April 2008

Autonomous Vehicle Control

Alexander Glencairn Cunningham
Worcester Polytechnic Institute

Robert Salvator Breznak
Worcester Polytechnic Institute

Zachary Andrew Kleinfeld
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.
Autonomous Vehicle Control

A Major Qualifying Project Report
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

By

__________________________________________________________________________

Robert S. Breznak

__________________________________________________________________________

Alexander G. Cunningham

__________________________________________________________________________

Zachary A. Kleinfeld

Date: April 18, 2008

Approved:

__________________________________________________________________________

Professor William Michalson

1. Robotics
2. Machine Vision
3. Autonomous Systems
1 Abstract
A practical knowledge base in the emerging field of Robotics was developed and used to create a framework for further experiments. The framework was designed such that modular parts could be replaced, allowing for future development without “reinventing the wheel”. To prove the framework, a semi-autonomous robot was implemented, including stereo vision sensors, an inertial navigation system, and a simultaneous localization and mapping algorithm.
# Table of Contents

1. Abstract .................................................................................................................. 2
2. Table of Contents .................................................................................................... 3
3. Table of Figures ........................................................................................................ 4
4. Table of Tables ......................................................................................................... 5
5. Executive Summary ................................................................................................ 6
6. Introduction .............................................................................................................. 1
7. Background .............................................................................................................. 5
   7.1 Robotic Applications .......................................................................................... 5
7.2 Vision ..................................................................................................................... 6
   7.2.1 Stereo Vision Tradeoffs ............................................................................... 6
   7.2.2 Vision Techniques ....................................................................................... 7
7.3 Positioning and Telemetry .................................................................................... 14
   7.3.1 Global Sensors ............................................................................................. 14
   7.3.2 Localized Sensors ....................................................................................... 15
   7.3.3 INS ................................................................................................................ 16
7.4 Simultaneous Localization and Mapping .............................................................. 16
   7.4.1 World Modeling ............................................................................................. 20
   7.4.2 Simple SLAM Implementations .................................................................. 25
   7.4.3 Probabilistic Implementations ..................................................................... 27
   7.4.4 Particle Filters .............................................................................................. 31
7.5 User Assistance ...................................................................................................... 33
8. Requirements ........................................................................................................... 34
8.1 Robotic Framework ............................................................................................... 34
8.2 Sample Application ............................................................................................... 35
9. System Implementation ............................................................................................ 37
9.1 System Design ....................................................................................................... 37
   9.1.1 Vision ............................................................................................................ 38
   9.1.2 Positioning and Telemetry .......................................................................... 51
   9.1.3 Simultaneous Localization and Mapping ...................................................... 57
   9.1.4 Platform ......................................................................................................... 67
9.2 System Integration ................................................................................................ 67
   9.2.1 Architecture .................................................................................................. 68
   9.2.2 Hardware ....................................................................................................... 72
10. Results ..................................................................................................................... 75
10.1 Unit Testing .......................................................................................................... 75
   10.1.1 Vision .......................................................................................................... 75
   10.1.2 Positioning and Telemetry .......................................................................... 77
   10.1.3 SLAM ............................................................................................................ 80
10.2 System Testing ..................................................................................................... 80
11. Conclusion ............................................................................................................... 81
12. References .............................................................................................................. 85
13. Attached Appendecies: ......................................................................................... 87
3 Table of Figures
Figure 7-1: Stereo by Camera Shift ................................................................. 8
Figure 7-2: A Disparity Map ........................................................................... 9
Figure 7-3: A Set of features is simply a table ................................................. 11
Figure 7-4: Stereo Triangulation .................................................................. 13
Figure 7-5: The world of a vacuum cleaning robot with limited sensing .......... 17
Figure 7-6: Basic SLAM timestep including both a change in robot pose and landmarks .......................................................... 18
Figure 7-7: Pose definition in 2D space ........................................................... 21
Figure 7-8: Choosing Landmarks to be representative of the geometry of the world ...... 22
Figure 7-9: Ambiguous data association with two known landmarks and a ghost landmark ........................................................................ 25
Figure 7-10: Example Gaussian Curve ............................................................. 28
Figure 7-11: .................................................................................................. 31
Figure 9-1: Overall System Structure .............................................................. 37
Figure 9-2: Vision System Hardware ............................................................... 39
Figure 9-3: The Camera Head ....................................................................... 40
Figure 9-4: Block Matching .......................................................................... 40
Figure 9-5: The Main Loop ........................................................................... 42
Figure 9-6: The Detector ............................................................................. 44
Figure 9-7: Vertical Line Detection, Annotated and color inverted. (Line of Most Interest circled) ............................................................................. 45
Figure 9-8: Remaining Features, Annotated and color inverted ....................... 46
Figure 9-9: Test Setup .................................................................................. 49
Figure 9-10: SunSPOT ............................................................................... 52
Figure 9-11: Accelerometer Axes on the SPOT .............................................. 53
Figure 9-12: SLAM System Block Diagram .................................................... 58
Figure 9-13: Vector components of transforming landmarks ........................... 63
Figure 9-14: GLADOS SLAM IDE User Interface ........................................... 65
Figure 9-15: System Architecture ................................................................. 68
Figure 9-16: INS Server Controls ................................................................. 68
Figure 9-17: GLADOS Host Controller Window .......................................... 70
Figure 9-18: GLADOS Host SLAM Visualization Window ............................. 70
Figure 9-19: Robot with Mobile System onboard ............................................ 72
Figure 9-20: Computer Systems Used ............................................................ 73
Figure 10-1: Accuracy has no correlation to distance ..................................... 77
Figure 10-2: A vision and INS Test, in progress ........................................... 78
Figure 10-3: A two meter displacement, with noise ....................................... 80
4 Table of Tables
Table 10-1: Vision Data Results .......................................................... 76
Table 10-2: Displacement Error Rates ..................................................... 79
5 Executive Summary
The problems with autonomous systems in the real-world are numerous. A robot must have some way to perceive its environment, but it becomes difficult to logically process only the data that is relevant, while ignoring the mountains of data that is not. A robot must have some way to localize its own position, but must be prepared to maintain this state even if any given sensor may not work or may be lying. A robot must have some way of changing its position, but must be able to perceive exactly how its position is actually shifting. Answers to these problems are sought in much of the research into robotics. It is the primary intent of this project to support this research. We provide a modular framework that will allow for future work to address these basic problems without reinventing the wheel. We also provide an example implementation of a semi-autonomous robot, using this framework, as a proof of concept.

The framework consists of three computer systems, a mobile system, a control system, and an assistant system. Each of these three computers run several different processes, and divide up responsibilities between themselves. Breaking primary tasks into multiple systems allowed the robot platform to be freed from on-board processing. Each application communicates over a high-speed network, thus allowing data to be collected at one location (i.e. on the vehicle or from a human interface device), processed at another location (i.e. a command center) and then have the results sent to yet another location (i.e. a data logger or web tracking system).
Using this framework, we implemented a semi-autonomous robot. We used stereo vision to determine the robot’s surroundings, an inertial navigation system to determine the robot’s motion, and a simultaneous localization and mapping algorithm to tie this data together. In testing, we showed that the system is between 75% and 80% accurate in determining its real position after a movement in controlled conditions. While not sufficient for many robotic applications, it proves the usefulness of the framework, and gives a clear path forwards.
6 Introduction

While the field of robotics has been developing significantly in recent years, robots still have a very long way to go before autonomous systems are viable in complex real-world situations. In controlled conditions, autonomous robots have proved to be extremely successful. Robots build cars, process mail, check parts, and chop chicken in factories. Robots chase balls, navigate rooms, and juggle balls in laboratories. But outside these controlled conditions, very few serious robots are available to complete even in what might seem the simplest of tasks. Robotic vacuums get stuck in corners and tangled in electrical cords. Robotic lawnmowers cut ridiculous random lines in the most uniform lawns. Autonomous cars can only be constructed with millions of dollars and specialized military-grade sensors. Effective robotic systems in uncontrolled environments tend to be either very limited or very expensive.

The problems with autonomous systems in the real-world are numerous. A robot must have some way to perceive its environment, but it becomes difficult to logically process only the data that is relevant, while ignoring the mountains of data that is not. A robot must have some way to localize its own position, but must be prepared to maintain this state even if any given sensor may not work. A robot must have some way of changing its position, but must be able to perceive exactly how its position is actually shifting. A fully autonomous robot must be able to make decisions to change its position based on the world it perceives; it should be noted, however, that this project does not include artificial intelligence within its scope. Answers to these problems are sought in much of the
research into robotics. For example, the Defense Advanced Research and Projects Agency’s (DARPA) Grand and Urban Challenges are largely characterized by these issues.

Because the issue of building sophisticated, robust robotic systems is well beyond the scope of a single MQP, our team decided to limit the scope to a more realistic sub-problem. The sub-problem chosen was the issue of world-building, based on real-time interpretation of sensor data. In the examination of such applications as the DARPA Urban Challenge, it was found that a major part of having a robot operate autonomously in uncertain environments is the ability to create and maintain a cohesive and persistent model of the world around the robot. This model should be able to place the position of the robot within the environment, which is an important step in creating a full system control loop.

With this base problem to investigate, our team decided on three primary goals for the project:

1. Develop a practical knowledge base for the implementation and use of world modeling systems in robotics
2. Create a general-purpose framework for development of systems with world-modeling applications.
3. Build a working semi-autonomous implementation of this framework.
This framework provides implementations for some basic sensor systems, in this case, stereo vision and inertial measurement, as well as the necessary algorithms and visualization mechanisms to develop the real-time world modeling system. Because its primary focus is to enable future expansion, best software engineering practices dictate a system with low coupling and high cohesion. This is achieved via modules that are designed with the open-closed paradigm in mind. Full use of design patterns, along with good object-oriented design, enables the system to be rapidly extensible.

The implementation of the framework is actually a semi-autonomous system designed for remote navigation via existing network technologies. The user has direct real-time control of the robot through network connections, and receives feedback as to the state of the robot through both a vision system and an inertial navigation system (INS). However, the vision and INS systems are more than just indicators; the information is processed on the control computer in order to create a map of the environment and display the information to the user in a meaningful way. This application simulates any of the remote robotic applications where a robot must perform in a possibly dangerous area, and the additional aids help to overcome some of the usual difficulties with these remote systems.

