Error 404 3735C Code Explanation

Overview

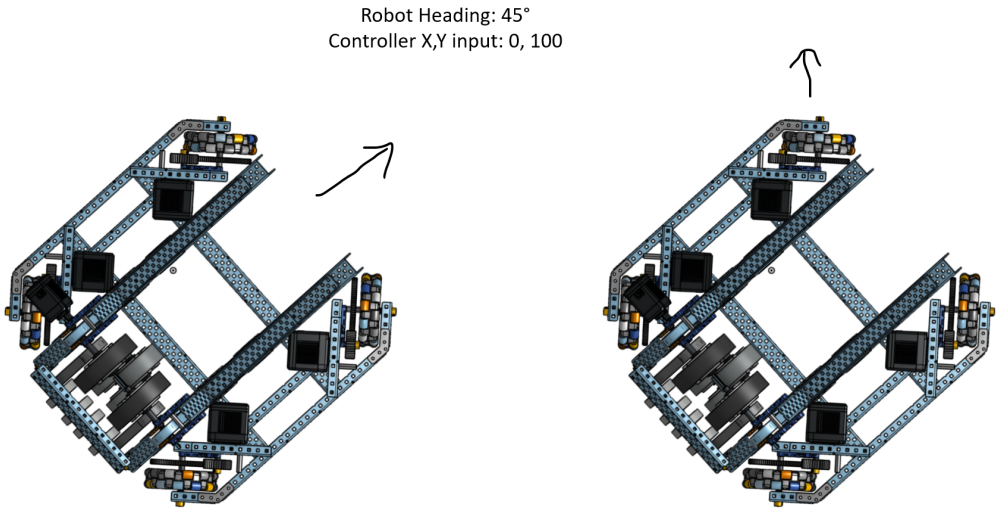
There are a few interesting bits of code I would like to explain. I'll divide them into three categories:

1. X-Drive Logic,
2. Shooting (flywheel, auto aiming)
3. Auton (wheel odometry)

First, a few miscellaneous points. Our school does not let us use VSCode or any third-party libraries, meaning we use the base V5 Pro code editor and have built all of our code from scratch. Also, we've saved all of our work and have it all documented through our GitHub commit history. Our code is open source and available for all to view.

X-Drive Logic

Our robot uses an x-drive drivetrain, which, though not incredibly common, already has pretty standardized code. This drivetrain can go forward, backwards, and strafe side-to-side. Our x-drive is unique because it is field oriented; it's the closest you can get to running a swerve drive on a vex bot. Field oriented driving means that, no matter what direction the robot is facing, it goes forward relative to you/the field, not itself. So if the robot is facing left, and you tell it to go forward, it'll strafe to its right.



(Outdated CAD btw)

Code is below:

```
double headingRadians = Inertial2.heading() * 3.14159/180;
double yPos = con1.Axis3.position(pct);
double xPos = con1.Axis4.position(pct);
double sineHeading = sin(headingRadians);
double cosHeading = cos(headingRadians);
// rotate the controller x/y coordinates by the negative of the robot
heading
// robot is facing 30 degrees and we tell it to go forward, then it needs
to strafe -30 degrees

// this is some linear algebra
double rotatedYPos = xPos * sineHeading + yPos * cosHeading;
double rotatedXPos = xPos * cosHeading - yPos * sineHeading;
```

Shooting

Shooting I'll further divide into two categories. The first is the flywheel code, and the second is our auto-aiming code.

First I'll explain our flywheel code. Our robot has the ability to shoot from anywhere on the field, since, using the vision sensor, we're able to estimate how far our robot is from the goal and the adjust the flywheel speed accordingly. This is an advantage over catapults, which have a fixed place on the field they have to launch from. We discovered that Vex's built in 'speed' methods don't actually directly control the motors. Instead, our speed input is first run through a PID controller that isn't tuned for our flywheel and gives an output that makes

the flywheel velocity oscillate. Since the speed wasn't consistent, we weren't able to score shots consistently. We found that, to bypass this, we can instead use voltage, which ignores the preset, badly tuned PID controller entirely.

```
// Vex internal PID means this code isn't turned correctly for our flywheel
vex::motor::spin(directionType::fwd, 100, velocityUnits::pct)

// Controlling the flywheel motors with voltage bypasses the built-in Vex PID
vex::motor::spin(directionType::fwd, 12, voltageUnits::volt)
```

Secondly, our robot auto-aims onto the goals. The code for this is really quite simple; all we have to do check the goal's offset from our vision sensor's center and run that through a PID controller. Then, our robot turns by that amount. The auto aiming is more consistent with the red goal than the blue goal. I won't explain in detail why, but in brief, the blue goal is dark, and there are a lot of common colors (like black) that the vision sensor confused with the blue goal. At TSA, where everyone is wearing blue shirts, we will have to manually aim the robot onto the goal.

```
// get middle of targetted object
// The vision sensor grabs the top left corner of the target, so we have to do a little bit
// of math to convert that to the target's center.
int targetMid = VisionSensor.largestObject.originX + (VisionSensor.largestObject.width / 2);
int error = screenCenter - targetMid;
// 'goal' is our PID controller
goal.setValues(0.2, 0.003, 0, targetMid);
int turning;

// i bound, only accumulates i if close enough to the goal
if ( error < 20 ) {
    turning = goal.getOutput(screenCenter, 0, true, targetMid);
} else {
    goal.resetError();
    turning = goal.getOutput(screenCenter, 0, false, targetMid);
}
```

