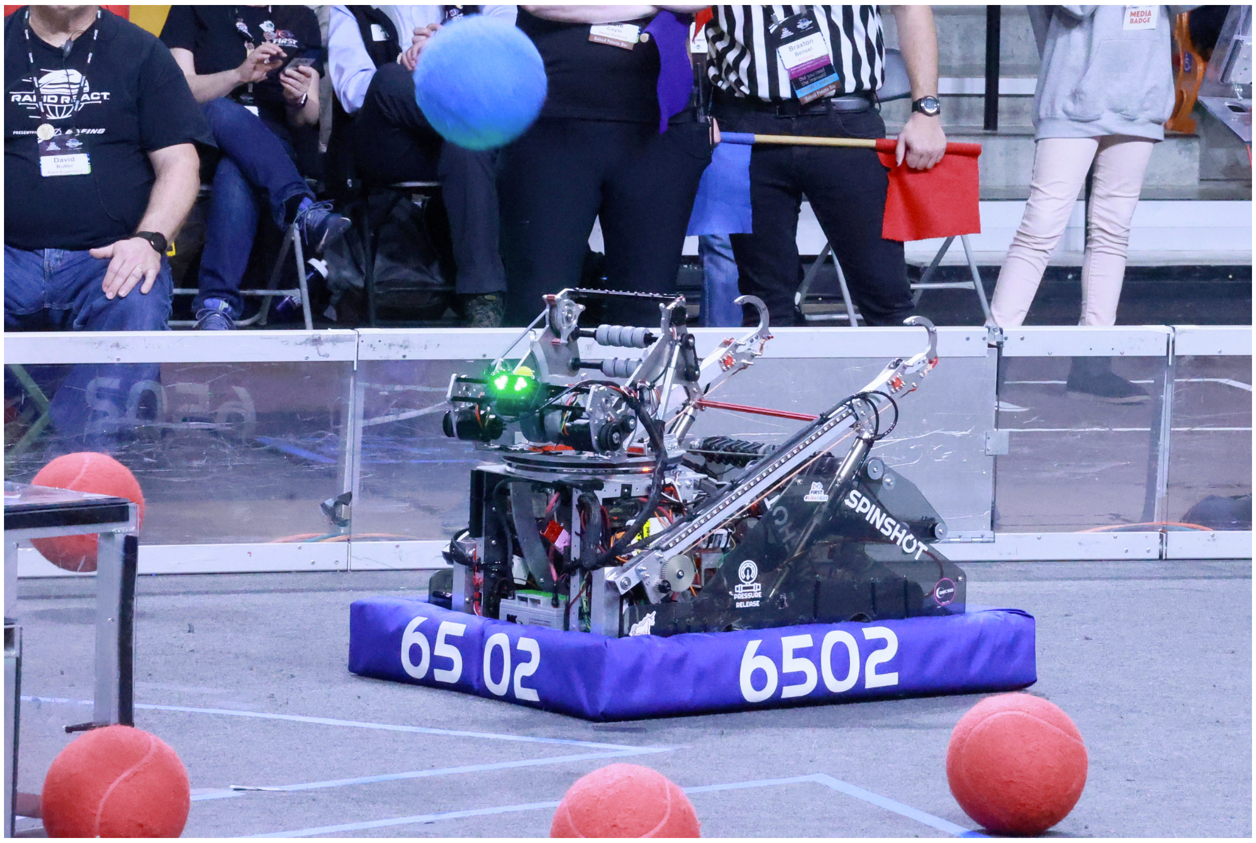
My name is Tate Staples and I am the software lead for team 6502 SpinShot. We won the NC's Innovations in Control Award and I wanted to take a moment to describe the software features of our robot, as a way to share with other teams and to create an opportunity for feedback. As part of my first real year programming for a real FRC event, I wrote over 11,000 lines of code, 75% of which was a library for our team called Kyberlib. I wrote this as both a personal exploration of the important parts of control theory and as a way to enable future members of our team to write code easier and faster. This code is written in Kotlin, a JVM language similar to Java.

