CSU: SACRAMENTO

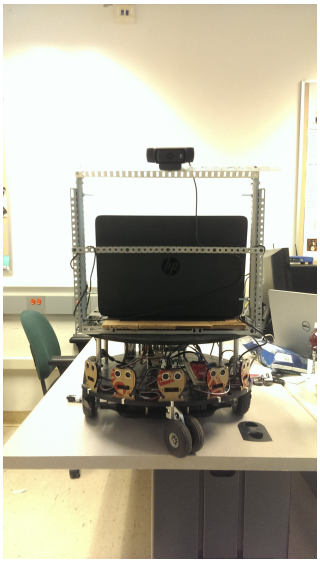
SENIOR DESIGN PROJECT: SLAM-BOT

End of Project Documentation

Authors:
Chris LANEY
Thomas HAYWARD
Curtis MUNTZ
Francisco ROJAS

Instructor:
Professor TATRO

In Collaboration With:
FMC TECHNOLOGIES SCHILLING ROBOTICS



Senior Design Team One is completing a project which will attempt to mitigate the risk of injury to people who work within hazardous environments. Simultaneous Localization and Mapping better known as *SLAM* is the method that team one chose to accomplish this. This paper explains the design process for the project from its inception to the completion of the Deployable Prototype. It will describe how the project was chosen, a work breakdown structure, and the technical details of the subsystems required to make our SLAM-bot work. Documentation is provided that will enable sufficiently prepared teams to reproduce the project.

Keywords: *Extended Kalman Filter, Machine Vision, Path Planning, Robotics, SLAM, Visual Odometry*

CONTENTS

List of Figures

List of Tables

I. Design Overview

A. Introduction

People who work within hazardous environments inherently put themselves at risk for injury. Rescuers entering a collapsed mine, a structural engineer evaluating a building after an earthquake, or a doctor providing treatment to an infectious patient are all occupations that carry an inherent risk of injury to the person performing their task. The risk is great due to the fact that the person needs to physically enter or operate within the hazardous environment. One solution to this problem is to have an autonomous robotic platform perform these same tasks thus, removing the individual from the risky environment altogether. However in order to have a truly autonomous robotic platform, the platform must be able to discern its location at any given time.

In the field of robotics this unique problem is better known as the, *Where am I?*, problem. This is a fundamental robotics issue. One method to solve the *Where am I* question is to implement Simultaneous Localization and Mapping, better known as SLAM. SLAM allows the robotic platform to take in sensory data from its environment using cameras, IR sensors, Ping Sensors, gyroscopes or accelerometers, and then fuse this data with wheel encoder data to create an accurate map of its surroundings. Ideally, a SLAM algorithm can be modified to adapt to any robot platform and be used to maintain a fixed position or localization in an unknown environment. It is this application of SLAM that interested FMC Technologies Schillings Robotics to offer this project challenge to students at CSUS. Our team accepted the challenge of creating a SLAM robot that could perform the specific criteria set by FMC Technologies Schillings Robotics and they offered to fully fund the project.

The objective of this project is to focus on a way to mitigate the risk of injury to people who work within hazardous environments. It is our hope that our efforts will have a positive effect on society as a whole. We intend to accomplish this by developing a SLAM algorithm that could be applied to a robotic platform. By the end of this two semester project our robotic platform will be able to be placed into an unknown environment for which it has no prior knowledge, and automatically explore the room and provide a 2 dimensional map birds eye view of that room along with specific object of interest within it. Once the room has been mapped it will be capable of navigating to a specific object within the room. In order to accomplish this we have identified key features that must be completed for our system to work as envisioned. These features are what we based our design idea contract on, each feature will be covered in depth later on in this report.

1) Robot Communication
2) Kinematic Model
3) Visual Display
4) Path Planning
5) Visual Odometry Interface
6) Collision Avoidance
7) Filtering
8) Gyroscope and Accelerometer

In the fall semester, we focused on completing the essential aspects of our feature set. Specifically items 1, 2, 3, 4, and 6. Item 1, Robot Communication, was integral to the completion of this project. If we were unable to communicate with our robotic platform, then we would have been unable to move forward with our project. Fortunately, we were able to complete this feature early on. Item 2, the Kinematic Model, was very important to the complete this project because it is how we calculate the robots position in the global environment. Initially, we started the Fall semester utilizing a differential drive kinematic model based on wheel encoder data. However, towards the end of the semester, we learned that certain assumptions we made were incorrect and we adjusted our kinematic model. Item 3, Visual Display was first deemed to have less importance than the other items since at the time the only use of the visual display would be towards the end of the project for the display of the 2-D map. However, we quickly learned that we desperately needed a method to display the data that we were gathering. So we quickly created the visual display to aid us in our debugging. Item 4, Path Planning is crucial since a key aspect of this project to deliver an autonomous platform, which cannot be completed without the use of a path planning algorithm. The Fall semester saw the implementation of a quick exploration algorithm using a wall hugger method. Item 6, Collision Avoidance is an integral part of our system, it is needed to successfully avoid obstacles within the unknown environment. The Fall semester saw a prototype implementation of this feature.

The spring semester oversaw the implementation of path planning, facial recognition, and refined visual odometry filtering interface. After these features were initially implemented, we spent our time fine tuning the system in full feature system testing, refining the usability of our platform as we continued to test, and fixing issues that we encountered along the way. Now, all of our features have been implemented and integrated into a working system.

B. Project History

Near the end of the Spring 2014 semester FMC Technologies Schillings Robotics approached Sacramento State University to look for a team to take on a project for them during upcoming senior design year. They were looking for a team that would be interested in designing and building a visual based SLAM robot. It was this pitch that interested and rallied four young students to come together as *Team One*. After some paper work and the signing of several non disclosure agreements. Team one was officially sponsored by FMC Technologies Schillings Robotics. Team one, composed of *Chris Laney, Thomas Hayward, Curtis Muntz, & Francisco Rojas*. Each student brings a unique background to this project. A brief summary of each student is provided in the section below.

1) Team Member Summary: All group members are students studying at CSU Sacramento, working toward degrees in Electrical and Electronics Engineering. The teams resumes are attached in the Appendix.

Curtis Muntz- Curtis has focused most of his classes on control theory, and has a solid background in machine vision. He will be focusing his efforts on the machine vision problems in this project. Curtis has a background in Linux systems administration, which will be helpful in maintaining a software platform on top of Linux.

Francisco Rojas - Francisco has a strong interest in Digital Signal Processing, and as such he will be focusing his efforts on the filtering aspects of this project. He is also interested in working with the various sensors of the project.

Thomas Hayward - Thomas has a strong background in troubleshooting and debugging hardware-software interfaces. He has experience in developing software designed to automate testing of complex systems such as radar. He has focused most of his out of classroom education on the implementation of software to satisfy embedded system design requirements.

Chris Laney - Chris has an extensive background in hardware applications, communication systems and career experience in the Defense industry working with a multitude of different systems. He is new to programming and has had to learn C and C++ programming languages as well as learn how to program microcontrollers such as the Atmega 328, Propeller, Microchip and Altera FPGAs all within the last year.

C. Detailed Descriptions

In order for a robotics platform to replace a human in environments like those mentioned previously, it must be able to enter an environment, produce a map of the environment, plot where objects are located, and remember its history within the environment. This process is called simultaneous localization and mapping or Simultaneous Localization and Mapping (SLAM). Our design idea is to create a SLAM algorithm that has the capability of producing a 2-D map of a pre-defined environment with a pre-defined set of objects that can be applied to commercially available robotic platforms. The following sections describe some existing algorithms and platforms, along with our design choices.

1) Vision as a Sensor: In terms of the human sensory organs, the vision system is arguably the most powerful sensor. Likewise, in terms of electronic sensors, few devices can outperform a camera in the types of data that can be extracted to form output information. In recent years, the cost of cameras has been decreasing significantly. Because of this, cameras are replacing traditional sensors in applications such as automation, security, and robotics. Our system will be comprised of a vision system utilizing cameras as our primary form of sensor information.

Camera data is inherently complex, because the only output is an image. Any information gained from the image must be processed using computer vision techniques. In the case of a high resolution cameras, each frame can be comprised of an enormous amount of data making this processing very challenging. Because cameras produce so much data, and to work with them requires very complex linear algebra, standard microprocessors do

not have sufficient processing power to satisfy our requirements. Vision processing must be done with a computer using heavily optimized visual tools. To help assist in optimization, we need the ability to run processes in parallel with each other. Having the ability to process our vision data separately from our other processes will greatly improve our overall system performance.

2) Visual Odometry: The robotic platform must be able to discern locomotion through visual stimuli. In order to satisfy this requirement, we are to implement a system to perform VO. This concept is relatively new, having only been around for about a decade, and exists in both monocular and stereo forms. The overall process revolves around gathering motion data by processing sequences of frames that are on a moving platform. Most implementations follow the models presented in early VO research, such as analyzing the optical flow as shown in [1]. One of the first usages of this technology was on the Mars rovers. In high wheel slip environments, such as those found on extraterrestrial bodies, wheel encoder data becomes almost useless. In order to compensate for the massive amount of wheel slippage on their robots, NASA used VO [2]. The results of using VO were highly effective and even compensated for the imprecise odometry data coming from the encoders.

Stereo based VO implementations are more accurate and typically produce better results as shown in [3]. The main reason for this is due to the scale ambiguity problem - a monocular system cannot determine the scale of the odometry that it produces [4]. Our goal is to make our system platform independent. If we require our system to be a stereo implementation, there are many complex calibration steps that need to be run in order to produce valid data. By avoiding stereo implementations, we are able to skip the complex extrinsic calibration steps. This will simplify the calibration which will then allow our system to run on most platforms. Because we are only using one camera for our implementation, we will have to focus on a monocular solution to VO

Various implementations of monocular SLAM currently exist and most of these require fixed camera heights in order to attach a scale to the odometry output such as [5] and [3]. We will attempt to avoid these assumptions in order to make our system platform independent. If it is decided that we need scale output from our VO system, we must be able to assign the camera height of the system, by assigning it as a variable at run time.

3) Ranging: Computing the output from a vision system can take considerable time. Therefore, sensors that are less versatile are often used to provide faster ranging data for collision avoidance. Examples of these sensors are ultrasonic, infrared, laser, or small scale radar. Using lasers to gather range and distance information has proven successfully in applications described in [6]. Use of small scale radar systems has recently shown to be promising [7] for gathering range data. Both lasers and radar provide ranging information than can be used to for obstacle avoidance, but converting the input for immediate use for obstacle avoidance can be computationally intensive.

Our proposed solution is to use a microcontroller to gather data from multiple ultrasonic and infrared sensors. In order to lessen the amount of processing required by our primary hardware we will off load the data acquisition from the ultrasonic and infrared sensors to a microcontroller. This will also afford us the ability to make our system more modular. Whether this system is implemented by infrared or ultrasonic sensors the main control loop will still be able to use the data. By applyinh a threshold to the incoming range data, we can make informed control decisions and implement path planning to avoid obstacles. This data can also be used in conjunction with the VO feature to aide in producing a map of the robots environment.

4) Encoders: Another common sensor used to help provide localization input to a robotic platform are wheel encoders. As each wheel spins the sensor will gather data about the angular position change of the wheel. Once the encoder input has been captured it must be converted to linear velocity and used by a robotic kinematic model to create an estimate of the distance traveled and possible error. There are many ways of creating this model. The mathematical model of the robot platform routinely becomes more complex as the number of drive wheels are increased. An example of just how complex these mathematical models are can be found in [8]. However the most common and heavily documented robotic platform is the differential drive robot.

5) Differential Drive Robotic Platform: There are many advantages to designing our system to in-

terface with a differential drive robot. Do to its popularity, there are many differential drive platforms available for use. It provides a superior amount of maneuverability in confined spaces. It is also used as the basis for modeling more complex robotic drive systems that have similar drive characteristics. The differential drive kinematic model easy to compute because it has forward velocity and its lateral velocity can be set to zero in calculations. It is well documented and by using a well researched model we can get a strong estimate of our robots location from the velocity information collected from the encoders.

6) Filtering: There is an inherent amount of error in any sensor. A requirement of any robotics system is to use the sensor data and an estimate of the error associated with that data to increase the accuracy of the system. There are many types of filters used in robotics to perform this task. These filters are based upon using probability to increase the likelihood of gathering viable data from multiple sources. In essence they are the fundamental building blocks of sensor fusion required in SLAM. They usually are derived in some form of Gaussian or Markov filters. They each have their own advantages and disadvantages. A complex description of all possible filters is beyond the scope of this paper. The two most common filters used in autonomous mobile robotics SLAM systems are a particle filter or a Kalman filter that has been extend to function on non-linear systems.

Systems based upon particle filters are often referred to as FAST SLAM systems. The general concept of the particle filter is to track as many key points via the primary sensor as possible and form a mathematical relationship between those points to gather information about the systems location and pose. They track many more features for mapping than an Extended Kalman Filter (EKF). Particle Filters have proven to be highly effective, but are still experimental. A disadvantage of the FAST SLAM approach is that it is so new that using it as a solution to our system will be difficult. Another difficulty is our system will be need to incorporate localization data from multiple sources and this task will be more difficult if all data must be converted into a particle filter based SLAM algorithm.

The Extended Kalman Filter has been explored in robotics systems for a number of years and is considered to be well suited to the task of a visual SLAM system [9]. The EKF also has the ability to be adapted to accept data from localization and ranging sensors and incorporate this data in its output. The concept of accepting localization and ranging data into the same EKF filtering algorithm is commonly referred to as EKF-SLAM. It is also considered to be a viable solution to the full SLAM problem of autonomous navigation in an undefined environment.

D. Feature Details

The previous section discussed various existing research and implementations, while simultaneously describing our proposed solution. This section explains the specific hardware that we plan to use, an estimated number of man hours needed to implement, and other required features of our product.

1) Kinematic Model: A fundamental feature of our system will be the need to interface with a robotic platform and software to convert raw data to useful information. This feature should include software that gathers angular velocity information from the navigation/path planning software or encoders and use this angular velocity information to create a kinematic model of the robot for localization purposes. The same kinematic model should be the mathematical basis for use by the navigation and path planning model to perform the inverse operation.

The angular velocity information provided should distinguish between the left and right drive components of the robotics platform. This data is expected to be gathered from wheel or shaft encoders. After this feature has gathered the angular velocity of the robot it will convert it to linear distance traveled. The accuracy of this measurement needs to be within 30% of ground truth. The large allowance of accuracy is due to the nature of the filtering methods that are going to be used and the error that wheel encoders produce in high slip environments. An estimate of the error of this data will be plugged into the filter prior to final localization estimate.

2) Path Planning: A key feature in almost any mobile robotics system is the ability to use available sensor data and create a path to a desired point. Path planning is the term typically used to describe this process. There are many models currently available for path planning. This feature will be required to plan a path to a predetermined location form

the current location. This location can come from another feature or from a submodule.

This path planning feature must account for the physical dimensions of the robotics platform, surrounding area, and sensor data. The path planning section will incorporate data from the collision avoidance feature. It will also have access to a map that contains information such as the current and previous locations of the robot, and the locations of known obstacles.

This feature can be considered finished when it has the ability to navigate a defined space between 3 to 6 meters that contains two obstacles without contacting the obstacles. The navigation goal can be set by another feature or manually through a software interface such as a computer terminal with a physical user.

3) Visual Display: A key requirement for our system will be to provide an end user with data about the robot's location and the physical environment it has explored. Our proposed feature to address this issue is to provide the end user with a map displaying known obstacles and the robots path traveled. This feature does not have to be real time, but the system should be able to replay the robots path as it explored its unknown environment or display the systems end estimate of the unknown environment.

This feature can be considered finished when it has the ability to display the robots estimated path and detected obstacles goals on a 2 dimensional bird's eye map. This map should display all data to an appropriate scale so that further debugging or data can be collected from the end user.

4) Filtering: An essential requirement to incorporate multiple sensors into a design such as our system will be a probabilistic filtering scheme to perform sensor fusion. Our system will accomplish this task by using a Gaussian filter that has been extended to work on a non-linear system. Examples of such filters are the Extended Kalman Filter, Unscented Kalman Filter, or Sparse Extended Information Filter. This filter will need to be implemented in software, preferably writtin in C++. The filter can be a programmed by the team or open source third party software can be used if available.

This feature can be considered accomplished when it can accept input from the VO, kinematic model, and Inertial Measurement Unit (IMU) features and provide meaningful output about the robots location in relation to its surroundings. The output of the filter needs to be more accurate than the least accurate sensor input that is provided to it when compared to ground truth.

5) Collision Avoidance: Another essential requirement is to incorporate two different types of collision avoidance sensors. Our system will use five ultrasonic and five infrared distance sensors. One sensor of each type will be placed in five pre-determined locations across the front of the platform. The ultrasonic sensors will detect semi-rigid and solid objects while the infrared sensor can detect loose fabric material. The two sensors provide redundancy and provide an optimal setup for proximity and object detection. These sensors will be programmed to work with a microcontroller using C and C++.

This feature can be considered accomplished when the microcontroller can communicate independent sensor collision status via USB cable to the path planning feature for path adjustments.

The workload for this feature should be 150 hours. This allows 30 hours to research the interface requirements for the two sensor types to the microcontroller. The implementation and testing of the microcontroller software with the two types of sensors should take about 50 hours. The time requirement to integrate the collision avoidance output into the path planning feature should take 50 hours. The remainder of time will be used for debugging and physical testing of the feature.

6) Gyro & Accelerometer: An essential requirement is to incorporate both a gyroscope and an accelerometer sensor. Our system will accomplish this task by fusing a 3-axis gyroscope and a 3-axis accelerometer to provide x, y and z orientation data.

This feature can be considered completed when the microcontroller can communicate with both the gyroscope and the accelerometer sensors, and provide the fused sensor data to the control system feature using serial communication.

