Abstract— The UCSC Rover Team will participate in the NASA Sample Return Centennial Challenge held at the Worcester Polytechnic Institute (WPI) in Worcester, Massachusetts as their Capstone project. The objective of this competition is to construct a planetary rover that is capable of autonomously finding and collecting samples which are randomly distributed across a large outdoor playing field.

I. BACKGROUND

The objective of the NASA Sample Return Centennial Challenge competition is to construct a planetary rover that is capable of autonomously finding and collecting samples which are randomly distributed across a large outdoor playing field. The competition simulates the real-life challenges of planetary exploration by prohibiting the use of any Earth-based technologies such as GPS, ultrasonics, or magnetometers. Any planetary rover must be at least partially autonomous, due to the time delay in interplanetary communication, especially a rover on a distant body such as Mars. However, while most currently operating rovers are still remotely operated by space agencies, our robot must be entirely autonomous. It must be fully prepared before deployment and reliably plan and execute all required tasks in the field without any outside help.

Our robot will be UCSC’s second try at completing this competition. Last year’s team did remarkably well, despite other teams having a one-year head start. Their robot was nearly able to complete Level 1 of 2; it successfully navigated to and from the pre-cached sample, but did not successfully collect it. Our team plans to complete Level 1 by improving on last year’s design.

This challenge will serve as the Capstone project for our undergraduate degrees. Although the competition has been running for two years, no team has yet completed the challenge, making this an excellent opportunity for us to prove ourselves and represent UC Santa Cruz in a tough, high-profile, world-wide competition.

II. MECHANICAL

The mechanical portion of our robot consists of a chassis, a drive system, a steering system, and a sample collection system.

A. Chassis

Our chassis needed to support the weight of all of the rover’s components while traversing a playing field of well-trimmed grass with mild elevation changes. Our robot’s weight was limited by the competition’s imposed maximum of 60kg. It wouldn’t come into contact with obstacles or other robots, so it only needed to be strong enough to support its own weight and the weight of samples that it carries. It needed to be stable enough to keep the cameras from jiggling while going over terrain, but since the terrain wasn’t too rough, that wasn’t much of a challenge. A simple design would serve.

There were many possibilities to consider for the design of the robot chassis. Last year’s team chose to use a “Rocker-Bogie” style chassis design, citing high payload capacity, high stability, and good obstacle-climbing at the cost of complexity. We created a CAD model of their chassis, shown below.

![Last year’s rocker-bogie chassis.](image1)

The six-wheeled rocker-bogie design distributes the weight of the robot more than a four-wheeled design, meaning that it maintains drive traction even when some of its wheels have left the ground, and allows the robot to drive over soft surfaces such as sand without sinking in. Four of the wheels must be mounted on two rocking segments which move separately from the robot chassis, as shown below.

![Top view of last year’s chassis showing rocker sections](image2)

The left and right segments must be connected in the center by a differential gear system which causes one side to lift whenever the other one dips. The result is that the robot chassis remains at the average angle of the two pieces, reducing the rocking of the chassis by half in comparison to other suspension systems [1].
Finally, driving all six wheels allows the robot to climb over obstacles up to twice the wheel diameter without any wheel leaving the ground. This is very useful on rocky terrain.

The main drawbacks of the rocker-bogie design are the complexity and cost. Each of the six wheels must be driven, and at least four of them must steer. This means 50% more wiring, 50% more moving parts, and an approximately 50% increase in cost.

Our first step in working with the chassis was to fully evaluate last year’s “rocker-bogie” design. We determined that the front and rear wheels were driven, but the center wheels were not. This hobbles the obstacle-climbing capabilities of the rocker-bogie.

On the rocking portions of the chassis, the front wheels were powered and the middle two wheels were unpowered. The unpowered wheels were mounted closer to the rocking axis such that they bore three times as much weight as the powered wheels, cutting drive traction severely.

The two rocking sections were not joined by a differential gearbox, but by a piece of ⅜” threaded rod running through the chassis. This threaded rod bent under strain, causing the two rocking segments to cave inward, tilting the drive wheels somewhat and making the robot drive in a slight curve. The two rocking sections also rocked independently without a differential gear, so that the chassis did not maintain the average angle of the two. Thus, I determined that last year’s implementation of the rocker-bogie design reaped none of its benefits.

Next, we took a look at the competition field and tried to determine just how much weight distribution, suspension, and obstacle climbing was necessary. If the field was challenging enough, we might have decided to modify last year’s design in order to gain the benefits of the rocker-bogie configuration. However, since the field is fairly well-kept and almost uniformly grass with only small rocks or tree roots to trip on, the requirements of the drive system appeared to be fairly mild.

The obstacle climbing aspect of the rocker-bogie appeared unnecessary, since rocks are small and sparse. The weight distribution appeared unimportant, since we won’t be driving over sand or mud, and the chassis shouldn’t be rocking too much, aside from when the robot exits and re-enters the starting platform. Upgrading to a real rocker-bogie would have been very expensive and complex. It would require the purchase of an additional set of drive motors and H-bridges, as well as a differential gearbox.

One additional design constraint was that we wished to be able to execute a “tank turn”- a turn without forward or backward motion, in order to simplify the navigation software. Additionally, steering with all four wheels would allow us to strafe directly left and right to make small adjustments as the robot approaches a sample.

Various chassis designs were proposed, such as a tricycle design with one steering wheel in the front and two drive wheels in the back. However, we feared that this design was susceptible to tipping or getting stuck in potholes. A four-wheeled design has a wider, more tip-resistant base and more redundancy. If one wheel becomes stuck, the others can still drive until the disabled wheel is free. Even if one wheel fails completely, the others may be able to compensate and carry the robot through the round.

In order to accomplish all of these requirements while maintaining stability and handling unexpected terrain, we decided to go with four-wheel drive and four-wheel steering. The number of drive wheels guarantees traction even if some wheels leave the ground or become stuck, and the rectangular base gives us a robot which is resistant to tipping.

This chassis was easily prototyped by removing the rocking segments from last year’s frame and simply attaching the existing drive wheels at each corner. Essentially, we used the same frame, but removed the troublesome center wheels.
The strength of this frame was somewhat questionable, since it was made from 1/16” wall stock and had many holes drilled through it. Once the rover was equipped with our 40lb batter pack, the main beams of the frame sagged visibly under its weight. Additionally, if one wheel of the robot were to leave the ground during competition, that part might sag even worse. While the beams would never come close to the point of breaking, the flexing could stress our electronics enclosures and the PCBs mounted on them. To estimate the amount that the beams bent, we could assume Pure Bending. The radius of curvature $p$ would be equal to the Young’s modulus of 6065 aluminum ($E = 70GPa$), multiplied by the moment of inertia of the box beam denoted ($I = 3.58 cm^4$), divided by the moment induced by the maximum weight of the rover unrealistically concentrated at the center of the frame to simulate a worst-case scenario ($M = 169 Nm$).

For the 1/16” beam:

$$p = \frac{E * I}{M} = \frac{70 GPa * 3.6 cm^4}{(1/2 * 34 in * \frac{1}{2} * 80 kg * g)} = 3.3 \text{ deg}$$

The only difference between 1/16” beam and 1/8” beam would be the increased moment of inertia of 6.54 cm$^4$. For the 1/8” beam:

$$p = \frac{70 GPa * 6.5 cm^4}{(1/2 * 34 in * \frac{1}{2} * 80 kg * g)} = 1.8 \text{ deg}$$

The only other significant effect of this change would be the doubling of the weight of the frame. To reduce this effect we decided to only replace the rectangular base and continue using lighter beams for the upper portions. Since our robot was underweight, this was not a problem.

The bending of the beams was no longer noticeable to the naked eye. The 1/8” beams were also much easier to tap threads into and didn’t deform as much when bolts were tightened down onto them firmly. The frame held up nicely during transportation to and from WPI.

B. Drive

Driving was successfully accomplished last year on all four wheels. The only problem they encountered was that the robot rolled down hills when it wasn’t driving up them- it had no braking system and went into neutral while idle [Last Year’s Report]. This year, our drive system would need brakes or feedback control to keep it steady on a hill.

First, we took a close look at last year’s drive system. The system consisted of two drive wheel kits from Trossen Robotics. Each kit included a matched pair of left-and-right car window motors fitted to symmetric aluminum “worm” gearboxes. The gearboxes had spaces for optical encoders to mount for approximate distance or drive speed encoding. Some of the encoders seemed to be of different resolution than the others and some were missing. The wheels were 6” pneumatic wheels with shallow treads, mounted onto the ½” aluminum drive shafts via roll pins.

![Figure 6 – Trossen Robotics wheel and encoder kit [8]](image)

Last year’s drive system couldn’t exceed the competition speed limit of 2 m/s at no-load speed. Their 6” diameter wheels moving at their unloaded speed of 150 RPM would only drive the robot at:

$$v_{max} = \frac{150 \text{ rev/min} * 1 \text{ min} / 60 \text{ s} * \pi * 6 \text{ in} / 1 \text{ rev} = 1.2 \text{ m/s}}$$

The motors were designed for torque, not speed, and far exceeded the torque necessary to move the rover forward in real life tests. However, the actual torque of the motors was unknown, as the “datasheet” available was extremely limited. We decided that the motor kits were a good choice. Their high torque would be able to drive larger wheels if necessary, and their reversibility was their only flaw, since it would allow the rover to roll down a hill. With the encoders replaced, we could use feedback control to take care of this later on.

The wheels didn’t have much in the way of treads, and they were fairly small. Since the competition field could theoretically be wet, we decided to upgrade them to larger wheels with deeper treads. We picked out some 8” pneumatic wheels from Andy Mark, and some aluminum hubs designed for mounting them onto a ½” shaft with a double keyway [8]. 8” wheels would increase the max speed of the rover, but not beyond the limit:

$$v_{max} = \frac{150 \text{ rev/min} * 1 \text{ min} / 60 \text{ s} * \pi * 8 \text{ in} / 1 \text{ rev} = 1.6 \text{ m/s}}$$

![Figure 7 – Trossen Robotics wheel and encoder kit](image)

Our drive shafts were equipped for roll pins, not keys, so we had to have holes drilled into the hubs to accommodate the roll pins. Our first attempt at this turned out to be wrong when it came back from the machine shop. The shafts wiggled by up...
to 13 mils in real life, but the Solidworks model didn’t show this. The result was that the heads of the machine screws used to hold the wheels onto the hubs interfered with the gearbox and couldn’t be installed properly. We had to modify the Solidworks assembly provided by Trossen to reflect the jiggling of the actual shaft and have the holes re-drilled. Assembling the wheels and mounting them onto the motors required long #10 machine screws with hex heads - a somewhat unusual part which took us about a week to obtain and slowed down our progress somewhat.

Finally, we encountered one last problem during assembly - it was impossible to press the roll pin in while the machine screws were in place, so we needed to file down the edge of the holes which interfered so that the screw could be slid in from the side. The screw could then be inserted from the side after pressing the roll pin through the hub and shaft and the hub could be properly mounted. The tires were then inflated to 30 psi and all four were assembled and mounted onto the gearboxes.

After all the assembly problems were worked out, the wheels fit onto the motors nicely and turned smoothly. They provided excellent traction on the linoleum floor of the lab, the concrete walkway outside, and on the grass of East Field. The only times that they were observed to slip were on dirt.

C. Steering

In order to facilitate navigation, we decided that our steering system would need to support tank turns. For sample collection purposes, it was decided that we should also aim for crab drive. This means that the robot can turn all four of its wheels 180 degrees, allowing it to spontaneously drive in any direction without turning the chassis. The steering would also need to be able to push small rocks out of the way, or wiggle the wheels free of obstacles such as tree roots without getting stuck, meaning that each steering quadrant would require a high-torque actuator. Steering would need to be precise enough that all wheels could point in approximately the same direction, as well. These stringent requirements made steering design a somewhat difficult and expensive process.

Steering was accomplished last year through the use of four hobbyist analog servos linked to the pivot shafts by small steel bars. The servos could be commanded to a specific angle through PWM control. This worked well enough for them to navigate to and from the sample in the competition. However, they were not able to drive very straight [2].

The steering did not appear to be in working order when we inherited it. The pivot shafts were ½” threaded rod and, like the rocker axle, sagged under load, resulting in awkward angling of the wheels. Over time, the angle between the servo and the pivot shaft also seemed to drift due to slippage of the part which connected to the bar on the shaft which it was attached to.

The steering shaft was secured to a beam which was, in turn, secured to the chassis with bolts. It was mounted onto the beam with skateboard bearings, which are very tough and can withstand both radial and axial loads, allowing for smooth motion under a variety of loading conditions.

We decided to replace the steel bars and threaded rod with a new steering shaft and drive system which would be more sturdy and reliable. The options we came up with were chain-and-sprocket or timing belt. Either system would allow us to increase torque if necessary by replacing the sprockets/pulleys to increase the gear ratio. We chose chain over belt for cost reasons, and because we discovered that sufficiently strong servos with sprocket heads were easily available and compatible with the current mounts, which simplified the design and made it quicker and cheaper to implement.

Further research and observation of other robots participating in this competition would have shown that such a system was an unusual solution. Most steering systems appeared to use direct drive from a DC motor with a gearbox. Position feedback was typically accomplished through the use of an encoder [3].

The first prototype of the chain-and-sprocket steering system was assembled and tested on the same threaded rod steering shaft as last-year’s system. However, the rods were now filed down to form flat surfaces for the sprocket’s set screws to bite into, preventing slippage of the sprocket on the shaft. ANSI #25 roller chains were made to size using a chain tool and looped around the servos’ sprocket heads and the steering shaft sprockets. The sprockets were adjusted to the same height by stacking washers underneath them to boost them up. The same bearings were recycled from last year.