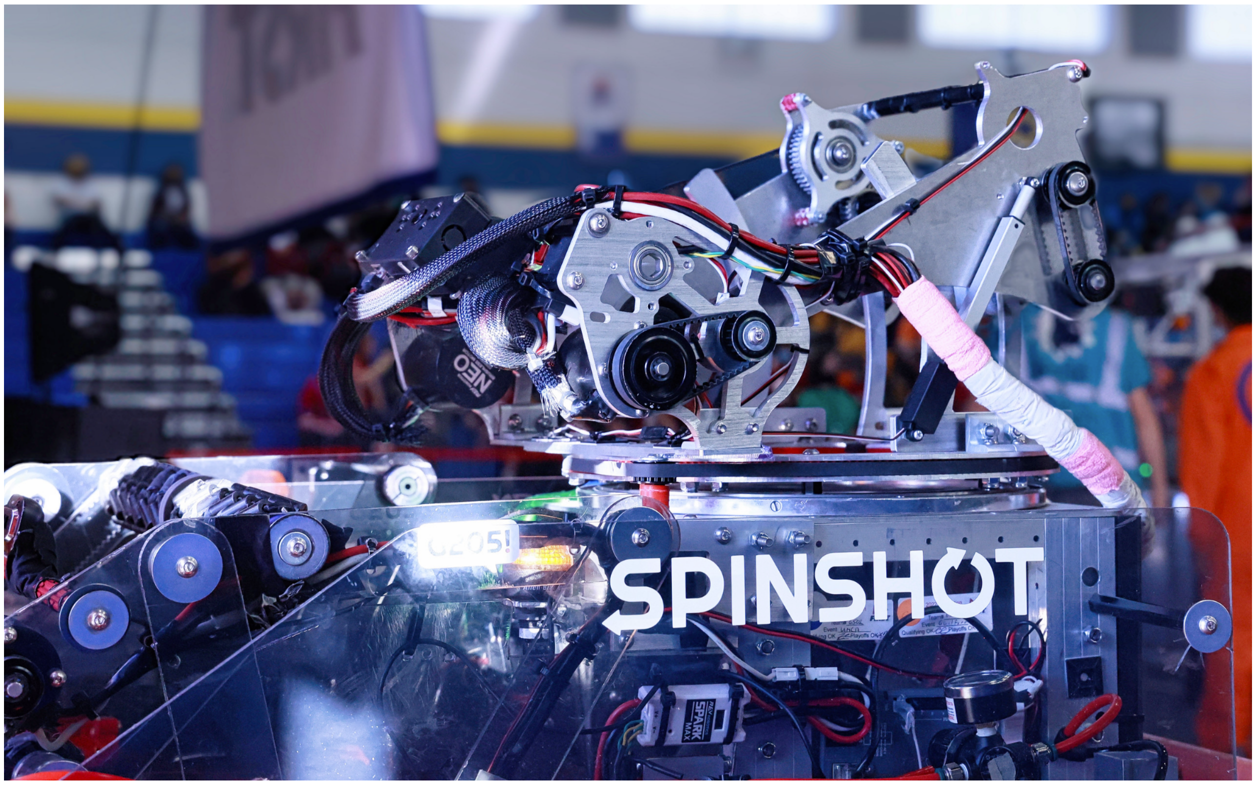


First, I would like to review the features of our 2022 robot, SpinShot. SpinShot had a 6 wheel west-coast drive with a 360º turret, active hood, mid-climb, and deployable intake. Electrically, we used Neos and Neo550s controlled with SparkMaxs.

The first major feature we added this year that was a cornerstone of the rest of our robot was the new pose estimators added to WPILib. By combining our wheel encoders, Pigeon gyroscope, and distance updates from our limelight; we were able to track the position of the robot throughout the entire match. This worked very well for us because the vision updates kept us locked onto the hub regardless of wheel slippage and defense.