Autonomous Odometry

Since we don't have distance sensors or a GPS sensor, our robot runs its autonomous routine based on wheel odometry. Odometry is just a fancy word for saying that we estimate the robot's position and movements based off of the degrees each wheel has turned. Combined with our inertial sensor, we've implemented easy-to-use methods that make our auton routine easy to program. For example, the goTo method takes three parameters: the y

to drive, x to drive, and the time it should take. So, if I want the robot to stafe 5 inches to the right in 1 second, I just have to call:

```
goTo(0, 5, 1);
```

Similar methods have been implemented for turning. Most important is the turnToAbsolute method. If we want the robot to turn to a very precise heading, wheel odometry can be inconsistent because of a combination of gear and motor play. Instead, we can get much more consistent results with the inertial sensor, which tells us exactly where the robot is facing. The turnToAbsolute method turns to a specified angle from 0-359, regardless of where the robot is facing when the method is called (which would be turnToRelative). By doing some fancy math, the robot calculates the fastest way to turn to the specfied angle and goes there. We don't have to worry about robot inconsistency this way, since the inertial sensor always knows where it is and where it's facing.

Here's an example below:

```
void autonR(void) {
    AutonCommands::goTo(0,16,0.7);
    AutonCommands::goTo(-2,0,0.1);
    AutonCommands::spinUpFlywheel(12);
    AutonCommands::starting();
    AutonCommands::doRollerfast();
    AutonCommands::goTo(2.5,0,0.3);
    AutonCommands::turnToAbsolute(330, 60);
    AutonCommands::turnToAbsolute(335, 20);
    AutonCommands::shoot1(1.5,13,2);
    AutonCommands::turnToAbsolute(190, 80);
    AutonCommands::turnToAbsolute(200, 20);
    AutonCommands::spinIntake();
    AutonCommands::goTo(-60,0,1);
    AutonCommands::spinUpFlywheel(10);
    AutonCommands::turnToAbsolute(295, 80);
    AutonCommands::shoot1(1.75,11,3);
    AutonCommands::stopIntake();
}
```

This is as close as text code can get to block code. Sometimes, programming members aren't there to code the bot, meaning a mechanical member has to help out with the auton. Generally, the separate teams keep with their roles; however, since these methods are so easy to implement, our mech member was able to successfully code part of our skills auton routine by themselves. All of our code has comments and is all sufficiently documented to the point where anybody on the team can use it.

Wheel odometry isn't actually too difficult to calculate. We convert from wheel diameter and gear ratio to circumference to rotations based on how far the robot is supposed to drive, and do a few more calculations to determine the speed at which each wheel's motor should turn at per second (since we're strafing to x/y, the wheels are not necessarily turning at the same speed). The most difficult part about it is that an x-drive has wheels that are vectored

outwards at 45 degrees. This means that to drive forward, we can't directly calculate how far a normal wheel would drive. By doing a little bit of trigonometry, we can solve for how much we need to scale the driving by, the square root of two. Without overcomplicating it too much, we use the special 45 45 90 triangle identity and solve for the hypotenuse.

Code is below:

```
double WHEEL_DIAMETER = 4; // inches
double CIRCUMFERENCE = 3.14159 * WHEEL_DIAMETER;


// 84 teeth on wheels and 36 on motors
// also 7/3 gear ratio
// changed gear ratio back to 1:1
double GEAR_RATIO = 1/1;
double x_rotations = (xToGo / CIRCUMFERENCE) * GEAR_RATIO;
double y_rotations = (yToGo / CIRCUMFERENCE) * GEAR_RATIO;
/*
*     robot wheels looks like this:
*   /   \
*   \   /
*
*   they vector outward at 45 degrees each
*  however, we want to drive forward a certain amount of inches
* not driving 45 degrees in a direction
* if you extend everything out then you get a square
* with a line going from one corner to another that is as far as you want
* your robot to drive.
* You get a 45 45 90 triangle, where the legs are each x and the
hypotenuse is x root 2
* that means each of the legs is the distance you want to go divided by
root 2
* so we have each of the robot motors turn that far instead
*
* then we multiply by 1.1 to overshoot a little bit because our motors
have play
*/
double xDegrees = x_rotations * 360 / sqrt(2) * 1.1;
double yDegrees = y_rotations * 360  / sqrt(2) * 1.1;

// different direction where the wheels are pointing so different degrees
to turn
double front_left_degrees = xDegrees + yDegrees;
double front_right_degrees = xDegrees - yDegrees;
double back_left_degrees = xDegrees - yDegrees;
double back_right_degrees = xDegrees + yDegrees;

double front_left_degrees_per_second =
front_left_degrees/secondsToComplete;
double front_right_degrees_per_second =
front_right_degrees/secondsToComplete;
```

```
double back_left_degrees_per_second = back_left_degrees/secondsToComplete;
double back_right_degrees_per_second =
back_right_degrees/secondsToComplete;
```