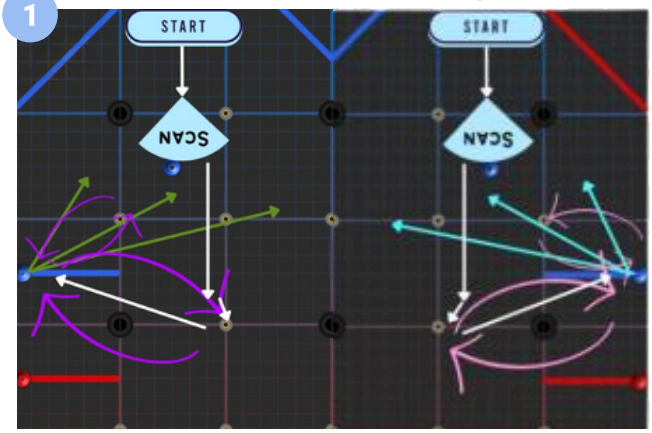
GAME STRATEGY

GOAL OF AUTONOMOUS

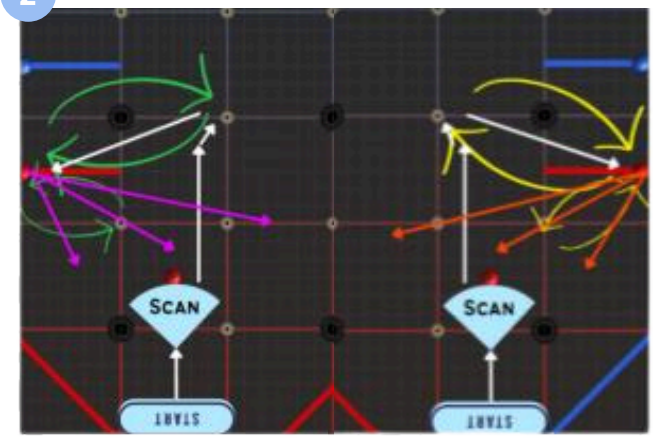
Empty our alliance cone stack before parking in the position detected by our custom signal sleeve.

1 Left Side Autonomous Diagram



The robot begins by on the left side of the playing field. It scans for the signal sleeve to determine which are to park in. It continues along the white path to the end, from which it begins pink/purple cycles. The robot then takes a teal/olive path to the designated parking area.

2 Right Side Autonomous Diagram

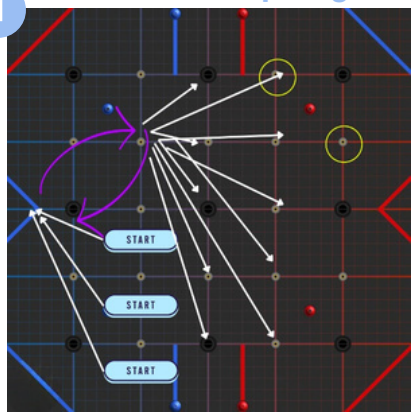


The robot begins by on the right side of the playing field. It scans for the signal sleeve to determine which are to park in. It continues along the white path to the end, from which it begins yellow/green cycles. The robot then takes an orange/magenta path to the designated parking area.

GOAL OF TELEOP

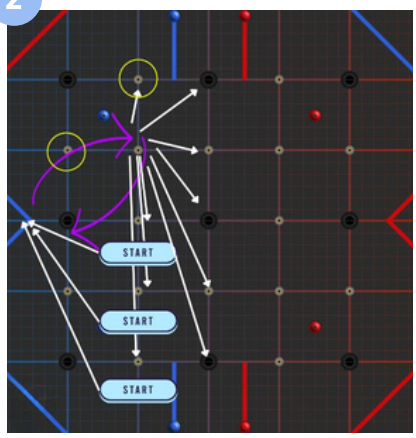
Our robot is versatile, allowing us to execute different strategies based on our alliance partner.

1 Offensive Tele-Op Diagram



The robot begins in one of three possible parking spots. It will head towards the substation and grab a cone. From there, it can take any of the white arrow paths or cycles depending on the situation of the game and how far into the game we are. The yellow spots are potential spots for defensive beacon placement.

