FTC WI.R.E.S Software Platform for Centerstage (2023-24 season)

Instructions document

Released on 10/26/2023, Update on 11/13/2023

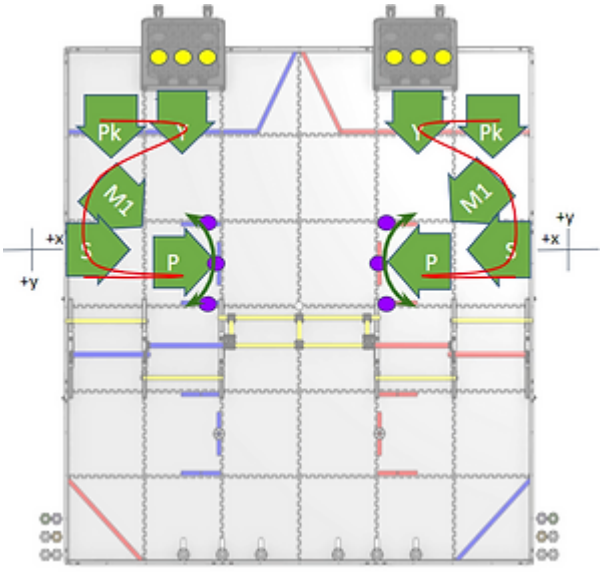
For more information, check
https://www.ftcwires.org/softwareplatform

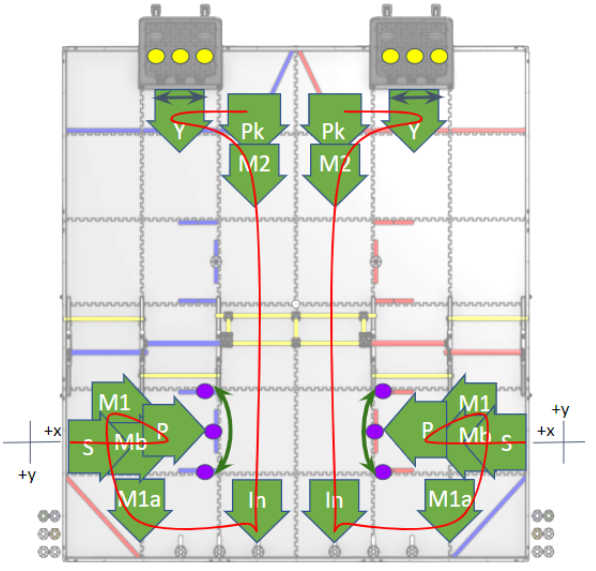
or contact : ftcwires@gmail.com

ftcwires Software Platform for easy Autonomous coding - designed for Rookie teams to have a good Autonomous mode at their first qualifier itself!

FTC WIRES Software Platform for CENTERSTAGE with 9.0.1 and RR 1.8 is now released

Integrates New Roadrunner 1.8 for motion planning , Vision library for Pixel detection, Sample code for autonomous path.





Intention: This platform is targeted to be used by rookie teams or teams who are learning autonomous programming. The aim is have all teams in Wisconsin have a basic autonomous mode working before their qualifiers.

Why: During the FTC WIRES survey in 2021-22 season, it was observed that many of the rookie teams and newer teams did not have a working autonomous mode in the early qualifiers. This was a demotivator for the teams as well as their alliance partners. This platform should ease the process of building a good and working autonomous mode in 1-2 days. We released the first FTC WIRES Software Platform in 2022-23 Powerplay season. Most Wisconsin teams had an autonomous mode in the year, The platform was subscribed by more than 70 teams worldwide too. Based on request, we are releasing the new version of FTCWIRES Software Platform for Centerstage 2023-24 Season

Acknowledgement: Acmerobotics and Ryan Brott for providing the Roadrunner library, Noah Bres for Learnroadrunner.com and Team Hazmat 13201 who created the FTC Wires platform

Note: The document may be 35 pages. But most of it is pictures and explanations!. The actual code change is not more than 25 lines, and effort could be completed in 2 hrs.

Table of Contents:

Update 11/13/2023

Step 11.1 introduced - Only for Dead Wheel Encoders, Need to update tick count for parallel and perp encoder position.

What does the platform contain :

- The platform is a fork from Road-runner-Quickstart based off of FtcRobotController SDK 9.0.1 released by FIRST and can accessed at https://github.com/ftcwires/centerstage-road-runner
- Roadrunner (Rev 1.8) is the newly released version of the motion planning library developed by Acmerobotics (Ryan Brott) . Designed primarily for autonomous robotic movement, it allows for complex path following and generation while maintaining control of velocity and acceleration. This enables bots to have more accurate and advanced path following capabilities. We are going to use Drive Encoder based odometry. (Detailed information on this is available at https://rr.brott.dev/ and https://github.com/acmerobotics/road-runner-quickstart but the idea here is to help teams who find those pages overwhelming, so you don't need to look!)
- It also includes integration of the Vision Portal TensorFlow detection of the white pixel (default one, not of your customized team game element), to find the spikemark to drop purple pixel and the location to drop Yellow pixel on Backdrop in autonomous mode. This code is derived from the `ConceptTensorFlowObjectDetection.java` provided as an example in the FTC sdk.
- Using these, an example Autonomous mode for Centerstage is implemented. You could modify this easily to develop your own autonomous mode This also includes a simple tuning process for Roadrunner, as well as easy way to program positions for autonomous modes based on robot centric coordinate system.
- This also includes a sample version of TeleOp with motion management based on Roadrunner.

Assumptions :

- You have a robot that uses mecanum wheels with the motor encoders connected for odometry (80% of FTC teams use this). We call this Drive Encoder based odometry
- You should also have a webcam connected and positioned in a way to see the pixel on the spike mark
- The robot design assumes pick up of pixels from the front of the robot and drop of pixels on backdrop from back of the robot. (If you have a different orientation, all you need is to change the position coordinates)
- Your robot would need to add the code for mechanism to drop purple pixel, ability to intake pixels and ability to drop pixels on backstage or backdrop.
- You have a minimum understanding of Java and using the sdk, and coding using android studio.

If you have "No" on one of the assumptions, you could still use it :

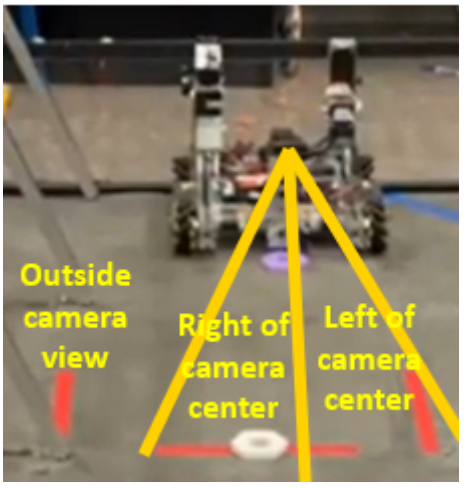
- If you are using block programming, you could always finish your teleop mode in block programming for all your non-drive systems and then transition to this platform, get to the java code and add to this.
- If you are using tank drive, you still could use this, but have to figure out how roadrunner works with tank drive from the tuning docs of Roadrunner
- If you want to use dead wheel encoders, you can still use this. Just need to make the localizer changes and do tuning (based on roadrunner docs). We have not verified if all the instructions are accurate for this.

Disclaimer :

- This is a basic autonomous mode - better than a crude one, but it is certainly not the best.
- This only uses minimal roadrunner capability in terms of motion profiles possible. Roadrunner has several additional motion profiles to use, ability to time actions in parallel, sequential, stop and start, etc. which is not used here.
- Roadrunner also provides the ability to visualize motion on a digital dashboard. This version of the program won't support it, since the coordinate system assumed (for simplicity) is based on the starting position of the robot. Dashboard requires a field centric coordinate system.
- If you feel like you will miss the fun of discovering how to code the autonomous mode the hard way - don’t look, this is just to make the journey easy for those who want to start with an example.

How does the FTCWires Autonomous Mode work :

The sample code provided gives an implementation of the autonomous mode based on instructions in the Game Manual 2. There are 4 modes to select - based on the starting location of the robot (Red Left, Red Right, Blue Left, Blue Right). The starting point of the robot is assumed to be on the starting tile, and along the edge farther from the truss legs. You should also have a webcam connected and positioned in a way to see the middle spike mark and the spike mark away from the truss (and ideally nothing else). We assumed the camera to be in the center of the robot.

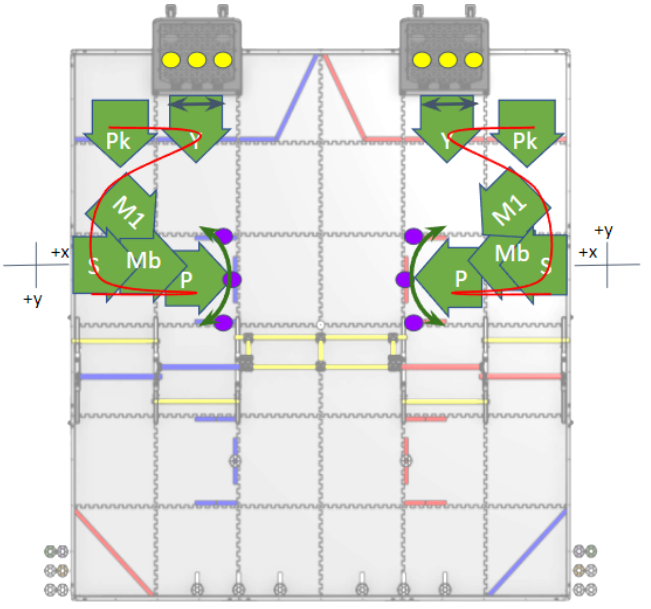


Blue Left and Red Right Autonomous mode

1. Robot starts in the position S marked in the picture (aligned to tile edge farther from the truss)
2. On Init, The robot vision is started and it identifies the white pixel in the spike mark (Left, Middle or Right).
3. On Start, the robot first moves to position Mb (to avoid hitting the truss legs) and then to the spike mark detected by vision (position P). The code for dropping purple pixel needs to be executed at this point.
4. Robot then moves back to M1 (to avoid the purple pixel) and moves to position Y (based on the spike mark detected). The code for dropping Yellow pixel on the back board needs to be executed at this point.
5. Robot then moves to parking position Pk.

