ChimiSwerve White Paper

Everything You Wanted to Know About Team 1684's Swerve Code
(Except the Code) AND MORE!

Revision: 2020.8.17

0.0 About This White Paper
- 0.1 Swerve IS HARD
- 0.2 We're Not THE EXPERTS... Just Experienced People
- 0.3 Don't Be Afraid To Ask For Help!
- 0.4 Where's Your Code, Bruh?
- 0.5 This Is A Living Document

1.0 Hardware/Electrical Info
- 1.1 About Our Swerve Modules
- 1.2 About Our Gyros
- 1.3 About Our Motor Controllers
- 1.4 Other Sensors

2.0 Software Info
- 2.1 Swerve in Teleop
  - 2.1a How It Should Work
  - 2.1b The Math
    - Controls and Inputs
    - Field Centric Transformation
    - Inverse Kinematics (Wheel Speeds and Azimuths)
        - Intuition
    - Power to the ground (Speed and Azimuth drivers)
      - Azimuth
        - Inversion Awareness
- 2.2 Swerve in Auton
  - 2.2a How It Should Work
  - 2.2b The Math Behind the Magic
      - Forward Kinematics (FK)
      - Odometry
      - Calibration
- 2.3 Integrating Other PIDs into Swerve
  - 2.3a PIDs (PIDFs) Loops: A Primer
    - Why Do We Even Need Fancy Control Loops?
    - What Do These Letters Even Mean!?
      - P - Proportion
      - I - Integral
      - D - Derivative

F - Feed Forward
Other Control Loop Terms
Open Loops
Closed Loops
Internal Loops (our version)
External Loops (our version)
Positioning Loops
Velocity Loops
“Inside Outside” Loops (our definition)
General Tips for Tuning Control Loops
Start with kP
Next is kD
Then kI
What about kF?
2.3b Goal Centric (Starring the Limelight)
Vision Targeting
Essentials
Easy Mode
Advanced Mode
Hard Mode
Expert Mode
Goal-Centric
Introduction
How It’s Done
2.3c Game Piece Centric (Starring the Pixy)
Introduction
How It’s Done
2.3d Maintaining Your Heading (aka Not Spinning… aka Driving “Straight”) While in Field Centric
Introduction
How It’s Done
3.0 Other Misc. Tips & Tricks
3.1 “Zero” Azimuth Position & Pre-Match Module Testing
3.2 Protect Your Reset Buttons!
3.3 Document Your CANbus Network Topology
3.4 Know Your Electronics

3.5 Invest in Good Controllers… And You Should Interpolate
3.6 Seriously, Implement Brownout Protection!
3.7 Handling Gyro Fault Conditions
THE END… or is it…?

0.0 About This White Paper

0.1 Swerve IS HARD

We're not joking. ***SWERVE IS HARD!*** Please do not attempt to do swerve if your team does not have:

1) lots of mechanical knowledge
2) lots of programming experience
3) substantial funding for a crazy expensive drivetrain
4) ridiculous persistence and dedication

We DON'T want to deter you from achieving your dream machine. *We DO want to limit the number of broken-down robots on the field.* As our drive coach Jon Uren says, *"Build within your means."* If your team doesn't meet all the requirements listed above, please reconsider attempting to do swerve. Poorly implemented swerve (in both a programming and mechanical sense) will do your team a major disservice. Sure, that robot might look cool or sound epic. *But, does it work!?* Most of the time, the simpler the better.

0.2 We're Not THE EXPERTS... Just Experienced People

If your team decides to go down this path, please pay close attention to the advice in this whitepaper. We have lived through the joys and sorrows of swerve. Please note that we are not claiming to be THE EXPERTS at swerve. Regardless, we are *just experienced people*. We just want to share everything we've learned since day 1 of the 2020 season. Like everyone, *we are still learning from our mistakes*. We will continue to iterate our designs so our robot can be its absolute best. We urge you all to do the same.

By the way, this is our 2020 reveal video. Please watch so you know exactly what we're talking about.

0.3 Don't Be Afraid To Ask For Help!

We all know that time on the robot is limited. Programmers always wish for more time on the finished robot. Time is the most valuable build season commodity. Before something does not go as planned, it should be part of your programming strategy to know when to reach out for help. Help can mean looking for sample code, asking on a forum if someone has had a similar issue, or even reaching out to other teams to review your code. Remember that other FRC programmers will likely be in the same semi-frantic state that you are trying to figure out the one thing that isn't working. Someone outside of your programming team may not have the time to help you. Asking for another team's time during build season is a BIG ASK!

We were honored and blessed to collaborate with the EngiNERDs on the swerve drive design. That gave us two programming teams working on the exact same problems with the same equipment. We were able to work out some of the kinks together.

Some day, you may be the one being asked for assistance. It is an honor to be worthy of being asked to help. Give whatever assistance that you can, but your responsibility will still be getting your own robot ready to go.

0.4 Where's Your Code, Bruh?

Due to the way we structured our 2020 code, we feel we cannot release it for general use. It was not designed to be used as a library for other robots. We have a very different approach to the way we write code. In the future, we hope to improve upon this. We strongly feel releasing our 2020 code wouldn't be helpful to others.

That is why we have dedicated so much time into creating this document. We still wanted to share our knowledge and experiences with others, just in a format that would actually be helpful. We have included some code snippets to help enhance your understanding.

Hopefully, you learn something from this document. We certainly did.

0.5 This Is A Living Document

We intend to revise this document as we acquire new swerve knowledge. Additionally, we will modify this document as needed to help add clarification and answer commonly asked questions.

We want your feedback! Please let us know how we can improve this whitepaper!

1.0 Hardware/Electrical Info

1.1 About Our Swerve Modules

We are using the MK2 off-the-shelf swerve modules from Swerve Drive Specialties. We made some modifications, but they are mainly as manufactured. We added a custom made laser cut gear to collect encoder readings from our absolute encoders on the turning gear. We are using 4" DT Versawheels from Vexpro. The wheels wear rapidly on the actual field, however, the tread gives us easy indicators of change in diameter as the diamond pattern wears down. We are using the US Digital MA3 encoders, which are absolute analog encoders.

1.2 About Our Gyros

We use two gyros on our robot; The Gadgeteer Pigeon IMU from Cross the Road Electronics (CTRE) and navX-MXP (navBoard) from Kauai Labs.

We like each one for a specific reason. The navX plugs right onto the Roborio, so mounting is simple. We have used this sensor for many years, so we are comfortable with it. The Pigeon is small and can be attached directly to a Talon SRX. They perform many of the same functions with complete libraries.

1.3 About Our Motor Controllers

Our choice for motor controllers for the swerve modules was the Falcon 500 with the integrated brushless motor by VEX Robotics and the Talon FX controller by CTRE. The reason for using this as the drive motor was simple, power and control. The integrated encoder is adequate for counting the distance travelled using the velocity mode to get the most consistent results regardless of voltage differences (within reason.) We use a positioning loop that runs in the Roborio that sends a corresponding velocity command to the Falcon every 50 hertz cycle.

We were willing to try the unproven Falcon this year because we had such success last season with the Neo brushless motors, and the motor performance specifications listed for the Falcon indicated that it would outperform the Neo. We had great success with the Talon SRX by CTRE during many previous seasons. We expected the same kind of performance from the FX. The build season was not without pain when it came to using the Falcon. Out of 20 motors that we had purchased, we found that 2 had problems related to the motor controller prior to competition. We replaced the 2 motors after *12 hours of troubleshooting*.

The turning motor is also a Falcon 500. This choice was primarily for packaging. The integrated motor controller used less space, so we did not have to find another location for the controller. It is also aesthetically pleasing and symmetrical to have both motors side by side and identical. We used a loop that ran inside the Roborio with a basic positioning loop and limited output.

Another consideration that is worth mentioning is a collaboration between our team and the EngiNERDs (Team 2337). We used the same modules, motors, and encoders. That allowed us to split up the development a bit and some of the modifications. We may not have tried something this bold without their collaboration.

1.4 Other Sensors

We use several other sensors that weave into our swerve code. The most obvious is the *Limelight2+*. We also use two *Pixy2s* and a *Lidar Lite3*. We use a *Beetle microcontroller* to process the data from the *Lidar* so that we do not bog down the *Roborio*. We will address the exact usage of these sensors in later sections.

The *Limelight* is used to visually target the goal. We use the x and y offsets that are in the *Limelight* to turn the shooter to the goal and gauge the distance from the goal. The *Limelight* has excellent range and is able to target the goal from anywhere on the field that has a shooting solution. The *Limelight* is mounted next to our shooter. It would be ideal to have it in line with the shooter, but space did not allow it. Because it is not exactly in the same vertical plane, we have to compensate in code for the offset.

The *Pixy2* is also a visual targeting camera and processor, but we were only able to get one digital and one analog signal to the *Roborio*. The digital signal tells us if there is a ball within view of the camera. The analog signal tells us the position of the ball on one axis of the camera's view. The effective range is between 6 and 10 feet dependent on lighting. The *Pixy2* is programmed directly with a laptop to "pick out" specific targets by color, and it is programmed to output only those two signals described above. We have a *Pixy2* mounted on the center of each of our intakes to assist with ball pickup.

The *Lidar Lite3* is a laser rangefinder that we have pointing forward from the front of our robot. We process the returns from the lidar with the *Beetle*. We were able to get the distance from the lidar to the *Beetle* at a speed of at least 33 hertz (depending on the distance). The *Roborio* then can get that measurement at 50 hertz.

2.0 Software Info

New in the 2020 season, WPILib has some swerve classes available. We did not use these classes. We do not know how well they work when they are properly implemented. If you decide to try using these classes, your results may vary.

There was a brief period of time that we attempted to use one of these classes, but we quickly opted to write our own. For the purposes of future debugging, we wanted to understand every bit of our code.

Throughout our adventures down the path of swerve, we discovered many useful resources. One of the most important things we found was this Chief Delphi post.
We used *SwerveTester_8.xls* to understand how each wheel reacts with different inputs and to verify our teleop swerve control was working properly.