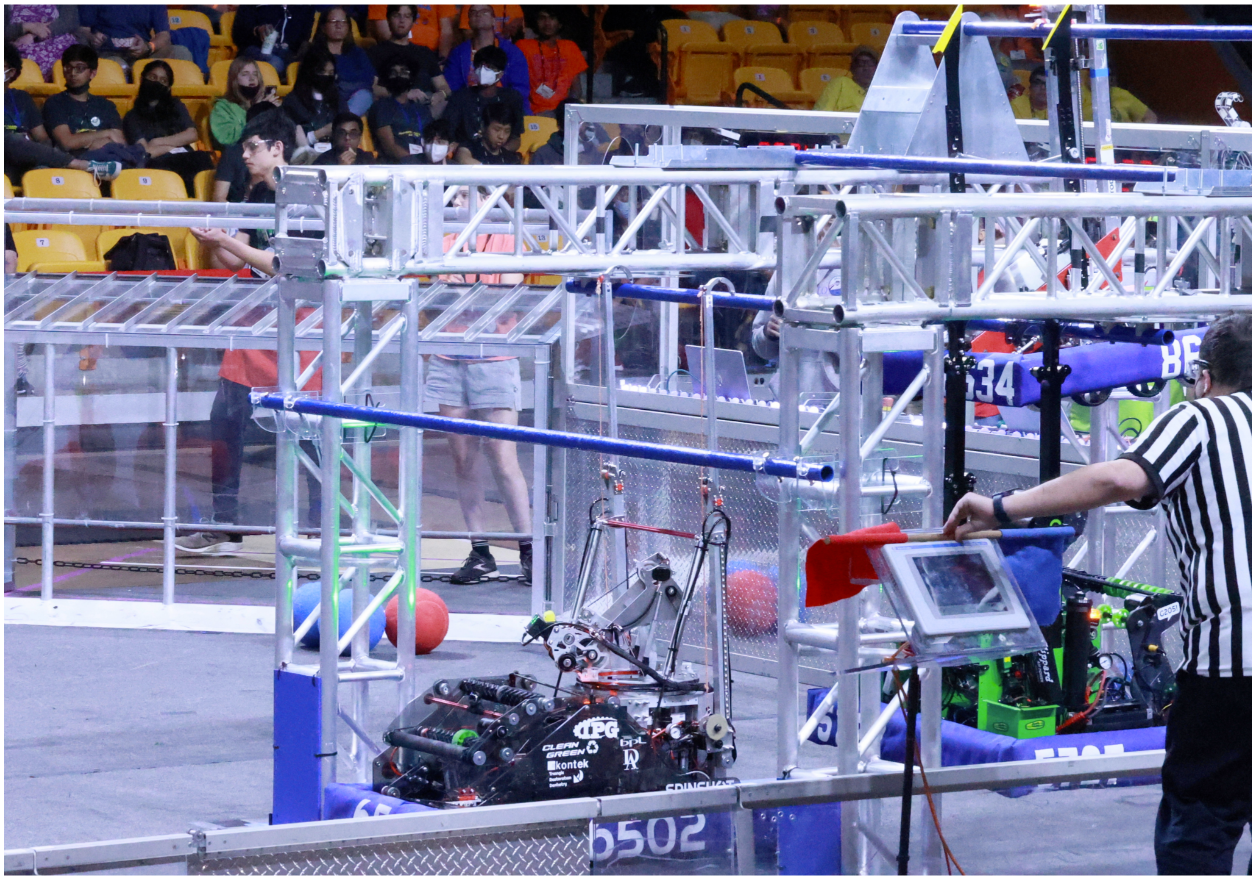
For autonomous, we integrated Pathweaver for our trajectories and had our own custom integration into Pathweaver that allowed us to drag and drop commands into our autonomous routines. This allows for quick iteration on autonomous routine and simple communication with alliance partners. We developed a reliable 4-ball and inconsistent 5-ball. Additionally, we created a 2 ball auto with 2 ball disposal. One area for further study is that we kept oversteering in our autonomous routines and we will need to figure out the issue (I suspect our track width was not correct).

The main focus of our robot was our shooting system. Our turret was controlled using state-space for position control and had a hall sensor for zeroing position. While visible aiming was handled by a filtered limelight offset, and when not visible turret angle was determined by our position tracking. This blind tracking was accurate enough that we were capable of shooting throughout a game even without the limelight connected. In order to improve our aiming, the turret was also adjusted to compensate for side spin introduced by the wheel feeding the ball to the flywheel. Our turret was used both for shooting into the Upper Hub as well as disposing of enemy alliance cargo towards our Hangar for field control.