We do not address autonomous decision making. We address environmental perception with a simple stereo vision system. We address vehicle movement via a remote control interface and a simple five-degree-of-freedom inertial navigation system. We address world-building via a simple Simultaneous Localization and Mapping algorithm. We
primarily provide a framework; a modular hardware setup and software application suitable to have its pieces removed, replaced, or ignored. It is hoped that, using our framework, research can be done on one single robotics problem independently of the others.
7 Background

Because a major goal of this project is to build a practical knowledge base for robotics, the first phase of the project must be research into existing concepts in the field. This background research is necessary for both the creation of this knowledge base as well as to focus on certain topics and techniques.

7.1 Robotic Applications

As the field of robotics advances and the foundation of knowledge builds, projects that are on the forefront of the field will become increasingly more advanced. This can be directly seen in the complexity and scale that was seen in the DARPA Urban Challenge when compared to the initial DARPA Grand Challenge. The Challenge started with a goal of building a system that could autonomously navigate on open terrain, to expecting a high success rate in the construction of an autonomous navigation system that could drive though an urban setting while obeying traffic laws, avoiding pedestrians and other real-world obstacles. This transition occurred swiftly once a solid body of knowledge had been generated.

As other robotics-based projects come to fruition, the body of knowledge will increase and the progress of robotics will follow the same trends in progression that computer science and electrical engineering have enjoyed for the past century.
7.2 Vision

When evaluating sensor systems, one of the most obvious options mimics humans: stereo vision. In theory, the sensor is very simple. Two cameras are positioned some distance apart in a near-parallel configuration. Images are then captured from them, simultaneously. By comparing the image shift (the disparity) between the two images, the depth of objects appearing in both images can be calculated.

The applications relevant to vehicle control are numerous. Depth data enables object detection and identification. A disparity map allows for easy feature detection, which in turn allows the implementation of simultaneous location and mapping (SLAM) system.

7.2.1 Stereo Vision Tradeoffs

By selecting stereo vision as a sensor, we gain great capability, but at a cost. By using cameras, we can gain a speed advantage over scanning sensors; for example, a laser rangefinder. Rather than be forced to mechanically pan over our field of view, most cameras have a 30-50 degree field of view. With creative lenses and multiple cameras, it is possible to gain 180 degree views without any moving parts. As the sensor is solid-state, this also increases the reliability of the sensor itself. We also can create a sensor for minimal cost; low end LIDAR (Light Detection and Ranging) typically run over $2000, while low end cameras sufficient for basic vision can be obtained for less than $50. We can create a sensor with very few precision-engineered parts; no sophisticated mirrors and mounts are required (although they still certainly help).

On the other hand, stereo vision does present several limitations. The quality of a depth result is subject to a significant error due to a great many of factors, including lighting,
camera quality, mounting quality and algorithm correctness. Sensor readings are therefore neither as accurate nor precise as those that would be obtained by laser scanner [1]. Lastly, with several notable exceptions, the image matching algorithms remain proprietary, very complex, or insufficient for real-time applications such as robot navigation.

7.2.2 Vision Techniques
However, selecting stereo vision is not specific enough; there remain several methods of actually extracting data from a pair of cameras.

The first technique we evaluated did not, in fact, actually use two real cameras. By capturing an image, then moving the camera (in our case, moving the robot base), a single capture device can simulate a second camera. This technique was dismissed for two reasons. First, there are twice as many time-steps involved in matching the images, as one must always wait for the next image before discarding the previous one. Second, the transform needed to use images which may vary in six degrees of freedom are quite complicated, and are easily avoided by selected a different method.
The second technique involved use of “structured light”; lasers could be used to generate a framework of features a camera system could then use as a reference when building a disparity map and detecting objects. While the technique seems to have great promise, it was decided that such a sophisticated and cutting-edge method was beyond the scope of this project.

The final two techniques were attempted in this project, and are detailed below.

**Dense Matching**

One technique involving two cameras is to attempt to match every pixel in the first image to every pixel in the second. In an ideal universe, with perfect cameras and infinite
resources, this technique essentially gives a depth reading at every single pixel present in the images’ overlap. This makes the method suitable to making sophisticated models of objects in an environment. This method is by far the most common, and produces the human-understandable disparity maps many are familiar with.

![Figure 7-2: A Disparity Map](image)

In our experience, however, this approach is time, memory, and error intensive. Determining what set of pixels matches which target set is generally very difficult and requires a great number of operations. Accurate algorithms are generally extremely complex, while simpler algorithms are generally ridiculously inefficient.

There’s little way to guarantee an algorithm here won’t run in \( \Theta(n^2) \) time. Even in the best case (a very inaccurate but efficient algorithm), every single pixel in an image must be compared to at least one pixel in a second image. For a 640x480 image that’s 307,200 comparison operations (which themselves may be complex). In the worst case (an accurate but inefficient algorithm), every pixel must be compared to every pixel in the second image, using 94,371,840,000 (that’s 94 \text{ billion} \) operations) comparison operations.
For reference, a modern AMD Athlon 64 is capable of 12 billion instructions per second [2].

Assuming each comparison operation is ten instructions, that gives us an astronomical (and unrealistic) 3906 frames per second in the best case (suggesting each comparison is most likely well more than 10 instructions). However, this also reveals an abysmal (and completely unusable for real-applications) 0.0127 frames per second in the worse case.¹

Because of this, when speed and accuracy are important, dedicated hardware is often used to implement this method, including pre-aligned cameras and dedicated processors.

**Sparse Matching**

To attempt to ameliorate the inherent difficulties of dense matching, we can only calculate depth at certain distinct features in an image. This method is generally thought to be more similar to how biological systems work, and is suitable to operate robot navigation and other applications where accuracy is less important than efficiency [3].

First, both captured images are processed (either with a sophisticated algorithm or a set of filters) to separate interesting features from the image at large. Depending on the method used, a feature might be a corner, a vertical edge, a bright segment of an edge, or any number of more complicated mathematical features. Feature detection algorithms suitable

---

¹ It should be noted this isn’t merely a worse-case scenario that never occurs; an early algorithm on our test computer (running an AMD Athlon 64) behaved at approximately this level of efficiency, taking, on average, nine to ten seconds per frame.
for stereo matching are numerous, relatively straightforward, and generally have open-source or reference implementation available on the internet [4].

Second, the identified features in one image are matched to the features in the other. If a feature with similar properties (size, position, strength of the feature) is found in both images, a match is made, and the position difference is saved. Otherwise, the feature is regarded as an anomaly, and discarded. Thus, instead of disparity being saved at each pixel, only a table of disparity is saved, with values at a smaller set of specific features.

![Table](image)

**Figure 7-3: A Set of features is simply a table.**

As a consequence of this technique, we make great gains in speed and memory usage. First, we must separate features at every point in both images; in the example of a 640x480 image, we must do at least 307,200 * 2 = 614,400 comparisons for a simple intensity filter. Realistically, we’re going to have to do multiple matrix convolutions, which we can estimate at 5 times that many operations, say, 3,072,000 operations. Unlike dense matching, this is going to be more or less a constant, however. In the worst case, we end up with no features to match, but, even with sophisticated feature detection algorithms such as FAST Corners or SIFT Features, this shouldn’t cause runaway complexity.
Once that’s done, we only have to search through the number of matches, not the number of pixels. Estimating the number of matches is difficult. Under almost every feature detection scheme, more than a single pixel is required to construct a feature; ten is a more realistic number. Thus, in the best case, we spend only 30,720 comparison operations matching, while in the worst case, 943,718,400 (that’s 943 million) operations are used.

In total, this gives us a range of 3,102,720 to 946,790,400 operations per frame. While in best case, dense matching takes fewer operations, in the worst case, sparse matching takes almost one hundred times less. For comparison, using the same assumptions as above, that’s an unrealistically high 386 frames per second to a reasonable (if slow) 1.2 frames per second.

However, the tradeoff is two-fold. First, we only get depth data at specific places in the image, rather than at every pixel. Second, we have an extra feature detection phase that may introduce errors by detecting false positives and by missing features.

**Depth Calculation**

In any of the two-camera methods above, we still have to calculate the real-world depth of a pixel or feature. To do so, we require knowledge of the camera’s position, relative position, pitch, yaw, band, focal length, and attitude of image planes. These values can either be obtained through specification or simply measured, and then can be used as in Figure 8-4.
In the figure above, the base distance, \( B \), is the measured distance between the centers of the cameras. The original data is characterized by the values \( x \) and \( y \), the position of a feature in the image plane. Lastly, the original data includes the focal length \( f \) (which should be the same for both cameras.) Unknown quantities in the figure are \( x' \), \( y' \), and \( z' \), the three dimensional position in the real plane.

Ultimately, using basic geometric rules, we can reduce this relation down to Equation 8-1:

\[
D = \frac{\beta f}{\Delta}
\]

Where \( D \) is the real-world depth, \( \beta \) is the base distance between camera centers, \( f \) is the focal length of both cameras, and \( \Delta \) is the disparity.
7.3 Positioning and Telemetry

A number of systems and techniques exist to determine the position of a vehicle. The primary division among position tracking systems is whether the system takes measurements based on an external (global) scope or internal (local) scope. Systems that utilize an external scope (i.e. GPS) will determine the position as an absolute value where the vehicle exists on an existing map. Localized system (i.e. wheel encoders) will measure different aspects that are associated with the movement of the vehicle across the driving a plane and determine the position of the vehicle based on previous positions of the vehicle.

7.3.1 Global Sensors
Global seniors report the value of their location as an absolute location on a preexisting map. For example GPS uses a series of satellites in orbit around earth to enable a receiver to triangulate and determine its exact position on earth [5], [6]. When the position is polled at different points in time the path of the vehicle can be determined and the velocity and accelerations can be calculated. In order for a global positing system to function correctly, it is required that the environment that the vehicle is in to have already been mapped and beacons or markers to have been already placed.

Accuracy is the primary comparison points for a global sensor. The accuracy describes how close the system can accurately determine the position of the receiver. Accuracy within 1 meter describes that the device will be within a meter of the reported values. Typical accuracy values are within 15 meters [5].
7.3.2 Localized Sensors
Localized sensors measure aspects that are local to the vehicle itself. Wheel encoders, accelerometers and gyroscopes are examples of sensors that rely only on the vehicle to collect data. Accelerometers will measure the inertial forces acted upon the vehicle during travel and will respond back with a signal that is proportionate to the amount of force that was measured. By using the direct accelerations experienced by the vehicle one can, over time, determine the path traversed by the vehicle by calculating position from the acceleration data. Combining multiple types internal sensors together will produce an inertial navigation system (INS).