2.1 Swerve in Teleop

2.1a How It *Should* Work

You move the controller and the robot magically does what you think it should. Isn't that the goal? Of course it is! The way that we chose is the same way that our driver would operate a drone. Although, we have not worked out the levitate functionality.

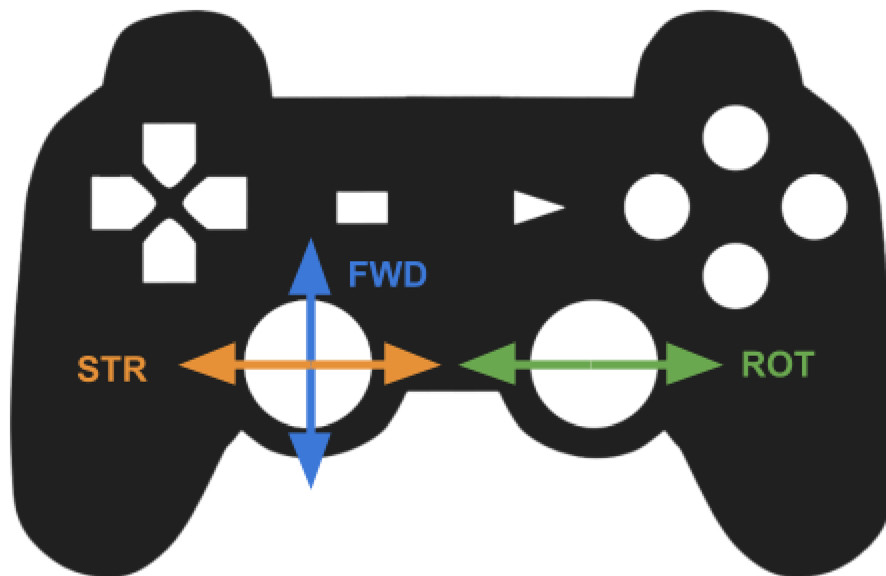
The robot should be free to move in any direction on the single plane of the field. It should be able to do that at the same speed and torque.

Given the extra freedom that swerve gives the robot, both literally and in terms of holonomic movement (can freely move in any direction, ideally with no preferred direction), operating in a field-centric manner is immensely helpful. We consider field-centric motion essential to utilizing swerve effectively at a competition level.

2.1b The *Math*

Controls and Inputs

The driver has some intended output that they'd like the robot to do. In order to communicate this, we define three command inputs - Forward, Strafe, and Rotation, or FWD, STR, and ROT. Let's define these as variables, each going from -1 to 1, with 0 indicating no control input. There are many ways to map driver inputs to these commands via joysticks, but we chose to use a method where one stick controls translation rate (FWD and STR) and the other's left-right axis controls rotation rate (ROT), as shown on the controller. These joysticks are implemented using standard libraries in WPILib, and input is [filtered and transformed](#) to provide a better experience to the drivers.



In addition to the driver's inputs, we can have inputs incorporated.

Field Centric Transformation

Once we know the intended movement that the robot should be making, we should apply our field-centric math. This is what's known as a *transformation*. In simple terms, we take what we know about the robot's *orientation* relative to the field - in this case, its angle (as measured by a gyroscope) and our known desired *forward/back* and *strafe* commands, and adjust them so that the robot moves relative to the field.

We define FWD and STR as the desired *forward/back* and *strafe* commands from the joystick, relative to the field. These can both range from -1 to 1 from the joystick, assuming things are filtered correctly, and the magnitude of the input vector (FWD, STR) shouldn't ever be greater than 1. We'll find that this doesn't matter later.

We'll call the angle of the chassis relative to the field θ , and assume that it's 0 when the front of the robot is aligned with the field, away from the driver. This angle can be either defined from -180° to 180°, or from 0 to 360 degrees - the math in this case works the same. With these in mind, we can transform our FWD and STR commands as follows:

$$FWD_{new} = FWD * cos(\theta) + STR * sin(\theta)$$
$$STR_{new} = STR * cos(\theta) - FWD * sin(\theta)$$

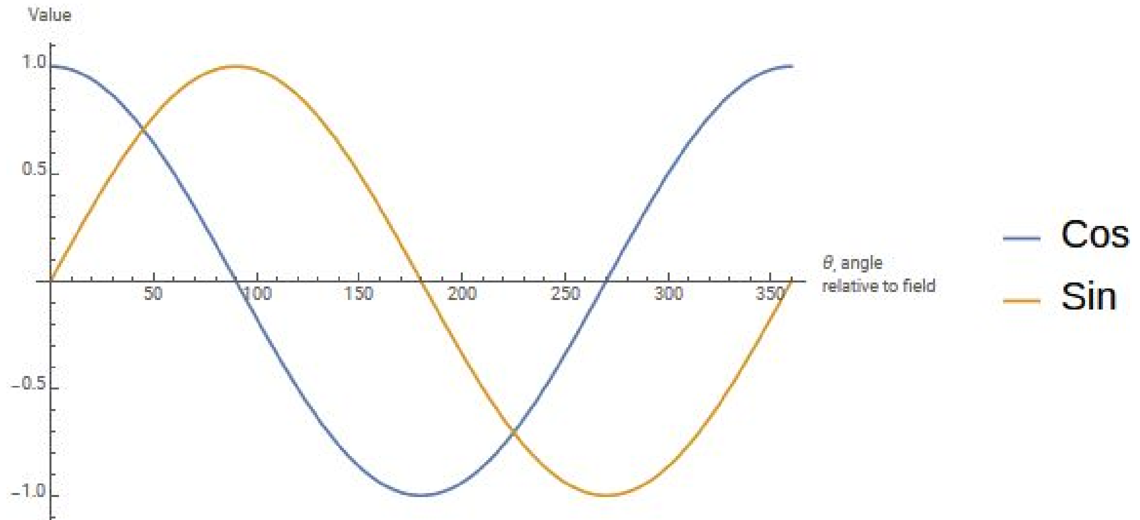
First, we calculate our new FWD, which can be thought of as simply figuring out how much of each original command (FWD and STR) are in the actual direction of the robot's "FWD". Likewise, we can do the same for our STR command. This effectively transforms our commands, *considered relative to the field*, into commands that are *relative to the robot*, using information about the robot's orientation to the field to align everything. If you know a bit of linear algebra, you might recognize this as the result of a *rotation matrix*, applied to the vector (FWD, STR).

Our ROT (rotation) command actually doesn't come into play at all in this transformation. To understand why this is, consider that

To explain this further, let's look at the results of the the following three situations:

Example 1: Robot is aligned with field

The key here is to think about what the value of θ is, and how the value of our sines and cosines changes as a result. Here, since we're aligned with the field, θ =0°. Let's check our sine and cosine plots to see where that puts us at, in terms of values:



cos(0°) is 1, and sin(0°) is 0. If we think about a right triangle, with one of its angles being 0° and a hypotenuse of 1, the opposing side of that triangle would have no height - and the adjacent side would have the length of the hypotenuse, or 1. Let's see what our math does with this.

$$FWD_{new} = FWD * cos(0°) + STR * sin(0°) = FWD * 1 + STR * 0 = FWD$$
$$STR_{new} = STR * cos(0°) - FWD * sin(0°) = STR * 1 - FWD * 0 = STR$$

Our math tells us that we haven't changed anything by doing this! Physically, this makes sense; if the robot is aligned with the field, what it considers to be forward is the same as the field's forward; we don't need to modify our commands at all.

Example 2: Robot is at 90 degrees to field

Let's say that we're now at a right angle to the field; our robot is sideways, so our angle θ =90°. Again, let's check our trigonometry. cos(90°) = 0, and sin(90°) = 1. What does our math do with these?

$$FWD_{new} = FWD * cos(90°) + STR * sin(90°) = FWD * 0 + STR * 1 = STR$$
$$STR_{new} = STR * cos(90°) - FWD * sin(90°) = STR * 0 - FWD * 1 = -FWD$$

Our commands have switched (with a small change of sign)! Again, physically, this makes some sense. The robot is sideways; in order to go forward, it has to do what it considers a strafe, and in order to strafe, it has to go what it considers forward.

Example 3: Robot is at 30 degrees to field

We're now at some angle that doesn't produce as nice a result. The robot is aligned differently from the field, so to go straight forward or straight sideways, the robot is going to need to do some

combination of what it considers going forward and going sideways. With our angle θ =30°, cos(30°) = 0.866, and sin(30°) = 0.5. Following through the equations, then,

$$FWD_{new} = FWD * cos(30°) + STR * sin(30°) = FWD * 0.866 + STR * 0.5$$
$$STR_{new} = STR * cos(30°) - FWD * sin(30°) = STR * 0.866 - FWD * 0.5$$

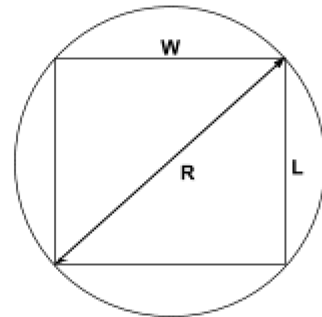
Both FWD and STR contribute to each of the transformed values. This pattern will carry through for any non-right angle, and as long as we consider our angle properly (with clockwise as positive) we're good to go!

Inverse Kinematics (Wheel Speeds and Azimuths)

Now that we have our FWD and STR commands transformed, we need to figure out what each wheel should be doing to execute them. What we're doing here is commonly referred to as *inverse kinematics*, or IK. We have some goal output we want to get to, and we have some actuator parameters (or joints) we can adjust, those being our wheel speeds and azimuth angles. Inverse kinematics is the calculation of a unique set of output settings that will give us our overall output. Forward kinematics (which we'll cover elsewhere) is the opposite - determining the state (or *pose)* we're in based on what our actuators have been doing.

First, we should account for the wheel layout. We're going to assume our wheel pods are on the vertices of a rectangle. To put it another way, all four wheel axles are able to lie on some circle. This represents the majority of swerve drive setups (ours was square). If the swerve modules are not on a square (e.g. if the robot is not square), we'll account for that as we make our calculations.

