flags:

--run
: This will spin up the ROS graph and get everything going onboard the EZ-RASSOR.

--install
: This will install all required components for ROS, Gazebo, and all libraries. No additional installation should be required after running the operations.sh script with this flag.

--build
: This is a development flag that will build and install new changes to the repository onto the development system for testing.

--test
: This flag will automatically run all unit tests for the whole system.

These flags can easily be passed to the operations.sh script like so:

```
$ bash operations.sh --install
```

Figure 57

These flags will also be compoundable and may have secondary arguments to operate on only a segment of the software. Here are some examples:

```
$ bash operations.sh --install --test
$ bash operations.sh --test ros
$ bash operations.sh --test gazebo ros restful
$ bash operations.sh --run restful
```

Figure 58

The operations.sh script is the most organized way to script this software and it will be the clearest way to provide end users with a simple interface to interact

with the scripts. The operations.sh script takes all the guesswork out of installation and configuration and for that reason will make our software much easier to use for people, from Python and ROS nerds who love sifting through the details to children and observers who just want to watch the EZ-RASSOR do parkour.

Translations of Controller Inputs to Action Array

Almost all controller input is translated from a single integer that corresponds to a button on the controller into an array of integers that is published to the Action topic which the Wheels Node, Arms Node, and Drums Node is subscribed to. The first index of the array is the input for the Wheels Node, the second index is the input for the Arms Node, and the third index is the input for the Drums Node. Below are the translations from the basic controller inputs into the action array which is published to the Action topic.

Controller Input	Corresponding Command	Corresponding Action Output Array		
0	Stop	0	0	0
1	Forward	1	0	0
2	Backward	2	0	0
3	Turn Left	3	0	0
4	Turn Right	4	0	0
10	Front Arm Up	0	3	0
11	Back Arm Up	0	4	0
12	Both Arms Up	0	5	0
13	Front Arm Down	0	6	0
14	Back Arm Down	0	7	0
15	Both Arms Down	0	8	0
16	Front Drum Dig	0	0	3
17	Back Drum Dig	0	0	4
18	Both Drums Dig	0	0	5
19	Dump Back Drum	0	0	6
20	Dump Back Drum	0	0	7
21	Dump Both Drums	0	0	8

Figure 59

Note: The basic actions translated above only affect one array index at a time, however, if multiple controller inputs are coming in at once (i.e. Drive forward while raising arms), more than one node will be doing an action, thus requiring the action array.

Note: Not all controller inputs are mapped directly into an action array. For the more complex actions of the EZ-RASSOR that involve autonomous functions, the controller inputs are instead published to the Swarm AI topic for the Master Auto Node to handle the execution of these actions.These actions include Auto-Dig, Auto-Dump, Self-Right, Z-Config, and Auto-Drive.

Wheels Node Input

Input	Corresponding command	Front Left	Front Right	Back Left	Back Right
0	Stop Movement	Stop	Stop	Stop	Stop
1	Move Forward	Forward	Forward	Forward	Forward
2	Move Backwards	Backwards	Backwards	Backwards	Backwards
3	Turn Left	Backwards	Forward	Backwards	Forward
4	Turn Right	Forward	Backwards	Forward	Backwards

Figure 60

Arms Node Input

Input	Corresponding command	Front Arm	Back Arm
0	Stop Movement	Stop	Stop
1	Stop Front	Stop	No Action
2	Stop Back	No Action	Stop
3	Raise Front	Raise	No Action
4	Raise Back	No Action	Raise
5	Dual Raise	Raise	Raise
6	Lower Front	Lower	No Action
7	Lower Back	No Action	Lower
8	Dual Lower	Lower	Lower

Figure 61

Drums Node Input

Input	Corresponding command	Front Drums	Back Drums
0	Stop Movement	Stop	Stop

1	Stop Front Drum	Stop	No Action
2	Stop Back Drum	No Action	Stop
3	Front Drum Dig	Spin Forward	No Action
4	Back Drum Dig	No Action	Spin Forward
5	Dual Dig	Spin Forward	Spin Forward
6	Dump Back Drum	Spin Backwards	No Action
7	Dump Back Drum	No Action	Spin Backwards
8	Dual Dump	Spin Backwards	Spin Backwards

Figure 62

IMU/Accelerometer Implementation

The production RASSOR possess the ability to self right itself when fallen over. This is a feature we fully intend to incorporate into the EZ-RASSOR. This first step in solving the issue of falling over is to detect if the EZ-RASSOR has fallen over. This can be done with either an Inertial Measurement Unit (IMU) or a Three-Axis Accelerometer. IMU's tend to just be accelerometers with additional components, such as gyroscopes, which measure angular velocity, and magnetometers, which are similar to a compass and detect magnetic fields such as the north and south poles. However the lines get blurred as sometimes accelerometers contain gyroscopes. Both an IMU and a three-axis accelerometer can be used to sufficiently solve this problem, and the decision of which to use entirely depends on the amount of accuracy you want and if you need any additional features that an IMU would provide. However, since a three-axis accelerometer is fully capable of solving the issue, and an IMU contains a three-axis accelerometer, I will continue on from here explaining how to solve the self right issue with only the use of a three-axis accelerometer.

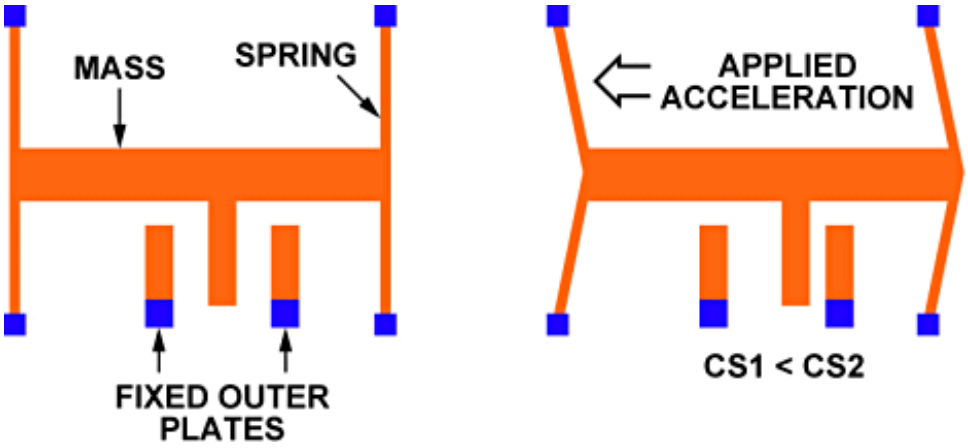


Figure 63 | Diagram of the internals of an accelerometer

An accelerometer works by measuring the capacitance between a set of electrically charged fixed metal plates and a separate electrically charged plate mounted on a spring which moves in response to acceleration [27]. When the accelerometer is moved, the non fixed plate will be shifted towards one of the fixed plates, lessening the gap between the fixed plate and spring plate, resulting in the increase of capacitance in that section of the system. A three-axis accelerometer takes this concept and applies it to all dimensions of movement. These three dimensions of movement are referred to as the “Roll,” the “Pitch,” and the “Yaw” [28].

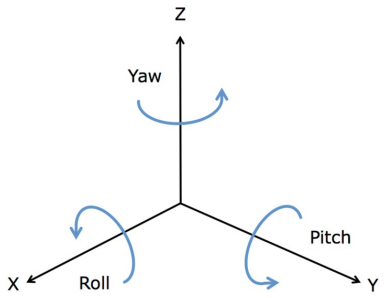


Figure 64 | Example of Roll, Pitch, and Yaw

We will attach a three-axis accelerometer to the on board computer of the EZ-RASSOR. We will orient the accelerometer so the Z-plane corresponds to the yaw of the robot, the Y-plane corresponds to the pitch, and the X-plane will correspond to the roll. When the RASSOR turns left and right, the accelerometer will detect changes in the yaw, when the RASSOR goes up or down a slope, the accelerometer will detect changes in the pitch, and when the RASSOR starts to slant towards either one of its sides, the accelerometer will detect changes in the roll. The changes in these values can be monitored by the EZ-RASSOR’s onboard computer by reading data off the accelerometers serial, which will continuously post the current roll, pitch, and yaw values. To avoid false positives of a fallen RASSOR we will need to set a high threshold in changes to the roll and the pitch. The accelerometer will detect changes in the roll simply from the RASSOR driving along the side of any hill with any slope, and changes in pitch from just going up or down a hill. We only need to execute a self right command if the roll exceeds beyond a 75 degree change from its base orientation, or a flip over command if the roll or pitch exceeds beyond a 165 degree change from its base orientation, and stays beyond this threshold for more than five seconds.

Once the roll exceeds 75 degrees for a period of five seconds, it will be safe to assume the EZ-RASSOR has fallen onto its side. If the roll or pitch exceeds 165 degrees for a period of five seconds, it will be safe to assume the EZ-RASSOR has fallen onto its back. The five second requirement also avoids starting the self right sequence if the the EZ-RASSOR is in the process of continuously rolling down a hill. Once the self right command is initiated, it will continue indefinitely until the EZ-RASSOR's roll falls back below 75 degrees.

The self-right sequence will be a series of arm movements similar to a snake's slither. The goal of the movement is to allow the EZ-RASSOR to slide along the ground until it comes in contact will some solid object (e.g. a rock). When the EZ-RASSOR comes in contact with an object, the nature of these arm movements against the object will cause the EZ-RASSOR to tilt back over onto its wheels. However, there is no guarantee that EZ-RASSOR will tilt back over right side up. This is why the flip over function is crucial to these scenarios. If the EZ-RASSOR finds itself upside down for any reason, it will be able to return to the proper upright position. This action will be completed in a series of arm movements that take advantage of the EZ-RASSOR's ability to extend its arms well beyond 90 degrees in either direction.

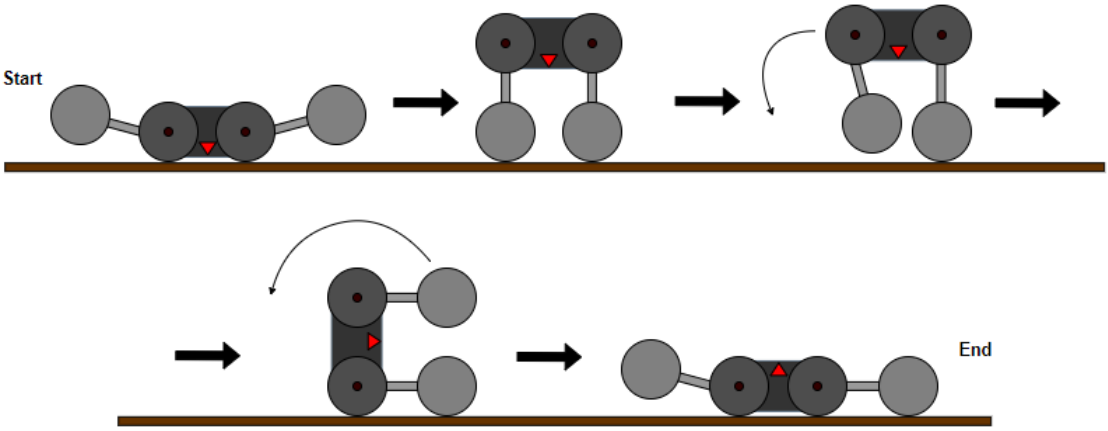


Figure 65 | Diagram of the "flip over" sequence

Once upside down for longer than five seconds, the flip sequence will begin. The EZ-RASSOR will begin to move its arms towards the ground and continue to do so until it is eventually standing on its drums. Once in this position, it will bring one of its wheels more towards the center of the robot, causing the EZ-RASSOR

to tilt to one side. Eventually the it will tilt so much that it will fall into a position where it is standing on one set of wheels and one set of drums. From here the EZ-RASSOR will swing its free arm up and over, causing a weight shift once again, resulting in the EZ-RASSOR to fall into the correct, upright position.

RC Car Backend

We have decided to serve the Raspberry Pi using a RESTful API. By using a stateless protocol and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow, by re-using components that can be managed and updated without affecting the system as a whole, even while it is running. We will be able to send commands over the internet to perform tasks, implement a mobile app controller, and network with other robots.

Installation

We're going to use a Python web framework called FLASK to turn the Raspberry Pi into a dynamic web server. In order to install FLASK, we need to have pip installed.

```
pi@raspberrypi ~ $ sudo apt-get install python-pip
```

Figure 66

After pip is installed, we can use it to install FLASK and its dependencies:

```
pi@raspberrypi ~ $ sudo pip install flask
```

Figure 67

To test the installation, create a new file called **hello-flask.py** with the code from below.

```
## Load the FLASK module into your Python script
from flask import Flask
## Create a FLASK object called app
app = Flask(__name__)
## Run the code below this function when someone accesses the root URL of the
server
@app.route("/")
def hello():
## Send the text "Hello World!" to the client's web browser
    return "Hello World!"
## If this script was run directly from the command line
if __name__ == "__main__":
## Have the server listen on port 80 and report any errors.
```

```
    app.run(host='0.0.0.0', port=80, debug=True)
```

Figure 68

Before the above script is ran, we need to know the Raspberry Pi's IP address. Run `ipconfig` to find it.

A key line of code we must consider is `@app.route("/")`. The `("/")` means whatever function below it will be called every time the root URL of the server is accessed. We can edit this to make our endpoints to perform tasks such as “begin excavation cycle” or “perform swarm task A”. We also plan to use API endpoints to implement our app controller, which we go into further detail later.

Once we know FLASK is running properly, we can now deploy the server, which must be done as a super user.

```
pi@raspberrypi ~ $ sudo python hello-flask.py
 * Running on http://0.0.0.0:80/
 * Restarting with reloader
```

Figure 69

From another computer on the same network as the Raspberry Pi, we type our Raspberry Pi's IP address into a web browser. If your browser displays "Hello World!", we know we've got it configured correctly.

```
10.0.1.100 - - [27/Nov/2018 00:31:31] "GET / HTTP/1.1" 200 -
10.0.1.100 - - [27/Nov/2018 00:31:31] "GET /favicon.ico HTTP/1.1" 404 -
```

Figure 70

The first line shows that the web browser requested the root URL and our server returned HTTP status code 200 for "OK." The second line is a request that many web browsers send automatically to get a small icon called a favicon to display next to the URL in the browser's address bar. Our server doesn't have a favicon.ico file, so it returned HTTP status code 404 to indicate that the URL was not found.

Calling the RESTful API from our React Native Controller will be as easy as making a call to any typical API. React Native provides an asynchronous Fetch API for all our networking needs.

```
async function moveForward() {
  try {
    let response = await fetch (
      'http://0.0.0.0:80/RobotMovement/Forward',
    );
    let responseJson = await response.json();
    return responseJson.reply;
  } catch (error) {
    console.error(error);
  }
}
```

Figure 71

We plan to implement the app controller in such a way that all EZ-RASSOR functionality can be achieved by these API calls. We will implement a input buffer queue to help keep API calls at a reasonable volume. For instance, If we want the robot to move forward, the user must press the forward button on the app and then the command is added to the input buffer. Once the d-pad is released, a second command is added to the input buffer letting the robot know do stop moving forward. Every few seconds this buffer is sent as an API call to the robot. The request request is read and the robot follows directs in the order received. This same basic functionality can be said for all manual movements. In regard to pre-programmed tasks and autonomous control, we can implement a set of buttons all labeled with each specific task. When a button is pressed, that specific task is added to the queue and executes until completion. The commands we plan on implementing in Phase 1 this project is listed below. *Notice that commands 1 through 17 also have a counteracting "Cancel" function when it's specified button is released.*

0. Stop
1. Forward
2. Backward
3. Turn Left
4. Turn Right
10. Arm 1 – Move Up
11. Arm 2 – Move Up
12. Both Arms – Up
13. Arm 1 – Move Down
14. Arm 2 – Move Down
15. Both Arms – Down
16. Drum 1 – Excavate
17. Drum 2 – Excavate
18. Both Drums – Excavate
19. Drum 1 – Dump
20. Drum 2 – Dump
21. Both Drums – Dump
25. AUTO_DIG
26. AUTO_DUMP
27. SELF_RIGHT
28. Z_CONFIG
29. AUTO_DRIVE

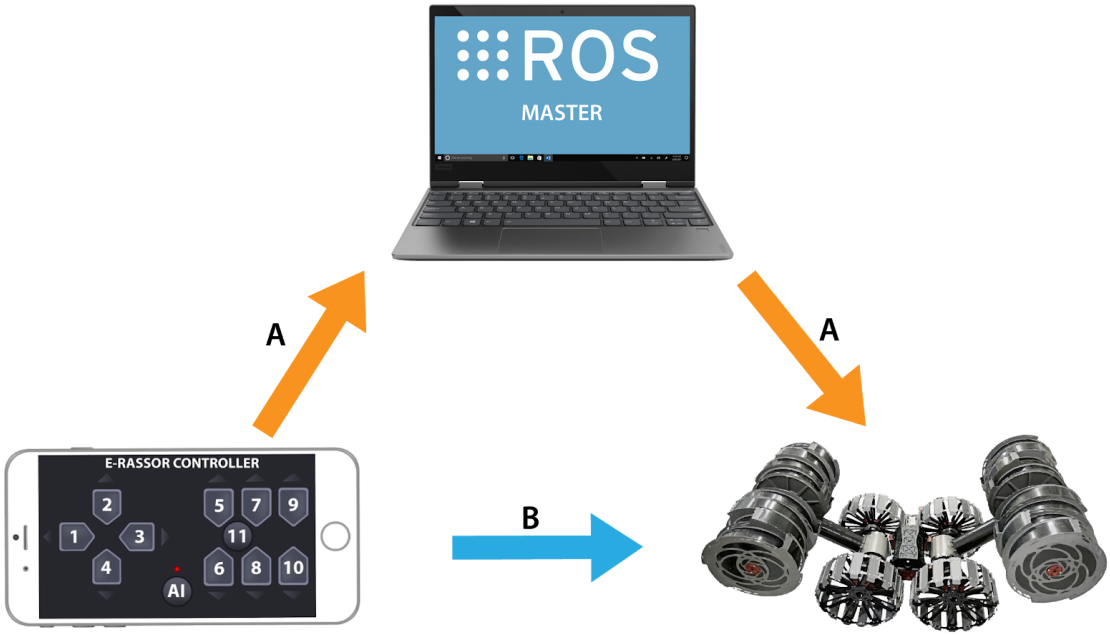


Figure 72 | This displays the different paths that the mobile app can communicate with the EZ-RASSOR

The two command paths considered are paths A and B shown above. For the first stage of this project, we plan on moving forward with path A. This is because the team does not currently have NASA's hardware. Instead we are using a Raspberry Pi robot, which we dont believe will be able to handle the computations needed. As our robot's hardware becomes more powerful, we will then implement path B for improved response times.

Test Plan for Self Right (Real World)

Once the EZ-RASSOR is assembled and the three-axis accelerometer is a attached, we will test the self right function and the flip over function in a linear fashion. Upon completion of a phase, we will move to the next, more complex phase. Each phase increases in difficulty and demands of what the EZ-RASSOR must accomplish. It should be noted that in the first two stages of testing, we will likely not use the actual self right and flip over sequence, but simply some basic action, like turning the wheels, to show the accelerometer is working correctly and that the onboard computer is successfully reading the data off the accelerometer serial and publishing it onto the accelerometer ROS node.

The first and most basic test will be an accelerometer side test. We will simply hold the EZ-RASSOR in the air and slowly tilt it to its side, increasing it's roll. We will continue this roll until it has been rotated 90 degrees and the side of the

EZ-RASSOR will be facing straight down. We will hold it in this position for at least ten seconds. However, if things are working properly, the self rite sequence should initiate after five seconds. We will run this test four times. One for a left roll at 90 degrees, one for a right roll at 90 degrees, one for a left roll at 76 degrees, and one for a right roll at 76 degrees.

After the accelerometer side test, we will begin the accelerometer flip test. This test is similar in nature to the side test. We will hold the EZ-RASSOR in the air and slowly rotate it until it is completely upside down. We will hold it in this position for at least ten seconds. However, if everything is working properly, the flip over sequence should initiate after five seconds. We will run this test eight times. One time for a left roll of 180 degrees, once for a right roll of 180 degrees, once for a left roll of 165 degrees, once for a right roll of 165 degrees, once for a forward pitch of 180 degrees, once for a backwards pitch of 180 degrees, once for a forward pitch of 165 degrees, and once for a backwards pitch of 165 degrees.

Following the completion of both the accelerometer tests, we will initiate the real world flip test phase. In this phase we will place the EZ-RASSOR upside down and allow it to attempt to flip itself back over from a completely upside down position. We will lay the EZ-RASSOR onto its back on a flat surface and wait. After five seconds the flip sequence should begin. We will test this sequence for both flat land and various sloped terrains. Since the flip sequence is a crucial part to getting out of any fallen situation, we will remain in this phase until the EZ-RASSOR can consistently flip itself over from an upside down position.

Upon completion of the real world flip test, we will move onto a flat land test. For this stage of testing we will be using the actual self right sequence. We will begin by placing the EZ-RASSOR directly onto its side. After five seconds the self-right sequence should initiate and the arms should begin to move in a manner similar to a snake’s slither. If we have implemented the sequence properly, the EZ-RASSOR should begin to wiggle across the floor. We will run this test at least two times, once for each side. If the sequence does not cause the EZ-RASSOR to move or if the result of the sequence is not up to our standards, we will not leave this stage of testing until the self right sequence produces the desired results.

Once we have completed and obtained the desired results from the flat land test, we will begin our basic real world test. This test is very similar to the flat land test, starting by placing the EZ-RASSOR onto its side. However, in this test, we will place sturdy objects in the environment around the EZ-RASSOR so it may attempt to successfully self right for the first time. The goal of this test is to have the EZ-RASSOR initiate the self-right sequence, move through the environment on its side, come in contact with one of the surrounding object, and use the object to successfully right itself back onto its wheels. Since this is a core test of the self-right feature, there is no telling how many times we will run this test. We will run it numerous times on both side, with various sized objects with various amounts of sturdiness. We will stay in this phase of testing until we feel the complete action of self righting in the most basic scenario is consistently successful. We will not move on until the EZ-RASSOR can self right itself using various size objects regardless of what side the EZ-RASSOR is on or the distance to the object.

The next phase of testing will be the advanced real world test. There are many situations in which the EZ-RASSOR may fall over. A prime example is if it fell over while on the side of a hill, or it may fall onto a rock. The fact that the EZ-RASSOR may not always fall onto flat land means we need to be prepared for those situations. In this phase of testing we will place the EZ-RASSOR, on its side, in numerous different situations. These various situations boil down into two main categories: sloped environments with various degrees of elevation, and extremely rocky environments which may result in the EZ-RASSOR laying on top of one or more rocks. To execute these tests, we will take the EZ-RASSOR outside and look for different hills and rocks to place the robot into a fallen scenario. We will strenuously test all the different situation we can think of that the EZ-RASSOR may find itself in. Although we intend for the EZ-RASSOR to be able to right itself from the more basic variations of slope and rocky environment, it is possible that the high level of proficiency needed to complete this phase of testing may not be accomplished until phase 2 of the project. We will deem this phase of testing complete when the EZ-RASSOR can successfully right itself from the most strenuous situations the vast majority of the time.

ROS Test Plan

The ROS graph will be tested in 3 main ways: through Python Unittest unit tests, Python Unittest acceptance tests, and shell script acceptance tests that monitor the ROS graph as it runs. More details for each follow.

Python Unittest Unit Tests

All of the nodes written for our ROS graph will be written in Python, and Python includes an excellent unit testing suite called Unittest. This test suite allows us to write small tests for each new function and new class that we create. These tests will cover edge case inputs and expected outputs, however they will not be excessively thorough (otherwise the team would produce more tests than actual code). They will test standard base cases like 0, negative numbers, the None type, and a handful of others.

Python Unittest Acceptance Tests

Acceptance tests have a larger scope than unit tests (unit tests test a single unit of code). Our team will write many acceptance tests for this software that will test things like chains of functions and the overall functionality of any particular ROS node from start to end. For example, one of our nodes will read directional data commands and emit wheel commands to topics that other nodes are reading from. We will write acceptance tests that input dummy directional data commands to this node and expect certain wheel commands to be emitted as a response. Tests like this span the entire node, not a single or group of functions, and thus are better categorized as acceptance tests rather than unit tests. These tests also typically employ nominal (standard, non-edge-case) data.

Shell Script Acceptance Tests

The fully populated ROS graph for this software will be as many as a dozen or so nodes and as many topics. This situation will be complex, and so it will be difficult to write effective Python Unittest tests that cover the system from beginning to end. To test the whole system at once, our team will write shell script acceptance tests whose scope contains the entire ROS graph. As an example, a test might start by simulating a call from a control app to execute an AI procedure, such as the auto-flipping procedure. This simulated call will be interpreted by the ROS graph inside the central processing nodes, which will then give control to the AI nodes. The AI nodes will produce movement instructions that will use the wheels

and arms to attempt to correct the orientation of the robot. All of these nodes communicate through topics and messages which will be monitored using tools that ship with ROS. These tools will be glued together with scripts into tests to ensure that the entire system is operating nominally, from initial simulated call to corrected (or not) robot.

Gazebo Execution Plan

One consideration that must be made with Gazebo is the varied hardware that the team is developing on. Gazebo is only officially supported on Debian based environments. Ideally it would be best if everyone on the development team can run Gazebo natively on their machines. Unfortunately as we have come to find out, it is not feasible for everyone. Gazebo is a mission critical program, that will be required to be run by everyone on the team. There have been ports of the software to other operating systems, but they usually lag behind the main version, and encounter some other issues that come from running in a non Linux OS. Initially we tried to run Gazebo through a virtual machine, but this proved not very effective as performance takes a significant hit, compared to running it natively, specially when we consider working with the Swarm AI which would take significant processing power. This being the case, as will be mentioned on another section, the implementation that we are going to go with is going to be through a container. There is an official container that contains this software along with ROS that will be used in order to get everyone working on the same page.

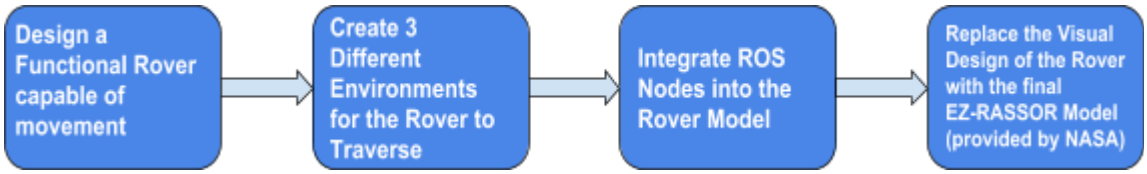


Figure 73

This flow diagram is the high level plan that the Gazebo sub-team will be following to satisfy the simulation requirements. This team will begin by first creating a functional rover with 4 wheels. The initial build will also consist of two drum arms on each side of the rover to simulate this hardware requirement from the final design. Since the development of the simulation will be incremental, this is the ideal place to begin this portion.

The initial model for the rover may not be a replica of what we anticipate the final product to appear like. This is because the NASA Swamp Works team will be working on the final hardware design in parallel to our development efforts. Only towards the end of this project will we receive the final design models from the Swamp Works team. At this point, we will be able to update the aesthetics of the rover without altering any of the added functionality that our team is responsible for creating.

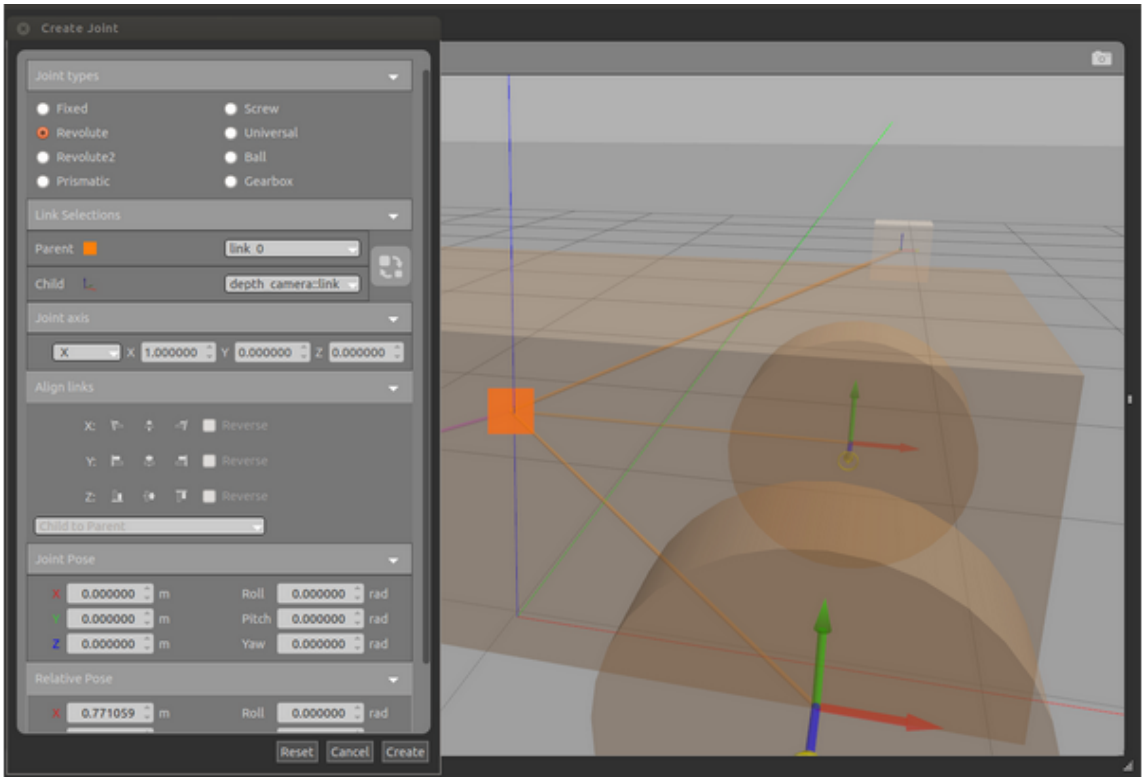


Figure 74 | Screenshot from Gazebo showing a very simply designed vehicle

Once a functional rover is designed, we will continue the simulation process by designing three environments in which the rover can operate. The original RASSOR was designed for operations on the Moon and Mars. Since our product will be used as a demo unit for the RASSOR, we will produce similar environments in our simulation. Specifically, we will create the following environments:

1) A flat Earth environment that will serve as our control environment. This will be a simple environment inside an enclosed room with square boxes scattered around.

2) A dirt-filled Earth environment, complete with obstacles, and rock formations. This environment will have a day mode and a night mode so that we can accurately simulate how the stereoscopic camera will behave in either environment.
3) A rock-filled Earth environment, that also contains patches of dirt. This environment will provide the biggest challenge to the successful functionality of the rover as it will need to navigate across obstacles and identify dirt areas that can be dug through.

We intend on pulling multiple resources from the open source community while creating the environments. We are more likely to find specific environment objects such as a rock or dirt and are less likely to find completed environments for outdoor areas that will suit our testing. Having to design these environments by hand will ensure that we have complete knowledge and control over environment variables and can work with NASA subject matter experts (SMEs) to fine-tune the details.

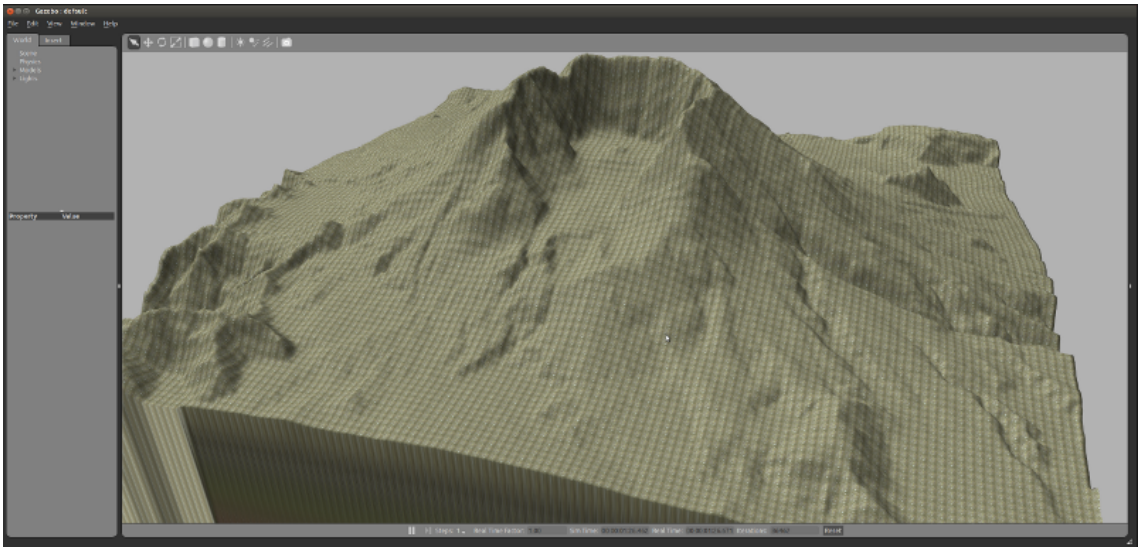


Figure 75 | Example of a rocky terrain environment created in Gazebo

With functional rover and environments at our disposal, we will then work with the ROS sub-teams to implement the ROS nodes and functionality that will be ported over to the final design of the EZ-RASSOR. We will begin by creating a ROS plugin in C++ to add the ROS nodes to a given environment and model. We will ensure that the nodes load correctly and are following best practices as provided

on both the ROS and GazeboSim websites. Once these plugins are created, they will be ready to be run to demonstrate specific EZ-RASSOR / ROS functionality.