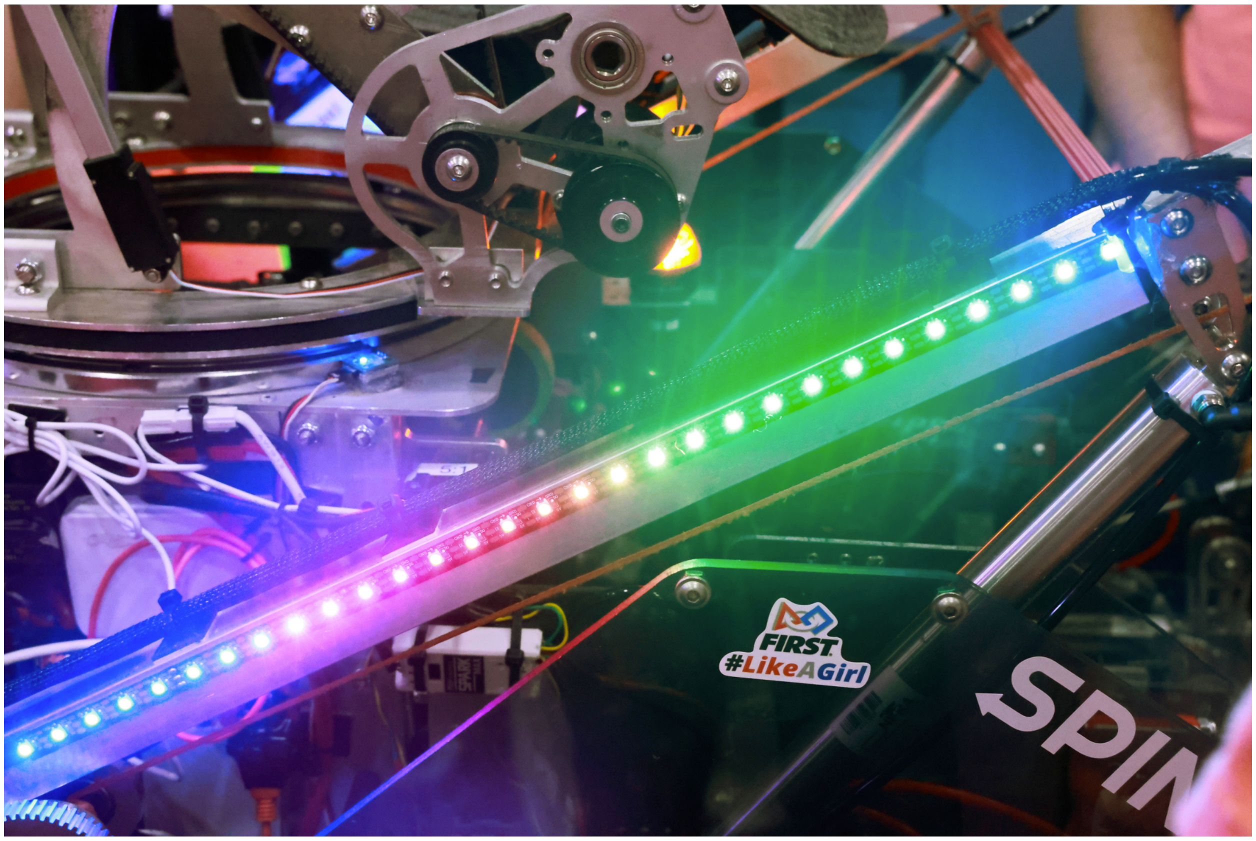


Our shooter used a flywheel driven by 2 Neos (later switched to Falcons) geared at a 2:1 ratio. The flywheel was sped up using the native PID controller and FF. Hood was controlled by linear servos capable of angle to hood between 15º and 50º. The target velocity and hood angle was determined by several factors. First, we used an interpolator to calculate the target speed based on distance from the Hub. Then we added in a fudge factor for the angle of the turret (shooting with the turret facing backwards didn't shoot as far by default because of our hopper system). We also adjusted the target position based on robot movement which enabled us to shoot on the move.

Our climber was added last minute so our software is still incomplete. For manual climb, we just use open loop voltage control with options for syncing both arms or individual control. We additionally added the ability to automatically extend the climb arms while driving to the hangar to increase the climb speed. Other features that were written but not extensively tested were reaction wheel stabilization (using the drivetrain and flywheel) to prevent the robot from swinging and using the gyroscope to keep the climb arms synced (in coordination with encoders).