The whole system was mounted on a foot-long piece of 1/16” wall stock from the previous year’s frame. It was predicted that chain tension might be an issue, so slots were machined into the beam, allowing the servo to be moved back farther if the chain got loose.

The servos chosen were hobbyist sailboat winch motors. They were analog like last year’s, and had slightly lower torque - 180 oz-in instead of 200 oz-in. The 3:2 gear ratio on the sprockets would make up for that difference and place the torque somewhere around 270 oz-in - an increase over last year.

One full mechanism was assembled at first, to work out any obvious problems. A few things jumped out as us, such as slippage of the fender against the nuts that secured it to the
steering shaft. This solved by sandwiching it between lock washers. Once the nuts were all tightened correctly and the lock washers in place, the servo was able to apply its full stall torque without mechanical failure anywhere in the system. We then went on to test all four quadrants simultaneously and put the robot on the ground to test it out.

The wheels were observed to turn to slightly different angles, and to struggle far too much. It appeared that the PID tuning of the servos was such that they would give up once within 5-10 degrees of the target angle, possibly due to the fact that they were sailboat winch servos meant for low-torque applications. Although the torque was theoretically higher than before, the tuning of the servos was such that they didn’t apply that full torque when necessary.

When driven forward, the robot quickly went duck-footed and began to bend the all-thread. Additionally, the chains seemed to loosen after running for just a few minutes to the point that they began to fall off. This led us to decide to redesign the steering, at least partially, and so it was disassembled. To address the problems, we knew we had to change the design either partly or completely. After much back-and-forth discussion, it was decided that we should stick with the overall idea to save time, but improve the design in some ways. We decided to:

- Increase torque. This would make our robot less susceptible to getting stuck on rocks. Theoretically, it might also improve the tracking of the servos by giving them an easier job. The meager effort they put out near the set point might then be enough to get them all the way there.
- Thicken the shaft. The all-thread was not an acceptable load-bearing part and would need to be replaced with a thick solid shaft or thick pipe stock to prevent bending.
- Add position feedback. If the servos couldn’t all point the same angle, then the steering system could be damaged by the torque of driving duck-footed.

Torque could be added easily by increasing the gear ratio with a larger sprocket. We also decided to replace the shaft with a ½” solid aluminum shaft. We chose this over a pipe in order to reduce the necessary size of the box beam around the steering, and to reduce the cost of the bearings. This design was CAD’ed and the parts ordered.

After more discussion about the design, it was brought up that a DC motor might be a more appropriate solution than a servo. While watching videos of the other robots from the competition, we discovered that almost all of them appeared to use DC motor direct-drive for their steering.

If we were already planning to add position feedback, then the use of a servo would no longer be necessary, at all. A DC motor could be made to behave like our servos, or better, using feedback control. A DC motor could directly drive the steering shaft without the use of chain, eliminating the backlash of the sprocket system and the danger of chain stretching, and simplifying the whole assembly drastically. The feedback control could also give us better performance than any of our servos, since we could tune it ourselves to make it work however we wanted. Professor Elkaim offered us four suitable DC motors after hearing about this idea.

The Bodine gear motors were 11.6RPM 24V motors with a 240:1 ratio, resulting in 40 in-lbs of torque. Running at 12V, they would take almost five seconds to execute a 90 degree turn. The rear shaft would rotate 240 times faster and was small enough to fit through a hobbyist encoder.

With the high speed of the rear shaft, we determined that we wouldn’t need a high resolution encoder. We could use a kit that would fit directly onto the rear shaft without any modification or circuit design.

At this point, we realized that the Bodine motors were too good to pass up. They provided far more torque than any servo we could afford to buy, they were designed with position feedback in mind, and the shaft was thick enough to serve as the steering shaft on its own. These features would save us a tremendous amount of time and money. We could return all parts of the current steering system (~$500), aside from the parts which were custom machined (~$120).

Having decided to use the Bodine motors, we needed to choose encoders. We settled on AM-103 capacitive incremental quadrature encoders, which provide 48 PPR at their lowest resolution setting. They were extremely cheap at under $25 apiece and came with a variety of tools for attaching them to a variety of shafts.

This time, we worked closely with the machinists to develop an extremely simple, durable, and effective design. The only moving parts were the motor and the drive assembly. The motors were mounted face-down against an aluminum mount plate bolted to a piece of box beam. The ½” keyed shaft stuck down through the bottom of the box beam, then through a thick Delrin washer which provided a durable low-friction bearing surface. The washer was sandwiched between the box beam and drive assembly by the weight of the robot. A new hole was drilled into last year’s aluminum fender and a 1/8” keyway was reamed into it for the shaft to mate with via key. The key was glued into the keyway of the Bodine shaft using cyanoacrylate adhesive, and a hole was drilled into the side of the fender so that a set screw coming into the side of the fender could press the key into place and hold the whole assembly together in the event that the wheel left the ground and the steering system had to support its own weight.
After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and use the naming convention prescribed by your conference for the name of your paper. In this newly created file, highlight all of the contents and import your prepared text file. You are now ready to style your paper; use the scroll down window on the left of the MS Word Formatting toolbar.

The encoders were mounted onto the rear shafts of the motors using a mount system lasercut from ¼” medium-density fiberboard. The mounts were secured into place using tab-in-slot construction and fit so tightly that they needed only a dab of hot glue to keep them totally immobile. The encoders were screwed down onto the MDF and spun smoothly with no problems.

Since the encoders were incremental, not absolute, they needed an initial calibration to center them. To accomplish this, we attached limit switches to the steering system which clicked when the wheel turned all the way to the left.

We first tried turning the motors back and forth to see if anything unexpected occurred. They moved slowly, but surely and with no issues. Next, one quadrant was loaded with 16.5lb and the motor was tested again. It continued to turn at the same speed as before, taking five seconds to turn 90 degrees, despite the heavy load and increased friction.

Once software was written for the steering system, it appeared to work perfectly. The steering angle was very precise, and the Bodine motors were very difficult to back-drive, so maintaining the steering angle while driving would be very easy.

D. Sample Collection

Competing at level two of the competition would require a mechanism capable of collecting and storing a variety of samples. The most important rules to consider during the design process were the “sterile handling” rules. We were not to damage the samples in any way— even just scratching the paint— so they would need to be handled relatively gently. No two collected samples could come into contact with each other at any point during the match, so we would need to store them separately, possibly in different compartments. Samples would have to comprise the majority of material collected, by mass, meaning that we couldn’t just scoop up huge chunks of dirt or collect every rock and squirrel we encountered. Finally, the samples would need to be “easily removable” by the judges without any special instructions, so we couldn’t use strong adhesives or finish the round with the sample held in a death grip.

The samples we needed to collect ranged from the pre-cached sample— a bright white cylinder with a ferromagnetic hook protruding from the top for easy handling— to various distinctly-colored, regularly-shaped objects, to “non-ferrous metallic objects” inscribed with high-contrast abstract symbols.

For level one, we would need to collect only the pre-cached sample above. In level two, we would need to collect that sample plus at least one additional sample, with prize money based on the number of samples and their “difficulties”. The sample collector could be reconfigured between rounds, if desired. Our initial brainstorming process resulted in three general concepts for a sample collector type: the scoop, the claw, and the carousel.
The scoop concept was loosely based on a broom and dustpan. It involved lowering a platform to the ground and using a lever arm or linear actuator to move the sample onto that platform. The platform would then be raised up into the body of the rover and the sample would be pushed off into a storage bin somehow. This approach would require three separate actuators. A linear actuator would raise and lower the mechanism. A second would need to open and close the scooping mechanism. The third would move sample into separate storage containers.

The claw approach was loosely based on a “poop scoop”, and would involve lowering a claw mechanism from directly over the sample, pinching the sample, and storing it. This would require high precision from the collection alignment procedure and three actuators. A linear actuator would raise and lower the Claw once the rover was correctly positioned over the sample. A second actuator would close the jaws around the sample, either encapsulating it or using friction to secure it. Once the sample had been brought back up to the rover chassis, a third actuator would somehow place it into its storage container.

In an attempt to eliminate one of the three actuators listed above, the carousel was devised to combine the scoop and storage mechanism into one motion. A cylindrical drum would be divided into multiple storage compartments by rotating fins. The open pie-slice-shaped orifice in the front of the Carousel is where the samples would enter. Once the mechanism was lowered down over the sample, the fins would rotate, scooping the sample up onto the base plate. Once inside the collector, the sample would already be stored, and a new compartment would be exposed to collect a new sample. The rotating fins would not only collect the samples, but would also separate them from each other, keeping them “sterile”.

Due to confusion with the machine shop, the first ‘prototype’ mechanism was a very expensive Carousel design. The ¼” aluminum base plate was 20” in diameter. The 1/16” rolled aluminum shell was attached to the outer edge of the base plate with 4-40 screws. The top plate was similarly attached to the shell at a height of 10”. In the center of the Carousel was a 1” aluminum hex shaft, vertically positioned between the upper and lower plates.

The Hex shaft had positions for up to six fins to be screwed into it, radiating outward. Once the rover had positioned itself directly over the sample, the carousel would be lowered to the ground by a winch. To accomplish this, a Bodine motor was fixed to a three inch diameter spool. One inch wide polyester webbing was attached to both the spool and the sample collector. The motor would raise and lower the Carousel as dictated by the microcontrollers. After the sample had been aligned in the center of the collection area, a 12V, 10 RPM motor would rotate the hex shaft and fins to scoop the sample up onto the platform and into a compartment.
The electrical components of the robot consisted of the power system, sensors, and microcontrollers.

### A. Power

The power needs for the rover were primarily set by the motors and on-board computing. It would need to supply a fairly high current for up to two hours of competition time to keep these components running. It would also need to be rechargeable and last for many cycles, as well.

The four drive motors operate at 12VDC. Limited data was available for them, but load testing them proved to be sufficient for our needs. The motors draw a max current of 3.42A when stalled.

The four steer motors were 24VDC Bodine Electric gear motors with a 240:1 gear ratio. In order to run the Bodines at 24VDC, we would need to use either a 24V battery or a step-up DC converter. Putting the batteries in series for 24V would cut our capacity in half. We briefly considered putting some in parallel for most of the rover functions but only a few in series for the Bodines. However, we decided that this would complicate the charging process too much. A boost converter could provide us with 24V from the 12V rail, however, it would be inefficient and waste some of our precious battery power. Despite the slow speed of the Bodine motors, we decided to run them at only 12 volts. Taking five seconds to make a 90 degree turn is boring for a human, but not too bad in the scope of a thirty minute competition. At this operating voltage the motors pull 0.375 amps while stalled.

The rover’s on-board computing is accomplished by an HP ProBook operating at 19V and pulling 4.74A max. The PC is powered by a DC-DC converted hacked from the PC car charger. The PC power supply draws from the 12V distribution rail.

Other parts of the rover drew far lower currents, for the most part. The power usage for those parts is listed in Table 2.

The large voltage step downs could be accomplished using the PTN78020 switching regulator made by Texas Instruments. The PTN78020 is a high-efficiency, step-down Integrated Switching Regulator (ISR). The wide-input/wide-output voltage range made this regulator a perfect choice early in the design stage of the rover. The 7-36V input range added flexibility to our power supply design, whether 12V or 24V configurations would be used. The regulator is also capable of supplying up to 6A, making a good choice for our distribution boards which would supply many components in parallel.

All switching regulators generate a voltage ripple. The ripple in the voltage is problematic for the sensitive equipment such as rangefinders and microcontrollers. The 5V rail was regulated from the PTN78020 rail using linear regulators to smooth out the voltage ripple as seen in Figure 16.

Originally, a 3.3V rail was included in the power system for limit switches. This was eventually deemed to be unnecessary complexity, as the Uno32 is 5V tolerant.

### Table 2 – Power requirements for rover components

<table>
<thead>
<tr>
<th>Device Name</th>
<th>V_{\text{max}} (V)</th>
<th>I_{\text{max}} (A)</th>
<th>Quantity</th>
<th>Total Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>19</td>
<td>4.74</td>
<td>1</td>
<td>90</td>
</tr>
<tr>
<td>Drive Motor</td>
<td>12</td>
<td>3.42</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Steering Motor</td>
<td>12</td>
<td>0.375</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>Range Finder</td>
<td>5</td>
<td>0.033</td>
<td>2</td>
<td>0.33</td>
</tr>
<tr>
<td>Range Finder Servo</td>
<td>6</td>
<td>0.5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Micro Controller</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>2.4</td>
</tr>
</tbody>
</table>

The large voltage step downs could be accomplished using the PTN78020 switching regulator made by Texas Instruments.
Three sets of eight 12V lead acid gel cells were donated to the rover team, each holding 5Ah of charge. The cells were rearranged into parallel configuration, supplying 40Ah overall. To achieve this configuration, the ‘Power Squid’ approach was used. This approach used 10 AWG wire with female spade crimps to connect each positive and negative lead to a respective terminal. This terminal spliced each bundle of conductors to a single 10 AWG wire to be distributed to the rover loads.

The Power Squid requires an excess of wire and very large connectors. This increases the overall size of the battery box and number of solder joints to be made. An alternative approach would be utilizing multi-connection Butt Crimps to connect batteries in a similar configuration with considerably less wires and solder joints. However, the Power Squid allowed more flexibility, since batteries could be disconnected and taken out of the battery box at any time with no problem.

The theoretical max current for this rover is 20A. To accommodate this possible current draw from the batteries, 10AWG cable was used. The 12 volt power and ground was directly connected to the control box through a Dean Connector pair.