Once all of the environments are designed and we have a functional rover to traverse the environments, we will continue working on the EZ-RASSOR. Our work will be to apply granular details to the device such as it's exact weight given the components we intend on purchasing and details concerning rotors and friction. The intention will be to make the model function as close to the real device as possible to ensure that our simulations are accurate tests for the rover. This part of the process will also require heavy input from the NASA SMEs as we would like to use similar tweaks to what the Swamp Works team has done with their own RASSOR. Most of these tweaks can be done through the Gazebo UI but in cases where we need to modify some aspect of the physics of the rover that do not exist in the simulator UI, we will have to modify the C++ scripts for the model. All model scripts will be backed up to our GitHub repository so that they can be easily maintained and shared.

After all three of these pieces are completed, we will have a fully functional simulation environment for the EZ-RASSOR. This is crucial because even if there is a delay in obtaining the final model, this simulator portion can still be shared with future senior design teams to allow new contributions. Additionally, it will give our team a chance to test our designs before building the hardware with the main hope being that the simulations will help us find any hardware changes we need to make for the success of this project.

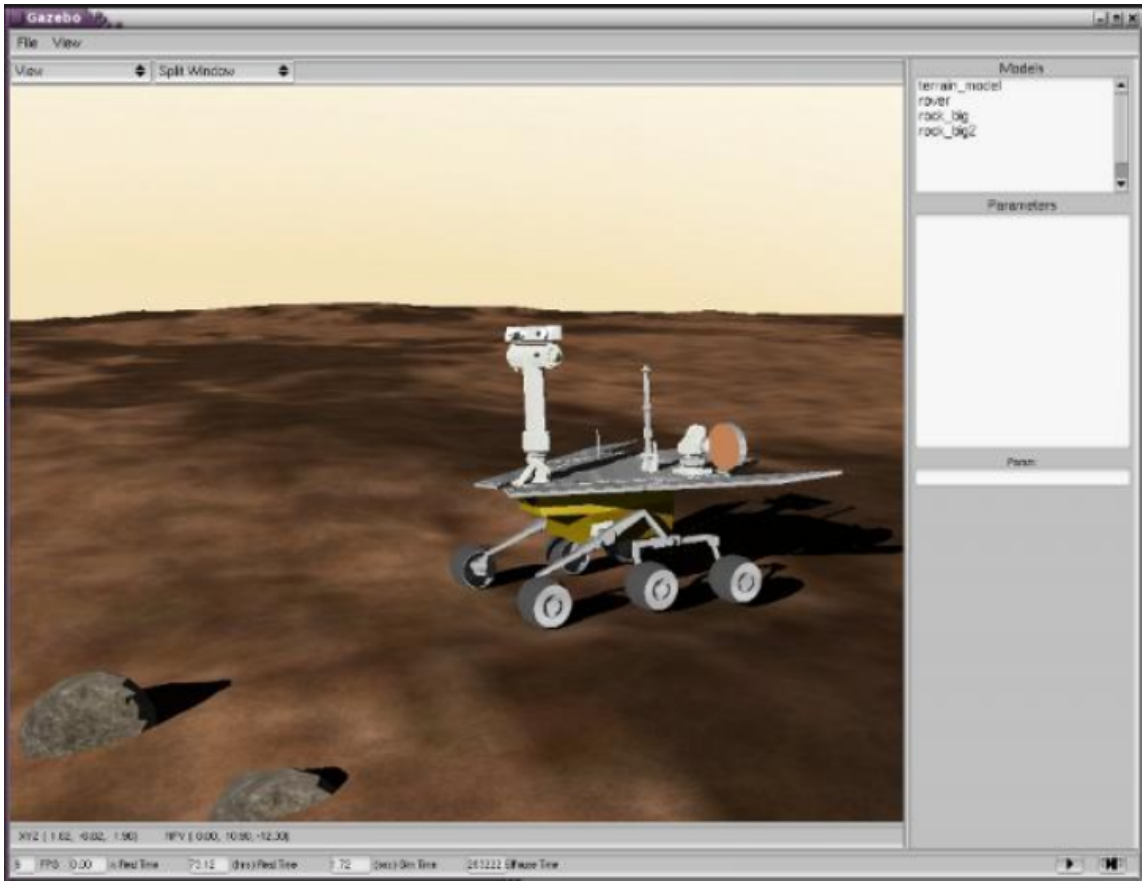


Figure 76 | Screenshot of another Mars Rover, the MER, operating inside of Gazebo

Gazebo Test Plan

As Gazebo is a simulation software, much of its purpose is to, in and of itself, test other modules of our project software. However, one of the guarantees that we are able to provide is a fully functional simulation that is independent from any hardware delivery from Swamp Works. As such, we will be performing incremental tests of of simulation functionality as they are implemented. All Gazebo testing will be performed on an Ubuntu 18.04 machine with an NVidia GTX 1050Ti graphics card installed.

Once we create a functional rover with 4 wheels and two drums on the front and rear, we will perform a test to demonstrate successful movement. This is a very basic test and only involves verifying that the rover moves in the directions that we are intending, without any issues or screen clipping. Further gazebo development will be halted until this test can be verified.

After this basic functionality is verified, the team will move onto designing the aforementioned Gazebo environments. Upon the completion of these environments, a separate test will be performed in each environment. The purpose of the test is to ensure that collision detection is registering in Gazebo and that the model is responding appropriately to changing terrains. For the control rool with boxes, part of the testing will be to maneuver the rover into obstacles to see if it reacts as we would expect, either pushing the object forward or halting movement (in the case of a wall). In the other environments, we will have expanded expectations such as the rover needing to angle downward and continue moving when reaching a ditch, and being able to angle itself upwards to exit a ditch. We will not be testing for the gravitational correctness of such a maneuver as we are not yet taking gravity into account.

Following this, we will implement ROS functionality into the EZ-RASSOR simulation. Once this is complete, we will test out the ROS nodes by following the steps outlined in the previous section “ROS Test Plan”.

With ROS functionality validated, there is only one more incremental test to perform: a test to validate the updated physics of the EZ-RASSOR. After we carefully modify the configurations on the rover for things such as weight, we will need to demonstrate its accuracy by once again maneuvering the EZ-RASSOR on each of our environments. These environments will now have Earth’s standard gravity of $9.8\,m/s^2$. We will re-test the ability of the rover to push object by driving into them, especially how the rover reacts to driving into an significantly heavier or immovable object. Additionally, we will observe the ability of the rover to drive itself out of a ditch. In our testing of a rover inside of a ditch, we will expect that as the steep increases (from 0 degrees towards 180 degrees), that the rover will have an increasingly harder time in exiting the ditch. Eventually of course, we expect that the rover will simply not be able to exit the ditch given the angle of inclination. Last in this series of tests is to validate that the rover will slow down on its own when traveling in a straight line due to the normal friction that results from Earth’s atmosphere.

After all of these increments have been tested, we will perform an acceptance test of the simulation to ensure that we have satisfied the requirements of the simulation. Using the same environments that have been developed, we will re-test our ROS nodes given the new physics constraints to demonstrate the

functionality of the rover. The results of this test will be verbosely recorded as it will be against this data that we compare the results of any real-world EZ-RASSOR test.

This testing will satisfy the completion of the Phase 1 Gazebo requirements. Any Phase 2 requirements that are implemented will need to have their own Gazebo test plan defined which will need to be followed as well. If the added functionality will be used in the project then an additional acceptance test will need to be performed to ensure that no other functionality has been comp

Test Plan for Self Right (Gazebo)

Upon completion of a URDF Gazebo model, with all the necessary sensors, we will be able to begin testing the self right and flip over sequences in a simulated environment. These tests will be very similar to the ones we plan to do in the real world, but since we are in simulation we can test much more extreme scenarios. Being in Florida, extremely hilly and rocky environments are hard to come by. Additionally, the physical EZ-RASSOR isn't cheap, and we wouldn't want to risk damaging it. Thankfully, neither of these are an issue in simulation. Gazebo will allow us to repeatedly test these sequences in multiple different environments, and tweak the corresponding code much faster than in the real world.

The first few phases of testing will be to insure the simulated sensor is working. We can test this much more quickly in simulation than in real life. All we will have to do is print the accelerometer data to an output window and check to see how the values change when we place the EZ-RASSOR model on its side and back. When placed on its side we should see a +/- 90 degree change in the roll, or the X-axis, of the accelerometer. When placed on its back we should see a +/- 180 degree change in either its roll or yaw (Y-axis). If the outputs match the amount of degrees the model has been shifted, we will know the simulated sensor is working correctly.

Once we know the simulated sensor is functioning as intended we can jump straight into testing the flip over sequence. We will spawn the EZ-RASSOR model on its back into the default Gazebo environment. After five seconds the flip over sequence should begin. We will run this test as many times as necessary, editing the code for the sequence along the way to get the desired result. Once the EZ-RASSOR model can constantly flip over constantly, we will move on.

After testing the flip over function, we will begin to test the self right sequence. We will spawn the EZ-RASSOR model on an arbitrary side into the default Gazebo world. After five seconds on its side the EZ-RASSOR model should start the execution of the self right sequence. The ideal result would be for the model to start to “slither” across the floor, moving in a singular direction. Once successful in getting the EZ-RASSOR to slither, we will extensively test placing the model on both sides, ensuring the sequence works as intended regardless of what side it fell on.

Once the EZ-RASSOR model can consistently flip itself over and slither regardless of what side it is on, we will begin to place the model in various simulated environments to test how these functions will hopefully behave in the real world. These environments will be used to test situations where the EZ-RASSOR has fallen on its side or on its back in inclined environments with various degrees of slope, rocky environments where the EZ-RASSOR is not flush with the ground, and combinations of both of sloped and rocky environments. Testing the self right and flip over sequences in all of these different situations will result in the EZ-RASSOR to have a very robust ability to recover from nearly any situation where it has fallen over.

Docker Execution Plan

In order for the team to be able to work together seamlessly, without much issue we will require some sort of shared environment in which we are guaranteed certain core functionalities are confirmed work. This is especially an issue considering the nature of the toolsets we choose to use. Gazebo for example is only officially supported on Debian based Linux/GNU platforms (Although you can compile it to any other), and ROS is the same story. Consulting our sponsors they use and highly recommend developing on a Ubuntu environment as it’ll make the compatibility with almost all the software packages the easiest.

Unfortunately, with the team's wide reaching hardware we have it seems, highly improbable and logistically difficult for everyone to be able to natively run Ubuntu on their machines. Most would have to reformat and partition the machines, some even wouldn’t be able to get Ubuntu installed on their machines, due to hardware lockdown. Initially we had considered and even begun working on using a Ubuntu virtual machine for everyone to be able to have and develop in. Although even early on it’s flaws were clearly seen. The idea in principle was for

someone on the group to get an image from scratch and install the software packages needed to fully develop this project. Everyone would list out the software packages that would be needed and install them on the virtual machine. This would make testing and developing on other contributors code much easier for everyone. The main problem with using a virtual machine was the host hardware. For many our main development machine is some form of laptop. Our initial tests were conducted on a relatively new i5 Surface Pro. Just initializing the major performance hits were noticed, even just when idling just running the GUI. Even just simple text editing and running commands from the terminal proved to be a challenge for most on their hardware. It seemed like this implementation of having a shared workspace was going to be inefficient. Even more so the problem was that most of us would end up using a virtual machine or just programming on the hardware directly (eventually when we got to the other step.

Other options were to be explored, even just trying to run the ported versions of the software on each others native platform. But this seemed like it would create the most problems down the line that it would not even be worth it. Issues with incompatibility compiling, and just testing each others modules seemed like it was not going to be a good fit. Fortunately we came across containerization, specifically Docker. ROS features an officially supported version of it's software in container form. Which would make the development environment platform independent, just like the virtual machine implementation, all while at providing closer to the native host's performance.

ROS provides this simple to use docker container, that can easily be cloned and used. Even more so they have different containers that provide even more software bundled that is often used. One of these being Docker and Gazebo bundled together. Having our main components already available to use in a container. Even going as far to include extensive documentation. Which will especially be useful as most of us do not have any former or proper experience with containers and Docker. This does present an issue though, our initial expedition in how to work with this implementation, as there are a couple ways of doing this, which will further be elaborated in the test plan.

The main concern being how the Gazebo integration will work. The most recommended way of doing this is for Gazebo to actually be ran on the host OS rather than the Docker container, as you'd have full on GPU acceleration and better performance. There is a Docker container that does have a way of running

the X11 Gazebo GUI on through it but it is not straight forward; Mac support is limited, it requires configuration and does not support GPU acceleration. Which will cause similar problems that we experienced on the virtual machine. Rather the recommended setup to do is to setup ROS to broadcast through the network and have port forwarding setup on the container such that you can access it from the host machine. This would yield the most benefit as all the logic programming is done through ROS in its native environment while Gazebo, which is just showing the simulation would run at native performance having access to ROS running from the container. There are complications to this though. It is not simple to set up, it requires to have a foundational understanding of how ROS works and how the networking must be done in docker so you're able to connect to it from outside. This currently is still a work in progress but for the most part it will be the most effective solution. In the case that this implementation does not work the way we expect we will just try to use the built in Gazebo through X11.

Regardless of implementation, keeping a standard development environment across the board will be a challenge, but we are trying to mitigate the issue as much as we can by implementing Docker containers. Idealistically we would in the end have a container that would be able to compile and run all the files from this project, and include that container as part of deliverables. This way anyone coming in to work on the project for the next phases will have no issue picking up from where we left off.

Docker Test Plan

Since Docker itself is not a component of the final project, rather an interface for the development tools that are gonna be used, testing docker itself won't be necessary. Rather what will need to be done is to make sure everyone is able to run the container instance of ROS and Gazebo.

Thus the testing that will be done in the case of Docker, is going to be the creation of an image with ROS and Gazebo, and all the other necessary tools for the development of this project. What what be considered a 'successful' image is one single container that can be deployed easily in any other environment, all while being able to execute, compile and run all the deliverables.

We'd begin with the official image provided by the ORSF which is the the Open Robotics Software Foundation. They are the ones that develop and maintain

ROS and Gazebo, and they provide these images that contain the full installation. This container would be built upon and essentially forked. From this container all of the development software will be installed and everything will be tested to make sure that it will work.

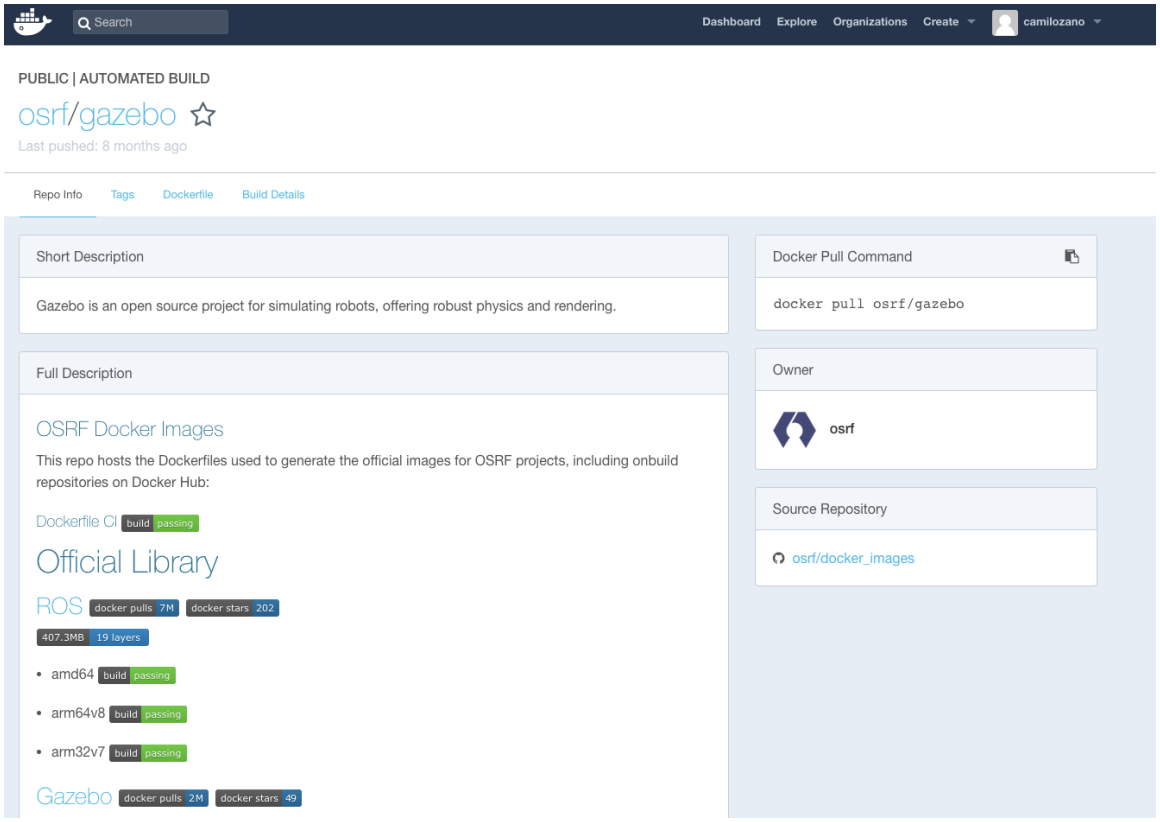


Figure 77 | The main repository by OSRF that will be used for the creation of our Docker container.

Even as far to possibly include the development tools used for compiling the mobile app, which would be React Native, and all of its dependencies, as well as Flask, for the routing that is going to used.

To run Gazebo we will be forwarding the ROS output through the container to the host machine. This will allow us to run Gazebo natively on the host machines, providing the best performance while still being able to make sure all of the logic is running in its proper native environment. Should issues arise from this implementation, where Gazebo could not be run on the native environment, then the best case would be to run Docker natively, though doing so would take a performance hit on rendering.

Should it all come together well in the end, what is going to be runned on the Docker container should be directly transferable to the RC car and to the hopefully the final hardware when it arrives. This will all come at the cost of having a slower startup all while everyone gets comfortable in using this development environment.

Packaging Execution Plan

This software needs to be packageable and distributable to facilitate widespread adoption and use of our code. As this software is being written for people to learn more about the EZ-RASSOR, it must be easily shipped to NASA for deployment on EZ-RASSORs but also easily shipped to people online who wish to build their own RC car demonstrations. Because the software is so modular, it also can be split into components and shipped individually. The ROS nodes may end up being so useful that they could be deployed in other robotics projects and so it would benefit the team to make them easily available.

The options for distribution include using PyPI (for the Python nodes), using ROS's custom packaging and shipping technologies, using Docker images, or only shipping through GitHub. Each option is discussed below:

Shipping via Python Package Index (PyPI)

Python packages can be distributed on PyPI for free to other users. They are downloaded and installed through the handy PIP command line tool that frequently comes with Python installations. This tool alone is enough of a reason to ship nodes through PyPI, because PIP makes installation and setup really, really easy. However, the team will not be distributing over PyPI for 2 reasons. First, this software contains a good amount of non-Python code and scripts which makes using PyPI more complicated. Also, shipping ROS software via PyPI is generally more complicated than it needs to be, and is less standardized than using ROS's built-in tools.

Shipping via ROS Build Tools

ROS provides several tools to build ROS nodes from scratch, like Catkin. Catkin wraps CMake and compiles source code into functional ROS nodes that can be installed onto a system. This is a good, standard way to distribute ROS software however it will not cover all code distribution if we have to include non-ROS software (such as our control apps). For this reason, shipping our ROS nodes

may be done this way but an additional method is needed for completely automatic packaging.

Shipping via Docker Images
One of the most intriguing packaging solutions available to us is Docker. If we build our environment inside of a Docker container, we can distribute that container very easily. This container would also be configurable to our exact specifications. This solution is the most likely path we will take to package our software, however if it proves more complicated or less workable than we initially thought, we may default to the next, and final, option.

Shipping via GitHub
Probably the least complicated way to distribute our software is to leave it on GitHub with clear instructions that explain how to compile and install it. This solution will require the least amount of time to implement, however it is not as professional or clean as shipping a Docker image or Catkin ROS nodes. The code will be left on GitHub no matter what, but the optimal (and planned) solution is to combine this option with a Docker image.

Mesh Network Execution Plan

When determining the best possible route for our mesh network where are several points that we must consider before moving forward. First of which is the communication range. If the range is fairly small, the robots will not be able to communicate unless they are nearby. On the other hand if the range is too big, we could encounter communication jams because every robot would listen to every message that is released.

Next we must consider the communication area. Ideally, robots cover 360° of space in communication protocols. Although, if communication is divided amongst multiple units, a 360° communication area can only be assured for only approximately 50% of the communication range. What this leads to is sustainability issues and could lead to problems in awareness in swarm behavior.

Another potential issue we must consider is the length of our messages to the swarm. Typically short messages would entail unnecessary overhead due to the excess additional information that is attached to the message. In contrast, excessively long messages tend to fail more frequently than shorter messages.

This means we must find equilibrium between a long and short message. In relation, we must also be aware of the amount of time it takes to convey a message throughout the entire swarm. This opens the idea of message forwarding to process and transfer messages. For example it will take considerably longer to send a large message to a robot that far from the host than one that is close. This opens up several pitfalls because robots could receive the same calls asynchronously, hindering swarm functionality. Also, at far ranges the message could be potentially lost. We could possibly overcome this problem by dividing a larger message into smaller sub-messages that get passed along from the robots near the host to ones far away. Once the far away robots receive the initial message, then the nearby robots could receive the same initial message immediately from the host.

The final concern we must consider is interference in our signal. This is unavoidable. Nearly any type of wireless communication must handle interferences as well as auxiliary external influences.

> When researching previous studies, we found that nearly everything electronic will interfere with our signal in some way. Below are a few examples of how a Wi-Fi network can be interrupted by either a microwave or security cameras. [29]

These examples are some of the most average cases in Wi-Fi interference. In our day-to-day web surfing, these interferences are unnoticeable due to our computers handling the data-loss when these issues arise and reprocess the request. When developing our mesh network, we must ensure the authenticity and validity of our request, and that we follow suit and be able to handle these interruptions in our network if any data is lost or corrupted.

In relation to ROS, the implementation of a mesh network should be faily streamline. Initially, network connectivity must be set up between the robots. Network configurations within ROS must meet two conditions.

- Requirement 1: There must be complete, bi-directional connectivity between all pairs of machines, on all ports.
- Requirement 2: Each machine must advertise itself by a name that all other machines can resolve

One might assume the first requirement is met through a couple of simple pings, but ping only checks that the ICMP packets can get between machines, which is

not enough. We need to ensure that we can talk over all available ports. This proves to be difficult because there are over 65,000 ports available. We can use netcat to test communications over arbitrary selected ports, but this can be time consuming.

In regard to the second requirement, we look into the topic of name resolution. When a ROS node sends out a topic, it holds a URI with a hostname/port combination that other nodes will contact when they want to subscribe to that topic. It is important that the hostname that a node provides can be used by all other nodes to contact it. The ROS client libraries use the name that the machine reports to be its hostname. This is the name that is returned by the command hostname.

After the network has been set up among all robots we want to include in the swarm, we can utilize various 3rd-party ROS packages for the swarm framework. One specific package I have been interested in is the open-source *micros_swarm_framework* package. This package would enable us to perform several different swarm tasks such as creating multiple swarms, joining and leaving swarms, executing swarm tasks, talking amongst different swarms, and a variety of other features.

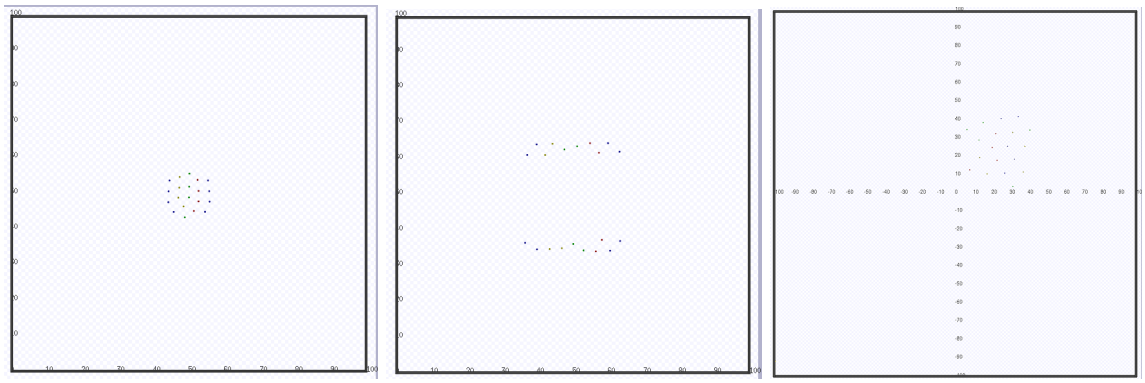


Figure 78 | The three pictures (shown above) depict some to the functionality included within the micros_swarm_framework. This includes motion and spatial coordination (left), separation into multiple swarms (middle), and flocking (right). Open-Source

We may or may not utilize the *micros_swarm_framework*, but using It would at least give us a framework to build off of to achieve the tasks required of us.

Mesh Network Test Plan

Implementing and testing a intelligent mesh network can prove to be fairly difficult. One might argue that the on-board artificial intelligence could always be “smarter” with the absence of an intelligence cap. To prevent this we will be tracking our progress through procedural milestones until we reach our desired result. Meaning, if our mesh network successfully completes a swarm task without any flaws, then we have achieved that milestone and we can move to the next.

The structure of our tests will be as follows. Each test will have a number associated to it which corresponds to its difficulty. These numbered tests will then be placed into a ordered list starting from the lowest numbered test (easiest) to the highest numbered test (hardest). We will not attempt to execute any test on the list until all tests before it have passed entirely. During the team’s weekly meetings, we will perform these regression tests to ensure that we are consistently making progress and that we don't commit any breaking changes.

Our problem became significantly easier when ROS was chosen as our primary development tool. This is because ROS allows us to execute out tests automatically. When developing our initial simulation, this will save us quite some time. Although, when we receive the physical robots, tests will receive a pass or fail by visual inspection. The two major testing frameworks supported by ROS are *gtest* for C++ and *unittest* for Python. One of these must be deployed with a package called *rostest,* which allows us to perform full integration testing across multiple machines.

Driver Station Execution Plan

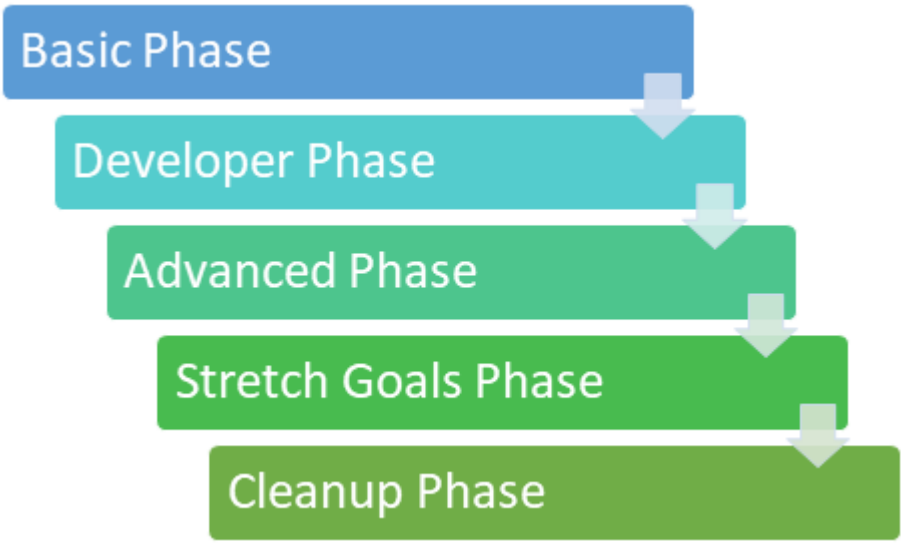


Figure 79 | Shows a workflow diagram of the driver station execution plan

The execution plan for the development of our driver station can be broken down into five phases. Within each phase a new GUI must be designed and developed using QT Creator and PyQt to contain any new features that are added. Once developed, the GUI needs to be integrated with ROS and have the functionality of each feature tested. Up until the cleanup phase, we plan on accumulating any additional features implemented from the previous phase. However, this plan may prove to be cumbersome as we have a limited amount of screen space that can be allocated to each feature. Once the GUI of a phase has been tested and debugged, all team members will have access to it in order to optimize the development process.

The basic phase implements features that are necessary, simple, and require little to no reliance on the systems being developed by the rest of the team. This phase will implement the raw camera footage from the EZ-RASSOR so that settings can be tweaked in order to optimize the video quality and frames per second. It will also implement the use of the gamepad controller to ensure that the input is being received and transmitted correctly. The phase will also display very basic information on the condition of the EZ-RASSOR such as its battery level. Finally, the phase will also implement the info terminal as it should be simple enough to implement while being very beneficial to the development team. The main goal of this phase is to test out the development and

implementation of the GUI as quickly as possible.

The developer phase implements features that will ease and optimize the development process but may not be useful to the end user during normal runtime. The main priority of this phase is the implementation of the rviz tool to give the team a better understanding of the robot's behavior while debugging. The implementation of rviz for motion planning should assist the developers working on the autonomous functionality of the robot while the implementation of rviz for environment visualization should assist the developers working on robot vision and mapping. The phase will also likely implement the IMU sensor display to assist the developer with debugging the autonomous self-right. Finally, the phase will also implement rqt_reconfigure to allow developers to quickly tweak the parameters to improve the functionality of the robot as efficiently as possible.

The advanced phase implements features that are more complex and/or are reliant on the implementation of the systems that the other developers are working on. This phase is implemented later so that the developer has a better understanding of how to develop and implement the GUI and give more time to the other developers to properly implement the systems the features are dependent on. This phase will implement the generated map which requires the implementation of the mapping system and the implementation of swarm AI to have the combined map. The phase will also implement the task list which the specifics of how it will be implemented will be dependent on how the autonomous functionality is developed. Finally, the phase will also implement the list of online robots that can be selected; it may rely on the implementation of the swarm AI but was also placed in this phase due to the potential complexity of switching control amongst robots.

The stretch goals phase implements all of the features that have been designated as stretch goals. The features that are designated to this phase may or may not be developed depending on the amount of time left to complete this project at the end of the advanced phase. Due to the necessity of the cleanup phase, it will have priority over the stretch goals phase. If it is determined that there is enough time to implement some or all of the features in the stretch goals phase, it will start by determining which features will be prioritized. The prioritization will be determined by a combination of usefulness to the end user and how likely they are to be completed in time.

The cleanup phase consists of simplifying the GUI to only implement the features that would be useful for the end user. Since, the features are constantly being added to the GUI in the previous phases, we anticipate that the GUI will be clustered with too much information. The issue with having too much information in the GUI is that it makes it much more complex for the user to operate system efficiently and makes the displayed information less readable. Therefore, we will be using the feedback from the previous phases to determine which features are the most important and which are non-necessities. By removing these non-necessities, we can place more emphasis on the important features such as allocating more screen space to the cameras or maps of the robot. Within this phase we will also be experiment with different ways of representing the information within our GUI to make it more user-friendly. An example of this would be the use of more plots and figures to represent information instead of writing out the information in the form of text.

Driver Station Test Plan

In order to provide the optimal user experience with the driver station, we will be conducting significant testing. It is critical that the implemented features are fully reliable during normal runtime and make the operation of the EZ-RASSOR as intuitive and efficient as possible. To accomplish this, the testing will be broken down into types of testing.

The first type will be the testing and debugging of the basic functionality. This testing will be conducted solely by the driver station developer before the execution phase is released to the other team members. Each of the implemented widgets will be tested within ROS and the Gazebo simulation to ensure that they are working as expected. Any discovered errors will need to be fixed before the execution phase is distributed to the other team members. This is done to ensure that using the GUI will optimize the development process instead of hindering it. Once each of the implemented widgets are established to be fully reliable, the execution phase can be distributed.

The second type of testing will be conducted after the advanced execution phase has been distributed. This testing will be conducted by the entire team and consist of them providing their feedback on their experience with using the GUI and the individual widgets. In order to standardize the feedback and to avoid slowing down the development of the other systems, the team members will be

filling out a created survey. Team members will be asked to place each of the implemented widgets into one of three categories to rate how useful they will be to the end user. The categories will be: necessary, useful but not necessary, and not useful. Widgets placed within the useful but not necessary category will be ranked by how useful they would be to the user. This feedback will be used when considering what widgets and features will be removed from the driver station during the cleanup execution phase. Additionally, the team members will be asked to provide feedback on the difficulty of interacting with the widget and the difficulty of understanding the information that the widget was attempting to provide. If the widgets are difficult to understand or interact with then they will be changed within the cleanup execution phase. Lastly, the team members will provide feedback on the visual representation of the GUI as a whole. This will be taking into consideration the size and position of the widgets and how easy it was to find desired information. This will feedback will be used to rearrange the GUI into a design that will make the operation of the driver station as efficient as possible for the end user during normal runtime.

Autonomous Control Execution Plan

The Autonomous control will consist of three major components that work together to avoid obstacles, generate a shared map, and efficiently work together to develop this map and mine regolith as a group. The first component of this system will be the object detection and avoidance, which will perform basic obstacle detection given the image data received by the cameras. The second component is the GPS-Denied Navigation which will involve a mapping of information gained on the ground to a top down representation that can hopefully be compared with orbital maps to anchor the relative positioning of the map to a global representation. The final component will be the swarm intelligence, which will function as a hive mind, controlling the objectives of individual EZ-RASSOR's in an attempt to collaboratively explore and mine the surrounding area without need for communication between individual EZ-RASSORs. The individual details of these implementations will be discussed below, however some aspects will be mentioned in this piece with respect to the information they provide to the path planning and avoidance protocols.