First, let's define the length and width of our wheelbase as L and W, and let's define an additional value R, as $R = \sqrt{L^2 + W^2}$, or the diameter of the circle that contacts all four wheel axles. The units of these values don't matter, as we're only going to be taking their ratios in our wheel calculations.

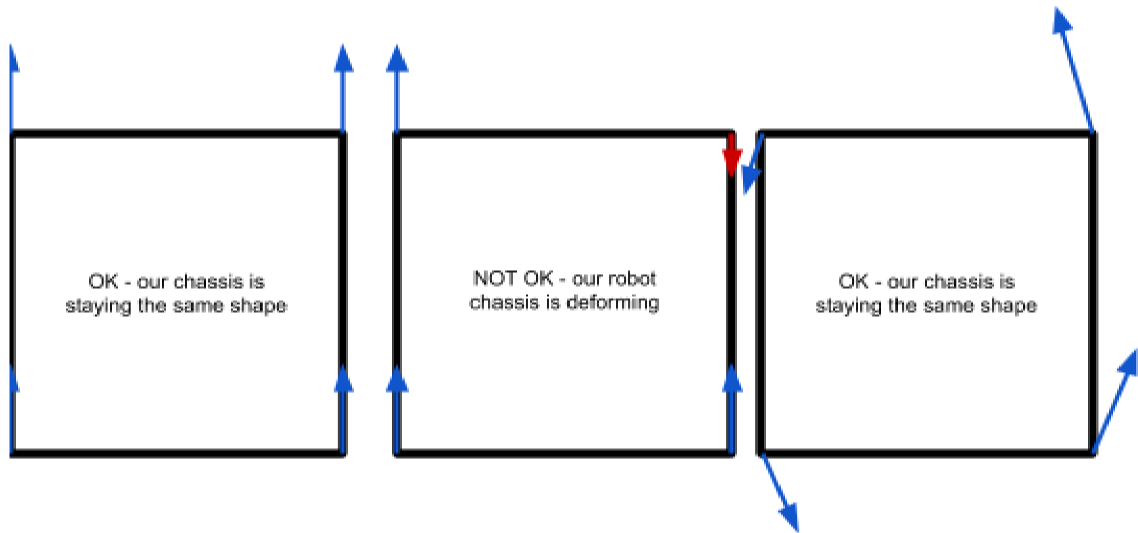


Intuition

To understand how we can turn our overall commands (what we want the robot as a whole to do) into what each wheel needs to do, we should build some intuition. It's perfectly possible to borrow the math and program it into place without this, but it really helps to understand how things work under the hood.

As we derive our inverse kinematics, we make some underlying assumptions, including that the robot's chassis acts as a *rigid body* - within its own reference frame, one point cannot move closer or further away to another point. Imagine a metal cube on a desk. You can move it forward, back, and spin it, but it's solid and unchanging relative to itself. From a kinematics standpoint, there's an important fact we can draw from this. If we look at one side of the robot, all points along it have to

have the same velocity forward or backward. If they had different speeds, the side would be changing in length. Hopefully that's not actually happening, and if our robot is built sufficiently rigidly, it shouldn't be.



We can back this assumption out by doing some *vector math*. ChiefDelphi user Ether's "Derivation of Inverse Kinematics for Swerve" document explains this in detail. To do this, we can combine our desired strafe and speed commands into a *translation vector*, and our rotation command can be combined with our robot geometry to form a rotational component. By adding these vectors, we get our required wheel movement vector, in the form of a direction and a magnitude.

From here, we'll define some variables to save us some work. Standard convention has them as A, B, C, and D.

$$A = STR - ROT * \frac{L}{R}$$
$$B = STR + ROT * \frac{L}{R}$$
$$C = FWD - ROT * \frac{W}{R}$$
$$D = FWD + ROT * \frac{W}{R}$$

We'll use these to calculate our resultant wheel speeds and wheel angles (or azimuth angles), or *ws* and *wa* respectively.

$$ws_{FR} = \sqrt{B^2 + C^2} \qquad wa_{FR} = atan2(B, C)$$ [1]
$$ws_{FL} = \sqrt{B^2 + D^2} \qquad wa_{FL} = atan2(B, D)$$
$$ws_{RR} = \sqrt{A^2 + C^2} \qquad wa_{RR} = atan2(A, C)$$
$$ws_{RL} = \sqrt{A^2 + D^2} \qquad wa_{RL} = atan2(D, D)$$

What we see here is that we have common factors between our wheels. For both front wheels, we have a common horizontal factor B, and for both rear wheels, a different common factor A. For both

[1] *The atan2 function is defined in many programming languages, and provides quadrant-aware calculations for the arctangent function, meaning the angle output will range from -π to π, instead of 0 to π/2.*

right wheels, and left, we see the same thing, with common forward/backward components C and D. This is the mathematical realization of the fact that our robot's chassis can't change in size or shape. Bonus: if you know a bit of vector math, you'll notice that we're doing a transformation from cartesian to polar coordinates.

Where FR, FL, RR, and RL refer to front right, front left, rear right, and rear left wheels. Our azimuth angles should range from $-\pi$ to π, with positive as clockwise and zero being straight ahead. These can be converted to degrees as necessary - just multiply by $180/\pi$. Our wheel speeds should range from 0 to 1, absolute, but we'll need to check if they need to be normalized. To do this, we just check if the maximum of our *ws* values is greater than 1, and if it is, scale the values such that it's 1.

$$ws_{max} = max(ws_{FR}, ws_{FL}.ws_{RR},\ ws_{RL})$$
$$if\ ws_{max} > 1.0 :$$
$$ws_{FR} = ws_{FR}/ws_{max}$$
$$ws_{FL} = ws_{FL}/ws_{max}$$
$$ws_{RR} = ws_{RR}/ws_{max}$$
$$ws_{RL} = ws_{RL}/ws_{max}$$

Now we have all *ws* values ranging from 0 to 1. The wheel speed assumed to be in the direction the wheel is facing (*wa*), and thus does not go negative, as that would imply the wheel has turned 180°.

Our algorithm to this point brings up a problem. If our control input quickly changes from, for example, moving entirely forward to moving backward, our wheel pods need to turn all the way around in order to execute the new command. Given that we're working with motors that can go full power forward or backward, that's an inefficient thing to do. Ideally, we'd just flip the motor into reverse and move on.

In order to solve this problem, we implement what we call *inversion awareness*. If we assume we know the wheel's current azimuth angle, and we've calculated the azimuth we need, we can check if we'd need to "flip" our module or not. If we would, we can just invert the speed output, readjust where our azimuth is headed, and continue on our way. We'll implement this in a bit.

Power to the ground (Speed and Azimuth drivers)

Finally, we're at the point where we can get things moving. We have the parameters we want, a set of wheel speeds and angles, generated from our desired output, transformed to be oriented relative to the field. Now we want to get our swerve modules to execute those commands. These next bits of code can be considered analogous to drivers in a computer system - layers of code that handle the nitty-gritty low-level communication and control while providing an abstracted interface for control. In our case, we grouped azimuth and wheel speed into one module that we can use to set a wheel's target state.

Azimuth

Azimuth control is accomplished using a proportional feedback controller sensing the angle of the wheel with an absolute encoder. Depending on the specific encoder used, this may come in the form of a degree value, a voltage, or an integer number of ticks. For our US Digital MA3 encoders, the output is in the form of a 0-5V signal, with the full range representing 360° of rotation. In many cases, the encoder housing cannot repeatably and precisely be oriented to the "correct" physical position. This means that the reading of our encoder when our azimuth is straight forward (0°, the wheel is straight) can be *any* value. As such, we're going to have some offset value on a per-module basis, and we'll factor it into our calculations.

For our feedback controller, we need to calculate the error between our setpoint and position. Because we're working with angles and have a range of 0-360°, we need to use the *remainder* function to make sure our error is calculated properly. We'll also use this for incorporating our offset. Assuming we have the angle we want for this wheel (wa), we can calculate our error:

$$encoder_w = Encoder.GetValue()$$
$$azimuthAngle = remainder(Encoder.GetValue() - wheelAngleOffset)$$
$$azimuthError = remainder(azimuthAngle - wa)$$

Inversion Awareness

Using the Talon FX's *SetInverted* method makes implementing inversion control very straightforward. We simply "flip" our azimuth error to the other side.

$$azimuthError = azimuthPosition - wa\,;$$
$$if\ abs(azimuthError)\ >\ 90:\ //assuming\ our\ angles\ are\ in\ degrees$$
$$\quad azimuthError = azimuthError - 180 * sgn(azimuthError)$$
$$\quad SpeedMotorController.SetInverted(true)$$
$$else:$$
$$\quad SpeedMotorController.SetInverted(false)$$

2.2 Swerve in Auton

2.2a How It *Should* Work

We want to tell the robot to go to a coordinate on the field and face a specific direction when it gets there. Sometimes we want the rotation to happen at different places along the route. Sometimes

we want it to rotate first or last. Sometimes we even want the robot to travel around an obstacle. The robot should switch modes as desired along the overall route to achieve the game objectives. We used a case generator to give us each motion in the desired order and change states primarily by sensors. The end result should be a robot that travels at the maximum controllable speed while achieving the desired positions with maximum accuracy.

2.2b The Math Behind the *Magic*

One of the core differences when operating in autonomous mode is that the robot can't receive information about where it is from the driver, in the form of controls. Early on, we established that we want to accurately and repeatably control our movements and actions. To do this, the robot needs to know where it is at all times - or at least have a good guess. This is known as odometry within the field of robotics. Without accurate odometry, we're forced to use dead reckoning, or worse, time-based movements at approximate speeds.

In order to perform our odometry, we decided to use our wheel encoders combined with a gyroscope. By performing the reverse of what we do in teleop, which is taking an intended output *pose* and figuring out what each individual actuator should be doing to achieve that pose. Again, a *pose* is simply a set of unique characteristics describing the robot's position or motion at an instant in time. By taking information about the velocity and direction of each wheel and combining it using the kinematics we described in section 2.1, we can obtain an overall velocity of the robot in two directions; this is referred to as Forward Kinematics, or FK. By integrating this over time (just multiplying the velocity at this time with the time step of the controller), we can get our position.

