Final Project Rover Report

Fundamentals of Robotics

Olin RoboLab

Luke Raus

Team Delta

with teammates Ally, Rajiv, Chris B, Nathan, Anusha, Maya

Olin College of Engineering

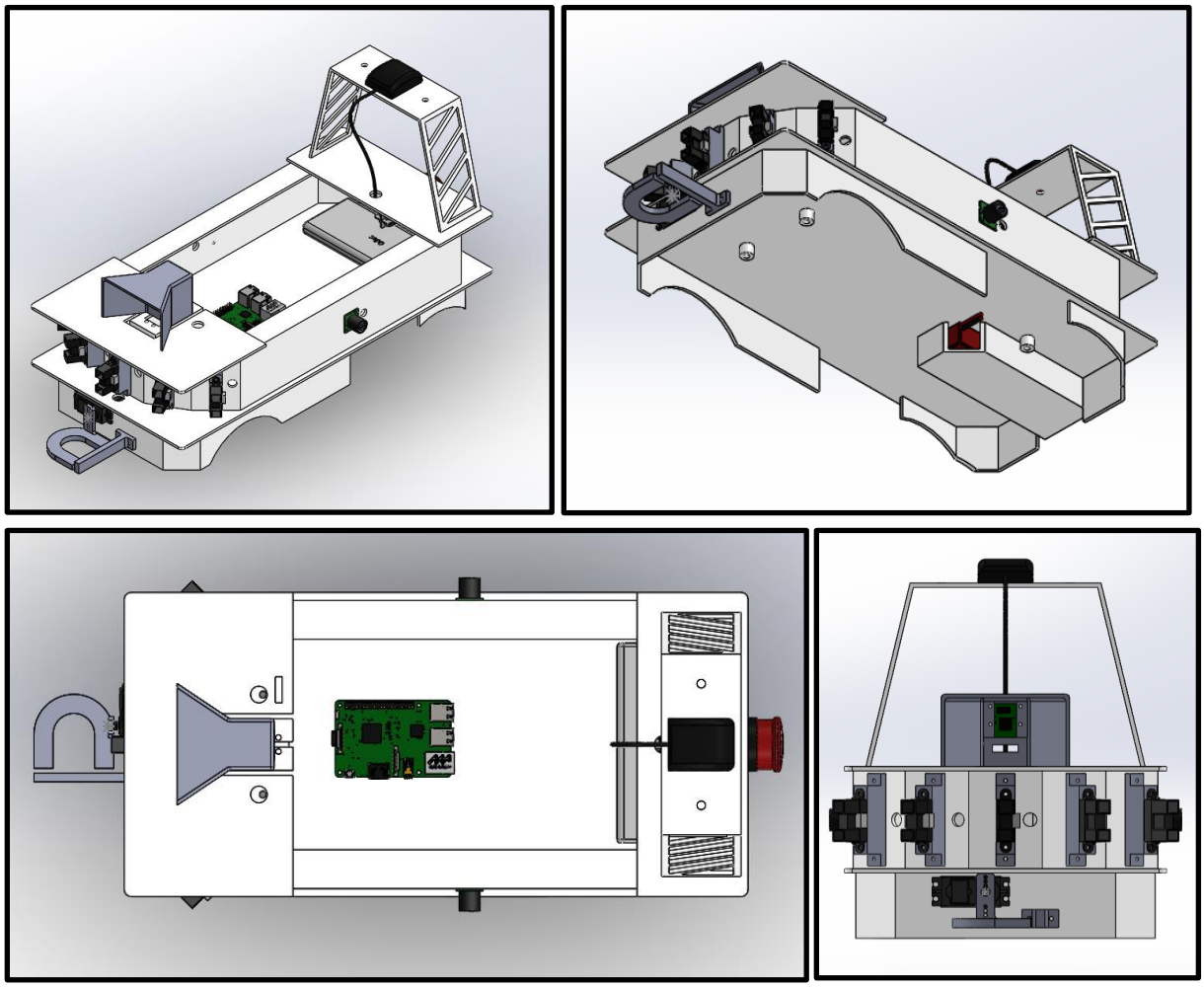
May 14, 2021

Contents

Mechanical Design

Overview

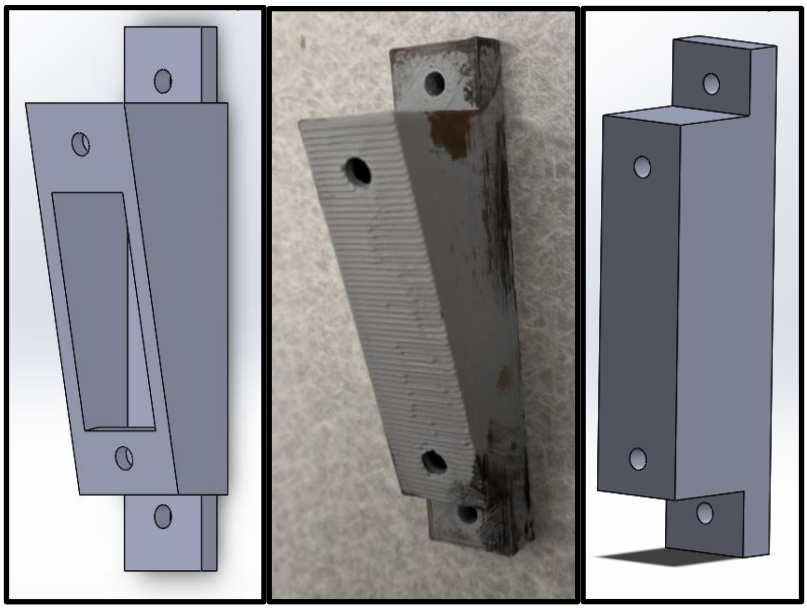
Since we wanted to have a functioning, constructed rover ready for software development and testing as rapidly as possible, our design was largely based on the given hull design. Therefore, most of the mechanical effort on this project went towards mounting our sensors (such as IR sensors and the pan/tilt mount), the payload deposit mechanism, additional structural elements for robustness, and aesthetic enhancements.



The above images show the final rover CAD from several angles. The main body shape is retained from the original. Our IRs fan out ahead to detect cones or walls and the sonars face sideways to detect our robot's lateral distance from a wall. Note our additions of the payload deposit mechanism, placed on the front of the robot for easiest alignment in software, and our custom camera mount.

Sensor mounting

When choosing how to mount our Sharp IR sensors, we needed to decide whether pointing them angled toward the ground or straight ahead would produce better results. Over the course of the project, we designed and 3D-printed three iterations of the mount, shown below, to get real-world data to inform our choice.

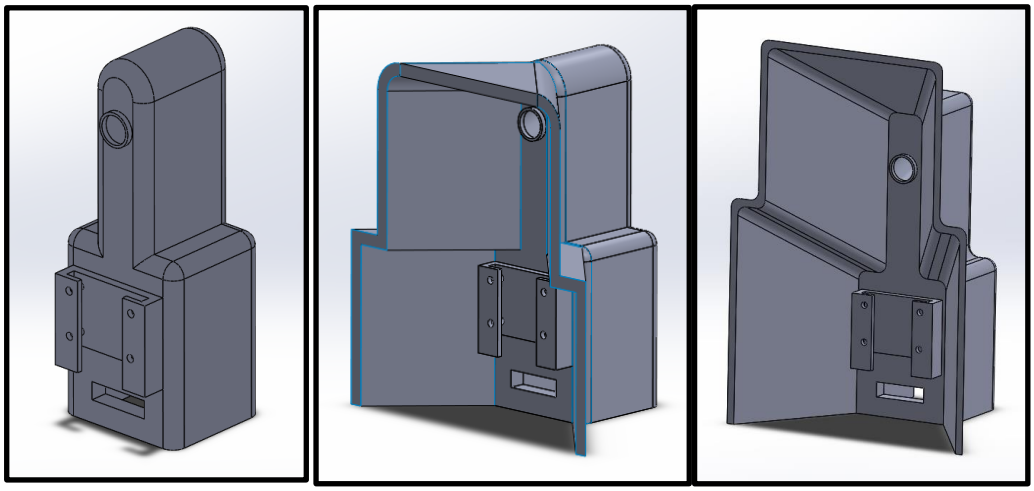


After gathering data, we realized that the tilted-down infrared beam's collisions with the textured brick surface led to excessive noise in our readings, which was undesirable. We then chose to mount them straight forward, which proved to more accurately let us detect whether an object was in range or not, since we would get an "infinite distance" reading instead of measuring to the ground. Hardware iterations also improved the screw mounting by giving us more plastic to in which to thread our self-tapping mounting screws.