BLUE LEFT

Start S = 0, 0, 0d
MoveBeyondTruss Mb = 15, 0, 0d
Purple Pixel P =
  Left 26, 8, 0d
  Middle 30, 3, 0d
  Right 30, -9, -45d
Midway M1 = 14, 13, -45d
Yellow Pixel Y =
  Left 23, 36, -90d
  Middle 30, 36, -90d
  Right 37, 36, -90d
Park Pk = 8, 30, -90d



RED RIGHT

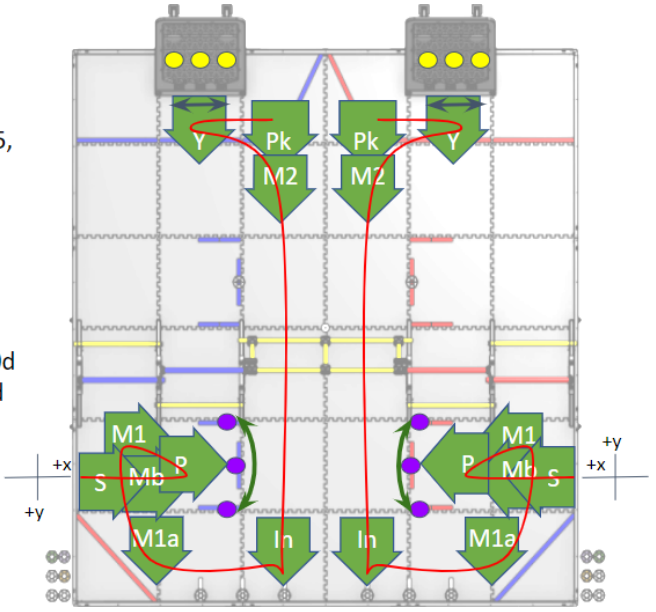
Start S = 0, 0, 0d
MoveBeyondTruss Mb = 15, 0, 0d
Purple Pixel P =
  Left 30, 9, 45d
  Middle 30, 3, 0d
  Right 26, -8, 0d
Midway M1 = 14, -13, 45d
Yellow Pixel Y =
  Left 21, -36, 90d
  Middle 30, -36, 90d
  Right 37, -36, 90d
Park Pk = 8, -30, 90d

Blue Right and Red Left Autonomous mode

1. Robot starts in the position S marked in the picture (aligned to tile edge farther from the truss)
2. On Init, The robot vision is started and it identifies the white pixel in the spike mark (Left, Middle or Right).
3. On Start, the robot first moves to position Mb (to avoid hitting the truss legs) and then to the spike mark detected by vision (position P). The code for dropping purple pixel needs to be executed at this point.
4. Robot then moves back to M1 and M1a (to avoid the purple pixel) and moves to position in front of the pixel stack. Code for picking a pixel needs to be added here.
5. Robot then moves to position M2 (through the central path. If the stage door needs to be opened for the robot to pass, code needs to be added for it.
6. Robot then moves to position Y (based on the spike mark detected). The code for dropping Yellow pixel on the back board needs to be executed at this point.
7. Robot then moves to parking position Pk.

BLUE RIGHT

Start S = 0, 0, 0d
MoveBeyondTruss Mb = 15, 0, 0d
Purple Pixel P =
  Left 27, 9, 45d
  Middle 30, -3, 0d
  Right 26, -8, 0d
Midway M1 = 8, -8, 0d
Midway M1a = 18, -18, -90d
Intake Stack = 52, -19, -90d
Midway M2 = 52, 62, -90d
Yellow Pixel Y =
  Left 27, 86, -90d
  Middle 34, 86, -90d
  Right 43, 86, -90d
Park Pk = 50, 84, -90d



RED LEFT

Start S = 0, 0, 0d
MoveBeyondTruss Mb = 15, 0, 0d
Purple Pixel P =
  Left 26, 8, 0d
  Middle 30, -3, 0d
  Right 27, -9, -45d
Midway M1 = 8, 8, 0d
Midway M1a = 18, 18, 90d
Intake Stack = 52, 19, -90d
Midway M2 = 52, -62, 90d
Yellow Pixel Y =
  Left 27, 86, -90d
  Middle 34, 86, -90d
  Right 43, 86, -90d
Park Pk = 50, -84, 90d

Instructions for Setup:

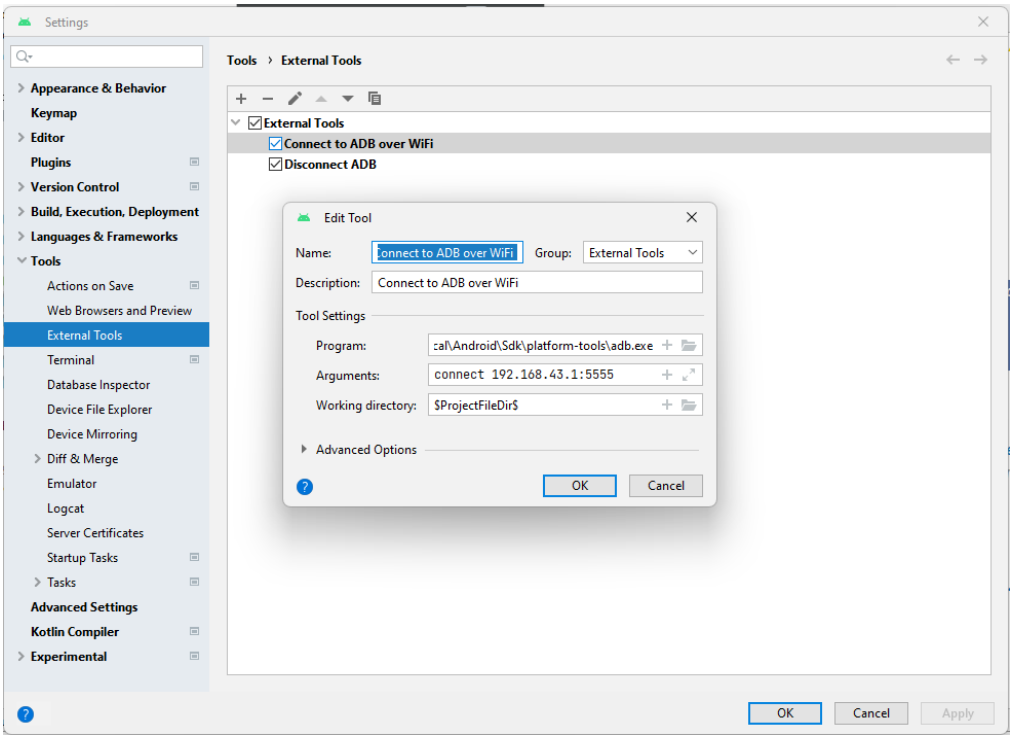
1. The code for FTC WIRES Software platform is at :
    - https://github.com/ftcwires/centerstage-road-runner
    - Go to the page. If you don't have a github account, create one and login to see this.

2. Click on the green Code button and Download zip file and uncompress on your computer.
    - If you are comfortable with git, you should create a new fork of the project https://github.com/ftcwires/centerstage-road-runner.git . Will help in keeping sync in case FtcRobotController project or roadrunner is updated.

3. Create a new project on Android Studio with this downloaded code (or clone the git project).
    - Ensure you are connected to the internet and click on File -> Sync Project with Gradle Files.
    - Follow any instructions that Android suggests to upgrade Android versions.
    - Click on Build -> Make Project and see that it completes successfully.
    - Congratulations you have the set up ready
    - If you have trouble setting up AndroidStudio, please refer to https://ftc-docs.firstinspires.org/en/latest/programming_resources/android_studio_java/Android-Studio-Tutorial.html. If you are still having trouble, reach out to us at ftcwires@gmail.com

Connect the Rev Control Hub over wifi :

To create a wireless connection from Android Studio to the robot, Go to File-> Settings and on the popup window. Go to Tools->External Tools. Click on + and create a tool code as in picture below:



Name : `Connect to ADB over WiFi`
Description : `Connect to ADB over WiFi`
Tool Settings:
Programs:
`C:\Users\<your_user_folder>\AppData\Local\Android\Sdk\platform-tools\adb.exe`
Arguments : `connect 192.168.43.1:5555`
Working Directory : `$ProjectFileDir$`

Once you click Click on Tools -> External Tools and you will see a button for Connect to ADB over Wifi. Connect your laptop to the WIfi network of your Rev Control Hub. It is best to have a USB wifi adapter on your laptop. This way, your laptop's main wifi can be connected to the internet, while the USB wifi can connect to your Rev Control Hub.

Download Code to Rev Control Hub:

Once connected, You will see the Rev Controller showing up as a connected device in the top bar of Android Studio. Click on Run (Green Arrow) button to download the built code to the Rev Control Hub.

You should be able to see the Opmodes `FTC Wires Autonomous Mode` and `FTC Wires TeleOp`, along with a few other Opmodes.

Congratulations! Your setup is complete!