Forward Kinematics (FK)

Of course, going from wheel encoders to overall velocity is not entirely straightforward. The problem comes when we look at our information and desired results: we have eight variables, four wheel speeds and four directions, and we want three outputs: the robot's forward, sideways, and rotational velocity. If you're familiar with algebra, especially linear algebra, you might recognize that this makes our system overdefined. We have more information than we need, and can't obtain an exact analytic solution. There's a couple ways to solve this. By setting up the exact equations and putting them into a computer algebra system, or by assembling the equations in a matrix-vector format and using a linear equation solver, we can obtain a "best-fit" to the system.

However, this sort of inexact fitting can be difficult to implement, often requiring inclusion of extra libraries. Due to the time sensitivity of the season, we chose to take a simplified approach. Courtesy of Kyle Lanman of team 2841, we adapted an algorithm that averages variables until we get from eight down to the three we need. While we expect this to be less accurate than more advanced methods, we found it to be remarkably accurate after calibration. As long as wheel speeds and

acceleration are kept below the point where they'd slip under normal conditions, this proves to be a suitable odometry method for the limitations of the 15 second autonomous period, especially when combined with other sensors to "close the loop" on navigation.

In this formulation of forward kinematics, we start from a position where we assume we know our wheel speeds and wheel angles; call them ws_{FR}, ws_{FL}, wa_{RL}, and so on, with F/R and L/R again representing front/rear and left/right. Using motors that have a built in encoder, such as CTRE Falcon 500s or REV Neos, is advantageous for determining wheel speed.

We do need to make sure we've consistent, physically meaningful values for our wheel speeds. Whether we're using encoders built into the motor or ones installed manually, they're likely putting out some counter of ticks, or count of revolutions, or some speed of ticks/second or revolutions/second. We'll need to use this in combination with our drivetrain reduction and wheel diameter to get some conversion rate, and thus be able to get our wheel speeds in units of distance per time, like in/sec, ft/sec, or m/sec.

We can first calculate A, B, C, and D values from our wheelspeeds and angles, but this time, we're going to calculate them for each wheel. These will be in units of velocity (distance per me), as we're simply multiplying our wheel speed (which has those units) by an angle component.

$$B_{FL} = sin(wa_{FL}) * ws_{FL} \qquad D_{FL} = cos(wa_{FL}) * ws_{FL}$$
$$B_{FR} = sin(wa_{FR}) * ws_{FR} \qquad C_{FR} = cos(wa_{FR}) * ws_{FR}$$
$$A_{RL} = sin(wa_{RL}) * ws_{RL} \qquad D_{RL} = cos(wa_{RL}) * ws_{RL}$$
$$A_{RR} = sin(wa_{RR}) * ws_{RR} \qquad C_{RR} = cos(wa_{FR}) * ws_{RR}$$

Well that wasn't much help with simplifying our variables! In order to make this more reasonable, we're going to do some averaging. For each value of A, B, C, and D, we're going to take the two values we have, and average them. These averaged values will still have units of velocity, as they're just averaging two other velocity values.

$$A = (A_{RR} + A_{RL})/2$$
$$B = (B_{FL} + A_{FR})/2$$
$$C = (C_{FR} + C_{RR})/2$$
$$D = (D_{FL} + D_{RL})/2$$

Excellent, that's brought our complexity down some. Now we need to find our rotational velocity, or ROT. For this, we need physically accurate measurements of what we earlier defined as the length and width of our wheelbase, L and W. Then, we'll calculate the possible rotational velocity from our knowledge about the front/back (A/B) and left/right (C/D) velocities, and average those again to get a single value. Note that it's also possible to get this ROT value (which should be in radians/second) from a gyroscope, which we should already have on our robot so we can drive field-centric.

$$ROT_1 = (B - A)/L$$
$$ROT_2 = (C - D)/W$$
$$ROT = (ROT_1 + ROT_2)/2$$

From here, we're simply going to incorporate this with our geometry and A, B, C, D values to obtain (again) two values each for forward and strafe velocities, and average them to get our final estimates of forward and strafe speed.

$$FWD_1 = ROT * (L/2) + A$$
$$FWD_2 = -ROT * (L/2) + B$$
$$FWD = (FWD_1 + FWD_2)/2$$

$$STR_1 = ROT * (W/2) + C$$
$$STR_2 = -ROT * (W/2) + D$$
$$STR = (STR_1 + STR_2)/2$$

Great! Now we've got something to work with. If we want our distance values to be *useful*, though, these velocities should be transformed so they're field-centric. Back to our trusty field centric transformation, we'll again need our angle relative to the field, usually provided by a gyroscope.

$$FWD_{new} = FWD * cos(\theta) + STR * sin(\theta)$$
$$STR_{new} = STR * cos(\theta) - FWD * sin(\theta)$$

Odometry

From there, we can now figure out how fast we're actually going along the field - pretty nifty! To get to a position, we just need to *integrate* these over time. This is as simple as initializing a timer and comparing its value at the current loop run to its value in the previous run to determine our *timestep*. For most robot code, this timestep is somewhere around 0.020 seconds, or 20 milliseconds; however, this is only a nominal value, and it can vary up and down (mostly up) depending on the behavior of the robot's code. In any case, we can take this timestep and our speed and integrate it into an accumulator value to get our position relative to where we started counting.

$$timeStep = LoopTimer.Get() - lastTime$$
$$positionAlongField = positionAlongField + FWD * timeStep$$
$$positionAcrossField = positionAcrossField + STR * timeStep$$

We're calling our axis associated with strafing *along* and our axis associated with moving forward or back *across.* This turned out to facilitate communication within our team more easily than axis conventions like *x/y/z.*

We now have our odometry. When using it in autonomous routines, we reset this value to zero at the start, and consider our coordinates relative to where the robot starts. This means robot positioning is very important, as any error will be carried through the odometry.

When you add the change every cycle, you always know where that wheel is. The same is done for the other 3 wheels. We use that information to find the center of the robot. The direction that the robot is facing could also be determined in the same manner, but we chose to use our gyros for that. We only store the current position of the center of the robot. We do , however, send that position to the dashboard and record the data there. We are able to graph out the position that the robot reported and impose it over a map of the field to allow us to make improvements to our autons even without having access to the robot.

The next part of the puzzle is to tell the robot where to go next. We reverse the tracking process to instruct each wheel on where to go next. We input the desired x,y coordinate into the subroutine. The angle that each wheel needs to face is calculated. The angle will be the same for all wheels unless the robot is going to spin while moving. If the robot will be changing heading while moving, the amount of turn correction will be factored in causing the wheels to face different directions and have different relative speeds until the spin portion is achieved. We use a positioning loop to assign the wheel speed. We are only using kP * error. We can change states by several different criteria. We might use an achieved distance, an intake sensor, or a targeting feedback to tell the robot that it is done with that task. The robot then moves on to the next task.

Calibration

If we used our nominal wheel diameters and gear ratios, these values should be pretty close to real-world values, but they probably won't be perfect. We'll want to calibrate our overall speeds by applying a multiplier. This calibration can be as simple as marking off a set distance (the longer the better) and driving the robot across this distance, keeping it as straight as possible. Once the distance is reached, the reported distance value can be compared with the actual distance value and ratioed to produce a correction factor - actual distance over reported distance. This can also be used to account for wheel wear, which changes the effective wheel diameter and can cause inaccurate distance measurements when not accounted for.

2.3 Integrating Other PIDs into Swerve

Swerve is really cool on its own, but you know what's even better? MORE FLEXIBLE CONTROLS!
"But isn't field-centric good enough?" some may ask.
In return we ask, "Do you want to be the very best that no one ever was?"

Let's get down to business. Swerve enables such freedom of motion, it's a waste to not utilize it for your advantage as much as possible. If done successfully, your robot will perform at a new level.

Not convinced yet? Think about it this way. The more control the driver has, the more natural driving will feel. The more comfortable the driver is, the better they will play. The better they play, the better your whole team will do. That is why we are so hard-set on having solid controls. We want to perform well. And we want you to as well.

2.3a PIDs (PIDFs) Loops: A Primer

Why Do We Even Need Fancy Control Loops?

Control. We are completely serious. Loops enable us to finely control practically any part of the robot. Yes, you can just set power to a motor with a joystick. That is fine for some mechanisms. If you haven't even tried doing this yet, STOP RIGHT NOW! Go and make your robot move! Start simple, then build your way up to complex controls.

Other mechanisms are very difficult to manage. Anything from battery voltage, to wheel wear can impact how effectively the mechanism is operated. No matter what, there will always be physical variance affecting robots. Never ever assume you will be driving your robot under certain, specific conditions. Nothing is ever perfect, even in auton!

We are able to compensate for these differences using control loops. Yes, we also use some awesome sensors to help us. However, control loops are the heart of reliable systems. Using a *well implemented PID* or *PIDF* system will produce smooth, repeatable movements.

It is important to also note that there are other types of control loops. Compared to the others, *PID* and *PIDF* are arguably the simplest and universal. That is why we use them wherever we can.

What Do These Letters Even Mean!?

These values are technically called *gains*. They help scale the *error* (the difference between our *target state*, "where we want to be", and our *current state*, "where we are right now") in different ways. This ends up being the *correction* we apply to the motor(s). For a more detailed explanation, please see this article. Below we will talk about each *gain*.

NOTE: These *gains* can be used together in different combinations.
NOTE: These sample calculations are not tuned PID loops. These are simply examples to show how the numbers work.

P - Proportion

We multiply the *error* by *P*.

$$error = targetState - currentState;$$

$correction = error * kP;$

All we are doing is converting the error into a useful number (e.g. a motor controller input).

Here is a simple numerical example:

$$error = targetState - currentState;$$
$$= 2000 - 1000;$$
$$correction = error * kP;$$
$$= 1000 * 0.0001;$$
$$= 0.1;$$

In this scenario, our *correction* is 10% motor output.