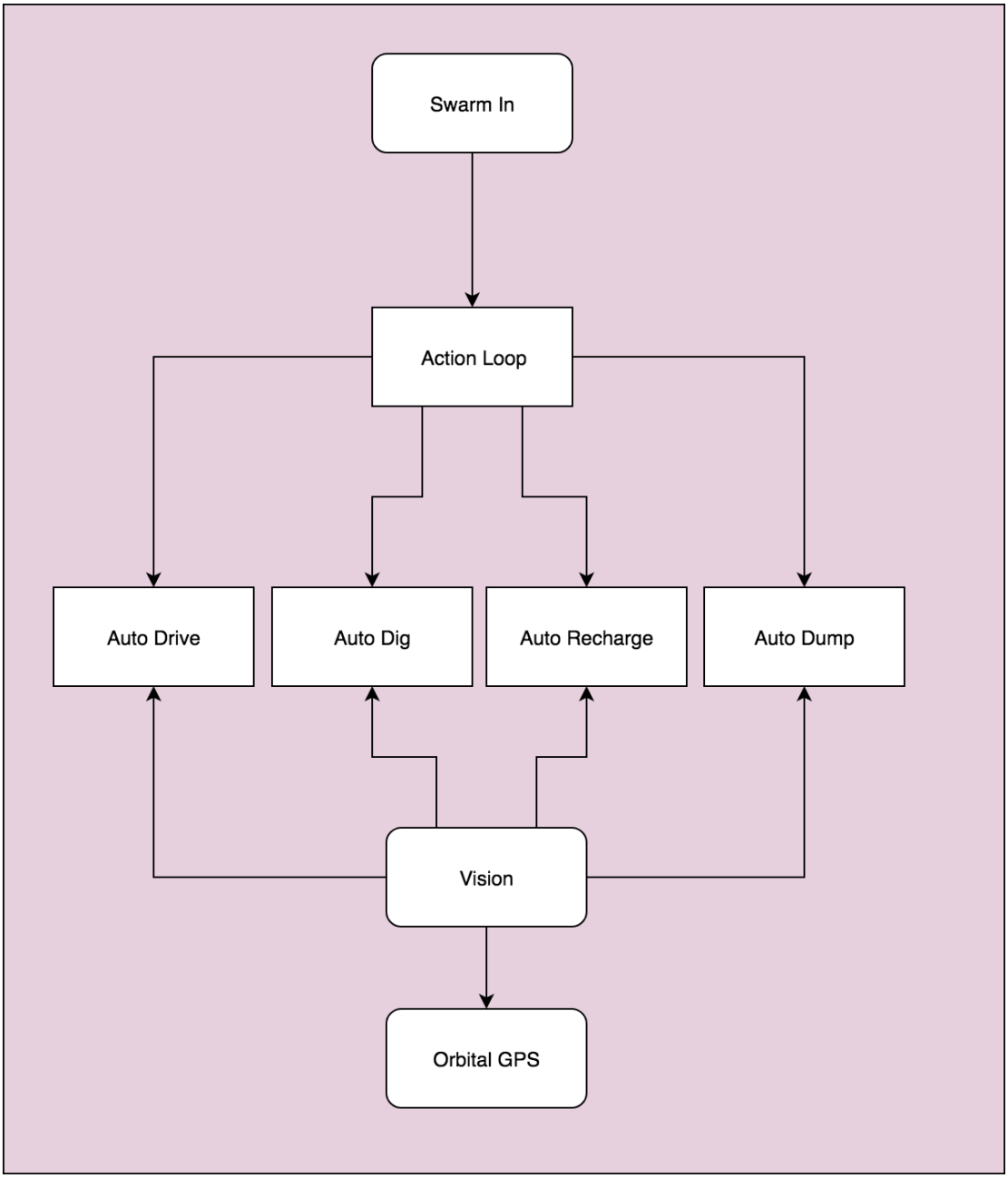


Figure 80

Autonomous Control Without Swarm

Despite the entire system being designed around autonomy being controlled by the central swarm, the EZ-RASSOR should also have the capability of operating as a single unit. The implementation of this system will function as an emulation of the swarm commands; still providing high level operations similar to the explanation below. Without swarm, the EZ-RASSOR will be commanded to

explore at a random location continually until it finds a suitable area for digging. Then the system will loop Auto_Dig repeatedly until the battery is low, then it will recharge and continue the loop again until the hopper is full or autonomous control is ended.

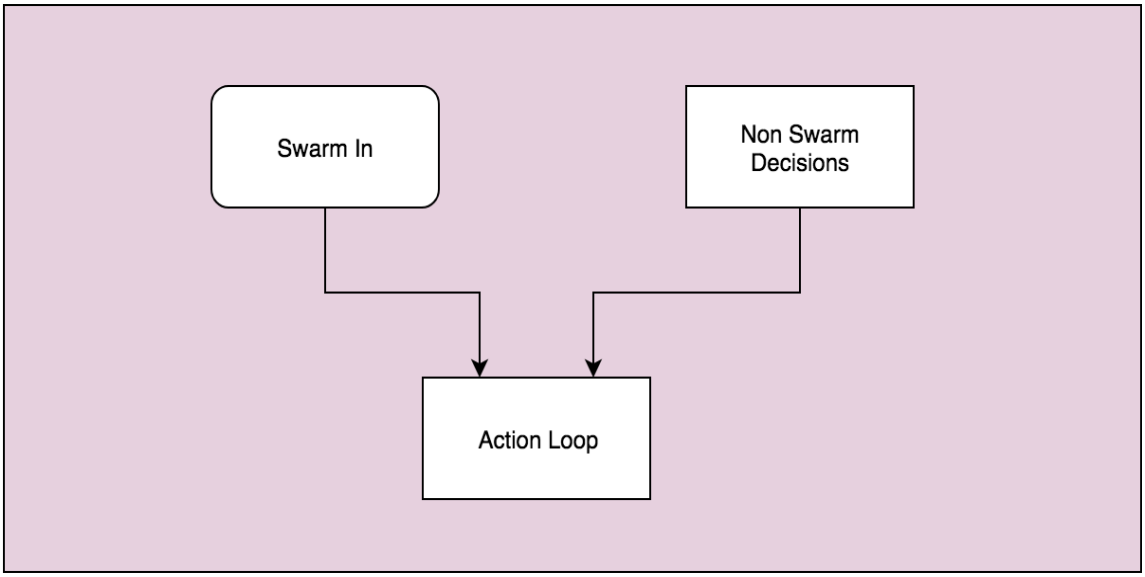


Figure 81

SLAM Mapping 3D to 2D

The result of the SLAM algorithm is a 3D point cloud representation of the environment as seen from the camera. This representation is useful for navigation and providing the agent with a mechanism to understand its position within the environment. However, for the goals of collaborative mapping, path planning and obstacle tracking we could utilize a 2D grid world representation of the world. The most effective way to do this is to calculate the first and second gradient of the point cloud and then to consider all of the places where the gradient is larger than a threshold to be non-traversable obstacles. Of course this method will be highly susceptible to noise, so before the data will first be filtered using bilateral filtering [30].

Auto Functions & Controller Mapping

Another large part of the autonomous control is its interface with the motor control software. In order to simplify this system, we decided to provide the same output values from the AI, as come from the control app and driver station. While this simplifies the software and ROS graph significantly, it will require a final mapping from objectives determined through AI, down to individual buttons

pressed on a controller. Given that the system will already contain a library of auto functions such as “auto drive”, “auto dig”, and “auto dock” the AI can offload a lot of of this individual decision making by simply calling these functions based on the current objective of the system. Based on the available commands coming from the swarm intelligence, the EZ-RASSOR will have three major objects: “explore”, “dig” and “recharge”. They will be made up of the following combination of auto routines.

“Explore” = Auto_Drive + Auto_Spiral_Search + Auto_Drive

“Dig” = Auto_Drive + Auto_Dig + Auto_Drive + Auto_Dump

“Recharge” = Auto_Drive + Auto_Dock

This system will function as a queue which will enqueue these auto functions in the order of their execution with the proper targets and parameters when a new high-level operation is decided. For example, if the system receives a new operation of “Dig”, it will enqueue Auto_Drive with the “Dig” target followed by enqueuing Auto_Dig and then Auto_Drive with a target of 0,0 (home base) and finally Auto_Dump which will dump the load into the base.

Beyond the high-level operations provided by the swarm, the system will also be capable of taking auto functions as objectives directly. This is because the Phone app and Driver station will have the option of semi-autonomous mode which will allow for performing an action like Auto_Drive and Auto_Dig with the press of a button. When this action occurs, the main action interpreter will shut off button inputs coming from the controller and pass the auto-function to the AI via the swarm-in topic. Using this topic instead of creating a separate one facilitates the process in a smooth manner and avoids the need for extra flags to determine which topic should be read and when.

A* Search Algorithm

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

```
function A_Star(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
   // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := an empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue             // Ignore the neighbor which is already evaluated.

            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)

            if neighbor not in openSet       // Discover a new node
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue;

            // This path is the best until now. Record it!
```

```
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] +
                        heuristic_cost_estimate(neighbor, goal)
```

*Figure 82 | Source: https://en.wikipedia.org/wiki/A*_search_algorithm*

go_around_obstacle()

Helper function to temporarily override pathing to avoid small obstacles in the EZ-RASSOR's direct path. The strategy for this may change but at the moment we will simply turn left travel a short distance turn back and check if the

```
def go_around_obstacle():
       align_heading(self.heading+90)

       while(self.current_location != self.current_location + delta):
              control_inputs.publish(forward)

       align_heading(self.heading-90)

       if(vision.obstacle()):
              go_around_obstacle()

       return
```

Figure 83

align_heading()

The align_heading() function will take the the new heading and continually change the direction until the new heading is reached. The angle system will be initialized based on the initial starting angle of the EZ-RASSOR upon the beginning of operation. Positive heading angle changes will be counterclockwise rotations and negative heading angle changes will be clockwise rotations.

```
Def align_heading(new_heading):
       while(self.heading != new_heading):
              if(self.heading < new_heading):
                     control_inputs.publish("left")
              Else:
                     control_inputs.publish("right")
       return
```

Figure 84

Auto_Drive()

The auto drive function will take a target location and the current grid map built by the robots. It will use the A* algorithm to generate a path using the grid world representation of the map generated by SLAM and will also avoid unexpected obstacles with basic hazard detection.

```
def Auto_Drive(target, grid_map):

current_grid_position = grid_map(self.current_location)
target_grid_position = grid_map(target)

path = A*(current_grid_position, target_grid_position)

while(!path.isEmpty()):
       center_next_grid = center_coords(path.peek())
       heading = vector(self.current_location, center_next_grid)
       align_heading(self.heading)
       while(self.current_location != center_next_grid):
              if(!vision.obstacle()):
                     control_inputs.publish("forward")
              else:
                     go_around_obstacle()
                  current_grid_position = grid_map(self.current_location)
                  target_grid_position = grid_map(path.peek())
                  path.push(A*(current_grid_position, target_grid_position))
                center_next_grid = center_coords(path.peek())
                  self.heading = vector(self.current_location, center_next_grid)
                  align_heading(self.heading)
     path.pop()
```

Figure 85

Auto_Dig()

The Auto_Dig() function will assume that the EZ-RASSOR is on a fairly flat surface and that the angle needed for the arms to touch down will be the same. The function will begin by lowering the arms to digging position (approximately 30 degrees below horizontal) and then begin by spinning the drums in opposite outward directions. There are two ways to end this operation, the one being utilized here will depend on the available sensors on the EZ-RASSOR. If the algorithm has access to the drum capacity sensor then the algorithm will simply continue to send the rotate signal to both drums until the capacity has been reached. Otherwise, an approximation for the amount of time to fill the drums will be determined through testing and the algorithm will run for this amount of time.

Obviously, this method is less efficient and may cause over or underfilled drums, but it will be our best choice without the proper sensor.

```
def Auto_Dig():
	move_arms(-30)
	while(self.drum1_capacity > 0 and self.drum2_capacity > 0):
		if(self.drum1_capacity > 0):
			control_inputs.publish("drum1 clockwise")
		if(self.drum2_capacity > 0):
			control_inputs.publish("drum2 counterclockwise")
		if(!vision.obstacle()):
		   control_inputs.publish("forward")
		else:
			go_around_obstacle()
	return
```

Figure 86

move_arms()

```
def move_arms(angle):

    while(self.arm1_angle >= angle and self.arm2_angle >= angle):
	    if(self.arm1_angle > angle):
		    control_inputs.publish("arm1 down")
	    if(self.arm2_angle > angle):
		    control_inputs.publish("arm2 down")
    return
```

Figure 87

Auto_Dock()

The Auto_Dock() function will perform the alignment of EZ-RASSOR with the charging port once the EZ-RASSOR has returned to the home base. The first step will be to identify the charging port on the base. Then it will navigate to a place directly in front of the charging port. Next, it will slowly rotate in place until aligned with the port and begin to go forward until connected.

```
def Auto_Dock():
	dock_approach_vector, dock_pos = search(“dock”)

	new_heading = arccos(norm(dot(dock_pos,-self.position))

	align_heading(new_heading)

	while(self.current_location != dock_pos-dock_approach_vector):
		control_inputs.publish(“forward”)

	align_heading(dock_approach_vector)
```

```
        while(!self.charging):
                control_inputs.publish("forward”)

        return
```

Figure 88

Auto_Dump()

Similar to the issue with Auto_Dig, the availability of the sensor to determine the drum capacity will greatly influence how this auto function operates. Just like with the Auto_Dig implementation this pseudo code will assume that the drums have a sensor to detect capacity. The function will need to align with the

```
def Auto_Dump():
        hopper_approach_vector, hopper_pos = search(“hopper”)

        new_heading = arccos(norm(dot(hopper_pos,-self.position))

        align_heading(new_heading)

        move_arms()

        while(self.current_location != hopper_pos-hopper_approach_vector):
                control_inputs.publish(“forward”)

        align_heading(dock_approach_vector)

        while(!self.docked):
                control_inputs.publish("forward”)

        while(self.drum1_capacity > 0):
                control_inputs.publish(“drum1 counterclockwise”)

        while(self.current_location != hopper_pos-hopper_approach_vector):
                control_inputs.publish(“reverse”)

        align_heading(-hopper_approach_vector)

        while(!self.docked):
                control_inputs.publish(“reverse”)

        while(self.drum2_capacity > 0):
                control_inputs.publish(“drum1 clockwise”)

        return
```

Figure 89

The implementation of these auto functions will be done in phases to ensure that critical functionality like Auto_Dig and Auto_Drive is completed before significant work is done on less critical functions such as Auto_Dock and Auto_Dump. The system to determine action choices without the swarm input will also be developed early on in the cycle to avoid waiting on tests before integration with the swarm system has been completed.

Autonomous Control Test Plan

The test plan for the autonomous control will be broken down into tests in Gazebo and tests in the real world using the RC-Car developed with the project. If the EZ-RASSOR hardware is completed there will also be significant real world testing using this actual hardware. The testing will consist of the following:

RC-Car Real World Acceptance Tests

1. Given a already created map of the environment using a camera will the following be successful:
    a. Can the RC-Car successfully create a path to a location using A*
    b. Can the RC-Car Navigate from grid square to grid square successfully using emulated vision inputs.
    c. Can the RC-Car perform the Auto_Sprial search function

Gazebo Acceptance Tests

In an environment with several large and small obstacles can the model inside the gazebo simulation perform the following:

1. Navigate to a particular location given a drive command and an x,y coordinate.
2. Fill its drums to capacity while moving using given a dig command
3. Navigare to a particular location and initiate the spiral exploration sequence.
4. Accurately map the 3D point cloud data to a 2D grid world with obstacle edges around each of the large objects in the scene.

5. Maintain navigational heading when encountering a small obstacle and avoiding it using the go_around_obstacle method.
6. Navigate to the homebase
7. Identify the charging dock with visual data only
8. Identify the regolith hopper with visual data only
9. Navigate, align, and dock with the charging dock
10. Navigate, align, and dock with the hopper

Autonomous Control Function/Method Tests

1. Provide Auto_Dig with proper data, check that actions are published to the input_controller topic.
2. Provide Auto_Drive with proper data, check that actions are published to the input_controller topic.
3. Provide Auto_Dump with proper data, check that actions are published to the input_controller topic.
4. Provide Auto_Dock with proper data, check that actions are published to the input_controller topic.
5. Provide A* with a 2d occupancy grid and check that a stack of actions is returned.
6. Provide go_around_obstacle with emulated vision data, check that the routine completes, maintaining the same heading and adjusting to new position.
7. Provide move_arms with a new angle and check that both arms are adjusted to the new angle relative to the old one.
8. Provide align_heading with a new heading and check that the EZ-RASSOR model adjust its heading to the new heading.
9. Start autonomous operation and measure time between controller input topic updates. Publishing rate should be similar to that of controller/driver station update interval.

Robot Vision - LSD-SLAM Execution Plan

LSD-SLAM Implementation

The latest version of LSD-SLAM currently has only been tested up to version 14.04 of Ubuntu and ROS indigo. So to guarantee a solid performance, use these versions of the software while setting up the algorithm, otherwise some changes will have to be made. This will also require the ability to clone from github and needs a current version of openCV downloaded. This guide is based off of the installation guide found here [30].

Step 1 : Make sure that you already have a rosbuild workspace setup. If you don’t, the following commands will start one.

```
sudo apt-get install python-rosinstall
mkdir ~/rosbuild_ws
cd ~/rosbuild_ws
rosws init . /opt/ros/indigo
mkdir package_dir
rosws set ~/rosbuild_ws/package_dir -t .
echo "source ~/rosbuild_ws/setup.bash" >> ~/.bashrc
bash
cd package_dir
```

Figure 90 | [30]

Step 2: Install System Dependencies

```
sudo apt-get install ros-indigo-libg2o ros-indigo-cv-bridge
liblapack-dev libblas-dev freeglut3-dev libqglviewer-dev
libsuitesparse-dev libx11-dev
```

Figure 91 | [30]

Step 3: Clone the LSD-SLAM repository into the ROS package directory (Current Directory if step one is followed)

```
git clone https://github.com/tum-vision/lsd_slam.git lsd_slam
```

Figure 92 | [30]

Step 4: Compile the program

```
rosmake lsd_slam
```

Figure 93 | [30]

LSD-SLAM can be run live from a camera using the following commands:

```
rosrun lsd_slam_core live_slam /image:=<yourstreamtopic>
/camera_info:=<yourcamera_infotopic>
```

Figure 94 | [30]

Or if you want to use a calibration file (which can be found in the `lsd_slam_core/calib` directory) :

```
rosrun lsd_slam_core live_slam /image:=<yourstreamtopic>
_calib:=<calibration_file>
```

Figure 95 | [30]

Keep in mind that the text in <> will have to be replaced with what it is referencing.

Once you have SLAM installed, you will have to modify it slightly to work with four cameras instead of two. As it stands right now, the temporal stereo approach uses one camera but cycles through both on the double stereo camera mounted on the front. This approach is being used to fill in the cracks of what the static approach is missing. The robot will have two cameras mounted on the front though, and two on the back. So if we modify the code to cycle through all four cameras, the resulting map will be stitched together a lot more cleanly, as it will be catching even more of what the static approach is missing initially.

To do this, you will want to go to the following directory and then find the following code located in the file:

```
lsd_slam/lsd_slam_core/src/LiveSLAMWrapper.cpp
```

Figure 96 | [30]

```
LiveSLAMWrapper::LiveSLAMWrapper(InputImageStream* imageStream,
Output3DWrapper* outputWrapper)
{
	this->imageStream = imageStream;
	this->outputWrapper = outputWrapper;
	imageStream->getBuffer()->setReceiver(this);

	fx = imageStream->fx();
	fy = imageStream->fy();
	cx = imageStream->cx();
	cy = imageStream->cy();
	width = imageStream->width();
	height = imageStream->height();

	outFileName = packagePath+"estimated_poses.txt";


	isInitialized = false;


	Sophus::Matrix3f K_sophus;
	K_sophus << fx, 0.0, cx, 0.0, fy, cy, 0.0, 0.0, 1.0;

	outFile = nullptr;


	// make Odometry
	monoOdometry = new SlamSystem(width, height, K_sophus, doSlam);

	monoOdometry->setVisualization(outputWrapper);

	imageSeqNumber = 0;
}


LiveSLAMWrapper::~LiveSLAMWrapper()
{
	if(monoOdometry != 0)
		delete monoOdometry;
	if(outFile != 0)
	{
		outFile->flush();
		outFile->close();
		delete outFile;
	}
```

```
}
```

Figure 97 | [30]

One imageStream handles the data from both of the cameras on a stereo camera. So this code should be modified to have two image streams, one for the front cameras, and one for the back cameras. You will then want to duplicate the image stream startup code. You will also have to update the function name in header file of the same name. You will also need to update the code in the command line parameters sections to now accept two camera locations and two calibration files.

The installation guide for LSD-SLAM can also be found on the official GitHub [30] page.

Robot Vision - LSD-SLAM Test Plan

The goal of the test is to make sure that the live version of LSD-SLAM will be functioning properly to allow for proper mapping and AI testing to take place later. So to test the successful download and installation of the LSD-SLAM software, we will be comparing the output of a live camera, and the output of a prerecorded model provided on the official Github page [30]. This test plan is adapted from [31].

To begin with, let's run the example model. The first step is to install the example(http://vmcremers8.informatik.tu-muenchen.de/lsd/LSD_room.bag.zip) sequence and extract it to the location of your choosing. Once that is done, you can launch the viewer with:

```
rosrun lsd_slam_viewer viewer
```

Figure 98 | [31]

Then you will launch the lsd_slam main ROS node with:

```
rosrun lsd_slam_core live_slam image:=/image_raw
camera_info:=/camera_info
```

Figure 99 | [31]

Then you can play the example using this command:

```
rosbag play -r 25 ~/LSD_room.bag
```

Figure 100 | [31]

Where ~/ represents the path to the directory you have stored the extracted example in. Also, “-r 25” is added to the command to speed up the example simulation. If you want to view the example at a different speed just adjust this number.

The output should be two windows. One showing the current keyframe with color-coded depth(as shown in the LSD-SLAM research section), and another window showing the map. It that is not the case then hit the r key on the keyboard once. If it is still not the showing the two windows, then you should refer to the github account to verify that everything is installed properly [30]. If the two windows show up, then the next step is to run it on the actual device itself. Cameras have distortion though. Because of this we will have to make a calibration folder that we can feed to LSD-SLAM, that will take our robots camera into account using the information found in this link [31]. This step should be completed even if the camera being used is a simulated camera in Gazebo, just to ensure that there is no distortion happening in that view either. Once that is completed, you can run the following set of commands:

```
rosrun lsd_slam_core live_slam /image:=<yourstreamtopic>
_calib:=<newly_made_calibration_file>
```

Figure 101 | [31]

```
rosrun lsd_slam_viewer viewer
```

Figure 102 | [31]

If the result is similar to the example, then the installation was a success and it is now usable. If not, then there was either an error with the installation, or some of the software has become outdated and will need some manual adjustments to become compatible with other softwares necessary for the project. The source code is four years old for LSD-SLAM, so this may be necessary. This link claims to have gotten the code working with version 16.04 of Ubuntu with ROS with some minor adjustments to the code and will likely be a good place to start troubleshooting if you run into problems in this step [31].

Robot Vision - Object Detection Execution Plan

Vision Implementation

This implementation is based off of the resources found here [32]. One of the main concerns we have with the robot is processing power. Because of this, we intend to combine the SLAM mapping algorithm and the object detection stages as much as possible. The SLAM algorithm will return a point cloud that relates to a depth map. If we perform the computer vision algorithm on the point cloud data, then we will be able to map the hazards to the depth map, and then the local map, very easily.

Implementing the YOLO algorithm has several complications. The first being that LSD-SLAM algorithm does not auto-generate the point cloud. You must press the letter “p” manually to generate each one that you want. So the first step would be to modify the LSD-SLAM code to auto-generate the point clouds at a rate of our choosing. Once that is done, you can begin downloading the YOLO base system. OpenCV is an optional dependency for this algorithm. Begin by cloning the repository and making it:

```
git clone https://github.com/pjreddie/darknet.git
cd darknet
Make
```

Figure 103 | *[32]*

You will have to wait for an output, once that happens you can enter the following command:

```
./darknet
```

Figure 104 | *[32]*

If the result is:

```
usage: ./darknet <function>
```

Figure 105 | *[32]*

That means that the installation was successful. The next step is to run a pretrained model to verify that the YOLO algorithm can run on this system. Run the following commands:

```
wget https://pjreddie.com/media/files/yolov3.weights
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

Figure 106 | [32]

If the output looks like this then you are able to move on to training your own model:

```
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32  0.299 BFLOPs
    1 conv     64  3 x 3 / 2   416 x 416 x  32   ->   208 x 208 x  64  1.595 BFLOPs
    .......
  105 conv    255  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 255  0.353 BFLOPs
  106 detection
truth_thresh: Using default '1.000000'
Loading weights from yolov3.weights...Done!
data/dog.jpg: Predicted in 0.029329 seconds.
dog: 99%
truck: 93%
bicycle: 99%
```

Figure 107 | [32]

To train the YOLO algorithm, you will need a large collection of point clouds containing examples of flat surfaces that are suitable for mining(mine), the charging station(charge), the regolith drop-off site(dump), and a wide array of potential hazards(hazard). The mining label should be of relatively flat surfaces that could be mined by both arms at the same time, and be relatively free of large objects. The hazards label should include anything that could potentially block the robot(including other robots), pitfalls, or rough terrain that could cause it to slip(such as a steep slope).This should be the most time consuming step in the vision category, as we must run the LSD-SLAM algorithm, take the point clouds generated of the objects mentioned above, and then label each of the point clouds (using the label in parenthesis). You label the data by making a .txt file with the same name as each of images you have, and having the following be the contents. Filling in the values depending on the label and the width and height of the image.

```
<object-class> <x> <y> <width> <height>
```

Figure 108 | [32]

Once you have labeled all of your data, you will need to make a file called train.txt that contains the names of all of the images you want to train. You should also make a test.txt file that can be used for the algorithm to test how successful it is while it is training itself and a labels.names that contains a list of all of the labels that are being used. Once these are done, you will need to modify the config file to look for these in the darknet directory. The directory is called cfg, make a file called “EZ.data”. You will then want to add this to that file.

```
1 classes= 4
2 train  = <path-to>/train.txt
3 valid  = <path-to>/test.txt
4 names = data/labels.names
5 backup = backup
```

Figure 109 | [32]

We now need to download the pre-trained convolutional weights:

```
wget https://pjreddie.com/media/files/darknet53.conv.74
```

Figure 110 | [32]

Once we have those we can now train the model:

```
./darknet detector train cfg/EZ.data cfg/yolov3.cfg darknet53.conv.74
```

Figure 111 | [32]

Once that finishes running, you should now have a model capable of analyzing the point clouds generated.

The YOLO algorithm will most likely be run on a normal CPU and not on a GPU, which means it won’t be able to run in real time. It should take approximately six to ten seconds for the algorithm to analyze each image. The best approach would be to test the algorithm on the actual robot starting with the upper limit of ten seconds. This will be accomplished by having the LSD-SLAM algorithm generate a new point cloud every ten seconds. If the YOLO algorithm is able to analyze the image without a delay, then you can decrease the delay by a second. Keep doing this until you find that the algorithm is unable to keep up with the rate that the point clouds are being generated. Once that happens, then begin using the previous amount of time as the FPS for how often you are running the YOLO algorithm. This will allow us to be running the vision algorithm as often as

possible. The next step is to test it and verify it can tell what everything is correctly.

Robot Vision - Object Detection Test Plan

The first thing you will have to do for testing is generate another point cloud for each of the labels, that were not used in the training session. Once that is done, run the following command for each test image, replacing the word mine.ply with the name of the specific image.

```
./darknet detect cfg/yolov3.cfg yolov3.weights data/mine.ply
```

Figure 112 | [32]

You should end up with something similar to this, for each of the labels:

```
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32  0.299 BFLOPs
    1 conv     64  3 x 3 / 2   416 x 416 x  32   ->   208 x 208 x  64  1.595 BFLOPs
    .......
  105 conv    255  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 255  0.353 BFLOPs
  106 detection
truth_thresh: Using default '1.000000'
Loading weights from yolov3.weights...Done!
data/mine.ply: Predicted in 0.029329 seconds.
mine: 95%
```

Figure 113 | [32]

This means that when the algorithm is run with LSD-SLAM, we will be able to update the locations on the map that correspond to the resulting labels we just trained the system to look for. Once the map is updated, the AI and driver will be able to use that information to decide where to dig, the safest and most efficient route to get to that location, and it will be able to dock the charging stations and dump the regolith easier by being able to distinguish those from other objects in its way.

It is also worth noting that we have not trained our model to look for other robots. If we distinguished the robots from other hazards, we would then need to implement an optical flow algorithm to track its movements. The robots would then need to recalculate their path in real time. This might not be a big deal with a small number of robots, but if other robots were constantly in view, then our

rover would constantly be running an additional computer vision algorithm to track their movements and update its own heading. This would eat up a significant amount of processing power. Our approach instead, is to treat the robot as a general hazard. When the robot sees a hazard blocking its way, its response will be to stop and slightly alter its course to attempt to get around it. This may seem like a naive approach, but the swarm AI will be plotting all of the robots paths, and will specifically be plotting to avoid this situation from happening, so the few times needed to stop for another robot will be worth the extra processing power.

Stretch Goal Requirements Execution Plans

These execution plans reflect the same requirements as those listed under the “Phase 2 Requirements” section in our Statement of Work document. These plans were crafted after the development team completed their research on how one might tackle the stretch goals. Due to the nature of these requirements, the execution plan may be abstract in its approach compared to the plans outline in the Main Requirements Execution plan.

Control App Execution Plan

The controller application for the project will be fairly straight forward. Essentially all that is needed are the buttons to perform the various tasks listed in the “Pi Backend” section. This includes Stop (when a button is released), Forward, Backward, Turn Left, Turn Right, Arm 1 – Move Up, Arm 2 – Move Up, Both Arms – Up , Arm 1 – Move Down, Arm 2 – Move Down, Both Arms – Down, Drum 1 – Excavate, Drum 2 – Excavate, Both Drums – Excavate, Drum 1 – Dump, Drum 2 – Dump, Both Drums – Dump, AUTO_DIG, AUTO_DUMP, SELF_RIGHT, Z_CONFIG, and AUTO_DRIVE. The autonomous functions will be on a seperate page then the tropical user-driven controls. If time allows, we also want to implement a live feed of the on-board camera, as well as sensor data which will display the EZ-RASSOR vital information.

Control App Test Plan

To test our remote control application, we will be utilizing Jest. Jest if a testing framework built into React Native that make stesting streamlined and easy. We plan on testing our initial app design first by writing tests to ensure the app renders correctly as well as all buttons used. Furthermore, we will need to test

that all of the buttons register to touch input properly and that each button is linked to an event that triggers it's associated function. Once we know that all the buttons render correctly and are event-invoking. We will then test our asynchronous function. THis can be achieved by using Jest and REST client services (like POSTMAN) together. As a button is pressed on the app, we will send our request to a mock-server within REST services and Jest will verify that the API request was made correctly. Using the REST service we are also able to validate the data sent for formatting and accuracy. This process also works in reverse. We will be able to use REST services to give a mock-reply to our request. Jest will then validate that the app was able to receive the request successfully. At this point we would use console logging to validate the reply. Once all of this criteria is met, wll that would need to be done is changing our API url to the RC car's server and perform the same tests without the REST client.

Gazebo for AI Execution Plan

In order to properly implement the various autonomy and collaboration capabilities such as SLAM, GPS Denied Navigation (GDN) and SWARM, we must incorporate several plugins to Gazebo, as well as designing new environments to test and train these new technologies. The overall design of these environments will be to maximize the demonstration of their particular use cases and to minimize potential failure cases coming from other areas of the software. The final product will be tested in a comprehensive environments, but the early stages of development will be best supported by a simplified Gazebo environment.

Starting with the GPS Denied Navigation, the first and most simple environment will involve several plugins and a few simple geometric objects. The first step will be to create a ROS node that will function as stereo camera input coming from the Gazebo environment. This input will then be output to a separate panel, as well as, being used in the surface images processing in the GDN algorithm. The next step will be to attempt an approximation of a satellite image by placing a camera high above the ground within the Gazebo environment that can capture the various features of the landscape from a top down view. Images captured from this camera will also be stored, displayed in a separate panel and utilized in the GDN algorithm. Once this has been established the next step is to create several simplistic geometrical objects that can be used for testing the mapping of objects from surface level to top down view.

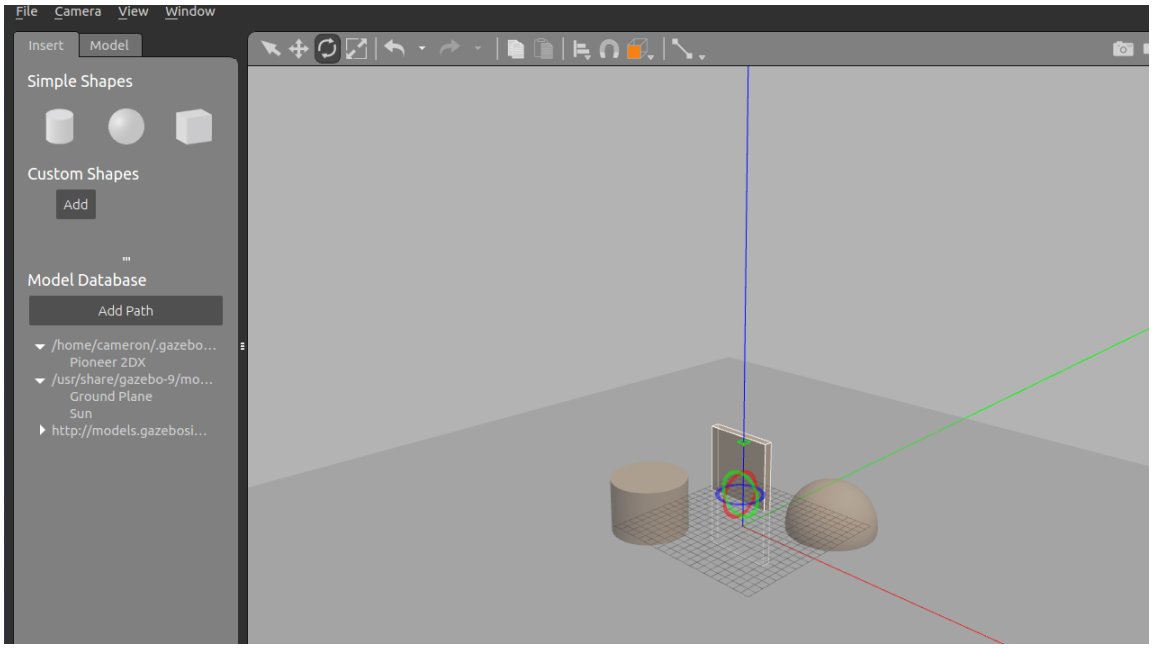


Figure 114

Gazebo for AI Test Plan

The majority of the Gazebo for AI test plan will be acceptance tests. The tests will check the properties of the objects in the scene, as well as, ensure that the various plugins such as the mono and stereo cameras are functioning properly. The acceptance tests will be as follows:

1. Activate mono camera and check that video is being displayed in separate GUI window dedicated for the mono camera feed.
2. Activate mono camera and save several frames of the image data from the Vision module of ROS to ensure the data is being properly piped to the camera data topic.
3. Activate the stereo camera and check that video is being displayed in separate GUI window dedicated for the stereo camera feed.
4. Activate the stereo camera and save several frames of the image data from the Vision module of ROS to ensure the data is being properly piped to the camera data topic.

5. Manuver EZ-RASSOR model around environment and ensure that environment objects are properly appearing in point cloud generated by SLAM.

6. Activate Stereo camera and check that depth map is being displayed in separate GUI window dedicated for the depth map.

7. Activate mono camera in orbital positioning and check that resolution and size of objects in resulting feed is similar to what could be obtained from a real satellite image.

GPS-Denied Navigation Execution Plan

The implementation of the GPS-Denied Navigation (GDN) is a collection of implementations of several techniques that analyze data from both orbiter and EZ-RASSOR imagery, and also an implementation of the consolidation of this data to reach a conclusion on location.

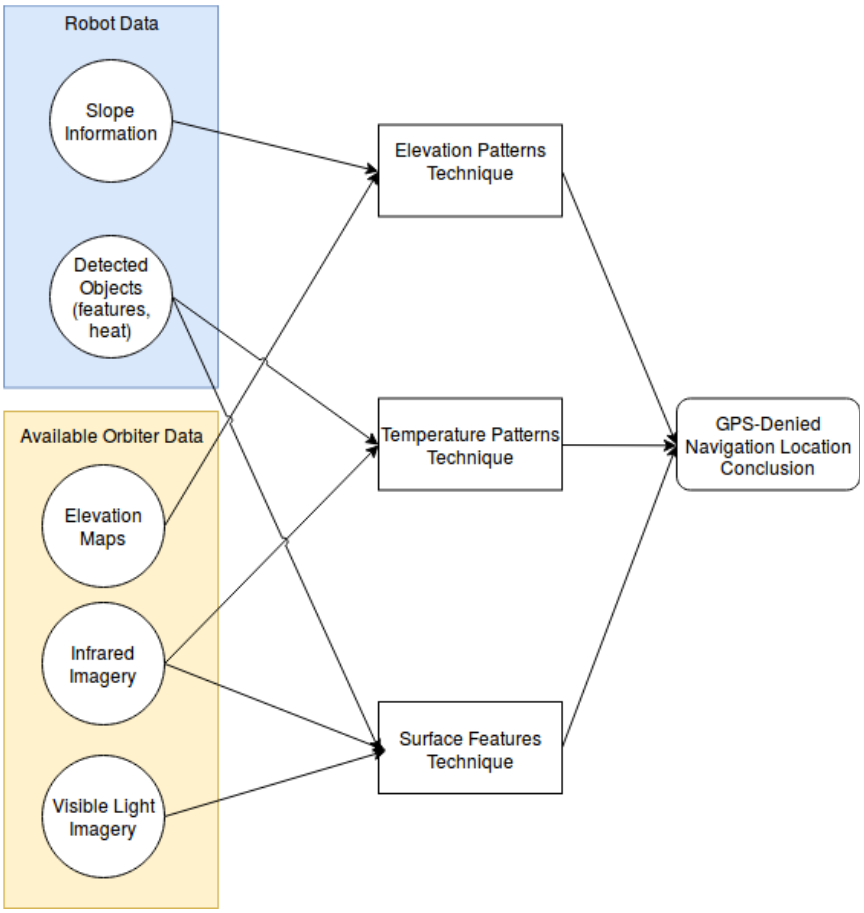


Figure 115 **|** *General architecture of the GPS-Denied Navigation system, showing data inputs from EZ-RASSOR and public orbiter datasets.*

Object Relationship Technique

Using visible-light images from orbiters of the target celestial body, we will find interesting features and landmarks that can be used to facilitate the determination of the EZ-RASSOR's location, by pairing the data provided from these orbiters with the data that is acquired on-board the EZ-RASSOR. This technique in particular will mimic the behavior the humans follow when trying to discern their location by matching what they see in person with what they are seeing on a map. When considering this behavior, the first type feature that we look for in order to find correspondence between first-person views and top-down views are patterns in structures. In the EZ-RASSOR system, the following approach will be taken to automate this aspect of pattern correspondence.

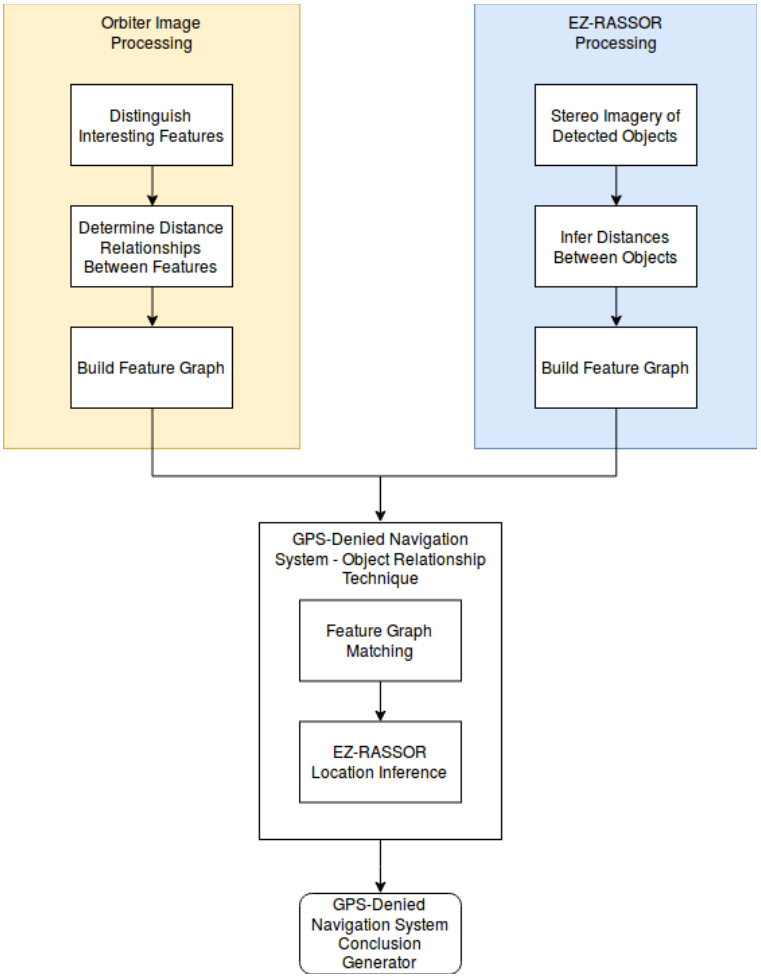


Figure 116 | The flow of information and procedures involved in the object relationship technique that uses visible-light orbiter data.

The integral part of the procedure of creating the location inference based off of the feature relationships using the visible-light data from orbiter images and EZ-RASSOR imagery is the construction of the two feature graphs that will be used to find a correspondence in features. The two feature graphs are created based off of two different perspectives - one from above and one from ground level.

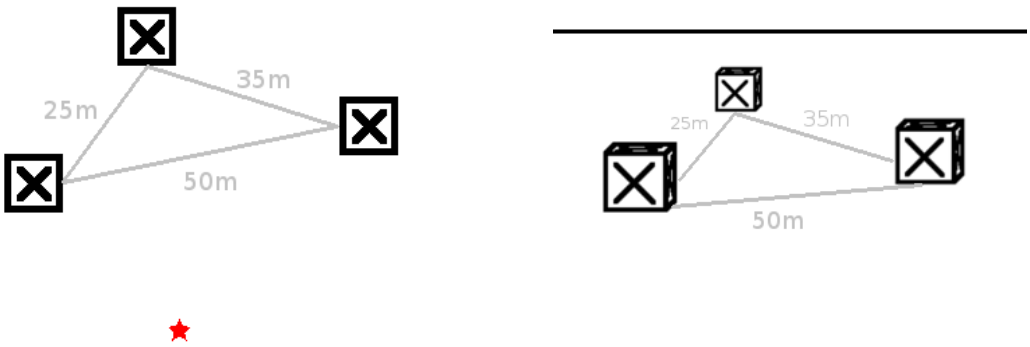


Figure 117 | Two representations, top-down and first-person, of feature graphs with corresponding features. The inferred location of the robot is marked with a star on the top-down view.

In order to support the location information that the EZ-RASSOR determines using the collaborative mapping based on object detection, the GDN will find a correspondence between the objects in the EZ-RASSOR's current locale and the objects observed from orbital imagery. As measurements from both entities are not exact, errors in observations must be taken into consideration, and the best guess be chosen. To perform this matching, the two feature graphs will be searched for close values and relationships between features. Due to the high probability that one feature graph will include objects that the other does not, we cannot rule out an object pairing due to the existence of an additional object in only one graph. This eases implementation, but clouds final confidence, as there are less factors that might rule out a potential pairing of observations of an object.

To procedurally determine this matching, a scoring system must be employed in order to score potential matches, compare scores, and reach the conclusion of highest confidence. The score for a match in object pairs will rely on two variables:

- Match in the distance between the two objects
- Number of matches already made involving one of the objects and another object

With this, it is realized that the algorithm will require multiple passes. The first pass will simply create naive matches based only on minimizing the difference in distances between object pairs. Subsequent passes will evaluate the number of associate matches and optimize its matches based off of those.

Elevation-Based Technique

An additional method to help determine the location of the system is to utilize what can be gathered from the robot's imagery in terms of elevation, namely that which is provided by the slope detection described in the *Slope Detection: Possible Algorithm for Hazard Detection* section, and from the map that is built with SLAM. What can be gathered from this information is the general trend in the change of slopes over the field of view, which can be used to determine changes in elevation. By taking into account the changes in elevation that surround the system, changes in elevations derived from elevation maps can be used to form an inference on the robot's location. The following figure shows an elevation map of the Moon, made available by NASA [33].

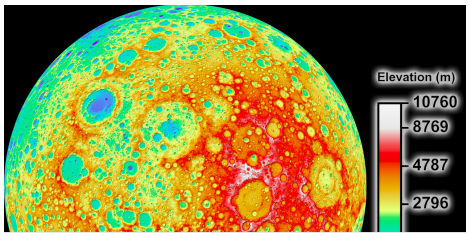


Figure 118 | Elevation map of the Moon, provided by the Lunar Reconnaissance Orbiter. Pixel resolution of 328 feet. [33]

Temperature-Based Technique

When considering the temperatures of surface features of celestial bodies, the first thing that comes to mind is that they vary with time each day, complicating the problem of creating associations between multiple temperature-based observations. In other words, if one entity observes the temperatures of a range of objects to be a certain set of values, another entity must make its observations of those objects at the same time of day and under similar weather conditions, etc. in order to achieve relatively similar measurements.

To overcome this, instead of attempting to match absolute values of the temperatures of surface features, the relationships between the temperatures will be used. The presumption is that if one entity makes an observation of surface temperatures under some set of conditions, and builds an understanding of the differences between those temperatures, and another entity makes the same an observation of the same features under different conditions, that entity's understanding of the differences between surface feature temperatures should relatively match that of the other entity's.

Following this train of thought, we realize that we are able to utilize temperature measurements that have been taken at any point in time, which allows for the use of the entire spectrum of past measurements to be utilized, providing an incredibly robust dataset.

Pink's Algorithm

To further provide effective location data using orbiter imagery, Pink's algorithm will be implemented. The process involves two parts: feature detection from orbiter imagery (executed beforehand), and feature matching in real-time from the rover's stereo imagery.

Feature Detection - Orbiter Imagery

For detecting features from orbiter imagery, the following procedure is to be followed:

1. Each pixel in the image is classified based on whether it belongs to a feature or not
2. Each pixel is added to a cluster depending on their Euclidean distances
3. Cluster centroids are calculated

For lunar purposes, a feature will be a crater or any elevation-based feature (small mounds or dips in surface) of any size (referred to generally as “craters” from now on), rather than a lane marking as used in the demonstration illustrated in the original paper. To classify whether a pixel is part of such a feature, it must be determined that it is within a crater, rather than a lane marking. This will be done by determining whether the elevation at that point, provided by the elevation map, is increasing or decreasing from a local maximum elevation, indicating that the pixel is indeed within a crater.

Feature Detection - Rover Imagery

The process for detecting features from the first-person view is different and must be done live, and thus must be done in a way that is not computationally intensive. The process for detecting features from the first-person view is as follows:

1. Run Canny Edge Detector
2. Cluster pixels based on image proximity
3. Calculate centroids for clusters
4. Associate each cluster with a weight based off of the amount of edge points that make up that cluster
5. Transform pixel-coordinates to real-world coordinates, with the camera being the origin

While Pink’s algorithm relies on a flat surface as no elevation data is available in his demonstration, we have the data necessary to allow for changes in elevation provided by elevation maps.

The clusters are then matched using the Iterative Closest Point algorithm. After the clusters are matched, the rover location is determined using the relative distances to the points determined by the rover and the absolute locations of the features determined by from orbiter imagery.

To create this matching to determine the rover’s location and direction, a pair of coordinate systems is used, as illustrated below. The blue coordinate system indicates that of the aerial view, and the red is the coordinate system relative to the rover, with the rover being the origin.

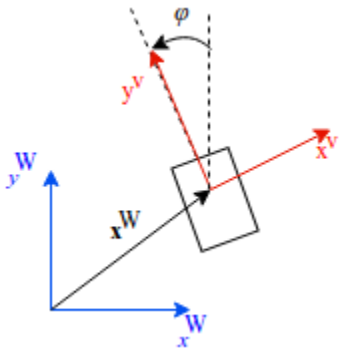


Figure 119 | Coordinate systems of the world (blue) and rover (red)

Pink denotes the set of n features in $\mathbb{R}^2$ from the aerial view as the set $\mathbf{S} = \mathbf{s}_1^W, \mathbf{s}_2^W, ..\mathbf{s}_n^W$, where W indicates the world coordinate system. The set of m features from the rover view is denoted as $\mathbf{T} = \mathbf{t}_1^V, \mathbf{t}_2^V, ..\mathbf{t}_m^V$, where V indicates the rover coordinate system. The goal is to determine $\mathbf{x}^W$, the exact rover position, and φ, the heading of the rover. Given the exact position and heading, the transformation from world coordinates to rover coordinates is simply given as follows:

$$\mathbf{s}_i^V = \mathbf{R}^{-1} \cdot (\mathbf{s}_i^W - \mathbf{x}^W) ,$$

where the matrix $\mathbf{R}^{-1}$ is the following inverse rotation matrix:

$$\mathbf{R}^{-1} = \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix}$$

Pink then goes on to use the initial estimates for position and heading, provided by GPS systems in his demonstration. Instead, as there is no GPS provided, our initial estimates will be based on the on-board location and heading data that is calculated from the collaborative mapping and SLAM processes. This initial estimated position is denoted as $\widehat{x}^W$, and the initial estimated heading as $\widehat{\varphi}$. A maximum error will be imposed for both values, and denoted with ~.

$$\begin{aligned} \widehat{\mathbf{x}}^W &= \mathbf{x}^W + \widetilde{\mathbf{x}}^W \\ \widehat{\varphi} &= \varphi + \widetilde{\varphi} \\ \widehat{\mathbf{R}} &= \widetilde{\mathbf{R}} \cdot \mathbf{R} . \end{aligned}$$

Finally, Pink derives an objective function that shall be minimized.

$$f(\widetilde{\mathbf{x}}, \widetilde{\varphi}) = \sum_{j=1}^{m} \gamma_j \cdot w_j \cdot \left\| \widetilde{\mathbf{R}}^{-1} \cdot \mathbf{t}_j^W - \widetilde{\mathbf{x}}^V - \widehat{\mathbf{m}}_j^V \right\|^2 .$$

The two inputs are the error in position and heading. The summation iterates for every interest point. γ represents the weighting for each point, $\mathbf{t}^{\mathbf{W}}$ the world coordinates of the feature from the aerial view, $\mathbf{x}^{\mathrm{V}}$ the error in position, and $\widehat{m}^{V}$ the matching first-person point.

The final estimations for the two values are iteratively utilized to gradually improve the accuracy of the values in the rover location. The values during each iteration are defined by the previous iteration as follows:

$$\begin{aligned} \widehat{\mathbf{x}}_{k+1}^{W} &= \widehat{\mathbf{x}}_{k}^{W} - \widetilde{\mathbf{x}}_{k}^{W} = \widetilde{\mathbf{x}}_{k}^{V} \cdot \widehat{\mathbf{R}}_{k} \\ \widehat{\varphi}_{k+1} &= \widehat{\varphi}_{k} - \widetilde{\varphi}_{k} \,. \end{aligned}$$

This is repeated until convergence in rover position. [12]

GPS-Denied Navigation Test Plan

This module really consists of two larger parts that are used to form the conclusion: the processing involved to build the feature graphs from imagery, and the process involved in matching the feature graphs. These two parts need to be individually tested before the entire module is tested.

For the testing of the matching of feature graphs, a large set of randomly-generated pairs of feature graphs with somewhat realistic values will be created, each already solved (as in, the matchings are already known as they are generated). Then, the algorithm will be ran on each pair of graphs, and output the determined matching pairs. Comparing this output to the real, predetermined matches, we can determine the validity of the algorithm by maximizing the number of correctly-matched pairs.

Swarm Robotics Execution Plan

Implementing and perfecting a swarm behavior in these robots will require more time than this group will be afforded. Therefore we must lay a solid foundation and provide insight into methods which have been tested, and document our successes and failures for future contributors.

Swarm Architecture

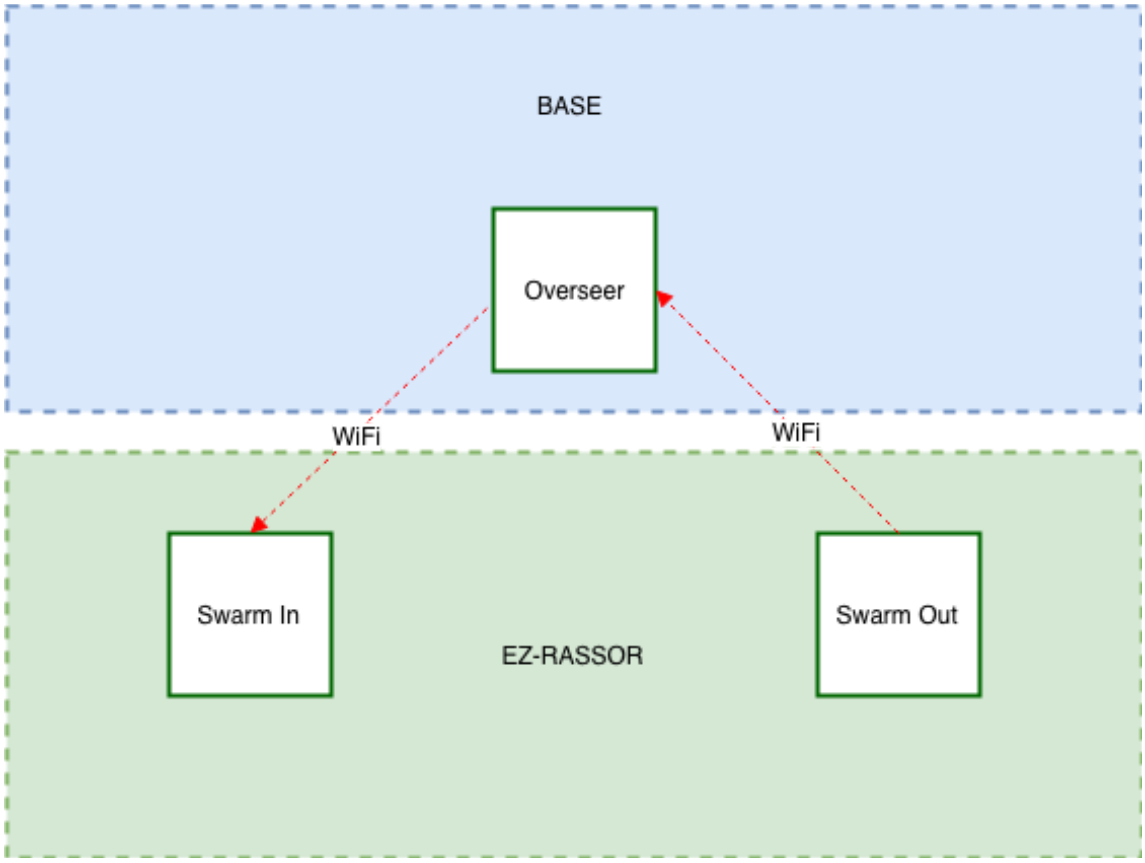


Figure 120 | Illustration of how the EZ-RASSOR will receive data from the base.

Our swarm implementations will consist of a hive mind, overseer approach. Taking advantage of the bases assumed onboard computing power and wifi capabilities, the robots will receive commands from the base. The Swarm-In topic will subscribe to the Overseer topic who will send commands over wifi along with target coordinate data. Swarm-Out will transmit current robot location data and current status data.

Swarm Data Received (Swarm-In)

Operations

- *EXPLORE* - Travel to target destination to initiate search routine.
- *MAP* - Execute SLAM.
- *DIG* - Travel to target destination to initiate digging protocols.
- *RESCUE* - Target is in need of assistance. Assist if possible.
- *RECHARGE* - Travel to target destination. Initiate recharge docking protocols.

Data

- *TARGET* - Coordinate data received as target destination.

Swarm Data Transmitted (Swarm-Out)

Status

- *TRAVELING* - Currently in motion toward target destination.
- *MAPPING* - Currently executing SLAM.
- *DIGGING* - Currently executing digging protocols.
- *EXPLORING* - Currently executing search routines.
- *STUCK* - Incapacitated. In need of assistance.
- *FULL* - Digging has stopped. Sensors read drums are full.

Data

- *POSITION* - Current location of the robot in 3D space.

Swarm Area Mapping

The first mission of the swarm will be to map their environment. We will be using a Simultaneous Localization and Mapping (SLAM) algorithm to compile environment data which each EZ-RASSOR will individually generate. For more information on this algorithm, the authors will direct the reader to the SLAM research and execution sections of this document. To intelligently map an area as a multi-agent system, a decentralized active information acquisition technique would be preferred. This would allow for the agents to make decisions based on worlds states and plan their movements to navigate an area and map the area autonomously. However, the perfection of such a technique may be outside of the scope of this phase of the robots development. The following approach will be a more straightforward iterative solution and will assume a team of 4 EZ-RASSOR robots to map a given area. These robots will orient themselves along the X and Y axis corresponding to the base position in 3D space. The

robots will then travel a straight line away from the base to a predetermined target distance. In this example we assign a distance of 200 meters.

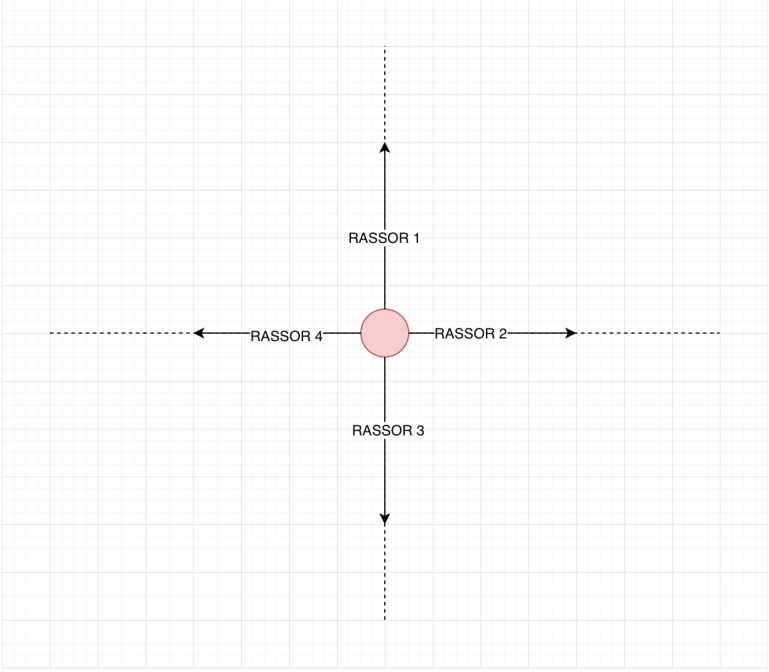


Figure 121 | All four EZ-RASSOR's will travel away from the base avoiding obstacles as they do so to initialize the mapping frame.

The robots travel along their respective axis while running the SLAM algorithm. Once the target distance is achieved the robot will then execute a 180º maneuver to return back to the base. On their return approach map corrections, via a loop closure, will be made and updated in memory. The robots will then rotate 90º and drive *v* distance along the adjacent axis, where *v* is the width of the EZ-RASSOR's field of view. It will then follow the following pattern.

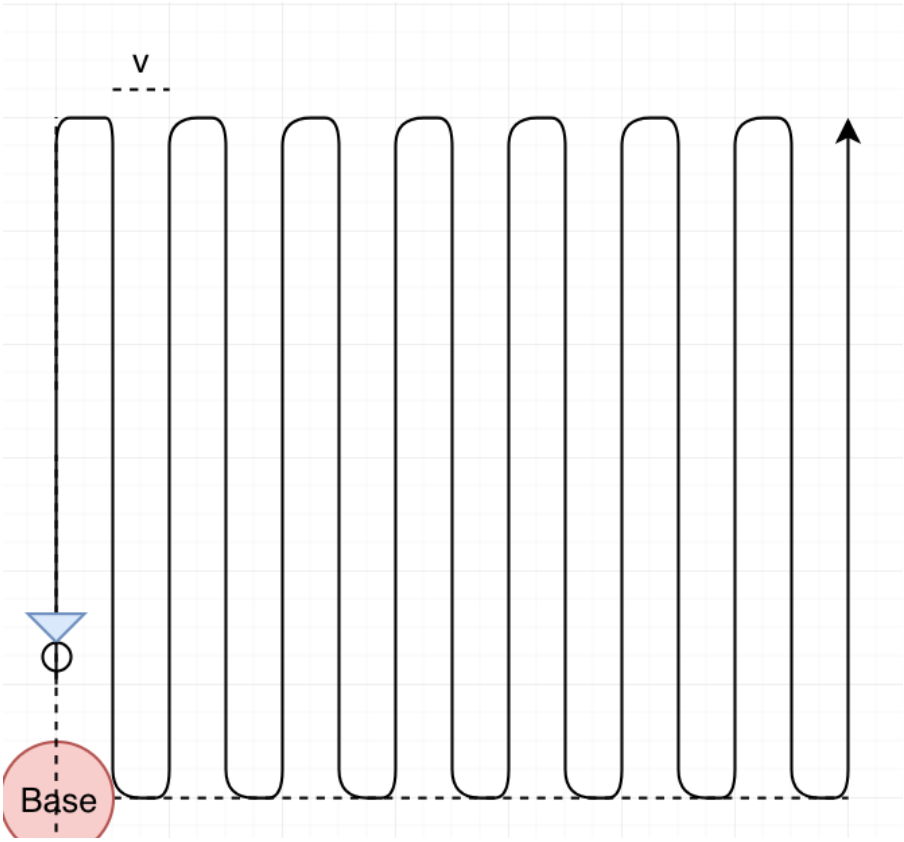


Figure 122 | A visual representation of a robots path during phase one of the SLAM algorithm in quadrant one. This behaviour would be expressed by all four EZ-RASSOR's in their respective quadrants.

Once the mapping is completed the robot will then execute a similar mapping pattern perpendicular to the first within the same quadrant. This will ensure any objects mapped will have been viewed from four different angles allowing the map to be more robust. Map information will be communicated to the base and all robots' map information will be integrated into the world map of the surrounding environment.

Swarm Resource Gathering

We will initially be basing our approach towards a swarm behavior as it pertains to resource gathering, on the research done by Melanie Moses and Joshua Hecker at the University of New Mexico [34]. There, they have been studying ant behaviors as it pertains to resource gathering. Specifically Desert Seed Harvester Ants. After studying these organisms at length they were able to create an algorithm which could express the ants foraging behavior. Figure 123, below shows the algorithm proposed by Moses and Hecker.

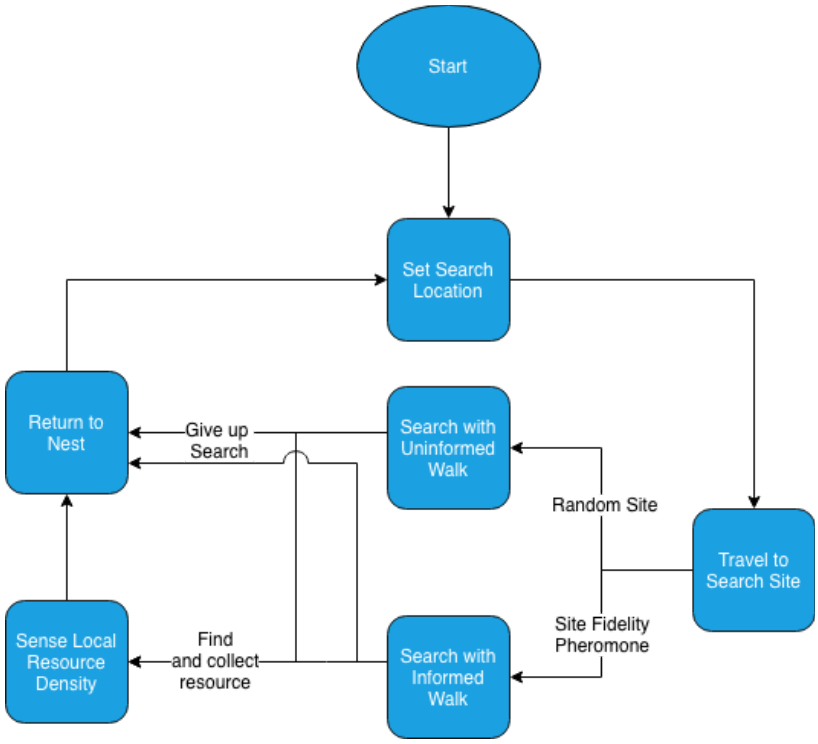


Figure 123 | The algorithm proposed by Dr. Moses and Hecker.

Using this approach, we can begin development on our first phase of this swarming behavior which will consist of the following:

- Setting desired search location.
- Traveling to that location.
- Executing a search routine to find optimal dig sites and, if no site is found, setting a new search location.
- Executing an automated dig routine once a site is discovered.
- Returning back to the base to offload acquired resources and communicate the dig sites location with the base.

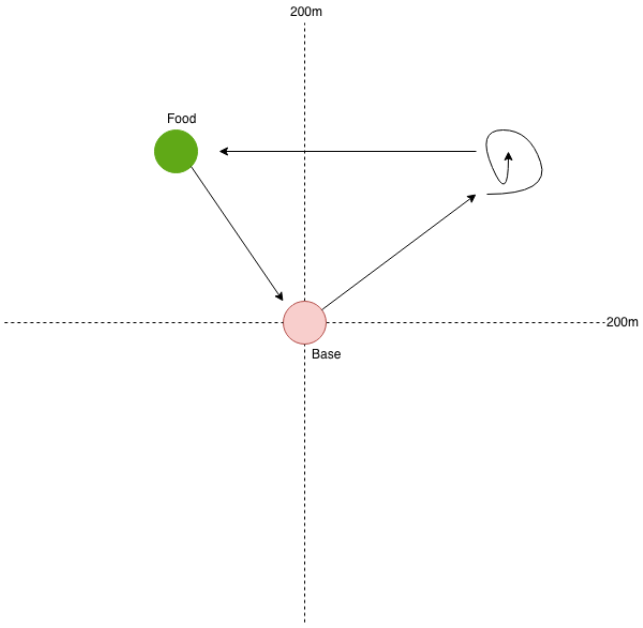


Figure 124 | A visual representation of the behavior of ants as they search for food in their environment.

The area in which the RASSOR will move will be represented as a cartesian grid with the base located at (0, 0). After the surrounding area is mapped using a Simultaneous Localization and Mapping (SLAM) algorithm, we will then set a 25 square meter search area located 100 meters away from the the base to begin searching for optimal dig sites.

Area selection will be achieved by passing a 25 square meter cell over the mapped grid and iteratively evaluating each position. At each position, a search area's value is calculated using an evaluation function that will calculate an areas value based on a set of features (Level ground, number of obstructions, etc.). Figure 125, below illustrates this process.

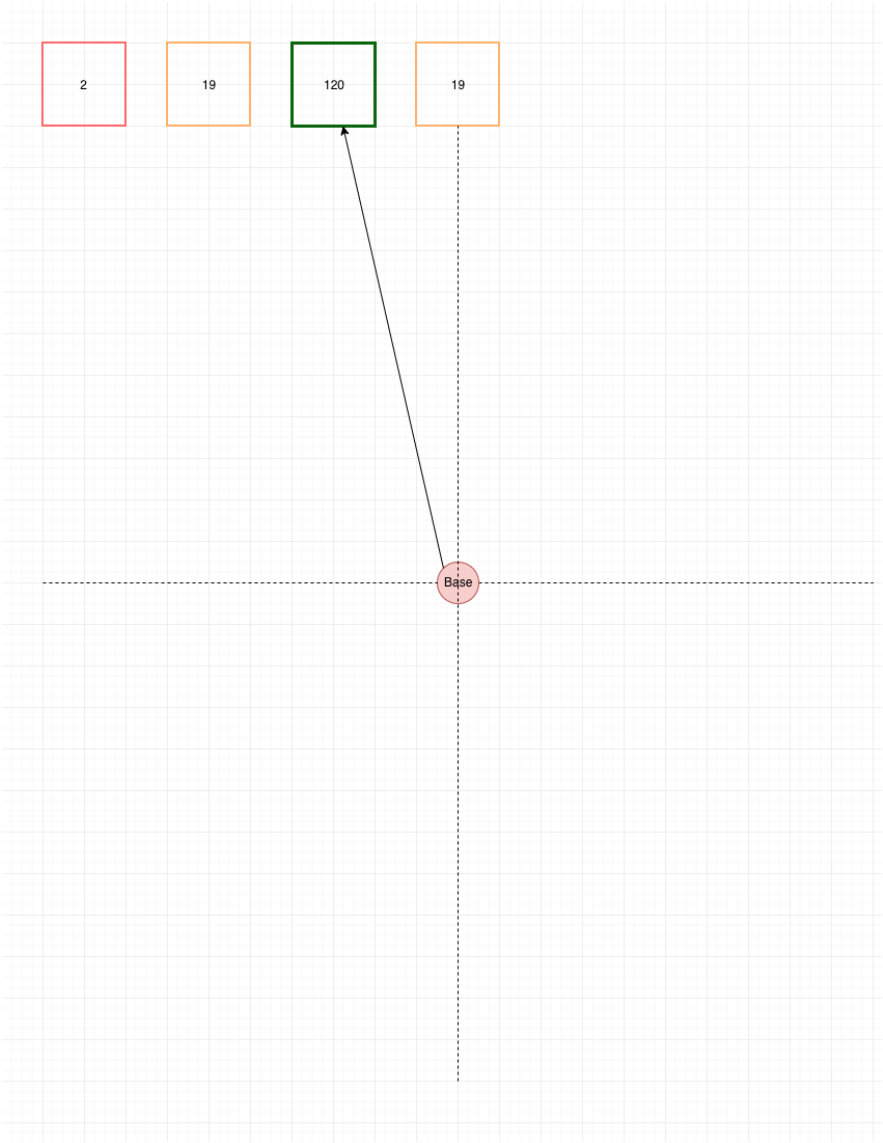


Figure 125 | A visual representation of how search areas will be determined. At each position a search area's location is assigned a value expressed by an evaluation function. This value will designate the area as a viable location to begin search for a dig site.

This candidate search area will then be passed into a priority queue at each evaluation, provided its value surpasses a threshold still to be determined. This threshold will protect against edge cases where all but one search area is realistically searchable. Candidate areas will be prioritized based on highest total value. The priority queue will then pop the first *N* candidates off of the queue, one at a time, and the candidates' center cell coordinate will be sent to a EZ-RASSOR until all *N* robots receive their respective coordinates. Each robot

will then travel away from the base to their respective coordinates to begin a search routine for an optimal dig site.

This search routine will then be executed using the onboard vision system to help identify possible dig sites. The search strategy will be executing a spiral pattern from the search areas center iterating outwards. This search algorithm will execute as follows:

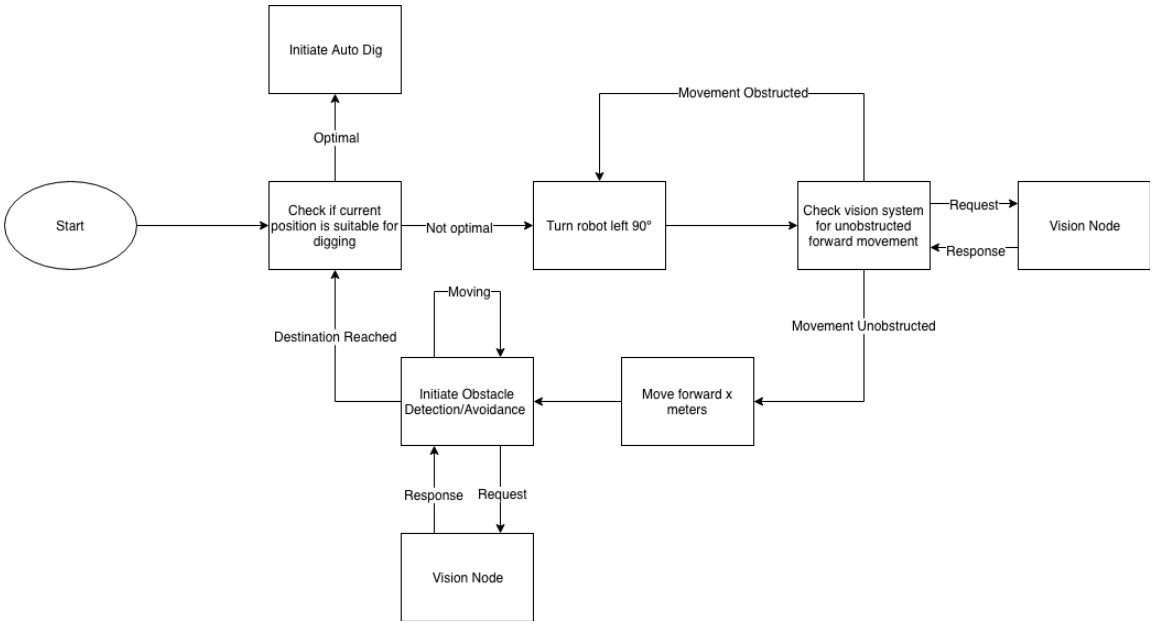


Figure 126 | State diagram of proposed search algorithm.

At each moment the robot reaches a destination it will determine whether a dig site is suitable. This will be done by first assessing the nature of the regolith on which it sits. Utilizing the side wheel facing camera system, wheel depth can be calculated which would imply a soft, easy to dig surface. The rear facing camera will also check for wheel tracks left by the robot to further characterize the surface quality. Should the site be suitable for digging, the robot will initiate the auto dig routine to begin mining the surface. If the opposite is true and the site not be suitable for mining, the robot will make a 90º turn to the left and send a request to the vision system to ascertain if forward movement is possible. If movement is obstructed the robot will execute yet another 90º left turn and query the vision system once again. Should the way be clear, the robot will set a target destination x meters ahead and begin moving toward that location. Along the way it will constantly make requests to the vision system to detect obstacles and execute the necessary avoidance maneuvers while maintaining its course

towards its target. Once the destination is reached, the process loops back to quantify the quality of the location's surface. Forward movement will have to increase by a incrementing weighted sum every two 90º left turns to maintain the spiral behavior of the search algorithm. However, if a robot finds itself in a situation where forward movement is blocked on all three sides the weight applied will have to be restricted from increasing to protect the integrity of the spiral pattern. Once the search area has been exhausted with no success, the robot will return to the base and a new search area will be set by the base as it pops from the priority queue.

There is some probability that two neighboring search areas could get assigned their own EZ-RASSOR to search within them. In this case we can make take steps to limit these situations from occurring. For example we can assign robot search areas to different quadrants of the map or multiples thereof, depending on the number of robots, so as to never allow robots in the same quadrant during search.

Optimal dig sites must consist of a clear plot of land large enough for the EZ-RASSOR to move in and out of, it must be relatively level ground, and it must be free of any hazardous objects or characteristics such as cliffs, crevasses, or pits the EZ-RASSOR may fall into. Ideally, the dig site would also consist of observably loose material which would ease the job of the EZ-RASSOR as it begins to dig. One solution would be to use a set of QR codes which can be easily identified by the onboard camera system. A set of positive QR codes and a set of negative QR codes could be used to simulate a sort of adversarial element to the search space. For instance, a suitable dig site may be within view but right next to it is a cliff or ditch which will decrease the value of the site and should discourage the robot from mining there. Another solution in regards to the simulation, is to randomly generate terrain that has assigned values the EZ-RASSOR can reference when in search mode.

When a dig site is found, the resources gathered, and the EZ-RASSOR has made its way back to the base, it will then query the base for any updates on the states of it's fellow robots, as well as update a database of a successful search and upload the coordinates of the dig site as well as a timestamp of its successful return. This timestamp will be used to quantify the richness of the dig site. Should the EZ-RASSOR make frequent successful returns to the base, it would stand to reason that the location is close and rich in material.

The goal is to simulate the pheromones ants lay down as they return to a nest once a food source has been found. This pheromone trail lets other ants know a food source has been found and guides them towards that food source. That trail becomes gradually stronger as more and more ants return with food and reinforce the trail with their own pheromones. We hope to achieve this behavior by attaching an ever increasing weight to the difference in time between visits. As more EZ-RASSORs return successfully and update the database, the priority value of this dig site will grow, requiring more attention from the robots.

There are only two reasons a robot would return to base, with the exception to recharge, and that is to offload materials due to successful mining, or to return from an unsuccessful search and to receive new search coordinates. However in this case, known dig sites will take priority over search commands, so any robots returning to receive new instructions will be sent to the dig site to optimize mining efforts. Should more than one dig site be found and logged into the database, priority will be set to the site with more frequent check-ins. This will ensure that the closer site is consumed over another site which may be further away. This behavior will have to be counteracted by a cooldown timer which will allow the simulated pheromone trail to degrade over time as to protect robots from not indefinitely returning to the same location long after its resources have been consumed.

After a dig site has been consumed and the trail runs cold that sites entry is then deleted from the table and the next available dig sites take priority.

Location ID	X	Y	Last Check In	Current Check In
A46F31	34	12	12:05:44:36	12:25:14:59
E12CCB	56	19	12:10:23:11	12:18:23:22

Figure 127 | A representation of the table maintaining dig site data on the base.

Over time the priority queue containing any and all remaining candidate search areas will eventually become empty. At that point the robots will have to begin mapping a larger area. Once this larger area is mapped the same process will