Camera mounting

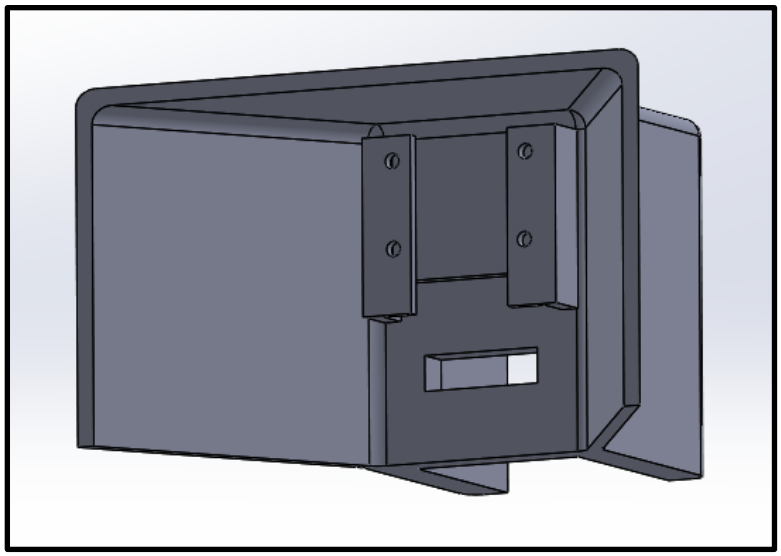
We realized early on that being able to point our PiCam towards the ground to perform CV on ground features (such as granite squares) would be highly advantageous. We ensured that our design gave the camera a reasonably high and forwards-pushed vantage point with which to take data.

We also integrated the laser line module into our early designs; while we were unsure whether we would develop code to use the laser, we wanted to have the option. Thus, the mounts also included an offset and angled mounting channel for the laser above the camera itself.



The three iterations above show the same basic design but with the addition of, and then improvements to, a sunshade to keep ambient sunlight off the camera and laser. The initial sunshade required horizontal overhangs that proved difficult to 3D print, so we changed to an angled taper which printed more easily and without support material.

As the demo approached, we realized that we would not be able to make use of the laser line so designed and printed a more streamlined pan-tilt head without the mast necessitated by the laser. This mount was ultimately used on the robot and proved effective.



Bumper addition

When initially driving our robot around with the joystick, the long delay between sending commands and their execution made it very easy to crash the rover. We had a few mishaps where we crashed the robot and broke some 3D printed parts and especially tended to crack the fragile Sintra board.

We realized that our robot needed to be able to collide with real-world obstacles without breaking apart every time, so we designed and fabricated a heavy-duty bumper to let the rover crash head-first into walls without breaking. The bumper was cut from scrap polycarbonate plates, which was a good material choice because of its resistance to cracking and overall impressive durability.

This bumper proved quite effective and gave us tremendous peace of mind while developing software, knowing that a mistake would not require us to halt development to fix the rover hardware or replace any broken sensors.

Deposit mechanism

Our team chose to mount our payload deposit mechanism on the front of the rover to make approaching deposit stations effortless. We wanted a simple and robust mechanism which would keep the payload secure while driving, and then smoothly and reliably drop the payload when desired.

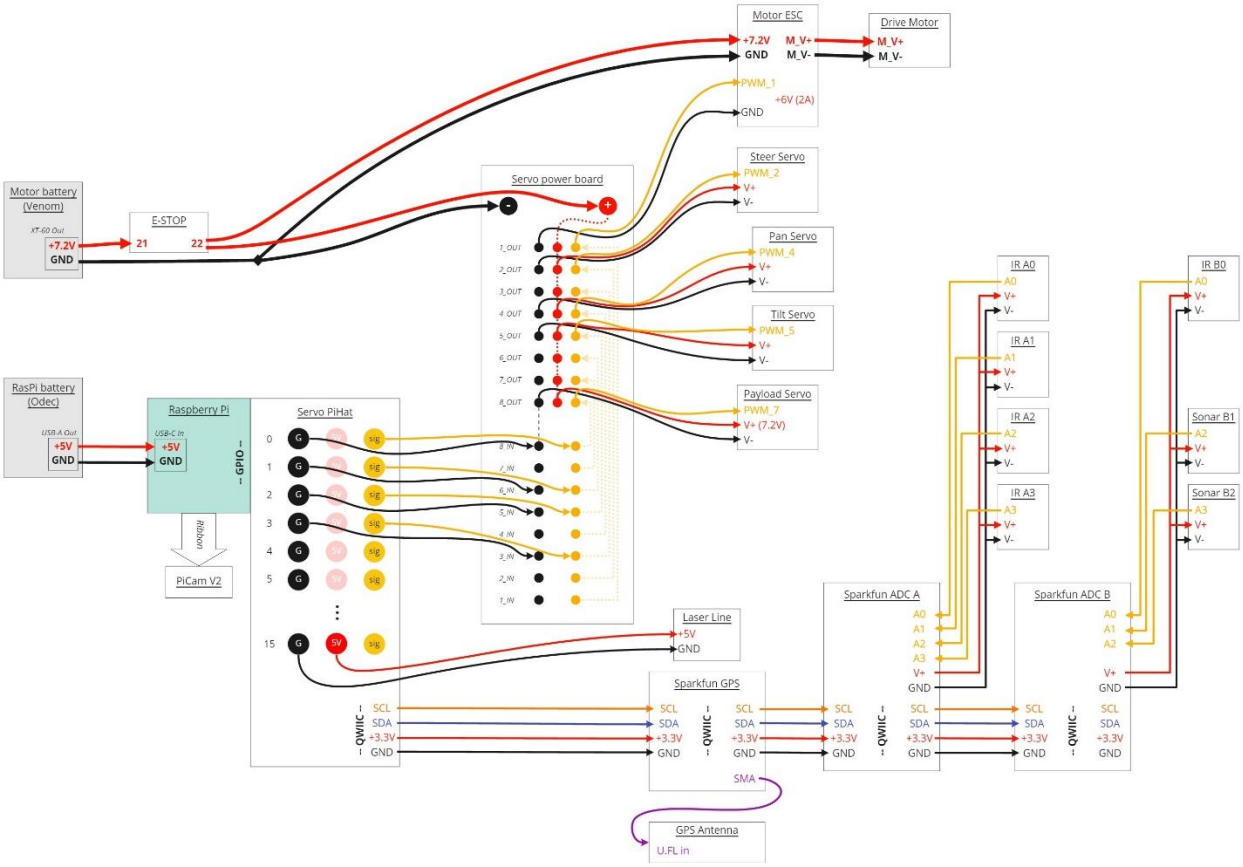
A simple fork which supported the top flat portion of the payload fulfilled these design needs for us. The payload put very little torque on the servo when held steady, and we could simply rotate the servo sideways as shown below to drop the payload. We added a short bar to ensure that the payload could not slide out of the fork unless actuated. The deposit sequence is shown below.

The deposit mechanism integrated seamlessly with our polycarbonate bumper; we designed the two such that the payload would easily fall through the bumper gap, and the bumper simultaneously protected the payload and its relatively fragile 3D printed parts from impacts or collisions.