I - Integral

We continuously accumulate the *error* when we are very close to our *target* (within an *IZone*). Then, we scale the *integral* by the *responseTime*. Finally, we multiply this number by *I* and add it into the *correction* calculation.

$$if((abs(error) < IZone)\ integral += error;$$
$$if(abs(integral) > ILimit)\ integral *= responseTime;$$
$$correction = error * kP + integral * kI;$$

All we are doing is allowing a little more power to be applied at the end.

The integral cannot be stored as a local variable. It must be external to your *PID* function. We need to have a record of all previous tiny *error* values, otherwise the *integral* will not be large enough to do anything. Because of this, *you have to reset the integral before using your PID* again.

Additionally, *it is very important that you use an ILimit to restrict how strong the integral can get*. Without this, the *integral* can cause *overrun* and/or *oscillation*.

We can creep right to our target at an adjustable rate using *responseTime*. Slower systems have smaller values, while faster systems have large values.

Depending on your system, you might not need to incorporate the *integral*.

Here is a simple numerical example:

$$error = targetState - currentState;$$

$$= 2000 - 1991;$$

$$if((abs(error) < IZone)\ integral\ += error;$$

$$if((abs(9 < 10)\ integral\ += 9;$$

$$if(abs(integral) > ILimit)\ integral *= responseTime;$$

$$if(abs(9) > 100)\ integral *= 0.02;$$

$$correction = error * kP + integral * kI;$$

$$= 9 * 0.0001 + 9 * 0.002;$$

$$= 0.0009 + 0.018;$$

$$= 0.0189;$$

In this scenario, our *correction* is 1.89% motor output. Yes, this is a very small number. However, after a few more cycles the *integral* will make *correction* grow strong enough to creep the system right to the *target state*.

D - Derivative

We take the difference between our *current error* and our *previous error* divided by our *responseTime*. Then, we add the *derivative* into the *correction* equation.

$$derivative = (error - prevError) / responseTime;$$

$$correction = error * kP + derivative * kD;$$

Think of this like we're slowly using the brakes instead of stomping on them. The derivative helps us smooth out our movements.

We can also think about this in graphical terms. We are finding the slope of the *correction* line. The slope tells us what the *correction* should look like in the near future. This helps to smooth out the *correction*. As a result, *derivatives* can eliminate both *overrun* and *oscillation*.

Unfortunately, control loops using *derivatives* are highly susceptible to *noise* issues. Sudden jumps in values will cause unexpected behaviors.

Depending on your system, you might not need to incorporate the *derivative*.

Here is a simple numerical example:

$$error = targetState - currentState;$$

$$= 2000 - 1000;$$

$$derivative = (error - prevError) / responseTime;$$

$$= (1000 - 1010) / 0.02 = -500;$$

$$
\begin{aligned}
correction &= error * kP + derivative * kD; \\
&= 1000 * 0.0001 + -500 * 0.001; \\
&= 0.1 - 0.5; \\
&= -0.4;
\end{aligned}
$$

In this scenario, our *correction* is 40% reverse motor output.

F - Feed Forward

We multiply the *target* by *feed forward*. Then, we add that into the existing *correction* equation. All we are doing is providing the system an initial boost in power based on our existing knowledge of the system.

$$correction = error * kP + feedForward * kF;$$

Here is another way of thinking about it. We are supplying a known starting value to get us into our operating range. Think about a shooter wheel. If we want the wheel to run at a constant velocity, we already know how fast we want it to go. The *feed forward* gives our system a boost, so the rest of the *PID* doesn't need to work as hard.

Other Control Loop Terms

Open Loops

Dead reckoning: controlling the system based on *time*. There is no *feedback* from *sensors*, so the system is not able to *correct* for changing conditions. This method should be used as a starting point for building *closed loops*.

Closed Loops

PID(F) loops. Based on *feedback* we get from sensors, our system is able to automatically *correct* for changing conditions.

Internal Loops (our version)

PID(F) loops that are built into the motor controller. The *feedback sensor* is directly communicating with the motor controller. Use this whenever possible. They react quicker than *external loops* because the motor controller has a faster running clock than the RoboRIO (typically 1 ms vs. 50 ms).

External Loops (our version)

PID(F) loops that run on the RoboRIO. The *feedback sensor* is directly communicating with the RoboRIO. Use this when you have to. They react slower than *internal loops* because the

RoboRIO has a slower running clock than the motor controller (50 ms vs. typically 1 ms). Also, you are responsible for writing the control loop yourself.

Positioning Loops

PID(F) loops that have a *target position* in mind. Using an encoder (or other distance measurement device), they allow you to achieve a certain distance in a timely, consistent manner.

Velocity Loops

PID(F) loops that have a *target velocity* in mind. Using an encoder (or other velocity measurement device), they allow you to achieve a certain speed in a timely, consistent manner.

"Inside Outside" Loops (our definition)

Using an *external position loop* to set an *internal velocity loop*. The encoder (or other distance sensor) communicates with the RoboRIO, while another encoder (or other velocity sensor) communicates with the motor controller. Use this for ultimate control IF you are experienced. It can negate variations in battery levels.

General Tips for Tuning Control Loops

Good ole "trial and error". The cool kids call it "The WAG Method": Wild ______ Guess. (Use your imagination.)

Tuning control loops can be a very time-intensive process. It is a necessary step though. You have to customize the *PID* or *PIDF* to fit your robot. Each robot is different, so no two *PIDs* are alike. Even practice and competition robots might need to have slightly different *gains*. This is primarily due to weight differences between them.

See this article by CTRE for more information about tuning control loops.

Start with kP

NOTE: If the *control loop* for your system is going to maintain a certain *setpoint*, then it is a good idea to start with kF, and come back to this step.

Make a logical guess based on the units of measurement you are using and your output units. Let's think about a simple *positioning loop*. Let's say our encoder reads 100 ticks/inch and we are using percent output. Approximately how much power do we want applied to at a certain distance? We already have an idea of how far we want to move: we know the field measurements. So, let's say we want this *positioning loop* to give 100% output when we are 10 ft away. This is what the math would look like:

$$correction \ = \ error * kP;$$
$$1 \ = \ 120000 * kP;$$
$$kP \ = \ 1/12000 \ = \ 0.000833;$$

Using that *kP*, we can do some quick math to see how this behaves when we are 3 ft from our *target* distance.

$$correction \ = \ error * kP;$$
$$correction \ = \ 3600 * 0.0000833;$$
$$correction \ = 0.299 \ = \ 30\%;$$

Based on this, it seems we are within our operating range. Onward to tuning time!

It is recommended you double your *kP* value until you see *oscillation*, or what we call *"wagging"*. If your robot starts shimmying and shaking, that means your *gain* is too high. Try going 75% of the previous value. If that looks good, continue increasing the *gain* slightly until you see more *oscillation*. Once you see more *oscillation*, lower the *gain* a tiny bit.

Now that we are content with our *gain*, we need to make sure it works throughout our entire operating range.We have to test the control loop under different conditions. In our earlier example, we would need to physically test our *positioning loop* at different distances. We want the robot to always achieve its distance, no matter the distance (within reason). To perform this test, we set up a range *target* distances, both traveling forwards and backwards. *DO NOT OVERLOOK THIS STEP!* Please don't ever assume your loop will work correctly in both directions.

If a system is traveling too quickly in certain scenarios, it may be a good idea to apply a *correction cap*. This allows us to keep our tuned *gain* without sacrificing control due to momentum.

We want to maximize the responsiveness of our system. When it is on the edge of *oscillation*, the *gain* is just right. That is why we go through this process. Just calculating a value alone is not enough. You have to test and tweak the *gain* to fit your system.

Next is kD

Now that we are happy with our *kP*, we can start tuning *kD*.
NOTE: Depending on how your mechanism is designed and the type of *control loop*, you might not need to use *kD*. Brushless motors themselves behave much differently from brushed motors. We have found that *control loops* using brushless motors and *kD* are much harder to tune. They have a lot more torque, making the *derivative* difficult to control.

It is recommended that you start the *kD* at 10 times *kP*. In our previous example, that would mean:

$$
\begin{aligned}
kD &= 10 * kP = 10 * 0.0000833 = 0.000833; \\
derivative &= (error - prevError) / responseTime; \\
&= (3600 - 3620) / 0.02 = -100; \\
correction &= error * kP + derivative * kD; \\
&= 3600 * 0.0000833 + -100 * 0.000833; \\
&= 0.299 - 0.0833 = 0.217 = 22\%;
\end{aligned}
$$

Based on this, it seems we are within our operating range. Onward to tuning time!

It is recommended you double your *kD* value until you see you come short of your *target*. If your mechanism *overshoots* (or travels past your *target*), that means your *gain* is too low. If it stops abruptly, that means your *gain* is too high. Try going 75% of the previous value. If that looks good, continue increasing the *gain* slightly until you see it come short again. Once you see more of this, lower the *gain* a tiny bit.

You may want to increase your *kP slightly* to maximize the speed of your system. Ideally, you want to find the balance between speed and accuracy with any *control loop*.

NOTE: The *derivative* is not intended to get you exactly to your *target*; that's what the *integral* is for. Instead, we use *kD* to help eliminate *overshooting* the *target*.

Now that we are content with our *gain*, we need to make sure it works throughout our entire operating range.We have to test the control loop under different conditions. In our earlier example, we would need to physically test our *positioning loop* at different distances. We want the robot to always achieve its distance, no matter the distance (within reason). To perform this test, we set up a range *target* distances, both traveling forwards and backwards. *DO NOT OVERLOOK THIS STEP!* Please don't ever assume your loop will work correctly in both directions.

We want to maximize the responsiveness of our system. When it is just shy of the *target*, the *gain* is just right. That is why we go through this process. Just calculating a value alone is not enough. You have to test and tweak the *gain* to fit your system.

Then kI

Now that we are happy with our *kD*, we can start tuning *kI*.
NOTE: Depending on how your mechanism is designed and the type of *control loop*, you might not need to use *kI*.

