Finding a Sensor

Collaboration is an important part of the process that Tesseract uses to develop code. We find that sharing ideas can be hugely beneficial to both new teams and veterans. This is why we have hosted the FTC Workshop for the third year in a row – to exchange ideas. At the workshop this year, Swerve Robotics showcased their code libraries and demonstrated some of the work that they did with the BNO055 rotation sensor. There is one main thing that differentiates the BNO055 from its competitors: onboard Kalman filtering. The sensor combined readings from a magnetometer, accelerometer, and gyroscope automatically so that the only data you need to read from the device is the rotation. Since the filtering happens at a hardware level, the update rate is extremely good and results in little to no sensor drift. The onboard Kalman filtering also ensures that the sensor's readings aren't impacted by nearby magnetic fields generated by motors. After talking to Swerve's mentors for half an hour after their presentation, we immediately ordered one of these amazing sensors.

Moving Straight

Our team spent about a week experimenting with the sensor. We were able to use Swerve's libraries to interface with the sensor, which made this part of our work easy. Using the sensor's rotation data, **we were able to create a robot that could move quite straight for a very long distance. However, the robot was susceptible to drift and would easily veer off course when an obstacle was placed in its way.** These discoveries led us to consider a Proportional Integral Derivative (PID) control to steer the robot. A PID algorithm looks at the past rotation values, the rate of change in rotation values, and the difference between current and target rotation values. It uses these different numbers to turn back onto course.

PID

We implemented a rudimentary PID control used to make the robot move straight, and the two weeks spent on this achieved great results. **The robot moved straight without drift, and would quickly correct its course even when kicked. This control was done with the Integral and Proportional components of PID.** These controls correct for past error as well as correcting for the current error. However, the derivative component of the algorithm proved harder to implement. We weren't able to look at the rate of change between the last two updates, because the update speed is not consistent. Instead, we took the average rate of change over the last ten updates (approximately 60 milliseconds) and applied some smoothing. After these improvements, **we wrote code that successfully kept the rate of change (derivative) of the robot's rotation at 0.**

Turning

Precise turning of the robot using the BNO055 proved much harder than going straight. **While turning, either the robot would never reach its target rotation or it would oscillate wildly arounds its target rotation.** Our first approach was to write our program so that the robot would always oscillate, and would tell it to stop as soon as it hit its target rotation. However, the robot takes considerable time to stop completely, so this method resulted in **errors upwards of 7 degrees**.

Our next method was far more successful. We wrote our program so that the robot would either turn perfectly or undershoot. Then we added code which checked the standard deviation of the robot's rotation; the higher the standard deviation, the more the robot was moving. **Once the standard deviation became very low, we ended the turn. The results from this method were mixed: the robot always ended its turn promptly but the error could be as high as 5 degrees.**

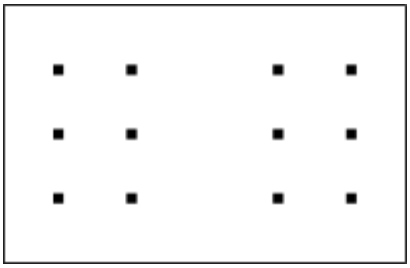
For our last and most successful method of turning, we made use of the derivative of the robot's rotation. We wrote code that made the robot turn 90 degrees per second towards its target rotation. Combined with the rest of the PID algorithm, this forced the robot to never stop early and also prevented oscillation. **It had a maximum error of about three degrees.**

Full Autonomous

The full autonomous makes use of all the code that we wrote to turn the robot and make it move straight. Our robot begins by moving forward with our PID control. It then puts its back brace down, and uses our turning libraries to move 45 degrees to the right. It then deploys a shield to prevent blocks from beginning lodged in its tracks. The robot will try to move straight, stopping when the encoders reach a certain value or if the program detects that the movement is taking longer than it should. This last part is very important; **the robot occasionally overshoots and becomes pressed against the side of the field, so a timeout function allows the program to work even in these situations.** The robot will turn and raise its back brace, and lastly will back up and score the two climbers in the bin. This program was very successful in the second league match: it scored climbers in three of the five matches from both sides of the field.