Electrical Design

Overview

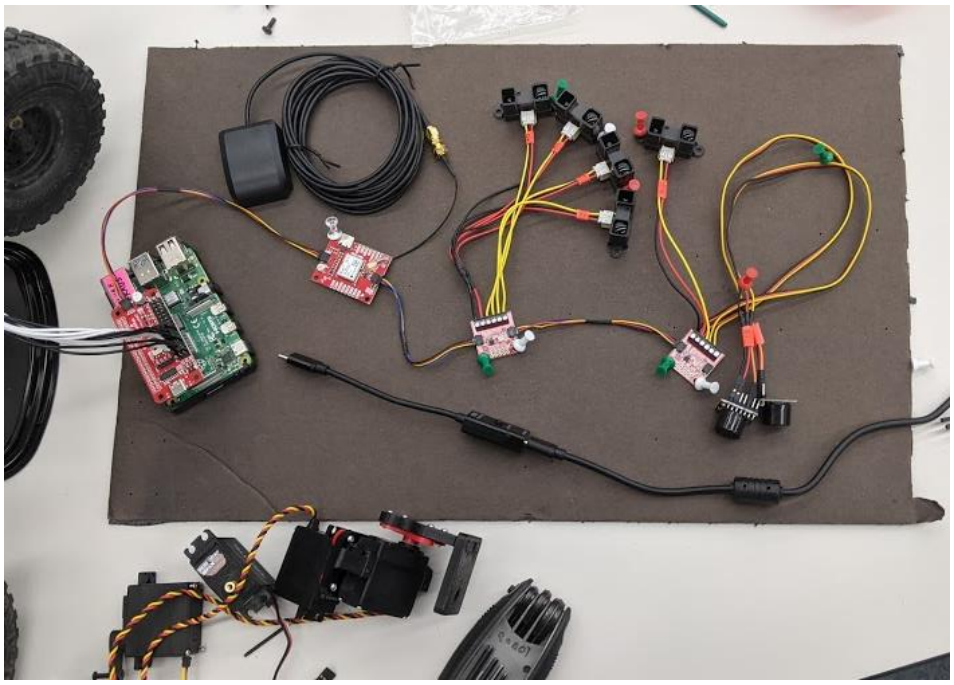
Our robot's electronics allow it to perform processing, sensing, and actuation. Fundamentally, this means that the Raspberry Pi which hosts the robot's central computation needs to be able to receive signals from our suite of sensors and send output signals which are used to drive our motors and servos. Separate power buses are maintained to keep the clean low-current power needed by the Pi and sensors from a dirty higher-amperage bus used for actuation.



As seen in the overall wiring diagram shown above, we used a Servo PHat to output PWM signals to each of our motors to drive them to a desired position or velocity. We also used Sparkfun QWIIC ADS1015 analog-to-digital converter modules to read analog inputs from our IR, ultrasonic, and GPS sensors and send this data to the Raspberry Pi over I2C via the PHat's QWIIC connector.

The diagram also highlights the separation in power supplies. A servo power board is used to isolate the servos' power draw from the PHat's 5V output. No power connection is made between the PHat and each servo, and instead only the PWM signal and reference ground signal are passed to the servos. The servos draw their power directly from the LiPo motor battery as supplied on the servo power board; while the servos are officially rated to support up to 6V power, each servo happily accepted the 7.2V power (which was actually more like 8.4V on a full charge) from the Venom LiPo battery. It is clear that no power connection is made between the upper "actuation"-half of the diagram and the lower "computation/sensing"-half of the diagram, as split at the servo power board.

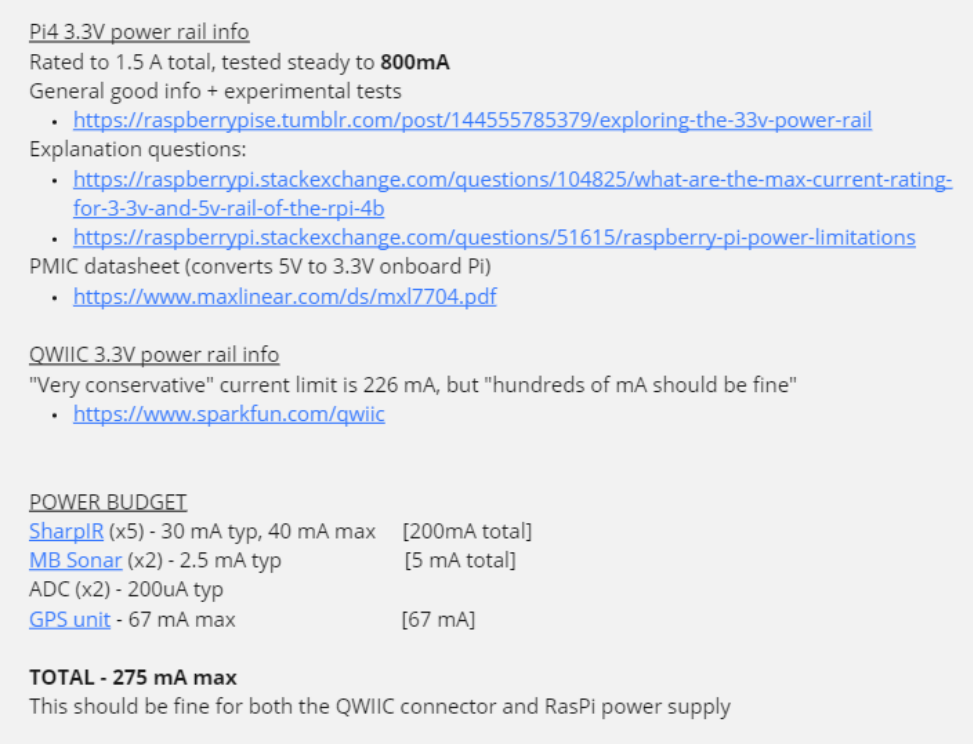
An Emergency-Stop button was also added to the system to allow an operator to cut power to the motors and servos to stop any of the rover's movements when necessary (i.e. when on course to crash into a wall at full speed). The E-Stop sits between the motor battery's 7.2V output and both the servo power board and motor ESC, while ground is always held common. Thus, when the button is pressed, the motors and servos get no power while the rest of the system on the 5V logic-battery bus is completely unaffected and can continue running with the E-Stop pressed. This saves lots of rover rebooting in the field!



Above, the low-voltage computation and sensing system is wired on a roadkill harness board, with connections made to the high-voltage rover actuation sweet out of view on the left. This let us validate all components and run initial software tests before committing to mounting each component on the actual rover body.

Power bus design

To ensure that all of our sensors would be able to run off the 3.3V power rail regulated by the Raspberry Pi and supplied by the QWIIC bus, we found the maximum current draw specified for each component and summed these as appropriate. We also had to read a bit more into the Pi's power management IC, which is responsible for stepping down the supplied 5V to 3.3V.



Pi4 3.3V power rail info
Rated to 1.5 A total, tested steady to **800mA**
General good info + experimental tests
- https://raspberrypise.tumblr.com/post/144555785379/exploring-the-33v-power-rail

Explanation questions:
- https://raspberrypi.stackexchange.com/questions/104825/what-are-the-max-current-rating-for-3-3v-and-5v-rail-of-the-rpi-4b
- https://raspberrypi.stackexchange.com/questions/51615/raspberry-pi-power-limitations

PMIC datasheet (converts 5V to 3.3V onboard Pi)
- https://www.maxlinear.com/ds/mxl7704.pdf

QWIIC 3.3V power rail info
"Very conservative" current limit is 226 mA, but "hundreds of mA should be fine"
- https://www.sparkfun.com/qwiic

POWER BUDGET
SharpIR (x5) - 30 mA typ, 40 mA max [200mA total]
MB Sonar (x2) - 2.5 mA typ [5 mA total]
ADC (x2) - 200uA typ
GPS unit - 67 mA max [67 mA]

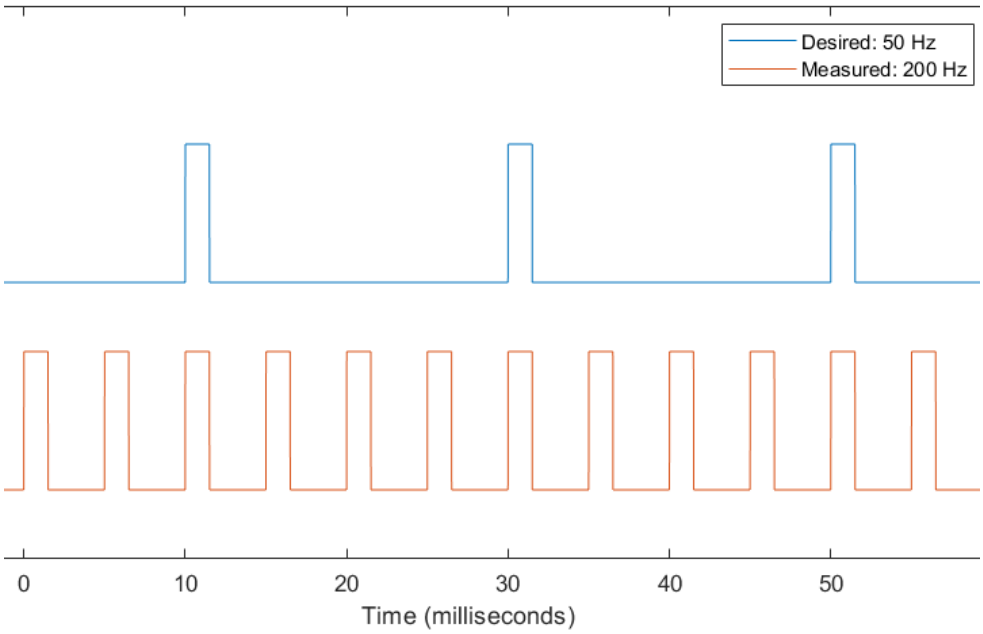
TOTAL - 275 mA max
This should be fine for both the QWIIC connector and RasPi power supply

While we technically exceed the specified max current for the QWIIC bus, this was noted as a very conservative limit, and our system was calculated as drawing no more than a few hundreds of milliamps at 3.3V, which is easily handled by the Pi's PMIC. We went ahead with this architecture and indeed experienced no issues with the 3.3V bus.