The use of Cadence and the PCB layout process turned out to be a very helpful tool for this project. After taking EE 174 in the fall, the online tutorials and libraries allowed us complete control over PCB design. Furthering our experience through Professor Peterson, Circuit Cam and Board Master were also added to the tool belt. This allowed us to take a circuit full circle from the schematic to PCB layout to a prototype PCB board routed in-house within a single day.

The PTN78020 regulators require external components to set the output voltage. We designed a breakout board, which I constructed using the Cadence suit. The linear voltage regulation is implemented directly on the distribution plug boards in the front and rear. The schematics for the plug board and regulator board can be seen on the following page.

The competition rules dictated that no part of the rover could be powered prior to the activation of the “Master Power Switch”. This rule was strictly enforced, and not even quiescent current draw was allowed. The Master Power Switch would have to be the only switch engaged by the judges at the start of the round. All robot operation must proceed from it automatically.
The rover was also required to have an Emergency Stop Button. When this button was pressed, power would have to be cut to all rover systems instantly. This meant that even if the computer had to be cut off instantly, with no shutdown procedure allowed.

We inherited an Emergency Stop Button from the previous year, but it broke partway through. We replaced it with a cheaper Chinese button rated for 10A and placed it in series with the positive terminal of the battery, such that it could engage and disengage power to all rover systems.

We originally chose a 15A breaker as our Master Power Switch. Booting the onboard laptop on startup was tricky, since the laptop would need to have its power button pressed to boot. Our first idea was to have the laptop’s supply in parallel with the battery and use its power button as our Master Power Switch. An NPN transistor with its base connected to the USB +5V rail through a current-limiting resistor would be triggered when the laptop was booted and the USB ports turned on automatically. This transistor would energize the coil of a high-current relay with 12V. The relay was placed in series with the Emergency Stop so that it would turn power to the entire rover on or off.

Unfortunately, the judges decided that our laptop’s switching power supply drew too much current while the rover was powered down. It would need to survive an extended spaceflight to another planet without running the main battery dead. The blue LED and switching of the supply would have been problematic. Instead, we took the relay out of the circuit and used it in a different way. We opened up the PC and placed it in parallel with the power button. We connected the base of the transistor to a digital I/O port on an Uno32, instead, and placed that Uno on the 5V rail. Now, the Emergency Stop was the Master Power Switch, and the breaker was left in the On position perpetually as a fuse. When the E-stop was disengaged at the start of the match, the Uno32 would wait a second, then press the laptop’s power button for a quarter of a second and release it.

B. Microprocessors

The physical layer consists of an array of hardware and sensor components with different communication and power supply needs. See the following page for the system block diagram. The first devices that were researched were microcontrollers. The early decision to use evaluation boards was made because designing microcontroller boards was beyond the scope of the project. It was obvious from the start of the project that there would be lots of hardware and sensors that needed PWM outputs and analog or digital inputs. However, there were additional sensors considered that would require other types of interface such as I2C, CAN, or PWM input. These factors necessitated a fast but cheap microprocessor with lots of analog and digital pins and as well as support for a range of I/O methods. The original budget expected microcontrollers to cost no more than $50 apiece. With those constraints, the PIC32 UNO, MSP430G2 Launchpad, STM32 and the STM8S Discovery Boards came to mind. Other more exotic microcontrollers were expensive, or were hard to source. The below table breaks down some criteria that guided the selection process.

<table>
<thead>
<tr>
<th>Microcontroller Selection</th>
<th>PIC32 UNO</th>
<th>MSP430G2</th>
<th>STM32</th>
<th>STM8S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor</td>
<td>Microchip</td>
<td>Texas Instruments</td>
<td>ST Microelectronics</td>
<td>STMicroelectronics</td>
</tr>
<tr>
<td>Processor</td>
<td>PIC32</td>
<td>MSP430</td>
<td>STM32</td>
<td>STM8S</td>
</tr>
<tr>
<td>Price</td>
<td>$8</td>
<td>$7</td>
<td>$27</td>
<td>$15</td>
</tr>
<tr>
<td>IDE</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SPI &amp; I2C</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>UART Ports</td>
<td>All Pins</td>
<td>All Pins</td>
<td>2 Ports</td>
<td>3 Port</td>
</tr>
</tbody>
</table>

Table 3 – Microprocessor Selection

After careful consideration, the STM32 Discovery and PIC32 UNO were the most favored microcontrollers. Both the development board prices were below budgeted expenses for microcontrollers, they both supported a large range of hardware and software requirements in the original system design, as well as proposed hardware and software improvements. The final choice to use the PIC32 UNO was made due to a few additional reasons. While the PIC32 was more favorable because it could be programmed without a size limited IDE, the greatest benefit would be that most team members were already familiar with the PIC’s hardware and software library. Since a few important base software modules were already available, a lot of time would be saved by using the UNO instead of the STM32.

Taking into consideration the previous mentioned points, the only advantages of using an STM32 were lower cost and that it could support a more flexible UART system (any pair of pins could be configured for UART). However, we only expected to need one UART channel. Looking at online forums, there seemed to be more support for the STM32 than the PIC32, however, several professors on campus were experienced with the PIC32 and many other Senior design teams were using the PIC32. On all important points other than online support, the PIC32 stood even with the STM32. With time of the utmost concern, the team decided that sticking with a device it was already familiar with was a better choice despite having less online technical support available. The PIC32 UNO board was considered a tried and true option with existing software modules which would help accelerate the integration of hardware on the robot, and any help available from professors was considered enough to offset some benefits of having good online support.

Another major concern for the robot was how components would transfer data or interpret sensor input as well as communicate with the Linux computer. The options here were to use standard UART, SPI, I2C, CAN, or some other message framework. An early consideration was the use of a CANbus (Controller Area Network bus) protocol. Other message frameworks were considered overly complicated for the system and a conclusion was made that there would be too much setup overhead involved if we attempted to implement one. Considering that all the connections in the system would be more or less centralized by being located on the same singular vehicle.
A CANbus system uses differential RS-485 as seen on the following page, which can be helpful in high noise environments and has built in error correction when used with CAN controller boxes[4]. Other message frameworks are similar and commonly used with standardized hardware such as maritime sensors and aerospace subsystems. These bus frameworks were considered beyond the scope of the project complexity and tended to be expensive. However, UART communication using single ended RS-232 to differential RS-485 conversion was not ruled out as an option.

To implement full duplex differential signaled data transfer, a twisted pair bus could be run around the robot with termination resistors on the bus edges. Transceivers would be placed at appropriate locations to split off the signal wires to microcontrollers. To simplify wiring, half-duplex differential could be implemented. However, this method would require some extra software implementation work to handle token passing between transmitters to handle line sharing (Jones). A token passing protocol would be needed to prevent different devices from transmitting data over the bus simultaneously, which would corrupt both messages.

Other options considered were asynchronous and synchronous serial communications. The original considerations sided for use of an SPI network to communicate between multiple microcontrollers. However, SPI would require select wires to direct data to slave devices which would possibly complicate wiring more than necessary if a large number of microcontrollers and other SPI devices were chosen for the system. This led to the consideration of I2C which could be connected in a daisy-chain network to propagate data down a line of devices. This data transfer method seemed simple and easy to implement, however I2C was a synchronous serial protocol that only specified data rates of a few kBaud.

Another consideration was that the onboard laptop made the idea of trying to juggle data for robot control between just the microcontrollers or worry about timing overhead of the processors with synchronous messaging unappealing.

After exploring the more advanced message protocols, synchronous serial communication methods, and making some control technique decisions, the motivation for using standard UART became clear. UART was asynchronous and could easily be interfaced to USB on the Linux laptop. At the system level, the laptop was intended to handle navigation, image processing and centralize control of the rover. This was most simply implemented if each microcontroller communicated directly with the laptop rather than indirectly if a master microcontroller on a bus of slave microcontrollers. Using an asynchronous serial protocol also made more sense than synchronous since data between the microcontrollers and laptop only needed to be transferred on occasion to send commands, report completion of a command or report any obstacles detected.

At the system level, wiring was a major concern as there were lots of sensors, motors, and microcontrollers that would be on the robot. Deans connectors were used for electrical wiring but signal wiring would be another issue. Since the sensor hardware was not selected, a flexible design would need to be created to allow for any number of connections to parts of the robot. A key constraint to the wiring configuration for the robot would be to isolate signal wiring as much as possible from electromagnetic interference due to high current AC noise. Additionally, connectors would need to be easy to detach and reattach yet locking so that they do not fall out on their own during the competition. Molex connectors were originally considered, but they are expensive and time consuming to use in large amounts since shielded cabling, crimps and connectors would have to be purchased and assembled by hand. Another option would be to use double-shielded Ethernet CAT7 wiring. This would be simpler than Molex because Ethernet connectors are the crimps as well, making assembly much easier and faster.

To get the robot initially on the ground even though full hardware integration design was not complete, the original physical layer was significantly modified. However, there were still many wired signals that would run around the robot that required careful consideration to noise from motor and power wires due to the fact that a wire and cabling solution had not be implemented yet. This was done by adding EMC electrical metallic conduit along the sides of the chassis to shield signal wires from 12V high current wires that needed to power the front motor H-bridges. Also, to minimize the possibility of noise issues, the steering microcontroller was moved to the front so signal wires from the front encoders, steering limit switches, H-bridge PWM and direction pins would be as short as possible. Power and motor wires were also twisted to minimize high current AC noise from motor wires.

For the winter check off the, the message protocol involved exchanging single byte messages between the laptop and microcontrollers. The microcontrollers did their own calculations to ensure straight driving, speed control, self-calibration on startup and sensed obstacles. Since USB can be split by a hub and transmit/receive data through UART-Serial conversion, adding additional micro-processing devices as well as web cameras for image processing would be simple.

The original design of the robot required two microcontrollers. However, mechanical and hardware design changes necessitated a number of additional PWM and direction control pins. Without any motorized collection system, the winter check-off demo could still be accomplished with two microcontrollers, but any additional motor driven components would necessitate a third microcontroller.

The message protocol used for the winter check-off exchanged simple single byte commands to execute pre-programmed operations. A more complex module for sending actual data and feedback for the Linux computer to process would consist of a framing system to pack and unpack data. A working module was completed and implemented after the check-off. The following message framing table describes how a generic message would be formatted. While most messages only contain a few bytes of data, the message protocol allows messages to contain carry quite a bit of data such as infrared data, encoder information as microcontroller output or speed, acceleration, output torque, turning angle, or capture system maneuvers. Messages could also be multiplexed by grouping
them into categories since only 16 messages were originally designed into the framing format. The reason for not simply allocating more bits to allow for more message definitions was because we wanted to minimize the size of the smallest possible message. When the robot is travelling using D* full control of the robot is not very critical since the robot would only need to drive, turn, and stop at predetermined distances, angles or times respectively. However, when executing sample collection, and maneuvering the robot to align the sample collector over the sample, the laptop would need significantly more control of the robot to actively maneuver and handle information from the robot’s on-board sensors. By making the minimum message length shorter, the more complex, longer control messages would be minimally affected by shorter simple messages.

For a checksum, a 16-bit Fletcher algorithm was chosen. The reasoning for using Fletcher was that it is like a cyclic redundancy check but keeps a lot of the computational speed of simple addition checksums. Rather than a simple checksum which would be insensitive to the order of bytes, the Fletcher checksum ensures the order of bytes does not change. Fletcher is calculated by dividing binary data words into short blocks and treating the blocks as individual numbers[6]. This is important for data that may be sent over the course of several bytes because it preserves the ordering of bits in the data word. There are other more complex checksum algorithms but it was reasoned not to explore checksums further until they needed to be implemented and because this aspect is currently beyond the scope of the project.

Lastly, the message module also implements a circular buffer that can store up to 32 full messages. While this could be detrimental to the program memory of the microcontroller by taking up 672 bytes on every microcontroller that implements the protocol, the tradeoff would allow the laptop to control detailed aspects of the robot’s behavior.

Since some messages were sent in only one direction, for example drive wheel speed from the microcontrollers to the laptop, messages id values could be allocated differently going in the opposite direction, laptop to microcontrollers. Therefore, potentially 16 messages can be supported in each direction totaling up to 32 messages.

One planned design would consist of a number of box enclosures with female Ethernet connectors on a PCB that would spider out to female header pins to jump signals to nearby electronics. The enclosure would consolidate electronics to parts of the robot and EMC conduit could be used to run signal wiring between box enclosures.

Although the wiring method used for winter check-off could have been used for competition, it was very messy and prone to mis-wiring as well as faulty connections due to wear and tear. The envisioned solution was to implement an elegant cable wire solution with locking connectors. Such a system would make assembly, disassembly and hardware debugging of the robot quick and easy, a major priority due to the team’s busy cross-country travel plans to and from the WPI NASA competition. Options considered were Neutrik Ethercon jackets, and locking Molex connectors or JST connectors. Considerations for connectors and cable choice included expected weather conditions, noise electromagnetic interference from PWM motors, and wiring simplicity. In particular, a locking connector was desired because we wanted to easily be able to interchange devices but prevent accidental unplugging.

<table>
<thead>
<tr>
<th>Micro ID</th>
<th>MSG ID</th>
<th>MSG Length</th>
<th>Data</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
<td>&lt;=120</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4 – Message Framing

For a checksum, a 16-bit Fletcher algorithm was chosen. The reasoning for using Fletcher was that it is like a cyclic redundancy check but keeps a lot of the computational speed of simple addition checksums. Rather than a simple checksum which would be insensitive to the order of bytes, the Fletcher checksum ensures the order of bytes does not change. Fletcher is calculated by dividing binary data words into short blocks and treating the blocks as individual numbers[6]. This is important for data that may be sent over the course of several bytes because it preserves the ordering of bits in the data word. There are other more complex checksum algorithms but it was reasoned not to explore checksums further until they needed to be implemented and because this aspect is currently beyond the scope of the project.