begin to identify and classify possible search areas as well as to command searches and mining protocols. Over time this search space could grow to become quite large, and assuming future missions will establish multiple mining operations it would be safe to assume the older mining operations would have had more time to map their respective areas and therefore have set a very wide reach. Should the mining operations be established as neighboring bases, a multi-base approach may be a better approach moving forward. In doing so we would allow the robots to be free to visit any of the various bases to off-load materials and update dig site locations. Should a robot find a suitable dig site, the optimal action would be to deposit its findings at the closest base rather than the base which its last check-in was made. Although this task may be outside of the scope of the project, it is a problem we hope to explore as we move forward.

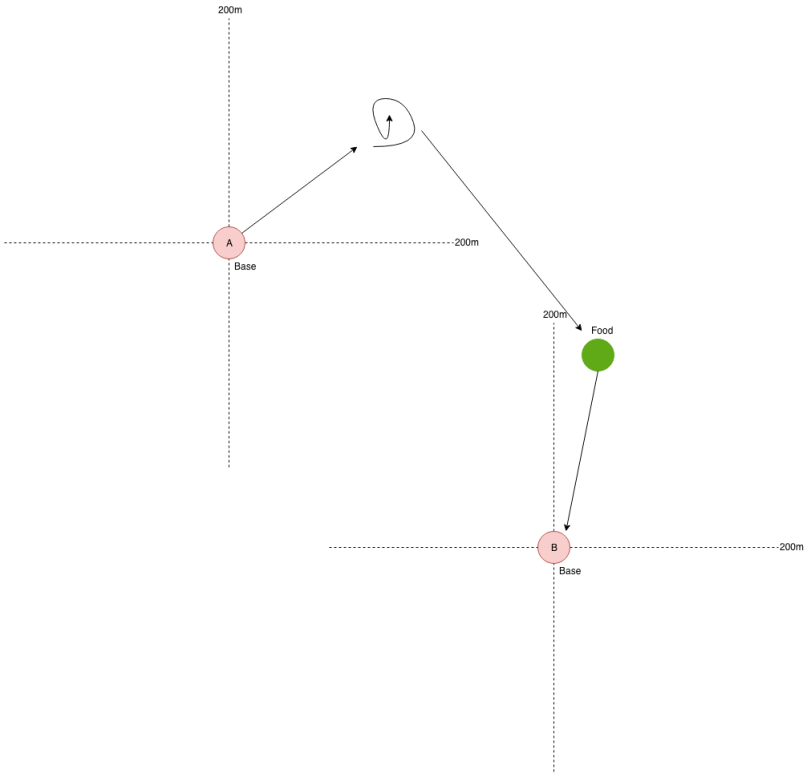


Figure 128 | In this illustration the benefits of allowing robots to visit multiple bases is clear.

This ensures that robots which are also checking in at base B will be taking advantage of the short distance to the mining site rather than the robots

associated with base A which may be further from the site. This strategy can be highly scalable if successful and could allow for very large mining operations.

Swarm Robotics Test Plan

During testing we will explore the capabilities of our approach as it pertains to swarm behavior. We will begin with simulation in Gazebo.

The first test to be executed will be to test communication between the base and the assorted robots. Swarm-In and Swarm-Out topics will be tested to observe whether commands and data are being received and handled in the correct way.

We will test our Search Site Evaluation functions against randomly generated maps containing surface quality values. It will be during this testing that we will define our thresholds in which to assign Search Area classification.

We will test the the search algorithms against a varying degree of placed obstacles and suitable dig sites within a search area. It is here where we will judge whether a set list of search protocols will be beneficial or if true decentralized planning agent will be a more practical solution. We will average the number of success and failures we observe statistically analyze and document our approach.

Likewise, we will test our mapping algorithm against a varying degree of placed obstacles and hazards to create a statistical model of our approach. We will test these various approaches using a varying number of swarm sizes (4, 8, 16, 32, , etc.). In doing so we will test average robot collision, average resource gathering time, and average search time, this will provide information on a swarm threshold which is easily maintained as well as efficient.

We will also test results in using a single base vs. multiple bases. We will observe how efficiency is affected by the number of bases in an environment.

Facilities and Equipment Provided

Orientation Description

On October 23rd, the team made our first trip out to Kennedy Space Center. After visiting the badging center, we made our way to building M7-0360. Here is where we performed an obligatory respectful workplace seminar. We then signed a non-disclosure agreement as well as enduring a brief explanation of information we will be exposed to that is subject to export control.

After this we met our project mentor, Jason Schuler. He is a Mechanical Engineer that is working on the full-size RASSOR robot. He was incredibly excited to meet us and he shared his interest and passion in this project. He informed us that were working on cutting edge technology and NASA has never sent a robot of this likes of this to space ever before. Per Jason, “The RASSOR needs to be tough, rugged, and smart. It needs to overcome any challenges it faces and never fail”. He really put into perspective the importance of the RASSOR and how if it’s successful, it will change space exploration and colonization significantly.

Jason then gave us the incredible opportunity to take a tour through NASA’s Swamp Works building. Swamp Works is a place that establishes rapid, innovative, and cost-effective exploration mission solutions through a highly-collaborative “no walls” approach, leveraging partnerships across NASA, industry, and academia. The goal of Swamp Works is to accelerate innovation for NASA and for Earth-benefit, from the idea stages, through development, straight into application. Here is where we got to meet the real RASSOR face to face.

Sadly, the RASSOR was in pieces during our visit due to upgrades that were being performed on it’s internals. Nonetheless, we were granted with a plethora of information pertaining to the RASSOR and the goals it is trying to achieve. Jason made it clear that due to the capacity of the shuttle, it is wasteful and expensive to send heavy things to space, at least with our current technology.

This is where the RASSOR becomes useful. Jason made the analogy “If you were making a road trip from Orlando to Los Angeles, you wouldn't take all the fuel you needed for a round trip with you before you leave”, this applies to space exploration as well. NASA wants to be able to take a relatively light payload to the Moon or Mars, and then make the resources they need at the destination to return. This practice is resourceful and sustainable. If we are able to make landing pads, rocket propellent, etc. on an extraterrestrial planet, this allows us to lower the cost of space exploration considerably, leading to rapid innovation and development outside of Earth.

Jason also shared some successes accomplished and challenges faced. Like how to deal with hilly terrain, being that the Moon is subject to meteor impacts the surface is far from uniform. Their solution is as follows. As torque on rotating drums increase, meaning a positive slope on the terrain, the RASSOR will raise it’s arms. As torque on rotating drums decreases, meaning a negative sloped terrain, the RASSOR will lower its arms lower arms. Due to low gravity, electrostatic forces, Van der Waal forces, and high friction forces between the particles of regolith, cohesion of the granular material is increased. This denotes that the regolith it is very easily compacted. This raises the issue of clogging the drum scoops. The most effective way NASA has found to avoid this is by excavating the very top layer of the regolith, only about a fourth of the scoop’s width. This proves to be a challenging feat in combination with the issue of a hilly terrain. In later designs of the RASSOR, NASA plans to address in issue by opening the size of the scoop opening to prevent bridging or by vibrating the drum to free the particles. Every one of the RASSOR’s movements need to be measured and exact, and NASA can not afford to have the RASSOR loose functionality or need manual repair. Jason stressed that this needs to be a very smart workhorse robot.

Figure 129

In relation to the RASSOR, Jason showed us the regolith simulant test bin as well as their “Dust to Thrust” regolith processing plant. The regolith simulant test bin has a very interesting story behind it. In Figure 130 (shown below) we see a large windowed box and directly to its right is a much smaller box which was the first version of the test bin. The reason it is so small is because that is all NASA

could afford! What made this so expensive was the artificial regolith it contained which was engineered by NASA. They started with samples of lunar regolith supplied from previous lunar missions. They were able to determine the exact composition of the regolith, find those components locally and mix them together to recreate lunar regolith, or at least their best interpretation of it. The issue with this was that the process costed approximately $100,000 per ton. This led to a less-than ideal sized test bin. Fast forward a few years later to one of NASA's annual mission tests at the U.S. Army's Yuma Proving Ground in Arizona. An astronaut working at the time noticed large piles of dirt scattered around the desert. He made the conscious decision to try and climb up to the top of one of them. What ended up happening was the astronaut sunk into the pile up to their waist and was unable to get out! He inevitably had to be rescued, but he made an interesting observation. This astronaut was been on a lunar mission before and made the correlation that these mounds of dirt highly resembled lunar regolith. As it turns out these mounds were waste from some government operation. NASA took samples of these mounds and found that not only did it highly resemble lunar regolith, but it was better than what they had previously engineered! Ultimately, NASA made a deal to buy this material for only $4 a ton. This was a huge success and allowed NASA to create the 625 sq. ft. testing bin that is in Swamp Works today.

Figure 130 | A view from inside Swamp Works. The large windowed container is the regolith simulant test bin. The RASSOR uses this test bin to test its functionality in an extraterrestrial-like environment.

After our tour of Swamp Works, our orientation came to a close. Jason set the expectation that we will be meeting once again early in November to dive further

into requirements and provide us with data and materials that would aid in the successful completion of our assignment.

Implementation Phase

This development phase took place strictly during the Spring 2019 semester. In this phase, the UCF development team was responsible implementing the requirements that we agreed to in the Statement of Work Document. Moreover, the team also had to address design challenges while implementing the execution plan we had mapped out in the previous semester. Deviations from the pre-defined execution plan are stated in this section as well. This information is beneficial to show the differences between real-world testing and conceptual development.

Government Shutdown

The UCF development team began the implementation phase on the first day of the Spring 2019 semester: Monday, January 7th, 2019. Our requirements gathering the previous semester had been done in direct relation to the Swamp Works engineers who are of course government employees. Unfortunately, the US government had shutdown on December 22nd, 2018 and continued to remain shut down even after we began the semester. As a result, we were unable to contact the NASA engineers as they were prohibited from responding to any emails sent to their government email addresses.

This shutdown persisted throughout the first two weeks of our semester, which was a crucial time period for our team as we had “hit the ground running” on our requirements. Through this time frame, we moved forward with implementing our requirements but did have the constraint of working on this software without any idea of how the hardware would be designed. This is because we had intended on spending the first two weeks of the semester with the Swamp Works team to ensure that we tailored our software according to how they were designing their hardware. Instead, we were forced to come up with our best guess on the designs which influenced which development choices we made.

Thankfully, the government reopened on January 19th, 2019 and the Swamp Works employees were able to return to work. Our work at that time had

remained unchanged since the Swamp Works team had a higher priority to readjust to their remaining work upon return. We became extremely grateful for their dedication to this project because, even though they had to return to their other delayed commitments, they invited us back to Swamp Works only four weeks later on February 13th to show them our progress and discuss any design questions we had.

Based on this timeline, it is clear that the government shutdown presented a real challenge towards the success of this project. While the shutdown ended only two weeks into our semester, we were unable to get any feedback on our efforts until nearly six weeks into the Spring semester.

EZ-RC Engineering: Versions 1 & 2

The development of EZ-RC Version 1 was done on a Raspberry Pi 3 car kit. The goal of this version was to be able to test the current code we had on actual hardware. Our main goal with this version was to test our current code on the actual hardware. This would allow us to be able to test our code on real hardware in preparation of NASA's delivery of the actual hardware for the EZ-RASSOR. One of the main things we learned with this part of the project was writing a modular driver for the hardware.

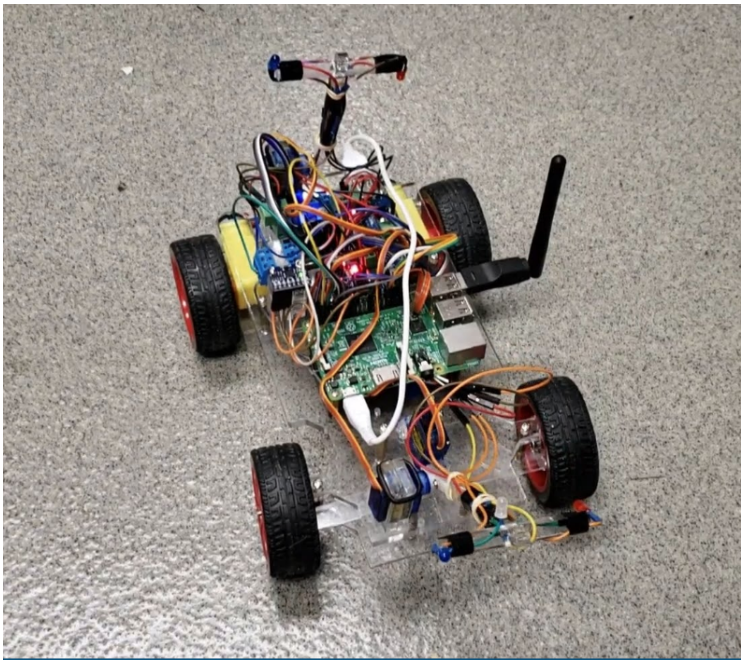


Figure 131 | EZ-RC Version 1

Overall EZ-RC Version 1 was a huge success. Building the hardware for it was a slight challenge for the team as we are not too experienced with putting hardware together. We were then able to test the code we had up to then. Such as the Flask request server, the controller app, and testing the bitstrings. Version 1 came with some problems, such as not having tank turning.

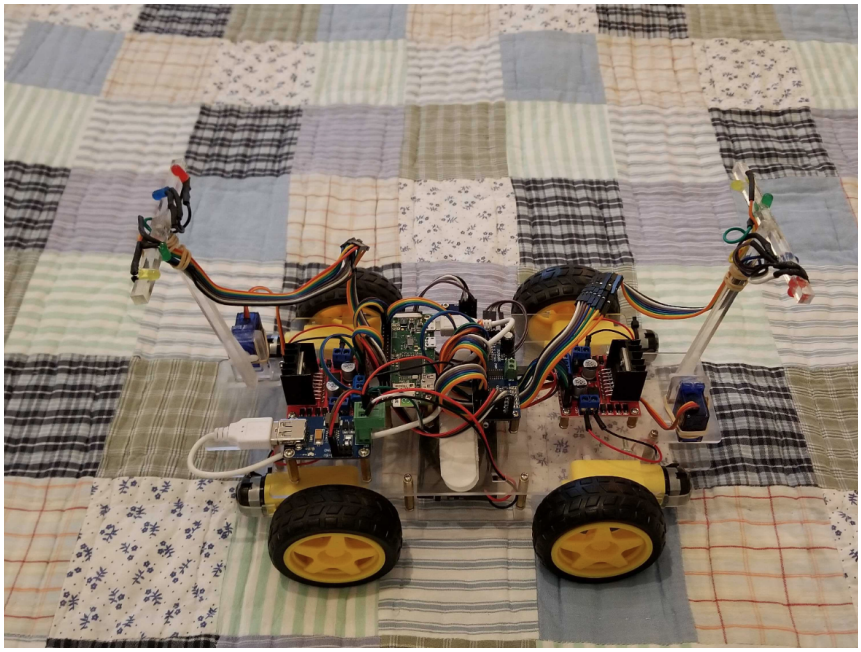


Figure 132 | EZ-RC Version 2

Version 2, was redone, to solve the issue of the tank turning, and it improved a couple of other features such as minimizing the hardware specification, such as using a Raspberry Pi Zero W instead of a Raspberry Pi 3. This change allowed for us to test the limits of what could support our code. Some issues did arise from this change, and in the end the hardware isn't fully working.

Mobile Application Engineering

Development of the mobile app was fully done and implemented in React Native. React Native allowed for a quick and easy deployment and testing of the app. Fortunately we were able to test the mobile application on both Android and iPhone, both working well in testing.

While the application had initially begun as just a simple controller for the the EZ-RASSOR for just movement. From the beginning that was the goal of the mobile app, in fact, most of the mobile app was written and developed before the rest of the project was up and working for actual testing. For that reason, the mobile app was developed simply within a single main file. The reason behind that was the original simplicity. It was just expected for the app to be able to just perform these functions.

As development progressed throughout the rest of the project the functionionality required in the app became much more complex. Other features that were included were then eventually added was the ability to toggle the autonomous functions.

One of the initial issues that were faced was with the implementation of multi-touch functionality. Multi-touch was supposed to be implemented so that the user would be able to press and hold more than one button at the same time. The reason for this is because of how when moving with the app you would want to press two buttons at the same time such as movements in the up/down and left/right direction. Due to the nature of React Native's implementation this was not as easy as it sounds. The way the app sends commands to the backend is through an API endpoint, when a button is pressed, a change state is triggered and then the callback request is sent. The problem was this was that when multiple buttons were being pressed the state of the app's bit string wasn't being parsed correctly. On the first press the request was being sent fine, but once a second button was pressed the app the app would register that the same first button being pressed a second time. Thus a forward and right butter press would register as a forward forward; even worse since the way the app handled the bitstring was by adding and subtracting the button values to an app state, it would for example add 2048 for the forward and then 2048 again, thus the state would become 4096 and cause the app to become unresponsive. A whole lot of research was done in how to implement this functionality. Certain components had to change in order to be able to work. The state management had to be revamped in order to support this. Even after these changes were implemented on testing on the actual hardware it wasn't working. After even more research we found out there was ongoing open github issue for React Native on Android. Testing on iPhone worked without an issue. Eventually the app was left as is only fully working with the multi touch, on the iPhone.

Figure 133 | A screenshot from a working mobile app

Driver Station Engineering

During the design phase of the project it was thought that it would best to make use of rqt_gui implemented in ROS. However, once the development phase began, it was quickly realized that it would not be able to achieve the desired result since rqt_gui heavily limits the developer's ability to customize the GUI. It restricts the user to using premade rqt elements that could not be customized and piece them together like a puzzle. The issue with these premade rqt elements is that they are designed to be very generic, so that they could be implemented in any ROS system, and the design is focused on being used by developers familiar with ROS instead of the end user. The issue with the rqt elements being too generic is that most of them looked clunky and would display all possible information of that type. For example, if implementing a graph to display value changes in certain nodes, the rqt element would have a dropdown for all nodes even if it was unnecessary to see or if there were no values that would be changed within them. This is an issue with rqt's complexity: many of the rqt elements would be too confusing for users. However, it was desired that the GUI be as intuitive as possible. While rqt_gui would be a good choice for a small project used almost entirely by the developer, it was not a good choice for the desired result. Thus, we ended up terminating development with rqt_gui.

Since rqt_gui was no longer being used, the GUI needed to be built from scratch. The development of the GUI application began with using Qt Creator to design the visual aspects of the GUI and then converting to Python code using PyQt.

After the GUI was converted to Python, functionality can be added to it. The issue with this approach was that changes could not easily be made to the design of the GUI through Qt Creator once it was converted to Python. The workaround to this that was initially used was to have a widget-based design approach. In this design approach, the desired features of the GUI would be broken down into standalone widgets that could be created and tested by themselves. This design approach would heavily decrease the chances of needing to redesign the GUI through Qt Creator since separate simpler GUIs would be created for each widget instead of designing a GUI containing the entirety of the GUI. After the widgets were created and tested, they could be made into QObjects and be pieced together into a master GUI that would accommodate all the widgets.

Fig 134 | Widget to display raw camera footage

During the design phase it was suggested that GUI features would be quickly added and removed throughout the development process. This was anticipated

because it was initially believed that adding many features would be a simple process and features determined to be the least beneficial for end users would be removed to make the GUI more intuitive. However, since the GUI was now being built from scratch instead of implementing the rqt_gui it was necessary to more carefully consider the features that would be developed. Now a feature would not be developed until it was deemed to be beneficial for the end user and would remain in the GUI for the entirety of the project. Since features that may be removed further in the development process were no longer being developed, it was decided to no longer implement features that would be used only by developers. This decision was made to not only save development time but because the developers preferred to use the terminal instead of the GUI once they became familiar with ROS. This was because the use of terminal was often faster than loading and using the GUI, similar to how experienced Linux users would use the terminal for directory navigation instead of the GUI.

Development using PyQt and the widget-based design approach continued until encountering multi-threading issues. The ROS framework utilizes threads and some attempts at accessing data from a subscriber would cause the GUI to crash as it was contained within a separate thread. Up to this point one of Qt's most powerful features, signals and slots, was being underutilized. The signal and slots mechanism can be used to connect objects and allow them to communicate with each other. An object can emit a signal when a particular event, such as a button being clicked, occurs. Once the signal is emitted any connected slot functions are called. This feature was originally being used in its most basic form, to connect buttons and dropdowns to callback functions. After some further research on Qt's signals and slots, it was realized that it can be used to safely communicate to ROS and avoid the multi-threading issues. When a subscriber receives data, the callback function no longer attempts to directly display it to the GUI. Instead the callback function stores the data in a simplified form and emits a signal the matching slot function is then able to obtain the data and display it to the GUI in the desired form. While this solved the multi-threading issues, it required the GUI to be written in C++. Since the previously implemented features needed to be rewritten to C++ and the visual aspects of the GUI could once again be dynamically edited within QtCreator, the widget-based design approach was abandoned.

GUI Features

In the original designs of the GUI, we were attempting to provide the user with as much information at a single time. Thus it would display all types of information at

all times and rarely require the user to interact with the GUI so they can passively monitor it. While developing the GUI it was determined that it was not ideal to bombard the user with too much information at a single time if the GUI were to be as intuitive as possible. Additionally, it was realized that it was more likely for users to be actively interacting with the EZ-RASSOR than passively monitoring it for hours. Thus, the GUI was redesigned to be simplified and reduce the amount and variety of information that the user would be viewing at any given time. To accomplish this, the GUI now implements tabs that the user can switch between with each tab displaying similar types of information in a simplified format.

The main interaction that the user has with the GUI is through the side panel which stays displayed no matter what tab they are viewing. If the user is connecting to a single robot within a locally launched simulation, the GUI will be able to automatically connect them to it after clicking the connect button. If they want to take control of another instance of the EZ-RASSOR or connect to a simulation on another computer, they can change the ROS Master URI and Host IP before connecting, though it is required that they both are on the same network. The side panel also contains a launcher which allows the user to easily launch simulations or nodes. This launcher significantly simplifies the use of the EZ-RASSOR as it allows them to simply open the GUI and launch any simulation they desired. It also allows the user to only use the nodes that they wish to use instead of requiring them to run them all every time. Finally, the side panel contains an alert system using status message. This will inform the user of the condition and activities of the EZ-RASSOR using high-level language. This allows the user the to quickly figure out what the EZ-RASSOR is attempting to do and be notified if there are any issues that the EZ-RASSOR is able to detect. Each status is assigned one of five types: debug, info, warn, error, and fatal. The debug status type would be used when checking the condition of the EZ-RASSOR and no issues are encountered. The info status type would be used to inform the actions that the EZ-RASSOR is attempting to perform such as digging or driving back to the base. The warn status type would be used when detecting issues that may not be causing any current issues but will cause issues in the future if left unfixed such as the battery getting low. The error status type would be used when an unexpected event occurs that may or may not cause an issue, such as the EZ-RASSOR deviating from the current task, and it would be recommended that the user check the source of the issue when possible. Finally, the fatal status type would be used for critical errors that require immediate attention and may need tele-operation, maintenance, or retrieval. Each message will also contain the time that the status displayed at to allow the user to easily check when the event occurred on past status messages.

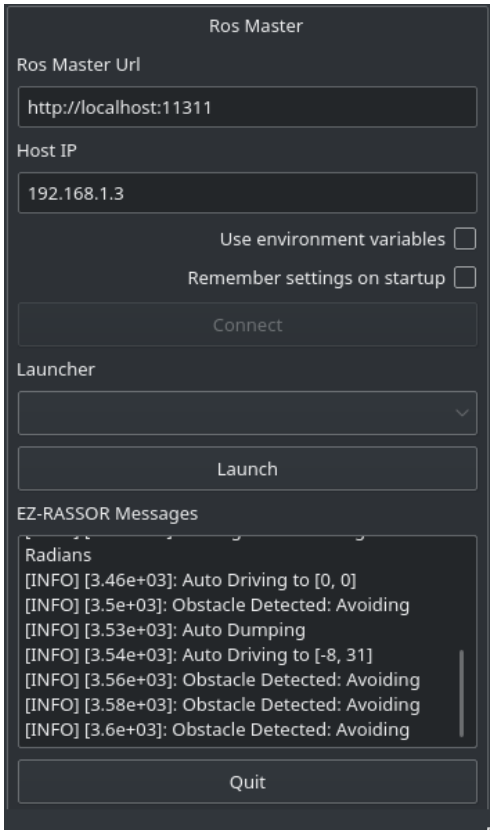


Fig 135 | Side panel of GUI

The video feed tab of the GUI contains the visual information obtained from the cameras of the EZ-RASSOR. The initial design of the EZ-RASSOR was going to use a stereo camera for each arm, allowing for potentially four camera feeds. Due to this the original design approach, it was originally intended to display the raw live footage of a single camera at a time and allow the user to switch between displays through the use of a dropdown menu. As the project progressed, NASA modified the design of the EZ-RASSOR to only use a stereo camera on the front arm. Because of this and the display of the GUI being simplified, it was decided to simply display both of the camera feeds for the front stereo camera at the same time. In addition to the raw camera footage, the video feed tab displays the disparity map that the EZ-RASSOR generates to give the user a better understanding of how the robot is processing the camera footage.

Within the design phase it was suggested to incorporate Rviz into the GUI to assist the development process. RViz is a powerful ROS tool that proved to be useful during the development process but it was easy for the developers to access through the terminal. Additionally, the full implementation of RViz is too

complicated for end users unfamiliar with ROS to handle. While the full implementation of RViz isn't useful for the end user, it can be broken down into simpler components to display information a graphical way that is easier for the end user to interpret. The three features the GUI uses RViz to implement is the IMU display, point cloud map, and pose viewer. The IMU display uses RViz to display an arrow in a blank 3D environment, this arrow indicates the current orientation of the EZ-RASSOR. In the correct orientation of the EZ-RASSOR, the arrow is pointing straight up but will often slightly move around due to the elevation of the environment not being perfectly flat. Along with the visual display of the IMU the IMU display tab also shows the current values of the orientation, angular velocity, and linear acceleration. The point cloud map is displayed within a movable 3D environment through the RViz plug-in. The implementation of the point cloud map allows the user to see how the EZ-RASSOR interprets the world around it. This would allow the user to better understand the navigation decisions made by the EZ-RASSOR while it is autonomously driving. For example, the EZ-RASSOR may be taking a path that the user believes to be inefficient and the user can check the point cloud map to see if the EZ-RASSOR interpreted an obstacle in an alternative path. Finally, the pose viewer displays a movable 3D model of the EZ-RASSOR in the interpreted current position. This can be advantageous to the user to check for mechanical issues with EZ-RASSOR or any errors in its logic. For example, if the EZ-RASSOR is attempting to dig with its arms in the air, the user can check the pose viewer to see if it believes the arms are down.

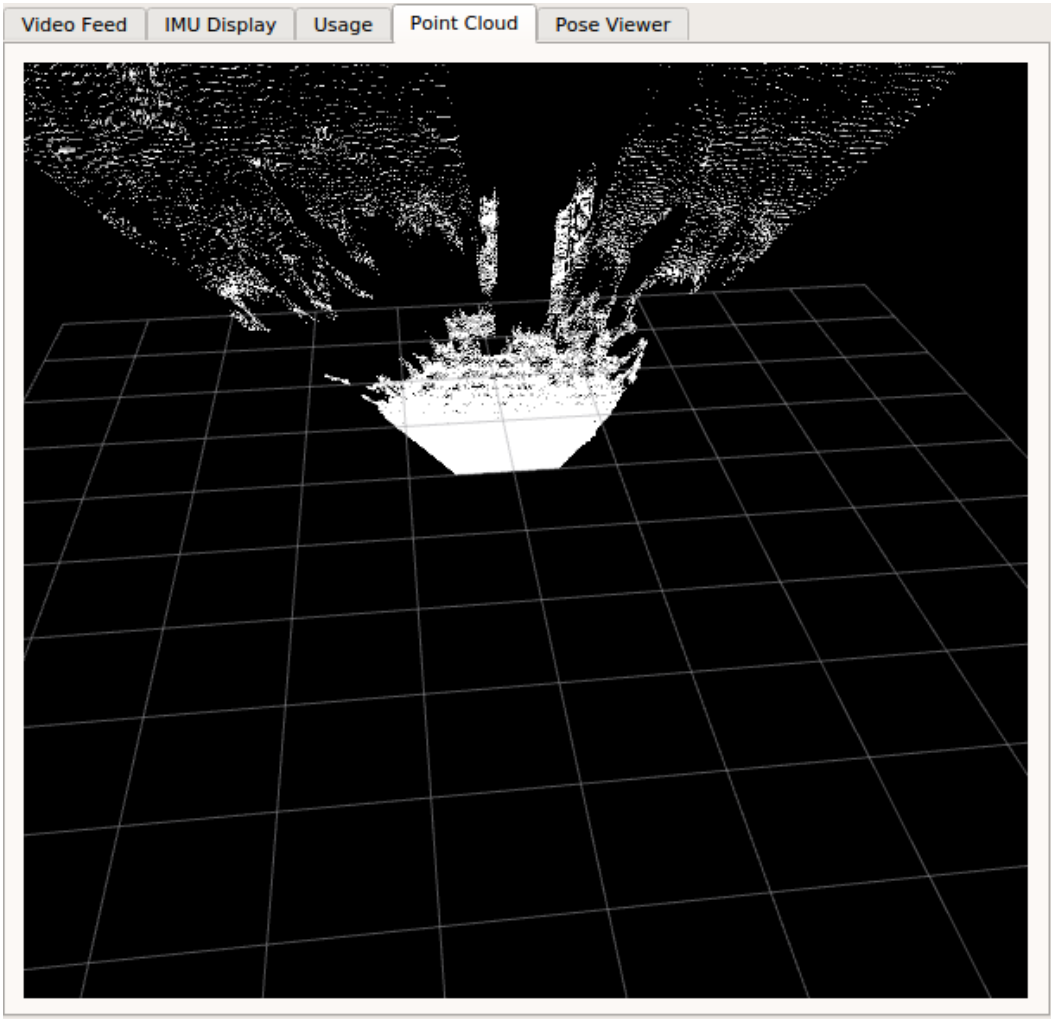


Fig 136 | Point cloud map

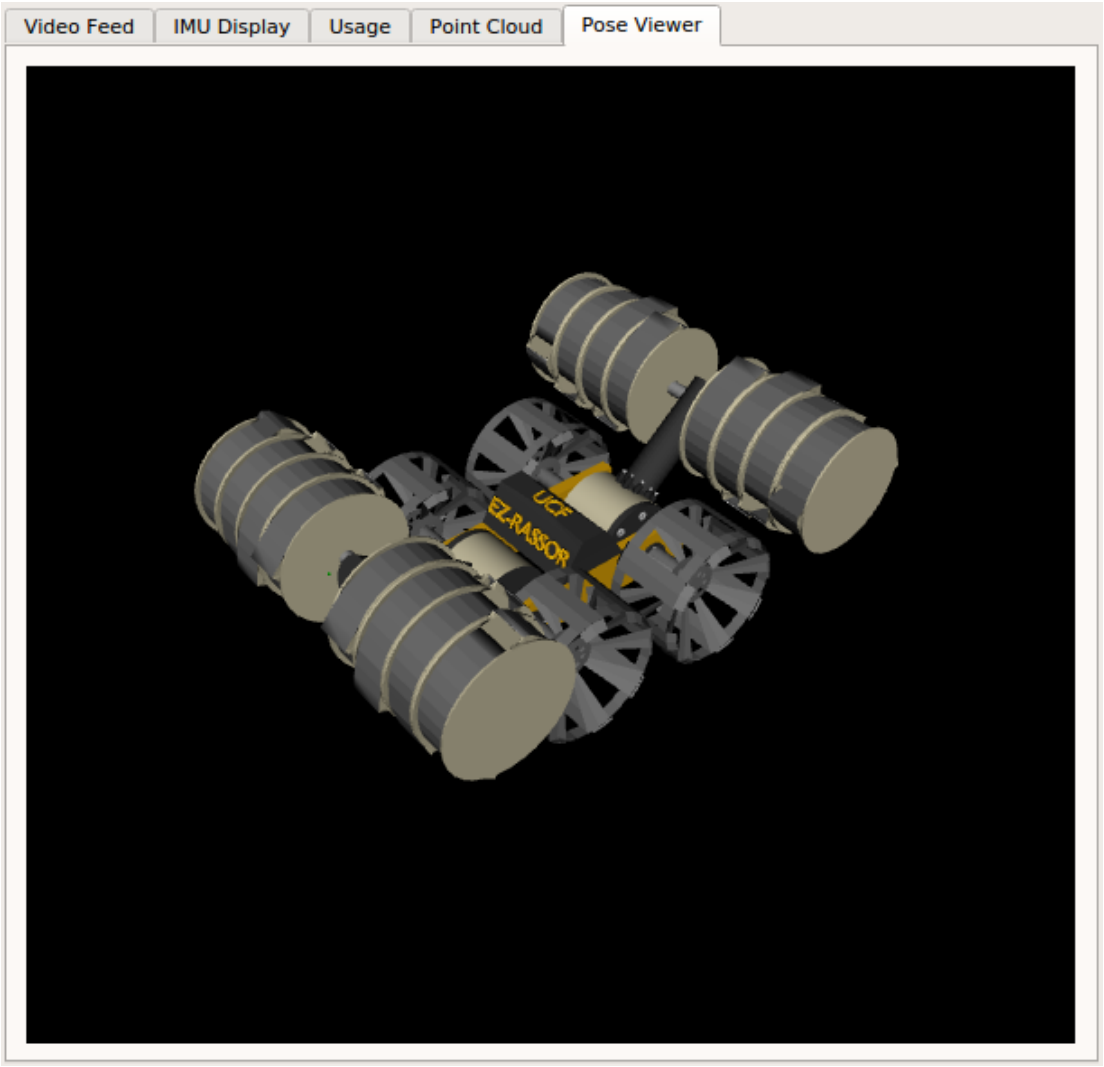


Fig 137 | Pose Viewer

The originally proposed design for the GUI has several major differences when compared to the current iteration. All of these changes resulted from changes with the physical design of the EZ-RASSOR and to improve the experience of the GUI for the end user. An overall simplification to the design allowed the GUI to be intuitive to users unfamiliar with ROS and the EZ-RASSOR while not limiting the ability for an experienced user to easily monitor and interact with the system.

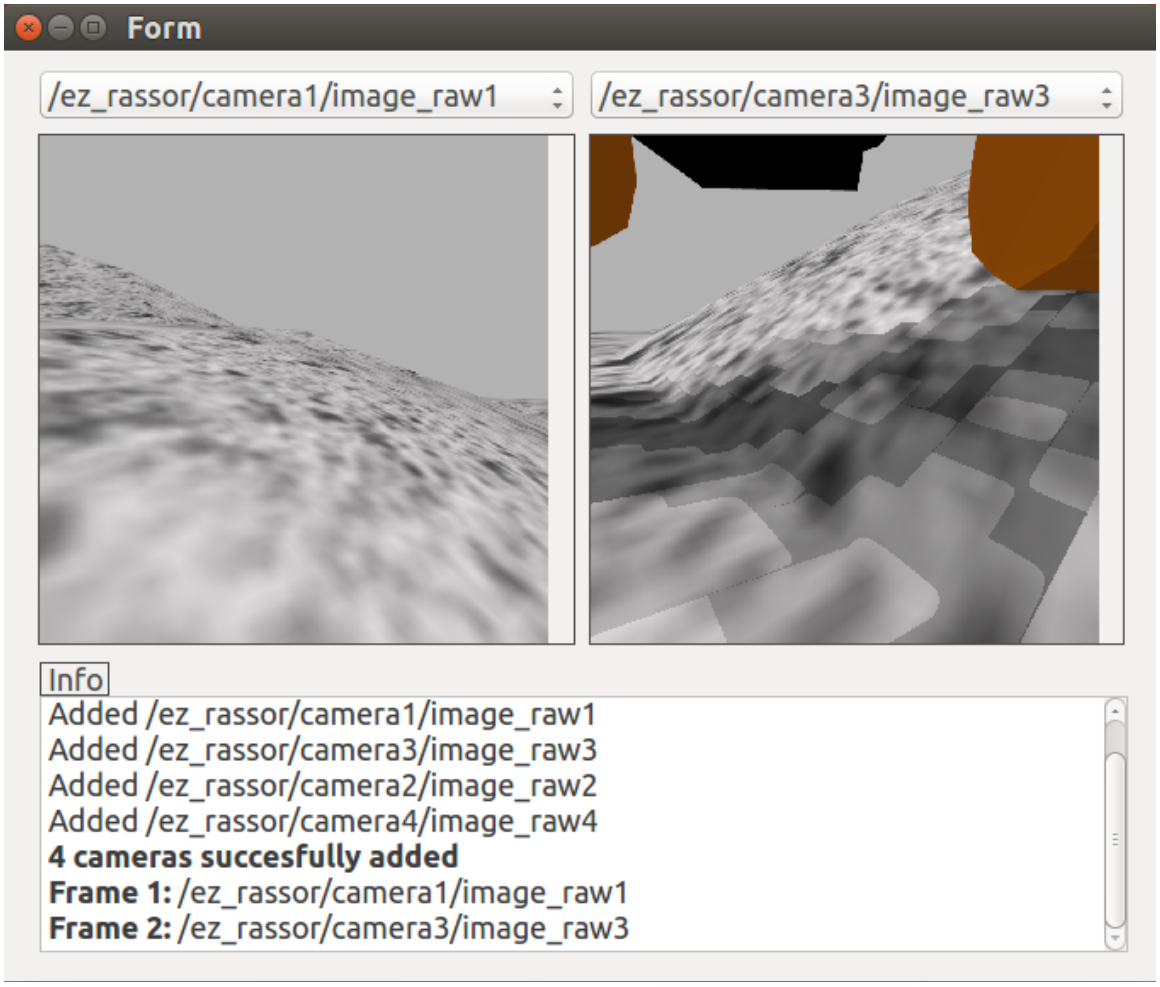


Fig 138 | Implementation of GUI during CDR

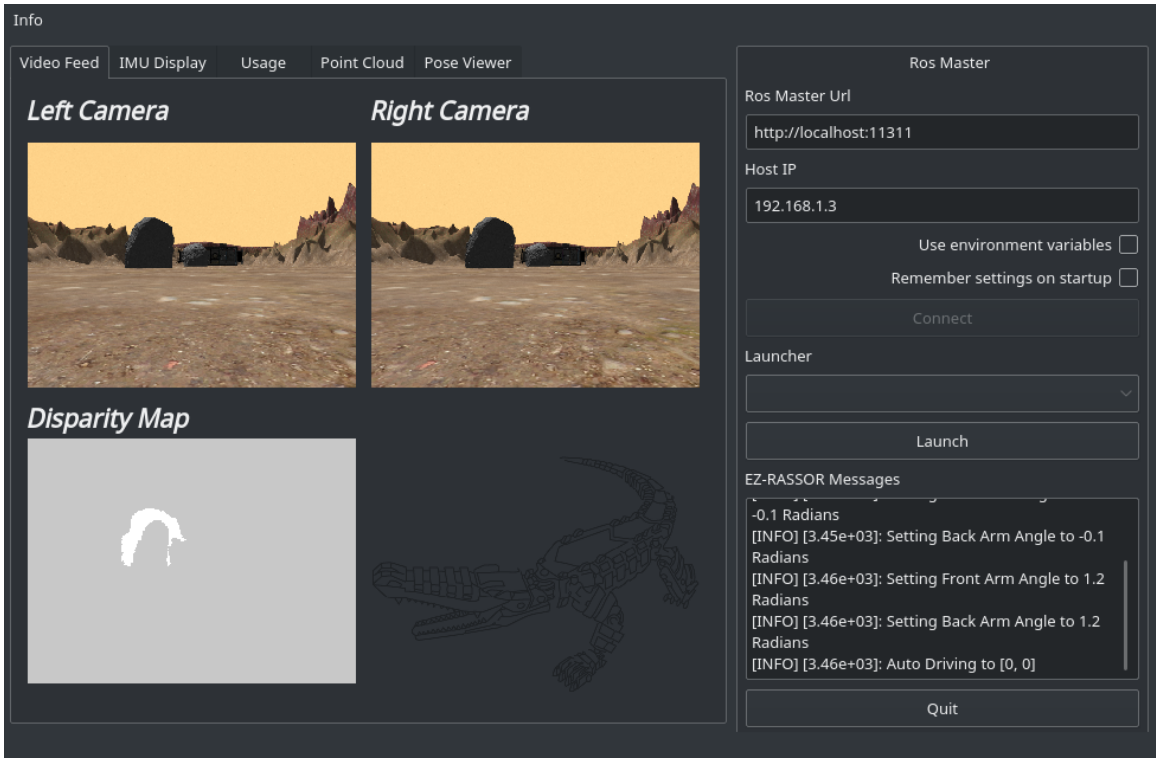
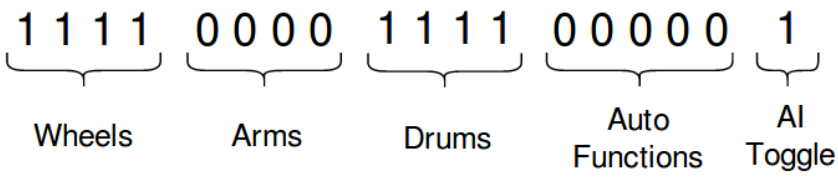


Fig 139 | Current implementation of GUI

Bit String Implementation

Seeing how ROS is a cornerstone for any modern day robotic system, we developed a ROS architecture we felt was safe, modular, and efficient. The efficiency comes from the type of data we chose to pass around a system. The data type is a bitstring represented as an integer. All movement commands and system toggles are initiated through an 18 bit long bitstring. Each bit represents an on, 1, or off, 0, for a given action. The first four bits control the wheels. In order these bits correspond to left wheels forward, left wheels reverse, right wheels forward, right wheels reverse. The following four are for the arms and represent front arm up, front arm down, back arm up, and back arm down. Following arms we have the drums. In order we have front drum dig, front drum dump, back drum dig, back drum dump. The next five bits after drums represent “auto functions.” In order these functions are auto drive, auto dig, auto dump, self right, and full autonomous mode. The last bit in the string is the AI kill bit. This is a safety precaution to insure we can regain control of the EZ-RASSOR even if it’s in the middle of an auto function. The system is designed to ignore competing commands. For example, if both front arm up and front arm down were activated, both commands would be completely ignored, and no motor function would take place. The system only acts on valid, non-competing commands.



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

Fig 140 | Bit string architecture and values

Controller Mappings to Joy and Hardware Motors

Joy Translator

To accomplish the task of taking joystick controller input and transforming it into individual motor and simulation movement, the Joy Translator creates and generates commands by parsing two arrays sent by the previous “joy_node” that encloses data related to which joysticks and buttons are being pressed and released on the controller. Once Joy Translator has accomplished the task of interpreting the arrays, it will set certain variables to specific values and then send these values to the lower level hardware. For wheel movements the node generates two vectors: one vector for linear velocity, i.e. straight line movement, and one for angular velocity, i.e. rotational movement. The two vectors can then yield information related to any possible movement the robot can make inside of a three dimensional space. As for wheel movements, the Joy Translator uses float values between 1 and -1 representing to forward and backward movement.

Exposing the Simulation to ROS

Our simulation work has been crucial in determining functionality of our ROS nodes since we are able to see the effects of our robot topics in real-time. Because of this, the simulation efforts were engineered to begin at the ROS-level. That is, we wanted to ensure that every time the simulation was started, it was spawned from ROS. (This design choice is standard and allowed us to spawn multiple pieces of the simulation on command.)

Launch Files

We achieved this functionality through the use of **launch files** which are special files written in XML format where we can specify which ros nodes and robots to spawn, the order in which they can spawn, and even the simulation environment that should be started for the robots to inhabit. Moreover these launch files do not have to be tied to the simulation. For real-world operation of the robot, the launch files will be used to spawn the various nodes that are essential for the normal functionality of the EZ-RASSOR. However, in the case of the simulation, we used the launch file functionality to pass parameters to the Gazebo client to affect various configurations on the simulation environment. All that is needed to spawn any individual launch file is to use the **roslaunch** bash command. The syntax here is **roslaunch [package_name] [launch_file_name]**.

Below is an example of a working launch file. As you will notice, we can take advantage of **launch arguments** to dynamically pass in different arguments that are needed in a launch file.

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
 <!-- Launch File to run the simulation environment
     Written by Ronald Marrero and Cameron Taylor-->

 <!-- Gazebo States -->
 <arg name="world" default="base"/>
 <arg name="use_sim_time" default="true"/>
 <arg name="gui" default="true"/>
 <arg name="headless" default="false"/>
 <arg name="debug" default="false"/>
 <arg name="paused" default="false"/>

 <!-- Create World Environment -->
 <include file="$(find gazebo_ros)/launch/empty_world.launch">
   <arg name="paused" value="$(arg paused)"/>
   <arg name="world_name"
        value="$(findezrassor_sim_gazebo)/worlds/$(arg world).world"/>
   <arg name="use_sim_time" value="$(arg use_sim_time)"/>
   <arg name="gui" value="$(arg gui)"/>
   <arg name="headless" value="$(arg headless)"/>
   <arg name="debug" value="$(arg debug)"/>
 </include>

</launch>
```