Servo frequency debugging

While testing our pan-tilt servos, we noticed that they behaved very noisily and jittery, especially in comparison with our other servos. One hypothesis suggested that the issue may be due to a voltage overload, since they were only rated for 6V instead of the 8.4V being supplied by the LiPo in practice. However, we observed the same behavior at 5V supplied by the PHat and at 6V supplied by the motor ESC, ruling this out as the likely cause.

Further investigation indicated that the PWM pulse width was not the issue, as PHat was outputting pulses with widths in the 1000-2000 microsecond range as expected.



Ultimately, measuring the output signal with an oscilloscope indicated that the PHat was outputting the incorrect PWM frequency, as it sent pulses at 200 Hz instead of 50 Hz, since the period between each pulse was measured at ~5ms instead of ~20ms. While the more modern digital servos could handle this increased frequency, the analog pan-tilt HiTec servos' internal circuitry could not. A quick MATLAB driver update for the PHat resolved this frequency issue and made the servos work smoothly as expected.

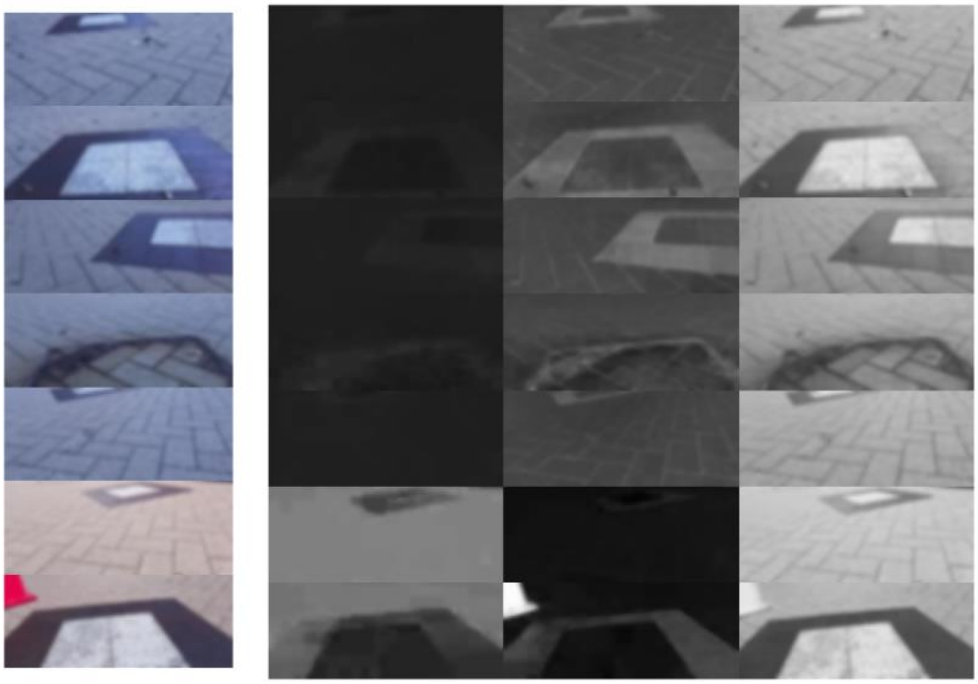
Control Software

Sense Overview

Lots of our robot's software falls under the "Sense" category: attempting to gain useful information from the sensors for further processing, such as by localizing, detecting goals, or detecting obstacles. We relied heavily on computer vision from the PiCam for these tasks, as some of our sensors (e.g. the GPS) were tested extensively but found to unreliable at the scale of the rover.

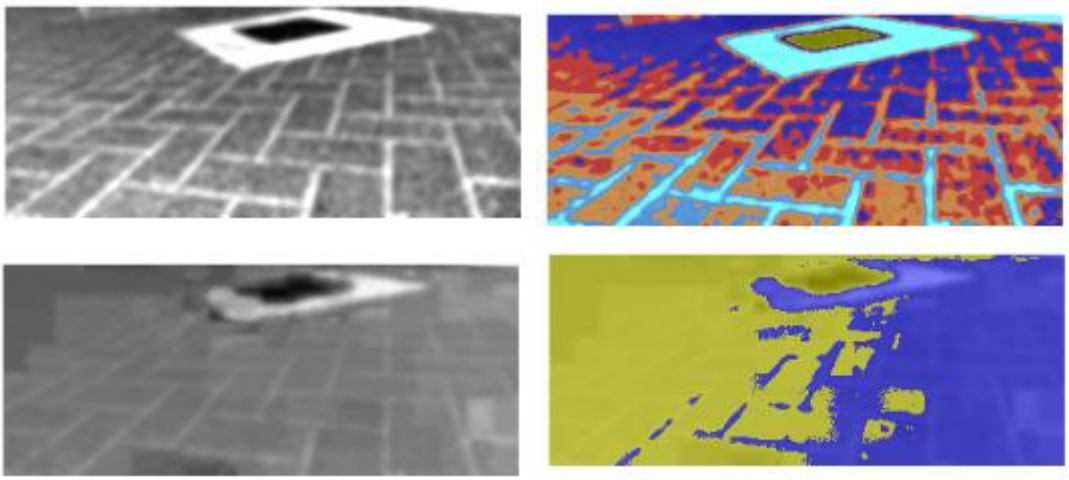
CV Landmark Detection (Granite squares)

Immediately upon inspecting the Oval track, we realized that the periodic granite squares (composed of a square of white granite surrounded by reddish-purple bricks amongst the tan bricks) should be identifiable with computer vision and would be useful for localization, as they would help us stay in the center of the circum-oval path. Efforts were made to attempt to reliably recognize the squares in the rover camera's field of view.



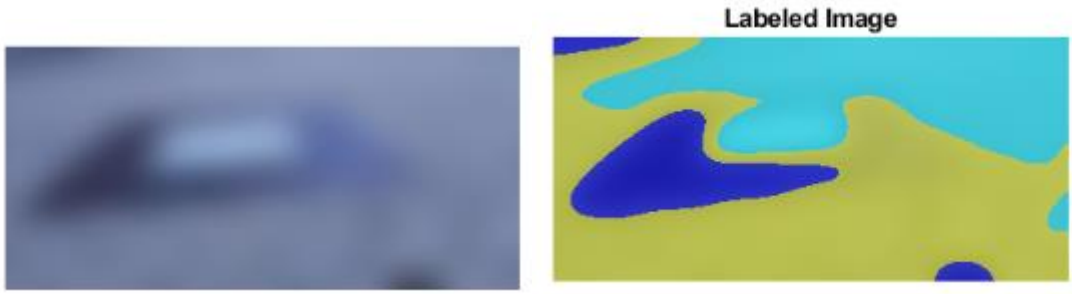
Instead of working in the RGB color space, we examined our collected images in the HSV color space, as we care much more about overall changes in lighting intensity and saturation than the (relatively arbitrary) red/green/blue channels. The above images show a color image decomposed to its H/S/V components, from left to right. Note that the purple bricks are distinguishable from the white and tan bricks in the hue and saturation channels.