7) Communication With a Robot Chassis: The system will be required to communicate with a robotic platform. Our feature set to perform this task will be to utilize a Serial Communication software solution. The software for this feature needs to be extremely modular to allow it to be modified to work on various robotic platforms. This feature will be software based and will essentially be a wrapper program for complex libraries. It is preferred that

the feature be programmed in C++, but Python is acceptable. The feature should use a well documented software library to interface between the operating system and the USB port. By utilizing well documented libraries it will prevent possible communication errors.

This feature can be considered accomplished when it can accept input from the Path Planning feature and communicate this data to the robotic platform via USB cable. The feature will have a minimum 95% success rate for data transmission. The feature must be able to sustain serial communication for over 45 minutes without causing system errors.

8) Visual Odometry Requirements: The VO system must be able to accept an image for processing and output odometry or equivalent data. About 90 hours of this will be performing research, with 100 hours of implementation, and the remainder of the time should be for debugging physical testing of the feature.

E. Computer and Hardware Requirements

Due to the fact that the vision system will be our most computationally intensive part of the project. To technically implement this, we will require our system to be able to run on a modern computer with a multi-core CPU processing speed of no less than 1.3 GHz. Because the desired platform is a robotic system, this computer needs to be portable. This implies that we must use a laptop for a portable computing environment. Even still, it is estimated that our system will require further optimizations to increase performance and get more accurate data. For optimization purposes, we require the ability to process separate tasks in parallel with one another. One existing technology known as the Robot Operating Software or Robot Operating System (ROS) will be chosen to help us accomplish parallel processing. ROS is a technology that was initially developed by Willow Garage, and is heavily used in robotic research and development [10]. Because ROS is currently only supported on Ubuntu Linux environments, our laptop computer must be able to run Ubuntu Linux. This Linux system must be maintained properly to ensure a stable computing environment.

The type of camera needed for this project first and foremost needs to be compatible for use within a Linux environment. It also needs to have a minimum resolution of 640x480. We also need to be able to control many parameters of the camera itself including: resolution, white balance, auto focus, brightness, and sharpness. Being able to fix these parameters to known values allows for a more consistent and controllable testing environment.

F. Testing, Debugging, and Specific Documentation

In order to increase modular development and the lifespan of the system, we need to test each module to make sure it works. Debugging will occur in parallel to testing to insure proper fine tuning of the system.

After the successful implementation of each feature, we will write specific documentation of said feature in order to allow others to replicate our final product.

G. Resource Estimates

The features that were described in the previous sections all require many hours of research and development. A estimated summary of the amount of hours per task is shown in Table I along with who will be assigned to what task.

TABLE I
Man Hours

Task	**Estimated Hours**	**Assigned to**
Kinematic Model	170	Chris
Path Planning	371	Chris & Thomas
Visual Display	180	Curtis & Thomas
Filtering	230	Chris & Thomas
Collision Avoidance	150	Francisco
Gyro & Accelerometer	160	Chris & Francisco
Robot Communication	170	Thomas
Linux Maintenance	150	Curtis
VO	297	Curtis & Francisco
Goal Detection	110	Curtis & Francisco
Robot Repairs	92	Everyone
Room fabrication	50	Everyone
Total	2130 hours	All Team Members

II. Funding Proposals

This project was fully funded by our corporate sponsor FMC Technologies Schillings Robotics. During our initial meetings with our sponsor, we estimated a budget that included the robot, additional hardware and sensors, a laptop, and miscellaneous expenses. This budget also includes the materials

cost to fabricate a testing environment that can be broken down and transported in a vehicle. Our proposed budget for the entire project was $3,000. Our estimated costs can be seen in Table II Actual project invoices can be found in the appendix.

TABLE II
Estimated Budget

Item	Estimated Cost
Parallax Robot Platform	$1,000
USB ra	$100
IMU	$30
(3) Additional Ping/IR Sensors	$150
Microcontroller	$50
Laptop	$900
Breadboards	$20
Miscellaneous Connectors/Wire	$50
Environment Setup	$150
Unexpected expenses	$300
Custom PCB for external sensors	$100
Total	$2,850

III. Work Breakdown Structure

This project is going to be developed by a group of four individuals. To manage the work breakdown of this project, careful consideration was taken with the background of each person in mind and was therefore assigned specific tasks to complete.

A. Outline of WBS

As described in our design idea contract, this project involves designing, building, and implementing a software algorithm for use on mobile robots that work inside of hazardous environments. With an aggressive nine month time-line, the work breakdown is as follows:

1) Kinematic Model: This feature should include software that gathers angular velocity information from the navigation/path planning software or encoders and uses this angular velocity information to create a kinematic model of the robot for localization purposes. The same kinematic model should be the mathematical basis for use by the navigation and path planning model to perform the inverse operation.

a) Capture Angular Velocity Information: A fundamental feature of our system will be the need to interface with a robotic platform and software to convert raw data to useful information. The angular velocity information provided should distinguish between the left and right drive components of the robotics platform. This data is expected to be gathered from wheel or shaft encoders.

- Estimated Research Time: 5
- Estimated Implementation Time: 20
- Estimated Cost: $0.00
- Assignee: Chris
- Deliverable: A software object or function that can gather angular velocity commands from encoder data.

b) Convert Angular Velocity to Linear Pose: After this feature has gathered the angular velocity of the robot it will convert it to linear distance traveled. It needs to attach a scale to the data.

- Estimated Research Time: 20
- Estimated Implementation Time: 40
- Estimated Cost: $0.00
- Assignee: Thomas
- Deliverable: A software model that can deliver a measurement of the distance the robot has traveled.

c) Estimate Covariance: The data generated by the model needs to have a covariance associated with it in order for it to be plugged into the kinematic model.

- Estimated Research Time: 20
- Estimated Implementation Time: 25
- Estimated Cost: $0.00
- Assignee: Chris
- Deliverable: A software object or function that can provide an estimate of the error of the data associated with subsection III-A1b.

2) Path Planning: A key feature in almost any mobile robotics system is the ability to use available sensor data and find a path to a desired point. Path Planning is the term typically used to describe this process. There are many models currently available for Path Planning. Our system needs to be more robust than a simple wall hugging robot.

a) Path Planning: This feature will be required to plan a path to a predetermined location. This location can come from another feature or from a sub module. If the plan path is blocked this section needs to find a method to navigate around the obstacle.

- Estimated Research Time: 55
- Estimated Implementation Time: 90
- Estimated Cost: $0.00
- Assignee: Chris

- Deliverable: A software object of function that has the ability to navigate a defined space between 3 to 6 meters that contains two obstacles without contacting the obstacles. The navigation goal can be set by another feature sets or manually through a software interface such as a computer terminal with a physical user.

3) Visual Display: We need to provide the end user with a map displaying known obstacles and the robots path traveled. This feature doesn't have to be real time, but the system should be able to replay the robots path as it explored its unknown environment or display the systems end estimate of the unknown environment.

a) Map Display: The map should consist of a two-dimensional top down representation of the robots workspace. This map can be made from pre existing software or generated by the group.

- Estimated Research Time: 50
- Estimated Implementation Time: 90
- Estimated Cost: $0.00
- Assignee: Thomas
- Deliverable: A software object or function that has the ability to display the robots estimated path and detected obstacles. This map should display all data to an appropriate scale so that further debugging or data can be collected from the end user. The map should display data from top down and be two dimensional.

4) Filtering: An essential requirement to incorporate multiple sensors into a design such as our system will be the need to use a complex filter to perform sensor fusion. The feature our system will use to accomplish this task will be a Gaussian filter that has been extended to work on a non-linear system. Examples of filters that satisfy this requirement are the EKF, Unscented Kalman Filter, or Sparse Extended Information Filter.

- Estimated Research Time: 80
- Estimated Implementation Time: 80
- Estimated Cost: $0.00
- Assignee: Thomas & Chris
- Deliverable: This feature can be considered accomplished when it can accept input from the VO, Kinematic Model, and IMU features and provide meaningful output about the robots location in relation to its surroundings. The output of the filter needs to be more accurate than the least accurate sensor input that is provided to it when compared to ground truth.

5) Collision Avoidance: Another essential requirement is to incorporate two different types of collision avoidance sensors. The feature our system will use to accomplish this task will be to install five ultrasonic and five infrared distance sensors. One sensor of each type will be placed in five pre-determined locations across the front of the platform. The ultrasonic sensors will detect semi-rigid and solid objects while the infrared sensor can detect loose fabric material. The two sensors provide redundancy and provide an optimal setup for proximity and object detection. The combination of these sensor types will also detect fabrics and glass. These sensors will be programmed to work with a microcontroller using C and C++.

- Estimated Research Time: 50
- Estimated Implementation Time: 100
- Estimated Cost: $250.00
- Assignee: Francisco
- Deliverable: A software object or function that will prevent the robot from crashing into walls.

6) Gyroscope & Accelerometer: An essential requirement is to incorporate both a gyroscope and an accelerometer sensor. The feature our system will use to accomplish this task will be a 3-axis gyroscope to provide yaw, pitch and roll data as well as a 3-axis accelerometer to provide x, y and z axis acceleration data. This data will be fused together using a complementary filter to resolve angular pose.

- Estimated Research Time: 40
- Estimated Implementation Time: 120
- Estimated Cost: $30.00
- Assignee: Chris & Francisco
- Deliverable: A software object or function that will read the data from IMUs and filter

7) Communication With a Robot Chassis: The system will be required to communicate with a robotic platform. Our feature set to perform this task will be to utilize a serial communication software solution. The software for this feature needs to be extremely modular to allow it to be modified to work on various robotic platforms. This feature will be software based and will essentially be a wrapper program for complex libraries. It is preferred that the feature be programmed in C++, but Python is acceptable. The feature should use a well docu-

mented software library to interface between the operating system and the USB port. By utilizing well documented libraries it will prevent possible communication errors.

- Estimated Research Time: 50
- Estimated Implementation Time: 90
- Estimated Cost: $0.00
- Assignee: Thomas
- Deliverable: A software object that can pass instructions from our main system to the robot using serial communication with a minimum of 95% success rate for data transmission. The feature must be able to sustain serial communication for over 45 minutes without causing system errors.

8) Visual Odometry Interface: The VO system must be able to predict the motion of the camera through careful processing of image frames. When completed, the VO system must be able to measure the odometry of the camera with measurable error with respect to ground truth. Whether this system is implemented in house or we use third party examples, it has to be able to fit within our main system, and therefore comply with our messaging requirements

a) Implementation: To publish the required odometry data, we must implement a system that receives camera information and images, and processes them to output odometry data. This will be accomplished through the 8-pt algorithm

- Estimated Research Time: 100
- Estimated Implementation Time: 50
- Estimated Cost: $0.00
- Assignee: Curtis & Francisco
- Deliverable: A software object that can generate odometry data from a moving camera.

9) Testing, Debugging, and Specific Documentation: In order to increase modular development and the lifespan of the system, we need to test each module to make sure it works. Debugging will occur in parallel to testing to insure proper fine tuning of the system.

After the successful implementation of each feature, we will write specific documentation of said feature in order to allow others to replicate our final product.

a) Testing & Debugging of the Kinematic Model: We have to verify that this system works by itself before we include it into the main project. In this stage we will attempt to estimate any errors and limitations associated with this feature. This testing and debugging stage involves attempting to fix errors and bugs produced by improperly calibrated sensors, or software issues. In the case that an error or bug cannot be fixed, it is to be documented for later tweaking.

- Estimated Research Time: 5
- Estimated Implementation Time: 35
- Estimated Cost: $0.00
- Assignee: Chris
- Deliverable: A set of tests that ensure that the accuracy of the III-A1b Section delivers an estimate of the robots position within 30% when compared to ground truth when performing simple.

b) Testing & Debugging of Visual Odometry: We have to verify that this system works by itself before we include it into the main project. In this stage we will attempt to estimate any errors and limitations associated with this feature. This testing and debugging stage involves attempting to fix errors and bugs produced by improperly calibrated sensors, or software issues. In the case that an error or bug cannot be fixed, it is to be documented for later tweaking.

- Estimated Time:
- Estimated Cost: $0.00
- Assignee: Curtis & Francisco
- unDeliverable: A software object that can generate odometry data from a moving camera.

c) Testing & Debugging of Filtering: We have to verify that this system works by itself before we include it into the main project. In this stage we will attempt to estimate any errors and limitations associated with this feature. This testing and debugging stage involves attempting to fix errors and bugs produced by improperly calibrated sensors, or software issues. In the case that an error or bug cannot be fixed, it is to be documented for later tweaking.

- Estimated Research Time: 5
- Estimated Implementation Time: 65
- Estimated Cost: $0.00
- Assignee: Chris
- Deliverable: A set of tests that ensure that the filter is performing as expected to scale and test for limitations.

d) Testing & Debugging of the Visual Display: We have to verify that this system works by itself before we include it into the main project. In this stage we will attempt to estimate any errors and limitations associated with this feature. This testing and debugging stage involves attempting to fix errors and bugs produced by improperly calibrated sensors, or software issues. In the case that an error or bug cannot be fixed, it is to be documented for later tweaking.

- Estimated Research Time: 5
- Estimated Implementation Time: 25
- Estimated Cost: $0.00
- Assignee: Chris
- Deliverable: A set of tests that ensure that the map is to scale and test for limitations.

e) Testing & Debugging of Path Planning: We have to verify that this system works by itself before we include it into the main project. In this stage we will attempt to estimate any errors and limitations associated with this feature. This testing and debugging stage involves attempting to fix errors and bugs produced by improperly calibrated sensors, or software issues. In the case that an error or bug cannot be fixed, it is to be documented for later tweaking.

- Estimated Research Time: 5
- Estimated Implementation Time: 30
- Estimated Cost: $0.00
- Assignee: Chris
- Deliverable: A set of tests that ensure that the accuracy of the III-A2 Section delivers an estimate of the robots position within 30% when compared to ground truth when performing simple.

B. Resource Estimate Summary

The features that were described in the previous sections all require hours of research and development. An estimate of the amount of hours per task is shown in Table III along with who will be assigned to what task.

TABLE III
Man Hours

Task	Estimated Hours	Assigned to
Kinematic Model	170	Chris
Path Planning	180	Chris & Thomas
Visual Display	180	Curtis & Thomas
Filtering	230	Chris & Thomas
Collision Avoidance	150	Francisco
Gyro & Accelerometer	160	Chris & Francisco
Robot Communication	170	Thomas
Linux Maintenance	150	Curtis
VO	230	Curtis & Francisco
Goal Detection	150	Curtis & Francisco
Total	1770 hours	All Team Members

C. Project Timeline

This project is intended to span two semesters and the tasks and work breakdown structure for the duration of the project was initially estimated at the beginning of the Fall semester. As this project has progressed through the Fall semester, the timeline was updated to reflect the upcoming tasks, goals and challenges that lie ahead to produce the final product.

1) Milestone 1: Visual Control of a Robot: Because our robot project is designed to use cameras as a sensor to be used on a mobile robot, it was determined that as a proof of concept, we should attempt to demonstrate that we can control our robot through mainly using machine vision. As a breadboard proof of our system, we demonstrated visual control of our robot by making it locate and drive to an orange soccer ball. This was a very important milestone as it demonstrated the base functionality of many subsystems. For this proof of concept to work, we needed to be able to demonstrate an assembled robot, functioning communication with the robot chassis, functional processing of image sequences, and basic robotic control through path planning. While most subsystems were in very rudimentary stages of implementation, their implementation demonstrated that we are prepared to meet our project goals and milestones.

2) Milestone 2: All Features Implemented: After we have demonstrated visual control of our robot, we are to set to work on implementing all the features listed in the feature set. When all of these features have been implemented, Milestone 2 is considered complete. The primary focus of this milestone is to get functioning data coming from the encoder odometry as well as the VO. These odometry models are to be used in tandem for producing better usable datas than each odometry model is capable of producing itself. Achieving this milestone begins a long and involved testing and debugging phase of our project.

3) Milestone 3: Filtering Odometry Models: After the odometry models have been verified to produce usable data within a certain percent error from ground truth data, we are to begin fusing the two data sources by using an extended kalman filter. This probabilistic filtering approach will enable us to tune our system for more precise odometry information. The testing and debugging phase involved in this will be extensive, but once the models are filtered, our system will be ready for the next face.

4) Milestone 4: Mapping and Path Planning: After our odometry models have been fused by using the EKF, our next focus is on the mapping and path planning milestone. This milestone will take our fused, known good data and bring it into a mapping application. It will also use this collected map data to form a path to known goals.

5) Milestone 5: Project Completed: Once all subsystems are reporting to the mapping application, we have another system testing and debugging phase. When this phase is completed, the overall project can be said to be completed. This is anticipated for completion at the end of the Spring semester.

IV. Risk Assessment & Mitigation

Part of any design is to perform a risk assessment and provide mitigation plan for the items that were discovered to have a risk of negatively impacting the project. The following section will address the critical risk factors and measures we have taken to address those risks. For each risk an associated value of Low, Medium, High, or Very High is assigned depending on the overall impact this will have on our project. Our estimated probability of this event occurring is assigned using the same scale. This allows us to focus on which risks need the majority of our attention in order to mitigate them. The following risk assessment chart outlines the risks and likelihood of failure of our feature set as perceived by our team.

Note: Risk assessment is in the following format (Likelihood, Impact)

A. Kinematic Model:

1) There is a chance that the wheel encoders introduce as much as 2.5 inches of error second. (Low, Medium)

	Impact			
Likelihood	Low	Medium	High	Very High
Very High	Kinematic Model (2)	Filtering (1)	Visual Odometry Interface (2)	Visual Odometry Interface (1)
High	Eddie Risks (2)	Path Planning (1)	Filtering (2)	Eddie Risks (1)
Medium	Visual Display (1)	Collision Avoidance (1)	Communication with Robot Chassis (1)	Camera Risks (1)
Low	Battery (1)	Kinematic Model (1)	Laptop Risks (1)	Laptop Risks (2)

Fig. 1: Risk Assessment

2) All kinematic models have constraints that must be addressed in the software and complex programming can cause considerable risk to a projects success. (Very High,Low)

Steps Taken to address risks:

1) We have researched how to update the encoders and possibly the firmware associated with those sensors.
2) We will test our model thoroughly prior to demo.