In localized sensors the main comparison points are resolution that describes what range and magnitude of can be reasonably accurately measured, polling rate is the frequency at which the device measures the forces acting upon it and drift over time, which is the amount of inaccuracy that will build over time.

The effective range and magnitude of the sensor gives an idea of what applications the sensors has been designed for and what types of accelerations the device can reasonable handle. An accelerometer with a 2g range may be useful in determining the orientation of a hand-held device, but would be a poor choice for use in the black box of an aircraft.

Drift is caused by small inaccuracies in the measurement building up over time. As the inaccuracy builds, any reported values will become skewed and unreliable [5]. The issue of drift is a nuisance for top-notch components but can be flat-out crippling for typical
commercial sensors as the drift in these cheaper components can quickly build in just a matter of seconds.

### 7.3.3 INS

Typically found in planes and ships more so than in land vehicles, requires only the initial position and velocity to be given and then the system will continue from that point by using measurements of accelerations and impulses experienced by the vehicle as it progresses to update its position, bearing and velocity statuses.

Being comprised only of localized sensors, the system is able to function within unknown and unmapped environments and is able to be shielded from interferences [7].

### 7.4 Simultaneous Localization and Mapping

Even after basic sensor data has been processed, it is important for there to exist a means to integrate the data into a cohesive model that can be used to make decisions. To examine the effect of an internal world model in a simple robotic application, it is instructive to look at a robot that lacks a persistent world model, such as a simple robot vacuum cleaner.
In this example of a robotic vacuum cleaner, as shown in Figure 7-5, the robot, represented by the circle, has maneuvered through the room which contains obstacles. Note that in this case, the sensor range of the robot is limited to only a small area that can only pick up a small segment of the obstacles in the room.

The general strategy used in a robot vacuum cleaner is to move in relatively random directions, while backing away and avoiding obstacles. While this will eventually cover an entire floor, it usually takes about four times as long to vacuum as it would for a human. If this robot had the ability to create a map of the area, and track its position in the room, it could remember where it had previously vacuumed, as well as get a rough position of where obstacles are.

Whenever the robot runs into a piece of furniture, it could store that point on its map, and rather than backing away randomly, could drive towards an area that had not been
covered and had few obstacles. The addition of this mapping mechanism with only simple sensors systems could significantly improve the effectiveness of a robot’s ability to complete tasks and deal with obstacles. Furthermore, if the robot were able to remember rooms over time, it would not have to relearn where the majority of obstacles are each vacuuming run, it would simply need to recognize the room.

The important characteristic of this problem is the inclusion of a persistent world model which includes both elements of the environment observed previously as well as a model of the robot’s current position in relation to the rest of the world. Figure 8-6 shows a simple 2D Simultaneous Localization and Mapping (SLAM) process, with a robot that has moved, and two landmarks. Note that only one of the landmarks is in the sensor range of the robot (denoted by the arc), but the known landmark outside of current sensor range is still left on the same map [8], [9], [10].

Figure 7-6: Basic SLAM timestep including both a change in robot pose and landmarks
The important distinction between this model and the original vacuum cleaner robot is in the persistence of the world. For a robot with only an immediate understanding of the world around it, the environment only comes into play when some external sensor, like a bumper, is touched. Once an avoidance behavior, such as turning around and driving away, is executed the robot will have essentially no memory of that obstacle. The persistence of a world model is important to ensure that the robot makes the best of the information it is able to gather [8], [9].

The formal name for this world-building problem is Simultaneous Localization and Mapping (SLAM), where the problem of building a persistent model is split into performing localization of the robot and mapping the environment. These tasks can be approached separately, given certain known values.

Localization [9] uses a known map of an environment and requires the robot to determine its position in the environment. Solving this problem relies on finding areas that are unique in the known environment to map to real environment. A robot performing only localization will need to rely on the quality of its pre-generated map in order to find its place in the environment.

Mapping [9], done in isolation, is the dual of localization: the robot is attempting to create an internal representation of an environment when the location of the robot is known. Mapping done without dynamically determining the location of the robot is primarily useful in the case of a robot that is fixed in place, or even a system such as a
fixed camera. Because the position of a fixed viewing platform can be assumed to have a very small amount of error relative to any moving robot, the integration algorithm can focus only on generating the map.

While these may seem like separate tasks, they are intricately connected, as measurements taken to determine both the position of the robot and objects in the environment are interdependent. If the previously mentioned robotic vacuum cleaner ran into an obstacle after an hour of vacuuming, it would need to know where it was in relation to the room (localization) in order to fix the obstacle in the world model (mapping). Essentially any mobile robot that can sense the environment will need to handle these problems simultaneously.

7.4.1 World Modeling
There are a very large number of variables that can go into a working model of the world, and the complexity of the model depends on a number of constraints. At a given moment, there are a large number of parameters that can describe the world’s state. For an arbitrary robot, the state will include parameters to describe the size and position of the robot, as well as describing the dynamic characteristics of the system. As for the world description itself, every object would need to be modeled as a state variable, with various characteristics of the objects stored as well. The state can be broken down into two major classes, however: the dynamic state and the static state. The dynamic state tracks the motion of parts of the environment and robot that are changing, while the static state should be consistent during the robot’s operation [9]. For the purposes of this paper, the
only dynamic state variable being tracked in this discussion of SLAM is the state of the
robot and the rest of the environment will be assumed to be static [9].

The robot *pose* will need to model the state of the robot, and be continuously updated as
the robot moves. This pose must, at a minimum, contain information about the position
of the robot in 3D space, but also important is its orientation on rotational axis. In Figure
8-7, a 2D definition of pose is shown, which is defined by three terms, the *x* and *y*
coordinates and rotation Θ around the axis normal to the *x/y* plane. For the purposes of
simplicity, graphics used in this section will use the 2D definitions of data, but the overall
algorithms work for full 6 degree of freedom coordinate systems [9].

![Pose definition in 2D space](image)

The robot can save a series of poses over time to generate path, from which additional
information, such as velocity and acceleration can be determined.
The environmental sensors return information about the external world in the form of *landmarks*, which are simply points that the sensor system has identified as significant. In the internal world model, the landmarks are simply 3D points stored on a global coordinate system along with the robot pose. These landmarks should not simply be random points, but rather points that will provide the most descriptive and consistent model of the world [10].

![Choosing Landmarks to be representative of the geometry of the world](image)

**Figure 7-8: Choosing Landmarks to be representative of the geometry of the world**

Good descriptive landmarks would be corners and edges of obstacles, as these can be used to infer the structure of objects. The landmarks need to be persistent between samples, so that the exact position of the landmark can be approximated more thoroughly. While all that is necessary is a location, landmarks should contain whatever extra information that can be extracted from the sensor system, such as color or texture information. This additional information can be helpful in associating data from a given sensor sample with specific landmarks in the world model.
From the set of landmarks, a map can be generated, which will consist of all the landmarks on the same global coordinate plane. The persistence of the landmarks is important in creating a map that can be used in the future by the robot or by a human observer. One test for showing determining the effectiveness of a slam algorithm is to record data from a fixed path in a loop. As the robot closes the loop, an effective SLAM algorithm should recognize the area over which it has already traveled. One difficult point in the creation of these maps is the handling of moving objects. In the simplest sense, a robot could assume itself to be the only moving entity in the immediate world, but this is frequently not true. The effect of this is erroneous landmarks that may be placed into the map. There are several means of handling landmarks that are either moving or erroneous, which will be described in more detail later. The focus of this project, however, is on situations where the world can be assumed to be essentially static and non-persistent landmarks can be handled as noise [9].

The inputs to a general SLAM algorithm consist of odometry information about the movement of the robot and environment information that describes the position of landmarks. For the purposes of this project, SLAM only functions as state estimation, rather than a full control system, so the “odometry” and “control” will be used interchangeably. A sample from the odometry sensors only needs to show the difference in position between the previous state at which a sample was taken and the current one. This is essentially a difference between two robot poses, and as such, is a 6 degree of freedom vector containing both a linear translation in three dimensions and rotational angles in three dimensions. The environment sensor inputs can come from a variety of
inputs, including laser scanning systems, sonar rangefinders, or vision systems. In each case, the environment sensor must provide a set of landmarks with a coordinate system local to the robot itself [9].

The largest issue with this modeling scheme is the nature of maps that consist primarily of points, which can be hard to extract human-readable information from. A human can look at a 3D or 2D rendering of the map and extract some information, such as inferring that a cloud of landmarks represents an obstacle of some kind. However, if the map is primarily used for internal robot navigation, then this sort of point map can still be quite effective at creating a persistent world model. The aspect of the mapping that can be understood by the robot is the path of the robot, which should be readily apparent from the output of a SLAM algorithm.

Another difficulty in implementing a SLAM algorithm is in associating a given new environment measurement with known landmarks. This problem of data association can be seen in the clearest form in Figure 8-9, below [10].
One aspect of SLAM that should be noted is the relationship to the rest of a robotic system: SLAM, as a whole, is not a control system that provides navigation data, but rather just an algorithm that determines the current state of the robot and the world. While SLAM could be a part of a closed loop control system, it does not necessarily imply any control that will be used. All uses of SLAM described and used in this project can easily be implemented as an offline algorithm that processes logged data from a sensor platform moved through an environment.

### 7.4.2 Simple SLAM Implementations

The most basic SLAM implementation [11] would simply place all the measured landmarks on a global coordinate plane and update the robot position based solely on the data collected from the inertial navigation system. In this case, there is no error correction which will result in measurement error that accumulates over time. For noisy
sensor systems, the error would likely be significant enough to render the SLAM algorithm very inaccurate for data logging runs of nontrivial length. However, this basic implementation is useful for demonstrating the basic transforms necessary perform basic SLAM calculations.

The basic transforms that comprise the SLAM algorithm simply take data from the local coordinate plane attached to the robot and translate them into a global coordinate plane that should be consistent between samples. Using the basic inputs to a SLAM algorithm, which are the change in robot position (as measured by odometry and IMU systems), and the locations of all the landmarks local to the robot, a basic implementation has three main steps:

1. Calculate robot position: The change in position from the last sample is added to the last known pose to generate a new global robot pose.

2. Calculate global landmark positions: Using the new global robot pose, translate the local landmarks into global landmarks.