We wanted to use a more sophisticated than simple RGB color masks, which are highly susceptible to changing lighting conditions. One approach we investigated was K-Means image segmentation, whereby the values for an image are clustered into a given number of groups which attempt to best represent the data. In theory, if a feature is obviously distinguishable from the rest of an image in a certain channel, a 2-cluster K-means might be able to label the feature and the rest of the image. However, getting useful information from K-means is often difficult, as shown in the examples below with K-6 and K=2, respectively.



Note that having many groups leads to irrelevant features getting clustered with the main feature, as in the brick cracks in the first case being identified as the purple bricks. But having fewer groups also means that the clusters tend to get picked based on more gradual shifts, as with the second image generally getting brighter towards the right-hand side leading to, effectively, a left and right cluster.

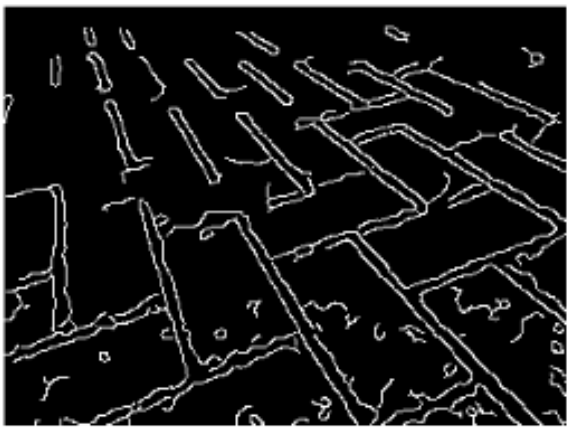
Attempting to smooth the image using Gaussian or Median blurring as below helped somewhat, but also made the lighting variations across the image more broadly pronounced.



After lots of exploratory work was done attempting to reliably detect the squares under diverse lighting conditions, it was realized that having this data might not even be useful after all if we could not robustly localize to the square (for instance, by mapping a quadrilateral to the detected shape) or distinguish between the various squares using some other localization technique. Rather than ignoring the swaths of tan bricks as "background", we shifted attention towards using these regular bricks to gain constantly-available input to the rover.

CV Orientation Detection (Brick edges)

We realized that the orientation of bricks on the ground is a useful thing for the robot to measure, as the bricks surrounding the oval are laid in many different sections, where the bricks rotate slightly to maintain a certain orientation relative to the oval's tangent: that is, they are laid to be rotated 45 degrees away from the oval's tangent at every point. This creates two sets of parallel lines, the average of whose angles should like tangent (or, if 90 out of phase, perpendicular) to the oval. Thus, by measuring these lines and angles, our rover should be able to tell how far its heading is from the desired heading to let it traverse the oval.

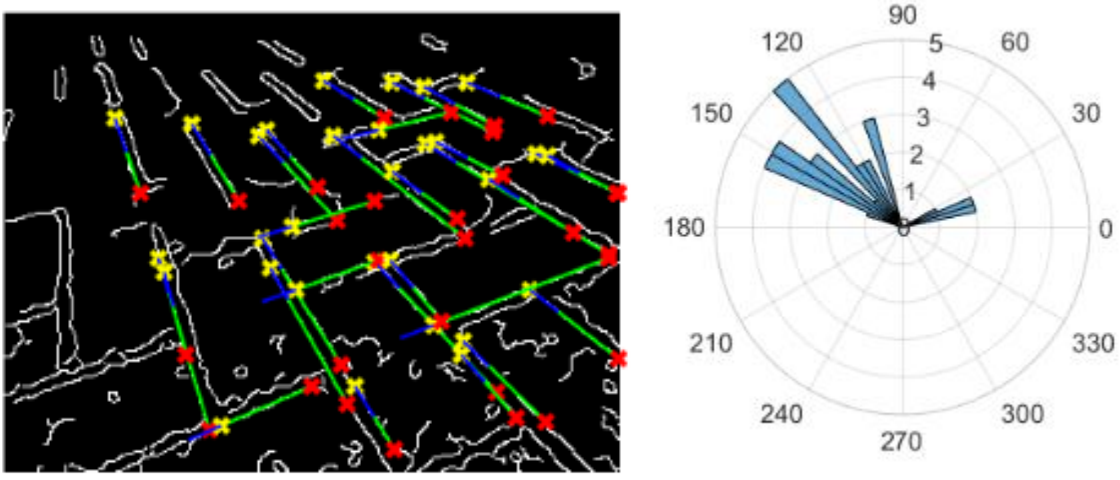


Computer vision code was developed to extract the angles of edge lines from a black-and-white image of the oval's bricks. Seen above is an adjusted snapshot from the rover's perspective which has been used for edge detection using an algorithm built into Matlab, resulting in the binary image at right. The "Canny" method in the *edge(...)* function was found to pick up the brick edges quite well while ignoring most of the noise from lighting and surface deviations.

The following code lightly filtered the image, adjusted the contrast, ran edge detection, and then Hough filtering.

```
V = HSV(:,:,3);

I = medfilt2(V, [5,5]);
I = imadjust(I);

% Run edge detection using "Canny" method
EdgeBW = edge(I,'Canny',0.3);
EdgeBW = EdgeBW(4:end-4, 4:end-4, :);    % crop edges

% Hough filtering
[H,T,R] = hough(EdgeBW);
P = houghpeaks(H,80,'threshold',ceil(0.32*max(H(:))));
lines = houghlines(EdgeBW,T,R,P,'FillGap',6,'MinLength',36);
```

Hough filtering is a technique where a binary edge image as produced above is searched for the line segments of various angles which best represent it. This was useful because it let us extract many different line segments as seen below. Having several dozen such segments made us less reliant on any single edge or small set of edges, thus giving lots of edges across the entire image a vote in the outcome. Of course, it was crucial to tune the minimum length of the detected edges to ensure that small noisy features were not counted too frequently.

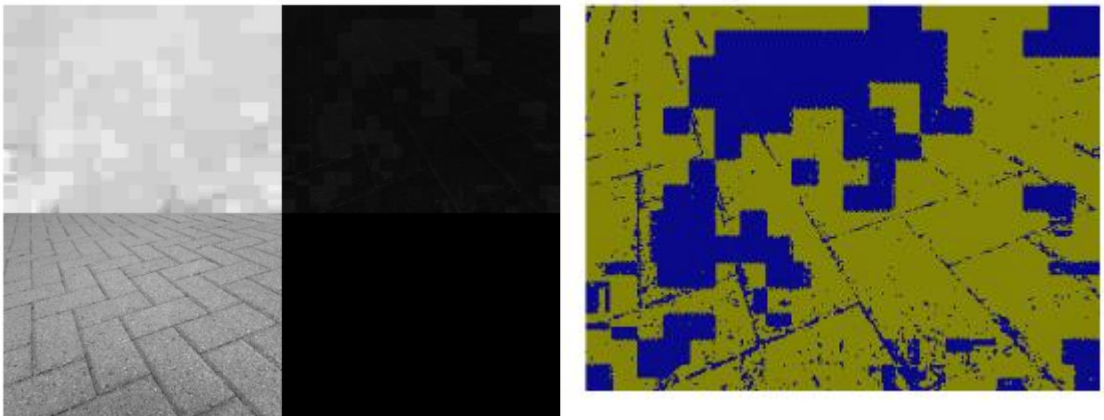


In the above image, the line segments detected by Hough filtering have their angles computed and plotted in an angular histogram. We can clearly see two distinct groups emerge, a large one around 130 degrees representing the leftwards-leaning lines and a smaller one at 20 degrees for the rightwards near-horizontal ones. This would indicate that the desired heading at halfway between these angles is around 75 degrees as labeled on the histogram.

```
nLines = size(lines, 2);
if nLines > 10
    % atan2 takes dY, dX
    % Y increases as you go down... so take negative of dY
    thetas = zeros(nLines, 1);
    for k = 1:nLines
        p1 = lines(k).point1;
        p2 = lines(k).point2;
        thetas(k) = mod(atan2( p1(2)-p2(2), p2(1)-p1(1) ), pi);
    end

    % Attempt to find an angle which is as far apart from thetas as possible.
    % Any thetas below this threshold get 180 added to them before separation.
    theta_split = fminbnd(@(x) sum(1./(abs(thetas(:)-x))), 0, pi/2);
    toShift = thetas < theta_split;
    thetas(toShift) = thetas(toShift) + pi;

    % Now that data isn't split at 0/180, find angle to separate batches
    theta_split_2 = fminbnd(@(x) sum(1./(abs(thetas(:)-x))), min(thetas), max(thetas) );
    theta_median_1 = median( thetas(thetas <= theta_split_2) );
    theta_median_2 = median( thetas(thetas >  theta_split_2) );

    theta_median_1 = rad2deg(mod(theta_median_1, pi));
    theta_median_2 = rad2deg(mod(theta_median_2, pi));

    if abs(theta_median_1 - theta_median_2) < 50
        heading = NaN;
    else
        heading = 90-mean([theta_median_1, theta_median_2]);
    end
```

This code then performs some simple statistics on the array of angles to determine the average angles of the two groups and thus the robot's heading relative to the seen set of bricks. First, an angle is chosen which maximizes its difference from all the existing angles to split the set into two buckets. (Note that some additional code is necessary because, the way angles are set up, lines that have nearly physically identical slopes might be associated with drastically different angles if they lie close to the horizontal and thus wrap from 180 to 0 degrees.) The median of these buckets is chosen to represent the buckets (to hopefully limit the effect of outlier data points), and the mean of these two medians is the heading relative to the brick angles.