2 Defensive Tele-Op Diagram



The robot begins in one of three possible parking spots. It will head towards the substation and grab a cone. From there, it can take any of the white arrow paths or cycles depending on the situation of the game and how far into the game we are. The yellow spots are potential spots for offensive beacon placement

*These cycles may be changed depending on the driver, state of the game, alliance, etc. They are built around the idea of turret adaptability.

AUTONOMOUS PERIOD CODE

ROAD RUNNER: A MOTION PROFILING LIBRARY

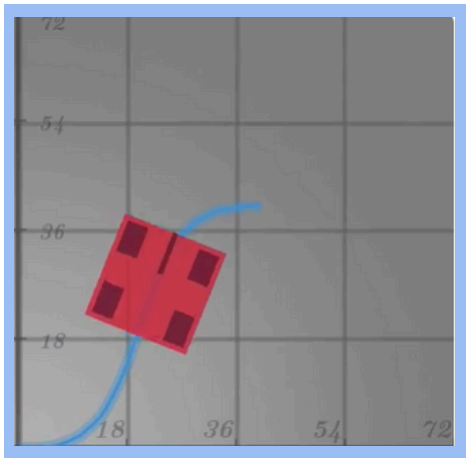
As of now we are now using the motion profiling library, roadrunner, to coordinate autonomous movement. We decided to move to this system in order to gain better control over the precise movement of our robot. By implementing roadrunner into our movement scheme, we have been able to take advantage of the curved paths, known as splines. Splines allow the robot to precisely move in a pre-programmed path called a trajectory which optimizes robot movement and allows the robot to move faster and more efficiently. The main functions that we are using for robot movement are .SplineTo and .AddMarker.

[AddMarker]

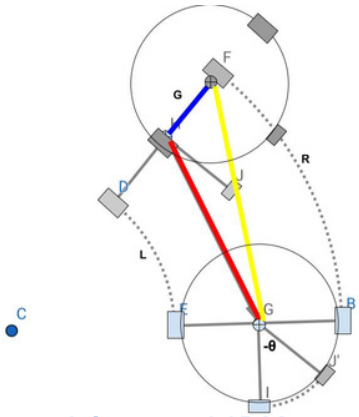
This function enables the robot to be able to perform certain tasks while driving along a trajectory. We use the AddMarker function to move the turret and slides during the autonomous period while the robot is driving to and from a junction during the cone stack cycle.

[SplineTo]

This function accepts parameters of an X,Y coordinate and a heading angle



Roadrunner Spline Path[SplineTo]



Odometry GCP Vectors

BACKUP GLOBAL COORDINATE POSITIONING

In the case the roadrunner does not function properly, we have a backup global coordinate positioning system. Using traditional odometry calculations, we can determine the position of the robot without the roadrunner library.

[dist]

Distance is derived from the input that is given by our odometry and global positioning function. Using odometry wheels, the inches that the robot has moved are calculated. With this function, we input the number of inches we want the robot to move.

[allowedDeviation]

The amount of error that the robot is allowed to move from the distance that it is set, oftentimes in order to minimize the amount of time that we can take we have to give a margin of error otherwise the robot will move back and forth.

GYROSCOPIC INTEGRATION

The function turnToAngle(); turns the robot to a specified angle from the robot's starting position. It keeps logs of the angles and changes in angles. In order to not overcompensate, the robot continuously moves clockwise and anticlockwise until it reaches within 2 degrees of the target, when it finally breaks out of the function. Without this failsafe, the robot may rotate too much due to the momentum of the rotation motion.

getAngleFromLastReset()

This function uses the last stored angle from the resetAngle(); function and subtracts it from the current angle.

resetAngle()

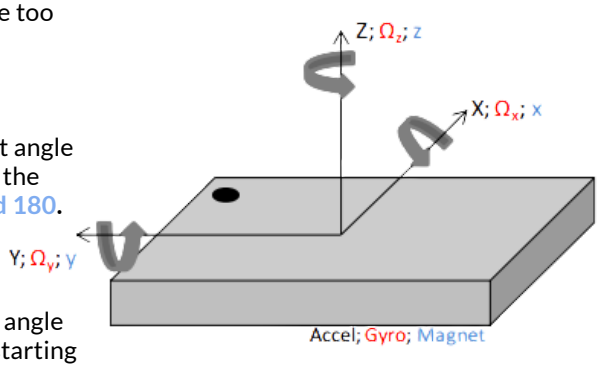
This function stores the final angle collected from getCurrentAngle(); and recalibrates the robot's gyroscope sensor.

normalizeAngle()

The function takes the current angle and adds or subtracts to keep the angle value between -180 and 180.