Configure and Tune the robot

We assumed a robot design that has 4 mecanum wheels, and has a mechanism to pick up pixels from the front and drop the pixels on the backdrop from the back of the robot. (If you have a different orientation, all you need is to change the position coordinates). We also assumed the robot has a webcam in the front center and at a height of 7-8 inches from the ground, pointed in a way to “see” the spike mark from the starting position of the robot.

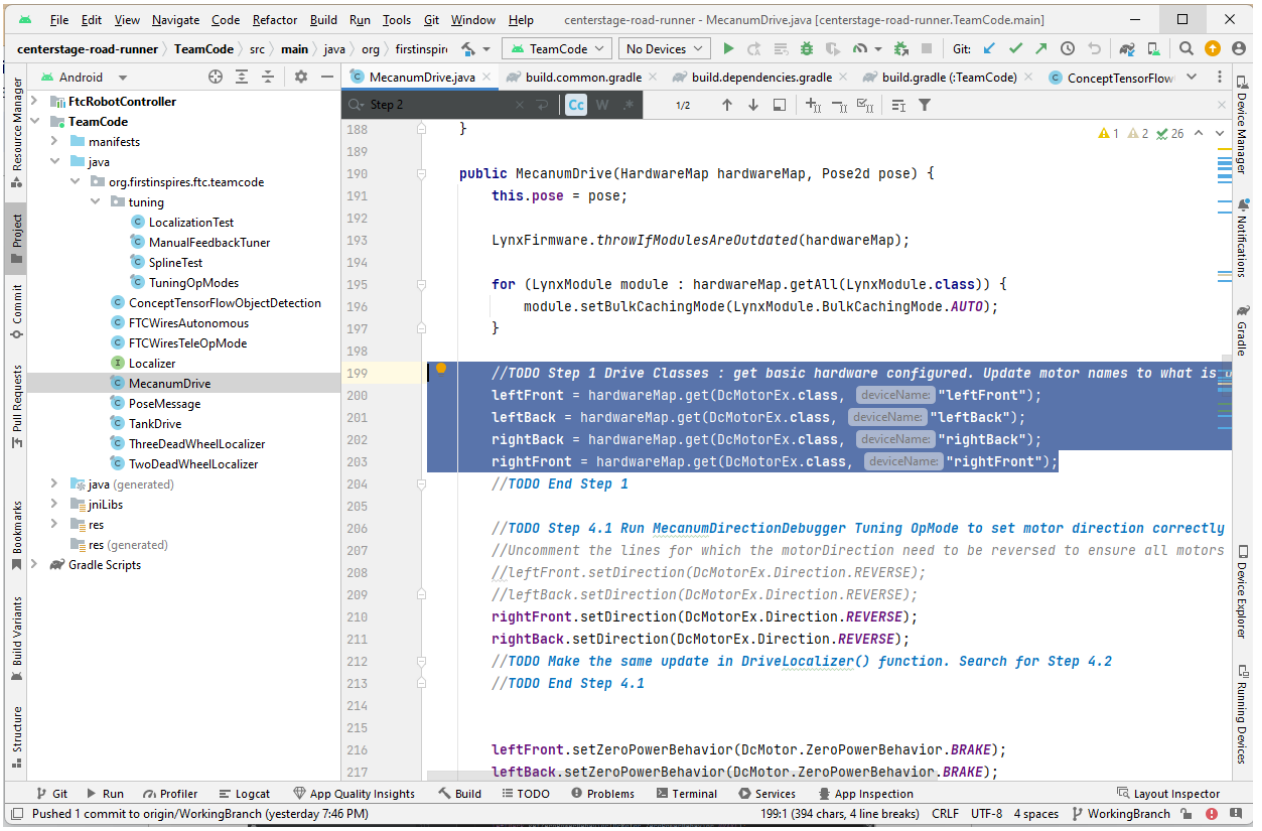
Create a configuration on the Driver Hub with the 4 motors named as `leftFront, leftBack, rightBack, rightFront`. The webcam should be configured as `Webcam 1`.

Tuning the Robot

Each robot is different, and Road Runner needs to be calibrated to your robot. The process consists of running several op modes and either adjusting parameters manually depending on the response or automatically fitting parameters to data collected. The entire activity shouldn’t take more than two hours.

Step 1: Motor names in MecanumDrive Class in `MecanumDrive.java`

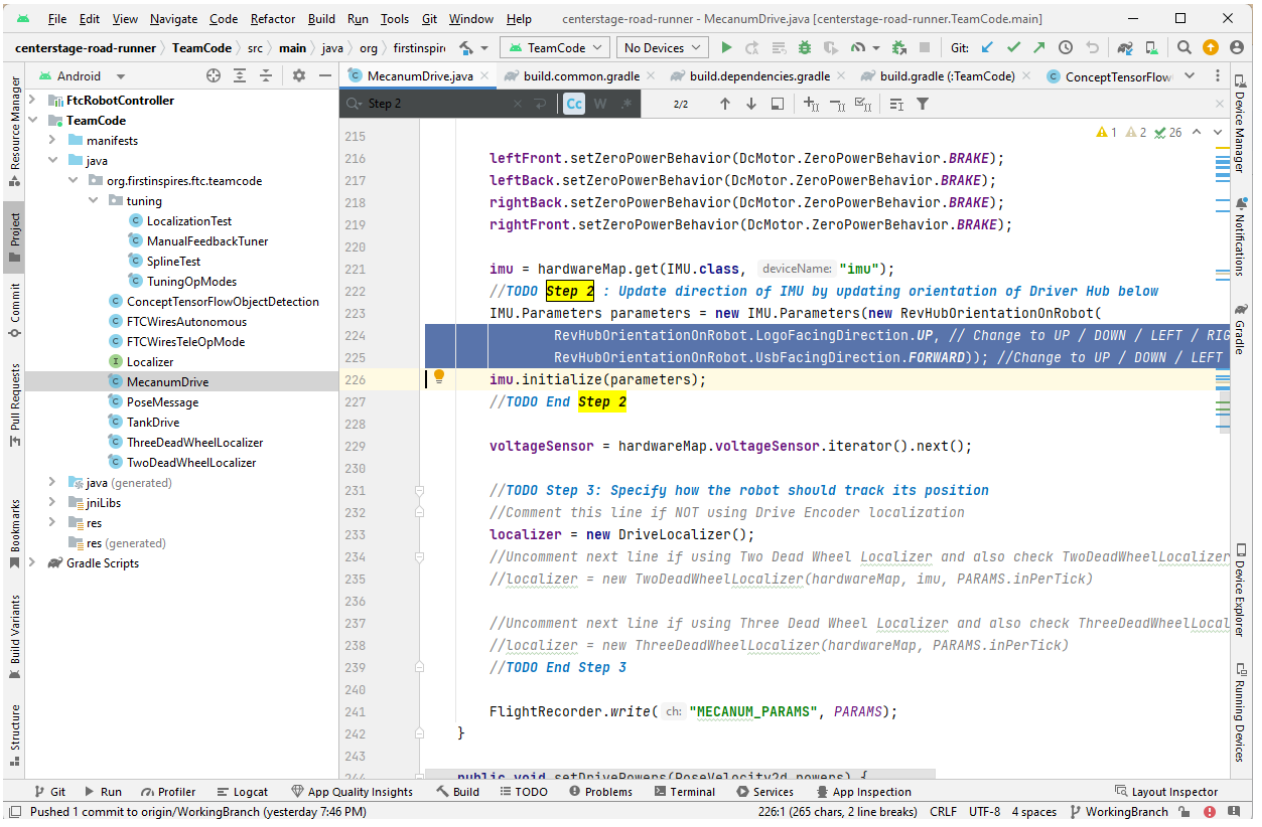
- Open `MecanumDrive.java` on Android Studio under `teamcode` package:



- Search for “`TODO Step 1`”. Make sure the motor names listed match the motor names in your robot configuration. "`leftFront`", "`leftBack`", "`rightBack`", and "`rightFront`"
- Remove the TODO word in the comment as you complete the step.

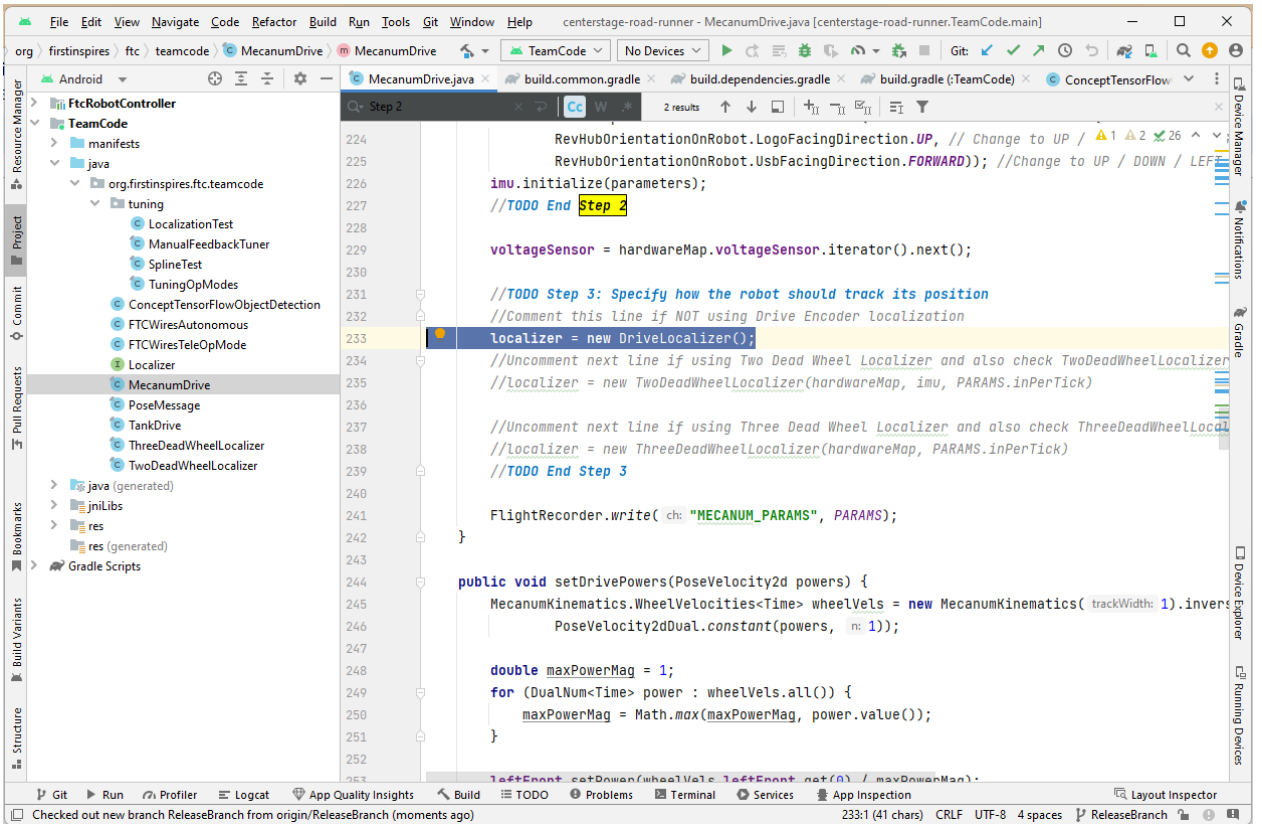
Step 2: IMU setup in MecanumDrive Class in `MecanumDrive.java`

