Software

Development Process and Collaboration

The team utilizes Git for version control, and uses BitBucket as a hosting service, due to its more flexible implementation of repository forks over other solutions, such as GitHub. The programming lead works in the primary repository, and all junior programmers work in their own forks. Once a junior programmer completes a feature, a pull request is submitted to the primary repository with the relevant changes. These changes are reviewed by the programming lead and are then accepted or denied, based on the individual pull request. In the event of a denial, the programming lead communicates with the relevant junior programmer and discusses the needed changes.

Software Structure and System Modularity

Our codebase is broken up into two main packages, which each contain multiple packages of their own. The first main package is a "library" package, and contains classes that aren't game-specific, such as universal utility or navigation classes. The second main package contains Skystone-specific classes, such as subsystem classes and opmodes.

Furthermore, our codebase (primarily the game-specific component) is built around modularity and flexibility. Each subsystem is controlled by its own class, and they are integrated by a main "Robot" class. This allows us to easily enable, disable, or add new subsystems. Additionally, most of our sensor data is contained within a single class, which optimizes our loop times by minimizing unnecessary sensor queries. Rather than call a method to acquire data from the sensor directly, the class does this on every loop and stores that data. Then, that data is handed off to every system that needs it, which means that we don't have to request data multiple times from sensors that add functionality to multiple subsystems. Our sensor reads have also been optimized through the use of "bulk reads," which allow us to acquire encoder data from all of the encoders on a hub in one go, rather than sequentially, which significantly reduces loop times. Finally, our motors have been optimized to cache the previous motor power. This means that the motor will only receive a command if the new power is different from the old one (which might not be the case if power is being set every loop), which reduces congestion by eliminating unnecessary commands.

Sensors

We utilize a wide array of sensors on our robot for a multitude of tasks, both in autonomous and teleop. First, our drive train uses motor encoders on each wheel, which gives us precise velocity control in autonomous. This significantly improves our autonomous reliability and accuracy. However, we have a different solution for positional feedback and tracking. We use two "odometry" wheels (which are unpowered wheels sprung into the ground with encoders on them, resulting in a much more precise and accurate position estimate, due to the reduced wheel slippage) for positional tracking, as well as an inertial measurement unit for heading. We have a third odometry wheel as well, but we were unable to get it working consistently enough for our first competition. Additionally, we have an external encoder reading directly from our lift spool, which provides clearer data with less noise. We also have a hall effect sensor at the bottom of the lift, which allows us to zero the encoder once the lift is down. On our intake, we have a REV Robotics 2m time-of-flight distance sensor, which we use as a break beam to determine if we have a stone in the intake. We also have a color sensor mounted to the back of our hopper, which tells us if we have a stone in our hopper, enabling our deposit automation. Furthermore, the combination of these two sensors allows us to verify that we have collected a Skystone properly in autonomous. If we have a stone in our hopper and we don't have one in the intake, the autonomous proceeds as normal. If we have a stone in our intake and not in the hopper, we oscillate the intake to try and bring the stone in. If we have a stone in both our intake and hopper, we reverse the intake to eject the second stone. These steps ensure the likelihood of collecting a Skystone and reduce the chance of incurring penalties due to delivering multiple stones.

Computer Vision

We use computer vision (through OpenCV, an open source computer vision library) primarily to determine the Skystone location in autonomous. We use a custom OpenCV pipeline to accomplish this. This pipeline has been optimized for speed and accuracy, even in changing lighting conditions. To do this, we process the image in the YCbCr color space, rather than the more common RGB or HSV spaces. This is because the Cb and Cr channels represent blue and red-difference, which is very useful for this application, because we're comparing a near-black sticker to bright yellow. As such, the sticker stands out very strongly in the Cb channel. Furthermore, we only analyze a small region (about 25 square pixels in area per stone) of the frame, which eliminates external noise sources (such as another robot across the field). Additionally, we average the color in this region before analyzing it, which reduces the effect of things such as different lighting conditions. Due to these features, our detector has been incredibly consistent and very fast.

Computer Vision (continued)

We've also investigated the possibility of detecting stone distance and orientation (which is an essential component of distance) via the camera. Unfortunately, this feature is not ready for competition use, but we hope to use it in the future to increase our scoring capabilities in autonomous.

Navigation

Our drive is controlled using motion profiles. A motion profile is essentially a collection of "states" of the drive over time. These states contain position, velocity, and acceleration data (and potentially jerk, the derivative of acceleration, though this is unused in our navigation system due to the limited benefit seen in FTC). As such, plotting a motion profile will result in a graph of the robot's displacement, velocity, and acceleration. Furthermore, these profiles can be limited in both acceleration and velocity, which can reduce strain on the system and reduce wheel slippage when accelerating or decelerating, which is a significant issue for accuracy when using mecanum wheels. For two-dimensional motion, "paths" are used to describe the arbitrary positional data, without any time, velocity, or acceleration backing. Then, motion profiles are used to define paths over time. This combination of paths and motion profiles is referred to as a trajectory. These trajectories contain both two-dimensional position data and the associated state of the drive at any given point. This system results in a more accurate navigational solution than a simple PID controller, because the behavior throughout two points is defined, rather than up to the controller itself (as it would be with a PID that just aims to reach point B from point A). Furthermore, a few controller gains are used to approximate the behavior of the drive, which are collectively referred to as "feedforward control." This means that even before any feedback is added, the controller is able to reasonably follow a motion profile. Thus, the velocity and positional feedback controllers no longer have to do all of the heavy lifting, and instead just have to minimize the error between the feedforward terms and the ideal motion profile. This results in a more robust and responsive controller that is less prone to many of the traditional issues with PID controllers (such as inherent oscillation).

On the frontend, these paths are plotted using individual points, which are defined as (x, y, heading). This results in a relatively quick way to create new and adjust existing autonomous routes, especially once the controllers have been properly tuned.

Automation

We make extensive use of automation on our robot. Our automation philosophy is that we want the drivers to be able to focus less on the minutiae and more on decision making, but not at the expense of driver freedom. To accomplish this, our automated features are generally toggleable by the drivers (between manual and automated control), and the automated setpoints (such as the ones on our lift) usually have the ability to be finely adjusted, which allows the driver to compensate for non-nominal behavior of the robot. Our arm and clamp system is automated to grab a stone without driver input once the hopper's color sensor detects a stone. This means that the driver doesn't have to think about how to handle a stone once it is inside the robot and can instead focus on navigating back to the foundation to score it. Furthermore, our lift has automated setpoints, which are controlled by the secondary driver. These setpoints are manually moved up or down on a pre-determined interval (through a single button) as a stone is correctly placed. These setpoints can also be adjusted slightly if the lift has raised slightly too high or low, and the software will then remember that adjustment for the next cycle, unless the driver decides to clear the adjustment cache.