Figure 141 | Code Snippet from the sim_gazebo launch file

Because we are defining the *world* as a passed in argument, we can dynamically spawn the mars world and the moon world simply by specifying it in the command line. The defined arguments also have a default value in case no value is passed in. To successfully run the above launch file on the Mars world, you would simply need to run the following from a bash terminal:
roslaunch ezrassor_sim_gazebo sim_gazebo.launch world:=mars

Robot XML Format for ROS

While ROS and Gazebo as systems can communicate with each other almost seamlessly, both have their own formats for defining robots. ROS uses an XML-specification called the Universal Robot Description Format (URDF) while Gazebo uses the Simulation Description Format (SDF). Each has their own set of differences but our team came to a decision at the start of the Implementation Phase to use URDF over SDF since the former has 100% compatibility with the ROS infrastructure and allows us to use a greater set of plugins for the simulation environment.

When the simulation ROS launch file is called to spawn our URDF model, the model gets passed to the ros-gazebo package where the entire robot specification ends up getting converted to SDF. This allows Gazebo to still display the robot properly in the simulation world, and also allows our ROS graph to know about all parts of the robot since it is gathering information from the URDF version of our model.

This system has served our team well and our implementation would not have worked if we had stuck with the SDF format to define the model. Additionally, the conversion from URDF to SDF only happens on runtime and is transparent to us as developers and as users. The SDF that is generated gets deleted once the simulation ends which also shows that this method does not cause any unneeded data to persist on storage.

ROS Topics and Plugins

Gazebo is able to communicate with ROS so well due to the use of included **plugins.** These plugins allow Gazebo to perform such actions as publishing camera data to a ROS topic or even to allow for external control of specific objects in the simulation environment. While one can launch the simulation from ROS using launch files, this means that any additional interaction between the two systems once they are up and running in achieved through Gazebo plugins. This design for a system also makes sense, as we have been able to control which plugins are really needed in our system, as opposed to introducing a lot of overhead.

The EZ-RASSOR simulation robot has stereo cameras and an inertial measurement unit (IMU) and needs to be controlled from a gamepad. To expose this functionality to our ROS graph, we included the following plugins to the robot's specification:

- libgazebo_ros_multicamera.so
- libgazebo_ros_imu_sensor.so
- libgazebo_ros_control.so

With the first two plugins mentioned, we are able to define multiple properties of the plugin such as the topic name. This exposes the stereo cameras, for example, to a ROS topic immediately after the robot is spawned, allowing our scripts to subscribe to data from that topic. Lastly, the ros control plugin allows us to move the joints on demand which we are able to do in ROS via a movement_toggles topic. (This is the topic that our gamepad *and* mobile app communicate with to get the EZ-RASSOR to move inside of the simulation.) Here is an snippet from the EZ-RASSOR URDF specification where we are exposing the IMU to ROS via a plugin:

```
<!-- IMU -->
 <gazebo reference="imu_link">
   <gravity>true</gravity>
   <sensor name="imu_sensor" type="imu">
     <always_on>true</always_on>
     <update_rate>100</update_rate>
     <visualize>true</visualize>
     <topic>__default_topic__</topic>
     <plugin filename="libgazebo_ros_imu_sensor.so"
             name="imu_plugin">
       <topicName>imu</topicName>
       <bodyName>imu_link</bodyName>
       <updateRateHZ>10.0</updateRateHZ>
       <gaussianNoise>0.0</gaussianNoise>
       <xyzOffset>0 0 0</xyzOffset>
       <rpyOffset>0 0 0</rpyOffset>
       <frameName>imu_link</frameName>
     </plugin>
     <pose>0 0 0 0 0 0</pose>
```

```
  </sensor>
</gazebo>
```

Figure 142 | Code snippet from the ezrassor.gazebo file

As you have seen, the two systems, ROS and Gazebo, very easily communicate with each other but this communicate must be facilitated through the use of launch files (ROS -> Gazebo) and plugins (Gazebo -> ROS). Establishing both systems in our project set a solid foundation on which to test our autonomous functions and interpret real-time simulation data.

Creating the Simulation Environments

In our Statement of Work document, we committed to creating three worlds: a base world to serve as a control test, a Moon world, and a Earthquake world with various objects scattered across to demonstrate autonomous functionality. Since two of the three worlds mentioned take place on an Earth-configuration, we found that these were easily designed through Gazebo's world editor functionality, where various objects could be spawned from the simulation's online database into any pose that we desired. These environments could then be saved from Gazebo to a **world** file that we could spawn from a ROS launch file.

The Moon world however involved a significant increase in configurations. Properties such as the planet's gravity and slippage had to be configured so that it would create a reliable test environment. Additionally, the Moon is not a flat surface. If it were, then we could simply apply a texture to the floor in the simulation.

To overcome this problem, our team decided to take advantage of Digital Elevation Models (DEMs) which are three-dimensional representational maps of a planet's surface. These maps exist for multiple terrains including the Earth and even mars and can be generated from heightmaps using techniques such as Lidar. Excitingly, most of these DEMs are provided by the United States government at no cost since they are used by researchers around the world. Better still, Gazebo already ships with a built-in DEM for the Moon, all we had to do was specify the aforementioned properties such as slippage and gravity.

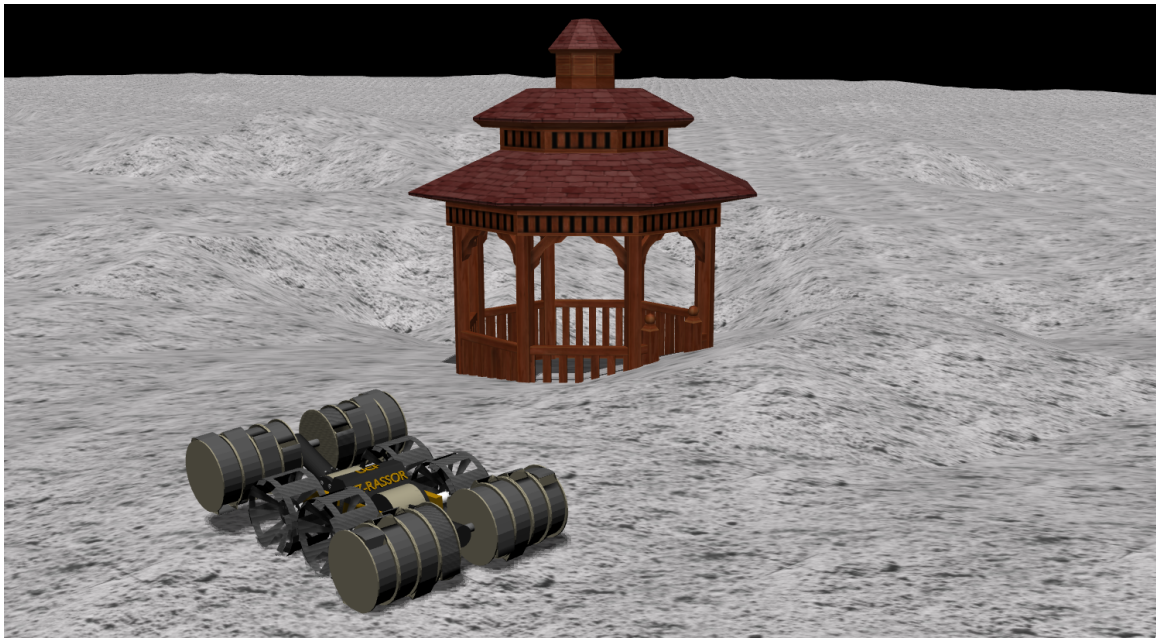


Figure 143 | Screenshot of the EZ-RASSOR operating on a Lunar environment in Gazebo

Since we were able to successfully add a Lunar world to our project, we decided to go an additional step and also add a Mars world. To accomplish this, we also used a DEM which was generated using Lidar. Since this DEM is not included in the installed Gazebo materials, we found an accurate and free to use map from https://astrogeology.usgs.gov/. The original file was 11GB large and was generated from the Mars Orbital Laser Altimeter instrument that was attached to a spacecraft that orbited Mars around 2006. With this being an enormous size, we extracted a 129m by 129m portion into a smaller DEM using a tool called **gdalwarp**. From there, we were able to add it to a new Mars launch file where we can spawn the EZ-RASSOR onto.

Figure 144 | Screenshot of the EZ-RASSOR operating on a Martian environment in Gazebo

Lastly, since the Mars environment introduced brand new materials that were not already existent in the project, we needed a way to access these materials using relative paths. However, Gazebo does not support relative paths when searching for files; it can only access paths that are defined in the Gazebo paths environment variable. To circumvent this, two lines were added to the package.xml file in the ezrassor_sim_gazebo package to append the current folder to those environment paths automatically when launching this gazebo package. (Please note, the DEM files may get corrupted if you attempt to clone the repository without git-lfs installed. Ensure that you have git-lfs installed before pulling the latest version of the repository which includes this file.) The lines we added were as follows:

```
<export>
   <gazebo_ros
       gazebo_plugin_path="${prefix}/lib"
       gazebo_model_path="${prefix}/.."/>
</export>
```

Figure 145 | Code snippet to show how we are able to share the DEM file with others

Designing the Simulation Models

Essential to the accurate simulation, visual representation, and development and testing processes of the EZ-RASSOR is the creation of the simulation models that are used in the simulation environments in Gazebo. As the team was not provided precise measurements or proportions for the robot, nor was the team provided with measurements for the actual RASSOR, the challenge lied with developing a physically accurate model for a yet-to-be-produced robot. For much of the development of the simulation model, the final design choices for the actual EZ-RASSOR were yet to be made, so we decided to mimic as closely as possible the RASSOR when creating the models.

The EZ-RASSOR model went through two main variations in this development process. The first was of a simpler design, to more readily allow for testing and integration with other modules of the EZ-RASSOR software in the simulation environment, without the testing of those modules having to wait until the final robot model was complete.

As Gazebo allows for separate components of a robot to be modelled and specified, then integrated together, the various parts of the robot were individually modelled. The components of the robot model include the wheels, the base link, the drum arms, and the drums. This architecture is the highest level of abstraction that allows for accurate physical simulation of the robot, as it is split by discrete moving parts.

Before developing the first test-ready model, a prototype model was made in order to solidify our understanding of the general, high level structure of the robot, test moving parts of a model, and learn interaction with the simulation software. Below is model 0, the prototype created for this purpose.