3. Add new landmarks to known map.

This forms a basic structure for most SLAM implementations; however the more advanced ones take into account error in all of the measurements, and use correction mechanisms to keep errors from accumulating. One extra phase that is not included in this very basic implementation is data association between the landmarks, which would determine whether landmarks are being spotted more than once over multiple samples. The simplest form of data association would be to compare each new landmark to the
known landmarks to determine if the distance between them is below a particular threshold. If a new landmark is very close to a known landmark, then they are considered the same landmark.

While this rudimentary implementation will work for extremely precise sensor systems or very short data collection runs, it will start to diverge wildly from the real robot pose and map very quickly. The more advanced algorithms use probabilistic mechanisms to model the error in the sensor systems as well as using landmark measurements and odometry to correct each other.

7.4.3 Probabilistic Implementations
The most prominent algorithms for handling uncertainty in real-world robotics applications use probability as a means of explicitly describing noisy systems. For example, a system with one state variable will be represented as a Gaussian curve with a mean that represents the most likely value of the variable, and a variance parameter that describes the shape of the curve.
This representation shows the amount of uncertainty in the approximation through the width of the Gaussian curve. The mathematical details behind probabilistic modeling of systems are beyond the scope of this paper, which will focus on the qualitative implications of these mechanisms. The general approach will be described in brief, but more detail can be found in the following sources [8], [9], [12], and [13].

These techniques can be implemented in a more useful application with a Bayes Filter, which stores all the state variables in this probabilistic representation in order to create a belief, which represents the current state of the world model. A Bayes Filter has two primary phases for incorporating new data into the current belief. The first phase of a Bayes Filter makes a prediction of the new belief based on the current control and environment measurements. The second phase takes into account the uncertainty of measurement and by correcting the prediction made in the first phase using the probability that the measurement is correct. The result is a new belief with the sampled data incorporated. While this provides the basic structure, a pure Bayes Filter cannot be
implemented without approximation, and there are a variety of techniques to actually apply the Bayes Filter, the most notable of which is the Kalman Filter.

The Kalman Filter is an implementation of a Bayes Filter that uses an internal model of the system to perform prediction. The Kalman Filter has a similar structure, described below:

1. Predict mean for variable – Using the model of the dynamics of the system, the next state is predicted using the previous state, the control and the measurements.
2. Predict covariance (multivariable variance) for variable – use a similar technique incorporating the model to predict the distribution based on the inputs.
3. Calculate Kalman Gain – A number that determines whether to trust the prediction or the measurement more, which will allow for corrections in measurement noise.
4. Update mean – use Kalman Gain to integrate measurement into prediction mean.
5. Update covariance – use Kalman Gain to perform correction of the uncertainty of the new state estimate.

There are a few additional requirements for a Kalman filter, specifically that the noise must be Gaussian and the model must be linear. While these work in some applications, most real-world systems have non-linear models, including most of those used in robotic systems. One solution is to linearize the model at each point so that the model appears to
be linear in the filter. There are two main approaches used to linearize the model, implemented in the Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF), which use Taylor series and the Unscented Transform, respectively [9]. The EKF is used as a relatively common SLAM implementation, and is detailed in the next section.

7.4.3.1 Extended Kalman Filter SLAM

The SLAM problem can be approached by using a single, large EKF [9] [10] [12], which includes all of the state variables and incorporates all of the inputs at each sample. In this case the state variables include the robot pose, as well as an ever-growing number of landmarks. While this will provide a solution to the SLAM problem that mathematically converges properly, it does have some serious practical issues. The root of these flaws lies in an assumption made while adding new landmarks to the system, as the number of state variables increases. The covariance matrix to handle these landmarks also grows, but does so quadratically with the number of landmarks. This is a major performance issue when working in environments that have large numbers of landmarks, as some environments may have thousands of landmarks that will be tracked. The other issue with this system is that if there is a data association error in matching new landmarks to known landmarks, the quality of the overall state estimation drops significantly. This brittleness to data association errors, which occur on a regular basis when working with feature-rich environments, also renders the system to sensitive to noise for real-world application [9].
7.4.3.2 FastSLAM

The FastSLAM algorithm [9] [11] [13] is one of the most effective techniques for factoring the SLAM problem into reasonable pieces. The basic change from EKF SLAM implementations is the assumption that landmarks do not need to be correlated with each other, and can instead be represented using individual EKFs. This is an improvement, because it no longer means that covariance matrices increase quadratically with number of landmarks; each landmark only must maintain a single covariance matrix of constant size. The basic algorithm is described in pseudocode below:

```
FastSLAM2.0(new_local_landmarks[n], control, cur_map, cur_pose)
    For each new local landmark
        New_pose = Predict pose(cur_pose, control)
        Use cur_pose to calculate global landmark
        For each known landmark
            Determine if new global landmark matches
            If a known landmark
                Add to temp list of known landmarks
        For each in temp list of known landmarks
            Determine connected pose position
            Perform Kalman Updates of new_pose using temp list
        For each new_local_landmark
            Use new_pose to calculate global landmark
            Determine probability of matching an existing landmark
            If known landmark
                Perform Kalman update on landmark in map
            Else if unknown landmark
                Add landmark to map
```

Figure 7-11: FastSLAM2.0 Qualitative Pseudocode

7.4.4 Particle Filters

One method improving the accuracy of SLAM algorithms is to use what is known as a particle filter [9]. In the case of a particle filter, the algorithm keeps multiple hypotheses of possible SLAM solutions, including a full map and robot pose. Each ‘particle’ receives the same input data, but makes slightly different approximations of the map and pose over time. While this may seem to provide a large amount of redundancy, it also provides a significant amount of robustness to the overall implementation. The
robustness of the particle filter functions much like genetic algorithms, which rely on making multiple guesses at a solution and then evaluate which hypothesis is most accurate. To improve the accuracy of the overall system, having a variety of solutions is useful because it improves the chances that one of the possible solutions is correct.

The basic flow for a particle filter is described for M particles below:

1. For each of M particles, independently perform their internal SLAM algorithm using the newly sampled data.

2. Create a metric to determine the most accurate solution out of the M particles.

3. Throw away particles with the lowest score.

4. Replace the particles that were removed with a duplicate of a higher scoring particle.

The phase at the end of the simulation, where inferior particles are removed and replaced with more viable solutions is known as ‘resampling,’ and is analogous to the fitness evaluation process used in genetic algorithms. There are a variety of techniques used in this process, which is a relatively active field in research concerning SLAM algorithms, but the details of which are beyond the scope of this paper.
7.5 User Assistance

In the development of semi-autonomous systems, it is important to create an interface that allows the user to manage what might be a very complicated system. A lack of intuitive interfaces that can relay complex information to a remote operator is a growing pain experienced by the robotics industry. The most familiar interfaces that operators have become acquainted with are the familiar joystick [14] with tank like operation. This type of control can be seen in some of the most simplistic remote control cars to the state of the art robotic arms used in space shuttles [15].

Once the remote operator is separated entirely from the vehicle and can no longer directly see what the robot is doing, reliance on the robot’s sensors becomes inevitable. By parsing and compiling the values from the robot’s sensors into an intuitive interface, the ability to effectively control the robot remotely becomes a much more obtainable goal.
8  Requirements

The system implemented by this project requires both the core components of the framework and a sample application of these tools in a semi-autonomous robotic application. While this application is not a full autonomous application of the tools, it does show some of the ways in which this sensor integration technology can be used.

Since the project consists of a framework and an implementation of a robotic system that makes use of these tools, there are two major sets of requirements to this project. The first portion of the system is the world-building framework components, which are designed to integrate data from multiple arbitrary sensor systems. The second part is the specific application, which needs to demonstrate the capabilities of the tools while also showing their efficacy in improving user experience for a semi-autonomous setting.

Another, more general, requirement, is that both components of this project are structured and documented in such a way to allow future developers to easily expand and apply the system in different scenarios.

8.1  Robotic Framework

The ability to read data from variety of real-world sensors and construct a viable world model that can be used for navigation is the primary goal of the world-building framework of this project. The reason for the importance of internal world building is to add a level of persistence to robotic sensor integration applications. By creating an internal world model, robots will be able to perform more complex operations in an
environment, which will be very useful in improving robots’ ability to operate in real-world scenarios.

A general, but major, requirement of the applications is that it be modular, extensible, and well-documented. Sensor interfaces of all kinds should be abstracted away from the core applications. Replaceable algorithms, such as those that run SLAM, telemetry, or sensor processing, should be partitioned from the core algorithms of the integrated system. Apart from simply being good programming practice, the application should provide a framework from which future work can be approached. The goal of this requirement is to allow rapid and incremental future development. If a researcher wishes to replace a sensor system, but has no interest in telemetry, no changes will be required to the telemetry portion of the system. Likewise, if a future developer wishes to substitute our sensors with more accurate devices, few changes to the software should be required.

While the application is developed in view of a mobile robot, it is intended that it ultimately will provide a jumping-off-point for other robotic applications. Even if a future developer wishes to replace the entire software system, it is hoped that the knowledge embodied in the application’s implementation will provide useful examples of how to effectively structure such a system.

### 8.2 Sample Application

The first phase of the world building system must be able to accept inputs from arbitrary sensor systems, which will largely be classed into two categories: environmental sensors and internal state sensors. The framework will standardize the input to allow for independent processing of the data from these sensor systems, so that the data integration phase of the system will not need to handle the details of extracting information.
The output of the system will need to be a ‘map’ that can be at least be used for internal navigation purposes and preferably in a human-readable form. This map should be a means to encapsulate the world model created by the system, and should include the following information at a minimum:

- Robot Pose – both 3D position and orientation.
- Robot Path – all previous robot poses.
- Landmarks – points in 3D space singled out for analysis by the environmental sensor system as being trackable and significant.

The sample application will integrate a stereo vision sensor, telemetry sensor and a SLAM algorithm in a total package that will allow the framework to be significantly demonstrated.
9 System Implementation

9.1 System Design

The software, Graphical Localization and Deterministic Orientation with SLAM (GLADOS), consisted of six main components: GLADOS Client, GLADOS INS Server (GIS), GLADOS Pylot (GP), Webcam-Server, GLADOS Wiimote Server, and GLADOS Host. The six systems communicated over a TCP/IP socket connection and would pass data packets between each other. Using this architecture, the system direction addressed the task to build a modular software system that would be readily extendable and would welcome future development.
Breaking primary tasks into multiple applications allowed the system to be freed from on-board processing. Each application communicates over a high-speed network thus allowing data to be collected at one location (i.e. on the vehicle or from an interface device), processed at another location (i.e. a command center) and then have the results sent to yet another location (i.e. a data logger or web tracking system).

