Central High School Sigma Programming

Brief History

The CHS Robotics team was founded in 2017. Programming for the team utilized the FTC blocks system, which consisted of connecting login blocks together to create an opmode. While this system is flexible enough for many teams, we found that our needs had outgrown its capabilities. In 2019, we switched to the FTC SDK in Java. While this opened up many new possibilities, it was overwhelming to adopt the new system at first. During that season, we learned how to integrate a basic autonomous workflow, and familiarized ourselves with the SDK. Familiarizing ourselves with this new system, as well as the new phyical components on the robot, namely the omnidirectional drive in place of a tank drive was our primary focus throughout the 2019-2020 season. As the 2020-2021 season began, we had more or less perfected the human controlled aspect of our software, and were quickly developing an approach to creating a cohesive autonomous system. These efforts culminated with the system that we have been using in the 2021-2022 season.

Introduction

As we entered the 2021-2022 season we first established our goals for the season, which began with a complete revision of our code's structure and organization. Previously we had integrated all of our code into a handful of highly condensed files. This year, we put more focus on making our code modular, making use of numerous packages, which each contained subclasses that acted as the backbone of our system. Moving forward, this would prove to be the most impactful change in the development of our code as a whole.

Physical Overview

The base upon which the robot is built is a drive train which consists of 4 omni-directional wheels in a square configuration 45 degrees off center. While this is not a popular configuration, it is functionally the same as the more popular mecanum drive. This is commonly called the holonomic drive, and it allows movement in any direction, even with simultaneous rotation. This allows us to perform advanced maneuvers in short periods of time. The two robots also share the same carousel-spinning wheel design, which consists of a single motor mounted vertically at the backs of the robots.

The Sigma robot features a linear slide as a lift mechanism, and an intake assembly consisting of a sheet metal scoop facing out the front of the robot, and a servo-powered finger to pull in and hold shipping elements.

Organizational Structure

The project is split into 2 distinct portions, one dedicated to our autonomous pipeline, and one dedicated to the comparatively much simpler driver control pipeline.

Autonomous

The following table outlines the structure of the autonomous portion of our project:

Package	Function
hardware	Classes responsible for managing the physical components of the robot, including motors, servos, and cameras
localization	Classes responsible for tracking the location of the robot through integrated encoders in the motors driving the wheels
waypoint	Contains classes that drive the system through which we pass movement instructions to our control engine

Package	Function
control	Contains the classes responsible for generating motor output based on values from localization and our target position for precise and accurate movement
actions	Contains the foundational code for the creation of all non-movement actions, as well as individual classes for each one of these actions
opmodes	Contains instances of FTC's OpMode class to interface with the FTC Driver Station app

The following table details classes that do not reside in any of the previously defined packages:

Class	Function
AutonCore	Unifies and executes autonomous components. This is the primary entry point
Constants	Contains numerical and boolean values that are used repeatedly throughout the code
Instructions	Manages the intersection between actions and waypoints. Used to create and modify instructions for autonomous programs

Localization The localization engine consists of an algorithm for finding the robots position on the field, which determines the position of the robot based on data from the motors. The orientation is provided by the on-board internal measurement unit (IMU).

The odometry algorithm gathers position data from the wheels through the following algorithm:

1. Gather information from the encoders integrated in the motors that drive the wheels. These values are measured in encoder ticks. There are a set number of encoder ticks per revolution of the motor.
2. Find the displacement of each wheel by taking the difference between the values read by the encoders and the values stored from the last iteration of the cycle, and multiply by a known constant (distance in mm traveled per encoder tick) to convert these values to millimeters
3. Take the mean of the displacement values to find the average displacement across all 4 wheels
4. Subtract the average displacement from all of the individual displacement values to remove the orientation of the robot from consideration in our original values
5. Translate the 4 wheel displacement values into 2 X/Y coordinates using vector addition
6. Find the orientation of the robot using the IMU
7. Take the orientation of the robot back into consideration for the final coordinate output

This process repeats several times per second to give the control engine up-to-date and accurate information in real time.

Navigation The algorithm for navigation converts a start-point and target-point pair (a `Waypoint`) into motor instructions for moving towards the target. This algorithm repeatedly runs and recalculates based on its measured position on the field, as calculated in the localization algorithm. Once the navigation algorithm is given a waypoint to execute, the following procedure is run:

1. Drive the robot to the starting point of the waypoint. This may seem redundant, but this is necessary for when the robot overshoots the previous waypoint, which could otherwise give rise to problems with accuracy
2. Collect and store the robots current position from the Localization pipeline
3. Pass that data into a PID controller, which adjusts the speed of the robot based on the distance between its position and the target. This accounts for the magnitude portion of our motion vector
4. Use the slope of the two points (target, and current position) to find the direction portion of the motion vector
5. Using the magnitude and direction values, calculate the power values for each motors
6. Using another PID, calculate the magnitude and direction of our heading

vector using an angle value from the IMU and a target value in radians
7. Add the outputs from the linear movement function and the rotation function to get the final motor power

This process repeats several times per second to ensure that the robot moves as accurately and precisely as possible.

The `Waypoint` class can also take 4 points as input, a start point, and end point, and 2 control points in between. Using these control points, we can calculate a cubic bezier curve for the robot to follow. In this case, the motion vector is calculated not by the distance and slope between two points. At the very beginning of the run, the approximate length of the curve is calculated. Then, at every iteration of the loop, the direction component of the motion vector is calculated by taking the derivative of the spline at a point `t` between 0 and 1. `t` is calculated at every iteration of the loop by dividing the total distance the robot has traveled since the beginning of the curve by the total arc length of the curve. Since this pipeline doesn't use a PID controller, its movements are only precise, but offer very little accuracy. To account for this, we add a linear move at the end of every spline to ensure we are exactly where we want to be before the next maneuver.

Driver Control

The Driver Control pipeline is split into just 2 simple classes. `Core` handles all of the hardware devices, and includes methods to interface with those devices. `Drive` Includes the same algorithm we use for calculating motion vectors and motor powers as autonomous, but takes joystick values as input instead of a set of points. It also contains code for controlling different features and peripheral devices on the robot, including a switch for actuating the intake, a switch for activating "Turbo Mode" which doubles the speed of the robot, and a PID controller for holding the lift in whatever position it is stopped at.