[getCurrentAngle()

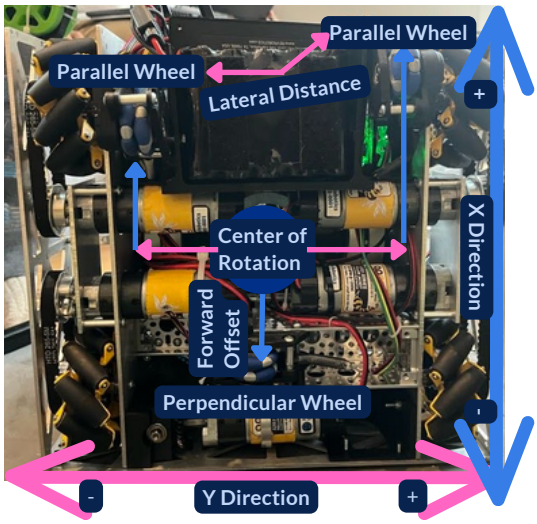
This function logs the present angle the robot is in opposed to its starting position, using the in-built gyroscope sensor in the REV Expansion Hub.



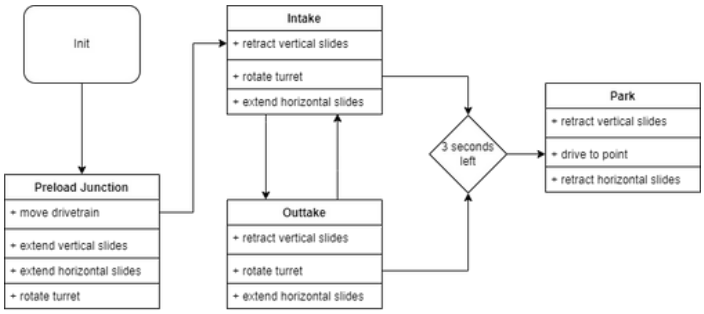
Rev Control Hub IMU

ODOMETRY/DEAD RECKONING

- The algorithm globalCoordinatePositionUpdate() which is implemented below uses inputs from the encoders which are placed inside of the free-spinning omni wheels on the sides and back of our robot.
- [leftChange and rightChange] - Both of these values are calculated by finding the differences between the current and previous position of each of the encoders.
- [changeInRobotOrientation] - This value is calculated by finding the difference between the change in position between the left and the right positions and then dividing that by the distance between the robot's encoder wheels.
- [Updating x Global positioning] - Takes the previous coordinate position and adds it to the quantity of sum of the average change between the left and the right encoder multiplied by the sine of the robot's angle and the robot's horizontal encoder's change multiplied by the cosine of the robot's angle.
- [Updating y Global positioning] - This is the same as the previous value except, the sine and cosine are switched.



3-Wheel Odometry



Finite State Machine Flow Chart

FINITE STATE MACHINES

We used finite state machines in autonomous to organize asynchronous actions based on the current state of auton. These states correspond to the following actions during the autonomous round: Navigating to the cone stack/high junction, Scoring the preloaded cone, Intaking from the cone stack, and Scoring with a cone from the cone stack. These states are continuously run until the end of auton, at which time the robot will park in the indicated signal zone.

OPENCV

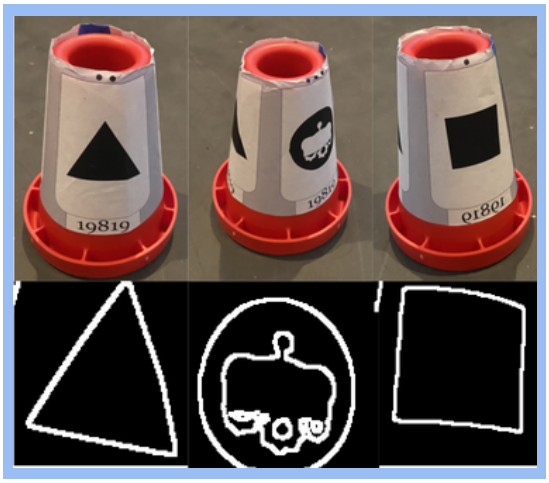
One of the first things in autonomous is to detect the side of the signal sleeve that is randomly pointed towards the robot, which in our case is a square, triangle, or AstroBruins logo. This determines end parking spot, and thus the auton path that follows. In order to effectively find the pattern, we decided to use OpenCV or computer vision coupled with an off-the-shelf webcam in order to correctly identify the image.