B. Path Planning:

1) There are some considerable constraints placed on our implementation of Path Planning, caused by the VO nodes requirements and this may lead to a much longer development time than expected. (High, Medium)

Steps Taken to address risks:

1) We have started research Path Planning models that are well suited for controlling differential drive robots that have the ability assist in working around constraints.

C. Visual Display:

1) Event driven display software is highly susceptible to runtime errors. (Medium, Low)

Steps Taken to address risks:

1) We will have to use some type of exception handling or error checking prior to sending data to this node.

D. Filtering:

1) Non-Linear Filters can be difficult to tune properly. (Very High, Medium)

2) We are using open-source software to perform our filtering tasks and they might not be fully functional. (High, High)

Steps Taken to address risks:

1) We have started implementing testing and debugging protocols to make sure that we eliminate as many environmental variables as possible during tuning.
2) We have heavily researched Non-Linear filters and linear algebra libraries to ensure that if the software fails we can implement a filter with minimal time delay.

E. Collision Avoidance:

1) Ultrasonic Sensors cant see all materials or objects. Our robots workspace is small, so collision avoidance is difficult and we might crash without perfect sensor data. (Medium, Medium)

Steps Taken to address risks:

1) We have IR sensors that can be brought into the system to allow a total of 10 close proximity sensors.

F. Communication with Robot Chassis:

1) Serial Communication can be prone to error. (Medium, High)

Steps Taken to address risks:

1) We have allocated a lot of testing time to ensure that this module is tested thoroughly.

G. Visual Odometry Interface:

1) We are using open-source software and we are unaware if it has been tested on an autonomous mobile robotics. (Very High, Very High)
2) Current implementation does not implement error checking, and allows data to degenerate. (Very High, High)

Steps Taken to address risks:

1) We have allocated a lot of testing time to ensure that this module is tested thoroughly.
2) Known issues will be worked around in our other systems.

H. Laptop Risks:

1) Complete Laptop failure due to hardware or software complications. (Low, High)
2) Complete data loss (Low, Very High)

Steps Taken to address risks:

1) We continually backup the complete hard drive image to an external 1 TB drive. Which allows for recovery in case of hard drive failure. In addition, the majority of the code that has been written is stored on Github, which allows for recovery if we need to reinstall ROS.
2) Because our system is modular, we can effectively swap the production laptop for one of our own personal laptops giving us a total a 4 backup laptops which will be able to run the complete system with minimal setup.

I. Camera Risks:

1) Complete camera failure due to hardware or driver support.(Medium, Very High)

Steps Taken to address risks:

1) We have purchased a total of three cameras, in case we ever have the primary camera fail.

J. Eddie Risks:

1) Eddie board failure. (High, Very High)
2) Eddie not as advertised. (High, Low)

Steps Taken to address risks:

1) All of the power inputs on the Eddie board have been fused for over current protection.
2) Beside the fact that the Arlo platform is labeled as a turnkey robotic solution, we chose it because of the proximity of Parallax and their guarantee that if we should run into problems they will support us.

K. Battery Risks:

1) Battery case cracks and leaks acid. (Low, Low)

Steps Taken to address risks:

1) The batteries sit in a dedicated battery housing under the lower Eddie platform deck. They are secured in place by hardware standoffs and the batteries lie between two thick plastic plates. There is very minimal risk of a battery case cracking to due to vibration or shock.

V. User Manual

This project requires very specific hardware and software in order to operate. It is assumed that the user is familiar with the ROS, Linux terminal commands, both Sketch and Spin IDE's, as well as C/C++ and Python programming. The system is designed to be operated either locally (all commands typed on the laptop mounted on the robot) or remotely using an additional laptop and router to remotely control the robot mounted laptop.

A. Room Requirements:

Eddie must be operated on a flat, smooth, hard surface (i.e. cement, asphalt, un-waxed tile or hardwood flooring, etc.) in an area no less than 3 x 3 meters, and ideally 5 x 5 meters. The room walls must be either a solid material (wood/brick/steel) or made of a thick covering such as 12mil polypropylene or heavier. The floor and wall surfaces must contain a scattering of small, flat items such as leaves, stickers, or alvar tags in order for the visual odometry to properly function. The room environment must have fairly uniform incandescent or fluorescent lighting and be free of ground debris such as small rocks, sticks/twigs or any object that will cause the motors to draw excess current to overcome.

Caution: The drive motor fuses are severely underrated for the amount of current the motors draw and the fuses will blow if any additional resistance is added to the path of travel, including an incline or debris. If "Eddie" is about to crash into something, immediately turn off the drive motor power switch on the distribution board.

B. Local Mode

To operate this project as a standalone system with no external control:

1) Hardware Required:

- Laptop with at minimum an Intel i3 processor, 1600Mhz DDR3 RAM 4Gb, wireless adapter, 3+ USB 2.0 ports
- Parallax "Eddie" robot platform w/Eddie Control Board [1]
- Ultrasonic Ping/IR sensors [2]
- Logitech C920 USB HD Pro Web Camera

[1]http://www.parallax.com/product/28992

[2]http://www.parallax.com/product/725-28998

2) Laptop Software Required:

- Ubuntu 14.04 [3]
- ROS Indigo [4]
- Python PyQT4 [5]
- Sketch IDE [6]
- Simple IDE for Linux [7]

To start the system up:

1) Turn on laptop and wait for system to fully boot up.
2) Place the faces around the environment
3) Turn on the "Eddie Control board" power switch (right switch) located on the power distribution unit. Next, turn on drive motor power (left switch) on power distribution board.
4) Plug the Eddie control board USB cable into the laptop. NOTE: This MUST be the First usb device connected.
5) Plug in the Atmega328 microcontroller development board and verify that all 5 ping sensors are flashing green lights. If not, open the Sketch IDE and re-load the collision_avoidance.ino file. Verify all 5 ping sensors are flashing before proceeding.
6) Plug in the USB camera
7) Open a Terminal window
8) Launch the production SLAM program from a bash terminal:
   - $ roslaunch production master.launch
   - On the control panel, press the "explore" softkey and the program will begin.
   - The map on the terminal will continuously update the explored area and will also place a number 4 thru 7 (based on the face). When the pre-set number of faces has been identified, the robot will automatically perform path planning from its current location to the starting point, go to each of the faces detected, then return home and stop.

To drive the robot around:

1) Place Eddie at least 50cm from any walls or objects
2) Option 1: Perform above instruction including typing $ roslaunch production master.launch

[3]http://releases.ubuntu.com/14.04

[4]http://wiki.ros.org/indigo

[5]http://wiki.python.org/moin/PyQt4

[6]http://arduino.cc/en/main/software

[7]http://learn.parallax.com/propeller-c-set-simpleide

- Use sliders and commands on control script to move robot forward, backward or turn.

3) To "STOP" robot, press [stop] button on control script. Allow 200mS for the robot to come to a complete stop.

C. Remote Mode

To run the system in remote mode, where it can be controlled from a WiFi connected workstation, additional hardware is required:

- A second laptop with at minimum an Intel i3 processor, 1600Mhz DDR3 RAM 4Gb, loaded the with same software and ROS Production file as the robot's laptop
- Linksys WRT-110 wireless router
  - router settings: DHCP, no firewall restrictions, no throttling

To set up the wireless router:

1) Enable wireless access
2) Enable SSID broadcast
3) Enable DHCP
4) Disable all firewall restrictions
5) Disable throttling

Run the following on the robot laptop:

1) Perform the above instructions until you reach "$ roslaunch production master.launch". DO NOT execute this command.
2) Verify the robot Laptop is connected to the correct wireless network
3) Determine robot IP address of robot laptop and record $ifconfig
4) Manually set IP address and port number of robot laptop: xxx-xxx-xx-xx:11311
   - $export ROS_MASTER_URI = http://(robot ip address from above):11311
   - $export ROS_IP = 'hostname -I'

Run the following on the remote control laptop:

1) Turn on remote laptop and open a Terminal window
2) Connect to the correct Wi-fi network
3) Enter the ROS directory:
   - $ cd curtkin
4) Connect to the robot laptop using SSH:
   - $ ssh -l team1 (robot ip address from step 8):11311
5) Enter password to login to the robot laptop
6) Determine IP address of the remote laptop and record
   - $ifconfig
7) Manually set the IP address of the remote laptop: xxx-xxx-xx-xx:11311
   - $ export ROS_MASTER_URI = http://(remote ip address):11311
   - $export ROS_IP = 'hostname -I'
8) Launch ROS SLAM program:
   - $ roslauch production remote.launch

Finally, to drive the robot around:

1) Use same precautions describe above in manual mode "driving instructions"
   - Option 1: Type $ roslaunch production master.launch and use sliders and commands on control script to move robot
   - Option 2: Press the "Explore" button

VI. Design Documentation

The hardware and software requirements to implement a SLAM robot may not appear very challenging at first - get a robot, load some software, plug in a few sensors and the system should work, right? The short answer: No, not even close. Our experience in choosing a commercially available robot and implementing the proper software was anything but straight forward and included hundreds of hours of research and extensive hardware and software troubleshooting. Our team researched several manufacturer's of differential drive robots ranging from Vex, iRobot, Dagu and finally decided on the Parallax "Eddie" robot. Once the robot platform was decided on, our team struggled with a means to integrate all of the software programs that must run simultaneously into a single control program structure. It was around this time we learned about the Robot Operating System (ROS). Our teams' introduction to this amazing program allowed us to fully integrate all of our hardware and software in an environment that performed parallel processing based on event driven programming. We finally had all of the necessary building blocks to begin our SLAM robot project.

The system documentation below is intended to provide a summary of the hardware and software used in this project. First, the hardware section will

describe all of the main hardware components, including interconnection diagrams and a system connection overview depicting the relationship between our hardware features and ROS. Second, each type of software program being used will be covered. This includes an overview of the external programs interfacing with ROS and associated flowcharts, and an in-depth look at the relationship of each ROS node used and the topics associated with each node. It must be mentioned that there are aspects of ROS which we do not fully understand due to its complexity. ROS is an open source program that has tutorials that enable the user to implement the code without being bogged down with the program's details.Therefore, processes happening internal to ROS will not be covered.

VII. Breakdown of Hardware Subsystems

Our core hardware is comprised of: A laptop, a Parallax Eddie robot, an Atmega 328 microcontroller, five ping/IR sensors and a camera. Eddie is a differential drive robot comprised of a microcontroller containing an integrated PID, two optical wheel encoders, two PWM driven motors and two lead acid batteries. Eddie is programmed to directly interface with the ping/IR sensors however, our SLAM algorithm could ideally interface with any robot platform that was differential drive. Keeping this in mind, we chose to use an Atmega328 microcontroller to control the ping/IR sensors, which allowed ROS to communicate only encoder data with Eddie. The assembled Laboratory Prototype hardware can be seen in Figure 2.

A. Encoders

Initially, Eddie was driven by a GO command which uses a set velocity for travel but does not give feedback as to how far the robot has traveled or in which direction. When we tried to use the GO SPD command, which uses the wheel encoders for movement, it did not work. We contacted Parallax and with some factory support, we managed to get an alternate set of less accurate 32 tick encoders to work without learning why our original encoders failed. Figure 3 shows the replacement encoders and Figure 4 shows Francisco hard at work debugging encoder data.

As our project progressed, we needed to use the original wheel encoders for their accuracy so,

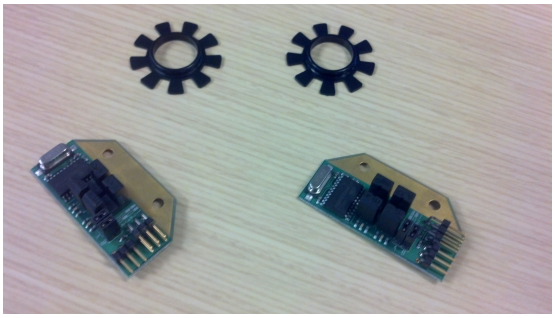


Fig. 3: 32 tick encoders

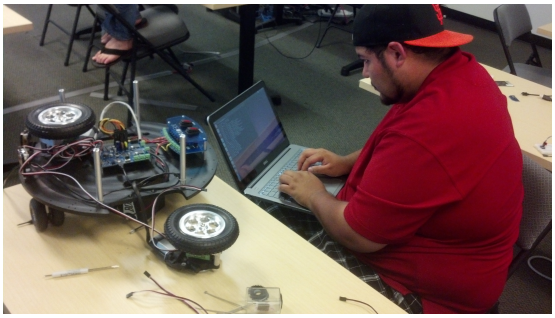


Fig. 4: Troubleshooting 32 tick encoders

after a second trip to Parallax and some extensive troubleshooting, we found that one of our original encoders was bad, which caused the original issue. Figure 5 shows the new encoders. Thanks to Parallax's help, we were able to get our project back on track. Figure 6 shows the team after troubleshooting the encoder issues with Parallax.

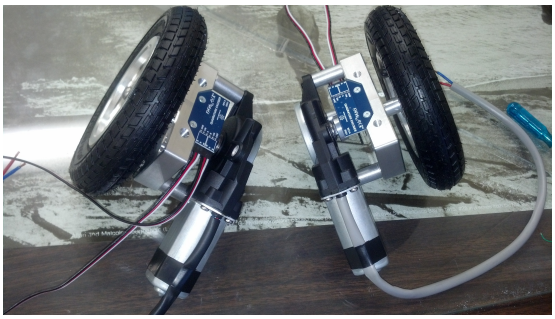
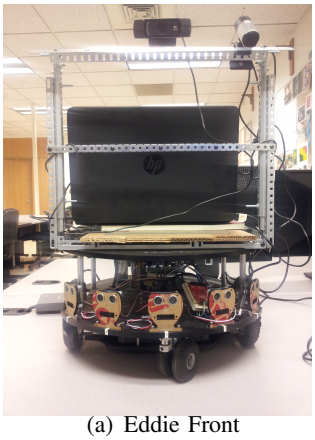


Fig. 5: Motors with 144 tick encoders

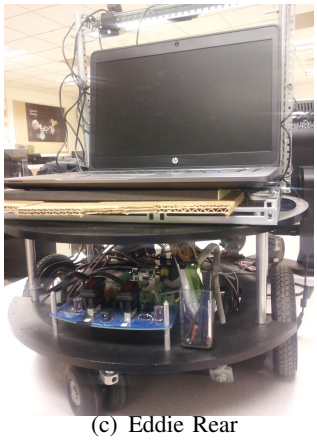
B. Camera

In order to process the vision data, we need a vision sensor. Because webcams are cheap, readily available, and easy to interface, it was decided to use one for this project's vision sensor. After looking at recommended options, the team decided on the Microsoft Lifecam Studio as seen in 7.



(a) Eddie Front

(b) Eddie Side



(c) Eddie Rear

Fig. 2: Eddie Robot Chasis

Fig. 6: Andy & Chris & Thomas at Parallax

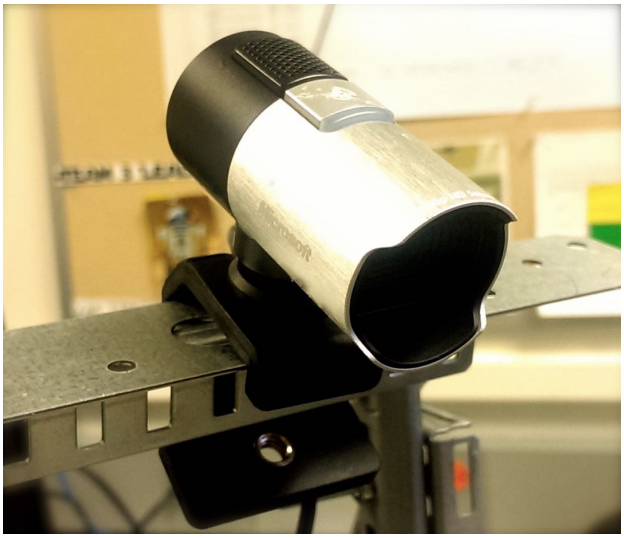


Fig. 7: Microsoft Lifecam Studio

This camera is capable of producing our desired 640x480 pixel image at 30 frames per second (FPS). Unfortunately, during the use of the Lifecam, it became apparent that it would not work for our project. The Lifecam has a numerous issues during use on the Linux environment. We attempted to implement some fixes onto our system (see the appendix), but we eventually deemed the camera too unstable to move forward with in the project. We did some research and physically tested various readily available webcams before arriving with our current choice, the Logitech C920 (seen in Figure 8). This camera produces the same 640x480 at 30 FPS video and in testing is a lot more stable within our software environment than the Lifecam.

Fig. 8: Logitech C920

C. Atmega 328 Development Board

An Atmega 328 microcontroller development board is used on this project for rapid protoyping a solution in order to offload some of the ping data processing that would normally be calculated on the Eddie control board. This move is made to isolate

the Eddie control board to only worry about wheel encoder data and drive commands.

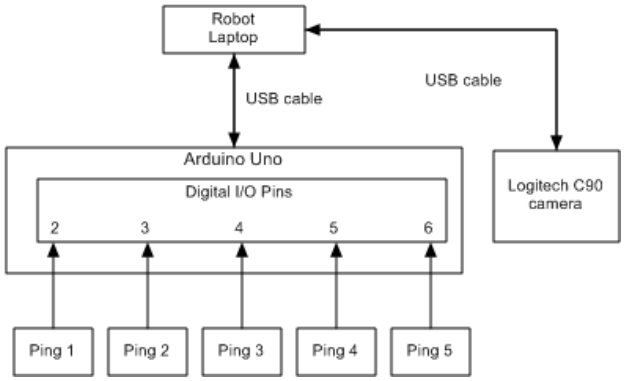


Fig. 9: Microcontroller w/Pings and Camera to Laptop

The overall hardware system flowchart can be see in Figure 10.