- Search for "`TODO Step 2`" in `MecanumDrive.java`.
- It is assumed that your IMU is named "`imu`" Update direction of IMU by updating orientation of Driver Hub below
- Update the `RevHubOrientationOnRobot.LogoFacingDirection.UP` parameter based on how the Rev Control Hub is oriented in your robot. Change to `UP / DOWN / LEFT / RIGHT / FORWARD / BACKWARD` as in robot
- Update the `RevHubOrientationOnRobot.UsbFacingDirection.FORWARD` parameter based on how the USB port of the Rev Control Hub is oriented in your robot. Change to `UP / DOWN / LEFT / RIGHT / FORWARD / BACKWARD` as in robot



Step 3: Localizer selection in `MecanumDrive.java`

- Search for "`TODO Step 3`" in `MecanumDrive.java`.. Specify how the robot should track its position. There are a few built-in localizers:
- Drive encoders: This is the default. The IMU will also be used on the mecanum to get better heading. (`localizer = new DriveLocalizer();`)
- Two (dead) wheel: Change the right-hand-side of `localizer = to new TwoDeadWheelLocalizer(hardwareMap, imu, PARAMS.inPerTick)`. The code expects the parallel, forward-pointing encoder to be named "`par`" and the perpendicular one to be named "`perp`".
- Three (dead) wheel: Change the right-hand-side of `localizer = to new ThreeDeadWheelLocalizer(hardwareMap, PARAMS.inPerTick)`. The code expects the two parallel encoders to be named "`par0`" and "`par1`" and the perpendicular one to be named "`perp`".
- If changing from default, make the change in also In `tuning/TuningOpModes.java`
- Download the code to the Rev Control Hub



Step 4 : Set the motor movement direction forward

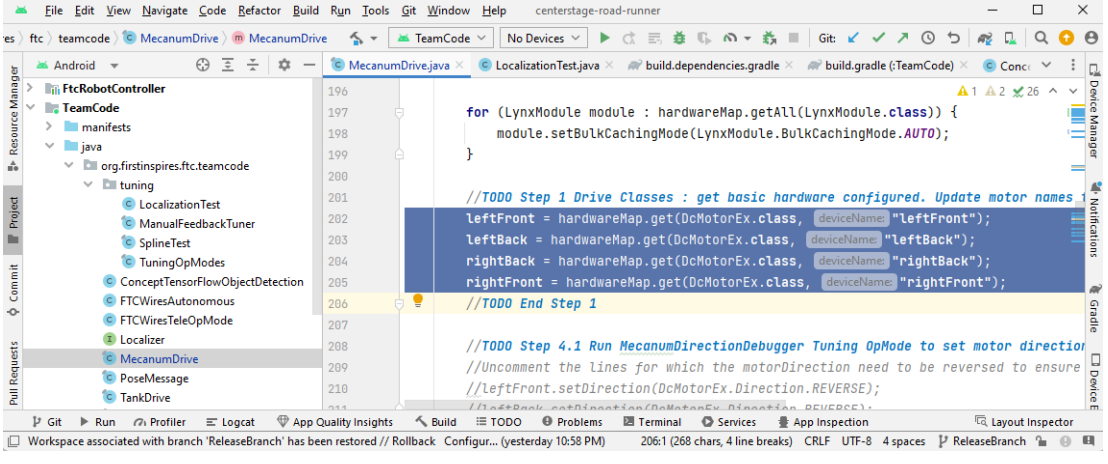
- From now on you would be using the robot to calibrate. Ensure the battery voltage is above 12.5V through the rest of the process.
- On the Driver Hub, Start the Opmode : `MecanumDirectionDebugger` to make sure all the directions are correct. The op mode uses the following button mappings:

```
/**
 * Xbox/PS4 Button - Motor
 *    X / ▢           - Front Left
 *    Y / Δ           - Front Right
 *    B / O           - Rear  Right
 *    A / X           - Rear  Left
 *       The buttons are mapped to match the wheels spatially if you
 *       were to rotate the gamepad 45deg°. x/square is the front left
 *       and each button corresponds to the wheel as you go clockwise
 *                    / ______ \
 *      ------------.-'      '-..+              Front of Bot
 *                /    _   ( Y )   _  \                 ^
 *               |   ( X )    _    ( B ) |     Front Left   \    Front Right
 *          ___   '.         ( A )      /|       Wheel       \       Wheel
 *       .'    '.    '-._____.-'   .'          (x/▢)          \      (Y/Δ)
 *      |        |                  |                           \
 *       '.___.' '.                 |           Rear Left     \   Rear Right
 *                 '.             /              Wheel        \     Wheel
 *                   \.         .'                (A/X)         \   (B/O)
 *                     \_______/
 */
```

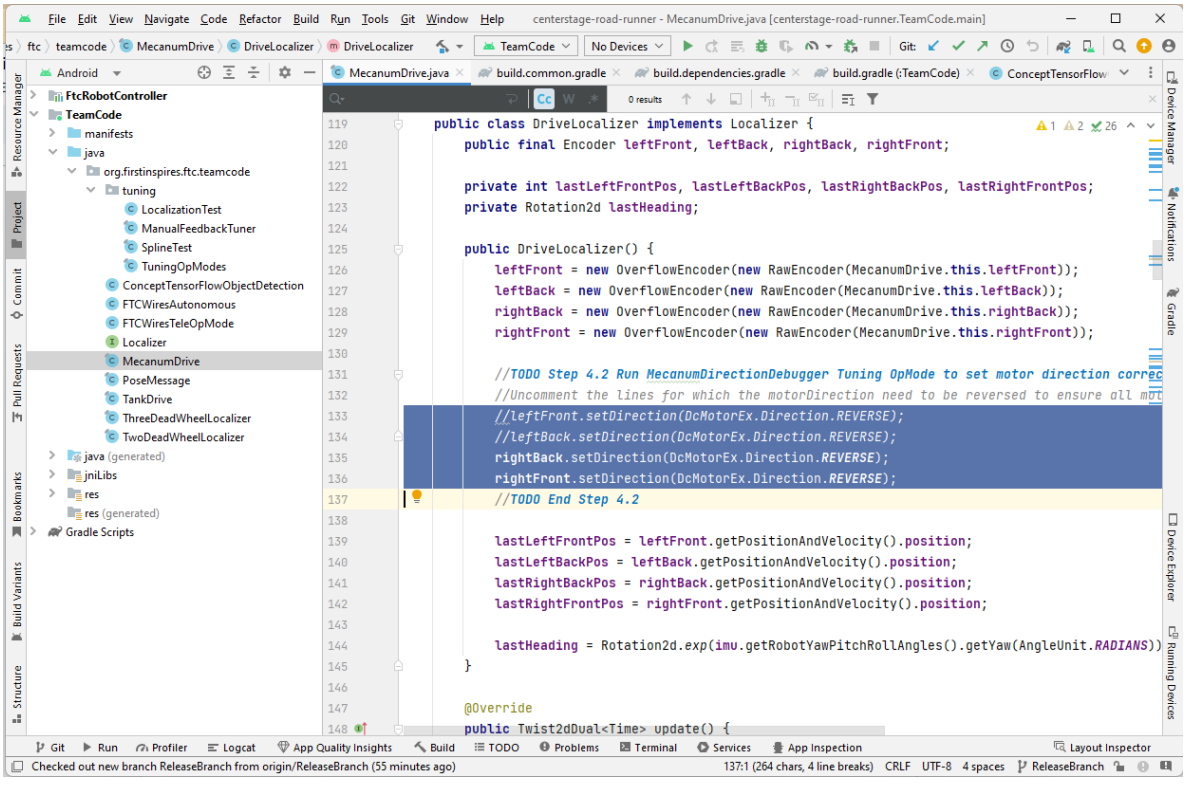
- Press the button corresponding to each motor and check that the motors spin in the forward direction. And if you're using drive encoders, the ticks recorded should increase in a positive direction.
- Search for "`TODO Step 4.1`" in `MecanumDrive.java`.The default code has the right side motors reversed lines as follows in MecanumDrive() method:

```
rightFront.setDirection(DcMotorEx.Direction.REVERSE);
rightBack.setDirection(DcMotorEx.Direction.REVERSE);
```

- If your robot has different motor orientation, any of the wheels are rotating backwards, comment / uncomment the corresponding line and download the code to the Rev Control Hub. And test again.