The OpenCV Class

In order to determine the number of vertices, which in turn determines the side of the signal sleeve facing the robot, we first apply a Gaussian blur to the image to cut down background noise. Then, we identify the contours of the image, searching for sharp turns to count vertices. An area threshold is applied to all shapes to further eliminate noise, with objects that do not meet the threshold disregarded, and the remaining object (signal sleeve shape) has its vertices counted - 4 vertices indicates the square, 3 indicates the triangle, and any other number would indicate the AstroBruins logo.

Major Functions

- GetSignalPosition - At the beginning of gameplay, this function scans which side of the signal sleeve is facing the robot, determining the parking area at the end of the autonomous period

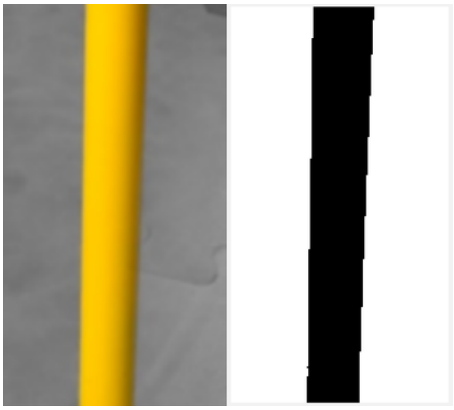


Signal Sleeve Detection with OpenCV

TELEOP ASSISTANCE

AstroCompass

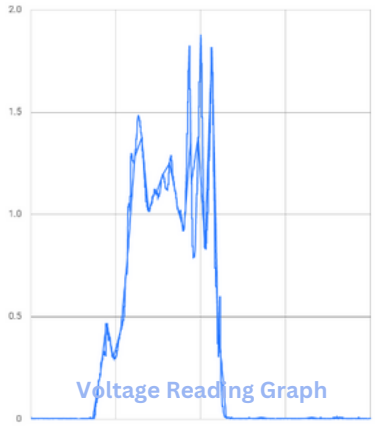
AstroCompass is a class that contains the function TurnToJunction. TurntoJunction is a custom pipeline developed using OpenCV to help with ease of use for drivers. Because yellow has the least amount of Blue, we can simply check the RGB value for the lowest B value. Once the pole is identified, we use a series of image processing functions to get the number of pixels on the left and right sides of the pole. Using this information, we can automatically align the robot with the pole, reducing the margin of error greatly. The second driver can then extend the vertical/horizontal slides and drop the cone onto the targeted pole.



Junction Image Processing using OpenCV

VOLTAGE LIMITER

Due to the fact that our horizontal slides extend too far upwards to use a conventional encoder wire, we used the built in voltage reader in the motor to prevent over extension and retraction. When voltage spikes past a certain threshold, horizontal extension and retraction are locked, preventing breakages.



Voltage Reading Graph

SLIDE ENCODERS

In order to prevent over-extension or retraction of our linear slides, we use slide encoders to get the exact position of the slides. When this position passes a certain threshold (fully extended/retraced), measured in ticks, the slide motors are stopped and the driver can no longer extend or retract, depending on the position the slides are in.

LESSONS LEARNED

The most important lessons learned by software this year came in many forms. From a technical perspective, we greatly expanded our knowledge in finite state machines and CV pipelines, which we developed through constant iteration over the course of the season.

Beyond that, we learned that communication, both within the software subteam and the greater AstroBruins team as a whole, is integral to success. To emphasize this aspect of software, we made sure to regularly consult both the build and drive teams to make sure our goals and implementations aligned with the needs of the entire team.

CONTROLLER 1

Driver/Reg Control Switch

Rotation of Turret

Movement of Robot

Robot Rotation

CONTROLLER 2

Align Turret w/ Junction

Intake System (Open/Close)

Horizontals Sensitive

Horizontal Slides Control

Power of Linear Slide/Turret Motors

Vertical Slides Extension

Vertical Slides Retraction