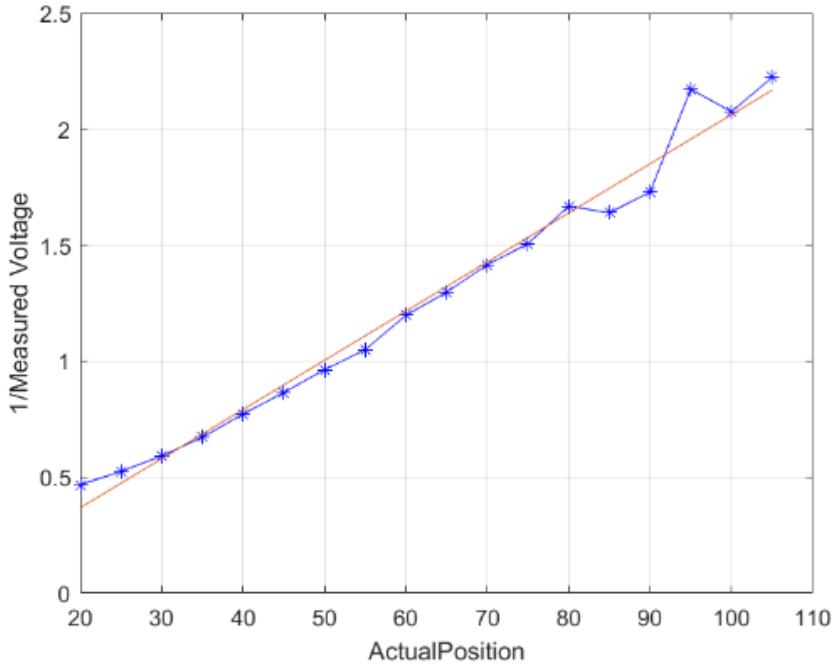
However, conditions were necessary to have the robot not get confused by granite squares, cones, or other obstacles which might add "lines" to our processing which do not correspond to the brick cracks as expected. We did this by looking at the saturation channel of the captured image. The brick cracks tended to have very low deviation in the saturation channel across the image, as shown below, while other features tended to have significant saturation differences.



As seen above, the hue and saturation differences across an image of just brick cracks were so small than K-Means clustering simply highlighted artifacts of the image processing. Thus, if the mean levels of the two groups was measured to be very small as in this case, we could be confident that the robot was just looking at the ground, whereas higher saturation differences indicated that a visual obstruction was in view and that the heading reading from this CV code should not be trusted. By incorporating a certain level of confidence-estimation into this function, it became much more useful out on the test track.

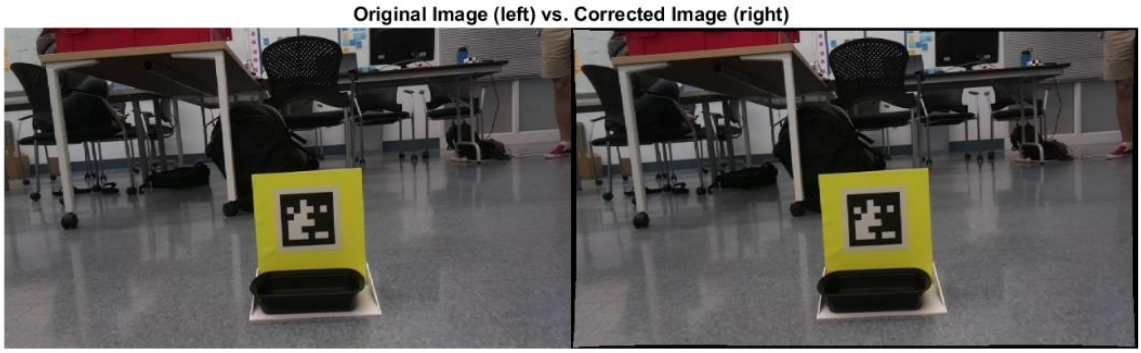
Sharp IR Calibration

To convert the voltage read from the ADC module from a Sharp IR sensor to a useful distance, the position-to-voltage function was linearized by taking the reciprocal of the voltage. Then, the data followed a line fairly well and let us read useful distances.

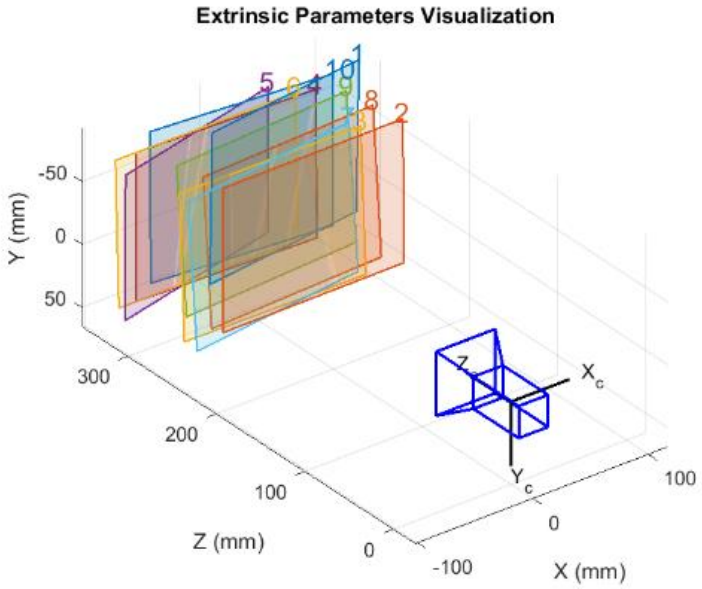


April Tag Calibration

Before we could run the readAprilTag function, we first needed to gather intrinsic data about our camera which could be used to correct for the distortions imposed by its optics or other physical characteristics. This series of steps allows a set of parameters to be generated which let the vision system generate an optically corrected version of its snapshot which is more useful to the April Tag progressing program.

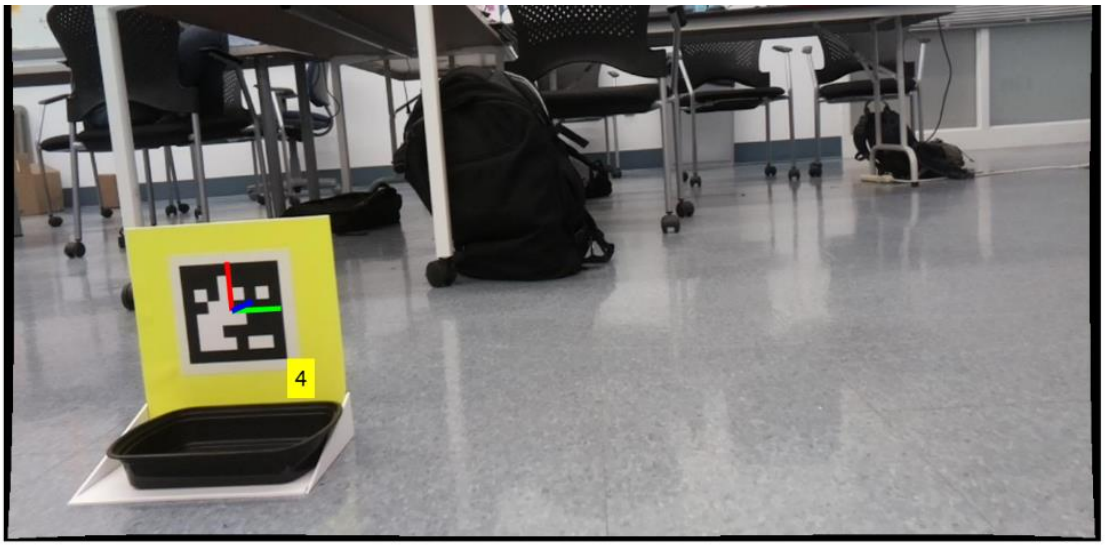


During the calibration procedure, the camera estimates its orientation and position relative to a specific printout of a grid of April tags. The calculated orientation of the sheet is visualized with respect to our camera below.