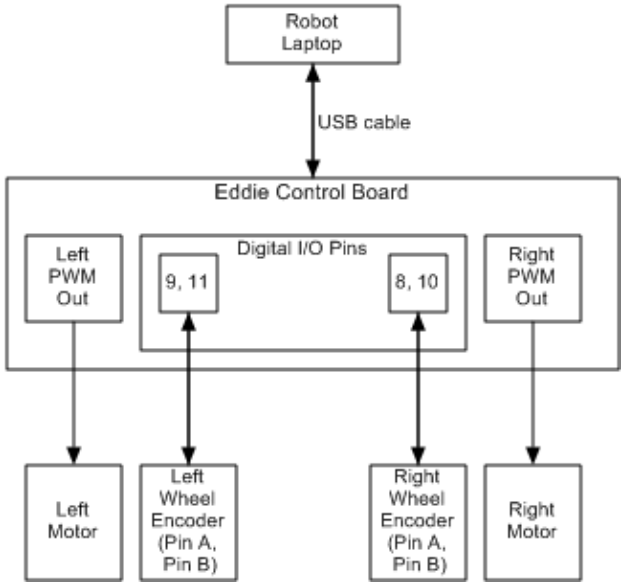


Fig. 10: Robot Hardware Flowchart

VIII. Breakdown of Software Subsystems

The software descriptions contained in the section are intended to provide an introduction to our overall software subsystems. Every program that we have developed or are using are developed to be utilized within the ROS environment. Unless specifically stated each section is running on it's own node and it's inputs and outputs are passed via topic through the ROS processing abstraction layer. The programs are designed to be implemented as standalone nodes which allows them to be worked on independently and improved or replaced as needed. Please note that although individual nodes are designed to be launched as standalone programs they will not perform their intended task unless launched as a whole as their collective collaboration is required to achieve any measure of success.

A. Robot Communication

With the selection of a robotics platform and a the laptop properly configured it was now time to establish communication between the robot control board and our programs that would need communication in ROS. This software will be the pathway that conveys and receives data between two of our major hardware components. We elected to go with a standard USB hardware connection with a ROS node to interact with the serial connection. There was minimal hardware design requirements due to using existing communication standards.

With the hardware requirements met we now began researching to find a serial driver that would provide the software communication between the ROS nodes and the control board. We initially investigated using a previously established ROS node that was written specifically for our specific robotics platform. This node would have provided the serial communication requirements with little to no alterations. While attempting to implement this existing code we ran into compatibility issues regarding specific software dependencies. We spent about a 80 combined ours attempting to get this software to work, but every path turned out to be a dead end.

We no existing software available to meet our needs we began development on our own ROS node to accomplish the requirements. The most import aspect when designing this software was to ensure that high reliability could be achieved from this. As this sub-system communicates the control systems desired velocity information to be implemented by the robots firmware board. If this communication were to break it could result in undefined behavior of the robotic chassis and catastrophic system failure.

With the design priority of reliability set we began researching existing C++ libraries that allow for the establishing serial communication via the USB connection on the PC. There were multiple implementations that were found, but in the end

we chose to find a solution based upon the boost libraries, because they are open source and highly documented. We were able to find a light weight open source wrapper of these libraries that allowed easy integration into our existing software development environment.

Once we had proven that we could establish communication between the robotic chassis and the PC via the USB cable we then began focusing on how we are going to integrate our software into the Robot Operating System environment. The modifications to our existing serial program involved allowing it to capture published messages from other nodes and relaying these messages to the robotic chassis. This allows multiple nodes to pass or receive messages at the same time.

During development we discovered that robotic chassis communicates with signed hex representation saved into standard c-string format. This proved problematic due to the inherent nature of humans to desire visual displays of signed decimal numbers, even when stored in c-strings. The existing standard libraries that exist for performing this type of conversion did not perform to our expectations.

Once this conversion had been established at began initial testing of this subsystem. For initial testing we developed a small program that essential allowed us have a basic terminal interface with the robot chassis. The testing program was it's own node that communicated with the our serial driver via publishing a ROS topic called eddie do. This essential simulated our control system providing commands for our robot chassis to implement. Initial testing proved that our implementation was sound and we began adding additional features to our serial driver node.

One of the tasks that the serial node is responsible for is retrieving the values of the encoder ticks from the robot control board. This is accomplished by sending the command "DIST" to the robot control board and waiting for it to provide the amount of encoder ticks that have passed from the last sample time. We had originally attempted to utilize standard threading to achieve the a desired delay between sample times to allow the robot firmware to execute other task. The addition of a stand thread delay resulted in the serial node being unavailable for time critical functions such as sending stop commands. We then implemented a standard time work around that allows us to control the frequency of the requests. There will be small amounts of time fluctuations between the samples, but nothing in our system requires this feature to be deterministic. We can adjust the amount of samples per second by simply adjusting the variable that controls this.

This sub-system has about two hundred hours of functional testing and seems to be functioning properly. Due to the critical nature of this subsystem we are considering adding additional runtime error checking procedures that would allow our system to perform dynamic communication establishing if the system is unplugged. This sub-system is in debugging phase

B. Visual Display

Visualization of data from multiple sources is critical to provide feedback for debugging and rapidly displaying the systems current status. The information that we needed to display consisted of the outputs from the Visual Odometry, Kinematic Model, and the output of the filtered data.

The display needed to provide a top down two-dimensional map of the robots environment and path. ROS has some built in utilities such as RViz which allow mapping. The interface with these utilities was complex and the time to implement them would have been considerable. Due to the time crunch it was decided that we would implement our own mapping node that would display the required features.

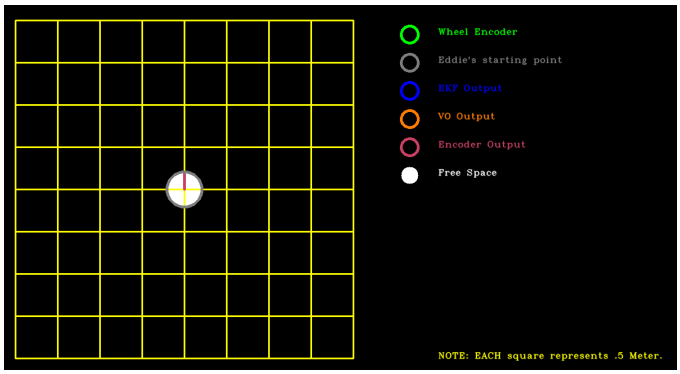


Fig. 12: Visual Display

Initially we decided on some of the key design features of this node. Since this feature is needed for assisting in debugging and troubleshooting of the system we needed to ensure that this data is accurate and to scale. We also decided that ensuring that this map could be rescaled to various environment sizes

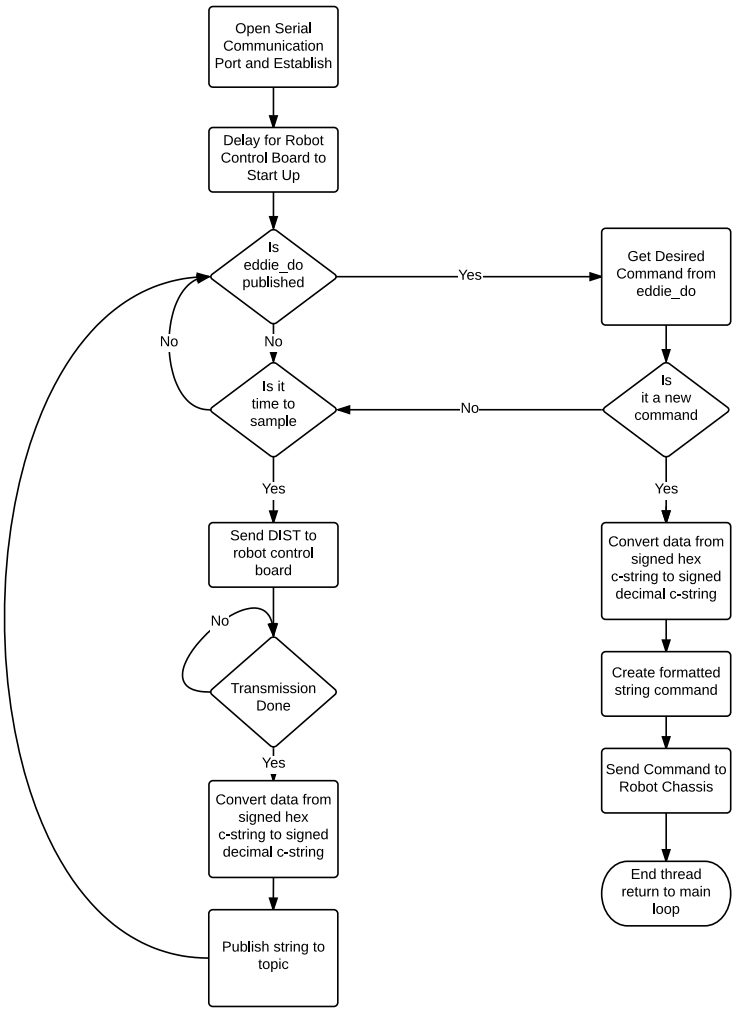


Fig. 11: Serial Signal Path

was a priority. Additionally the map needed to be threaded to ensure that it limited the amount of processing the PC had to do. Graphics representation can be very processor intensive.

We elected to implement this node in C++ because of the possibility of needing large amounts of data storage which means that the if we used a slower language we would increase the processing load of our PC. We decided on using the OpenCV library because it is highly documented and has many code examples. It is known for having a large amount of drawing functions and has many back end accelerators built into the it. In OpenCV an image becomes a matrix and it is easy to manipulate or draw on that image creating the display we needed.

This node is built into a parent node that contains

the kinematic model and pre-filter functions. This design choice was made because it allows all three of these sections to have access to the same data storage containers. The data storage containers we chose to use are the C++ standard vectors. This design choice was made because they are easy to use and highly documented. They allow much faster processing time than using standard C++ arrays and take care of dynamic memory processing much easier than doing it manually. The use of standard vectors minimizes the chance of memory leaks, when compared to implementing data storage alone. A vector is created for the outputs of the kinematic model, visual odometry, and EKF. The map then has access of this information so it can plot the location.

The first step when creating the map was to decide on how much space the image should take up. Originally we went with a smaller window, but this greatly decreased the ability to get precise visual information from the display. We then set it up to maximize upon opening. Now that we knew the working environment of our display we set out to actually create our display.

We elected to assume that our environment in square for the display purposes. If it turns out to be another shape it will still fit inside of a square with minimal rescaling. We split the display into two regions, one containing the square to contain the map of our environment and a second region to contain the map key and other data. This makes understanding the meaning of the map intuitive and easy to learn.

To ensure proper scaling of features in the map there was some basic steps we had to perform. The first region, the map square, has a predefined pixel width that is allocated at the top. At compile time the user adjusts the estimated physical size of the room. From this data the node creates a pixel scale by dividing the physical width of the room by the number of pixels. This attaches a meter scale to each pixel. This scale is important because it is used as the scaling factor to ensure everything in the map is plotted to scale.

The mapping updates at the request of the parent program which controls the thread, and thereby the frequency of the map update. The flowchart explains the detailed principle of operation.

We have verified that the scale is accurate enough for us to use for initial testing an debugging purposes. We have utilized the display in presentations to our sponsors from FMC Technologies Schillings Robotics and it has been well received.

Our current design satisfies our original design goal. We are still in search of a program that will allow us to display 3D data during our final demonstration. Currently we are exploring the ROS Rviz package and are hoping to have it ready for demo day. The reason that we are hoping to use Rviz is that it is a much more professional looking product that delivers a high degree of polish and refinement to the visual display. We will continue exploring Rviz until we decide that implementation will require more time or skill than we currently have.

C. Kinematic Model

Each individual robot has a to have a mathematical model so that odometry information can be generated from the on board sensors. In our case we are going to use data from our wheel encoders to generate localization data regarding our robots position in the global environment. This localization information is referred to as odometry, and will refer to it as wheel odometry.

We attempted to find existing software solutions to generate this data, but we were unable to find a suitable model. Having exhausted our search options we began brainstorming on how we were going to implement our own node to perform these kinematic calculations to generate wheel odometry data. We decided that speed, precision, and data storage were the essential design characteristics we needed to achieve. We elected to begin developing in C++, because it is well suited for speed and had the ability to interact well with the previously discussed nodes.

The actual kinematic calculations became part of a larger parent C++ class that the visual display and pre-processing for the EKF are part of. This allowed for a more rapid creation of the node and a more cohesive code flow. By creating the nodes using common data storage techniques it allows our program to be faster and easily improved.

We initially started looking into the various types of models that can be created for a differential drive robot. Because we are interested in tracking the location of the robot throughout its run we focused on forward based kinematic modeling. This simply means that we are going to use a system of equations to get the required data that we need.

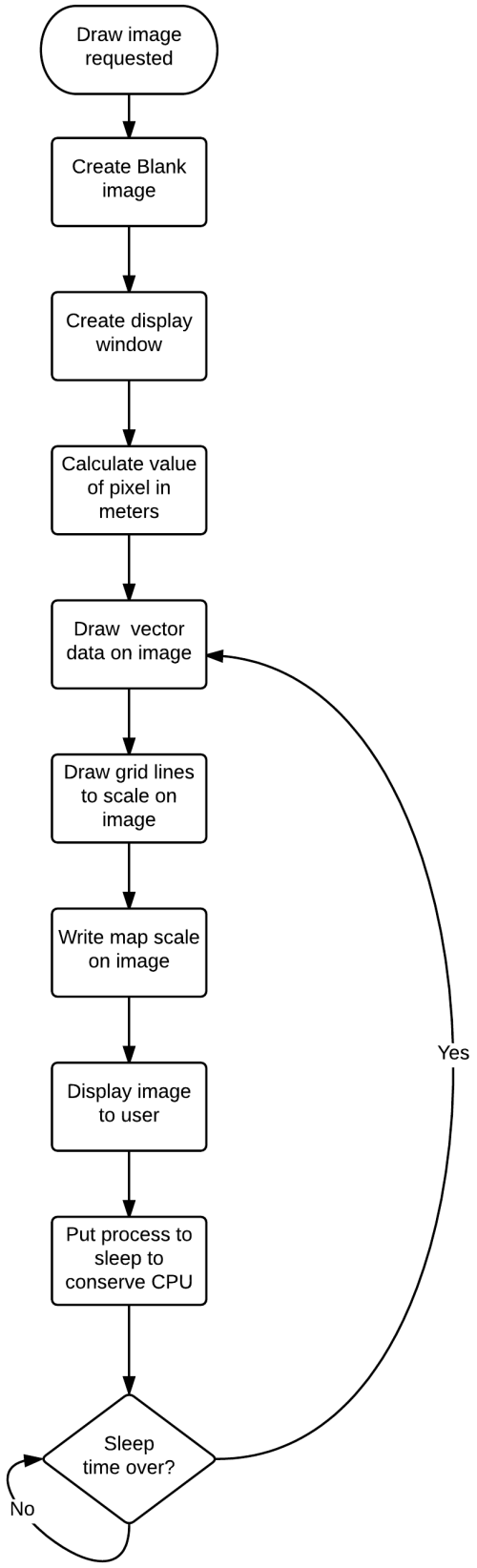


Fig. 13: Visual Display Flowchart

During creation of our first kinematic model we discovered that our wheel encoders didn't have the precision that we originally thought. We discovered that we had expected 64 encoder ticks per revolution, but were actually achieving 32. This means that any direct reading of the encoders could be off as much as .25 inch every reading. This error would accumulate over time and could cause significant error over time. Instead of attempting to directly read the wheel encoders we decided to capture the velocity commands being sent to the robot chassis and store a time stamp. We will then use this command and the time stamp to generate odometry information. This method makes a major assumption that the robot chassis will diligently executed and maintain this velocity. Equations 1 through 6 show how from just the left and right wheel velocities we can generate the required positional and velocity changes for the robot platform.

$$\dot{x_{[k]}} = \left(\frac{v_{left} + v_{right}}{2}\right) * cos(\theta_{[k-1]}) \tag{1}$$

$$\dot{y_{[k]}} = \left(\frac{v_{left} + v_{right}}{2}\right) * sin(\theta_{[k-1]}) \tag{2}$$

$$\dot{\theta_{[k]}} = \left(\frac{v_{right} + v_{left}}{radius * 2}\right) \tag{3}$$

$$x_{[k]} = x_{[k-1]} + \dot{x_{[k]}} * \Delta_T \tag{4}$$

$$y_{[k]} = y_{[k-1]} + \dot{y_{[k]}} * \Delta_T \tag{5}$$

$$\theta_{[k]} = \theta_{[k-1]} + \dot{\theta_{[k]}} * \Delta_T \tag{6}$$

The previously discussed method has proven to work well. In depth detailed comparison hasn't been established yet. During physical testing of our robot platform it was discovered that we could improve our encoder resolution from 32 ticks per revolution to 144 ticks per revolution. This is a considerable increase. In order to capitalize on this increase we elected to create a second kinematic model based upon the amount of distance traveled. This is beneficial because it allows us to create two separate models and choose which one we feel gives more reliable results. We used a different approach based upon distance traveled and not just the commands.

The following equations are derived from [11] and implemented in our C++ code.

$$v_{left} = \left(\frac{\Delta_{left}}{\Delta_T}\right) \tag{7}$$

$$v_{right} = \left(\frac{\Delta_{right}}{\Delta_T}\right) \tag{8}$$

$$\dot{x_{[k]}} = \left(\frac{v_{left} + v_{right}}{2}\right) * cos(\theta_{[k-1]}) \tag{9}$$

$$\dot{y_{[k]}} = \left(\frac{v_{left} + v_{right}}{2}\right) * sin(\theta_{[k-1]}) \tag{10}$$

$$\dot{\theta_{[k]}} = \left(\frac{v_{right} + v_{left}}{radius * 2}\right) \tag{11}$$

$$R = \left(\frac{radius * (\Delta_{left} + \Delta_{right})}{\Delta_{right} - \Delta_{left}}\right) \tag{12}$$

$$\omega = \left(\frac{\Delta_{right} + \Delta_{left}}{radius * 2}\right) \tag{13}$$

$$x_{[k]} = x_{[k-1]} - R * cos(\omega + \theta_{[k-1]}) + R * cos(\theta_{[k-1]}) \tag{14}$$

$$y_{[k]} = y_{[k-1]} - R * sin(\omega + \theta_{[k-1]}) + R * sin(\theta_{[k-1]}) \tag{15}$$

$$\theta_{[k]} = \theta_{[k-1]} + \omega \tag{16}$$

The major difference between the two kinematic models is the is that one is based upon calculating velocity information for each wheel and the other is based upon assuming that the robot executes its previously given command. We expect that the velocity command based model has more accurate twist information and that the encoder output based model has more accurate positional information.