It is recommended that you start the *kI* with a fairly small value. We only want the *integral* to be active when we are extremely close to our target. So, let's look at an example without the *integral*, then with it:

$$
\begin{aligned}
derivative &= (error - prevError) / responseTime; \\
&= (9 - 10) / 0.02 = -50; \\
correction &= error * kP + derivative * kD; \\
&= 9 * 0.000833 + -50 * 0.000833; \\
&= 0.0075 - 0.0417 = -0.0342 = -3\%;
\end{aligned}
$$

$$
\begin{aligned}
derivative &= (error - prevError) / responseTime; \\
&= (9 - 10) / 0.02 = -50; \\
&if((abs(error) < IZone)\ integral \mathrel{+}= error; \\
&\quad if((abs(9 < 10)\ integral \mathrel{+}= 9; \\
&if(abs(integral) > ILimit)\ integral \mathrel{*}= responseTime; \\
&\quad if(abs(9) > 100)\ integral \mathrel{*}= 0.02; \\
correction &= error * kP + integral * kI + derivative * kD; \\
&= 9 * 0.000833 + 9 * 0.005 + -50 * 0.000833; \\
&= 0.0075 + 0.045 - 0.0417 = 0.09415 = 9\%;
\end{aligned}
$$

See the difference? Without *kI*, we are stuck just short of our *target*. With *kI*, we will accumulate enough *correction* to get right to our *target*.
Based on this, it seems we are within our operating range. Onward to tuning time!

It is recommended you double your *kI* value until you see *oscillation*, or what we call *"wagging"*. If your robot starts shimmying and shaking, that means your *gain* is too high. Try going 75% of the previous value. If that looks good, continue increasing the *gain* slightly until you see more *oscillation*. Once you see more *oscillation*, lower the *gain* a tiny bit.

Don't forget that you can also play with *IZone* and *ILimit*. These values can help create a strong, yet controlled correction right to the *target*.

Now that we are content with our *gain*, we need to make sure it works throughout our entire operating range.We have to test the control loop under different conditions. In our earlier example, we would need to physically test our *positioning loop* at different distances. We want the robot to always achieve its distance, no matter the distance (within reason). To perform this test, we set up a range *target* distances, both traveling forwards and backwards. *DO NOT OVERLOOK THIS STEP!* Please don't ever assume your loop will work correctly in both directions.

We want to maximize the responsiveness of our system. When it is on the edge of *oscillation*, the *gain* is just right. That is why we go through this process. Just calculating a value alone is not enough. You have to test and tweak the *gain* to fit your system.

What about kF?

<u>NOTE:</u> To use *feedforward* effectively you have to have a good idea of how your system will behave ahead of time.

kF is the simplest *gain* to tune. You just need to find a value that gets you right into your operating range. *Feedforward* doesn't perform any *corrections*, rather it moves your starting point from 0 to "whatever you want". This makes it much easier to tune the rest of the *PID*. Having a tighter range to *correct* results in faster reactions and finer control.

Once your *feedforward* has your system off to a good start, then you can return to tuning *kP*.

2.3b Goal Centric (Starring the Limelight)

In this section, we shall discuss integrating the *Limelight* with swerve. We are assuming that you have some knowledge about the *Limelight*. If you are using another form of vision processing, the concepts we will discuss should still be relevant.

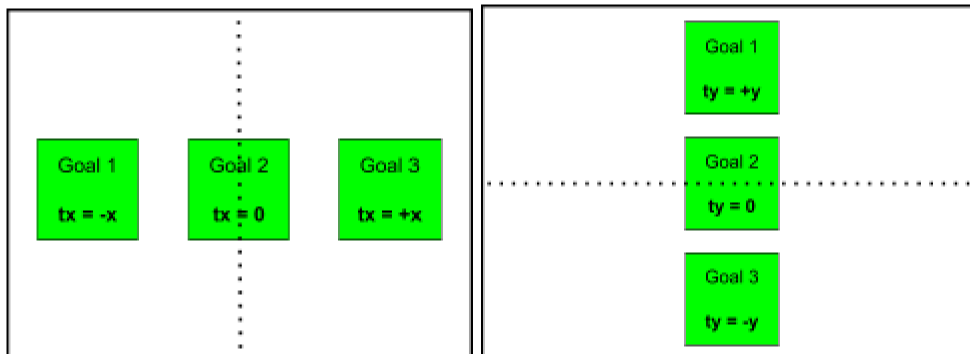
Vision Targeting

Essentials

Almost all vision processing software should give you two very useful variables: *tx* & *ty*. Below is the description of these variables from the *Limelight* documentation.

tx	Horizontal Offset From Crosshair To Target (-27 degrees to 27 degrees)
ty	Vertical Offset From Crosshair To Target (-20.5 degrees to 20.5 degrees)

In the simplest targeting implementations, all you need is *tx*. More complex targeting implementations will also utilize *ty*.



In the left diagram, we show how the *Limelight* reports *tx* values. The black dotted line shows us the y-axis, where *tx* = 0. We can see that *tx* tells us how far LEFT (-*tx*) or RIGHT (+*tx*) the target is from the center of the *Limelight.*

In the right diagram, we show how the *Limelight* reports *ty* values. The black dotted line shows us the x-axis, where *ty* = 0. We can see that *ty* tells us how far UP (+*ty*) or DOWN (-*ty*) the target is from the center of the *Limelight.*

Easy Mode

The simplest vision targeting *PID* you can do only uses *tx*. Basically, you just want to turn towards the goal. This is done by applying a calculated correction in the opposite direction of *tx*. Here is what your targeting *PID* function might look like:

```
double limelightXPID(double tx){
    double kP = 0.008;
    double correction = tx * kP;
    return correction;
}
```

The *kP* should convert the *error* into a number you can use later in your method to move the drivetrain (teleop or auton).
Unfortunately, this implementation isn't perfect. Oftentimes, the robot doesn't have enough power to correct when it is fairly close to the goal. The best solution we found is in the next section.

Advanced Mode

A more advanced version of the vision targeting *PID* implements a *minimum correction* and an *error deadzone*. Just like the previous one, *tx* is used the same way. However, the *minimum correction* allows the robot to creep into place and the *error deadzone* allows the robot to stop when it's close enough. Here is what your targeting *PID* function might look like:

```
double limelightXPID(double tx){
    double kP = 0.008;
    double correctionMin = 0.003;
    double deadZone = 0.05;
    double correction = tx * kP;
    if(correction < correctionMin) correction = copysignf(correctionMin, correction);
    if(abs(tx) < deadZone) correction = 0;
    return correction;
```

}

Hard Mode

A harder version of the vision targeting [*PID*]() adds an *integral* into the mix. Just like the previous approaches,9 *tx* is used the same way. However, the *integral* adds a little power at the end to help the robot crawl right into proper positioning. The *integral* replaces the *minimum correction* seen in the [previous]()section. Here is what your targeting *PID* function might look like:

```
double integral = 0;
double limelightXPID(double tx){
    double kP = 0.008;
    double deadZone = 0.05;
    double kI = 0.001;
    double IZone = 4.0;
    double ILimit = 1000.0;
    if(abs(tx) < IZone) integral += tx;
    if(abs(integral) > ILimit) integral = 600 * integral/abs(integral);
    double correction = tx * kP + integral * kI;
    if(abs(tx) < deadZone) correction = 0;
    return correction;
}
```

Caution: tuning the *integral* can be extremely difficult. From our own experience taking the [advanced]()approach is much easier and has very similar results to this implementation.

Expert Mode

A more complex version of the vision targeting [*PID*]() utilizes both *tx* & *ty*. Just like the previous ones, *tx* is used the same way. However, now *ty* is used to calculate the distance from the goal. Depending on the year and setup of your robot, your use of *ty* can vary. We will briefly discuss a few of the most common ones.

Because *ty* gives us the vertical offset (in degrees) we are away from the target, we can use that to achieve a specific position. (For the purposes of simplicity, we are using the pseudocode from [this]() example.) Here is what your targeting *PID* function might look like:

```
double limelightXPID(double tx){
    double kP = 0.008;
    double correctionMin = 0.003;
```

```
        double deadZone = 0.05;
        double correction = tx * kP;
        if(correction < correctionMin) correction = copysignf(correctionMin, correction);
        if(abs(tx) < deadZone) correction = 0;
        return correction;
}

double limelightYPID(double ty){
        double kP = 0.008;
        double correctionMin = 0.003;
        double deadZone = 0.05;
        double correction = ty * kP;
        if(correction < correctionMin) correction = copysignf(correctionMin, correction);
        if(abs(tx) < deadZone) correction = 0;
        return correction;
}
```

Together, these two PIDs enable your robot to turn towards and line up to the target, while driving to a desired distance away from the target. You can change the distance from target position by adding an offset to *ty* (*ty - 3)*.

Another use for *ty* is to use the "distance" to enhance your scoring mechanism. For example, you can change the speed of a shooter wheel based off of your distance, or the angle of a shooter hood. You can adjust the height of an elevator based on the vertical offset, or even disable scoring until your robot is close enough to place the game piece. The use of *ty* is only limited by your imagination. Because there are so many possibilities, we will not show any pseudocode.

Goal-Centric

Introduction

In goal centric, instead of the driver joysticks referencing the field (as in field-centric control), they reference a specific goal or target. In the 2020 season, we specifically targeted the Upper Power Port. In the 2019 season, we targeted the Loading Station, Cargo Ship, and Rocket for Hatch placement. Essentially, you want to target any major field element to make it easier and faster to score. If implemented properly, the robot will naturally face the goal and strafe in an arc around it.

How It's Done

If you are using just *tx* to line up with the goal, here is what your code may look like:

```
move(fwd, rot + limelightXPID(tx), str);
```

Notice that the *tx* [PID] is added to *rotate*. We just want our robot to look at the target. So, all we have to do is turn toward the goal. We are only aiming using the *Limelight*, not moving.

If you are using both *tx* and *ty* to line up with the goal, here is what your code may look like:

$$move(fwd + limelightYPID(ty), rot + limelightXPID(tx), str);$$

In addition to *tx*, notice that the *ty* PID is added to *forward*. We want our robot to drive to a specific location relative to the target. So, we have to drive toward/away from the goal. We are moving to a set position, while looking at the target.

These functions must be called every cycle. The robot can't properly correct invalid positioning based on old data.