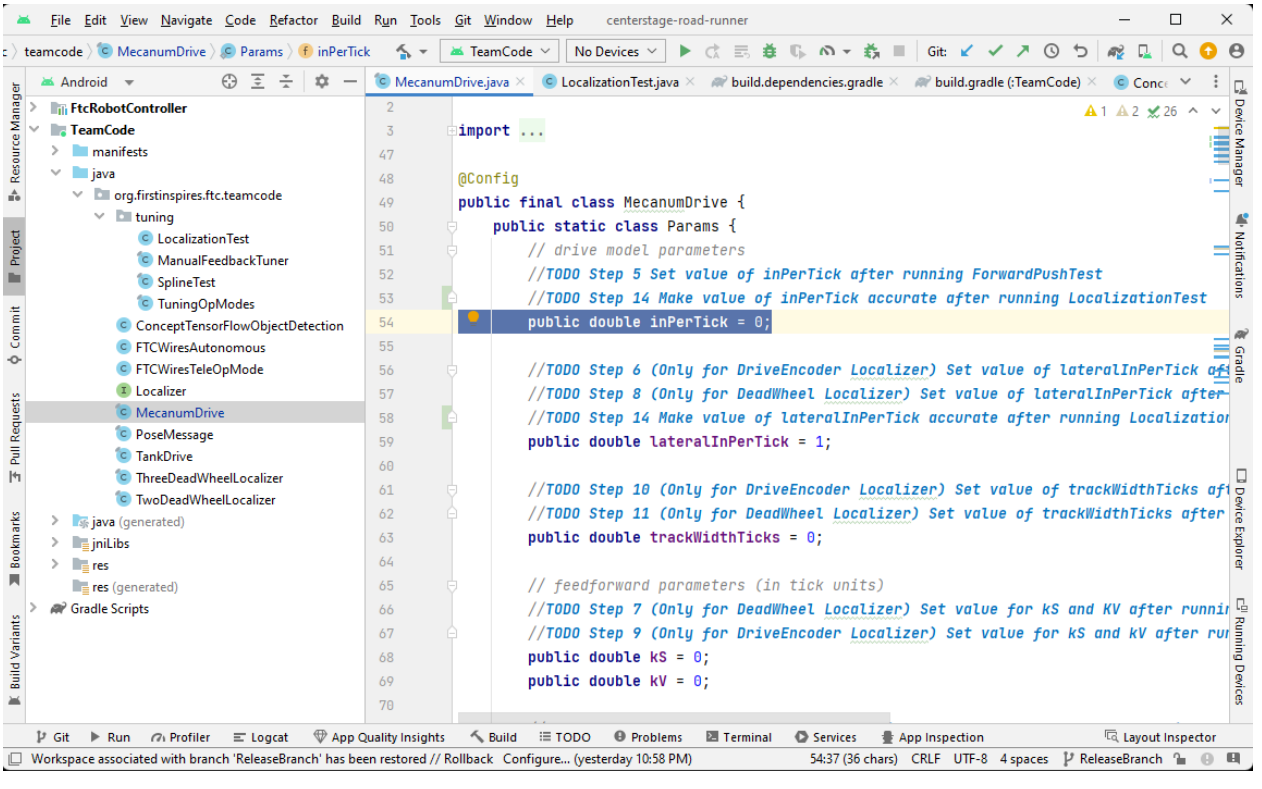
- Once all the motors are driving forward, note which motors are reversed. Search for "`TODO Step 4.2`" in `MecanumDrive.java`. And make the similar changes in the `DriveLocalizer()` method.



- Download the code to the Rev Control Hub

Step 5 : Forward Push Test

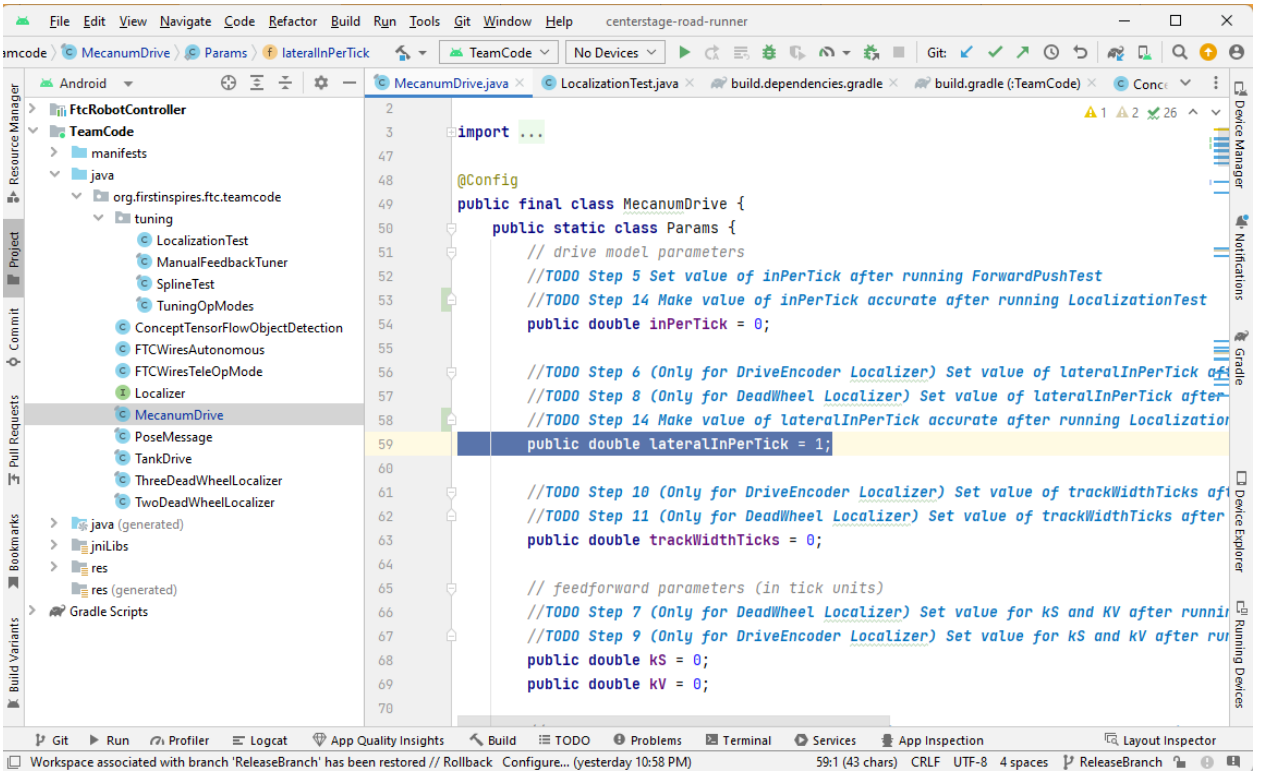
- The goal is to determine the distance in inches traveled per tick of the encoder `inPerTick` empirically
- Place the robot on the tiles with plenty of room in front. Square the robot up with the grid and make note of its starting position. Run `ForwardPushTest` Opmode on the Driver Hub and then slowly push the robot straight until the end of the tiles. Record the “ticks traveled” from display on the Driver Hub before stopping the op mode.
- Make sure the wheels don’t slip! The motors should spin freely. If the bot is too light or otherwise difficult, you can use theoretical values for ticks per revolution, gear ratio, and wheel diameter to compute `inPerTick` instead.
- If the ticks traveled is close to zero no matter how far you push, the motors on one side are probably reversed. Make sure the previous step is finished before returning (particularly Step 4.2 motor reversal matches the one in 4.1).
- Without moving the robot, record also the forward distance traveled on the tiles in inches. (Measure where the back of the robot was when you started to where it is now). Calculate the `inPerTick` value as the real distance traveled divided by the ticks traveled.
- As example for expected values, in our robot with 312rpm gobilda motors, we got 5709.25 ticks for 127 inches yielding `inPerTick = 0.02224460305`
- Search for “`TODO Step 5`” in `MecanumDrive.java`. Set the `inPerTick` value to the calculated value in the `static PARAMS` class in `Mecanumdrive` class



- Download the code to the Rev Control Hub.

Step 6 : Lateral Push Test (mecanum drive encoders only).

- The goal is to determine `lateralInPerTick` empirically
- This routine is simply similar to `ForwardPushTest`, the difference being that it measures rightward motion instead of forward motion.
- Start the `LateralPushTest` opmode on the Driver Hub.
- Set the robot up as before with space towards the right of the robot, slowly push it right, and Record the “ticks traveled” from display on the Driver Hub before stopping the op mode.
- Measure the actual distance traveled in inches and calculate the value of `lateralInPerTick` as the measured distance divided by ticks traveled.
- Note that there would be the inevitable strafing slip that occurs, so you should expect it to be smaller than `inPerTick` by a modest amount. Dont worry about it now, this would be corrected at the end.
- As example for expected values, in our robot with 312rpm gobilda motors, we got 6101 ticks for 126 inches yielding `lateralInPerTick = 0.02065573770`
- Search for “`TODO Step 6`” in `MecanumDrive.java`. Set the `lateralInPerTick` value to the calculated value in the `static PARAMS` class in `Mecanumdrive` class

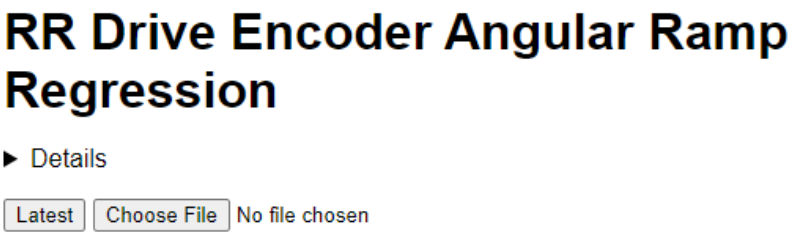


- Download the code to the Rev Control Hub.