We are actively monitoring which seems to produce better results. Due to the modular design of our system switching back and forth between the two sources is easily accomplished by just changing a few lines of code. An alternate, but more complicated, solution is to use the two models and filter them to get the best results possible, prior to plugging them into the EKF.

D. Collision Avoidance

Due to the fact that we ultimately envision our system to run autonomously, some type of collision avoidance algorithm must be implemented. Luckly as part of the *Eddie Bot* turn key solution we received from Parallax, five dual Ping/IR combo sensors were included with our purchase. The term combo basically refers to a fancy mount that holds both a Ping ultrasonic distance sensor and a Sharp IR sensor. This setup is designed to be dropped into our Eddie board. Since the Eddie board has slots for all five of the dual combo sensors we originally decided to use them as described from factory. However, during development we discovered that the IR sensors have a non-linear output and are subject to various environmental factors. For example the reading that our IR sensors output are different depending on the color of the object. This difference can be as much as 10% which makes utilizing the IR sensors for Collision Avoidance a very dangerous proposal. This is why we chose not to include the use of the IR sensors in our final design.

For the previously discussed reason we have chosen to only use the five ultrasonic ping sensors furthermore we discovered that the on board processing power could not simultaneously sample five ping sensors while sending and receiving serial drive commands. This created unwanted lag in our system so we opted to have an external controller dedicated to just the ping sensors this is how we arrived to the solution of a dedicated Atmega 328 microcontroller to control the ping sensors.

To ensure the validity of our ping data, we setup a small test bench were we performed two tests. For the first we placed the ping sensor 12 inches away from a book, for the second we moved the book to 28 inches. We then directly compared the result of each ping sensor to the actual measured distance. We did this for all five sensors and found that they were were accurate to $\pm\frac{1}{2}$ centimeters. After the individual testing we tested again with all the sensors connected and found that this time the sensor measurements were accurate to $\pm$ three centimeters. This was attributed to possible power fluctuations from the microcontroller power supply due to the increase of the sensors. Figure 14 shows how ping testing was conducted. For our prototype implementation we have averaged the distance that

the pings record with respect to every other ping sensor in order to return more accurate data.

Once the ping data is collected it placed into an array and sent to ROS node path planning via the microcontroller created topic collision avoidance. It is here that the actual collision avoidance takes place. The path planning node reads in the data and makes decisions based on how far away potential objects are. Path planning takes the read in data and decides how fast and how much to turn the robot depending on the its distance to the obstacle. The flowchart for the pings sensors is show in Figure 15

Fig. 14: Ping Testing

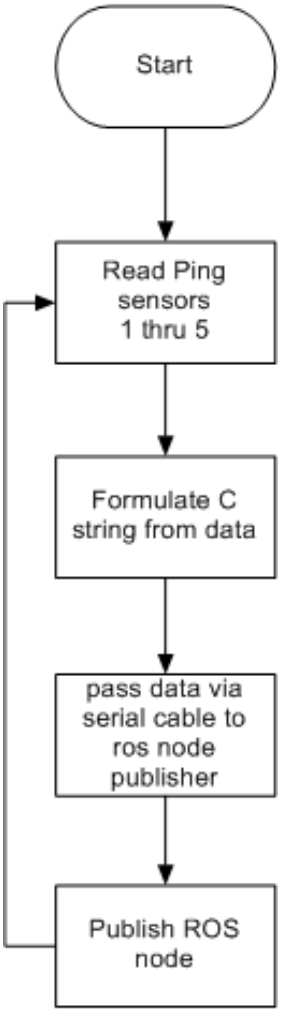


Fig. 15: Ping Sensor Flowchart

E. Gyroscope & Accelerometer

During our initial system development we were unsure of the type of vision system we would be using. We initially expected to be using a ranging solution or something to that effect. The gyro and and accelerometer where a design input from a faculty adviser in order to assist gathering odometry information.

The Gyroscope & Accelerometer were to help gather information regarding the angular velocities and angular position of our robot platform. The Gyroscope & Accelerometer are well-suited to perform this task, when initial filtering is performed via a complementary filter.

Due to our design choice of implementing a VO system we have began researching the side effects of removal of this feature from our system. The output of the Gyro & Accelerometer sub-system is the same as the output of the Kinematic Model and VO sub-systems. This means that we have three sources that all provide estimates of the same information. By keeping the Gyroscope & Accelerometer in our sub-system we add an increased level of complexity and don't expect to see much gain. Unless compelling reasons can be find we intend to request removal of this sub-system via change order to be submitted in February.

F. Path Planning

The path planning program has gone through several revisions through its development cycle. Initially, the visual odometry program had very specific non-holonomic limitations of how the robot could move without corrupting the visual odometry data. During testing, there were certain driving conditions that caused the visual odometry to rapidly interpret bad map data. This originally included when the robot performed a hard fast angular turn, when the forward velocity dropped below a certain threshold, or when the robot stopped or rotated on its axis.

Initially, the original exploration algorithm was implemented inside the path planing program. However, during the evolution of the visual odometry system, an alternate VO approach was used that relied on pure rotation and forward movement only. This resulted in a separate new exploration program that runs as its own node. This external program is automatically disabled when the path planning process is triggered and then the driving algorithm takes place inside the path planing program.

The path planning program receives data from 4 primary sources: Robot x,y,theta position data, ping sensor distance data, facial recognition x,y, ”who?” data and an ”explore” flag for the SLAM process to begin. See Figure 16 16 for a system flowchart, including incoming data. The path planning program uses the robot’s estimated position data and uses scaling to convert the distance the robot has moved in meters in the x and y axis directions to a grid map that keeps track of where the robot is and where it has been, along with unexplored area. Figure 17 17 shows the process for receiving and storing position data, as well as map generation. The testing environment was predetermined not to exceed 6 meters and the desired size of each grid is 0.2 meters. This was decided based on the robot chassis being 0.43 meters across. The map created also utilizes the ping sensor distance data to project free unobstructed space away from the robot up to 0.6 meters. Figure 18 18 shows the flowchart for incoming ping distance data processing. This allows for more squares of the of the environment to acknowledged as ”free space” that has been explored without the robot physically driving it. Finally, the facial recognition data is triggered by the camera software program seeing the face and predicting the x,y distances from the robots current position. It is assumed that if the camera can clearly see the detected face, then there are no obstructions between the robot and the goal. In this situation, depending on the angular orientation of the robot, all of the squares between the goal and the robot are filled in on the map.

The path planning algorithm, shown in Figure 19 19, is based on a depth first search pattern. During testing, a single search using a clockwise pattern starting at the 12 o’clock position plotted a very inefficient path to the detected object. This led to using 8 different variants of depth first search to calculate the most efficient path from point to point. The grid squares of the route of each independent goal to goal are stored separately and in the end, combined into the final driving path of the entire journey. In addition, an acceptable measure of what constitutes a successful ”robot has arrived at goal” point must be established. This goal was set to 40 centimeters or 1 robot width. This ensures the collision avoidance system can still properly function while getting goals in case the environment has any unexpected changes.

The path planning program must be programmed with a few commands to determine the type of operation. First, there must be an ”explore” mode, where the robot automatically roams around the room with a preset path plan and performs collision avoidance during travel. Next, there must be a ”find object” command that allows the user to choose an object at a known set of coordinates, and have the robot go to that object - still performing collision avoidance and obeying the non-holonomic constraints of the system as to not cause a software failure during goal seeking. And finally, there must be a remote stop command to terminate the program. There is an ”explore” button on the control script that begins the SLAM program. The path planning program can be preset to start after detecting 1 to 4 faces before automatically performing the path planning process, or the ”fetch” button on the control script can be pressed any time to begin the process. With no goals detected, the robot simply goes to the starting home point.

G. Visual Odometry Interface

One of the main design requirements of our project was to be able to control a mobile robot by using a camera as a sensor. Cameras can be used to control robots in a multitude of ways. We chose to implement a system that utilizes the camera to produce localization data. This is commonly known as VO.

VO is often used to compensate for imprecision’s on wheel encoder systems. When working with wheel encoders on mobile robotics, there is no guarantee that the robot traveled the distance that the wheel encoders measured. If the wheel spun out or slipped, the encoders still counted motion, which makes the robot think that its platform has traveled, when it actually hasn’t. VO is used to help fix this localization problem. While the VO sampling rate

is often much slower than the wheel encoder sampling rate, it can help alleviate the wheel slippage problem. When 100% wheel slippage occurs in a visual odometry system, no motion is measured by the camera because the camera did not see any change. Because of this, there is no motion output from the odometry system, which drastically helps overall control systems in compensating for robots that operate in these high slippage environments.

Our robotic platform needs to be able to carry around a camera that will be used to perform VO. The specific system is unimportant, our design needs to be able to have a module for VO to be plugged into. This modularity is important to our project, as it will allow us to replace any given set of VO libraries for another. One goal is, given enough time, be able to test out several VO libraries, chose the best option for our system, and produce recommendations for future robotics groups.

FMC Technologies Schillings Robotics requested that we implement a monocular VO system, which has turned out to be one of the main limitations of our project. This greatly limits our options in the open source VO implementations and in general, complicates the system. Unless mathematical assumptions are made, it is impossible to gather scale from a monocular system. This can be a very massive issue, as the system can tell you that you've driven in a direction, but not how far in that direction. While still usable, plugging this data into control loops results in generally poor control systems.

The current library we have chosen for our VO is libviso2, written by Andreas Geiger. The algorithm in this section is abstracted from his paper [3]. Libviso2 was written primarily for use on autonomous vehicles. There are two separate implementations: the monocular and stereo cases. The stereo case has no limitations, while the monocular algorithm has many.

The monocular library assigns a scale to the odometry data, but it does so by making some mathematical assumptions. First it assumes that the camera is at a fixed height and angle, and that the camera never deviates from these initial parameters. Knowing these values allows us to attach a scale to the output data. The algorithm performs the following steps [3]:

1) Using random sampling techniques, estimate the F (fundamental) matrix through an 8-point algorithm
2) Compute the E (essential) matrix using calibration data
3) Compute 3D points and rotation and translation
4) Estimate ground plane in 3D points
5) Use known camera height and pitch to scale rotation and translation
6) Capture new frame

Because libviso2 is a drop in solution to the visual odometry subsystem, we are mainly concerned on how it fits into our system rather than the intricate implementation. The overall flow chart for the VO signal path can be seen in Figure 20. As shown, the VO interfaces directly with the Camera, and the EKF. The EKF that our system uses is VO library independent. Any library that can produce a ROS odometry message type can be used.

This library allows us the unique luxury of getting scaled data. Other monocular implementations do not necessarily give us this luxury. However, because this library allows us to gather scaled data trough only camera frames, some mathematical assumptions have to be made. Because of these assumptions, we assume values for 3 degrees of freedom corresponding to the camera pose. This limits our ability to only be able to extract 3 degrees of freedom data from our camera system. In order to ensure the system provides proper output these mathematical assumptions become physical measurements and mounting requirements of the camera's position. These measurements provide the vertical distance from ground and the pitch of the cameras viewing angle. The third physical requirement is that the camera can not have any play to roll. The measurements are stored in a file that is loaded into libviso2 at run time.

One of the inherent problems with the libviso2 library is the fact that it cannot handle data from the camera if the data gathered was a result of pure rotation about its non fixed axis. If pure rotation data is inputted then the fundamental matrix cannot be trusted. This is due to a common problem in fundamental matrices known as matrix degeneration. This can happen when the fundamental matrix fails to find a unique solution to the epipolar transform between image frames [12].

While implementing the library was incredibly fast and easy thanks to pre-written ROS nodes found

online[8], this library has consumed a lot of time in testing and debugging. Initial tests seemed like the library was producing usable results, but we needed a way to view the data. Once our visual display was implemented, we noticed that the data was far from perfect. A lot of testing and research went into tweaking the many parameters of our system to optimize for the VO data. While the incoming data has improved over the past few weeks, the continued use of libviso2 is putting many difficult constraints on our overall system. We are actively trying to refine the VO system and find the constraints of libviso2 so that we can implement path planning methods that can work around them.

H. Probabilistic Filtering

The overall purpose of this sub-system is to minimize the effects of noise or error in the sensor readings and fuse the data from multiple sources into a single output. In the field robotics there are many options when considering using a complex filter to fuse sensor data from multiple sources. The two main types can be categorized as particle filters and Gaussian based filters. An in-depth explanation of the differences is beyond the scope of this paper. Instead we will focus on the filter we chose and why we chose it, as well as the implementation of our filter choice.

We spent considerable time researching filters during our initial system development. We elected to utilize a probabilistic filter based upon the EKF. The main reasons we chose to utilize an EKF was that it has a proven history of being successfully implemented in systems that are similar to the one we are attempting to build. The EKF also has many open source implementations and many detailed explanations that allow utilizing an EKF to be much easier than some of the existing systems. There are fancier forms of the Kalman filter such as the unscented variety, but they are newer and more difficult to implement.

We elected to find a existing EKF instead of programming one from scratch. This will allow us to focus on the usage of the filter and not the complex mathematics that make it work. The particular versions we found to was the EKF from the robot_pose_ekf from the ROS open source libraries. This particular realization of the EKF is the backbone of other SLAM algorithms and has a long history of proven success. One of the biggest factors in determining whether or not to utilize this particular software was that it is heavily documented. It is designed to take inputs from VO, wheel odometry, and IMU input. It will then fuse this data and publish the outputs.

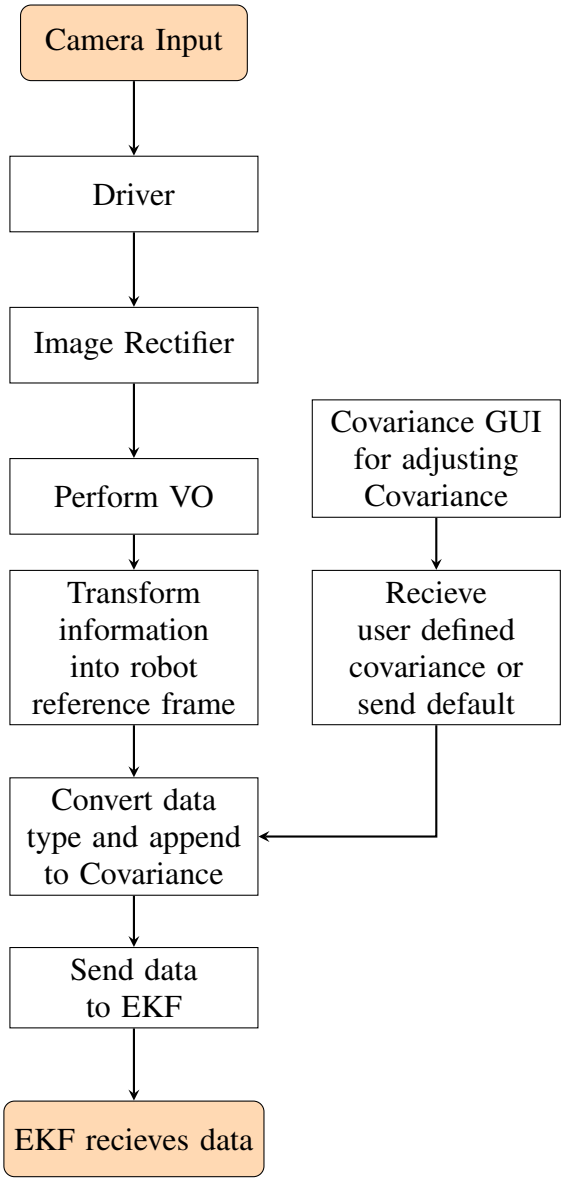


Fig. 20: VO Signal Path

The first step in setting up to utilize the filter node was to figure out the particular data types and topic names in subscribes to. By using the inputs as our starting point it allowed us to figure out what pre-processing and conversion needed to be performed on the individual data sets prior to publishing them to the filter. It was during this initial step that we

[8]http://wiki.ros.org/viso2_ros

discovered that we were going to need to capture all data sources and append our estimate of the covariance to it.

We created a C++ class that captures the output of the kinematic calculations and VO subsystem. The kinematic calculations occur inside of the same class and do not need to come from external input. The output is stored in the previously discussed vectors where it waits until time to be published to the robot_pose_ekf node.

The slower input is the VO so when the output of that system occurs it sets a Boolean variable that then causes the VO and kinematic odometry information to have the covariance attached to it and published to the EKF. The covariance adjustment for all of our odometry models is critical to the overall output of the filter. If the covariance values are incorrect then the output from the filter can be completely incorrect. For the duration

When the robot_pose_ekf node is done processing the data the same C++ class subscribes to the topic and stores the filter output for usage by the other nodes. The process happens rather quickly compared to the VO processing time. The output is then plotted on our visual display for visual comparison to the other sources.

We have currently fused the two kinematic models and have successfully implemented filtering of VO and wheel encoder data. Our sponsors have indicated that they would like to be able to have the vision sensor be the dominant sensor, but we have been unable to tune our filter for this. This is mostly due to the current status of our VO node. We are aware of the difficulty that tuning an EKF and are ready for the task.

During our initial testing we were forced to recompile our nodes every time we adjusted the covariance. This was time consuming and created a very slow tuning environment. As a way to help us expedite our tuning we have created Graphical User Interfaces in python to help us tune the covariance of the inputs during run-time. This should greatly increase our tuning and debugging time of the filter.