Lastly, the message module also implements a circular buffer that can store up to 32 full messages. While this could be detrimental to the program memory of the microcontroller by taking up 672 bytes on every microcontroller that implements the protocol, the tradeoff would allow the laptop to control detailed aspects of the robot’s behavior.

Since some messages were sent in only one direction, for example drive wheel speed from the microcontrollers to the laptop, messages id values could be allocated differently going in the opposite direction, laptop to microcontrollers. Therefore, potentially 16 messages can be supported in each direction totaling up to 32 messages.

One planned design would consist of a number of box enclosures with female Ethernet connectors on a PCB that would spider out to female header pins to jump signals to nearby electronics. The enclosure would consolidate electronics to parts of the robot and EMC conduit could be used to run signal wiring between box enclosures.

Although the wiring method used for winter check-off could have been used for competition, it was very messy and prone to mis-wiring as well as faulty connections due to wear and tear. The envisioned solution was to implement an elegant cable wire solution with locking connectors. Such a system would make assembly, disassembly and hardware debugging of the robot quick and easy, a major priority due to the team’s busy cross-country travel plans to and from the WPI NASA competition. Options considered were Neutrik Ethercon jackets, and locking Molex connectors or JST connectors. Considerations for connectors and cable choice included expected weather conditions, noise electromagnetic interference from PWM motors, and wiring simplicity. In particular, a locking connector was desired because we wanted to easily be able to interchange devices but prevent accidental unplugging.

<table>
<thead>
<tr>
<th>Cost Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOLEX Connectors 4 pos</td>
</tr>
<tr>
<td>Molex</td>
</tr>
<tr>
<td>Female</td>
</tr>
<tr>
<td>Cables</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

Table 5 – Connector Comparison

CAT6 ethernet terminated with RJ-45 connectors were chosen because CAT6 left open the option to switch to differential signal UART(4 twisted pairs in the cable) and maintained the level of electromagnetic shielding desired. CAT6 RJ-45 cabling is also faster to assemble than Molex connectors. Referring to a cost breakdown comparison, as seen in the above table, Neutrik Ethercon connectors were eliminated as an option because the connectors were expensive and overly rugged for the scope of the project. Water resistance was considered important but it was decided that putting a plastic bag over sensitive electronics would be enough to do the job if the situation demanded protection against rain.

The wiring and connector system design also required careful consideration of microcontroller configuration and inter-device communication plans. Up until the winter check off, microcontrollers were the sole controllers of the robot and were used only to operate it at the most basic level. The final design required some way to base the robot’s operation off of
image processing data when objects are identified on the competition field. The next design iteration also required more PWM and GPIO pins than what a single microcontroller could offer. These two constraints yielded several equally valid options with varying difficulty to implement with the common conclusion that more microcontrollers were needed or a single bigger one needed to be found.

Microcontroller Configuration Options considered:

5+1 microcontroller setup: This would be 4 identical microcontrollers dedicated for each quadrant of the robot and a single central microcontroller that forwarded control messages between them and the onboard laptop. I2C or four-wire RS-485 would be used to transfer data between the embedded devices. Asymmetrically placed peripheral sensors would have been placed on a dedicated sixth microcontroller or on the central microcontroller.

Two microcontroller (Front/Back): This would be a single microcontroller on the front and one on the back, each directly communicating over UART with USB-serial conversion. The microcontrollers would control their respectively placed motors and sensors. Asymmetrically placed hardware, like the single XBee or sample collector sensors would be controlled by the closest or most convenient microcontroller.

New Microcontroller: The last possible option was to choose a whole new microcontroller with more GPIO pins and PWM pins that could handle all the robots hardware and sensors. The single microcontroller would communicate directly with the laptop over UART with USB-to-serial communication just like the front/back configuration.

Three microcontroller (Motor/Servo Control, Encoders, Sensors & Xbee): A three microcontroller configuration considered having dedicated UNO32 microcontrollers, one for controlling all H-bridges, one for handling all encoders on the robot, and a third that read analog signals, capture pause commands from the Xbee, and read data off a gyro[5].

The final rover design used two UNO32 microcontrollers which communicated directly to the onboard laptop via USB-to-serial UART. One microcontroller is located on the front, another is located on the rear. Custom breakout shields were made to fit over the UNO32 to break out the pins to female RJ-45 connectors. CAT6 cable was used with RJ-45 to easily connect peripheral hardware. While both shields have identical pin mapped connections for the drive and steering H-bridges and encoders, they have additional connections that were designed or reworked for other specific connections. Refer to the front and rear Uno configurations in the following figure 30 and figure 31.

The following flowchart describes a top level data flow between the laptop, uno and drive wheel encoders. A similar data flow is followed for steering encoders, H-Bridge daughter boards, and other hardware like rangefinder modules and limit switches around the robot.

Figure 19- Top-Level Encoder Data Flowchart

Not shown in the picture, two RJ-45 ports were repurposed on the rear shield to support driving the sample collector H-bridge and reading the state of its reed switches and limit switches.

For every connector on the shields, a custom made breakout board was also made to accept the appropriate signals:

- The optical Drive encoder breakouts for the drive motor shafts
- The capacitive Steer encoder breakouts for the steering motor shafts
- H-bridge breakouts for all the Steer and Drive motors
- The limit switches on the sample collector
- IR-rangefinders module breakouts
- The Xbee/Light breakout board

Instead of using two connectors to connect to the Xbee breakout board, as shown in the rear configuration, only a single CAT6 cable was needed because Gyros and accelerometers were not implemented in the final robot design. Also, since the Xbee breakout board only needed 3 connections for UART Tx, Rx and ground, a PWM signal wire was added to take advantage of the spare wires in the cable rather than use a dedicated cable just for driving the light.

The Xbee pause box was a separate piece of hardware designed with small form-factor and extended battery life in mind. The competition required a wireless pause function for each robot so a judge can stop all of its motion if another robot on the field is about to collide with it.
This pause functionality needed to be available for the duration the mission, which can be up to two hours for the Level 2 competition. Also, since a judge will be walking behind the robot with the device, it needed to be as self-explanatory to use and as easy to hold as possible.

On the rover itself, a 9dBi antenna is connected to an Xbee mounted on a daughter board which also has a TIP122 transistor for driving the warning light. The daughter board was designed so that the light circuit was separated several inches away from the Xbee board by copper keep-outs with only the signal cline extending from an RJ-45 connector to the base of the transistor. This was done to minimize any noise due to 12v PWM switching voltages going into the Light.

A 3D-printed container with an easy to remove top was made to hold an Xbee radio, an UNO32 microcontroller, and 4 AA batteries connected in series.

Occasional data loss was observed when testing sending a pause message to the robot while running all its other software modules. To solve this issue a reliable data transfer protocol was implemented where the pause box would retransmit a pause or unpause command until it received an acknowledge signal back from the rover. The rear microcontroller on the rover forwards pause messages to the laptop when it receives a new pause or unpause command from the Xbee mounted on the rover (which captures pause commands from the pause box).

Once the laptop receives a pause message from the rear microcontroller, the appropriate control messages are transmitted back down to the microcontrollers and the laptop state machines enter an idle state until the rear microcontroller forwards an unpause message from the pause box. See figure 36.

When pausing, the laptop transmits speed messages to both microcontrollers to set the robot target speed to 0 rpm. PID speed control then ramps down the wheel speeds and attempts to keep them still.

There were some concerns that sticking the light circuit next to the Xbee circuit would cause noise issues, but there was no noticeable increase in data loss when Pause commands were sent to the rover-mounted Xbee from the Pause Box using the daughter board.

To achieve a smooth strobing effect from the warning light, a 3rd order exponential equation was used to control the duty cycle. The reason for using a nonlinear progression for changing the duty cycle was because the human eye’s light receptors interpret light (luminous flux) with nonlinear response to actual radiant flux. There is an entire scientific field devoted to linearizing luminous flux beyond the projects scope, but an attempt to have a smooth strobe was made by approximating an exponential. After adding the warning light code to the full rover software, there were significant timing issues that were encountered. The first attempt to solve the issue was to use a large lookup table instead of calculating values on the fly. This helped, but for some reason the microcontroller would get lost in its program memory after several minutes of strobing and the entire program would crash. This was likely because the lookup table was taking up
too much space, so the lookup table was minimized to only a few values trading out the smoothness of the transitions. For some reason, the light kept crashing the program after several minutes of running until just 3 discrete PWM levels were used for turning the light on and off. Since this was a very aesthetic portion of the rover, the light software was left as is with a very “jerky” strobe rather than smooth dimming.

The IR rangefinder placement on the robot was an important design decision for the robot that will need to be experimented with in the future. The configuration used for the winter check-off placed the rangefinders at vertical positions higher than the tallest sample in order to prevent identifying samples as obstacles while still being able to detect impassable objects that could not be passed over. Since the chassis bed sits above the tallest sample, the only area where impassable obstacles may not be detectable by the IR rangefinders would be around the wheels. Multi-obstacle dynamic detection was implemented with IR rangefinders. This was implemented by completing a fast sampling software module that could sample at minimal angle increments rather than large discrete angles. The software module allows for multiple obstacles to be identified and stored in a circular buffer of structures with fields that contain the size, range and rangefinder that did the detection.

![Figure 22 - Rangefinder Module Placement](image)

The design of the robot was to attempt a more continuous operation over the previous year’s discrete detection technique, which would necessitate a faster sampling technique while still maintaining acceptable detection capabilities of obstacles.

![Figure 23 – IR Rangefinder Detection Region](image)

Therefore, based upon the speed of the robot, the minimum frequency at which the IR rangefinder had to sample any given angle to maintain a safety range of about 30 cm in front could be determined. To maximize the range of coverage using two rangefinders in the front would be to scan a little over 90 degrees for the front right and front left quadrants of the robot. Additionally, instead of sampling at large discrete angle increments for long periods of time, sampling at the maximum rate and incrementing over minimal angles would increase detection resolution. This method would allow the robot to identify obstacles and determine their position in front of the robot. If two rangefinders are insufficient for maintaining a 30 cm safety range, a third rangefinder could be added to reduce the range of angles each has to scan, thus increasing the max rate any single angle can be sampled.

Figure 38 shows the IR rangefinders mounted inverted at the height of the chassis. The right figure shows the detection region that they cover.

The previous steering system used servos which gave no feedback to the micros. That system had issues with the steering systems not aiming in the same direction which would cause severe damage to them. We decided to prevent that from ever happening again by using precise position feedback.

Our requirements for the steer encoders were that they would need to be cheap and fit a 6mm shaft such as the rear shaft of the Bodine. We wanted a pre-assembled kit that wouldn't require any soldering, due to time constraints, and we wanted precision of around one degree.

We discovered the AMT-103 capacitive quadrature encoders capable of 48 to 2048 pulses per revolution (PPR) for only $25. The quadrature encoding produces two square wave outputs denoted A and B for a total of four possible states. By decoding the quadrature signals, one revolution of the encoder can be broken down into a number of ticks equal to four states times the PPR. With the 240:1 ratio of the Bodine output shaft to its rear shaft, the number of encoder ticks per revolution of the output shaft would be multiplied by 240, as well. With the lowest PPR setting of 48, this resulted in:

$$\frac{1 \text{ output rev}}{360 \text{ degrees}} \times \frac{240 \text{ rear revs}}{\text{output rev}} \times \frac{48 \text{ pulses}}{\text{rear rev}} \times \frac{4 \text{ quadrature ticks}}{\text{pulse}} = 128 \text{ ticks / degree}$$

This 0.0078 degree precision would be far more than we needed, and for the price it seemed like a bargain. We would later regret going with such high precision encoders.
When more code was added later, the steering system began to work perfectly, but then other system functions began to break, since they were being interrupted by the steering system. We had to cut back the amount of interrupts to the bare minimum to make steering work, which was easy, since the Bodines always move at a constant, slow speed. This left more CPU cycles for other functions, and the rover worked a lot better.

We needed to use the high-precision data from the steer encoder module to control the angle of the steering motors. Ideally, it would do so quickly and precisely. Precision would not be difficult, due to the quality of our feedback. However, speed would be more difficult. Being able to choose the desired angle required a sort of absolute encoding. Since our encoders were incremental, this required us to calibrate somehow at startup, and perform all steering incrementally, relative to the starting angle.

The Bodine motors move very slowly, taking five seconds to turn 90 degrees at fully duty. Because of this, we decided that we should always run them at full duty. The controller would thus be a "bang-bang" controller- it would always drive the motor at maximum power toward the setpoint. As the rover bumped along on the grass, the motors would constantly adjust to correct any deviations from the steering angle setpoint.

First, bump switches were attached to the fenders such that they would click when the wheel turned far enough in one direction. By measuring the number of ticks from the clicking point to center and hardcoding that value into the steering module, we could achieve "absolute" encoding by performing all steering relative to that known starting position. When the robot first starts up, it simply turns all of its wheels to the left until they click. When they click, they then turn the opposite direction- towards the center. They count out the hardcoded number of ticks and end up pointing directly ahead. They consider this position "zero" and can then count out ticks in either direction to turn to any angle.

Bang-bang control was accomplished by counting ticks and applying the function found above: 128 ticks / degree. When the angle is not equal to the set point, the steering motors turn toward the set point at max duty.

The steering calibration routine worked very well. It only had issues if the steering encoders skipped, or if the bump switches were bent during transportation. In the event that the switches were bent, the hardcoded values would need to be adjusted slightly to compensate.

The only issue we had with steering itself was that the steering would oscillate constantly about the set point. Since it was very precise, and always drove at max power, it would never quite reach the setpoint, but instead constantly overshoot it by less than a degree and steer back again. This wasn't a huge problem, but used up more battery power than necessary and

The Uno32 microprocessor is quite fast, but polling two encoders per micro at such a high sample rate took up more of our processor power than we would have liked. This turned into a major problem during integration.