We had 74 individually addressable LEDs that both looked cool and provided our driver information on the state of the robot. 14 LEDs on our shooter indicated issues with targeting and flywheel speed, and the 60 LEDs on the climb arms indicated the state of drivetrain and climber.



For controls we used a standard Xbox controller and had a single driver. We had a ControlSchema interface that could be selected for whoever was driving and allowed each person on the team the ability to choose the button map that was most intuitive to them. In addition to the basic controls, I also had the ability to manually control every motor on the robot and toggle many of the features listed above if they were having issues. This came in handy several times when we had issues on the field. For example, there were a few matches where we forgot to replace the battery. Despite operating on a faulty battery, I was able to keep the rio from browning out by disabling our compressor and a couple of the other motors. An additional way manual control came in handy was I could adjust all of the shooting compensation factors on the fly if something was going wrong with our shooter during the match.

All of these previously built features were built on top of our new library called Kyberlib. This took up 75% of the code I wrote and has no dependencies in the current robot code.

The main project of this library was the KMotorController class. This class wraps any encoded motor and provides simple methods and variables for controlling any type of motor. Wrapping motors like this made it a simple one line fix when we swapped out our Neos with Falcons on our flywheel. KMotorControl supports controls by:

- Percent output
- Voltage
- Follow another motor
- Position (angle or distance)
- Velocity (linear or angular)

- Motion Profiles
- Torque
- Force
- Current
- Music (falcon only)

It does this through builtin implementations of most FRC control algorithms including:

- PID
- FF
- Statespace (for linear systems)
- Bang Bang
- Custom lambda expressions

In addition to these algorithms, it has automatic support for simulated motors, whether actually in simulation, or simply not wired to the robot yet. This was an important feature for us given how little time we had for physical testing with the robot. Adding in simulation support allowed me to model the motion of what the motors should do based on their physical characteristics and develop some control options for rapid testing when I finally managed to do testing.

Another bonus feature I added into our motor library was a characterization system. This system can both do its own regression or export to SysId for visuals. I added this because of concerns of safety on constrained motors (like our turret) and better consistency on voltage controls (I don't think SysId runs voltage compensation during calibration which creates slightly off values).

For autonomous mode, Kyberlib supports several options. First, it automatically loads and stores all Pathweaver paths & routines located in the deploy directory. These routines are easily accessible through the TrajectoryManager Object. Second, we have a playback controller option that will record and save a period of controller inputs and simulate them back at the start of auto. Finally, I spent the first semester of this year implementing Trajectory generation with obstacle avoidance. This allows for autonomous navigation around obstacles (at the moment field obstacles, not robots) to get into any pose or position. This system involving using Informed Rapidly Expanding Random Tree* and was a really cool project to learn about. I also made a goal system that allows for the recording of objectives (such as cargo) that the robot can plan around and execute a command upon arrival (such as intaking). This system uses various Traveling Salesman algorithms I implemented as overkill.

As part of my efforts into my personal learning and preparing future programmers, I also made several template mechanisms for common FRC tasks such as drivetrains, elevators, arms, and flywheels. These offer both usable classes and stepping stones for future programmers to make their own variants of the algorithm that will better suit the nature of their competition. For example, I have written prototype code for Differential, swerve, and mecanum drive dynamics. These classes provide a standard interface for controlling robot movement, either through ChassisSpeeds or autonomous objectives. Additionally they are integrated into kyberlib's Navigator singleton.

Additional features included in Kyberlib are unit library with time, angle, length, and linear/angular speed to allow for more readable code and makes functions have better type safety; modifying joystick inputs with deadband & exponentials for fine control; a custom LED library that assigns animations to specific regions of the LED strip based on periodically checked conditions; a debug system that can print or display on NT important values with a settable debug level; 3d geometry classes; and polar coordinates and speed systems (really helpful with Rapid React)

One final thing I would like to mention is the Independent Study I did for my school last semester with the goal of making a fully autonomous mini-FRC robot. I had (limited) success with that project and it was really enlightening which helped me get started for this session and taught me a lot about robot localization, control theory, and pathing. Check out my blog here for more details on that.

Overall I would like to thank everyone in the FIRST community for the amazing system you have created. The heights you push high schoolers to achieve and the support you over is simply astounding. No matter the time of day or night, I could get someone knowledgeable to help me understand my problems. If you have any thoughts or comments for improvements in our code I would love to hear them!