The GUI allows us to monitor the output of our data sources as well as update our estimate of the covariance. The overall design was intended to be functional and not for presentation during our end of project presentation. The node that contains this GUI is written in python QT4. This same node also allows the user to update a text file that is read

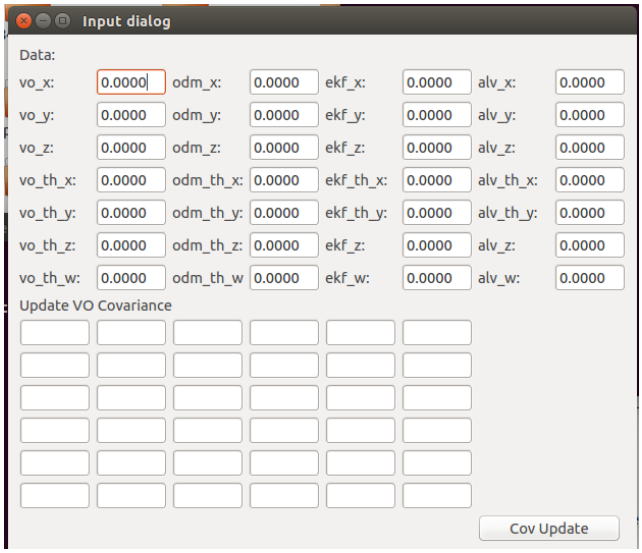


Fig. 21: Covariance Adjustment GUI

during run time that contains the default covariance values for all data sources. The file is updated each time the user sends a new covariance update. The current EKF C++ code doesn't access this file directly it subscribes to the covariance updates via ROS topics. This allows a user three options to adjust the default covariance: adjust in the C++ pre-filter code, adjust the text file, or dynamically change the value in the GUI.

We feel that we are in a strong position to finish the filter tuning in a timely manner. The initial testing of the overall system seems to deliver within the required accuracy of 30% and we are on track to completing this feature. We spent many hours teaching us the fundamentals of the EKF as well as understanding the implementation and limitations of the robot_pose_ekf node.

IX. Mechanical Drawings and Documentation

The mechanical portion of this project is limited to the robot chassis. The robot structure consists of a top and bottom deck pair, and an undercarriage for holding the batteries. This undercarriage serves to prevent any acid spray should a battery burst and also isolates the batteries from electronic components on the decks above.

The robot was assembled per the instructions in Parallax's assembly documentation. These instructions have been included in the Appendix.

X. Test plan for Hardware

The majority of hardware we are using is commercially available off the shelf hardware. Due to the nature of the software requirements we elected to spend very little time on hardware testing. If the hardware fails during software testing we elected to diagnose those errors on the spot.

We had originally hoped that by purchasing an expensive ”turn key” robot that we would avoid the majority of issues that could arise from such a complicated piece of hardware. This turned out to be a very flawed plan.

The fact is that the one of the single biggest issues we have had to overcome is our robotic chassis. It has failed for us repeatedly and we have had to replace three sets of wheel encoders.

XI. Test plan for Software

Our design is largely software and our testing philosophy will be to test each individual feature's software independently of the other sections. After all features have been verified our objectives will be to test the entire system as a whole. In this document, we will present the testing plans for our specific features in Section XI-A, and the testing plan for our system integration in Section XI-B.

A. Feature Testing Plan

Our project consists of seven features. These features are the Kinematic Model, Path Planning, Visual Display, Filtering, Collision Avoidance, Serial Communication, and the Visual Odometry Interface. Each feature has its own unique testing plan as described below. These feature testing plans are slightly different than you'd see in normal system testing, as we are dealing primarily with custom written event driven software which leads to dynamic errors. With event driven software the amount of time and amount of start/stop cycles are critical to detect run time errors. The longer we test, the more likely unplanned run time errors will surface.

1) Kinematic Model: The Kinematic Model feature is software that gathers information from the robots wheel encoders and uses that data to calculate an estimate of the robots position. This feature is realized in C++ and as such we need to test this feature by utilizing its existing infrastructure. As this feature is dependent upon other features we will be testing this feature in three separate phases.

a) Phase One: This phase will be to run simulations by creating a node that simulates the data that is published from the wheel encoders via the Serial Communication feature. By using known velocity controls we can precisely calculate where our robot should be and verify the output of the feature is correct. This Phase will test the following elements of our system.

- The mathematical accuracy of our kinematic calculations.
- The features theoretical ability to communicate with other features via ROS.

b) Phase Two: This phase will be to test the accuracy of the wheel encoders which are the inputs to the wheel encoders. Although wheel encoder data isn't specifically mentioned in the feature description it is essential to have a gauge of the accuracy of the data being inputted into the feature during actual run time. The testing method will be very simple. We will send commands to the robots chassis to have its wheels spin one complete rotation. By measuring the wheels distance move and comparing it the output of the wheel encoders we can gauge if there are errors in the wheel encoder inputs. This Phase will test the following elements of our system.

- Verify the precision of our encoders.
- Verify accuracy of incoming data to the kinematic model.

c) Phase Three: The third Phase will involve live testing of the feature inside the desired system. It is critical that the Serial Communication and Visual Display features are tested prior to beginning this phase of testing. This phase will involve sending commands to the robots chassis and comparing the output of this feature to the physical ground truth of the chassis. This Phase will test the following elements of our system.

- The physical accuracy of our kinematic calculations.
- The features physical ability to communicate with other features via ROS.
- The reliability of the feature when used with other independent features.

2) Path Planning: The path planning node is software written in C++. It accepts the outputs from the EKF, collision avoidance, and goal detection software. It will incorporate local data to create a map of it working environment. The path planning feature will need to have two distinct tasks. The

first task will be to explore its environment. When it believes it has explored its environment its next task will be to navigate to various detected goals throughout the room.

a) Phase One: The first phase of testing will be to ensure that the path planning feature has the ability to completely explore its environment. This test will be performed by placing the robot in a safe workspace. The workspace should be between 3 - 5 meters. Three obstacles will be placed in the room. The obstacles will have a circumference between 12 - 30 cm and a height between 12 cm - 30 cm. This will ensure that the collision avoidance feature can see the objects. The robot will be placed at a predetermined starting point and an explore command will be sent from the control GUI. The path planning node will be required to navigate throughout the room until it has explored all spaces without colliding with any obstacles or getting stuck in corners. This Phase will test the following elements of our system.

- The ability of the path planning node to control wheel velocities.
- The ability of the feature to navigate and explore its environment.

b) Phase Two: This phase of testing will be to ensure that the path planning feature can collect data from the collision avoidance feature as well as the goal detection feature. The path planning feature should be able to successfully interpret that data and store it. This test will be performed by broadcasting messages from the collision avoidance feature as well as the goal detection feature. Testing if the path planning node is receiving the data can be performed by printing the received data from the data structure in which it is stored. This Phase will test the following elements of our system.

- The ability of the feature to store values from collision avoidance and goal detection.

c) Phase Three: This phase will test the ability of the path planning node to go to two predetermined goals. This test will ensure that the path planning node has the ability to navigate to various preset destinations. This test can be accomplished by using the goal detection feature or by publishing artificial goals. When the GOTO button is pressed on the control GUI the path planning node should begin to navigate towards the first goal and then the second goal. The path doesn't need to be the optimal path, but the robot should not need to re-explore the environment to find that location. This Phase will test the following elements of our system.

- The ability of the feature to allow the robot to navigate to a specific desired locations.

3) Visual Display: The visual Display captures data from the kinematic model, IMU, EKF, and Visual Odometry features. It then displays this data for a user. This feature will be implemented in C++. The accuracy of the visual display is important because it will be used in debugging of the system. In particular if there are scaling issues it could result in improper tuning of the EKF which will result in a deviation from desired result. Testing of this feature will be in two phases.

a) Phase One: Phase one will be to test the ability of this features software to capture data from all sources and display it as it changes on the display. To perform this phase of testing a separate program will be written to publish data on the specific topic that each input will use. This program will be interfacing with a lot of event triggered functions so testing the input and output is essential to ensure proper display. This phase will test the following elements of this feature.

- The ability of the program to reliably receive data from multiple sources.
- The ability of the program to convert received data into a visual map.

b) Phase Two: This phase of testing will be to help gauge the accuracy and precision of the visual display. A program will be created to drive the robot for a preset distance. After the robot has travelled that distance we will measure the ground truth distance travelled and compare it to the visual display. As this will involve using a floating point scale to assign a distance to each pixel, we will need to be cautious of rounding error as our environment grows. We well be testing the visual display from distances between 0 to 6 meters for accuracy. We require that the map be at least 10. This phase will test the following elements of this feature.

- The ability of the visual display to produce accurate map data.

4) Filtering: The Filtering feature is a combination of third party and custom a C++ program. The overall structure of the filtering feature is to capture data from the vision feature as well as the kinematic model. We will then convert this data to

an appropriate data type and when ready publish the data to the third party EKF. The actual EKF is 3rd party software and will need to be tested thoroughly. We will test this feature in two phases.

a) Phase One: The first phase of testing will be to test the pre-filter software that we have written. This software is designed to accept data from the previously listed sources and the convert it to the required data type that the EKF allows. This is imperative so that there is no data type mismatches during run time. The software involves multiple event triggered call back functions so testing to ensure that each thread processes properly is essential to the reliability of the overall system. To perform this test data from the various sources will be published and the pre-filter software will have to capture, convert, and publish correct output. By populating the incoming data type with preset values we can observe that the output of the conversion algorithm correctly converts it to the appropriate data type. This Phase will test the following elements of our system.

- The ability of the pre-filter to reliably accept data from multiple sources.
- The reliability of the data conversion algorithms.

b) Phase Two: During the second phase of testing we will be testing the third party EKF software. This test will be performed by using post processed data to ensure that it is as close to real situations as possible. We will run the robot in a circle and record the output of the Visual Odometry and Kinematic features. The accuracy of the two sources isn't really an issue as long as they are being published on the same scale the EKF should be able to filter out the error and produce some type of usable data. Tuning of the EKF filter will be done via our Covariance adjust GUI. This phase will test the following elements of this feature. This Phase will test the following elements of our system.

- The ability of the third party EKF software to accept data at our transmission rate.
- The ability of the covariance GUI to adjust the covariance of multiple data sources.

5) Collision Avoidance: The Collision Avoidance software consists of ultra-sonic sensors, micro-controller and software, serial data transfer, and software on the PC to interpret the data to prevent collisions. The Path Planning feature should provide guidance based upon this features data, but this feature should contain a procedure that stops the robot if a collision is imminent.

a) Phase One: The first phase will be to test the data coming out of the ultra-sonic sensors. This test will also test the ability of the micro-controller to interpret data that is received by the sensors. The test will be to place an object in front of each sensor and see the distance measured by the sensors. This distance needs to be less than 3cm per meter, unless the distance is less than a meter in which case +- 3 cm is within acceptable standards. This Phase will test the following elements of our system.

- The accuracy of the sensors.
- The ability for the micro-controller to interpret the data being provided by the sensors.

b) Phase Two: The second Phase will be to test the ability of the micro-controller to provide data to the software in the PC. This will involve third party software for the serial communication and not the robot serial communication feature. For this phase of testing we will use the same type of test as phase one, except that we will be broadcasting the data to the software in the PC. The PC software only needs to display the output from the micro-controller. If the accuracy of the sensors is the same as the value measured in phase one then it is a safe assumption that there isn't any data corruption due to the third party software. This Phase will test the following elements of our system.

- The accuracy of the data after serial transmission.
- The reliability for the micro-controller to interpret the data being provided by the sensors.
- The reliability of the third party serial connection software.

c) Phase Three: The third phase will be to create PC software to use the data that is being transmitted to stop the robot. We are going to send velocity commands to the robot that will set it on a path to collide with a wall. The PC software that implements the collision avoidance must stop the robot from crashing into the wall. There must be little to no lag between when the stop point of 30 cm is passed and when the robot begins deceleration. This phase will test the following elements of this feature.

- The ability of the PC Software to prevent collisions.

6) Serial Communication: The Serial Communication model will be used to send data to and from the robots chassis control board and the laptop. The data being transmitted will be converted in the is node to allow decimal communication instead of hex, as the board requires. The feature will be realized in the laptop as C++ code and hardware connection will be with a standard USB cable. For the purposes of this section input will refer to data coming from the robot control board into the laptop and output will refer to data being sent from the laptop to the robot control board. This feature involves event triggered programming techniques, so we will be testing in 2 number of phases.

a) Phase One: A key element of the feature is its ability to convert data. Specifically the Robot Control Board requires that numerical values be transmitted in signed Hex stored in a standard C-string. This is not a standard form of numerical notation and is very hard for a human or C++ program to interpret the data. This feature will handle bidirectional conversion between the decimal and hex c-strings. This Phase will test the following elements of our system.

- The reliability and accuracy of our conversion algorithm.

b) Phase Two: The second stage of testing for this feature will be reliability of the overall connection between the two devices. This test will involve sending data from a testing node and having the serial communication software transmit it via the USB to the robot control board. The program should close the serial connection and reopen it at least 500 times to verify that the connection can be re-established without error. This phase should also have a time test that tests leaving the communication on for a very long time, such as 3 hours. This Phase will test the following elements of our system.

- The reliability of the connection and interface.
- The repeatability of the connection and interface.

7) Visual Odometry Interface: The visual odometry feature will accept images, process them, and output odometry data. The feature will be implemented in C++ or third party software. It is important that a precise measure of this feature is understood as our sponsors are very interested in this feature. The environment which these tests will be performed needs to be controlled and stale as the VO calculations will assume that only the camera is moving. This feature will be tested in three phases.

a) Phase One: The first stage of testing will be this features ability to accept an image and output unscaled Rotation and Translation matrix. This is the first stage of processing for the visual odometry system and it is imperative that this first step is understood and tested thoroughly. It is specific to note that during this phase of testing that run-time errors are understood, such as segmentation faults. When errors are found the specific run-time error should be repaired and then the testing cycle should start again completely. For example if 300 hours of testing are to be performed for the phase of testing to be completed and an error occurs at hour 290 the timer must be reset. To perform testing for this phase start the VO node and observe the output of the system for a certain number of time. By varying the cameras location new images will be produced which will help identify run time errors. By moving the camera left and right the tester should observe that the Rotation and Translation matrix change with the movement. The tester should keep in mind that the data is unscaled and will be dependent on the cameras current environment. This Phase will test the following elements of our system.

- The reliability of the features ability to accept and process images.
- The ability of the feature to produce consistent data.

b) Phase Two: The second stage is to correlate the data into real world coordinate axis. This can be accomplished via external calibration standards such as an IMU. The goal of this test is to ensure that when the translation matrix states that camera has moved forward in a specific direction that the robot has actually moved in that specific direction. By ensuring proper coordinate axis configuration we can help prevent scaling and debugging errors. This Phase will test the following elements of our system.

- The ability of the features ability to produce data on the appropriate axis.

c) Phase Three: The third phase of testing will be to test the ability of this feature to produce data that can be used by the control algorithms. We are currently unsure of how the control system will produce data. Testing for this phase will be re-evaluated upon completion of Phase Two of this section. We believe that it is critical to perform small

scale control testing of the data produced by this node prior to beginning system level testing.

B. System Level Testing

After completion of all feature testing it will be time to begin testing the system as a whole. This will mean that each feature is fully ready to be integrated into our overall control algorithm. It is our hope that by thoroughly testing and understanding our software prior to entire system testing that we will better understand our overall system and have be able to troubleshoot it.

d) Phase One: During this phase of testing we will attempt to launch all programs and be especially attentive to run-time errors. Our goal for this testing is course debugging of how the software interacts with each other as a hole. We need to ensure that the proper data is being published at the proper time. During this phase of testing our path planning, EKF, and VO will probably not be tuned properly. During this initial phase of testing we need to ensure the proper data is being published to each program. If errors should be monitored, but if possible allow the robot to attempt its mission. By allowing the robot to run a little longer after the first error is seen we can see how the error has effected other elements of the system or if other non-related errors have occurred. By carefully documenting the errors during this time we can gauge if the overall test is working properly. This testing should occur in the robots environment that it will be tested on during end of semester demonstration day.

e) Phase Two: During Phase two we will be working on fine tuning the system. Items such as tweaking the EKF, VO, and path planning nodes will be performed during this phase. By taking time to thoroughly tune our system we can develop a stronger understanding of the overall system. The tester should get enough data to make a decision as to if major overhaul of a feature is required or not. This phase of testing shouldn't be considered completed until every goal in the design idea contract is satisfied. After phase three it will be time to qualitatively evaluate the system as a whole. The key elements are how accurate each of the Kinematic Model, VO, IMU, and EKF outputs are compared to ground truth. This data is essential to gauging if the overall project is a success.

f) Phase Three: The main focus during Phase three will be to test the system exactly as it will be on demonstration day. Every effort should be made to test at the actual site and at the same time of day. By repeatably testing on sight in the same environment we should be able to get an idea of the reliability of the overall robotics system. If Phase two has passed then there shouldn't be any major changes made. If any changes are made they need to be carefully documented and all members of the team need to be aware.

1) Software Testing Results: Feature requirements successfully tested and meet the requirement of the Design Idea Contract. All features successfully integrated together and passed all phases of testing.

XII. Conclusion

This document has described our development towards an autonomous mobile robotics system that can localize based upon external sensor data to be used in hazardous environments. This robotics system implemented on a suitable robotics platform could be used to help reduce risk of injury to members of our society who put themselves in harms way on a routine basis. As members of the engineering community it is our responsibility to use our skills and knowledge to better society and we feel that our system can help advance research towards that goal. By developing a concrete and reliable SLAM algorithm it allows a robotics system to be utilized in dangerous situations with the desired goal of replacing humans in harms way with machines.

References

[1] K. Nagatani, S. Tachibana, M. Sofne, and Y. Tanaka, "Improvement of odometry for omnidirectional vehicle using optical flow information," in *Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, vol. 1, 2000, pp. 468–473 vol.1.