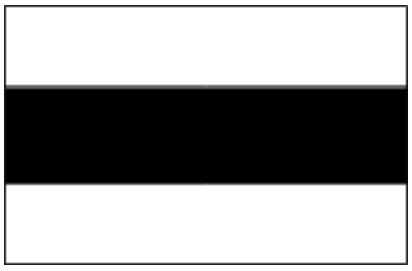
Image Processing

With the wealth of new sensors built into the new Android devices, we are able to take advantage of advanced sensors that were unattainable using the old hardware. One of the most powerful of these sensors is the camera. Being able to take in visual data from the robot's surroundings is tremendously useful. But it is as challenging as it is useful, as writing solid algorithms that can cope with a wide range of uncontrolled variables is very difficult. We decided that the most effective way to use the camera in this year's game was to detect what position the beacon was in. We came across a number of challenges through the process. Although capturing and saving image is a trivial process with Android, the algorithm by which that image is processed can vary greatly. We initially tried an extremely simple algorithm that sampled pixels from the image in the following pattern.

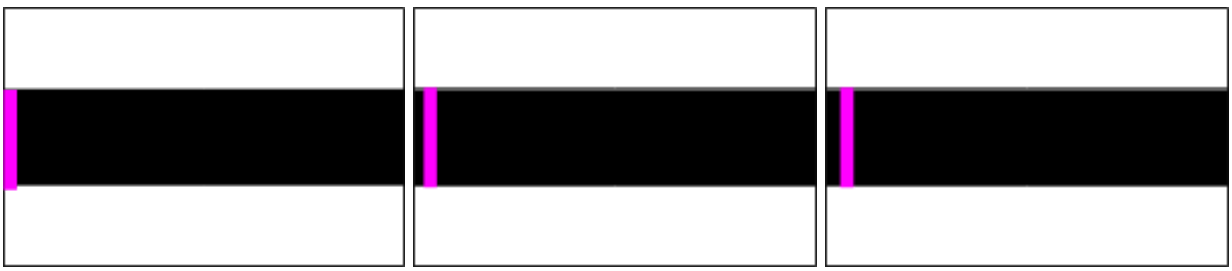


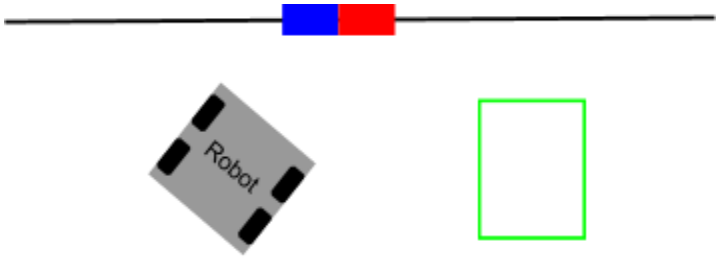
The issue with this approach was that if the beacon was skewed in the image, which happened a lot, the pixels that were determining the color for the “left” side of the beacon may actually have some of the right side of the beacon in its view. Since all the points on the left side of the image were averaged together and compared to the average on the right side, this approach still worked most of the time, but it was not nearly reliable enough to be put in a match, only functioning as intended about 75% of the time.

We then progressed to a much more complicated algorithm, the one we are still using now. It samples a bar across the middle third of the image. The reason we sample only a third is because this way, we know that the entire bar will be on some part of the beacon, as our robot does not have variance up and down, only to the left and right. We can position the phone on our robot so that this middle third is at the perfect height of the beacon, thus eliminating some of the background information at the top and bottom of the image that may have thrown of averages.



Then it essentially takes a box of five pixels wide and the same height as the bar, and moves it across the bar. The box averages each row of pixel to discern the average amount of blue in that row. Then it compares the averages of each row and if the amount of blueness is above a certain threshold and the change in the amount of blue across the row averages is less than a different threshold, it knows that it has hit a solid patch of blue. It then marks stores the place at which this blue patch started and repeats the process, this time looking for red. If it didn’t meet the requirements for a solid blue patch, the box moves one pixel to the right and tries again. After we get the starting values of the red and the blue it is a simple matter to see which is further to the left of the image, thus telling us which position the beacon is in.





After all this experimentation we quickly determined that it could be extremely useful if correctly implemented with solid hardware to back it up. It was an experiment in learning about utilizing the new technology available to us, that if successful could be implemented on the robot. But after weighing the risk and reward of the different scenarios, we decided against using the image processing during the first few competitions until we are confident enough in our hardware to follow through with the decision the image processing makes.