Nonetheless, we had to ensure that the steer encoders were sampled fast enough that they would never miss a tick. If they did miss a tick, they wouldn't realize it, and the steering would be miscalibrated for the remainder of the run. We also had to ensure that they weren't sampled any more than necessary, since that would place unnecessary strain on the micros which were already struggling to keep up.

To ensure that the encoders were serviced at exactly the right rate, we sampled them on timer interrupts. We roughly estimated the sample rate on paper, then experimentally determined the actual minimum sample rate for the encoders and set the interrupt frequency just above it. We then doubled the frequency and alternated between sampling the left and right encoder, which reduced the length of the interrupt.

When we first implemented the steer encoder module we polled it in the main loop of the program. At that point there wasn't much else going on in the main loop, so the sample rate was high enough that the system worked perfectly. However, when more code was added later, the steering system began to

The function above could be used to map the incoming encoder ticks to a measured steering angle. However, the incredibly high resolution turned out to be a double-edged sword. While it offered us a high degree of steering precision, it also required an exceedingly high sample rate. Since the motor executed a 90 degree turn in about 5 seconds, the encoders outputted:

\[
128 \text{ ticks} / \text{degree} \times 90 \text{ degrees} / 5 \text{ seconds} = 2304 \text{ ticks per second.}
\]

The Uno32 microprocessor is quite fast, but polling two encoders per micro at such a high sample rate took up more of our processor power than we would have liked. This turned into a major problem during integration.

Nonetheless, we had to ensure that the steer encoders were sampled fast enough that they would never miss a tick. If they did miss a tick, they wouldn't realize it, and the steering would be miscalibrated for the remainder of the run. We also had to ensure that they weren't sampled any more than necessary, since that would place unnecessary strain on the micros which were already struggling to keep up.

To ensure that the encoders were serviced at exactly the right rate, we sampled them on timer interrupts. We roughly estimated the sample rate on paper, then experimentally determined the actual minimum sample rate for the encoders and set the interrupt frequency just above it. We then doubled the frequency and alternated between sampling the left and right encoder, which reduced the length of the interrupt.

When we first implemented the steer encoder module we polled it in the main loop of the program. At that point there wasn't much else going on in the main loop, so the sample rate was high enough that the system worked perfectly. However, when more code was added later, the steering system began to

The function above could be used to map the incoming encoder ticks to a measured steering angle. However, the incredibly high resolution turned out to be a double-edged sword. While it offered us a high degree of steering precision, it also required an exceedingly high sample rate. Since the motor executed a 90 degree turn in about 5 seconds, the encoders outputted:

\[
128 \text{ ticks} / \text{degree} \times 90 \text{ degrees} / 5 \text{ seconds} = 2304 \text{ ticks per second.}
\]

The Uno32 microprocessor is quite fast, but polling two encoders per micro at such a high sample rate took up more of our processor power than we would have liked. This turned into a major problem during integration.

Nonetheless, we had to ensure that the steer encoders were sampled fast enough that they would never miss a tick. If they did miss a tick, they wouldn't realize it, and the steering would be miscalibrated for the remainder of the run. We also had to ensure that they weren't sampled any more than necessary, since that would place unnecessary strain on the micros which were already struggling to keep up.

To ensure that the encoders were serviced at exactly the right rate, we sampled them on timer interrupts. We roughly estimated the sample rate on paper, then experimentally determined the actual minimum sample rate for the encoders and set the interrupt frequency just above it. We then doubled the frequency and alternated between sampling the left and right encoder, which reduced the length of the interrupt.

When we first implemented the steer encoder module we polled it in the main loop of the program. At that point there wasn't much else going on in the main loop, so the sample rate was high enough that the system worked perfectly. However, when more code was added later, the steering system began to

The function above could be used to map the incoming encoder ticks to a measured steering angle. However, the incredibly high resolution turned out to be a double-edged sword. While it offered us a high degree of steering precision, it also required an exceedingly high sample rate. Since the motor executed a 90 degree turn in about 5 seconds, the encoders outputted:

\[
128 \text{ ticks} / \text{degree} \times 90 \text{ degrees} / 5 \text{ seconds} = 2304 \text{ ticks per second.}
\]

The Uno32 microprocessor is quite fast, but polling two encoders per micro at such a high sample rate took up more of our processor power than we would have liked. This turned into a major problem during integration.

Nonetheless, we had to ensure that the steer encoders were sampled fast enough that they would never miss a tick. If they did miss a tick, they wouldn't realize it, and the steering would be miscalibrated for the remainder of the run. We also had to ensure that they weren't sampled any more than necessary, since that would place unnecessary strain on the micros which were already struggling to keep up.

To ensure that the encoders were serviced at exactly the right rate, we sampled them on timer interrupts. We roughly estimated the sample rate on paper, then experimentally determined the actual minimum sample rate for the encoders and set the interrupt frequency just above it. We then doubled the frequency and alternated between sampling the left and right encoder, which reduced the length of the interrupt.

When we first implemented the steer encoder module we polled it in the main loop of the program. At that point there wasn't much else going on in the main loop, so the sample rate was high enough that the system worked perfectly. However, when more code was added later, the steering system began to

The function above could be used to map the incoming encoder ticks to a measured steering angle. However, the incredibly high resolution turned out to be a double-edged sword. While it offered us a high degree of steering precision, it also required an exceedingly high sample rate. Since the motor executed a 90 degree turn in about 5 seconds, the encoders outputted:

\[
128 \text{ ticks} / \text{degree} \times 90 \text{ degrees} / 5 \text{ seconds} = 2304 \text{ ticks per second.}
\]

The Uno32 microprocessor is quite fast, but polling two encoders per micro at such a high sample rate took up more of our processor power than we would have liked. This turned into a major problem during integration.

Nonetheless, we had to ensure that the steer encoders were sampled fast enough that they would never miss a tick. If they did miss a tick, they wouldn't realize it, and the steering would be miscalibrated for the remainder of the run. We also had to ensure that they weren't sampled any more than necessary, since that would place unnecessary strain on the micros which were already struggling to keep up.

To ensure that the encoders were serviced at exactly the right rate, we sampled them on timer interrupts. We roughly estimated the sample rate on paper, then experimentally determined the actual minimum sample rate for the encoders and set the interrupt frequency just above it. We then doubled the frequency and alternated between sampling the left and right encoder, which reduced the length of the interrupt.

When we first implemented the steer encoder module we polled it in the main loop of the program. At that point there wasn't much else going on in the main loop, so the sample rate was high enough that the system worked perfectly. However, when more code was added later, the steering system began to
put more wear on the encoders, so we added in some hysteresis to prevent it. Instead, the control would only kick in if the steering deviated from the setpoint by more than a set value. We chose to use one degree as our hysteresis value, allowing us to change the datatype from floating point to integer—far smaller and more convenient datatype. Each steering quadrant might be off by up to a degree in either direction, but we didn’t notice any major problems from that.

The Parallax 36-tooth optical encoders on the drive wheels were used by last year’s team to measure the distance driven by the rover. They used only one of the four encoders to measure the distance driven. We decided to use at least two encoders—left and right—to measure distance. This would allow us to measure if the robot was beginning to swerve off-course. Later, we decided to implement PID speed control on the drive wheels, as well.

The drive encoders were similar to the steer encoders in that they used quadrature encoding and had to be sampled rapidly and regularly. Fortunately, they had fewer teeth and were placed on the output shafts of the drive wheels instead of directly on the motor shafts, resulting in only:

\[
\text{36 pulses / output rev * 4 quadrature ticks / pulse * 150 RPM output revs / minute unloaded = 360 ticks per second}
\]

They didn’t need to be sampled at the same rate as the steer encoders, so we created a counter variable which incremented on each call of the ISR, and used bit masking to poll the drive encoders only when certain bits of it were high. This method was faster than other methods such as the modulo function which could also be used for the same purpose.

We calculated the speed of the drive wheels on a much slower timer, since it was far more computationally complex. The speed was calculated by taking a time stamp whenever that timer triggered and comparing it to the previous time stamp, in case the timing was slightly off. We divided the number of ticks/centimeters driven by that change in time, resulting in an angular velocity:

\[
1 \text{ rev / 36 pulses * 1 tooth / 4 tick * 1 / time difference = RPM}
\]

The drive encoders worked well from the beginning and never posed any serious problems, aside from the issues associated with calling the ISR too frequently. The only tricky part was figuring out what frequency to poll the drive encoders at, since the speed of the drive wheels varies so much. It was much easier with the Bodines, since they always move at a fairly constant speed. Once we had the right frequency set, the drive encoders didn’t pose any major problems.

In addition to providing distance measurements, the drive encoders could also be used to calculate drive speed. We decided that speed control would be useful for a number of reasons:

- It keeps the rover’s speed constant in spite of dropping battery voltage
- It allows the rover to stop on a slope or drive on a slope at a constant speed.
- It helps the rover to obey the maximum speed limit imposed by the competition rules.
- It allows the robot to ramp up its speed smoothly so that it doesn’t jerk when accelerating or decelerating. PWM could be ramped directly, but without the speed control, this will always result some acceleration or deceleration, and since jerk is the derivative of acceleration, a steadily ramping PWM, therefore, will always results in jerk.

With four separate and possibly conflicting streams of encoder data coming in, we had a lot of information to sort through. Some of the wheels could be slipping, causing them to spin at a higher rate. This would increase our estimate of how far the rover has traveled.

An estimator might be a good way to control the rover despite possibly faulty data coming in due to slippage. However, that would require developing a model of the dynamics of our rover, which seemed to be outside the scope of the project. Because of time constraints, we decided to use PID speed control. PID is simple, quick, and effective.

The only work to do, aside from writing the PID loop, was tuning the PID constants to reduce rise time, settling time, and
steady state error. We first tuned the constants while the robot was on blocks (wheels lifted off the ground). This was not a very useful tuning, since the values would need to change drastically depending on wheel friction. However, even with almost no resistance we found that no derivative term was needed. This prompted us to remove the derivative term to reduce computational complexity (calculating derivative requires division). Next, we moved the robot to the concrete outside our lab and tuned there. This was closer to our final values, since the wheels now had to struggle more to maintain their speed. Finally, we tuned PID on grass to get a PID controller which would most likely work in Boston. A final tuning in Boston was the final preparation before the competition. The final controller was a PI controller with no derivative term and far more I (integral) than P (proportional). This tuning resulted in the smoothest acceleration and smallest oscillations. Rise time was not as much of a concern, since the rover was intended to move slowly and carefully.

In order to prevent wheel slippage from affecting our data, we decided to only drive with two wheels at any time. By using the data from each driving wheel to control its individual speed and the data from each non-driving wheel to measure distance, we could reduce slippage of the drive wheels and ensure that it would not affect the distance measurements.

As an example, when the robot is driving straight it uses its two front wheels to drive forward and watches the rear wheels to figure out how far it has gone. It controls its speed constantly using PID on the front, and sets that speed to zero once the average distance measured by the rear wheels is greater than or equal to the distance driven.

PID speed control was tremendously successful. It allowed us to start and stop the robot smoothly, drive at a constant speed, and drive on a slope at the same speed as flat ground, depending on the steepness of the slope. A ramp function was not even needed to start and stop the robot smoothly. The rise time of the PID was slow and jerk was minimal. If the rover tried to climb a slope that was too steep, then the motors would be saturated and struggle to climb it. However, we determined that the hills on the competition field are very gentle based on the topographic map provided.

One of the most important features of PID was that it kept the rover moving at a constant speed despite loss of battery voltage. It would steadily integrate up to whatever effort was necessary to achieve the desired speed. If the rover could not achieve the desired speed, then we would know that it could only be due to insufficient motor torque at the current battery level and terrain conditions.

C. Steer Encoder Module

The previous steering system used servos which gave no feedback to the micros. That system had issues with the steering systems not aiming in the same direction which would cause severe damage to them. We decided to prevent that from ever happening again by using precise position feedback.

Our requirements for the steer encoders were that they would need to be cheap and fit a 6mm shaft such as the rear shaft of the Bodine. We wanted a pre-assembled kit that wouldn't require any soldering, due to time constraints, and we wanted precision of around one degree.

We discovered the AMT-103 capacitive quadrature encoders capable of 48 to 2048 pulses per revolution (PPR) for only $25. The quadrature encoding produces two square wave outputs denoted A and B for a total of four possible states. By decoding the quadrature signals, one revolution of the encoder can be broken down into a number of ticks equal to four states times the PPR. With the 240:1 ratio of the Bodine output shaft to its rear shaft, the number of encoder ticks per revolution of the output shaft would be multiplied by 240, as well. With the lowest PPR setting of 48, this resulted in:

\[
\text{1 output rev} / \text{360 degrees} \times \text{240 rear revs} / \text{output rev} \times \text{48 pulses} / \text{rear rev} \times \text{4 quadrature ticks} / \text{pulse} = \text{128 ticks / degree}
\]

This 0.0078 deg precision would be far more than we needed, and for the price it seemed like a bargain. We would later regret going with such high precision encoders.

![AMT-103 Encoder](image_url)

The function above could be used to map the incoming encoder ticks to a measured steering angle. However, the incredibly high resolution turned out to be a double-edged sword. While it offered us a high degree of steering precision, it also required an exceedingly high sample rate. Since the motor executed a 90 degree turn in about 5 seconds, the encoders outputted:

\[
\text{128 ticks / degree} \times \text{90 degrees} / \text{5 seconds} = 2304 \text{ ticks per second}
\]

The Uno32 microprocessor is quite fast, but polling two encoders per micro at such a high sample rate took up more of
our processor power than we would have liked. This turned into a major problem during integration.