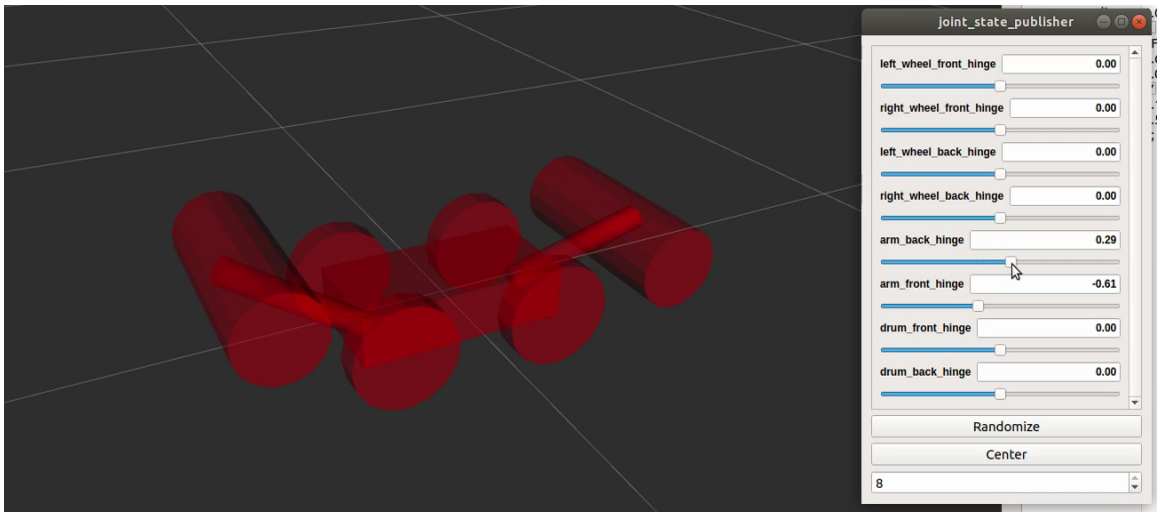


Figure 146 | The first model prototype

Following this phase was the development of the first model of the EZ-RASSOR for the simulation. As placeholders, stock wheel models and a simpler drum designs were used to have a working model in the simulation before the CDR.

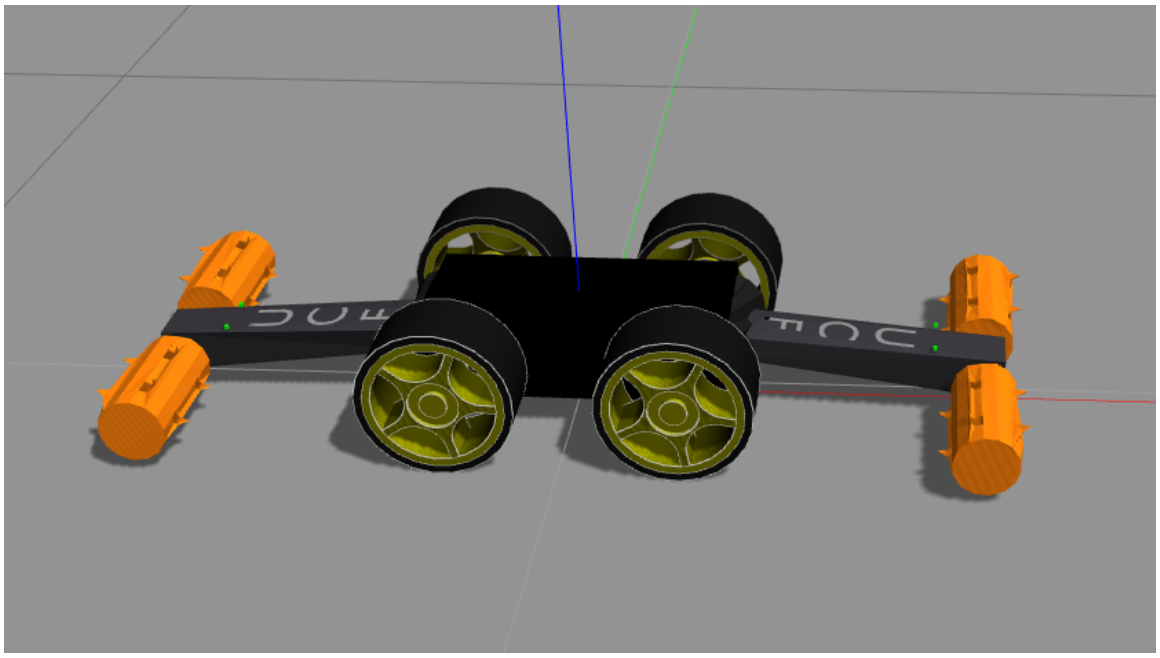


Figure 147 | Model featuring separate drums in Gazebo

While effective for testing and demonstrating high-level functionality, this model deviates greatly from the RASSOR, and with some discussion with the Swamp

Works team, steps to improve the model were made clear. This led to version 2 of the simulation model.

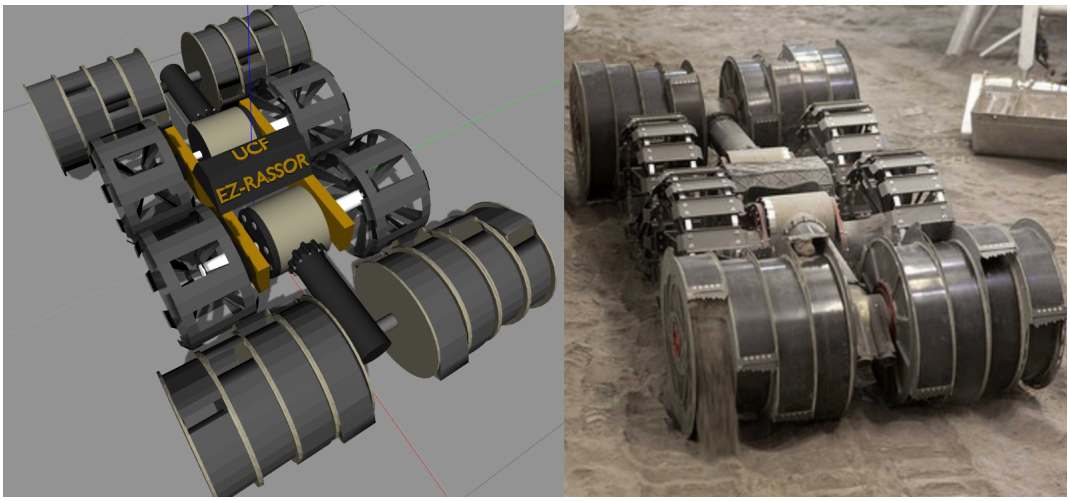


Figure 148 | Newest model (left), RASSOR (right)

Left is the final model of the EZ-RASSOR in Gazebo, with an image of the RASSOR to the right. Improvements from the previous model include a redesign of every component: new wheels, a new base link, new drum arms, and new drums. Additionally, the proportions in sizing between the various components have been made accurate, with the wheel width measuring to be one half of the a single drum width, and accurate drum arm lengths. The new base link also serves as an indicator for which side of the vehicle is facing upward, as it is possible for the vehicle to operate in the simulation in an upside-down configuration.

These models were created using Blender, an open-source 3D modelling program. After the objects were modelled, they were exported to COLLADA files (.dae), which can be used by Gazebo.

To modify the models of the EZ-RASSOR, open the object files located in the **blender/objects/** directory, make the modifications, and export the .DAE files to the **blender/meshes/** directory.

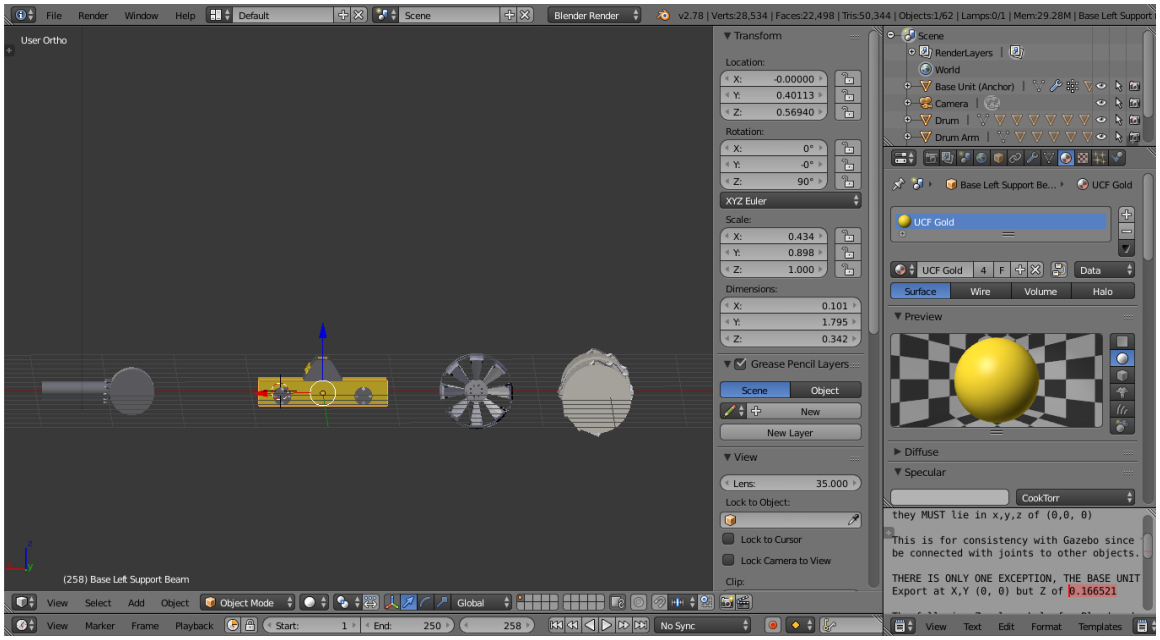


Figure 149 | EZ-RASSOR models in Blender

Removal of SLAM

The original idea was for the EZ-RASSOR to utilize Large Scale Direct Simultaneous Localization and Mapping (LSD-SLAM) and while this solution would be very beneficial for many autonomous robots, the extraterrestrial environment makes it difficult for the algorithm to produce any meaningful results. The low range of available colors and the bland nature of the environment mean that from a computer vision perspective the number of visible features is low. With this reduced complexity environment, the ability for the algorithm to accurately update the pose of the robot between frames is significantly hindered and the tracking is lost on many frames. After explaining these struggles, the team at Swamp Works advised us that something like a more simple stereo visual odometry would be sufficient for the EZ-RASSOR's needs and could potentially be much more successful in the lunar or martian environments.

Taking this advice, we worked to redesign our system to consider the use of only visual odometry and found that this provided a much simpler solution. The amount of computation required for the LSD-SLAM was very high and utilizing this visual odometry was much lighter and more efficient. The removal was also able to help us reduce quite a few dependencies of the project including the need

to exclusively use Ubuntu 16.04 and ROS-Kinetic. This will allow future teams to potentially port the code to ROS-Melodic or possibly even ROS2 and Ignite.

New Autonomous System Design

The original architecture of the EZ-RASSOR's autonomous system was fairly simplistic and while it was designed to accomplish all of the requirements, it wasn't designed well to fit in with the system in a modular way and also didn't fit with the ROS conventions, which is crucial for a successful piece of open source software. The new system is broken down into several components which include:

Component	Purpose
Autonomous Control	Main control node that handles publishing and subscribing to the EZ-RASSOR hardware controllers.
Auto Functions	Library of auto functions which autonomously perform specific actions.
Navigation Functions	Navigation helper functions library which provides calculations related to navigation.
Utility Functions	Utility function library which supplies conversions, movement of specific joints, and other miscellaneous utilities.
AI Objects	Library of classes which contain data and functions related to updating and accessing the world state and interacting with rospy.

The flow of information between these modules is defined in the figure below and shows that the main controller is the only component which talks directly to the hardware. The supporting libraries that contain the utility, navigation, and auto functions are also not official ROS nodes, but are instead simply python modules which have their functions imported into the nodes that actually publish.

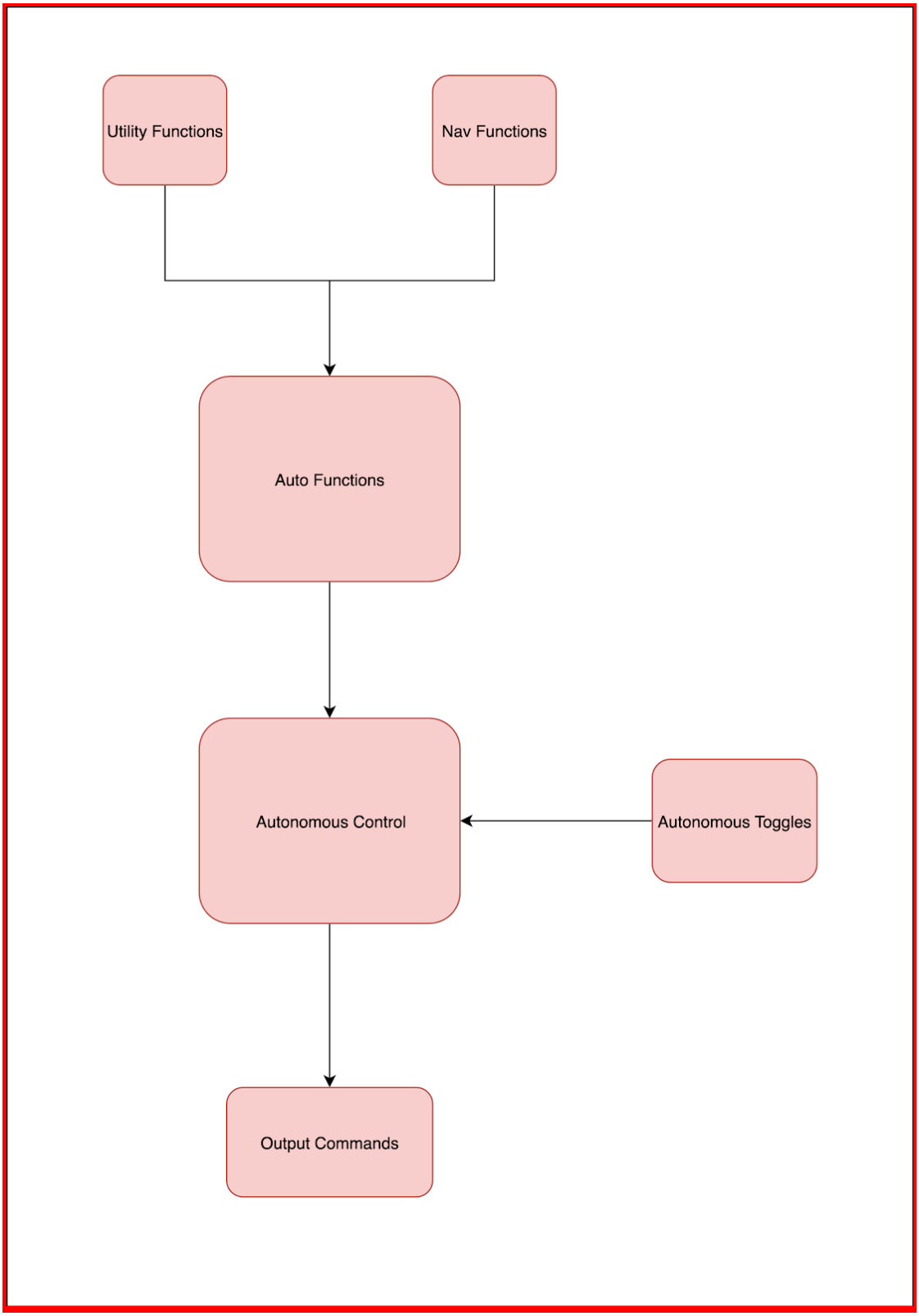


Figure 150 | Graph Describing information flow

Self-Right Autonomous Functionality Engineering

Self-right from side

The EZ-RASSOR is able to determine if it is on its side with the use of an onboard IMU. If the IMU is properly oriented on the robot, the Z-axis acceleration should increase significantly if the robot has managed to find itself on its side. In order to self-right from the side, the EZ-RASSOR takes advantage of the fact that its drums spread slightly wider than its wheels. Due to this fact, if the EZ-RASSOR is on its side, by just moving the arms past a 45 degree angle there is a high chance the robot will return to its standard upright position. To further increase the chance of a successful self-right from the side, we extend the arms straight out prior to raising them upwards. Extending the arms outwards gives us a consistent starting point for raising them upwards. Furthermore, extending them outwards also helps to solve some edge cases where the EZ-RASSOR has fallen against an object. Pushing outwards often pushes against the object and typically either gives the robot enough room to self-right, or ends up causing the robot to self-right before the sequence even reaches the arm raising portion. If at any point the robot self-rights before the sequence is fully completed, the sequence simply ends at the moment it is deemed to be upright.

Fig 151 | The difference in drum and wheels widths

Self-Right from upside down

The EZ-RASSOR will be able to make use of its onboard IMU to determine if it has been flipped upside down. When upside down the accelerometers Y-axis acceleration will read a negative value. The sequence will begin by the EZ-RASSOR straighten out the arms. We straighten out the arms for the same

reasons as for the self-right from side. The EZ-RASSOR will then begin to move its arms towards the ground and continue to do so until it is eventually standing on its drums. Once in this position, it will bring one of its wheels more towards the center of the robot, causing the EZ-RASSOR to tilt to one side. Eventually the it will tilt so much that it will fall into a position where it is standing on one set of wheels and one set of drums. From here the EZ-RASSOR will swing its free arm up and over, causing a weight shift once again, resulting in the EZ-RASSOR to fall into the correct, upright position.

Self-Balance Autonomous Functionality Engineering

One of the key features of the original RASSOR is its lightweightness. Usually for an excavator to be able to pick up large amounts of regolith it is required to be heavy. The RASSOR solves this issue by the circular digging drums, when the RASSOR begins to dig, the drums, try to balance out their torque in opposite directions, by moving in opposite directions. This was a problem in the simulator because actual regolith isn’t supported by Gazebo.

Thus it was required to be able to simulate a digging force being applied to the arms as its digging. Once a digging function is started then this self arm balance function is called. The way it works is there is a publisher always calculating a digging force for both arms, then once it does begin to dig, a digging force is then attempted to be equal on both ends. Thus we have two arms trying to always have similar digging forces. Right now for the demo, it is directly linked to the angle of the drums, so they are always just trying to match each other in that sense. The idea with how it was written was that this force can be replaced by either a function or some other publisher. The arms don't always have to be the same angle though in real life. It can be that one arm can be more heavier than the other yet they should still have the same force.

ROS Navigation Stack

The ROS Navigation Stack is a package that ships with ROS and simplifies the task of map generation, path planning, obstacle avoidance, and sensor fusion. All the system requires is the robot to provide four data sources:

- An odometry source

- Sensors to provide external environment information (Laser Scans, Image data, Sonar, etc.)
- Tf information.
- Base Controller

The odometry sources for the EZ-RASSOR are the IMU and the visual odometry provided by the robots vision system. These sources can be combined using the Navigation Stacks robot_pose_ekf package which handles sensor fusion using Kalman Filters. This information allows the Navigation stack to localize the robot for path planning and map generation.

At the time of this documents publishing, the only sensor available to the EZ-RASSOR which will provide external information is simply the stereo camera system. This provides data to the system which can be used for path planning, map generation, and obstacle avoidance.

The tf information is a tree-like data structure which allows the robot to calculate and transform data along a certain path in the tree. To illustrate this process, imagine a robot arm whose job it is to pick up an object. The robot is equipped with an image system which is disconnected from the physical arm. The vision system is capable of calculating the distance to the object in question to itself, however it must communicate this information to all off the necessary motors in the robot arm in which to make the proper pose adjustments to navigate to and grasp the object. In order for all of these movements to be executed properly, the data coming from the vision system needs to transform its values from the vision system's coordinates in 3D space to the coordinates of the base of the arm, then the joints, then finally to the grasping device. To implement this properly, every joint, sensor, and component of the robot must have their positions provided to the system to accurately execute this transform process. This process allows for the robotic system to behave more intelligently and accurately.

The base controller is the system provided by the robot which handles all of the movement commands for the robot. For the EZ-RASSOR this would be isolated to the wheels as the navigation stack has no need to control the arms. The Navigation Stack will publish movement messages for the base controller to execute when perform movement through its environment.

The ROS Navigation stack provides a powerful framework which the EZ-RASSOR can take full advantage. Although the robots specifications are somewhat unique, there is an opportunity for the EZ-RASSOR utilize some if not all of what the ROS Navigation Stack has to offer.

EZ-RASSOR Vision

The EZ-RASSOR's stereo camera is used to provide video data to the teleoperator of the robot as well as provide distance data from objects viewed in the video feed. It also allows for the collection of odometry information sourced from the camera's movement and the manipulation of the images due to such movement. This data allows the EZ-RASSOR to make more intelligent decisions and allows for greater autonomy.

The distance to objects viewed is calculated in the video feed by first calculating disparity values between corresponding pixels in the left and right images provided by the left and right cameras. This process is known as Stereo Matching.

Given two image planes, I_l and I_r, a target point P is viewed by each camera as light is received from the point by the left and right cameras optical sensors O_l and O_r. The points p_l and p_r are the corresponding points found in I_l and I_r that represent P. The relationship between O_l, O_r and P can be understood as three points existing on the same plane. This plane would then intersect I_l and I_r along the epipolar lines e_l and e_r in I_l and I_r respectfully. When trying to match p_l to p_r, both pixels must therefore exist along e_l and e_r respectfully. Once two pixels are matched, the respective horizontal positions x_l and x_r of p_l and p_r are then calculated and the disparity matrix D, is found by $d_{i,j} = x_{l_{i,j}} - x_{r_{i,j}}$, $d_{i,j} \in D$. ROS comes with a package called the stereo_image_proc which handles these calculations for us. It is capable of generating a point cloud, disparity maps, as well as image rectification. After attempting to implement various third party approaches, the ROS solution proved to be the fastest to implement and performed the best in simulation.

Once D is calculated, a distance matrix Z can be generated. To do this we first must know the baseline dista f, of the cameras. With this information we can then calculate Z.

$$z_{i,j} = \frac{fb}{d_{i,j}},\ z_{i,j} \in Z$$

We then utilize mean pooling to decrease the dimensions of Z as well as attempt to filter out noisy data generated by the cameras and stereo matching algorithm. The decrease in size is by a factor of eight transforming the shape of Z from 640 by 480 to 80 by 60. The relevance of this specific shape will be described later. Z is then split vertically into a left half, Z_l and right half, Z_r From here we begin assigning logic to the data we receive from both Z_l and Z_r. We define z_l and z_r to be the minimum values found in Z_l and Z_r respectfully. Should $z_l < z_r$, this would imply that there is an object approaching the robot from the left, and vise versa. This information allows the EZ-RASSOR to make decisions on which maneuvers it should make to avoid obstacles in its path.

Originally the camera system was to be mounted on the arms of the robot, between the drums. The Swamp Works team has since modified the placement to be on the underbelly of the robot just under the front arm joint. This placement presented issues for the team as the camera was then close to the ground and the disparity information and therefore the depth information where seeing the ground as an obstacle. This issue was remedied by adjusting the disparity generators min_disparity parameter to be a greater value so to ignore objects at that distance. A second issue that presented itself in relation to the disparity and depth calculations where the presence of hills in the environment. As the RASSOR approached a hill the camera system would register it as an obstacle and would therefore attempt to avoid it. While this may be a good approach for particularly steep hills, it proved problematic for traversing a relatively non-threatening environment. This problem was also remedied by the adjustment of the min-disparity value, however more testing will be needed to see if how the physical robot handles hilly environments.

In an attempt to take this data further, we can manipulate Z in such a way as to provide laser-scan data to the robot without the need for a planar laser to be provided. This is done by calculating the field of view of the cameras.

$$FOV_{rad} = 2\,tan^{-1}(\frac{w_p}{2f_p})$$

Where f_p is the focal length in pixels and w_p is the image width in pixels. We then take the center row of Z and treat the distance values, $z_{i,j}$, where $i = \frac{Image\ Height}{2}$, as laser-scan data sweeping across the FOV. Our test cameras in simulation have a FOV of 80 degrees which fit nicely with our 80 pixel wide distance matrix. Assigning each pixel as a degree in a artificial laser-scan we were able to achieve accurate laser-scan data generated in simulation. However, after further testing, we found some issues in our implementation of the depth map to laser scan conversion.

Given the nature of a laser scanner, the device assumes distance data to be read by emitting light while rotating around a central axis. This presented problems when when converting uniform data received by the depth map. For example, should the EZ-RASSOR be viewing a wall perpendicular to it, the expected result would be to view a flat surface. However, the surface would appear concave as the distances perceived as rotating around a central axis. To remedy this behaviour the team had to get creative. Assuming the distance data received to be uniform, we had to find a way to offset every distance reading to be that of a laser scan reading. As depicted in figure 152**,** the true representation of the data is the straight horizontal line of length α is curved due to the depth image to laser scan conversion**.**

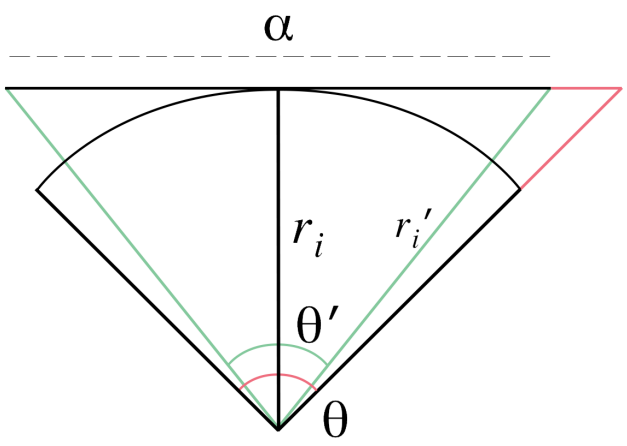


Figure 152 | Curvature of data

If measurement r_i is received, the value of that measurement is only accurate if its angle relative to the optical center is zero. As the angle diverges from zero, the measurements value requires more offset to be accurate.

Given a vector of distance readings, L, we compute the laser scan offset as follows:

$$\forall\, r_i \in L \mid L \in \mathbb{R}^n \text{, let } \alpha = \theta' r_i \text{, and } \theta' = 2tan^{-1}\left(\frac{FOV_{rad}}{2}\right)$$

$$\text{Then, } r_i' = \sqrt{\frac{1}{4}\alpha^2 + r_i^2}$$

Where r_i' is the original measurement plus the offset, α is the arc-length of the horizontal line, relative to the distance measured multiplied by θ', the angle of that measurement from the optical center. As the algorithm iterates through the values r_i in L and as i approaches $\frac{n}{2}$, where n is the dimension of L, we calculate $\theta'_{new} = \theta'_{old} - \frac{2w_p\pi}{FOV_{rad}\,180}$, as $i > \frac{n}{2}$, we calculate $\theta'_{new} = \theta'_{old} + \frac{2w_p\pi}{FOV_{rad}\,180}$, and when $i = \frac{n}{2}$, $\theta' = 0$. This approach allowed the artificial laser scan data to be offset as to accurately depict the objects in front of it. This conversion is not without its problems, as a real laser scanner would provide more accurate data. Now the question may arise as to the need for such redundancy in distance measuring, however, the conversion to accurate “laser scan” data would allow for the implementation of the ROS Navigation Stack and its GMapping package which explicitly calls for LaserScan messages. This depth map to laserscan conversion is still highly error prone due the need for pixel correspondence. Should the camera system view an object of uniform light and color, the stereo matching algorithm will be unable to find correspondence between pixels and no depth information can or will be calculated. This case could prove to be rare, however it requires a highly detailed simulation environment in which to test its accuracy. We found that objects in the simulation such as trees, were detected with much more accuracy as their texture mapping was more detailed. Conversely, objects such as blank walls or objects of a single color were invisible to the vision system due to this issue. In order to build a more robust obstacle

avoidance system additional hardware may be required and further research completed.

Visual Odometry

Much like the stereo matching problem stated above, the problem of visual odometry deals with the matching of corresponding pixels in an image. Where the two approaches differ is where the stereo matching algorithm matches pixels in a left and right image, visual odometry matches pixels from one timestep the next. As the robot moves the images received by the cameras will be manipulated in different ways based on the movement of the robot. This movement and change in position can provide the robot with information as to how it is moving.

We decided to utilize an open source solution to this process called Viso2. Viso2 is a premade package that is built for ROS to handle the heavy lifting behind visual odometry. It simply requires a specific transform configuration and subscriptions to the left_camera_rect and right_camera_rect topics, as well as the camera_info topics. This provides the package the data it needs to perform visual odometry. The package then publishes to the odom topic as well as broadcasts to the odom transform to allow higher order static reference points to understand how the robots pose is changing relative to it.

The performance of Viso2 in simulation was very resource intensive as our runtime in simulation slowed significantly. This is no doubt due the computationally expensive processing of image data from the image rectification, disparity map generation, point cloud generation, depth map computation, laser scan conversion, and visual odometry calculations, being processed on twin 640x480 pixel images being captured at 60 frames per second. Similar to the obstacle avoidance and depth map generation, this approach is heavily dependant on a detail rich, well lit environment to extract clean data. We found that the system behaved erratically in simulations lacking features in which to track as expected. It is our recommendation that due to the mission of the robot other means of odometry and robot localization be utilized either to replace or work with the visual odometry system. The addition of a GPU to the robot or perhaps parallel computing techniques utilizing a multicore processor would also increase the robots ability to handle the various resource intensive image processing.

Obstacle Avoidance

Whenever an obstacle is detected information regarding its location is sent to the main autonomous control which interprets this information and uses it to alter the current heading. If an obstacle is detected on the left the EZ-RASSOR will rotate to the right and if an obstacle is detected on the right the EZ-RASSOR will rotate to the left. It will continue to rotate until no obstacle is detected, maintaining a threshold for no detection signals to avoid exiting prematurely due to noise. Once the obstacle has no longer been detected consistently, the system then calculates a new intermediary target for avoiding the obstacle. This intermediary target is calculated by projecting the EZ-RASSOR's current position forward 1.5 times the distance away from the obstacle in the new heading after rotating the obstacle out of view. This system is fairly simple but performs well in environments that the EZ-RASSOR would typically operate which are relatively flat and have few large obstacles in close proximity.

Implementing Twist Messages for the EZ-RASSOR

Twist

Seeing how the open-source aspect of the EZ-RASSOR may lead the project into the hands of less experienced programmers to work on and play with, we plan to add support for Twist messages. Currently, the movement operations of the EZ-RASSOR are implemented using a bit string. Bit strings can appear to be cryptic and making changes to this design requires a knowledge of how they work. Adding support for Twist messages allows for a simpler way to handle movement controls. Additionally, Twist is considered to be the convention for ROS. Twist allows for programmers to worry less about how the physical hardware of the robot will interact with the code.

Another push for adding support came from the mechanical engineers at NASA. During the last two weeks of phase one, the mechanical engineers at NASA mentioned a familiarity they had with Twist. They noted that if Twist was supported, it could make their jobs a little easier by negating the need to learn the bit strings implementation.

A Twist message represents two vectors in free space: a vector for linear velocity and a vector for angular velocity. Each vector contains three float variables

corresponding to their x, y, and z values. The linear velocity refers to the robots velocity traveling in a straight, unchanging direction. The angular velocity represents how fast the robot rotates or revolves relative to another point in space overtime. Between these two vectors, you can get an accurate representation of any robots movements inside a 3D space.

The exact format of a Twist message can be seen by typing the following command into the terminal:

rosmsg show geometry_msgs/Twist

Which will result with this output:

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Figure 153 | Twist messages for the EZ-RASSOR

These vectors can be created using two equations involving both of joysticks on the controller. Both joysticks output a float of 1.0 when pressed all the way forward and -1.0 when held all the way back. The equations are as follows:

Linear velocity:

$$\frac{\text{Right Stick Value + Left Stick Vale}}{2}$$

Angular velocity:

$$\frac{\text{Right Stick Value - Left Stick Vale}}{2}$$

Adding support for Twist also comes with one last bonus. By supporting Twist we can take advantage of the ROS navigation stack. The ROS navigation stack is an open-source ROS package that does majority of the heavy lifting required by robotic path planning for you. By getting the navigation stack functioning, a considerable amount of strain can be taken off the AI sub-team. With the use of the navigation stack, the AI sub-team can put all their focus on improving swarm functionality.

To accommodate the need for Swamp Works to receive Twist messages on their hardware, we implemented a simple conversion node on our messaging topic to translate our bitstring into Twist messages. Since we received this additional requirements towards the end of our development phase, we decided to implement this simple fix. However, our entire architecture, from autonomy to gazebo control, will need to be rewritten to support Twist messages instead of bit strings if this is NASA's expectation.

The only downside to adding support for twist is that it will require a large portion of our system to be rewritten. We have already applied a crude implementation of twist in order to help the NASA mechanical engineers. However, this implementation is far from perfect and does not fully take advantage of what Twist messages can provide. For an indepth look into the system rewrite, check under the section "Future Work."

Development and Installation Scripts

This project features several scripts designed to make the development and deployment process simpler. One script automates most Catkin functionality and allows you to build and test the project without ever leaving the repository. Another fully installs the system, with all dependencies and other requirements for proper execution, in a fast and easy way. A third script is able to completely configure a Raspberry PI as a Wi-Fi hotspot.