After moving on from state we decided to update our algorithm to improve accuracy even further. We used the same moving average implemented in the algorithm already with the exception of now processing the top third of the image as opposed to the middle third. This is as a result of moving our phone further down on the robot because of wiring changes. These wiring changes were designed to cut down on an electrostatic discharge problem that had a large impact on our robot at state. We also made some change to how the data coming from the moving average is processed. We also smooth our data by lowering its resolution by 1/10th. This is an effective way of removing “noise” (misleading minima and maxima) from the data. With these new changes we have yet to see a failed attempt to pick the color of the beacon as a result of the algorithm. Hardware failures have been a problem but this improved algorithm has been steadfast throughout our testing.

Improving Reliability

After state it became very clear that we needed to improve our reliability. We found that when we had an electrostatic discharge (ESD) event during a match, there was a fairly large chance that the module that failed wasn't critical to the robot's functioning, especially during teleop. However, due to the way that the FTC app was written, a failure of *any* USB device on the robot would create an unhandled Java exception, causing the entire opmode to crash. For example, the Core Device Interface Module is used *only* in autonomous to interface with the BNO055 sensor, but the failure of this module during an ESD event in teleop would cause the entire robot to crash, bringing our teleop to a screeching halt. We wanted to solve this with a combination of two changes: making the app not crash when a non-critical device fails, and making the app wait for several milliseconds before determining that a USB device was unavailable. The latter change was due to the fact that during ESD events, devices would sometimes become unavailable, but would recover once the ESD event was over. However, by then it was too late, and the opmode would have already crashed.

In addition to the large changes described above, we also wanted to make some minor changes to the WiFi handling code in the app. At state we talked to another team's mentor who pointed out a battery-saving setting that we should turn off in order to improve WiFi reliability. We knew that if we made the app acquire a WiFi lock - that is, programmatically force the system to disable WiFi optimizations in a similar way to disabling the setting - we'd see less dropped WiFi Direct connections.

We spent several weeks trying to implement this on our own. Our general approach was to try to decompile the binaries that FTC shipped in the repo. By doing this, we could take app resources that we couldn't modify and turn them back into source code that we could actually make changes to. This process was easy, but the process of integrating our changed files back into the app wasn't so simple. We were decompiling AAR files (which are just ZIP files with a well-known structure), so we reasoned that we should be able to just rezip the files and replace the old version. However, this failed with mysterious build errors. We eventually realized that the Java source files were being byte-compiled before they were written into the AARs, so we tried using the javac program to compile source files ourselves. This also failed: since we were compiling outside of the normal Android environment, the compiler wasn't able to find all the Android SDK resources it expected. We tried adding SDK directories to the compiler's search path, but this proved to be fruitless, so we gave up and moved on to another solution: eschewing recompilation and simply integrating the decompiled source code directly into the app. This too, failed, for reasons we still don't understand.

At this point, we were out of ideas. We decided that the changes we were making, while incredibly important, simply weren't worth the massive amount of time we'd spent on the project. As a last-ditch effort, we decided to upgrade our SDK version in the hopes that it would be easier for us to fix the issues. We reasoned that this wouldn't take too much time, and in the event that it caused issues, we could easily revert it using Git. Luckily, Swerve is fantastic at writing software and provides detailed changelogs for every SDK release, so it was trivial to read through the notes and find out what would break. To our surprise, we also found out that Swerve (or FTC) had fixed virtually every problem we had. They added USB hotswapping, which would allow the app to gracefully deal with USB devices disappearing and reappearing during ESD events, and they added code to acquire a WiFi lock, improving connection reliability in exactly the same way that we wanted to. AJ spent about 20-30 minutes performing the merge and resolving merge conflicts, and then we tested the new code. As expected from the changelogs, nothing broke and we found that reliability was massively improved. It was an unexpectedly pleasing end to what had turned into a seriously frustrating project.

Next Steps

To improve the autonomous program further, we need to gain more precision from the hardware side of the robot. The vast majority of our robot's autonomous failures are as a result of a hardware malfunction or lack of precision. Our climber scoring is much improved and scored two climbers in all but one of our matches at state and the anomalous match we only scored one as a result of an unfortunate bounce. Where our autonomous struggles now is with pressing the beacon button. Our algorithm has yet to fail in our extensive testing after the bugs were worked out, but our mechanism for depressing the button relies on a level of precision in

our robot's drive train that cannot be achieved with our current tread configuration. Currently we are using a small piece of soft rubber attached to the back brace of our robot. We rotate the chassis to align this piece of rubber with the side of the beacon we have determined is correct with the algorithm. This results in a level of reliability much lower than the other parts of our autonomous. To improve this we would like to add a mechanism to our chassis that relies less on the drive train to align with the button.