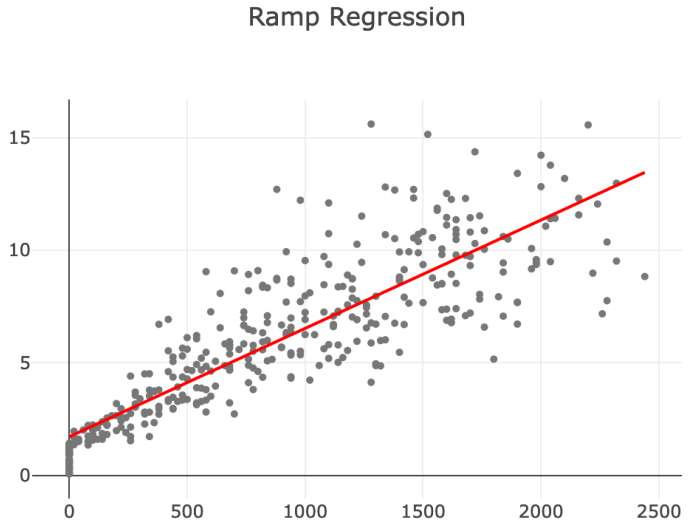
Step 7 : ForwardRampLogger (Only for dead wheel encoders)

Skip to Step 9, if using Drive Encoders.

- The goal is to tune the Motor FeedForward parameters kS and kV empirically. Theory about Motor feed forward is at https://docs.wpilib.org/en/stable/docs/software/advanced-controls/introduction/introduction-to-feedforward.html#introduction-to-dc-motor-feedforward
- This opmode `ForwardRampLogger` slowly increases the forward power given to the robot and measures the forward velocity over time to calculate the static and velocity feedforward parameters (kS and kV, respectively). By default, the power will increase by 0.1 each second until it reaches 0.9.
- Place your robot on the tiles with as much distance in front of it as possible. Start the op mode and press stop immediately when the robot nears the edge of the tile. Don't expect anything to be displayed in telemetry. All data collected is saved to a file for further analysis.
- Once you finish, connect your computer to the RC wireless network using these instructions.
- If you have a control hub, navigate to http://192.168.43.1:8080/tuning/forward-ramp.html .
- If you have a RC phone, navigate to http://192.168.49.1:8080/tuning/forward-ramp.html .
- From now on, I'll use Control Hub URLs.
- You should see a page that looks like this:

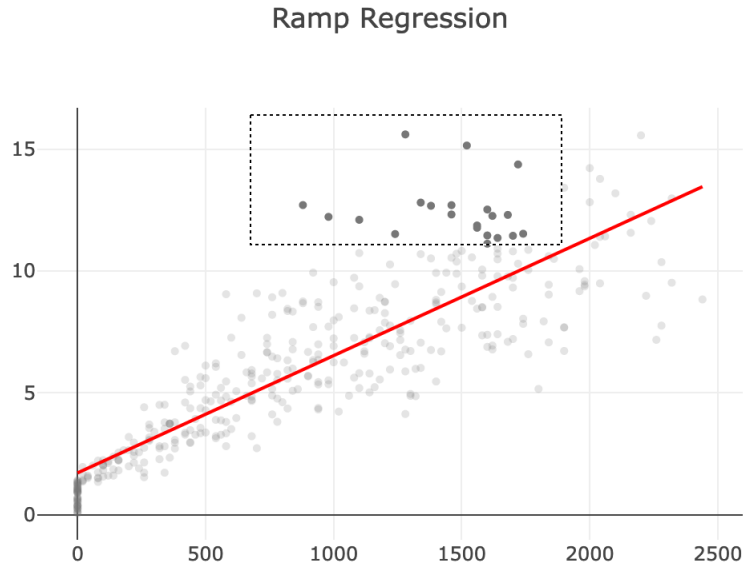


- Click the "Latest" button, and you should see a graph appear:

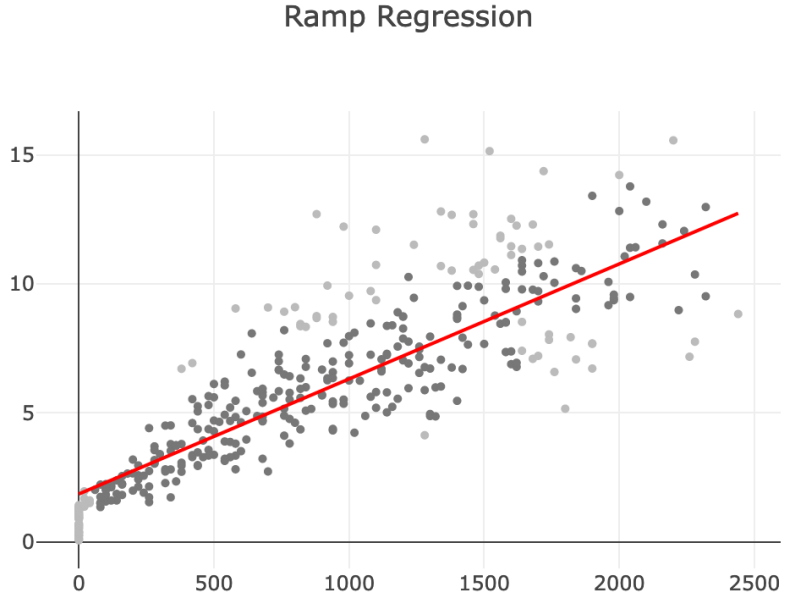


- The gray points are the measurements made by the tuner, and the red line is the line of best fit. Its from this line that the feedforward parameters will be extracted. In fact, you can see preliminary values for kS and kV already displayed above. But those estimates are limited by the presence of outliers. You can see that the red line doesn't quite fit the trend. Let's help the algorithm out by manually filtering out outlier points.

- First, click and drag to select a rectangular region of the plot:



- Then to remove those points, press the “e” key or the “exclude” button above. The selection box will disappear, and the points will turn a lighter gray. After repeating this process as many times as necessary, you should end up with a plot like this:



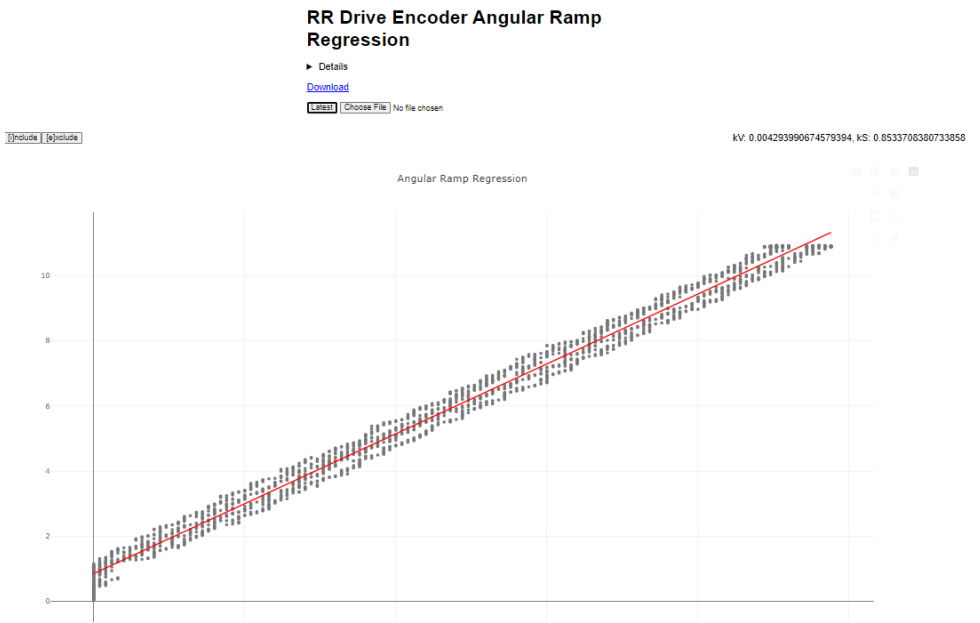
- Now you can copy the values above the plot into the kS and kV fields in the drive class.
- If you mess up and accidentally exclude non-outlier points, select them and press “i” or click the include button to bring them back into the calculation.
- If you have issues or the data looks weird, click the “Download” button to retrieve your data and include it when asking for help.

Step 8: Lateral Ramp Logger (Only for dead wheel encoders)
Skip to Step 9, if using Drive Encoders.

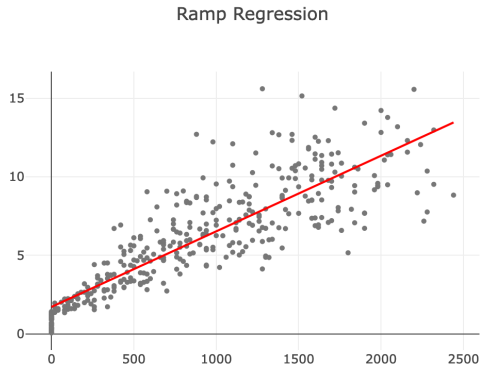
- The goal is to determine `lateralInPerTick` empirically
- If all is configured correctly, the `LateralRampLogger` OpMode will move the robot to the left, increasing in speed as it goes (similar to `ForwardRampLogger`). Stop it at any point, keeping in mind that longer runs will collect more data.
- When you're done, go to http://192.168.43.1:8080/tuning/lateral-ramp.html and click the "Latest" button. The data should be close to linear, and the slope reported is `lateralInPerTick`.