[2] D. Helmick, Y. Cheng, D. Clouse, L. Matthies, and S. Roumeliotis, "Path following using visual odometry for a mars rover in high-slip environments," in *Aerospace Conference, 2004. Proceedings. 2004 IEEE*, vol. 2, March 2004, pp. 772–789 Vol.2.

[3] A. Geiger, J. Ziegler, and C. Stiller, "Stereoscan: Dense 3d reconstruction in real-time," in *Intelligent Vehicles Symposium (IV)*, 2011.

[4] S. Choi, J. Park, and W. Yu, "Resolving scale ambiguity for monocular visual odometry," in *Ubiquitous Robots and Ambient Intelligence (URAI), 2013 10th International Conference on*, Oct 2013, pp. 604–608.

[5] J. Campbell, R. Sukthankar, I. Nourbakhsh, and A. Pahwa, "A robust visual odometry and precipice detection system using consumer-grade monocular vision," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, April 2005, pp. 3421–3427.

[6] M. Liu, S. Huang, and G. Dissanayake, "Feature based slam using laser sensor data with maximized information usage," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 1811–1816.

[7] M. Tatar, C. Popovici, D. Mandru, I. Ardelean, and A. Plesa, "Design and development of an autonomous omni-directional mobile robot with mecanum wheels," in *Automation, Quality and Testing, Robotics, 2014 IEEE International Conference on*, May 2014, pp. 1–6.

[8] J. Marck, A. Mohamoud, E. v.d.Houwen, and R. van Heijster, "Indoor radar slam a radar application for vision and gps denied environments," in *Microwave Conference (EuMC), 2013 European*, Oct 2013, pp. 1783–1786.

[9] B. Williams and I. Reid, "On combining visual slam and visual odometry," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, May 2010, pp. 3494–3500.

[10] S. Cousins, "Exponential growth of ros [ros topics]," *Robotics Automation Magazine, IEEE*, vol. 18, no. 1, pp. 19–20, March 2011.

[11] G. Lucas. (2001) A tutorial and elementary trajectory model for the differential steering system of robot wheel actuators. [Online]. Available: http://rossum.sourceforge.net/papers/DiffSteer/

[12] P. Torr, A. Zisserman, and S. Maybank, "Robust detection of degenerate configurations for the fundamental matrix," in *Computer Vision, 1995. Proceedings., Fifth International Conference on*, Jun 1995, pp. 1037–1042.

[13] [Online]. Available: http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html

[14] "Arduino ide setup for rosserial." [Online]. Available: http://wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup

GLOSSARY

EKF
: The Extended Kalman Filter is a probabilistic filter used to help remove noise from data. 4, 8, 20, 22, 25–27

FPS
: The rate at which frames are displayed. 16

fundamental matrix
: The fundamental matrix contains the data corresponding to the epipolar geometry between two image frames. 25

IMU
: An Inertial Measurement Unit is a sensor device that measures current pose data. It is often configured by sensor fusion of a gyroscope and an accelerometer.. 5, 7, 8, 26

odometry
: A representation of a robot's position based upon measured or calculated linear and angular positions and velocities. 3, 6, 9, 20, 25, 33

ROS
: The Robotic Operating System was created by Willow Garage, and is a software framework for use in robotic development. 6, 13, 17, 18, 20, 23, 26–28

SLAM
: Simultaneous Localization and Mapping is the process of producing a map using visible landmarks and localizing with respect to that map at the same time. 2–4, 26, 32

VO
: Visual Odometry (VO) is a process that generates odometry data through careful analysis of image sequences generated by a moving camera. iii, 3, 5, 6, 8–10, 23–27

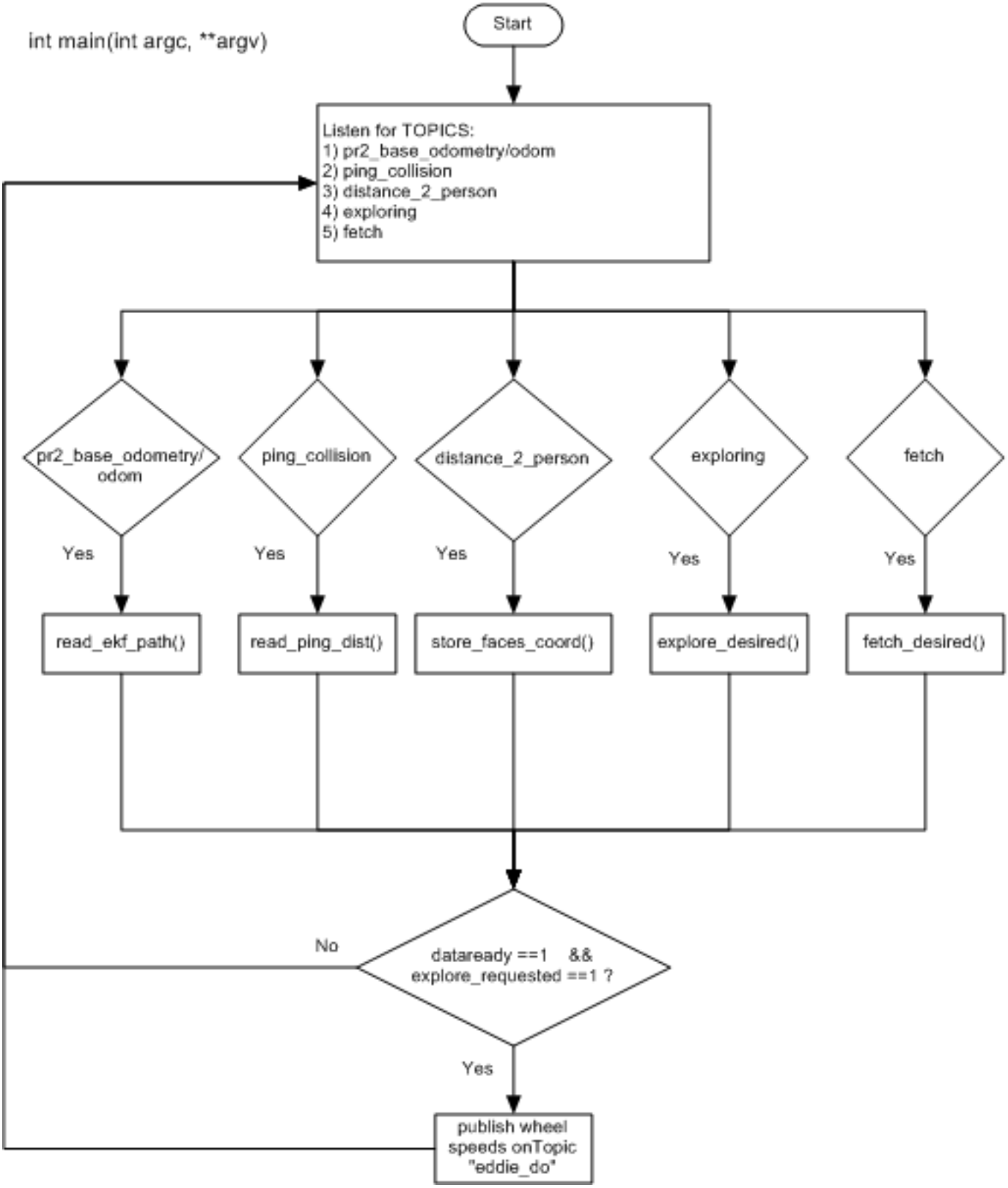


Fig. 16: Path Planning System Flowchart

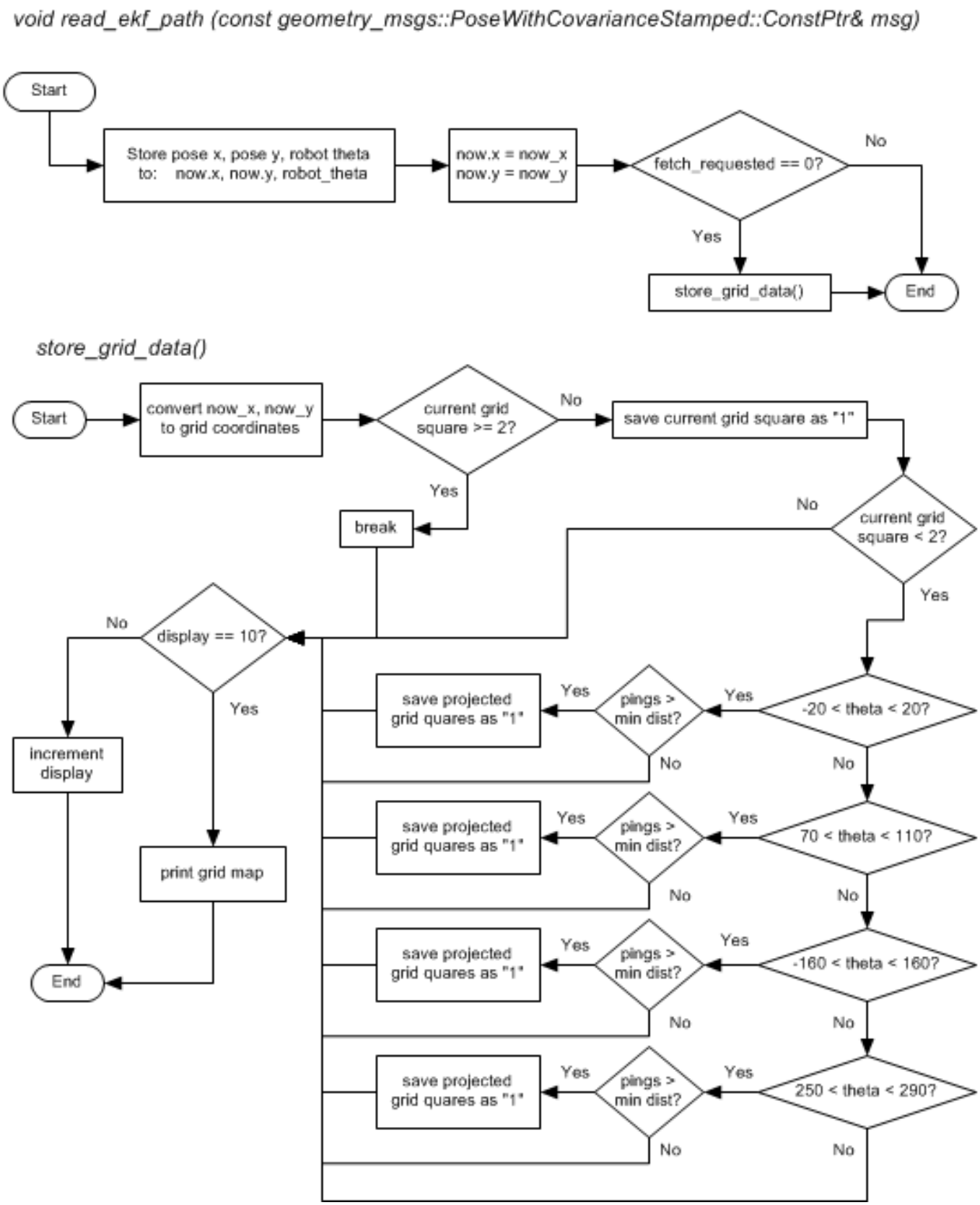


Fig. 17: Robot Position & Create Map Flowchart

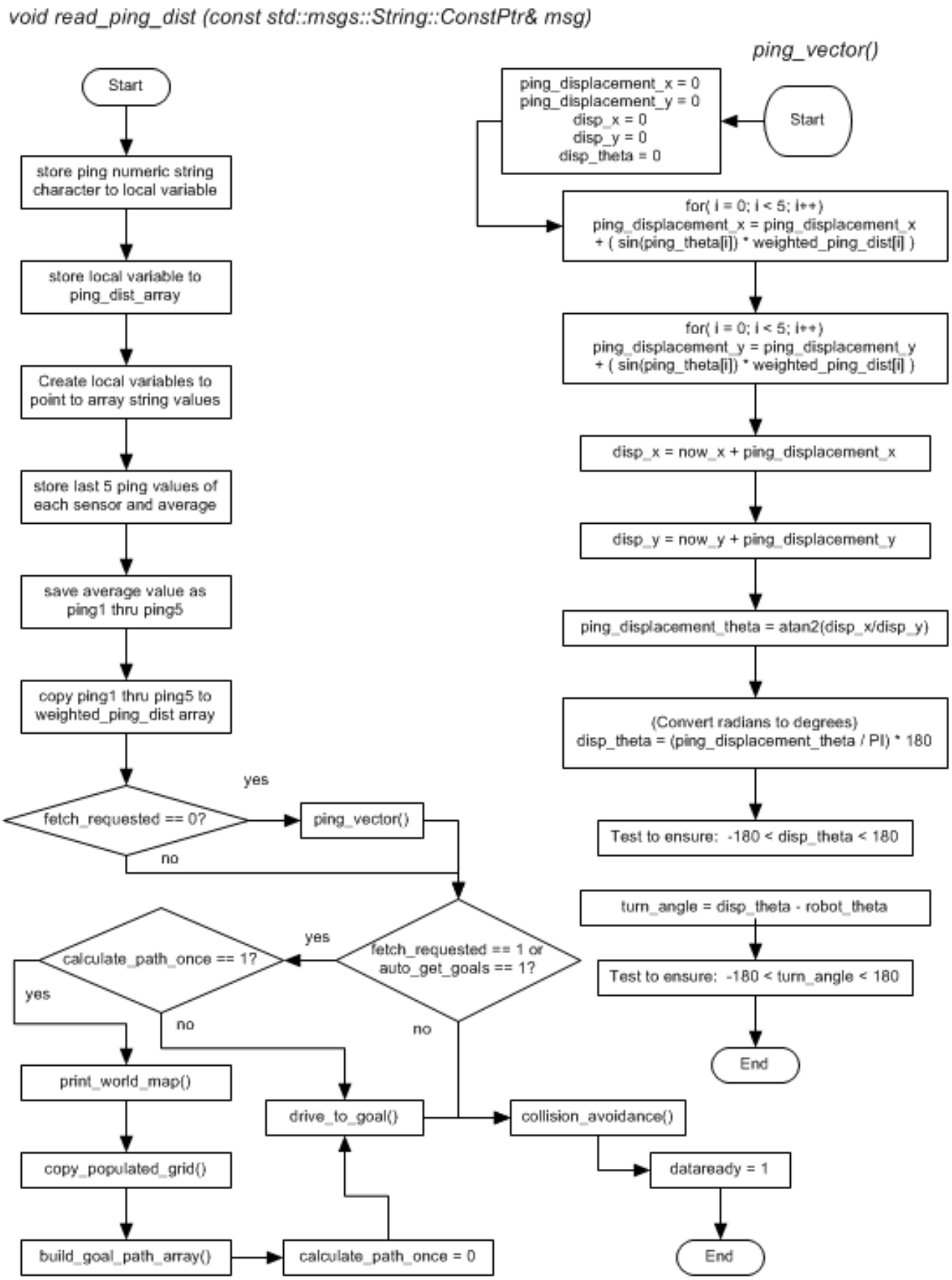


Fig. 18: Read Ping Distance Flowchart

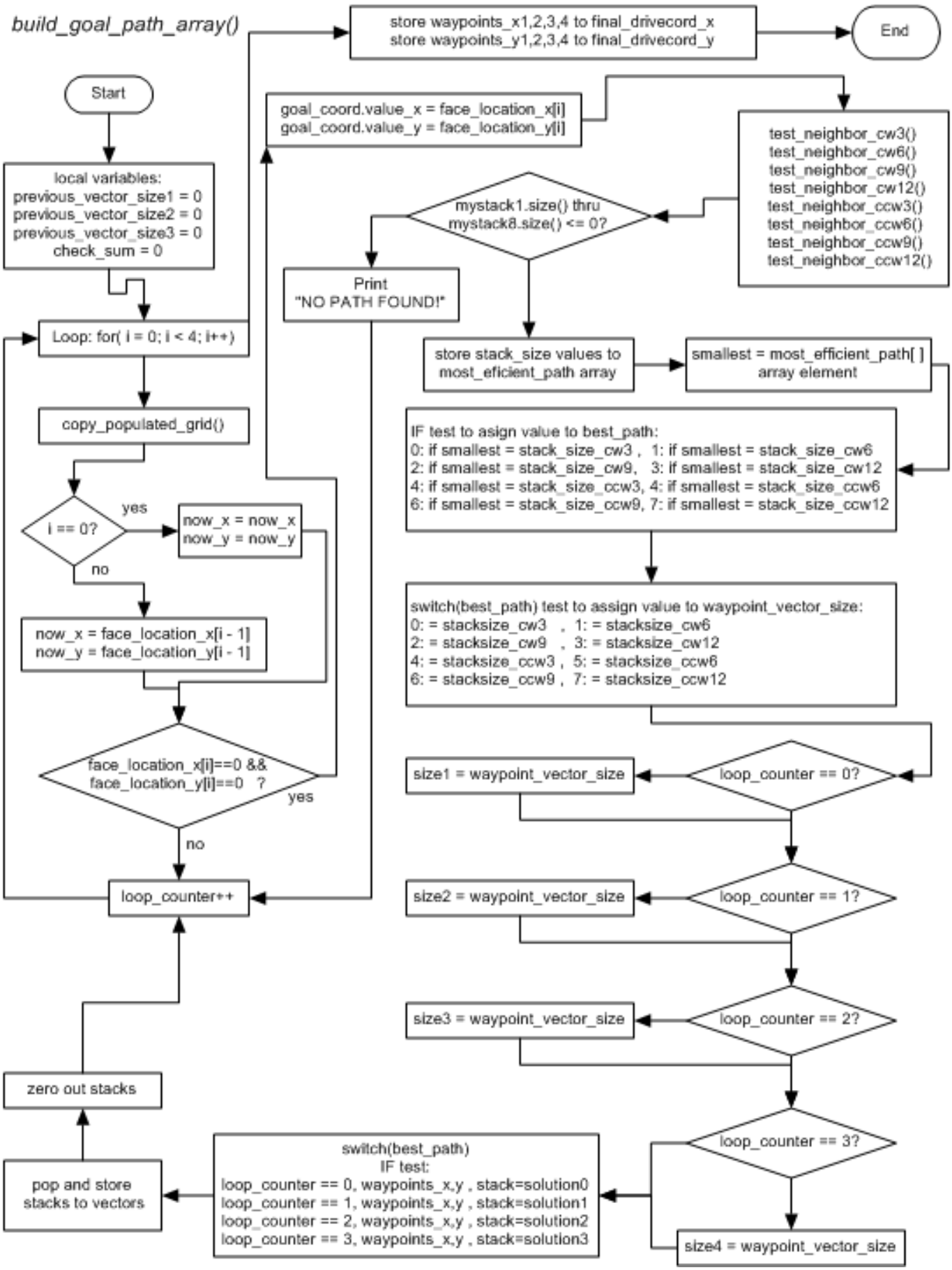


Fig. 19: Path Planning Algorithm Flowchart

APPENDIX

Appendix The remainder of pages in this document is supplied as supporting documentation for our project.