Nonetheless, we had to ensure that the steer encoders were sampled fast enough that they would never miss a tick. If they did miss a tick, they wouldn't realize it, and the steering would be miscalibrated for the remainder of the run. We also had to ensure that they weren't sampled any more than necessary, since that would place unnecessary strain on the micros which were already struggling to keep up.

To ensure that the encoders were serviced at exactly the right rate, we sampled them on timer interrupts. We roughly estimated the sample rate on paper, then experimentally determined the actual minimum sample rate for the encoders and set the interrupt frequency just above it. We then doubled the frequency and alternated between sampling the left and right encoder, which reduced the length of the interrupt.

When we first implemented the steer encoder module we polled it in the main loop of the program. At that point there wasn't much else going on in the main loop, so the sample rate was high enough that the system worked perfectly. However, when more code was added later, the steering system began to break. The encoders began skipping and losing ticks, causing the steering motors to travel too far.

At that point, we started putting all of the robot functions on timers to save CPU cycles. In particular, we placed the steer encoders and drive encoders on timer interrupts so that they would be serviced regularly and interrupt other rover functions, if necessary. When we first started using interrupts, then steering system worked perfectly, but then other system functions began to break, since they were being interrupted by the steering system. We had to cut back the amount of interrupts to the bare minimum to make steering work, which was easy, since the Bodines always move at a constant, slow speed. This left more CPU cycles for other functions, and the rover worked a lot better.

D. Steering Control

We needed to use the high-precision data from the steer encoders to control the angle of the steering motors. Ideally, it would do so quickly and precisely. Precision would not be difficult, due to the quality of our feedback. However, speed would be more difficult. Being able to choose the desired angle required a sort of absolute encoding. Since our encoders were incremental, this required us to calibrate somehow at startup, and perform all steering incrementally, relative to the starting angle.

The Bodine motors move very slowly, taking five seconds to turn 90 degrees at fully duty. Because of this, we decided that we should always run them at full duty. The controller would thus be a "bang-bang" controller- it would always drive the motor at maximum power toward the setpoint. As the rover bumped along on the grass, the motors would constantly adjust to correct any deviations from the steering angle setpoint.

First, bump switches were attached to the fenders such that they would click when the wheel turned far enough in one direction. By measuring the number of ticks from the clicking point to center and hardcoding that value into the steering module, we could achieve "absolute" encoding by performing all steering relative to that known starting position. When the robot first starts up, it simply turns all of its wheels to the left until they click. When they click, then turn the opposite direction- towards the center. They count out the hardcoded number of ticks and end up pointing directly ahead. They consider this position "zero" and can then count out ticks in either direction to turn to any angle.

Bang-bang control was accomplished by counting ticks and applying the function found above: 128 ticks / degree. When the angle is not equal to the setpoint, the steering motors turn toward the setpoint at max duty.

The steering calibration routine worked very well. It only had issues if the steering encoders skipped, or if the bump switches were bent during transportation. In the event that the switches were bent, the hardcoded values would need to be adjusted slightly to compensate.

The only issue we had with steering itself was that the steering would oscillate constantly about the set point. Since it was very precise, and always drove at max power, it would never quite reach the setpoint, but instead constantly overshoot it by less than a degree and steer back again. This wasn't a huge problem, but used up more battery power than necessary and put more wear on the encoders, so we added in some hysteresis to prevent it. Instead, the control would only kick in if the steering deviated from the setpoint by more than a set value. We chose to use one degree as our hysteresis value, allowing us to change the datatype from floating point to integer- a far smaller and more convenient datatype. Each steering quadrant might be off by up to a degree in either direction, but we didn't notice any major problems from that.

E. Drive Encoder Module

The Parallax 36-tooth optical encoders on the drive wheels were used by last year's team to measure the distance driven by the rover. They used only one of the four encoders to measure the distance driven. We decided to use at least two encoders- left and right- to measure distance. This would allow us to measure if the robot was beginning to swerve off-course. Later, we decided to implement PID speed control on the drive wheels, as well.

The drive encoders were similar to the steer encoders in that they used quadrature encoding and had to be sampled rapidly and regularly. Fortunately, they had fewer teeth and were placed on the output shafts of the drive wheels instead of directly on the motor shafts, resulting in only:

\[
36 \text{ pulses / output rev} \times 4 \text{ quadrature ticks / pulse} \times 150 \text{ RPM output revs / minute unloaded} = 360 \text{ ticks per second}
\]
In order to calculate the distance driven per encoder tick, we measured the actual circumference of the wheels. It turned out that it was around 625 cm.

\[
1 \text{ revolution} / 36 \text{ teeth} * 625 \text{ cm} / \text{output revolution} * 1 \text{ tooth} / 4 \text{ quadrature ticks} = 4.34 \text{ cm/tick}
\]

Since the purpose of these encoders was to control the speed of the wheels and distance driven rather than to control the precise angle, this lower resolution served fine. They had to be sampled fast enough not to lose ticks, but the speed only needed to be calculated as often as the speed control was calculated. Since the rover would be moving slowly (< 2 m/s), this could be fairly slow.

We started out polling the drive encoders in the main loop alongside the steer encoders at exactly the same rate. Later, when we started using interrupts, we placed the drive encoders in the same interrupt, alternating just like the steer encoders. They didn't need to be sampled at the same rate as the steer encoders, so we created a counter variable which incremented on each call of the ISR, and used bit masking to poll the drive encoders only when certain bits of it were high. This method was faster than other methods such as the modulo function which could also be used for the same purpose.

We calculated the speed of the drive wheels on a much slower timer, since it was far more computationally complex. The speed was calculated by taking a time stamp whenever that timer triggered and comparing it to the previous time stamp, in case the timing was slightly off. We divided the number of ticks/centimeters driven by that change in time, resulting in an angular velocity:

\[
1 \text{ revolution} / 36 \text{ pulses} * 1 \text{ tooth} / 4 \text{ tick} * 1 / \text{time difference} = \text{RPM}
\]

The drive encoders worked well from the beginning and never posed any serious problems, aside from the issues associated with calling the ISR too frequently. The only tricky part was figuring out what frequency to poll the drive encoders at, since the speed of the drive wheels varies so much. It was much easier with the Bodines, since they always move at a fairly constant speed. Once we had the right frequency set, the drive encoders didn't pose many problems.

F. Speed Control

In addition to providing distance measurements, the drive encoders could also be used to calculate drive speed. We decided that speed control would be useful for a number of reasons:

- It keeps the rover's speed constant in spite of dropping battery voltage.
- It allows the rover to stop on a slope or drive on a slope at a constant speed.
- It helps the rover to obey the maximum speed limit imposed by the competition rules.
- It allows the robot to ramp up its speed smoothly so that it doesn't jerk when accelerating or decelerating. PWM could be ramped directly, but without the speed control, this will always result some acceleration or deceleration, and since jerk is the derivative of acceleration, a steadily ramping PWM, therefore, will always results in jerk.

With four separate and possibly conflicting streams of encoder data coming in, we had a lot of information to sort through. Some of the wheels could be slipping, causing them to spin at a higher rate. This would increase our estimate of how far the rover has traveled.

An estimator might be a good way to control the rover despite possibly faulty data coming in due to slippage. However, that would require developing a model of the dynamics of our rover, which seemed to be outside the scope of the project. Because of time constraints, we decided to use PID speed control. PID is simple, quick, and effective.

The only work to do, aside from writing the PID loop, was tuning the PID constants to reduce rise time, settling time, and steady state error. We first tuned the constants while the robot was on blocks (wheels lifted off the ground). This was not a very useful tuning, since the values would need to change drastically depending on wheel friction. However, even with almost no resistance we found that no derivative term was needed. This prompted us to remove the derivative term to reduce computational complexity (calculating derivative requires division). Next, we moved the robot to the concrete outside our lab and tuned there. This was closer to our final values, since the wheels now had to struggle more to maintain their speed. Finally, we tuned PID on grass to get a PID controller which would most likely work in Boston. A final tuning in Boston was the final preparation before the competition. The final controller was a PI controller with no derivative term and far more I (integral) than P (proportional). This tuning resulted in the smoothest acceleration and smallest oscillations. Rise time was not as much of a concern, since the rover was intended to move slowly and carefully.

In order to prevent wheel slippage from affecting our data, we decided to only drive with two wheels at any time. By using the data from each driving wheel to control its individual speed and the data from each non-driving wheel to measure distance, we could reduce slippage of the drive wheels and ensure that it would not affect the distance measurements.
As an example, when the robot is driving straight it uses its two front wheels to drive forward and watches the rear wheels to figure out how far it has gone. It controls its speed constantly using PID on the front, and sets that speed to zero once the average distance measured by the rear wheels is greater than or equal to the distance driven.

PID speed control was tremendously successful. It allowed us to start and stop the robot smoothly, drive at a constant speed, and drive on a slope at the same speed as flat ground, depending on the steepness of the slope. A ramp function was not even needed to start and stop the robot smoothly. The rise time of the PID was slow and jerk was minimal. If the rover tried to climb a slope that was too steep, then the motors would be saturated and struggle to climb it. However, we determined that the hills on the competition field are very gentle based on the topographic map provided.

One of the most important features of PID was that it kept the rover moving at a constant speed despite loss of battery voltage. It would steadily integrate up to whatever effort was necessary to achieve the desired speed. If the rover could not achieve the desired speed, then we would know that it could only be due to insufficient motor torque at the current battery level and terrain conditions.

IV. SOFTWARE

A. Software Version Control

Software version control systems were a new thing to the majority of this team. Our first attempt at version control was through a combination of GitHub and Pogoplug. The idea was for the master branch of GitHub to be cloned onto an external hard drive connected to the PogoPlug, and everyone would modify code through the PogoPlug web utility. At the end of this, Ajay would be responsible for making commits and pushes from the PogoPlug device to the server hosted at GitHub.com. Since this proved to defeat the purpose of Git, as well as just add more confusion regarding the role of the Pogoplug device, another approach was conceptualized.

For our second approach, we decided to have the Pogoplug serve as our hosting server. This approach required us to install ArchLinux onto the Pogoplug device, making it inoperable with the Pogoplug web utility. The advantage here was that we would be able to maintain our own servers in the event that those hosted by GitHub.com were to go down. Since we also wanted to use the Pogoplug device to host a SQL database for our data mining app, this was another advantage of this approach. We began implementing this approach by purchasing a second Pogoplug device, and installing ArchLinux on the first one. After a few days of trying to get the ArchLinux installation done right, we decided that GitHub's servers would be reliable enough, and traditional usage of Git was chosen.

In the beginning stages of the project, attempts were made to ramp up on Git usage, but most of them were in vain early on. Once software became the main focus of the project, more effort was made to properly use Git. This was exemplified with a workshop on basic Git usage that Ajay held. After getting the whole team on the same page with basic Git usage, some scripts were written to simplify Git usage on both Windows and OpenSUSE. Most of these scripts were just wrappers around committing, pulling, and pushing, but a few others served higher-level purposes. One notable example include: git-shove, which performs a commit, pulls, commits changes before a local merge if needed, and pushes. Another example is git-resolveConflicts, which will walk through all conflicted files after a local merge attempt, and allow the user to resolve them using the diff tool of their choosing. These scripts will certainly be valuable assets to anyone using Git in the future and having spent a day to write them certainly increased the team's overall productivity.

B. Autonomy Scripts

By the rules of the competition, our rover must be fully autonomous. This means that as soon as the master power switch is turned on, the rover needs to do its full operation. Since we have an onboard laptop, this means that the laptop needs to automatically configure everything for the rover to run as well as run the main program once it finishes booting. Furthermore, if the emergency stop is engaged, the laptop will shutdown immediately. From our experience as Windows users, this effectively rules out Windows as a reasonable option as modifying the OS boot sequence in any way is much more difficult. We began exploring into Linux distributions and decided on OpenSUSE because it provides the ability to plug in scripts when power is connected or disconnected through GUI.

C. Shutdown

When exploring our options regarding total autonomy of the rover, we began looking into how the laptop would shut down when the Emergency Stop button was engaged. We initially thought to keep the laptop's battery inside it so that a proper shutdown sequence could happen when the emergency stop was engaged. We went so far as to writing a script that would put the laptop to sleep when the power was disconnected from it. By the competition rules, the emergency stop must cut power to all components of the rover and shut them down instantly. At first we explored into making the proper shutdown occur quicker, but ultimately relied on the power getting cut not doing any damage to the laptop.

D. Startup:

After abandoning the unnecessary shutdown script, we began exploring into the startup scripts. The startup scripts needed to run any configuration related to the connected devices and then start our program for the competition. To begin this, we needed to look into how OpenSUSE's boot process worked, so we could modify it without breaking anything we didn't understand. Doing some research led us to a custom script framework from the SuSE community. This framework basically plugs in after network drivers are initialized that allows users to plug in their own scripts into a provided directory. Not having to worry about writing kernel level scripts, we then moved onto the actual startup behavior.
Our first objective for the startup script was to ensure we could run an executable in user space. We made the assumption that the executable would exist prior to being run. This saves the startup script the trouble of also running a compiler and possibly not starting the rover if someone had recently added unfinished code to the repository. We started with a basic “Hello, World!” program that writes a word to a file and saves that onto the desktop. This was the best means of testing this as we don't have any terminal to send output to when the laptop is still booting. Once this was working, we began implementing our final startup script based on the physical procedures we took during rover testing.