With the parameters determined, we were ready to call the Matlab readAprilTag function to measure the relative pose of the tag relative to our robot. This function returns a 3-dimensional rotation matrix and position vector which we needed to process to give useful commands to our rover.

We realized that since our rover and the April tags would (more or less) be at the same ground level during the demo, we could ignore information about the Z-axis orientation. This reduced the problem to a 2-dimenstional top-down one, which could easily be solved with a bit of trigonometry and an inverse tangent. Thus, we could readily compute the rover's distance from the tag and the difference between e.g. the deposit box and the rover.




```
[heading,distance] = get_apriltag_location(U,params,tagsize)
```

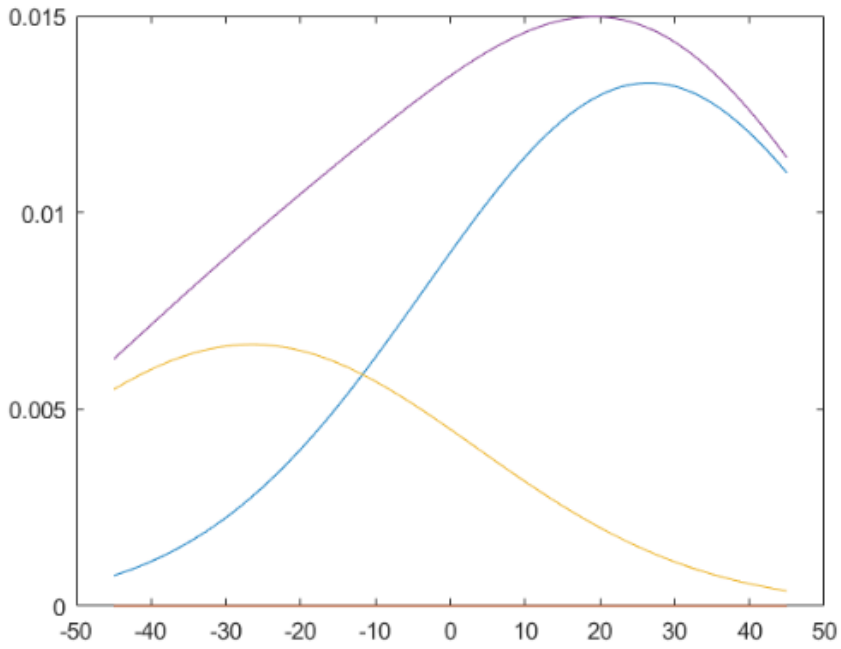
```
heading = -20.8979
distance = 2.0921
```

The April tag results were observed to be quite robust and gave us useful information for decision making and arbitration.

Think: Arbitration

Given desired headings from our several Sense functions, such as orienting to bricks, avoiding obstacles detected by IR sensors, and orienting to an April tag, we explored using a D.A.M.N arbitration system to decide what commands to actually send to the motors.

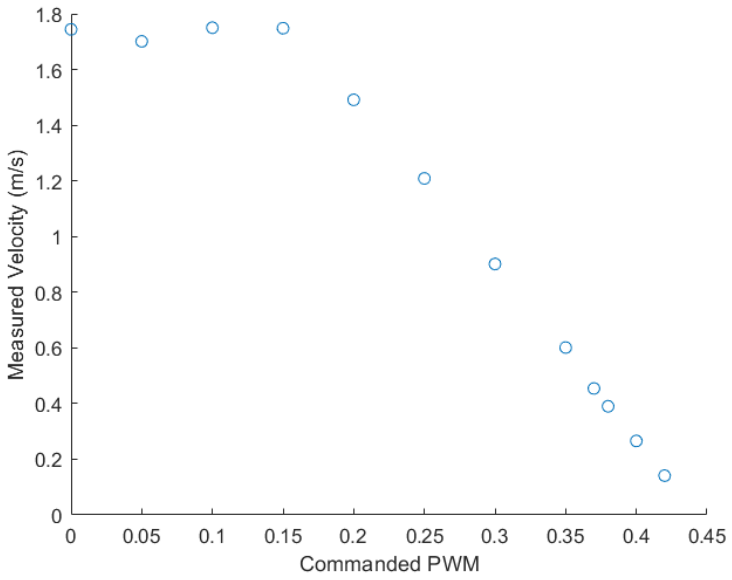
This process involves having each behavior, generally based on one main Sense function, output a brainwave expressing its desire to turn in any given direction or travel at any given velocity. Expressing the waves as many-element vectors allows for a great deal of expression with the brainwave. For example, a relatively indecisive behavior is able to output a very wide wave, indicating that many directions would be acceptable, while another might output a very narrow range of angles with great weight if it feels strongly. The weight of the wave can also be manipulated to, for instance, have obstacle avoidance vote for strongly the closer and obstacle is.



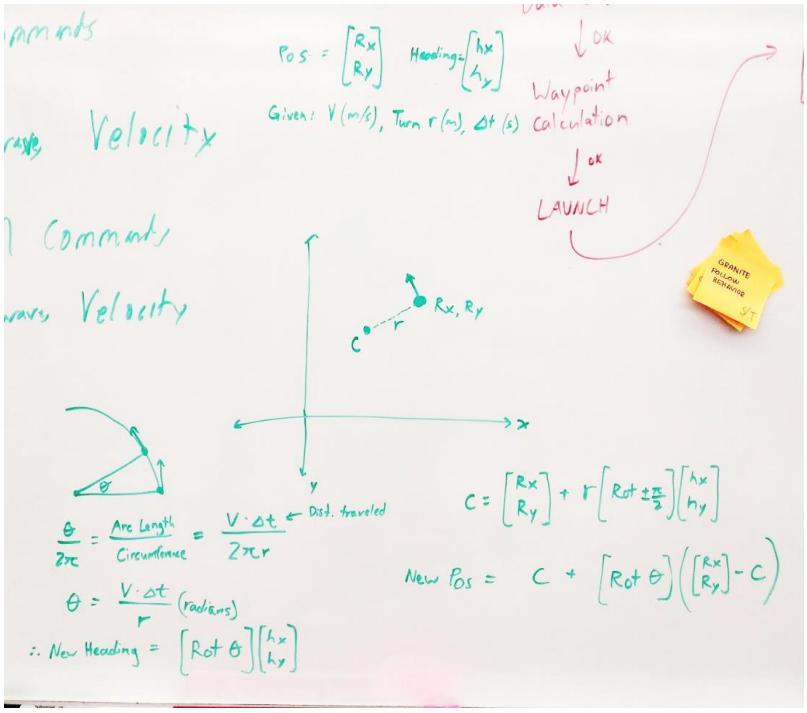
These vote curves are then added up and a single command value is selected as the maximum of the sum. In the example above, the stronger rightwards brainwave largely won out, but the lower and wider leftward wave managed to nudge to commanded heading slightly to the left. Such smoothly-interacting behaviors ensure that the robot is not brittly dependent on a single sensor output.

Act: Motor Calibration & Dead Reckoning

One localization method we hoped to get working reliably in the absence of super useful GPS data was dead-reckoning. This required us to calibrate the PWM signals sent to the drive motor and steering servo to physical velocities and steering radii.