9.1.1 Vision

The system created for this project has gone through several branches, taking four different approaches, before arriving at our final method. Ultimately, we used sparse matching, detecting strong vertical line segments as features. We did so with three matrix convolutions (to act as filters) followed by a matching phase.
9.1.1.1 Hardware

The vision system hardware consists of the camera head, the onboard processor and an off-board system.

The camera head is constructed with two webcams mounted 140 mm apart on near-parallel planes. The webcams used were Logitech STX Communicator Webcams, selected primarily for their low cost and immediate availability. They have a 42 degree field of view, and together can detect features within approximately the same field. While they work in this system, they are low quality cameras, and are subject to some drift, are not in perfectly parallel configuration, and generate camera artifacts depending on lighting conditions. Because the cameras were round, it was necessary to dismantle their cases in order to mount them securely in parallel. Difficulties mounting the camera capture cards against the head plate also caused the two devices to be pushed further out.
of parallel. This is partially corrected for in software, but camera and mount quality remains a limitation of the system.

Figure 9-3: The Camera Head

The onboard processor used is the mounted IBM Thinkpad laptop with a USB2.0 hub. The off-board processor is any laptop running the client application.

9.1.1.2 Software Evolution

At the first stage of the project, dense matching was implemented. Using an algorithm known as block-matching, a small set of pixels were selected from one image, and the second image sampled in order to find a set that most closely matches the first. This is repeated for every set of pixels in the first image.

Figure 9-4: Block Matching
This method proved to be effective only in ideal conditions. In order to calculate a reasonable covariance even between two blocks known to be matching, lighting must be near-perfect and the cameras must be rotated almost identically and be mounted on exactly the same plane.

With our camera system we could be reasonably sure they were mounted evenly, so we were able to add the optimization that matches are only looked for on the same level as the original block. This made changes in robot-pitch very detrimental to the vision system, but improved performance significantly.

Even so, performance was still closer to the theoretical worse-case than the best case. We were able to achieve very poor quality disparity maps (due to lighting, imperfect cameras, and mounting errors) at about 0.3 frames per second. It was decided that this sensor was not of sufficient quality or speed to use on our robot, and a new dense matching algorithm was pursued.

The second stage algorithm was a derivative of block matching using the KLT algorithm to detect matches between the images [16]. This resulted in a large accuracy increase, but at a great efficiency cost; frame rate dropped to around 0.1 frames per second.

At this point, it became clear that dense matching was going to be very complicated to get working correctly. Given all the tradeoffs between the two methods, it was decided to attempt implementation of a sparse matching algorithm. In the first iteration, and open
source implementation of FAST corners was evaluated [17], [18]. Ultimately, however, a simpler filter-based system was created, which became our final algorithm.

9.1.1.3 Algorithm
Our final algorithm consists of five parts, a capture object, a processing sequence using a detector object, a matcher object, and set of results. They are each called sequentially, in a loop.

Figure 9-5: The Main Loop

9.1.1.3.1 Capture
The capture object is really a small system that provides the key interface between the hardware cameras and the software. It consists of the camera drivers and a webcam web server onboard the robot and a stream connector off-board.
First, after some testing, we determined that for Linux, the open-source web camera drivers targeted at the Logitech STX series were sufficient for our purposes. A key concern for stereo vision is simultaneous or near-simultaneous capture of images from the camera head. If too much distance is covered between the instant when the first camera captures and the instant when the second captures, features will be harder to find, and calculated disparity may be incorrect. The driver we used doesn’t support multiple captures simultaneously, but instead relies on the Linux scheduler to determine when the next capture will take place. This tends to put “simultaneous” capture events not more than 1-10 ms apart, which is close enough to 0 ms for the velocities our robot is moving at, and proves to not be a limiting factor on the system’s performance.

Second, the images are served up using the open source utility webcam [19]. The server offers streams of JPEG images on two ports (one for each camera). The stream is updated whenever a new image is available, no image is in the process of downloading and the webcam server is ready. This proves to be somewhat unpredictable, depending on network latency, and delay from the server and network does limit the performance of the capture system.

Third, off board, is the stream connector. Two of these connectors poll the two ports, and when an image is available, they download it. Arriving as raw bits, they then assemble a Java BufferedImage object suitable for use in the rest of the system. The capture objects then inform any modules listening on the capture object that new data is available, and the algorithm proceeds.
9.1.1.3.2 Detector

The detector is responsible for separating features from the background of an image. To do, it uses three matrix convolutions, which act as filters.

First, the initial image is fed into an edge detector. Using an edge detection method (Canny, Soble, or a custom convolution matrix, depending on system version), all pixels that are not detectable edges are discarded.

Figure 9-6: The Detector
Second, the edges-only image is fed into the vertical line detector. Using a convolution matrix, only vertical pixel groups that are at least three-pixels high are kept; horizontal surfaces, diagonals, and curves that do not have a vertical segment are discarded.

This leaves a great deal of ‘orphaned’ pixels, which represent very small vertical lines not ideal for matching purposes (the vertical segment of a circle, for example). These are eliminated with a high-pass filter, which acts as a simple threshold transform, and filters out any features that are too small and therefore are experimentally unlikely to appear in a second image.
What remains is a solid black image, with all remaining pixels representing a feature or, if several features are tightly grouped together, a piece of a feature. These are then grouped according to some proximity threshold. Depending on threshold values set at compile-time, there are typically 10 to 200 features remaining in a 640x480 image (depending on the complexity of the scene). These are sent to a feature extractor, which builds up a table of the remaining features that appear on a certain color channel. This table represents the results from the detector phase. The system supports detection on an arbitrary color channel, but so long as the channel represents a primary color and is consistent with the other camera, little difference in performance, accuracy, or consistency is made.

9.1.1.3.3 Matcher
The matcher object is designed to correlate the features extracted from the left image with those in the right. Since, after the detector phase, we merely have a table of features, with their locations and a quality rating, we can simply threshold out features not within some
distance of our target feature, then compare the remaining features be position and quality
to determine the most likely match. This is a simple operation; matching features is not
processor intensive. It is, however, computationally complex; in the worst case, every
feature must be compared with every other feature, making this sub-algorithm run in \( \Theta(n^2) \)
time worst-case.

In practice, however, the fact we’re using sparse matching saves us; this complexity is
not especially limiting when we’re only comparing 10 to 200 features per image.

Finally, the matcher takes the feature pairs, and calculates the disparity between the two
locations using the standard method (as described in the background, above). Ultimately,
the matcher returns a list of features, with \( x \) and \( y \) coordinates taken from the camera
head, and a calculated \( z \) coordinate. Positions are relative to the local axes, but can be
transformed easily both from arbitrary units (which are a function of focal length) into
millimeters, and to a global set of axes.

9.1.1.4 Performance
The entire system can generate a single time-step’s worth of data in approximately 10ms,
with changes in scene complexity raising and lowering the time by up to 5 ms. This
allows for us to obtain speeds in the range of 6 to 15 frames per second captured and
processed.

Since the capture system spans across a network, latency is often an issue. If the capture
system does not serve at least 15 frames per second, we obviously will not be able to
process 15 frames. Likewise, even if the web server serves more than 15 frames, it is
necessary to successfully download all the frames (from two cameras) in order to process
a time-step (the system will abort if the capture system can not connect to both cameras
each time-step). This introduces a complication in that we must not only consider the
health of the vision system onboard and off-board the robot, but also the health of the
connection between the two.

The system performs precisely and is highly repeatable. In a test setup, the camera head
was presented with a very high-quality vertical edge to detect in an otherwise
uncomplicated scene. Correlation accuracy, distance accuracy and repeatability were
tested by comparing the results from two time-steps with known distance values and each
other.
The results revealed that the system is largely consistent between time-steps, as the same value was read for features in sequential steps. Correlation accuracy in this test was high; our predicted correlation points were indeed marked as the same point on both images. Distance accuracy was not as clear; because of our un-adjustable focal length in hardware, our output units end up as a value related to the focal length. By factoring out this value, the real distance value was obtained only within 25%. This is significant error. However, adaptations in other modules (namely the SLAM and INS systems) will largely correct for this uncertainty.
9.1.1.5 Strength
Testing reveals three primary strengths of the vision system. First, the vision system is fast. Written in Java, and thus running on a JVM, 10 frames per second is a reasonable speed, and comparable to similar sparse systems running on a virtual machine of some sort. Second, the system is precise; if put in front of the same scene twice, it will chose largely the same features, with the same ratings, and calculate the same distance. This is remarkably useful (perhaps critical) to functional SLAM systems and, in our development experience, precision vastly outweighs the importance of accuracy in FASTSLAM style navigational systems. Third, the system is remarkably straightforward. By using matrix convolution to accomplish feature detection tasks (and by leveraging Java’s powerful BufferedImage objects) it makes much of the typically complicated work of finding and matching features into basic image operations. Thus, by simply reading the source code (and perhaps this accompanying document) the program flow, design tradeoffs, and suitability to various tasks are immediately evident.

9.1.1.6 Flaws
The system does have several downsides. First, feature detection with convolution is not as reliable or predictable as the complicated algorithms it replaces (such as FAST corners). In simple scenes, such as the testing setup above, good features are easy to find. In more complex situations, it is difficult to determine what features the system will find (and therefore, if they are the best available features). Second, the matching process is not as accurate or efficient in complicated scenes as we would like it to be; the simple comparison of location and quality is fast and works, but if the threshold isn’t set

---

2 During development, a comparable system written in C#, running on Microsoft’s .NET 2.0 VM was compared. It also averaged 10 frames per second on a slightly faster test machine.
properly, features at distance can be matched erroneously (since they will appear close together). This is largely a problem associated with sparse matching algorithms in general. Finally, the system values precision over accuracy; it is not suited to applications where it must be known exactly how far away a given feature really is. It is not suited to true object detection and recognition, but rather is designed specifically to meet the needs of a SLAM module that has other sensors (such as INS) available.

9.1.2 Positioning and Telemetry

The goal of the INS is to provide an interface to collect inertial measurements as the vehicle is in motion and to report the new orientation and position of the vehicle to the SLAM sub-system. During the initial development phases it was identified that two key features would be necessary for the INS to be successful in the overall system and be a useful development for future research.

The INS needed to be robust enough that future expansion and experimentation with unforeseen sensors must be possible and reasonably easy while at the same time the INS must be intuitive to use so that future developers can utilize the existing INS without investing a great deal of time in order to learn the intricacies.

