Finding a Sensor

Collaboration is an important part of the process that Teseract 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. After talking to Swerve's mentors for half an hour after their presentation, we immediately ordered one of these amazing sensors. 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 or no sensor drift. The onboard Kalman filtering also ensured that the sensor's readings aren't impacted by nearby magnetic fields generated by motors.

Moving Straight

Our team spent about a week understanding the sensor. We used Swerve's libraries to read the rotation from 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, our creation 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. PID looks at the past rotation values, the rate of change in rotation values, and the difference between current and target rotation values. It uses there different numbers to turn back onto course.

PID

Our team spent two weeks implementing a rudimentary PID control used to make the robot move straight, and achieved miraculous 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 the control.** These kinds of control 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 preformed 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, the robot would never reach its target rotation or 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.

Next Steps

To improve the autonomous program further, we need to gain more precision in the final scoring operation. Both of the program's failures in the league matches were a result of either poor robot alignment or failed deployment. Building a more robust climber scoring mechanism is a high priority. Additionally, some sort of mechanism for removing blocks behind the robot would be extremely useful. Debris frequently prevents the robot from backing up properly and causes it to miss scoring the climbers in the bucket.