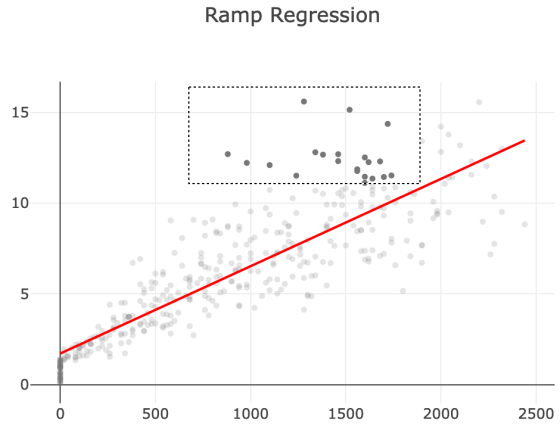
Step 9 : Angular Ramp Logger for Drive Encoders.
(Skip to Step 11 if using Dead Wheel Encoders).

- The goal is determine `trackWidthTicks` empirically
- This opmode `AngularRampLogger` slowly increases the rotation power given to the robot and measures the angular velocity over time to calculate the static and velocity feedforward parameters (kS and kV, respectively). By default, the power will increase by 0.1 each second until it reaches 0.9.
- Place your robot on the middle of your tiles. Start the `AngularRampLogger` op mode and press stop when the robot reaches maximum rotation speed (ie when is not increasing any further). Don't expect anything to be displayed in telemetry. All data collected is saved to a file for further analysis.
- Once you finish, connect your computer to the RC wireless network using these instructions.
- If you have a control hub, navigate to http://192.168.43.1:8080/tuning/drive-encoder-angular-ramp.html
- If you have a RC phone, navigate to http://192.168.49.1:8080/tuning/drive-encoder-angular-ramp.html .
- From now on, I'll use Control Hub URLs.
- You should see a page that looks like this (Click the "Latest" button, and you should see a graph appear:):

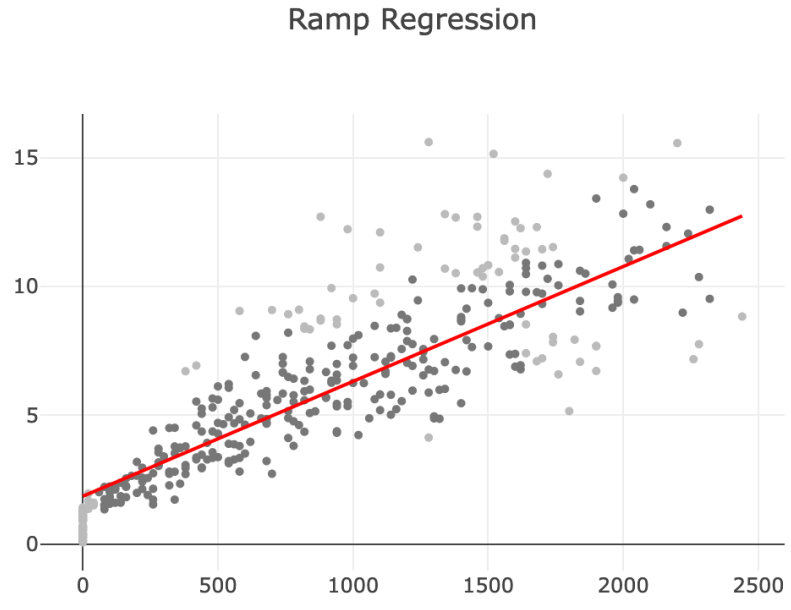


- The gray points are the measurements made by the tuner, and the red line is the line of best fit. Its from this line that the feedforward parameters will be extracted. In fact, you can see preliminary values for kS and kV already displayed above. The above graph is a example of a good fit.
- But in some cases, those estimates are limited by the presence of outliers sometimes as in the example below. You can see that the red line doesn't quite fit the trend. In such cases, we need to help the algorithm out by manually filtering out outlier points.
- First, click and drag to select a rectangular region of the plot:



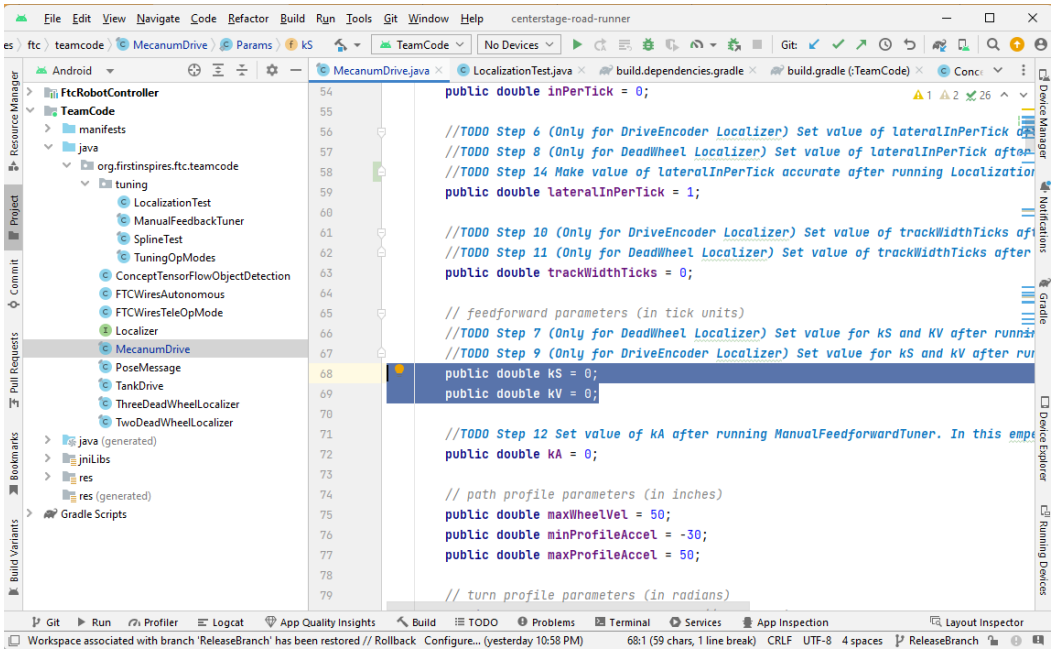


- Then to remove those points, press the "e" key or the "exclude" button above. The selection box will disappear, and the points will turn a lighter gray. After repeating this process as many times as necessary, you should end up with a plot like this:



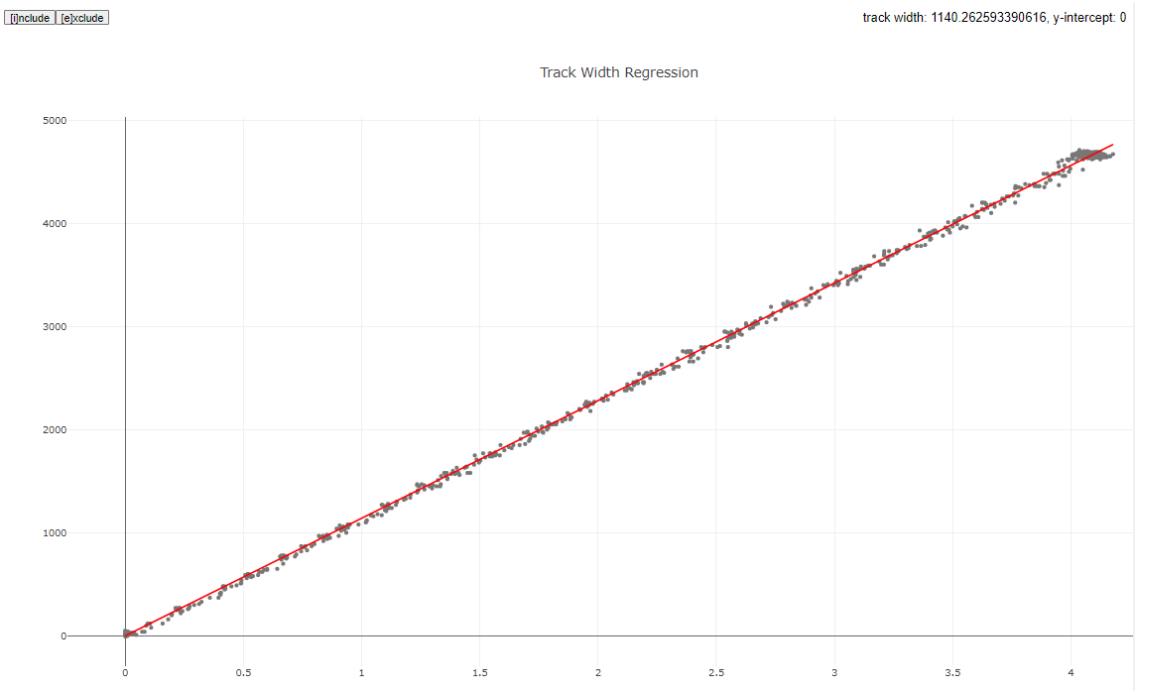
- If you mess up and accidentally exclude non-outlier points, select them and press "i" or click the include button to bring them back into the calculation.
- As an example for expected range, we got kS value of 0.85337083807 and kV of 0.00429399067

- Search for "`TODO Step 9`" in `MecanumDrive.java`.Now you can copy the values above the plot into the `kS` and `kV` fields in the `static PARAMS` class in `Mecanumdrive` class.