The second feature that was found necessity for was a robust communications platform. The INS would need to communicate with the host system and rapidly send data in a reliable fashion.
9.1.2.1 Microcontroller

Figure 9-10: SunSPOT

It was initially proposed to develop a microcontroller base from scratch, but this idea was quickly dissolved when the cost, manpower and time considerations were found to be beyond the scope of the project. Finding an existing base that is well documented and easy to develop with was crucial for the INS. As different options were explored, an emerging technology from Sun had emerged as clearly fitting the criteria for the basis of the INS. Small Programmable Object Technology (SPOT), from Sun Microsystems, are a small, wireless, battery powered experimental platform that has an array of built-in sensors as well as the ability to easily interface to external devices via standard serial I/O or General Purpose I/O (GPIO) [20]. These devices have a full JVM implementation that runs natively on the built-in ARM 9 and allows for Java applications to take advantage of the embedded peripherals, including an IEEE 802.15 radio module, 3 axis digital accelerometer, analog and digital GPIO and USB interface. This small device was a perfect match for the INS system’s implementation.
Utilizing the built in features and appending additional sensors via the open GPIO, an effective and reproducible INS could be built with many possibilities for expansion. Since the majority of the INS would be using features that were, by default, on the SPOT, a majority of the documentation and development process had been already been done and a community had been built to encourage future development [20].

9.1.2.2 Sensors
In order to effectively determine the orientation and displacement of the vehicle over time, multiple sensors were researched, but the most useful, low-cost and available solutions that were found was the accelerometer that was bundled with the SPOT and a gyroscope package from SparkFun.Com. With little modification of either the SPOT or gyroscope, the INS was hardware complete within an hour of physical build time.

9.1.2.2.1 Accelerometers

![Figure 9-11: Accelerometer Axes on the SPOT](image)

The SunSPOT has a ST Microsystems 3-Axis 2g/6g Inertial Sensor which was ideal for use the INS\(^3\). The sensor had two settable modes of operation (2g or 6g). Since the likelihood of the system exceeding 2g in normal operation was extremely unlikely, the 2g mode was used. This gave a finer resolution in within the exact range that was necessary for usable data.

\(^3\) Specifications are available under the part number LIS3L02AQ.
The accelerometer was mounted directly over the center of rotation as well as in the center of gravity on the platform. This gave the INS the ability to treat the entire vehicle as a single point when doing calculations. This greatly simplified the math necessary for effective use.

9.1.2.2.2 Gyroscope
Gyroscopes measure the angular rate of rotation around an axis. By utilizing a gyro and integrating its values, the total pitch, roll or yaw can be determined. Most electrical gyroscopes on the market today measure multiple axes.

The IDG-300 from InvenSense was used. It offers dual axis measurements, has a straightforward interface to connect to, and has a maximum rate of 500 degrees per second and a low drift rate of 0.014 degrees per second. This gyro was attached to the SunSPOT via 3 of the analog GPIO pins and the +5v supply line. The returned values were yaw and rotation of the vehicle.

Like the accelerometer, the gyro was also mounted in the center of rotation and center of gravity on the vehicle in order to reduce the complexity of the calculations needed for the INS.

9.1.2.3 Communications and Software
The USB hub and laptop that was attached to the system had a limited amount of bandwidth for USB devices, this lead to the USB link becoming unresponsive and unreliable when the maximum bandwidth was reached. The decision was made to separate the INS from the system attached to the vehicle. In order to accomplish this, the
802.15 radio was used on the free range SPOT along with a base station SPOT. By using an intermediate system to connect to the SPOT and relay its values over a socket connection back to the GLADOS Host, the USB hub was freed on the mobile system and helped to reduce USB bandwidth issues.

After the raw INS data has arrived at the GLADOS Client, it still needs to be processed. This task is handled in two parts; a displacement calculation, and a rotation calculation. Each task is executed once each time new data arrives (each time-step) excluding the first time, which is used to calibrate the sensors.

To calculate displacement, accelerometer data is used. Three values arrive each time-step; an acceleration vector on the X, Y, and Z axes, measured in mm/sec². Arriving from the network connection, they are given to the INS Manager object, which carries out the main loop of INS processing. The values can be filtered at this point; the INS Manager allows the addition of any modular filters that take and return three acceleration values.

**9.1.2.3.1 INS Algorithm**

Each Manager has three Integrator objects (one for each axis) specifically for the integration task. The Integrator encapsulates the entire math needed for transforming accelerations into useful data. This task is a classic calculus problem; given discrete points of an unknown function, one must integrate said function. The approaches are numerous. The first implemented was the well-known Simpson’s rule. In this approach, the function is, in effect, estimated, using quadratic polynomials, and the integral approximated using a combination of midpoint and trapezoidal approximations. In theory,
this approach is far more accurate than either the midpoint or trapezoidal approximations, while is far faster than other adaptive quadrature based algorithms.

**Equation 9-1: Simpson's Rule**

\[
\int_a^b f(x) \, dx \approx \frac{b - a}{6} \left[ f(a) + 4f\left(\frac{a + b}{2}\right) + f(b) \right]
\]

In practice, however, Simpson’s rule was more trouble than it was worth. To accurately estimate the underlying function \( f \), took a great deal of CPU time and, more significantly, a great number of data points that wouldn’t be available until many time-steps had passed. While interpolating points was an option to avoid this, it would likely have cancelled out the algorithm’s accuracy advantages by introducing many small sources of error.

Thus, the simpler Trapezoidal rule was implemented.

**Equation 9-2: Trapezoidal Rule**

\[
\int_a^b f(x) \, dx \approx (b - a) \frac{f(a) + f(b)}{2}
\]

The simple trapezoidal rule simply calculates the area under the trapezoid created by the time-step’s length (on the x axis) and the acceleration (on the y axis). The approach is fast and for the limited data we have, reasonably effective.

To use this equation, each Integrator must be state-full; they must save the last time-step they were invoked at, the last acceleration reading they received, and the last velocity they calculated. Ultimately, each integrator returns the single integration of acceleration (the instant velocity) to the INS Manager.
To calculate rotation, the angular accelerometer (which behaves largely like a gyroscope) returns x and y axis angular velocity. The conversion to angular displacement is straightforward, but the original value returned is the digital translation of an analog value; the units are not degrees per second, but arbitrary units between 0 and 1024. They must be converted, which the INS manager does. This conversion necessarily introduces a granularity into the INS system; movements at a low rate (less than 0.03 degrees per second) will be completely missed. It’s worth noting, however, that these movements would be minuscule oscillations likely lost in noise, in any case.

Ultimately, the system calculates the total displacement (really, the velocity vector, in three dimensions) and total rotation every time-step, typically outputting a new position every 6 milliseconds. This likely represents the fastest subsystem in the GLADOS operation. It accurately captures movement with accuracy and precision sufficient for SLAM applications, and does so even unfiltered.

9.1.3 Simultaneous Localization and Mapping

The Simultaneous Localization and Mapping (SLAM) system, as it is implemented, is primarily an output for the data gathered by the sensor systems. Closely linked with the SLAM implementation itself is the visualization for the SLAM module, which allows a user to see the path and map that the SLAM algorithm has created. Because this is primarily a framework for general use, careful attention has been paid to enhance the flexibility of the system. The SLAM algorithm only needs a set of local landmarks and a change in pose to operate, which means any future sensor system will only need to
provide information in one of those two formats. All of the inputs are passed through configurable filters to adjust for overall unit scaling and coordinate system corrections.

The SLAM particle provides the core of the SLAM implementation, which keeps track of the map and robot path. The basic SLAM implementation, without visualization and UI components, is shown below in Figure 10-12.

To interface with the rest of the software system, the SLAM module has acts as a listener for the each of the sensor inputs, and an overall process function is called whenever a new stimulus is ready. All new inputs need to pass through both the scaling filter and the coordinate correction filter before being processed by the overall SLAM filter. Both of these filters can be configured by the user to account for differences in measurement units and coordinate system.

### 9.1.3.1 Development Process

The development process for the SLAM system was more difficult than some of the subsystems as it inherently is dependent on sensor data from external systems. There were several early iterations of the system that started with a preexisting Matlab implementation, but there were a
number of issues found which made it unusable. While it did have full probabilistic modeling, it
the synthesized data it used to demonstrate the implementation was deeply intertwined with the
SLAM algorithm itself. Upon closer inspection, many parts of the SLAM algorithm cheated by
looking at real values for landmarks, or using perfect data association. Combined with a difficult
to follow structure, this implementation was scrapped in favor of a full Java system. Java was
chosen for an implementation environment due to much stronger typing, as well as improved
development tools available the Eclipse IDE. Also, much of the development for the rest of the
system was moving over to a Java platform at this point, which made a Java-based SLAM
implementation a natural solution.

In order to test the system, there were several attempts were made to create a separate data
synthesis tools, but these were eventually rendered unimportant once datalog recordings from the
real robot were created. A separate application was, however, useful in playing back the
datalogs, which allowed for adjustments to the input scaling and coordinate system filters in an
offline setting without the rest of the system running. While this IDE was only added at the end
of the development cycle, it proved useful in analyzing the results of the system.

9.1.3.2 Basic Transform SLAM

The implemented SLAM algorithm is one of the most basic implementations, which
simply translates the local landmarks and robot pose and translates them to a global
coordinate system. However, SLAM algorithm has been both into elements that can be
treated as particles, as well as multiple phases in the processing of each particle. The
structure of these phases is primarily designed to facilitate future development of the
system by allowing developers to swap out single sections of the SLAM algorithm and thereby speed up the development process.

The generic phases of the SLAM algorithm, in execution order, are listed below:

1. **Preprocessing** – This phase is a catchall for any adjustments to the data before the main SLAM algorithm starts, such as reordering of the landmarks, or culling landmarks that are too close together.

2. **Predict** – This phase primarily uses the control to determine the new robot state.

3. **Data Association** – This phase determines which new landmarks correspond to known landmarks. To perform this computation, this phase relies on the predicted robot pose.

4. **Update Pose** – Performs an update on the robot pose using the known landmarks to correct the original prediction.

5. **Update Landmarks** – Uses the newly updated pose to update all of the landmarks that have been spotted in the landmark measurement.

6. **Post Processing** – A variety of techniques could be included in this phase, such as culling landmarks that have not appeared when they should appear.