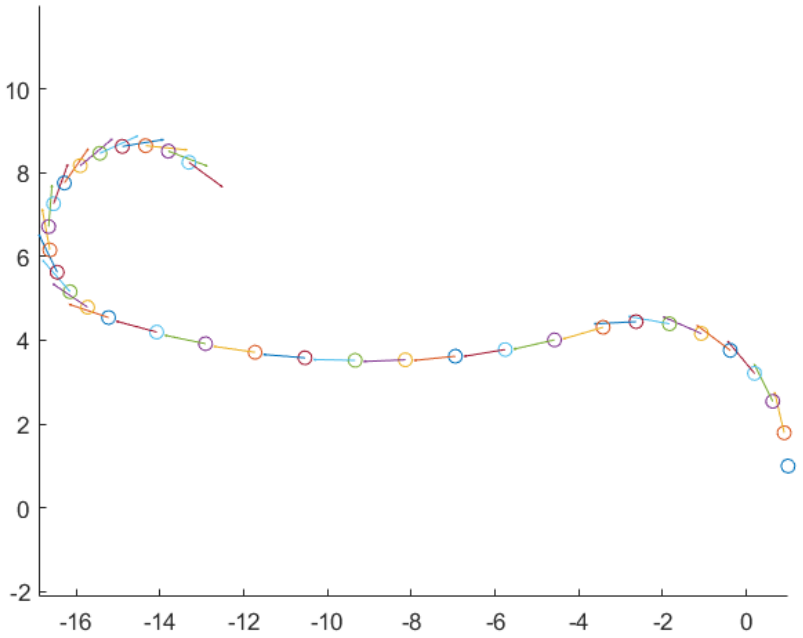
Calibrating the rover's velocity proved quite feasible, even over a short 3-meter test distance. As seen, the measured velocity responds very linearly relative to changes in the commanded PWM duty cycle, up to a point where the rover plateaus at its maximum velocity at around 1.75 m/s.



Given the rover's velocity and turning radius at every control-loop timestep, we wanted to be able to update the rover's new position and heading.

As seen above, we assumed that the robot will travel in a perfectly circular arc at every timestep (which is not a bad assumption, given the mechanics of a steered system like this). Then, in order to find the new position, you simply translate the center point of its instantaneous steering circle to the origin, rotate it around the origin by the angle it will travel given the radius/speed/time, and translate it back.

We wrote a Matlab script to visualize and verify this algorithm. Given a few discrete turns (some sharp and some wide) chained together, this is what we got out.



While the mathematics worked out as seen, transporting this to the real world was not straightforward. For one, errors accumulated quite quickly, especially because it was hard to accurately indicate or reference an initial heading (errors which increase proportional to how far away the rover is from its initial position) and the rover could not instantaneously react to changing commands. Plus, we suspect that the rover's velocity actually varied somewhat with battery charge. Thus, it ended up not being reliable enough to depend on for an accurate picture of our rover's position.

Demo Performance

Overview

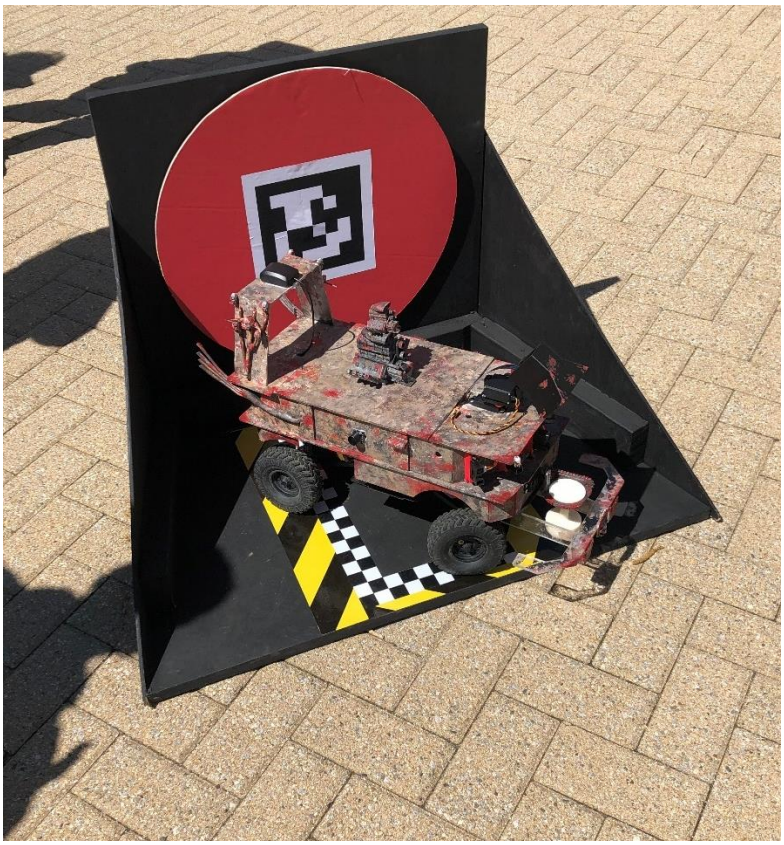
Our robot performed well in demo – well enough to earn us the Best Rover Overall award!! We relied quite heavily on our computer vision code, both for heading localization through brick edges, and April Tag sensing. We also managed to get our main oval traversal code running natively on the Pi via Matlab's Code Generation, which gave our team a substantial performance and reliability advantage.

Sensor inputs

Much of our demo code leaned on my brick edge-detection computer vision code and the heading it desired. For the most part, it did what we expected; it ran a form of bang-bang control where the rover overcorrected its heading slightly between each confident drive interval, so it gently zig-zagged with respect to the bricks' heading as it traversed the oval exterior.

However, we had a bit of a snag with the program failing to detect edges near the initial dock; it could not get a confident reading until the brick tangent had turned by 30 degrees or so. This suggests to me that my edge-detection computer vision function was not suitable to those particular lighting conditions. This makes sense considering it had previously struggled in a shaded region of the oval, suggesting that it is not robust to all scenarios. I imagine that in this case, it likely had something to do with the sun being low in the sky and casting very harsh shadows or eliminating most shadows in one direction and thus making my value-based code detect only the other direction. I mostly developed that code using example images taken later

in the afternoon and with overcast skies. In the future, being more cognizant of and testing for the probable lighting conditions (e.g. sun position) for the demo can help avoid such issues.



Our IR sensing code proved quite fiddly when testing right before the demo, so we chose to significantly limit its influence on the robot's arbitration. It seems like direct harsh sunlight was playing poorly with our infrared readings, which seemed much noisier and less predictable than normal. Again, being more explicit about testing sensors' real-world performance under lots of different conditions would have helped us discover these patterns earlier.

April tag detection, on the other hand, seemed to work just as reliably outdoors and in bright sunlight as it did in the lab. However, the way our code used the outputs of the April tag detection function to inform the robot's movements could have used a good bit more development and was just quite brittle overall – in one instance, the rover was so close to the deposit station that the tag was below the camera's field of view, but it still attempted to drive forward to get closer. Prioritizing these tasks earlier in the development cycle would have paid substantial dividends performance-wise.

Code Performance

Running compiled Matlab code during the demo proved quite effective, at least for the portions of our code that supported it. Our basic traversal around the oval could run entirely compiled, which gave us a big speed advantage over other teams since sensor readings and processing happened practically instantaneously.

However, still needing to maintain a Wi-Fi connection so we could stream the parts of code that needed it – essentially anything involving an April tag – proved somewhat inconvenient in comparison. Had we had the time, figuring out how to run April tag localization natively would be an important step forward, as the experience of running an untethered, fully-self-contained robot proved immeasurably better.

Individual Contributions

For the rover-building portion of the project, I took on the electrical system design, configuration, and validation. I was responsible for essentially the entire electrical planning process and most of the implementation, although I got a bit of soldering help from teammates when necessary, talked through decisions with them, and sought their help when debugging. I made the team's wiring diagram and installed the electrical components in the rover.

With the robot built, I transitioned quite quickly to code mode and focused on computer vision for brick detection. I did all of the testing with detecting the granite squares; then I pivoted to the brick edge-detection approach and I developed and tuned that computer vision code as well.

I also did some of the lower-level calibration tasks; I led the velocity calibration effort and calibrated the sonar sensors. From there, I also spearheaded our dead-reckoning calculations and helped refactor our main loop to allow that to be tested and for the main loop to be more easily built upon. Then, when it was time to wrap things up for demo, I helped the other main programmers (Ally & Nathan) prep the codebase for Code Generation by fixing the many compilation errors :)

All in all, I got involved in a lot of different ways in this project and dove in deep. I'm looking forward to future robot projects and to continuing to grow as a renaissance engineer.