2.3c Game Piece Centric (Starring the Pixy)

Introduction

In game piece centric, the rotate input from the joystick is overridden by the Pixy. The driver is free to move the robot in any direction while the Pixy keeps the robot pointed at a ball. This makes picking up balls on the far side of the field a little easier. Since we have a Pixy on the front intake and the rear intake, the driver has to select which one to use for this mode.

Here is a video of our 2020 robot demonstrating *game piece centric* control.

How It's Done

There are two signals that we receive from the Pixy. There is a digital signal that tells us if a target is within range. In this case a target is a yellow ball (Power Cell). There is an analog signal that gives us the location of the ball on one axis of the camera's view. In this case the x or left to right axis.

The Pixy's analog signal is from 0 to 3.3 volts with 0 being the leftmost view of the camera. We determined the center of the camera's view to be what we wanted to lock in on. We created a positioning loop that took over the rotational control and calculated the error between the target and the center of the camera. The error was used to calculate our desired yaw until the digital signal showed no more target.

```
double pixyVoltageRange = 3.3; //volts
double pixyHFOV = 60; //degrees

bool pixyFrontSeesBall(){
    return pixyFrontDigitalInput.Get();
}
```

```
double pixyFrontAngle(){
return (((pixyFrontAnalogInput.GetAverageVoltage())/pixyVoltageRange) − 0.5) * pixyHFOV;
}

double pixyFrontCorrection(){
	double pixyXkP = 0.004;
	return pixyFrontAngle() * pixyXkP;
}
```

In teleop

```
if(pixyFrontSeesBall()) {
	yawCorrection = pixyFrontCorrection();
	rot = 0;
}else{
	storedYaw = yaw;
}

move(fwd, rot + yawCorrection, str);
```

We use the Pixy in a different manner in auton. We use it to determine our position relative to the target ball to allow for automatic adjustment in the case of minor set up variances.

```
if(pixyFrontSeesBall() && (abs(pixyFrontAngle() + goalYaw) > 3.0)) {
	move(0, yc, −copysignf(0.1, pixyFrontAngle()), true);
}else{
	move(0, 0, 0, true);
}
```

In this case, we'll use the Pixy to center ourselves on the game piece if we're more than 3 degrees off from it in either direction.

2.3d Maintaining Your Heading (aka Not Spinning... aka Driving "Straight") While in Field Centric

Introduction

Driving "straight" isn't as easy as you thought, is it? When we say "straight", we mean moving without unintentionally turning. We've all been there: "why is the robot steering left/right even though I'm telling it to drive straight!?!" This is something we struggled with for a while too. *Unfortunately, there will always be physical variances between individual wheels or two sides of a drivetrain.* There will always be a need for a drive "straight" *PID*.

When it comes to swerve, driving "straight" is more complicated. Instead of the robot driving forward/backward at a specific *heading/angle*, we want the robot to move in any direction without spinning. *The robot should always stay facing the same way, unless we tell it to rotate. When it rotates, that angle will be our new heading to maintain.*

Now things get even more interesting. How do we do this when we have other *PID loops* (i.e. *goal centric*, *game piece centric*) occurring simultaneously? *The robot should switch between goal centric, game piece centric, and field centric seamlessly.*

Not spinning isn't as easy as you thought, is it? Don't worry, we're going to help as much as we can.

How It's Done

You need to have a variable that stores your *target heading*. Typically, we refer to ours as *storedYaw*. This variable must be updated every time you have a new *angle* you want to face.

In addition, you need to have a variable that stores your *current heading*. Typically, we refer to ours as *yaw*. This variable must be updated every cycle with data from your *gyro*.

Then, you have a simple rotation *PID* which might look like this:

```
double calcYawStraight(double targetAngle, double curentAngle, double kP){
    double errorAngle = remainderf((targetAngle - currentAngle), 360);
    double correction = tx * kP;
    return correction;
}
```

You would call this function in *Teleop* like so:

```
if(rot != 0){
```

$$
\begin{aligned}
&\quad storedYaw = yaw;\\
&\}else\{\\
&\quad if(\ abs(speed)\ >\ 0 \,||\, abs(strafe)\ >\ 0\ )\{\\
&\qquad yawCorrection\ = calcYawStraight(storedYaw,\ yaw, 0.004);\\
&\quad \}\\
&\}
\end{aligned}
$$

$$move(fwd, rot + yawCorrection\ , str);$$

Essentially, we want to keep facing the same *angle* whenever we aren't rotating. When we don't want to spin, we want to maintain our *heading* while we move. However, we don't want to *PID* our angle if we aren't trying to move (due to safety concerns).
To integrate this with other driving *PID loops*, make sure that maintaining your heading is your "default" scenario. *Goal centric* and *game piece centric* will modify your *yawCorrection* and *storedYaw* (or equivalent variables) as needed.

3.0 Other Misc. Tips & Tricks

3.1 "Zero" Azimuth Position & Pre-Match Module Testing

In order to make sure that your robot is functioning properly, you have to check to make sure that things remain in alignment. That includes making sure that the wheel rotation encoders stay true. There are a couple of things that mechanically could go wrong and misalign the wheels, the encoder could come loose, the gear driving the encoder could become damaged and a number of other things. To prevent these things from affecting our gameplay, we created a way to check our azimuth.

We made a few attempts to make physical measurement tools to simplify checking if the wheels were properly aligned. After several complex procedures, we developed a simple way to confirm that the wheels are parallel to the frame. We tip the robot on the side and put a level on each wheel after we told the wheels to point to the front.

We added a button in Test Mode that would tell all the wheels to face zero degrees relative to the robot. We also added a button to slowly rotate the wheels. That procedure gives us several points of feedback. We can observe the speed at which the wheels rotate relative to each other. We can listen for sounds that indicate mechanical changes. We can watch for smooth motion. It also allows us to move the wheels from the front-facing position then push the button to face front again.

Another part of the prematch test is running the modules in Teleop Mode with the wheels off the floor. That allows us to check for any mechanical changes to the drive motor functionality in addition to the spin function.

3.2 Protect Your Reset Buttons!

Reset Buttons: a necessary evil. They should only be pressed when you want them to be pressed, right?! Well, it's not something people think about as often as they should. Trust us, we fell into this trap too.

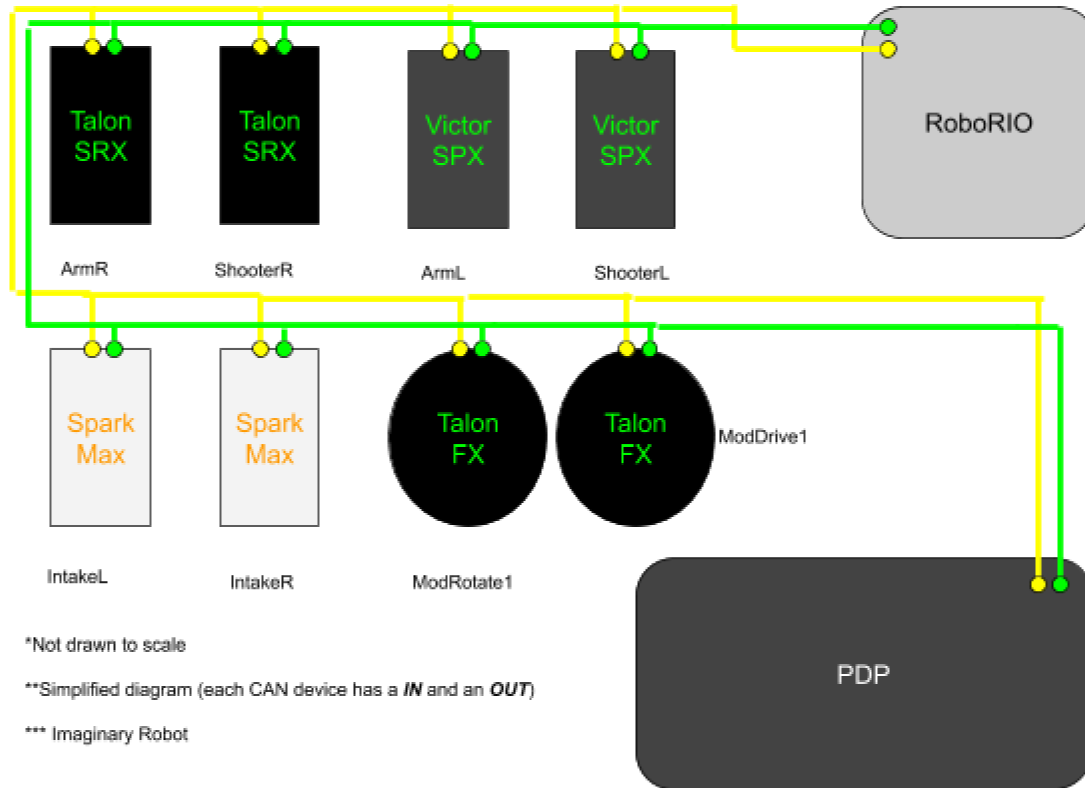
When we got our 2020 robot on the ground, our driver found a cool trick that made his forward direction become his backward direction. Our electronics panel is on the bottom of this robot in a very, very, very compact space. Our two gyros have somewhat open reset buttons that can *easily* be touched by anything. When our driver went over the shield generator, the reset button would be pressed and field centric is out the window. Every time it reset, he gave the programming team a nice little grin and said “front isn't front anymore.” We couldn't understand why the gyros were not working and would power cycle and reset. Finally we turned the robot upside down and found that a cable was leaning directly on the reset button of our NavBoard.