Each of these phases is implemented as a separate class with a default function to perform processing. For future development, these classes only need to be extended to add the new functionality to the system.
The actual data structures use a mix of libraries to perform vector math operations, including the `vecmath` library included with Java3D, as well as vectors from the JAMA Linear Algebra Library. The primary data structures used to represent the quantities are detailed in the list below:

- **Pose** – The basic representation of the robot pose is a pair of 3-element vectors, one vector to hold the linear position \((x, y, z)\) and another vector to hold the rotational position \((\Theta_x, \Theta_y, \Theta_z)\), represented as angles measured counterclockwise around each of the axes in radians.
- **Control** – The control inputs represent the change between robot poses, and are internally represented in the same way that robot poses are represented.
- **Landmark** – Each landmark is represented as a single 3D point, which is internally implemented as a single 3D vector. There is no distinction between global and local landmarks internally.

In order to group the system components in a more meaningful manner, two groupings exist:

- **PointMap** – This is the single-object representation of a map, which consists of a group of landmarks. Within the algorithm, PointMaps may represent either a global map or be used for a local landmark map.
- **Stim** – This is a wrapper interface for inputs to the SLAM algorithm, which facilitates a simple interface where the type of input is decoded inside the SLAM algorithm. Either a Control or a PointMap may be used as a stim to denote the INS or vision system Inputs.
At its most basic, noise-free implementation the SLAM algorithm can be described below:

1. Determine new Pose using Control input
2. Determine global position of the local landmarks using the new Pose
3. Add new global landmarks to the current map

The basic transforms necessary for this implementation requires being able to transform the change in pose and local landmarks to the same global coordinate system. These basic transforms are included as a core part of the SLAM implementation and can be used by any processing phase that can use them.

The control transform is the simplest, as each control input represents the change in robot pose since the previous pose, so they can simply be added as vectors to combine multiple controls together. To create a new pose, a control input can be incorporated using only matrix addition.

The more complicated transform is determining the global location of the landmarks, as it involves both a translation and a rotation. The 2D version of the transform is illustrated in Figure 10-13. In this picture, there are three interrelated vectors to handle. The global position vector for a landmark can be determined using the vector sum of the local vector to the landmark and the 3D pose of the robot. However, this transform must also account for rotations in the robot pose. This is accomplished by first negating the rotation of the robot pose, and then adding the transformed local position vector to the pose 3D vector. The Java3D vecmath library allows for simple rotation transforms, which are used in this
phase to perform the actual rotations. The internal implementation of the vector transforms makes use of a standard 4x4 matrix to record all of the transform effects [21].

![Diagram of vector components](image)

**Figure 9-13: Vector components of transforming landmarks**

Another feature built into the SLAM module is an automatic logging feature that can save all of the inputs to the SLAM system in a binary form that can be played back in the offline IDE. These output logs can be converted to a human-readable file using the IDE as well.

### 9.1.3.3 Visualization

The visualization module for SLAM uses a simple 2D top-down map that shows the current pose, all past poses, and landmarks. For reference, the X and Y axes are also shown. The implementation simply uses standard Java 2D library components to perform the rendering. There are a number of additional features, such as the ability to change the center location of the view by clicking on the map, switching the color scheme of the map, and GUI front-ends for the scaling and coordinate correction filters. More details and images of the visualization module are in GLADOS Host.
9.1.3.4 GLADOS SLAM IDE

The offline GLADOS SLAM IDE was created out of a need to be able to examine previously recorded data runs with the ability to reconfigure sections as necessary. This ability to use recorded files makes it much easier to determine how effective the SLAM implementation is by showing a visualization of the entire map and path, as well as the ability to step through the entire recorded data run. The main user interface is shown below in Figure 10-14 which shows the playback controls, the mapping visualization, and a console log of the process.
Figure 9-14: GLADOS SLAM IDE User Interface

One of the more important tools that the IDE can offer is the ability to adjust several of the parameters that the SLAM routine uses: the unit scaling for inputs, and the coordinate systems. In real sensor applications, the data frequently comes back in arbitrary units which will need to be converted into data that can be meaningfully used. The IDE allows the user to set a uniform scaling value for the vision and INS data separately, which is a rudimentary tool to place both on the same coordinate plane. The coordinate systems for the vision and INS systems are both configurable as well, since it is possible to have
these systems using reversed or twisted coordinate systems. With the IDE the coordinate systems can be rotated or reversed to ensure that all systems are using the same coordinates.

The IDE also offers several tools for playing back a loaded data log file, so that a user can step through the process slowly to see the effect of each new input to the SLAM algorithm. The data logs are stored as a list of stimuli to the system, which are either a set of local landmarks from the vision system or a change in pose. The full SLAM algorithm will only run when a set of local landmarks is encountered, and if multiple pose changes occur between vision updates, the SLAM algorithm simply sums the pose changes until the next vision update arrives. Through the IDE, the playback controls allow the user to skip to the next stimulus that produces a change, skip to an arbitrary stimulus, or run the entire data log.

One difficulty with using the data logs generated by the real-time version of GLADOS is that the logs are in a binary format that is not human-readable. The IDE has an export feature that allows the user to translate the logs into a format that details each of the inputs to the system. Analysis of this human readable log is useful in determining exactly what sort of information is coming into the system.

9.1.3.5 Future Work

Because this is a theory-heavy component of the system, there are a great many ways that it can be improved as future goals. One of the largest ways it can be improved would be the implementation of a true probabilistic routine that can mathematically represent the
uncertainty in the system. In addition, implementing a real particle filter system with a
resampling system would be a significant improvement, as it would add a significant
level of robustness to the overall SLAM implementation. There are some smaller
software additions that would be useful for improving the system, such as user-
configurable particles and a simulator that can test and compare SLAM results with a
known truth model when the noise parameters of the system are known explicitly.

9.1.4 Platform

The foundation of the vehicle was a Create from iRobot. This base is very similar to the
Roomba robots that can be found in many homes but has been modified for easier
integration into robotics projects. An open API over a common serial connection allows
the Create to be controlled by simply issuing serial commands. This allowed for quick
development of a mobile platform that could be controlled from the system sitting on top
of it. The Create is able to navigate with a zero-turning radius and can transport well over
30 pounds of cargo while retaining a battery life that allows for hours of continued usage.
Scattered throughout the Create are mounting points that allow for structures and
equipment to be easily and securely mounted to the base. Using steel stand-offs, custom
cut Lexan pieces and mounting hardware a sturdy platform was built. It offered precision
mounting the gyro and SunSPOT over the iCreate’s center of rotation as well as secure
attachment of the laptop and cameras, USB hub, associated cabling.

9.2 System Integration

Integrating the subsystems was done in view of the requirements that the system be
modular, extensible and platform independent. This directly led to the use of multiple
computers of varying architecture, solid design patterns (i.e. the Observer Pattern) and a distributed system approach.

9.2.1 Architecture

![System Architecture Diagram]

Figure 9-15: System Architecture

9.2.1.1 GLADOS Client
GLADOS Client was a small application, written in Java, that ran on the SunSPOT. This application would wait for a connection on the 802.15 radio link. Once a connection to the INS Server was established poll the gyroscope and accelerometers then send the values back to the host. In the event that exceptions or errors would occur, the system would cause the last LED on the SPOT to turn red, then the system would try to gracefully recover or, in a critical error, die. See the appendices for application documentation and source code.

9.2.1.2 GLADOS INS Server

![INS Server Controls]

Figure 9-16: INS Server Controls
GLADOS INS Server acted as an adapter between the GLADOS Client and GLADOS Host. Since the intent was to make the entire system as modular as possible, and communication over a 802.15 module was a specific implementation, the decision was made to separate the 802.15 communication ability from GLADOS Host. This allowed GLADOS Host to use socket connections for all of its communications and leaves the ability to create a specific implementation adapter on the implementer. This frees the system from a hardware constraint.

9.2.1.3 GLADOS Host
The GLADOS Host was the software powerhouse behind the project. Its purpose was to initialize the SLAM engine, manage the incoming connections and generate a live display for the user. The application consisted of two windows. The first was the Controller Window. This would display the incoming image streams for the user, allow the user to configure the connection to the vehicle and allow them to save the system logs. The second window was the SLAM visualization. This would display the map generated by the SLAM engine in an intuitive way for the user to utilize during remote operation.
9.2.1.4 GLADOS Pylot

GLADOS Pylot was a small two part client-server application. The server would adapt incoming messages from the client on a socket connection to outgoing messages on a
serial connection. The client would take user intuitive input from the user and translate it into a command for the iRobot Create and send the message to the GLADOS Pilot server. This would enable the vehicle to be driven remotely over a network connection. Since both the client and the server use sockets to communicate, it is very easy to replace the client with any other application that would send hex values across a socket. This allowed for quick integration into GLADOS Host.

9.2.1.5 Webcam-Server
This is a third-party application, licensed under the GNU GPL, that connects to a camera on a USB connection and generates an image stream that is accessible over the network directly [22]. Two instances of this application were run, one using port 7777 and one using port 8888. This allowed both stereo cameras to be used simultaneously. An added benefit of using an independent daemon for each camera was in the event one camera failed or a daemon was prematurely killed, the remaining camera and associated image stream were unaffected. A java applet that came bundled with the software was used as the basis for the image visualization.

9.2.1.6 GLADOS Wiimote Server
The GLADOS Wiimote Server (GWS) acted as an intermediate step that would link a Nintendo Wiimote over a Bluetooth connection to the GLADOS Host over a socket connection. GWS would additionally allow for basic emulation and connection management for the Wiimote.
9.2.2 Hardware

As seen in the figures above, three distinct computer systems run portions of the GLADOS system: The Control System, the Assistant System, and the Mobile System.

The Control System used varied, as all of our software is platform independent. It was tested on both an IBM Laptop running Windows XP and a MacBook running OSX. While any modern computer should be more than sufficient to run this part of the system, it runs the most resource intensive part of the GLADOS application. It is responsible for running the GLADOS Host.

The Assistant System used was a MacBook running OSX. It ran the GLADOS INS Server and the GLADOS Wiimote Server and Interface. Since this system is responsible for connecting to a remote SunSPOT sensor, so be able to communicate via the 802.15
wireless protocol. In our testing, this was accomplished by connecting a second SunSPOT to the Assistant System.

The Mobile System used was an IBM Thinkpad, running a recent Ubuntu distribution of Linux. It was strapped to the top-plate of the robot, and was primarily manipulated remotely. It ran the Pylot and the Webcam-servers. It was attached, through an un-powered USB hub, to both cameras and, via a USB to Serial converter, to the platform’s control inputs.