Each of these scripts is intended to take a load off of developers and end users alike when they wish to change or use our software. The philosophy behind this project's scripts is simple: any process which 1) can be automated and 2) is annoying/unnecessary to reproduce as an end user or developer **should** be automated. End users should never have to worry about installing dependencies, configuring yaml files, or fixing file paths. Developers should not have to think

about setting up or organizing a Catkin workspace. Our scripts automate all of these processes and more, and the end result is software that is more *useable.*

The first script, *develop.sh*, has a general syntax that looks like this:

```
sh develop.sh <mode> [arguments]
```

The modes supported by the script are listed below:

- ***setup***
    - Set up a Catkin workspace in your home directory to develop and compile ROS nodes. This workspace is named *.workspace* by default.
- ***new <superpackage> <package> [dependencies…]***
    - Create a new ROS package in the *packages* folder, under the appropriate superpackage. If the superpackage doesn't exist it is created. All arguments after *packages* are passed to *catkin_create_pkg* (these arguments are usually dependencies of the package). The newly created package is then symlinked into your workspace's *src* folder. If you've never run *setup* ensure that you do that before trying to make a new package, otherwise you won't have a workspace to develop in!
- ***link [-e, --except <packages…> \| -o, --only <packages…>]***
    - Create a symlink from all packages in the *packages* directory to the *src* directory of your workspace (so that you can build and test your software, without having to copy it into the workspace each time). You should execute this mode after creating a new workspace, or if you've renamed/reorganized the packages in *packages*. If you've done this, you'll want to *purge* before running this command (see below), otherwise your *src* directory could contain broken symlinks to removed/renamed packages. Exclude specific packages with the *-e* or *--except* flag. Link specific packages with the *-o* or *--only* flag.
- ***purge***
    - Remove all symlinked packages from *src*.
- ***relink [-e, --except <packages…> \| -o, --only <packages…>]***

    - Purge all symlinked packages from *src*, and then link all packages in *packages*. Ignore specific packages with the *-e* or *--except* flag. Relink specific packages with the *-o* or *--only* flag.
- ***build***
    - Call *catkin_make* in your workspace.
- ***install***
    - Install all built packages into the install target in your workspace (via *catkin_make install*).
- ***kill***
    - Kill all running ROS nodes and *roscore*.

The installation script provided for this project installs ROS and all desired packages onto the user's system permanently. It installs ROS's base system using Ubuntu's *apt* package manager, then it creates a temporary Catkin workspace and links all of this project's packages into the workspace. The script resolves dependencies using *rosdep*, then it installs all packages in the workspace. This script is the cleanest way to install and use our software on a new system, and it is also what users expect. Installing software should be easy and brainless, because most normal people don't want to and aren't able to walk through a complicated installation process.

The final script sets up a Raspberry Pi system as a Wi-Fi hotspot. It is the simplest of the three scripts, and it mostly edits configuration files and installs some dependencies for the hotspot daemon, then it enables the daemon.

Designing a System Built on Modularity

Our team went to great lengths to create a heavily modular system that is robust, conventional, and efficient. The goal of modularity had loomed large in our minds since the start of the project. In the beginning, we discovered ROS and its complex, yet valuable, organizational structure.

ROS divides unrelated software into separate packages and these packages communicate using a (relatively) simple messaging protocol. Our project is easily divided into multiple packages: autonomous control is a package, the EZRC hardware control is a package, the controller server is a package, etc. ROS provides many benefits to our team and future developers via this separation.

First, if part of the system crashes during operation, the rest of the system will continue to operate nominally. Code in one package does not rely on code from any other package because it is thoroughly separated and divided with ROS. For an example, the EZ-RASSOR simulation will continue to run if the autonomous control module crashes; in fact if either the controller or gamepad interfaces is running the EZ-RASSOR can be repossessed by a manual operator without having to reboot or reconfigure the code in any way. If neither of these interfaces is running, one can simply be spun up and it will interact with the currently running processes as expected.

Secondly, the divided nature of ROS is quite harmonious with a swappable, configurable system. This means that different components in the system can be upgraded, replaced, or discarded and the system will still function. The system can even be configured by swapping or removing components. The premiere example of this is demonstrated with the topic switch module. The topic switch has two inputs (a primary and secondary topic), and a single output topic. Its primary use is to switch between manual control of the EZ-RASSOR and autonomous control when requested by the user… when the autonomous routine concludes, the user regains control. The topic switch reroutes the outputs of either the user's controller or the autonomous controller based on which mode it is in, and so it acts as a type of middleman. What if the user wants sole control of the EZ-RASSOR, or the EZ-RASSOR is in full-autonomy mode? Does the system have to launch topic switches to route communications between the input and the output? Of course not! The module design of the system means that components, like the topic switch, can just be ignored and not launched whenever it does not make sense to launch them.

Finally, ROS provides an entire build system for individual packages and metapackages. This build system allows every component in our system to be independently deployable. If someone wanted a piece of our software for her own system, she is free to take a single portion and install it separately.

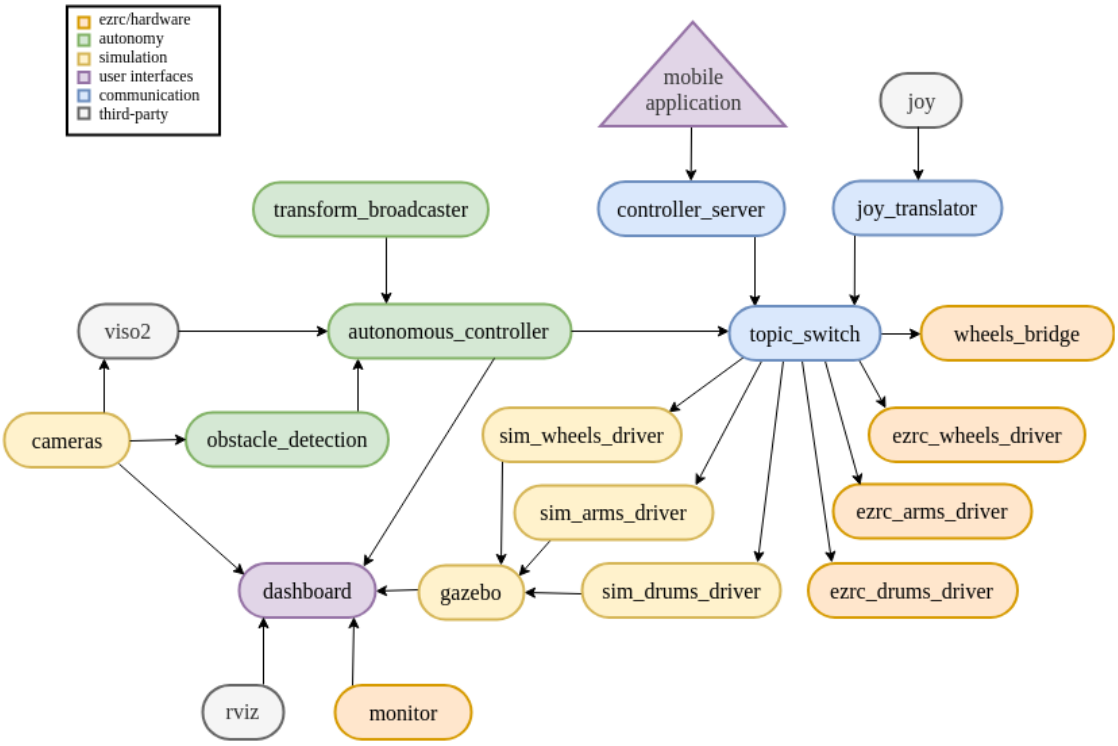


Figure 154 | Our current ROS graph

Figure 154 shows a recent revision of our ROS graph and is color coded to show major components. Each of these components is independent and can be modified, removed, or replaced without damaging the system.

ROS Integration Testing

ROS provides an excellent testing meta-framework called *rostest* that we use to test portions of our system. This meta-framework builds on top of *unittest*, the Python unit-testing framework. Our team wrote unit tests for the topic switch and controller server modules to confirm that data into the system is properly translated and modified before exiting the system. These unit tests are run by *rostest* at build time, and are designed to be comprehensive and cover normal operations. These tests are specifically designed **not** to test for edge cases and other dangerous decisions, but rather to that confirm the system operates as expected under normal conditions. These unit tests are combined and chained together to form full system integration tests.

One of the test suites within this repository tests the topic switch. It runs these four tests:

- Send data to the primary topic of the topic switch, then confirm that the data is returned through the output topic.
- Toggle the topic switch to listen to the secondary topic, then send data to the secondary topic of the topic switch. Confirm that the data is returned through the output topic.
- Send data to the primary topic of the topic switch, then randomly switch the topic switch to the secondary topic at some point. After this point send data to the secondary topic. Ensure that all expected data is returned through the output topic.
- Send data to both the primary topic and the secondary topic of the topic switch, and randomly switch the switch back and forth periodically. Ensure that all expected data is returned through the output topic.

Project Milestones

Fall 2018

12/21/18 - EZ-RASSOR Prototype Model Completed

- Test understanding of robot motion and interaction
- Establish understanding of links and joints in Gazebo

Spring 2019

01/11/19 - Installation scripts version 1

- Tested that everyone's development environment is working
- Demonstrated initialization of workspace

01/18/19 - EZ-RASSOR Initial model complete

- Working simulation model in Gazebo
- Connections between prototype model components

2/4/2019 - Critical Design Review

- Working complete demonstrations of software modules
- Gazebo simulations in presentable state

- EZRC and Gazebo-simulated EZ-RASSORs controllable via mobile app and gamepad

02/22/19 - Completion of User Driven Auto Functions

- The EZ-RASSOR should now be able to complete the following tasks:
    - AUTO_DIG
    - AUTO_DUMP
    - SELF_RIGHT
    - Z_CONFIG
    - AUTO_DRIVE (explore)
- These tasks may be called from command line
- Test startup scripts for EZ-RASSOR

03/1/19 - Obstacle Detection and Avoidance

- Simulation capable of detecting and avoid obstacles using obstacle detection
- Robot plans and re-orients to destination after obstacle has been bypassed

03/22/19 - Completion of Rover-Driven Auto Functions & Regression Test

- Re-test all previous functionality for any breaking changes
- The EX-RASSOR should now be able to execute the following rover initiated tasks intelligently
    - AUTO_DIG
    - AUTO_DUMP
    - SELF_RIGHT
    - Z_CONFIG
    - AUTO_DRIVE (explore)

04/12/19 - Integration Test at Swamp Works

- All Phase 1 requirements done and well-tested
- Hardware execution with NASA motors completed and verified
- Update simulation to modern Gazebo version used at Swamp Works

04/19/19 - **Delivery Of EZ-RASSOR Software**

Tasks Gantt Chart

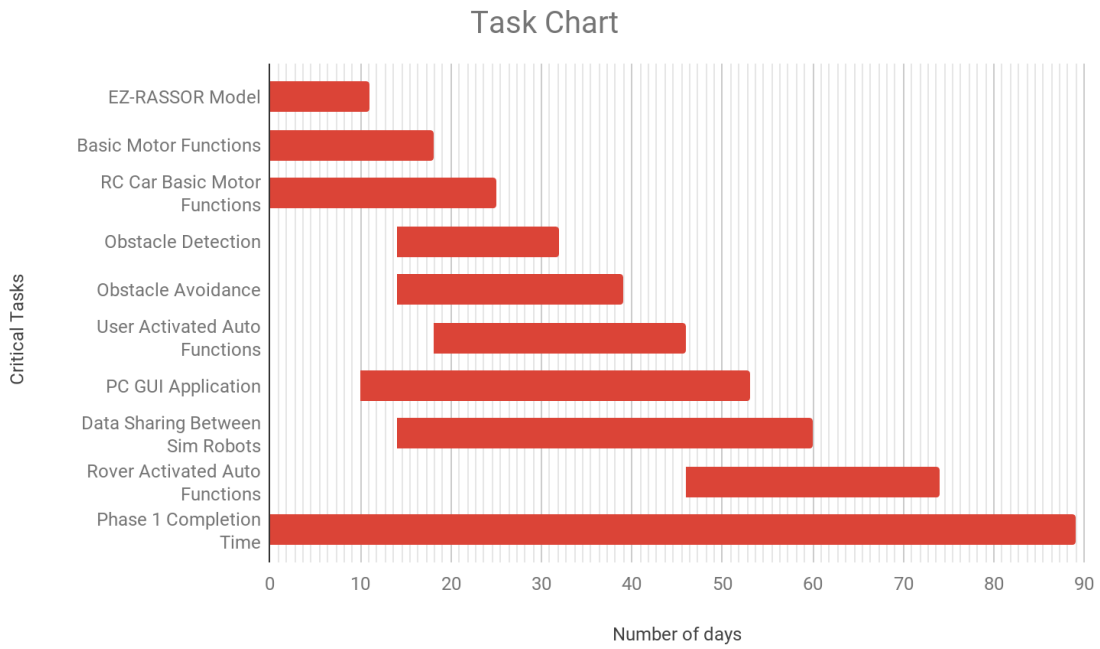


Figure 155

This chart shows the timeframe in which the team plans to complete all required components for phase one of the EZ RASSOR. Due to the EZ-RASSOR still being in its infant stage, numerous components we plan on developing rely on other components functioning properly. This is why we plan to focus on an Agile-like development cycle, where we consistently produce working prototypes at the end of each of our predetermined sprints. Our current plan is to have weekly sprints that each brings us closer to phase 1 completion. We believe the requirements of phase 1 are more than suitable to complete in the time allotted, so we also plan to immediately begin phase 2 as soon as 100% of phase 1 functionality is completed.

Source Control Procedures

This project will use git for version control and will be hosted on GitHub for public access. Git is the easiest and most commonly used tool for version control, and the entire team is already comfortable with it. It is also feature-rich and integrates perfectly with GitHub. Hosting the project on GitHub will allow other users to easily access all of the source code for this project. GitHub also provides many ways to interact with the community and improve the software over time. Ideally, as individuals work with the EZ-RASSOR and make their own modifications to the code, they will be able to fork and merge to our team's repository so that this project can live and improve beyond the scope of this year.

Note: Our GitHub repository can be found here. This is a private repository. Only authorized members can view the repository. As of the time of writing, the only authorized members are Mike Conroy (sponsor), any UCF students under the Florida Space Institute Organization, and Professor Heinrich.

This project's version control processes are as follows:

- The "master" branch is the main branch of the project and will always feature a stable build. This branch will feature the "release version" of the code for the length of that version release. For example, if the repository has been released as version 1.337, then for as long as 1.337 is the current release of the repository, master will not change.

- The main "working" branch of this project will be the "dev" branch. The dev branch will always be buildable and functional, but it may not always be perfectly stable/bug free. The dev branch will contain all of the most recent work on the project that has made it out of individual feature-branches.

- Each new feature and all actual development work should be done in feature branches or feature forks. These branches can be as stable or unstable as necessary and may not even be buildable in their current states. Each feature branch should be scoped to cover only one or a small

handful of features. New features should be created in new branches. Upon completion of a feature, the feature branch must create a pull request from dev and then merge into dev. All merges into dev must be functional. Any merges to dev that break dev will be reverted.

- Merges to dev do not need to be reviewed or signed by anyone.

- Merging to master will only occur upon a new iteration/release of the software. All merges to master should only occur after approved by the team and reviewed by Tiger Sachse (the official merge guardian).

The versioning scheme will follow the format vX.Y.Z where X represents the major version of the software. This version only changes for massive software overhauls. Y is the minor version of the software and will increment frequently. A new Y version typically indicates a new collection of features/improvements on old features. The Z version is the micro version of the software and will increment for emergency hotfixes/last minute critical features. The alpha version of this software will be in the range [v0.0.1, v0.1.0). The beta version of this software will be in the range [v0.1.0, v1.0.0). The release version of this software will be in the range [v1.0.0, infinity).

Performance Analysis

	Personal PC: 12GB RAM 2.7	Robot: 4 GB RAM 1.83 GHZ
Memory %	17.6%	52.8%
CPU %	27%	40.0%

While the development machines used for this project all had discrete graphics cards and multi-core processors, we intended on producing efficient software that would be able to run on the final EZ-RASSOR hardware. To this end, we ran a performance analysis on the full software and simulation environment running with all of the autonomous functions running in the background. We also did this analysis on the Intel Atom board that the EZ-RASSOR team will be using. (The latter analysis was done on the final week of development when we testing motor functionality).

As you can see, even when processing all of our nodes and functionality on the robot, the system is only operating at no more than 40% load. This information is very encouraging as it also means that future teams can continue to add functionality without exceeding the resources on the robot. Lastly, to gather this performance data, the team used the performance statistics available on our GUI application.

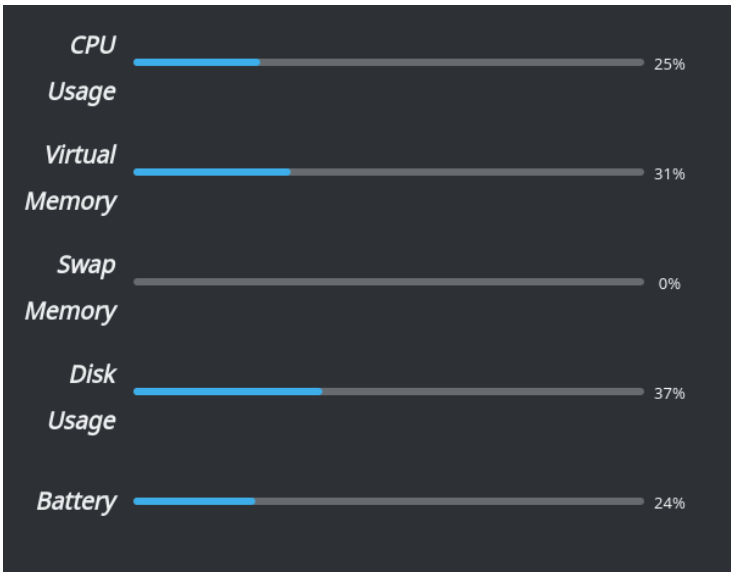


Figure 156

Open Source Conversion Strategy

This project is free and open-source under the MIT license. Anyone can fork this repository and submit a pull request, however all pull requests are subject to review and approval by this project's authors. All merged code becomes part of

this project, and thus is subject to the same license as the rest of the code in this project. Any code in this repository may be deleted, modified, or rewritten at any time. **Ultimately the authors of this project, the Florida Space Institute, and NASA have final control over this project's code.** By submitting a pull request, you voluntarily surrender all the rights you possess over your code to the Florida Space Institute, NASA, and the authors of this project (with the good-faith expectation that your contributions will be adequately credited to you). New authors may be named periodically, depending on contribution size and project demands.

Future Work

The following section describes work efforts that we were unable to accomplish before the conclusion of this project. Additionally, none of these topics were part of our Main Requirements and were always listed as stretch goals. Since future UCF development teams will iterate on this project, we believe it is useful to provide descriptions for what needs to be done to accomplish the following tasks.

DON SEE Integration

Our project sponsor, Mike Conroy, is also the sponsor of the annual NASA Simulation Exploration Experience (SEE) project. This project is a massive effort which involves several universities including UCF to improve upon NASA's proprietary simulation environment, the Distributed Observer Network (DON). This simulation environment simply tracks the movement of positional changes of robots in the environment to show where objects move to over time. It is less complex than Gazebo since elements such as angular velocity cannot be utilized, but it nonetheless provides accurate data to NASA on the change of objects over time. For integration with this project, a new ROS node will have to be developed to track the robot's position in real-time and then be able to send that data to the DON. This positional change will simply be (x, y, z) coordinates.

Added Support for Twist Messages

As noted in the Implementation Phase section, the UCF development team added support to export our bit string messages to Twist messages so that our software could communicate with the motors that Swamp Works produced.

Going forward, the entire EZ-RASSOR software infrastructure will need to be rewritten to support using Twist messages instead of bit strings.

Thankfully, due to the modularity on which we developed this project, very few topics will need to be changed. One major topic that will need to be modified is the movement_toggles topic where the bitstrings get translated into velocity commands for the various motors on the EZ-RASSOR simulation.

Swarm Control

From the beginning of this project, it was known that the RASSOR and therefore the EZ-RASSOR will need to implement some form of swarming behaviour in order for the robots mission to be successful. Since the EZ-RASSOR is aimed to be an open source, multi-year, and possibly multi-university project there is plenty of room for the development of some very exciting solutions. The team spent countless hours considering some of the issues future EZ-RASSOR's and RASSOR's may face with this task. Should the swarm be decentralized? Should there be a hive-mind implementation? How will the robots communicate with one another? Should they communicate directly or indirectly. There are pros and cons to each approach and when faced with the restrictions of the RASSOR's circumstances, the challenge becomes all the more present.

The reader should remain aware that though the EZ-RASSOR may never actually go to Mars or the Moon, the RASSOR is designed to do so. Therefore, it is the mission of the EZ-RASSOR to find solutions to the problems the RASSOR may face in these treacherous environments. While simultaneously facing the challenge of size and mass restrictions, cost, computational power, battery life, and environmental issues such as gravity differences, outdoor environments, extremely lacking infrastructure, and variable terrain types. Solutions that may prove to work in a lab environment must be tested outdoors and in environments similar to that of the Moon and Mars.

Whatever the approach, the team has designed the EZ-RASSOR framework such that multiple agents should have no problem getting started. The addition of artificial intelligence, multi-agent logic, distributive computing algorithms, and other complex layers should be easily implemented into our design. Should the individual robots remain independent worker bees making their own choices given the state of their environment, or have instructions explicitly given to them

from a mission overseer, the autonomy layers and controller layers will able to receive such processes and commands and execute them properly.

Although the team was unable to achieve this particular stretch goal, the future is bright with possibilities as this project's development continues with future teams and the continued contributions from current team members moving forward.

GPS Denied Navigation

With likely a large presence on planetary bodies with well-mapped surfaces such as the Moon and Mars, keeping track of several quick-moving autonomous robots is a task not reasonably suited for full-time dedication from a human team, thus the many measures taken in this project to form a solid foundation of location tracking and self-guidance. Another measure to be taken to further fortify the accuracy of the locations of the robots is the GPS-Denied Navigation method, which will rely not on data gathered solely by the robots themselves, but by outside sources, namely orbiters that have generated surface maps of different sorts.

From a high level, the process is rather simple: compare orbiter data with robot data, find matches, and conclude a robot location, much like checking a map and comparing it to your surroundings to find your location. This process will likely include the use of elevation maps and optical imagery more than any other kind of data

Conclusion

Over the past eight months working on this project, the team has made many great achievements but it did not come easy. We ran into our fair share of complications, and being the first ten person senior design team from UCF, we had a lot of pressure weighing on out shoulders. Luckily, we all couldn't have asked for a better group to work on this project. Each member brought a unique skill set that allowed us to rapidly develop and resolve issues quickly and efficiently. Although, there still was a multitude of hurdles to overcome. No one on the team had an extensive background in robotics. The ROS framework is notorious for its steep learning curve, but with dedication and the support of the team, all members managed to overcome this hurdle. The team was also given very little documentation or requirements to follow, and we did not have a full understanding of where to begin. Initially, NASA didn't know where this project was exactly going, but as development ensued, a clearer map to completion was drawn for us to follow. With all of the different technologies incorporated in this project, there comes a multitude of dependencies. Keeping track of them all was a daunting task, but became streamlined as our organizational structure became well-defined over time. From bridging the gap between React Native and ROS, testing visual-based algorithms in a low-poly simulation environment, to learning new requirements a week before delivery, the team managed to overcome all obstacles and triumph.

The team has gone above and beyond our expected requirements to ensure delivery of a superb product. We were given nothing to begin with besides three documents explaining what the RASSOR is, what it does, and details pertaining to the engineering of the robot rather than the software behind it. With as little as we had to begin with, the team developed an organizational plan and hit the ground running.

The Simulation team, led by Ron Marrero, laid the foundation of this project by creating a fully functional EZ-RASSOR model and several environments to test its capabilities. The model was completely custom-built from the ground up to imitate the likeness of the real RASSOR. The environments include a lunar and

martian environment that use real topographical maps from their corresponding planets.

The ROS team, led by Harrison black, developed a comprehensive hierarchy of topics and nodes to ensure the development of a system that is easily expandable and modular. This team developed the bit string used for the EZ-RASSOR's efficient mobility algorithms. This team also created the EZ-RC, a fully functional RC car that we were able to install our ROS graph on and display our work in a real-world environment. To ensure the viability of future development, the team also developed an installation suite so future developers may be able to quickly compile our work and add to the ever growing EZ-RASSOR project.

The Interface team, led by Christopher Taliaferro, developed the tools necessary for a non-technical audience to become a part of the EZ-RASSOR project. The team developed an iOS/Android application to control the robot in either a real-world or simulation environment. The application allows the user to fully control the robot, initiate AI functionality, and switch control between different EZ-RASSORs at a moments notice. This team also developed a desktop application to fully monitor the robot's perceived data and computational burden. This application includes visualization of the generated point cloud, disparity map, robot pose, internal measurement unit, and camera feeds.

The AI team, led by Cameron Taylor, developed a comprehensive autonomous system that brings the robot to life. The EZ-RASSOR is able to process visual data from its stereo cameras to map its surroundings and determine its world position using visual odometry. Using this visual data, the robot also creates a disparity map to analyze and detect obstacles in its path. The EZ-RASSOR can then avoid these obstacles in a seamless manner. Using the internal measurement unit, the robot is able to determine when it has been flipped over, and correct itself by flipping back over. These features combined make for a very intelligent robot that is able to maneuver and find its way out of most situations when performing its daily mission.

NASA's dedication to push mankind into the future has inspired us all. The RASSOR is a pinnacle of innovation and engineering that the team is grateful to be a part of. The EZ-RASSOR has the intention to make a lasting impact on future generations, exposing the world to cutting-edge technology in an easy to

digest form. This team wishes the best for all future development of the project, and hopes that the EZ-RASSOR will be able to complete its most important mission of all, educating the youth of the future.

Contacts

Michael Conroy - Florida Space Institute – Project Sponsor
Mike.conroy@ucf.edu

Jason Schuler - Robotics Engineer, Spaceport Systems Branch (UB-R1)
jason.m.schuler@nasa.gov

Kurt Leucht - Robotics Engineer, Spaceport Systems Branch (UB-R1)
kurt.leucht@nasa.gov

Christopher Comstock - Robotics Engineer, Spaceport Systems Branch (UB-R1)
christopher.j.comstock@nasa.gov

Tamara Belk -Administrative Officer of Exploration Research and Tech Programs
tamara.l.belk@nasa.gov

Robert Mueller - Senior Technologist of Exploration Research and Tech Programs
rob.mueller@nasa.gov

Project Terms and Definitions

Base

A stand-alone device that will be the central point of reference for all deployed EZ-RASSORs. Needed for relative navigation as well as tracking distance traversed.

Development Team

The 10 developers from UCF that are responsible for the completion of the Phase 1 requirements and the overall success of the project.

E-RASSOR

Stands for Educational Regolith Advanced Surface Systems Operations Robot and refers to the original design idea for this project. This naming has been deprecated in lieu of the EZ-RASSSOR.

EZ-RASSOR

Stands for Easy-RASSOR and refers to the current design idea for this project. This naming reflects the fact that the project is no longer geared towards education but instead to create a demo version of the production RASSOR.

GPS-Denied Navigation

Refers to the ability for track the EZ-RASSOR's position in environments where GPS navigation is not possible, such as traversal on other planets where a real-time GPS is not available.

IMU

Stands for Inertial Measurement Unit and is a device used to measure the specific force and angular rate of the object that it is attached to.

RASSOR

Stands for Regolith Advanced Surface Systems Operations Robot and refers to the production Swamp Works project. The project described in this paper is based on the RASSOR.

Regolith

Defined as the layer of unconsolidated rocky material covering bedrock. This is the target soil for the operations of the RASSOR and EZ-RASSOR and is where nutrients will be extracted.

ROS

Stands for Robot Operating System and is a crucial piece in the programming of robots. It allows for operations such as low-level device control and communication between robots.

Rover Drums

Two rotating drums that will be present on both sides of the EZ-RASSOR, just as they are in the RASSOR. They are responsible for regolith operations which include digging and storing regolith inside of the drums. They are also responsible for movement as they will be used to assist with the rover's balance and ability to overcome obstacles.

Self-Right

Refers to the function of the EZ-RASSOR to reorient itself after it has fallen on its side.

Sensors

Refers to stereoscopic cameras present on the EZ-RASSOR that will be responsible for object detection and assisting in movement functions.

Simulation Software

Refers to software that will be used to test the full functionality of the EZ-RASSOR, complete with differing environments to simulate real-world conditions.

SLAM

Stands for Simultaneous Localization and Mapping and refers to the task of constructing/updating an environment map while also updating the rover's position within it.

Swarm Robotics

Refers to the collection behavior of multiple robots with the environments that they are deployed to. This is usually a broad category but is specifically defined in this document.

References

---

[1] McLurkin, J. (2008). *Measuring the Accuracy of Distributed Algorithms on Multi-Robot Systems with Dynamic Network Topologies.* Accessed 15 October 2018.

[2] Jonathon, Hui, "Real-time Object Detection with YOLO, YOLOv2 and now ... - Medium." 18 Mar. 2018, https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088. Accessed 2 Nov. 2018.

[3] Williams, S., & Howard, A. M. (2008, May). A single camera terrain slope estimation technique for natural arctic environments. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on* (pp. 2729-2734). IEEE.

[4] Mur-Artal, Raul & Montiel, J & Tardos, Juan. (2015).. IEEE Transactions on Robotics. 3.

[5] Raul Mur Artal. Youtube. 21 October 2016. Web. 16 October 2018.

[6] Engel, J., Stückler, J., & Cremers, D. (2015, September). Large-scale direct SLAM with stereo cameras. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on* (pp. 1935-1942). IEEE.

[7] “ROS/rqt_reconfigure - ROS Wiki". Wiki.ros.org/rqt_reconfigure. 31 January 2018. Accessed 16 October 2018.

[8] “MoveIt! Quickstart in RViz”. Moveit.ros.org. 15 August 2018. Accessed 20 November 2018.

[9] Menglong Zhu, Kosta Derpanis, Yinfei Yang, Samarth Brahmbhatt, Mabel Zhang, Cody Phillips and Kostas Daniilidis. (2014). Single Image 3D Object Detection and Pose Estimation for Graspin. Accessed 22 November 2018.

[10] “ROS/rqt - ROS Wiki". Wiki.ros.org/rqt. 28 August 28 2016. Accessed 15 October 2018.

[11] “PySide vs PyQt”. Data source: Google Trends (https://www.google.com/trends). Accessed 25 November 2018

[12] Oliver Pink. Visual map matching and localization using a global feature map. In *CVPR Workshop on Visual Localization for Mobile Platforms*, 2008.

[13] Wilde, E., Pautasso, C. & Alarcón, R. (2014). *REST : advanced research topics and practical applications*. New York: Springer.

[14] Waltz, F. M., & Miller, J. W. (1998, October). Efficient algorithm for Gaussian blur using finite-state machines. In *Machine Vision Systems for Inspection and Metrology VII* (Vol. 3521, pp. 334-342). International Society for Optics and Photonics.

[15] Dec. 2018.http://www.cs.jhu.edu/~misha/ReadingSeminar/Papers/Tomasi98.pdf

[16] Douillard, B., Underwood, J., Kuntz, N., Vlaskine, V., Quadros, A., Morton, P., & Frenkel, A. (2011, May). On the segmentation of 3D LIDAR point clouds. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*(pp. 2798-2805). IEEE.

[17] McLurkin, J. (2008). *Measuring the Accuracy of Distributed Algorithms on Multi-Robot Systems with Dynamic Network Topologies.* Accessed 15 October 2018.

[18] Lou, Y., Ren, J., Lin, M., Pang, J., Sun, W., Li, H., Lin, L., (2018). *Single View Stereo Matching*. Accessed 15 October 2018.

[19] Cox, R., Ebert, T., Mueller, R., Nick, J., Schuler, J., Smith, J. (2013). *“Regolith Advanced Surface Systems Operation Robot (RASSOR).”* Accessed 13 September 2018.

[20] Dokos, A., Gelino, N., Leucht, K., Mueller, R., Nick, A., Schuler, J., Smith, J., Townsend, I. (June 2017). "*Regolith Advanced Surface Systems Operations Robot (RASSOR) 2.0".* Accessed 13 September 2018.

[21] ROS/rqt - ROS Wiki". Wiki.ros.org/rqt. 28 May 5 2013. Accessed 7 November 2018.

[22] OSFR. Vimeo. 28 January 2013. Web. 18 October 2018.

[23] Shi, G. & Keqiu. (2017). Signal interference in WiFi and ZigBee networks. Cham, Switzerland: Springer.

[24] Labiod, H., Afifi, H. & Santis, C. (2007). Wi-Fi, Bluetooth, Zigbee and WiMAX. Dordrecht London: Springer.

[25] Bhat, S. (2018). Practical Docker with Python : build, release and distribute your Python app with Docker. United States: Apress.

[26] "Zuckerberg's Biggest Mistake? 'Betting on HTML5'". Mashable. Retrieved 7 April 2018.

[27] Alex Khenkin, "Sonic Nirvana: MEMS Accelerometers as Acoustic Pickups in Musical Instruments" https://www.sensorsmag.com/embedded/sonic-nirvana-mems-accelerometers-as-acoustic-pickups-musical-instruments. Accessed 27 Nov. 2018

[28] Suneeta Godbole, "Average roll, pitch, and yaw angles." https://www.researchgate.net/figure/Average-roll-pitch-and-yaw-angles_fig2_262055313. Accessed 27 Nov. 2018.

[29] "WiFi and non-WiFi Interference Examples". Metageek. https://support.metageek.com/hc/en-us/articles/200628894-WiFi-and-non-WiFi-Interference-Examples. Retrieved 10 November 2018.

[30] JakobEngel, "LSD-SLAM - GitHub." https://github.com/tum-vision/lsd_slam. Accessed 31 Oct. 2018

[31] Jeroen Zijlmans,"LSD-Slam: Implementation on my Ubuntu 16.04 with ROS ... - Medium." 14 Aug. 2017, https://medium.com/@j.zijlmans/lsd-slam-bc80ae01a1bb. Accessed 21 Nov. 2018.

[32] Redmon, Joseph and Farhadi, Ali, "YOLO: Real-Time Object Detection ...." https://pjreddie.com/darknet/yolo/. Accessed 14 Nov. 2018.

[33] https://moon.nasa.gov/resources/87/high-resolution-topographic-map-of-the-moon/

[34] Lu, Q., Moses, M., Hecker, J., (2014). *A Scalable and Adaptable Multiple-Place Foraging Algorithm for Ant-Inspired Robot Swarms.* Accessed 3 November 2018.