1) OpenSuSE Bugs: The first OS bug we came across occurred once we finished our message protocol between the laptop and microcontrollers. This was the first time that we ever tried communicating with both microcontrollers in the same user program. On our initial attempt, we found that we were able to communicate fine with the front microcontroller, but the rear wouldn't respond. Looking deeper into the issue, we verified that the messages we were sending to the back were successfully written and as far as the user program was concerned, everything was behaving as expected. What we didn't see was any communication from the back actually being received by the laptop. After a day of exploring the issue, we found a workaround by running the screen command on the rear prior to running our top level program. What was unexpected was that once we killed the screen and ran the program again it still worked as expected.

Our second OS level bug came when we added further robustness to the message protocol. After doing so, the only way we could actually communicate with the microcontrollers was to run our top level program once, kill it, manually reset the microcontrollers, and run our user program again. When we encountered this issue, we couldn't afford to waste much time trying to figure out why it was occurring and settled on a workaround.

2) Final approach: To ensure that our rover would start once our laptop booted, we decided that our startup script needed to simulate the workarounds we had to get it to run, versus trying to solve those problems and just have the startup script run the top level program. Doing this correctly actually required the use of four separate scripts. This is because once a script begins running a program, it has no ability to kill that program and continue execution. One possible workaround to that would've been to have one script spawn a child process that would kill a program started from a parent process. With our custom script framework, we had a much easier way to have this behavior occur.

In essence, the four startup scripts all begin at specific times during the boot process. The first one runs in the foreground when our custom script framework begins executing and opens a terminal (the screen command) on the rear microcontroller. Our second and third scripts run in the background and are started after the first one. These background scripts can start in any order, but they cover killing either an instance of a terminal or the first run of our top level program. Essentially, the second script will kill the terminal on the rear and open one on the front while the third script will kill the terminal on the front and run the first instance of our user program. Our last script runs after the rest of OpenSuSE's boot process and kills the first instance of our program, and running the second. We initially tried adding in a microcontroller reset through a C program, but after further testing, realized this was unnecessary and that the steps described above are enough to startup our rover.

E. Top Level Rover Control

At the very basic description of Level 1 of the competition, the rover just needs to collect a sample at a known location and bring it back to where it started. As simple as this task is for a human to interpret and perform, having a machine do it is a significantly more complicated task. Breaking it down into a high level gives us this state machine.

![Figure 26 - Top level State Machine](image)

1) Initialization: First and foremost, the rover needs to calibrate its wheels to ensure it'll drive straight when we want it to. This behavior as well as the initialization of image processing and navigation parameters is handled in the initialization state.

2) Leaving the Starting Platform: Once the rover has calibrated, it then needs to leave the starting platform. For the competition, the judges control the orientation of the starting platform, while we control the orientation of the rover on the starting platform initially. The starting platform is set to face the amphitheater in the field with a spot to bolt down the team's home beacon. This makes getting off the starting platform its own challenge as we can't just point the rover straight at the pre-cached sample and tell it to drive. Since our rover can handle driving and steering all four wheels, the starting platform challenge only added minor complexity to our overall state machine. We decided to place our rover with the front pointing at the home beacon, treating it like a wall. Once we power up, the rover will strafe to the right off the starting platform, still pointing in the general direction of the pre-cached sample. From here, a quick tank turn will compensate for the remaining angle we're off from the pre-
cached sample. At this point, we just need to drive straight to the sample for the most part.

3) Get to Sample Area: Once we've left the starting platform, the rover will follow the shortest path from its current location to the pre-cached sample according to D* Lite. It does this in meter increments, determining whether it needs to make a turn to stay on the path. If a turn is necessary, the rover will stop driving, turn its wheels to match the tangent of a circle drawn around the center of the rover, and turns in place in increments of 45 or 90 degrees to stay on its designated path. If it determines that it should continue going straight, it won't bother to stop and continue moving straight. The entire time the rover is following a path designated by D* Lite, it's interpreting incoming data from the IR rangefinders to see if there are unexpected obstacles in the way. If we do encounter an obstacle, we note this on our D* grid and replan the shortest path to the pre-cached sample. Another process occurring in this state is image processing. At any point while we're following a path dictated by D* Lite, if we detect the pre-cached sample, we try to align our collection mechanism on it. However, if we reach the goal that D* Lite has provided us and don't see the pre-cached sample, we know that we've gotten to the general area, and just need to look around that area for the sample.

4) Search Sample Area: While searching the general area for the pre-cached sample, if image processing detects it at any point during our searching routine, then we'll begin aligning our sample collector over it. This searching procedure is also used to find the starting platform on our return trip. The entire process is depicted in the state machine in Figure 40 below.

To search the general area for the sample, we begin by doing a 360 turn in place to see if we just happened to be looking in the wrong direction once we got to the general area. If we haven't found the sample yet, we begin driving in a clockwise grid-based spiral. We continue this pattern until we either find the sample, or have gone far enough from our D* Lite goal node that we can venture to guess that our navigation was incorrect. At this point, we would backtrack to the D* Lite goal node and try again until we find it.

5) Collector Alignment:

a) Sample in Detectable Range: Once we've detected the sample, we now need to align our collection mechanism to pick it up. This works based on feedback from image processing. In essence, based on the frame that image processing last analyzed, it will tell the rover a direction to move to align itself with the center of the top camera. The rover will then drive in the appropriate direction. We keep taking directions from image processing until we've centered the collector with respect to the top camera. Following this, we begin driving straight until the sample leaves the top camera and comes to the bottom camera. When the sample is in view of the bottom camera, we center the rover again based on the same feedback from image processing, driving straight until the sample leaves the bottom camera and enters view of the sample camera.

b) Sample Below Collector: Now in the view of the sample camera, we now move the rover based on 8 cardinal
directions versus just left and right. If at any point, we lose track of the sample, the rover will backup until it's in view again. In the case that the rover can't find the sample again after backing up out of view of the top camera, it'll enter the searching state described above. Once the rover has aligned itself perfectly over the sample, we then attempt to pick it up. When the sample is a detectable distance away and then got lost, we exit the regular alignment state machine and backup until we can see the sample again. Once we relocate the sample again after backing up outside the detectable range from where we began to try and collect, we'll resume our searching sample area state machine as described previously.

6) Collection: (refer to figure 46)

To actually collect a sample, the rover must first lower the sample collector to the ground. A limit switch will trigger when the sample has hit the ground, telling us to begin the collection mechanism. Next, we spin the fins of our sample collector until a reed switch is triggered. When this happens, we've collected the sample and begin raising the sample collector. Another limit switch on the chassis will stop the sample collector from raising once triggered. Again, we still have image processing running to determine if we've lost track of the sample throughout this process. If we lost track of the sample at any point, we raise the sample collector, realign, and try to collect again.

7) Returning Home: Throughout the alignment and collection procedures, we still update our D* Lite map with our movements, so we know our actual position on the field at all times. This is crucial to our returning home procedure. To begin returning home, we start by changing the goal of D* Lite to our starting position, and following the path generated back home. The details of this return trip to the general area of our home base are identical to those of getting to the general area of the sample.

8) Parking: When returning to the home beacon, we instead are looking for the reverse side of our home beacon. Once we find the reverse side of the beacon, we run our collector alignment state machine as described above with one modification. Since we're not trying to collect a sample in this case, we use the IR rangefinders to get ourselves approximately a meter away from the back face of our starting platform. When aligned with the back, we perform three 90 degree crab movements to get ourselves around the backside of the starting platform and onto the platform itself. At the end of the run, the rover will be facing in the opposite direction of how it started.

F. Software Infrastructure: In order to implement software on microcontrollers there was some significant work that had to be done to get programming environments functioning. Although many base software modules were inherited from previous years for the PIC32 UNO board, there were a few things about them we did not like. We studied the PIC32 reference manual and were able to make an efficient UART module adapted from old code inherited from the previous year. We added a better string printing functionality using variadic functions resulting in a printf()-like subroutine as well tested simultaneous high speed UART over both ports. As opposed to a previously implemented UART library which was limited to 115200 baud, we flipped a UART register bit and increased a circular buffer size to achieve higher speed data transfer. With these additions we achieved near 1mBaud data rates verified using loopback on both UART ports on an UNO32 board.

When learning to use the MPLABX programming environment we developed coding standards using templates and automatic function block comment generation to ensure consistent coding style for the project.

On the SuSE side of software, we didn't have the luxury of an IDE to organize all software in a very pleasing directory structure. Instead, we opted to create our own directory structure and author a complex Makefile that could properly map to all the right folders. While authoring the more complex
where users will be able to place samples on a virtual field, this, we’ve proposed the creation of a data mining application, that have a higher probability of containing a sample. To do this, we’ve proposed the creation of a data mining application, where users will be able to place samples on a virtual field, and this data would be stored in a database, to be interpreted by the rover's navigation algorithms.

d) Artificial Intelligence Research: Faced with the challenge of establishing a sophisticated method of searching the field for a sample in Level 2, we began consulting some AI papers and textbooks. Our first paper consultation was the original paper describing D* Lite, “Fast Re-planning for Navigation in Unknown Terrain” [11]. From this, we gained a good understanding of how D* Lite derives from A* and traditional D*, as well as some holes in D* Lite that last year’s team didn’t discover. In particular, we noted that out D* Lite implementation needs to account for dynamic changes to the location of the goal node, whereas Level 1 has a static goal node [10].

Following D* Lite specific research, we turned to techniques used in Game AI for learning-based artificial intelligences. This research would be used in interpreting data given from our data mining app. From this, we discovered four techniques that were appealing to said interpretation. These techniques and their pros and cons are given below.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>Influence Maps</td>
<td>Can spread out probabilities of samples being in particular nodes on the map</td>
<td>More data attached to each node in 400 x 400 graph</td>
</tr>
<tr>
<td>Parameter Modification</td>
<td>Could possibly rule out bad data in data mining app</td>
<td>Implementation is notoriously slow</td>
</tr>
<tr>
<td>Reinforcement Learning</td>
<td>Could get very precise values on probabilities</td>
<td>Training would be of similar complexity to Hail Training</td>
</tr>
<tr>
<td>Tactical Path-finding</td>
<td>Can assign priorities to probability weights for concern blending</td>
<td>Weights are very difficult to tune and could invalidate heuristics for path-finding</td>
</tr>
</tbody>
</table>

Table 6 - Game AI techniques analyzed for interpreting data mining app data

e) Gamification of Level 2 of the Competition: Once we reach Level 2 of the competition, the dynamics of the competition change completely. Some of these changes include: collecting more samples, having multiple rovers on the field at once, having less time per sample to collect, returning to the starting platform is not as trivial as retracing the path to the pre-cached sample. At this point, level 2 becomes an optimization of an Easter Egg hunt. Because of this, we need to look at Level 2 as a strategy game to be optimized. The four main points of consideration and the proposed optimizations are detailed below.

- Other bot gets pre-cached sample first: In this case, we’re assuming that all other rovers on the field will plan to grab the pre-cached sample first, since they would already know its location. This would cause a large amount of congestion at the pre-cached sample’s location. To avoid being caught in the congestion, if we detect that another bot will beat us to the pre-cached sample, we will choose a new goal based on the map we retained from Level 1.
- Other bot is going for the same sample as us: If we detect that another rover is going for the same sample we are, we again risk being stuck in line for picking up the sample in question. Our approach
to this is to simulate (based on our own data), whether the other rover would beat us to this sample, and then use timing data from our sample collection mechanism to see if the other rover would also beat us to picking up the sample. If we figure that the opposing rover would beat us to the sample, we will scout for a different sample in the nearby area until the path to the sample in question is clear.

- Running out of time: We calculate whether we're running out of time based on how far we are from the starting platform and what we're currently doing. If we determine that time for Level 2 is short, we abandon paths generated from D* Lite in favor of a greedy path-finding approach. This means we'll take a straight shot back to the starting platform and swerve around obstacles as we encounter them.

- Choosing new goals: Our approach to choosing new goals directly overlaps into our Smart Searching Algorithm. In particular, we'll also take into account the localization data provided by WPI. Some of this data provides us with points of interest in the field. Points of interest are also likely points where a sample may be located. To handle the provided points of interest, each time we discover one, we'll add them into a priority queue. When we find ourselves in a situation where we can't compute a good probable path to some samples, we'll instead navigate to the nearest point of interest and search for samples around there. On top of this, we can identify further points of interest based on data captured from seeing judges walking in certain directions on the field.

- Data Mining App: Our data mining application began life as a PC game written by a group of UCSC Game Design students. In this game, players are attempting to colonize Mars by establishing a network of colonies, each of which sending out rovers to collect resources on Mars. While in the individual bases, players would place rovers in a predefined grid, and then program them to resource locations and back to the base [14]. The game was constructed using the Unity game engine, which provided support in browsers as well as integration with MySQL databases.

Having all of the necessary low-level game engine tasks already available to us, as well as most of the necessary functionality already written in code, using this game as the basis of our Data Mining App was an obvious choice. We began by first giving the code a thorough pass and playing with the basic functionality to understand how all pieces of the game are integrated together. After this, we then began a UI redesign of the application to something more fitting of our desired user experience as shown below.

On top of UI, the MySQL database was also setup using a hacked PogoPlug. Originally, the PogoPlug was to be used to host the repository for our software, but after some technical difficulties with hosting a Git repository on a PogoPlug device, and also determining that having direct control over our server wasn't necessary, it made more sense to relegate the PogoPlug device to serve as a database for the data mining application.

Despite the work put into our data mining app during Winter quarter, the workload during Spring quarter caused us to rethink whether using this app would be feasible. Originally planned to be completed by mid-February, the data mining app encountered various delays associated with just having Level 1 Navigation working as well as other senior design deadlines. By mid-April, with a looming qualification deadline for the competition on May 8th, the decision was made to abandon this app. Part of this decision came from the growing possibility of having an insufficient data set by the time the competition came around. Another part was simply proper prioritization of Navigation tasks. Resuming this application would be an early task for a future team to take on.