3.3 Document Your CANbus Network Topology

Despite the CANbus allowing devices to be connected in almost any order, it is still very important to know the network layout. Save yourself a lot of pain and suffering debugging your electronics! Here are some tips we have to do this effectively:

1. *Note the type, ID, and name for each device.*
2. *Trace the connections starting from the RoboRio to the PDP (or whatever the “end” of your CANbus is).*
3. *Record how each device is connected in the network in the same order you are tracing them.*

Below is a simple example:



Name	ID	Device Type	Connection In	Connection Out
ShooterL	1	VictorSPX	RoboRIO	ArmL
ArmL	2	VictorSPX	ShooterL	ShooterR
ShooterR	3	TalonSRX	ArmL	ArmR
ArmR	4	TalonSRX	ShooterR	IntakeL
IntakeL	5	SparkMax	ArmR	IntakeR
IntakeR	6	SparkMax	IntakeL	ModRotate1
ModRotate1	7	TalonFX	IntakeR	ModDrive1
ModDrive1	8	TalonFX	ModRotate1	PDP

By methodically writing down this information, it will be *infinitely easier* and *faster* to:

a. *Program these electronics on the robot.*
b. *Diagnose problematic devices & replace them.*
c. *Fix broken wires & connections.*

Please, do this simple thing. We didn't until we were forced to. That was a huge *mistake*. The day before our first competition in 2020, one of our motor controllers died. We spent at least 10 hours chasing the problem. We didn't start down the right path until we knew exactly what the CAN

network looked like. Seriously, this helps make debugging significantly faster! Don't let your entire electronics board falling out of your robot stop you from making it to your next match. We didn't; at least, not in 2020.

3.4 Know Your Electronics

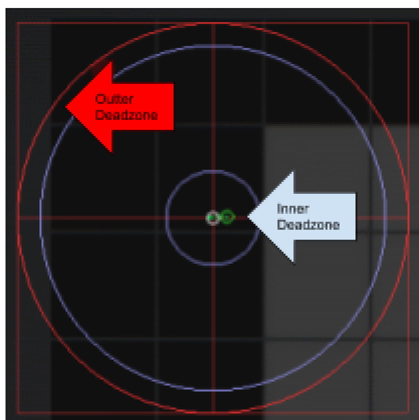
Options, options, options... Electronics are all the bits and pieces of hardware that we interface with through programming. We are always searching for that one sensor that will be even better than the previous one. Maybe you found a new thingy that does the "ooh yeah" (aka what you really need)! *Now you have to reliably control it.* Typically, we end up using less than 10 percent of the functions available of some of our electronics. That could be a motor controller, a gyro, a lidar, a proximity sensor, or a vision processor. You have to determine the capabilities and limitations of the electronics. It starts with the mandatory electronics: the *Roborio*, the *PDP*, and the *VRM*. In addition to the minimal operations, they all have programming in them that may be helpful. The error messages that they output could help you reduce any lag time. The diagnostic capabilities may allow you to prevent brownouts.

You should also have a variety of sensors on the robot. During the 2019 build season, we discovered that a reflective line sensor could be also used to detect hatches. The sensor would reflect the proper amount of light to signal the edge was within 24 inches. With that sensor at the correct angle, we could tell if we were in possession of a hatch. Feel free to explore alternate uses. We have used a variety of digital sensors to indicate the location of different things over the years. We also use analog sensors to give us useful feedback. Lidar and vision processing can give us distance. An accelerometer and gyroscope can tell you if you have run into something. A pressure sensor can tell you how much air is in the pneumatics. An encoder can tell you a position or a speed. *Knowing the capabilities of your electronics is the only way to maximize performance.*

Knowing the limitations of the electronics is a must for reliable programming performance. A simple example of this is knowing that most CTRE devices have clearable error codes. Sometimes the "sticky faults" will prevent the system from working at full speed because the expected data is not arriving quickly enough. Testing is often necessary to acquire the limitations of a sensor, encoder, or subprocessor. You will always need to know the point at which any feedback device is no longer reliable. It could be a speed, a distance, or an angle.

3.5 Invest in Good Controllers... And You Should Interpolate

Let's get real. Swerve feels AMAZING to drive. But, that's only if you have an AMAZING controller. "What do we mean by AMAZING," you ask? It boils down to "non-sticky" joysticks. Minimal dead zones. Want to make the robot do "victory spins" whilst driving at 45°? Your joysticks need to have a full range of motion.



This is an example of ideal radial dead zones. Realistically, no controllers are perfect. But, AMAZING controllers are very close. If you want to have even more natural feeling controls, then you should implement radial dead zone interpolation. Check out this website to learn more. http://blog.hypersect.com/interpreting-analog-sticks/

If you want to test and/or configure your controller, you can do so right in Windows 10. Check out this walkthrough for more information: https://www.howtogeek.com/241421/how-to-calibrate-your-gaming-controller-in-windows-10/

Makes sense right? To fully experience the epicness of swerve, you need to have quality inputs. Swerve simply can't be at its best if it's being driven with a poor controller.

3.6 Seriously, Implement Brownout Protection!

Brownout protection is very important with swerve, not only in auton but throughout the entire match. If we were to brownout, we could lose both our gyros. Once the voltage comes into the range of 6.3 to 6.8 volts our NavBoard would lose power as the 6 volt rail of the RoboRio turns off. In the range 4.5 to 6.3 volts, all but the joystick communications to the RoboRio are lost. The PigeonIMU can be powered by a Talon, therefore is more tolerable to brownout voltage conditions. The PigeonIMU, when powered by a Talon, will not lose it's telemetry until the voltage drops below 4.5 volts, at which time RoboRio shuts down. If the RoboRio shuts down, there will be no CANbus communication until it reboots again.

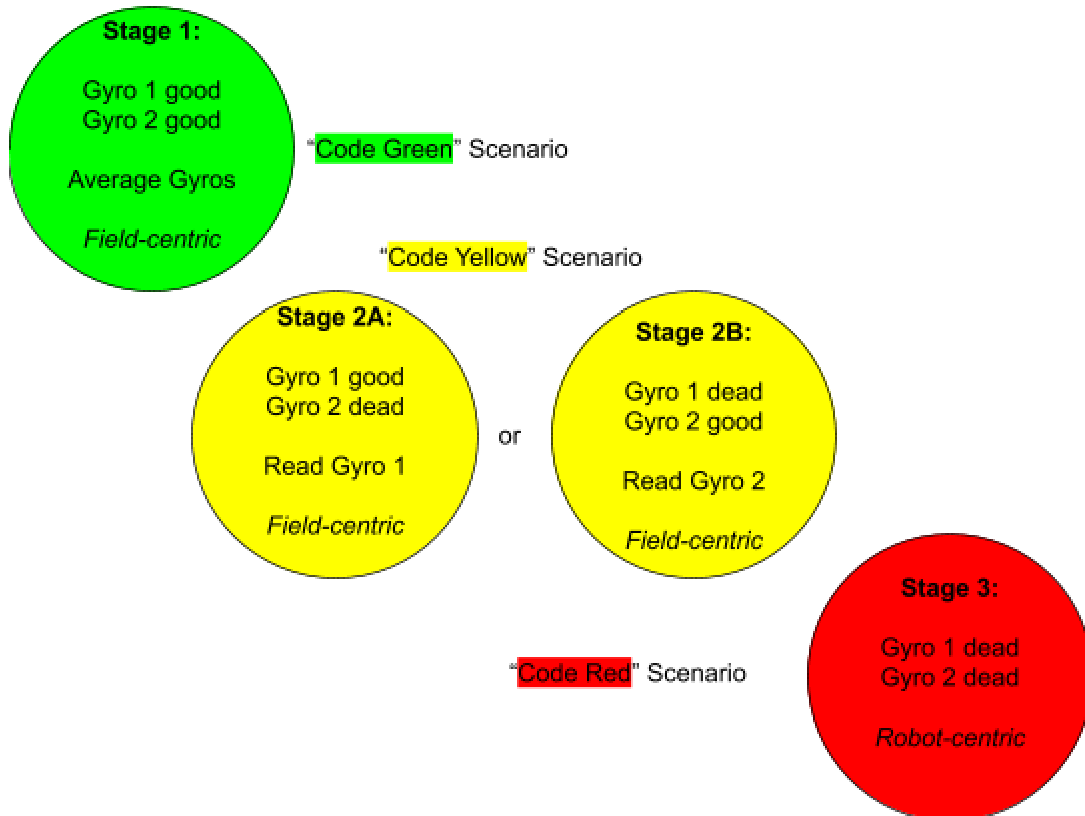
Battery Voltage	RoboRio	PigeonIMU	NavBoard
Fully charged - 6.8	Operates Normally	Operates Normally	Operates Normally
6.8 - 6.3	6 Volt Rail Shuts Off	Operates Normally	Loses Power
6.3 - 4.5	CANbus Shuts Off	No Communication (Still Operational)	Still Shut Off
4.5 and below	Shuts Off	Shuts Off	Still Shut Off

There are two ways to best minimize your chances of brownouts. The first way (that is multi-purpose) is ramprate. It prevents the motors from drawing maximum amperage immediately,

by forcing a 'ramp-up' of the motor controller output voltage, averting brownouts. The other purpose of ramprate is to keep from braking traction, by stopping the wheels from accelerating too rapidly. The second way to prevent brownouts is using current limiting. We used Falcon500 with the integrated Talon motor controller. Once we found the correct level to set the limiting at, it worked flawlessly. We used this method for most of our motors, from shooter to our climber to our drive train. On our shooter, current limiting worked extremely well because we could get it up to speed quickly without drawing too much amperage and browning out.

3.7 Handling Gyro Fault Conditions

We have two gyros on our robot. *In the event that one can't be used, we are still able to navigate. It's a backup.* We average both yaw readings together. If either one of the gyros goes down, the system will use the reading from the other gyro. If both gyros go down, the system will switch to *robot centric* mode until at least one gyro is regained.



We did program in a gyro reset function to account for this possibility, but it has not and should not be needed in competition. This emergency gyro reset function is used to restore the robot's heading to 0 degrees for "field centric" operation. There is a reason why this shouldn't be relied on during a match. If both gyros fail, you more than likely have bigger problems on your hands. Gyros don't fail "randomly" all the time. There is always a cause behind it. Resetting the gyros is not going to fix

electronic robot barf. Or perhaps a bug in the code. *It is dangerous to rely on sensor reset functions instead of resolving the true issue.* Please don't hear us wrong. *Having emergency reset functions is important; however, they should only ever be used in a true emergency.*

THE END... or is it...?

(It might be... who knows...)