We would also like to continue work on something that has been fondly dubbed, *The Algorithm*.

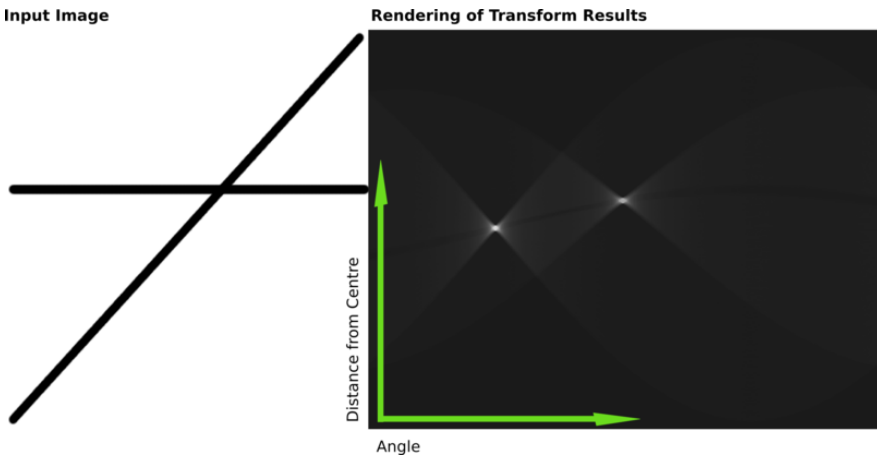
The Algorithm

Since the founding of Redshift Robotics, there have been whispers in the shadows of a mysterious and powerful program dubbed, *The Algorithm*. Some say it's only legend, but we decided to actually build it.

The Algorithm has gone through many stages throughout the years in Redshift Robotics. It is our solution to the problem of spatial awareness in autonomous, one of the hardest challenges in programming an autonomous robot. The first building block of *The Algorithm* was first implemented in our robot for last years game, taking the form of PID, which is explained more in depth earlier in the essay. Last year the PID implementation was designed to allow our robot to move perfectly straight down the ramp in Cascade Effect. It worked extremely well but hadn't implemented accurate turning yet and was only used for a short stretch of forward motion. This year we use PID much more extensively and have developed it into a reliable, easy to implement and extremely functional system. This was the first step in making *The Algorithm*.

Over the years, programmers on 2856 have always talked about implementing an extremely accurate way to position our robot on the field. PID helped, but we still couldn't update our robot's position and make new decisions on how to move part way through the autonomous. Once our robot started moving, it was completely reliant on the IMU to guess its position on the field. However with the new abilities and processing power we gained with the new Android ecosystem, we can now for the first time every work towards completing this system. We have begun implementing the library OpenCV, which natively runs in C and C++. However, through the use of the Android Native Development Kit (NDK) which allows us to run C++ code on the Android devices, we are able to take advantage of the full functionality of OpenCV on the Android phones. OpenCV gives us access to a plethora of extremely useful image processing tools.

We plan to use a combination of image processing and IMU data to determine our robot's position from on the game field using elements that will always be on the field. The element we plan to use are the black metal bars on the top and bottom of the field walls. We use a Hough Transform (a type of processing that detects lines in an image) to find these black lines on the field wall. We can then process based on how far apart the lines are and how the lines converge to determine both our distance and relative rotation to the wall of the field. We use the IMU to tell us which side of the field our camera is pointing towards and then use this Hough Transform to give us a more fine measurement based on the convergence of the lines. The way the lines converge can be determined by extending the lines that have been found on screen out until the inevitably intersect and determining that angle.



This diagram show two lines, similar to how our robot would view the wall of the field. The algorithm after some processing will create an image like the one on the right. The image is a probability cloud with the most white points being probable lines. As you can see there are two points that are predominantly white which means that two lines have been detected, along with their distance from the origin and their angle. With this data we can accurately determine how the camera on our robot is rotated relative to the field of the wall. Right now *The Algorithm* is in its early stages but we are confident that it will be implemented in a usable form by the later competitions of next season.

This is by far the most complex algorithm (or more accurately set of algorithms) that we at Redshift Robotics have ever undertaken and we are thriving on the challenge. We can only hope is that we're not building SkyNet.