![Computer Systems Used](image)

**Figure 9-20: Computer Systems Used**
GLADOS system: The Control System, the Assistant System, and the Mobile System.

The Control System used varied, as all of our software is platform independent. It was tested on both an IBM Laptop running Windows XP and a MacBook running OSX. While any modern computer should be more than sufficient to run this part of the system, it runs the most resource intensive part of the GLADOS application. It is responsible for running the GLADOS Host.
The Assistant System used was a MacBook running OSX. It ran the GLADOS INS Server and the GLADOS Wiimote Server and Interface. Since this system is responsible for connecting to a remote SunSPOT sensor, so be able to communicate via the 802.15 wireless protocol. In our testing, this was accomplished by connecting a second SunSPOT to the Assistant System.

The Mobile System used was an IBM Thinkpad, running a recent Ubuntu distribution of Linux. It was strapped to the top-plate of the robot, and was primarily manipulated remotely. It ran the Pylot and the Webcam-servers. It was attached, through an unpowered USB hub, to both cameras and, via a USB to Serial converter, to the platform’s control inputs.
10 Results

Following integration of the subsystems detailed above and testing of each subsystem’s units, it was necessary to test both the subsystem as units and the entire integrated system. Some of these results have been alluded to above, and follow in full.

10.1 Unit Testing

Each subsystem was independently tested as a unit, in view of its ultimate purpose. By independent testing, any bugs in the module were worked out and solved before integration. Each module was developed iteratively, with each iteration meeting more of the requirements and design goals.

10.1.1 Vision

The vision system was unit tested, and can be evaluated independently of the rest of the system. By setting up simple situations with known objects and known distances both accuracy and precision of the vision subsystem can be determined experimentally. While it is difficult to predict what objects will generate features of interest to the Detector portion of the vision system, we can encourage the system to choose certain reselected objects by setting up a solid color background and placing only a few high contrast objects.

Three situations were set up in this manner, with one object of interest per scene. This causes two strong features per image; the left and right sides of the high-contrast object. All three situations took place with different backgrounds and different foreground
objects. The first used a uniform white background with a darker crate for an object. The second uses a more distant, but less uniform, white background and a dark grey board as an object of interest. The third inverts the colors, using a dark red background and a light yellow object.

The data collection from the subsystem’s unit tests revealed that the units actually returned from the vision system were arbitrary, but uniform. It was expected that the units that should have been approximately equal to 10 millimeters, but results indicated that each unit was actually about 59.5464 mm. This represented a significant deviation, and our math was checked to find the error. Ultimately, a piece of epoxy was found on one of the camera’s focus knob, making our expected focal length incorrect. The error was correctly simply by adapting our expectations to the new conversion factor.

Eight features in these three scenes were used, with this conversion factor, to allow a calculation of the general accuracy of the system.

<table>
<thead>
<tr>
<th>Actual Distance (mm)</th>
<th>Measured Distance (Units)</th>
<th>Measured Distance (mm)</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>132.08</td>
<td>2.59259259</td>
<td>154.3796037</td>
<td>16.88%</td>
</tr>
<tr>
<td>111.76</td>
<td>1.609</td>
<td>95.81018755</td>
<td>14.27%</td>
</tr>
<tr>
<td>119.3</td>
<td>2.121212</td>
<td>126.3105777</td>
<td>5.88%</td>
</tr>
<tr>
<td>300</td>
<td>6.086956</td>
<td>362.4564301</td>
<td>20.82%</td>
</tr>
<tr>
<td>137.716</td>
<td>2.9166666666</td>
<td>173.6770539</td>
<td>26.11%</td>
</tr>
<tr>
<td>113.538</td>
<td>1.609</td>
<td>95.81018755</td>
<td>15.61%</td>
</tr>
<tr>
<td>130</td>
<td>2.59259</td>
<td>154.3794494</td>
<td>18.75%</td>
</tr>
<tr>
<td>300</td>
<td>6.086956</td>
<td>362.4564301</td>
<td>20.82%</td>
</tr>
</tbody>
</table>

**Table 10-1: Vision Data Results**

This data reveals an average accuracy error of 17.3% out to a maximum testing distance of three meters. At least out to this distance, this error does not appear to be linked to distance and no obvious correlation to object size or color could be determined.
Figure 10-1: Accuracy has no correlation to distance

By normalizing the data from similar points in the same scene, we can also calculate the precision error; we can predict how likely the system is to repeatedly measure a distance provided it selects the same feature to measure. We calculate that this has a precision error of 12.16%. It should be pointed out, however, that is a conditional probability, and does not take into account the probability that the vision subsystem will select the same feature a second time. While it is believed the probability of reselecting the same feature is high enough for SLAM applications, it was not measured in any of our tests.

10.1.2 Positioning and Telemetry

The inertial navigation subsystem was tested for three values: drift without movement, repeatability of angular displacements, and overall accuracy over a set of movements. Drift is always a problem of accelerometer based systems. To determine how much of a problem it would be with the INS subsystem, our three-axis accelerometer package was left on a stationary table for one hour, a time significantly longer than the longest run our
integrated system would be faced with. After this time period, drift was noticeable, even with the inherent noise, but was significantly less than 0.1g on every axis. This result is well within the tolerances of our integrated system.

Second, the accuracy and precision of our rotational measurements were tested. By calculating the angular velocity per digital unit, we were able to establish a “truth” turning rate. Experimentally, we could then rotate the robot a known angle at an approximately known rate and compare the resulting INS reading.
The ultimate goal of the angular velocity is to point the time-step’s linear displacement in the correct direction. This testing revealed values sufficient to transform a time-step’s displacement vector appropriately.

Lastly, we ran full-motion test runs of the INS system, including both displacement and rotation. These tests would start at a known position, (0, 0, 0) at an angle of 45 degrees, typically, and run some known distance. Simple tests would be displacements of two meters. More complex tests would then include a rotation and another displacement.

Accuracy is mixed. The final displacement from the test runs can be calculated against the known displacement. Three typical runs are shown below:

<table>
<thead>
<tr>
<th>Measured Displacement (mm)</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>141.5901657</td>
<td>29.20%</td>
</tr>
<tr>
<td>172.7541606</td>
<td>13.62%</td>
</tr>
<tr>
<td>180.2652723</td>
<td>9.87%</td>
</tr>
</tbody>
</table>

**Table 10-2: Displacement Error Rates**

With an average error of 17.57 %, this system is not especially reliable even over short distances. The issue is not unexpected, however; the system clearly is subject to significant noise. The GLADOS application has, effectively, a filter slot which is unfilled in our example implementation and the data shows the consequences.
Nevertheless, for the requirements of our system, this accuracy should be sufficient to demonstrate the SLAM subsystem and to act as a visual aid to a remote controller.

10.1.3 SLAM

To test the SLAM as a unit, it was necessary to virtualize the sensor and telemetry inputs. To do so, truth-based models were recorded and fake telemetry and vision readings generated from the model. By introducing various levels of noise into the fake sensors, or eliminating noise altogether, the subsystem can be shown to be correct. Because the subsystem was built modularity in mind, this virtualization did not affect the inner-workings of the subsystem whatsoever.
Through testing, we can show that, as would be expected, SLAM runs with perfect INS telemetry are the most successful.

### 10.2 System Testing

Finally, the integrated system was tested. These tests really occurred in two parts: the system test, and the actual integration process itself.

First, the actual integration process is the best way to test the modularity and extensibility of the final system. By implementing the integrated GLADOD via the observer pattern, we were able to hook our separate subsystems into the same application simply by implementing the most-relevant interface (one for sensors, one for cameras, one for control, and one for SLAM processing) and registering it as a listener in the appropriate place. Overall, our results indicate that it is possible to add a new sensor to the system by modifying two files; GLADOS must eventually know what type of data the sensor will give it, and GLADOS must know what function to call when it wants new data. Any other idiosyncrasies of the sensor happen independently of the GLADOS application. As each module improved iteratively, the entire system was able to take advantage of increased functionality.

Second, the entire system was tested together. Five controlled data runs were made that can be compared to truth. The results of these runs were captured via GLADOS's recording mode, and were played back and compared to the truth data. Examination showed that these runs suffer a combination of the errors associated with the INS and Vision subsystems; landmarks were off by the typical error of about 20%, while SLAM
showed the INS was faulty by its typical 20%. It appears that the integrated system is more or less exactly the sum of its parts.

One long-distance test was made via remote control, the vehicle was driven unknown distances (well over 100 meters) over real-world indoor terrain, with multiple turns. The GLADOS application stayed stable and plotted large path. Once the robot was out of view of the controllers, the camera and map system proved effective enough to control the robot for more than 30 minutes until network errors caused the stereo cameras to fail, and the test was stopped.
11 Conclusion

The GLADOS application and associated hardware do make up an effective, extensible base. Throughout our testing, we have shown that we have the ability to modularly change system parts, quickly and easily. Because the system runs over a network, it could run on any reasonably modern platform with an arbitrary amount of processing power behind it. Because the integrated system doesn’t depend on any single subsystem, (and, in fact, can run with any of them disabled) our goal of insulating the subsystems from the system as whole has been successful. The implementation of subsystems on top of the system show how actual sensor and processing pieces can be fit into the GLADOS application. All the subsystems are functional, but are subject to limitations.

Likely more important than the actual implementation of each subsystem is the body of knowledge built into them. For example, the vision subsystem is not suited for real-life applications in an uncontrolled environment, but is a useful example of how sparse matching actually works. The SLAM algorithm doesn’t support many of the particle-based probability constructs of more successful applications, but does include hooks where such a construct could be added and tested.

The extensibility of the system is a major advantage when considering further work. Future projects into any particular subsystem can replace the subsystem of interest, and see how it affects the results. For example, an improved stereo vision system could be
implemented simply by replacing the relevant classes in GLADOS. One could then simply use the existing framework to unit test the stereo algorithm.

Of particular interest to us is the concept of fully virtualized sensors. As we unit-tested the SLAM system, we created fully virtual sensors that would get data not from reality and not from pre-set values, but from a computerized “truth model”. It might be of interest to test new subsystems, algorithms, and robots by completely replacing other subsystems with virtualized versions. Allowing some sensors to run in real-space, while making others run in a truth-space, (and switching which ones run where) might have interesting applications. Ultimately, this system should allow these types of experiments to be conducted rapidly, and with a minimum of wheel-reinvention.
12 References


13 Attached Appendicies:
Appendix A: Application Notes
Appendix B: Javadocs
Appendix C: Source Code