- Smart Searching Algorithm: Given the considerations in our gamification of Level 2 and statistics provided by the Data Mining App, we can develop a logical process for using this data under these considerations to make sophisticated decisions on where to go in the field, and how to handle particular unique situations. This is illustrated in the state machine (refer figure 51).

- Level 1 Navigation: For our implementation of navigation for Level 1, because last year's team was successful in navigating for Level 1, we were able to borrow a lot of ideas and source code from their successful implementation. In our case, we were also able to add in features that last year's team did not that give our Level 1 navigation scheme some serious optimization. In particular, our Level 1 navigation scheme can handle 45 degree turns and 90 degree turns, versus just 90 degree turns.

Given that we had last year's implementation of this very problem, most of the challenge in actually implementing the scheme came from filtering out the weaknesses of last year's implementation. Most notably, when reworking the navigation code to control the rover as it is now, several state machine iterations were done in the midst of the software being written.
This process was essentially doing logical testing of the code before it was completely written and ready to go.

Another challenge that arose from the Level 1 navigation scheme was turning the rover according to its position on the grid. A brute force method was taken to implement this aspect, considering each separate case. These are illustrated in figure 34.

![Figure 34 - All possible 3-way continuous path connections](image)

**i) Proposed Testing Process Description:** Early in the quarter, we derived a testing scheme for the navigation of the rover at all levels of production. This was broken into 3 stages as follows:

- **Stage 1:** Strict Software Simulation with OpenGL visualization tool
- **Stage 2:** Physical test on mini-rover provided to us
- **Stage 3:** Integration on final rover design. Each stage in this scheme had a well-defined purpose. These are as follows:
  - **Stage 1:** Test strictly the logic behind our navigation scheme so that we can figure any software flaws before introducing more points of failure on the actual rover.
  - **Stage 2:** With the logic behind the software thoroughly tested, we can now get an idea of how integration with basic hardware.
  - **Stage 3:** See how the physical aspects of the final rover design affect the performance and decisions made by navigation software.

Given the rapid progress on the mechanical aspects of the final rover design, stage 2 of this testing process was eliminated as it became unnecessary. Furthermore, stage 1 of this testing process was heavily reduced due to the demands of the Data Mining App as well as the amount provided from last year's implementation.

**j) Driving in a square:** Our first navigation test revolved around driving a very simple, pre-determined path. This test was more to ensure that our driving and steering controllers were fully functional and calibrated. On top of this, we were ensuring that our communication between laptop and micro-controllers was functional.

**k) Driving path dictated by D* Lite:** Included with last year's navigation implementation was a D* Lite Visualization tool written with OpenGL. As far as we can tell, this tool went unused last year, but for us, it provided the basis of our own visualization tool. The tool allows us to make arbitrary changes to a predefined grid, and see how D* Lite replans its shortest path given this data. We expanded this tool to give some insight into the behavior of our navigation state machines as well as our wheel encoders that are controlling driving and steering.

![Figure 35 - D* Lite Visualization tool](image)

This served as the basis for our second navigation test. In this case, we're attempting to see whether the rover can follow a path generated by D* Lite, as well as dynamically change its path once a change to the D* Lite path is made, whether this be from detecting an obstacle, or just changing our goal location.

**H. Image Processing**

What was required for image recognition was a vision library toolkit that used a C interface. The toolkit had to be able to operate in real-time analyzing 2D images. Through research, we found that the open source library OpenCV was a good match for these criteria. OpenCV is a C++ based interface. It offers different methods for sample detection such as color detection, edge detection, and LBP/Haar Cascades. Researching each recognition option available, we learned about the different features they offered.

Color detection uses different RGB thresholds to detect a specified color. These thresholds are typically predefined in code. As a consequence, colors are not always able to be detected due to RGB pixel changes from different lighting exposures. If we were to use this during the competition, this phenomenon might occur while experiencing different
weather conditions. Because of this, color detection was not a good candidate for sample recognition. An example depicting the difficulties of choosing color thresholds can be seen in Figure 36.

![Figure 36 – Color tracking. (Left) Blue object being detected. (Right) Gradient of blue that is recognized from object.](image)

Edge detection finds edges by recognizing the differences in color intensities. In order to detect this, the Sobel derivative is taken. The Sobel derivative takes web camera image matrices and takes the derivative of their sub matrices with respect to their x, y, and z components. This allows color intensities from different sub matrices to be compared resulting in edges to be found. For our sample recognition, even if the edges of samples were detected, it would not necessarily have been the answer to finding samples. Some type of detection cascade would have to be added.

![Figure 37 – (Top) Picture of steam engine. (Bottom) Edge detection used by applying Sobel derivative.](image)

This is a fast detection method processing wise, but did not seem to be the best choice for our application initially. What we thought would work better is a method that maps the exact sample color thresholds. What we ended up choosing for sample detection initially were Haar Cascades. Haar Cascades work with image pixel intensities to detect samples. Each pixel is scanned to be compared against a sample’s cascade deciding whether to check the next pixel, or return that there is no detection. While scanning each pixel is computationally intensive, it allows for less false positives.

2) *Implementation*: Using Haar cascades, our team needed to create a training set for the desired samples to be detected. A set of approximately 100 positives and 20 negatives were used for the first iteration of Haar training. This number was less than what was recommended, but our team wanted to verify that Haar cascades were a practical method with respect to the training process. After the set was trained, an XML classifier was produced. This XML file was then be used by the sample detection program.

After verifying that we could perform the Haar training process and seeing how long it took to generate Haar cascades, we realized that we needed more computing power to iteratively improve our object detection. At first, we entertained a few options with more powerful computers we owned. Unfortunately, even with more powerful desktop computers, the turnaround for new Haar classifiers was only marginally better. A super computer was our best option after that, and UCSC was able to give us access to their campusrocks and citrisdance servers. Each server was designed with a different purpose; campusrocks being used for many batch jobs and citrisdance for long jobs.

Our first attempt at speeding up Haar training was to use the multiple clusters of the campusrocks server, where each cluster would be running a different instance of Haar training for a different process. This would’ve involved submitting just one script to the server that would start up training for each sample and do the distribution on the clusters itself. Having little background in distributed computing, no ability to install software on the server, as well as no ability to ssh into the campusrocks server, this wasn’t a very practical option and we moved onto citrisdance.
Once we added more overall samples, we saw a number of issues with OpenCV’s Haar Training process. The first of which was in the creation of sample vec files. When running the training program with over 1000 positive images, we saw that the creation of positive samples process would drop some positives. Once our training script got to the cascade training, it would crash due to the lost positive samples. Turns out this is a known bug in the training program and the recommended workaround is to give 90% of the actual positive sample count to the cascade training program. This is one of many “delicate hacks” we needed to work around to get Haar training working [18].

Another issue with the training we encountered was in how the cascades themselves were generated. When trying to load the cascades we trained into our image processing module, we would always fail to load due to the XML file not having a type ID that the loading function required. A quick call to sed to add in this type ID ourselves was made to our training script to account for this.

During our time getting image processing training to work, we also explored into making LBP cascades to recognize specific patterns given on harder samples as well as for use with our home beacon. One important test we did in this regard was comparing the training times between Haar and LBP cascades. We first tried this with Haar and LBP cascades for the pre-cached sample. It was noted that a 40 stage Haar cascade running on the citrusdance server with 4GB of RAM took about 5 days to complete. A comparable LBP cascade took about 8 hours. Being baffled as to why the Haar training took as long as it did, we took a deeper look into how the OpenCV cascade training program works. Our initial assumption was that if GPUs were available, the code would be parallelized on GPU and complete significantly faster. After browsing through the source code and getting an idea of how the machine learning works, we found no evidence of GPU support in the training program at all. As of this writing, the OpenCV repository contains an incomplete version of GPU support for Haar training that specifically says to use at your own risk and works completely different than how we were doing Haar training previously. In effect, using citrusdance to do our image processing training was not the significant improvement in turnaround that we were hoping for. We wanted to turnaround cascades within a few hours, but having 32GB of RAM on CPU that we could safely use still gave a notable improvement over using one of our lab computers.

After finally understanding all of how Haar and LBP cascade training worked, we began to ways to optimize the process. To OpenCV’s credit, training can actually be interrupted and restarted again at a later time. This is due to the training program generating an XML file for every stage of training it completes [19]. If the user stop training at one point and wants to resume it later, they can just invoke the exact same program. This proved to be invaluable for us as we could start training at one location and end it at another.

The program to detect the Haar classifier had three main goals. The goals were to see if the desired sample was detected multiple times, check both distance and pixel boundaries to rotate between cameras, and draw a rectangle around any detected sample.

\[
distance_{x,y} = \frac{\text{focal length}_{\text{image}} \times \text{height}_{\text{sample}} \times \text{height}_{\text{camera}}}{\text{height}_{\text{pixel}}}
\]

Figure 39 – Equation used to calculate sample distance in meters [20]

In addition, a message was returned saying if the sample seen in the image was to the left, right, or centered in the window. This was accomplished using pixel boundaries as a method of hysteresis.

After the first iteration of generating classifiers for sample detection, we decided to add more positives and negatives to our training set. We added approximately 2,500 more positives having each sample’s positive total being around 3,000, and add negatives having their total be approximately 10,000. The images positives taken of the samples have 360 degree views as well as from the top down.

At this point of the project, a total of three cameras were used. The first two cameras were those mounted on the top and the bottom of the rover. The top camera detected samples off in the distance and initiated the pursuit. Once the top camera detected the desired sample at a specified distance threshold, the rover transitioned to its bottom camera. Continuing the pursuit of the sample, using the bottom camera, the rover centered on the object.

As the sample came closer to the camera, having its center point move down in the camera window, a transition was made to the third camera; the sample camera.

During the transition from the bottom to sample camera, a blind spot between the cameras was experienced.
To be sure the rover does not lose sight of the sample when experiencing its blind spot, a watchdog timer was implemented. The watchdog timer was used to confirm a single detection of a sample before remaining in the sample camera state. If the first detection was not completed within the time limit, the rover switched back to the bottom camera. After the first detection was completed and the current state is the sample camera, the rover attempts to center its collection mechanism directly over a sample. At the point of 500 confirms of the sample being in the center of the collector, an acknowledgement gets sent to drop the collector and retrieve the sample.

Testing was conducted after the sample detection was implemented and classifiers were compiled. As a first test, a control was conducted indoors. The test was to see if the web camera could detect the pre-cached sample while mounted on a laptop. A green folder was held behind the sample due to the classifier being trained with negatives from a grassy green field.

During the first test, the pre-cached sample was able to be detected and locked on from a range of 2 to 20ft. This can be seen in figures 60 and 61.

For our second test, the top and bottom web cameras were tested on the rover. The test was conducted on a grassy field during a sunny day. The sample that was used to test object detection was the pre-cached sample. Detection was attempted at multiple distances ranging from three to twenty feet. When the rover attempted to detect the sample at twenty feet, it was not detected due to heavy light on the camera image.

One feature that performed well in this test well was locking detection. If the object was lost around 15ft and then redetected, it did not need to be re-centered to be seen. In addition, when the sample was detected closer than four feet, the top web camera switched to the bottom as seen in Figure 44.
Bringing the sample closer, the same issue continued. At this point, it was apparent that the pre-cached sample was being washed out completely by the sun light. Acknowledging this, a shadow was casted over the sample to confirm that this was the heart of the issue. Testing this theory out, the sample was detected at a distance of ten feet. Observing this behavior led to the conclusion that a larger training set was needed with more diverse lighting conditions.

Solving this issue, we compiled new Haar classifiers with approximately 2,500 positives and 10,000 negatives for each sample. At this point the problem was no longer not detecting objects in saturated lighting, but detecting too many of them. In other words, a significant number of false positives were being detected. Trying to detect the pre-cached sample, backgrounds such as the white clouds and buildings were being mistaken as the desired object. In order to fix this, a moving average filter was implemented to make multiple detections using sample classifiers, and take the average of their center points. Having the averages a single detection of the sample’s center point is returned. This allows for actual detections of the samples to outweigh false positives given the classifiers are robust.
Photos

Figure 20 – Plug Board Schematic

Figure 21 – PTN Regulator Schematic
Figure 22 – PTN Regulator PCB Layout

Figure 23 – Final Rover Startup Circuit
Physical Layer Description
(Winter Checkoff)

Figure 24 – Differential Signaling

Figure 25 – Differential Signaling
Figure 30 – Front Uno Configuration

Figure 31 – Rear Uno Configuration
Figure 36 – Pause Functionality State Machine

Figure 26 – System Block Diagram
Physical Layer Description

Figure 27 – Differential Signaling

Figure 46: Sample Collection State Machine

Figure 47: UML of Subsystems on Master Laptop
Figure 48: Last year’s path-finding algorithm research

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Calculate from destination</th>
<th>Calculate from start</th>
<th>Node based</th>
<th>Grid based</th>
<th>Linear</th>
<th>Heuristic function</th>
<th>Recalculate based on new data</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Simple</td>
</tr>
<tr>
<td>LPA*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
<tr>
<td>D*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Very Complex</td>
</tr>
<tr>
<td>Focused D*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
<tr>
<td>D* Lite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Simple</td>
</tr>
<tr>
<td>Field D*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Same at D* Lite</td>
</tr>
<tr>
<td>RRT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
</tbody>
</table>

Figure 51: Smart Searching State Machine
I. References

https://www.youtube.com/watch?v=caxqV7R_VfQ
http://www.nasa.gov/images/content/660225main_srr_green1_466.jpg
[12] UCSC-Rover-2013’s repository: https://www.dropbox.com/sh/u6f8okj4dxop98y/9wT8LDkAEk