1) Vendor Contacts
   - The industry vendor contacts are listed.
2) Project Setup Guide
   - The base implementation of the system on top of a Linux environment is documented.
3) Parallax System Schematics
   - Schematics for the circuits on the Eddie control board.
4) Logitech C920 Data Sheet
   - Datasheet for the project's camera of choice.
5) HP Elitebook 840 Data Sheet
   - Data sheet for the project's laptop of choice.
6) Eddie Robot Platform Assembly Instructions
   - Assembly instructions for the Eddie Robot platform.
7) Project Member Resumes
   - The resumes of all project members are included.

VENDOR CONTACTS

Vendor Contacts Parallax Inc.
599 Menlo Drive, Suite 100
Rocklin, CA 95765 USA

Daniel Harris dharris@parallax.com

Andy Lindsay alindsay@parallax.com

FMC Technologies Schillings Robotics
Administrative Offices
260 Cousteau Place
Davis, California 95618, U.S.A.

Adwait Gandhe Adwait.Gandhe@fmcti.com

Project Setup Guide

This section should outline the layout and implementation of all software choices and configurations performed on our project laptop. If you follow this document and run all the specified commands on your own laptop, you will be able to run our project.

Operating System

Because our project will be running on top of a software development platform known as ROS, the operating system we selected was Ubuntu 14.04, and the following steps were taken to install Ubuntu:

1) Download Ubuntu 14.04 64-bit from https://www.ubuntu.com/download
2) Burn the .iso file onto a DVD or a bootable, LiveUSB
3) Restart computer, booting into the bootable DVD or the LiveUSB
4) Install Ubuntu on top of LVM

Basic configurations such as username and password are left to interpretation. The only major requirement is that the system is installed onto LVM software (to be discussed in a later section). Once the OS is installed, it may be wise to perform a bitwise backup of the system using cloning/imaging software such as Clonezilla, or for more experienced users, Unix's dd command.

For this project, a base image was produced using Clonezilla directly after the Ubuntu 14.04 installation procedure.

User Management

To simplify user management, it is proposed that we have one login to the laptop. This complicates some things, primarily for those who develop on the laptop itself. While running a git push, there will be no sufficient blame history if everyone's pushes come from "team 1". To assist with this massive design issue, I have set the git global config user.name to "team1". This at least gives a little better blame history.

File and Software Management Considerations

Considering there will be four developers maintaining code that will likely be ran from one laptop, there will often be need of a way to distribute the code. Github was chosen as our distribution platform, and git was chosen for our revision control system.

In terms of outside software (stuff that we did not develop ourselves), it is wise to keep this out of our github repository. Any software that you borrow should be added to the .gitignore file, and documented on the installation process, up to, but not necessarily including a script to perform the software install.

Detailed instructions on how to install the software required for this project are shown in the later sections of this paper.

Wireless and Proprietary Drivers

Internet access is required in order to complete the remainder of the installs. It was determined that since our laptop is not guaranteed to be plugged into wired etherent, that we should enable wireless. Because our HP laptop uses a Broadcom radio, wireless only works if we use a proprietary driver. We are using Broadcom 802.11 Linux STA wireless driver source from bcmwl-kernel-source (proprietary). This was enabled in the third party drivers utility on Ubuntu. To perform this install, the laptop must be plugged into Ethernet.

After the driver was enabled, a wireless network was joined. Specific network choice is left as an exercise to the reader.

Use of LVM

Because we will be developing software to run on a computer, we need to prepare the system for easy re-imaging in case of hard disk failure. As such, the OS was installed on top of a Logical Volume Manager (LVM). A graphic of the underlying LVM system is shown in Figure 22

The main benefit of LVM technology is the ability to snapshot the underlying operating system and

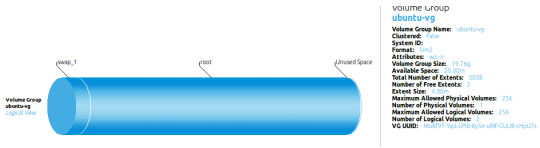


Fig. 22: *LVM Layout*

data drives for instantaneous backups, as well as the ability to be able to revert to these point-in-time snapshots. The scripts that we use for local backups and then scripts for offloading local backups using snapshot technologies can be seen in the later sections labeled “Backups” and “Restores” respectively.

To continue with the install, we have to customize our LVM install a little bit. We need to make space for a new logical volume where we can dump our backups onto for a primary backup solution. To accomplish these tasks, reboot into a LiveOS like Ubuntu, click “try” and open a terminal. Type the commands shown in Listing 1.

```
# note, it seems the best option for this
    port is the first USB port on the
    left hand side of the computer (
    closest to user on the left)
# resize default root partition, reduce
    by 50GB
sudo su
lvreduce --resizefs --size -50G /dev/
    ubuntu-vg/root
exit
```

Listing 1: LVM configurations

Next, reboot the computer and load into the native install of Ubuntu in order to finish the LVM configurations. Type in the commands shown in Listing 2. In this stage, we create logical volumes on top of our newly free space for a backup mount point, and add this to auto-mount during startup using fstab.

```
# restart computer, booting into the
    normal OS
# create a backup volume
sudo su
lvcreate --name backups --size 40G ubuntu
    -vg
mkfs.ext4 /dev/mapper/ubuntu--vg-backups/

# make backup directory, configure this
    for automount via fstab
mkdir /backups
mount /dev/mapper/ubuntu--vg-backups /
    backups
echo ``/dev/mapper/ubuntu--vg-backups /
    backups ext4 defaults 0 1'' >> /etc/
    fstab
exit
```

Listing 2: LVM Finalizations

Verify that the automount works by rebooting the OS and checking if /dev/ubuntu-vg/backups is mounted. In our project, after these configurations were made, the backup scripts shown in the “Backups” section of this paper were ran.

Software Installs

This section provides installation procedures for all software used by our project, whether by apt-get or by compiling software from scratch. The code listings in the following section list all software to be installed and a brief explanation of their purpose.

Update Existing Software and set up Appropriate PPAs: We begin with a basic software update, after adding a prerequisite PPA that we will use later in the installation process (OpenCV install requires ffmpeg). Type the commands from Listing 3

```
# Because ffmpeg is a requisite, we add
    the ppa before apt-get update
sudo add-apt-repository ppa:jon-
    severinsson/ffmpeg

# Update existing software
sudo apt-get update && apt-get upgrade
```

Listing 3: Update Existing Software and Set up Appropriate PPAs

Install Leveraged Software: Next, third party software is installed. Type the commands in Listing 4.

Git is required to interface with our github repository. Remote access is required, so a ssh server is installed. These will need further configurations that will be shown in the software configuration section.

Libarmadillo-dev is a linear algebra library for use within cpp programming. guvcview is a dynamic parameter adjuster for camera, to test various settings like autofoucs, etc. System-config-lvm is a GUI for LVM management, and while it is not strictly needed, it is helpful for visualization purposes.

```
# Install leveraged software
sudo apt-get install git system-config-lvm libarmadillo-dev guvcview openssh-server arduino
```

Listing 4: Install Leveraged Software

Install OpenCV: Since ROS Indigo, OpenCV is not released from ROS infrastructure. Its ROS-interface package vision_opencv depends on standalone libopencv* packages. The following script was modified from the code taken from Ubuntu's help center [13]. Type the commands in Listing 5

```
# Install OpenCV onto computer
cd ~/Desktop/
git clone https://github.com/jayrambhia/Install-OpenCV.git
cd Install-OpenCV/Ubuntu/
chmod +x opencv_latest.sh
./opencv_latest.sh

version="$(wget -q -O - http://sourceforge.net/projects/opencvlibrary/files/opencv-unix | egrep -m1 -o '\"[0-9](\.[0-9])+' | cut -c2-)"
#move opencv samples out of install folder and into the documents folder
cp OpenCV/$version/samples ~/Documents/OpenCV_samples
cd ~/Desktop
#rm -rf Install-OpenCV
```

Listing 5: OpenCV Installation Script

Install ROS Base Software: This project requires ROS Indigo (ros.org). Type the commands in Listing 6. Note that this script should be supplied credentials before it is ran. This can be done by running a "sudo-apt-get update" before running the script.

```
# Install ROS. NOTE YOU SHOULD SUPPLY CREDENTIALS TO BASH BEFORE RUNNING THIS
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.list'
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get install ros-indigo-desktop-full
sudo rosdep init
rosdep update
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Listing 6: ROS Installation Script

Finalize Software Installs and Other OS Configurations

SSH Configurations

First, we will configure our SSH server, as currently it is a security risk. We want to modify it so that only one user is allowed access to the server. To do this, we have to modify the server daemon file, and add the line "AllowUsers team1". This process can be done manually, or with the commands seen in Listing 7

```
# first, backup the old file
sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.factory-defaults
sudo chmod a-w /etc/ssh/sshd_config.factory-defaults

# Add AllowUsers team1 to the configuration
sudo su
 echo "AllowUsers team1" >> /etc/ssh/sshd_config
 restart ssh
exit
```

Listing 7: SSH Configuration Script

Pre-compiled ROS installations: There are two main nodes that need to be installed for our software to work correctly. The first is the camera driver "uvc_camera". Next, for serial communication, it is wise for us to use pre-existing drivers provided through "rosserial". These are installed using the commands in Listing 8

```
source ~/.bashrc
# Install uvc_camera driver
sudo apt-get install ros-indigo-uvc-camera


#install rosserial
sudo apt-get install ros-indigo-rosserial
roscd rosserial


# the following is required for serial comms to the arduino. this will require some setting up, which will be described in the next section
```

```
sudo apt-get install ros-indigo-rosserial
    -arduino

# Install ros-ar-track-alvar, required
    for alvar tagging
sudo apt-get install ros-indigo-ar-track-
    alvar
```

Listing 8: ROS Third Party Software Installation Script

Configuring Arduino IDE, Dailout, and rosserial-arduino: Because the arduino is used in some of our nodes, we have to configure some backend options. First, the Arduino IDE should have been installed by our very first script. Launch the IDE from Ubuntu's menu. The permission prompt shown in Figure 23 is how we enable dialout for our system. Dialout is required any time the Ubuntu system wants to talk over teletype (tty). Click the "add" button, and give the system the sudo password.

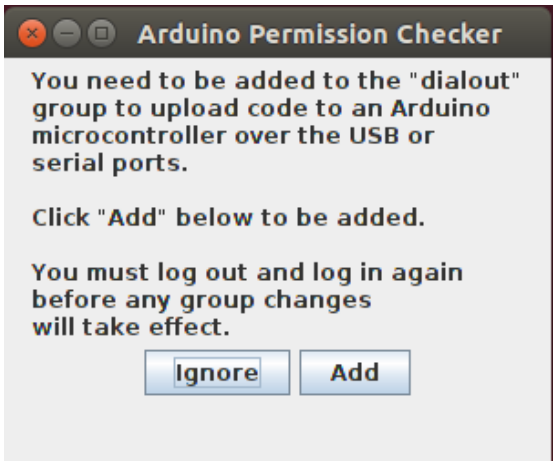


Fig. 23: *Arduino IDE Dialout Permission Request*

Now that we have dialout permissions, and Arduino IDE has been ran once, we can configure our rosserial settings to work with the Arduino IDE by following the documentation found on ROS.org [14]. In this, we rosrun a command to copy the libraries into the Arduino IDE's folder. Type the commands shown in Listing 9.

```
#Remove existing libraries and copy the
    necessary rosserial_arduino libraries
    into their appropriate folders.
cd ~/sketchbook/libraries
  rm -rf ros_lib
  rosrun rosserial_arduino make_libraries
    .py .
```

Listing 9: rosserial Configuration Script

Cloning Project Repository: Assuming you have been granted access to the github page, it is now time to clone the repository down for local use. Before we clone the system, it is important to give git a couple of settings. These settings can be seen in Listing 10. It is very important to set up these fields with correct names and email addresses. Failing to do so will result in improper blame histories, and can make debugging a nightmare.

```
# Tell git who you are
git config --global user.name "John Doe"
git config --global user.email
    johndoe@example.com
```

Listing 10: Configure git

Type the commands in Listing 11 to clone the master branch onto the local system.

```
# create a git repository
mkdir ~/curtkin && cd ~/curtkin

# initialize empty repo
git init

# add remote
git remote add origin https://github.com/
    haywardt916/Senior-Design.git

# update
git fetch --all

# pull down master branch
git pull origin master
```

Listing 11: github Clone Script

If typing in your credentials to github is too wearing on your fingers, you can set up git to cache credentials by typing in the command in Listing 12. Verify that the system compiles all required software

```
# configure credential caching
git config --global credential.helper
    cache
```

Listing 12: Cache Credentials

Verify that your project builds by running a "catkin_make" now. There should be no compiling or linking errors.

Modify bashrc

Now we can modify our bash configuration file to include a couple of important configuration changes. Run the commands in Listing 13 to finish up configuring the bashrc file.

```
# add a new source:
USER=$(whoami)
echo "source /home/$USER/curtkin/devel/
    setup.bash" >> /home/$USER/.bashrc

# opencv flags
# THESE NEED TO BE ADDED TO THE BASHRC
    MANUALLY. add them to the end of your
    bashrc file.
# OpenCV_INCLUDE_DIRS=$(pkg-config --
    cflags opencv)
# OpenCV_LIBRARIES=$(pkg-config --libs
    opencv)

# re-source
source /home/$USER/.bashrc
```

Listing 13: bashrc configurations

Special (compiled) ROS Software Installations: Interfaced libraries:

- viso2_ros mono_odometer
  - borrowed from Andreas Geiger [3].
- robot_pose_ekf (part of the navigation stack)
  - borrowed from willow garage's Navigation stack

To install viso2_ros mono_odometer, go to the project's github page https://github.com/srv/viso2 and download the repository, extracting into your catkin_ws/src directory. After this is extracted, a simple catkin_make should build the appropriate nodes.

To install robot_pose_ekf, we need to install the BFL library (install via sudo-apt-get ros-indigo-bfl). Once this is installed, we download the project from https://github.com/ros-planning/navigation, and extract the robot_pose_ekf to your catkin_ws/src directory. After this is extracted, a simple catkin_make should build the appropriate nodes.

Rapid Prototyping Software: While not strictly needed to run any of the project's production features, the Octave environment was used to rapid prototype many nodes, ranging from object isolation, and motor PID control analysis.

It can be installed on Ubuntu by typing "sudo apt-get install octave" into the terminal.

Running all the ROS nodes

Because we have set the ROS environment up to optimize for processing power on the laptop, we have shifted some of the processing to other computers.

For simplicity, consider the two computers "Master" and "Remote". "Master" will be the main control guy, performing all mapping and data aggregation, as well as some remote controls. "Remote" will be the remote computer that will be gathering the data.

To launch the nodes required by these two computers, first we need to set up the environments. If these computers are configured with static IP addresses on a c

Backups

The following scripts will perform primary backups (via snapshots). Post processing copies them to the /backups directory. Tertiary backups are also implemented.

Primary Backups

```
# NOTE THIS MUST BE RUN AS ROOT
# NOTE THIS MUST BE RUN AS ROOT
# NOTE THIS MUST BE RUN AS ROOT
lvcreate -L10G -s -n rootsnapshot /dev/
    ubuntu-vg/root
mount /dev/ubuntu-vg/rootsnapshot /mnt/
    snapshots

#back up the snapshot
$TODAY = `date +%F'
mkdir /backups/$TODAY
tar -pczf /backups/$TODAY/root.tar.gz /
    mnt/ubuntu-vg/rootsnapshot
dd if=/mnt/ubuntu-vg/rootsnapshot conv=
    sync,noerror bs=64K | gzip -c  > /
    backups/$TODAY/root.dd.gz

#cleanup the snapshot (remove)
umount /mnt/snapshots
lvremove /dev/ubuntu-vg/rootsnapshot
```

Listing 14: Primary Backup Script

Secondary Backups

The file system is backed up to the "/backups" mountpoint, so that a backup is freely available at any point in time on the actual system should they be needed. There is still a single point of failure here: if the hard drive crashes.

Secondary backups should be performed by copying the "/backups" directory off onto an external HDD.

Restores

If, in the case we lose our root partition, we can restore simply by booting into a Linux LiveUSB, opening a terminal and running the following set of commands:

```
# NOTE THIS MUST BE RUN AS ROOT
# NOTE THIS MUST BE RUN AS ROOT
# NOTE THIS MUST BE RUN AS ROOT
$TODAY = `date +%F`
gunzip -c /backups/$TODAY/root.dd.gz | dd
    of=/dev/ubuntu-vg/root
```

Listing 15: Restore Commands

Microsoft Lifecam Fixes

Due to the fact that the Lifecam has serious issues, we have to implement the fixes seen in 16

```
echo autospawn=no > ~/.config/pulse/
   client.conf
pulseaudio --kill
sudo modprobe uvcvideo quirks=0x80
```

Listing 16: Lifecam Fixes