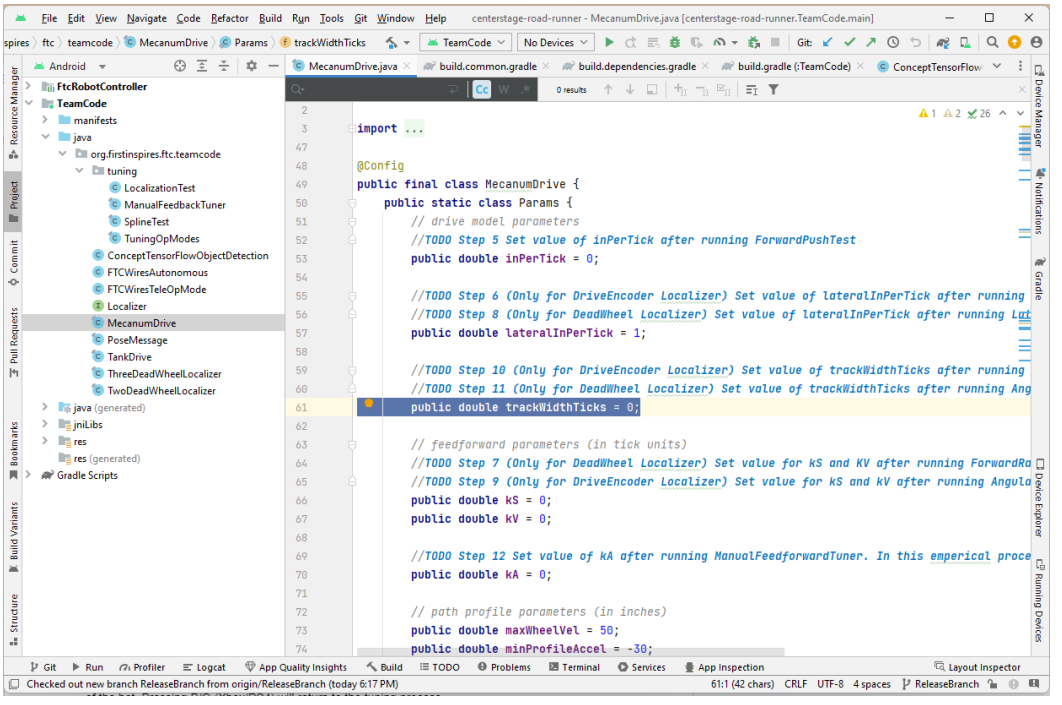


Step 10 : Set Track Width

On the same Angular Ramp Logger plot as in Step 9, Scroll down to find the plot for Track Width Regression.



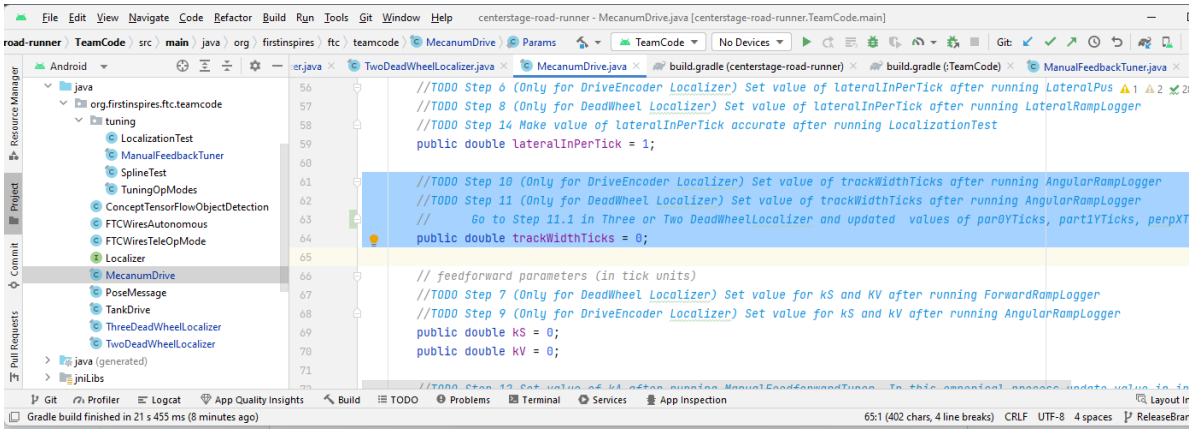
- In case the graph is not as clean as above, use the same outlier-exclusion technique on the “Track Width Regression” and set `trackWidthTicks` to the “track width” value displayed above when you’re finished.
- As an example of expected value, we got value of 1140.262593390616



- Download the code to the Rev Control Hub.

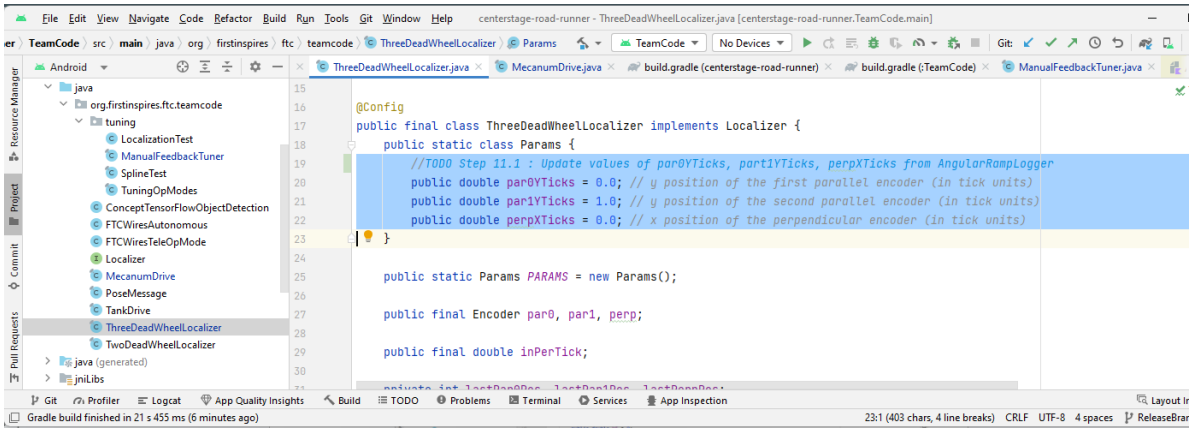
Step 11: Angular Ramp Logger for Dead Wheels Encoders.
(Skip to Step 12 ff using Drive Encoders).

- This opmode `AngularRampLogger` slowly increases the rotation power given to the robot and measures the angular velocity over time to calculate the static and velocity feedforward parameters (kS and kV, respectively). By default, the power will increase by 0.1 each second until it reaches 0.9.
- Place your robot on the middle of your tiles. Start the `AngularRampLogger` op mode and press stop when the robot reaches maximum rotation speed (ie when is not increasing any further). Don't expect anything to be displayed in telemetry. All data collected is saved to a file for further analysis.
- Go to http://192.168.43.1:8080/tuning/dead-wheel-angular-ramp.html and click the "Latest" button. Copy the `kS` and `kV` values from `ForwardRampLogger` into the appropriate boxes and click the "update" button. Use the instructions from the ForwardRampRegression analysis to fill in `trackWidthTicks` in your drive class and the wheel position fields in your dead wheel class.
- To get an accurate `trackWidthTicks` value, you need accurate values for kS, kV.



11/13/2023 Update

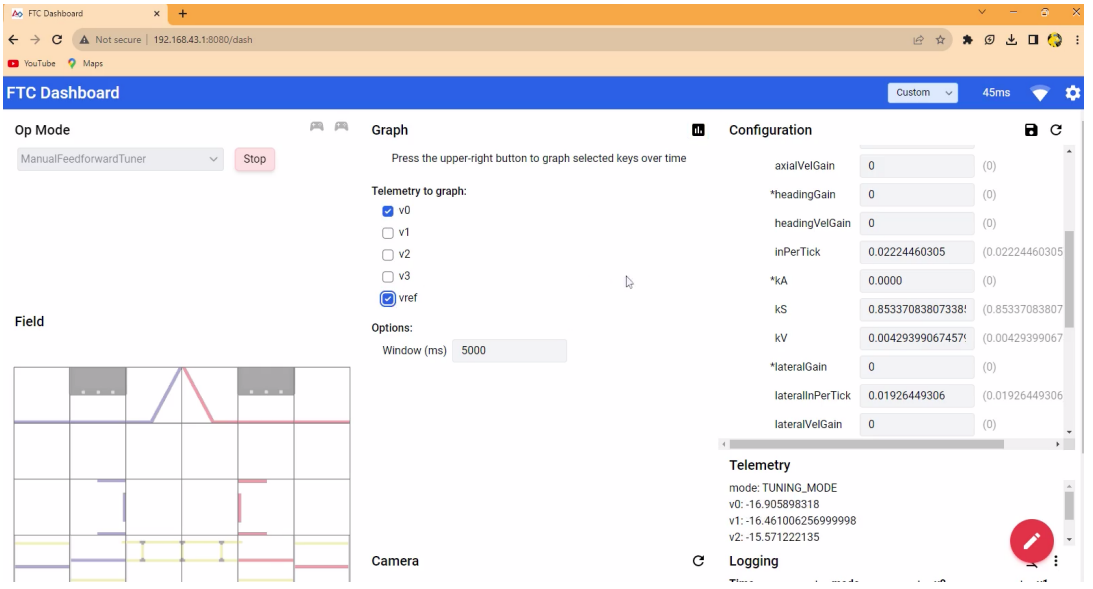
- **Step 11.1** Scroll down http://192.168.43.1:8080/tuning/dead-wheel-angular-ramp.html and if using Three Dead Wheel configuration, find curve fits for `par0YTicks, par1YTicks, perpXTicks`. Go to `ThreeDeadWheelLocalizer.java` and update values there



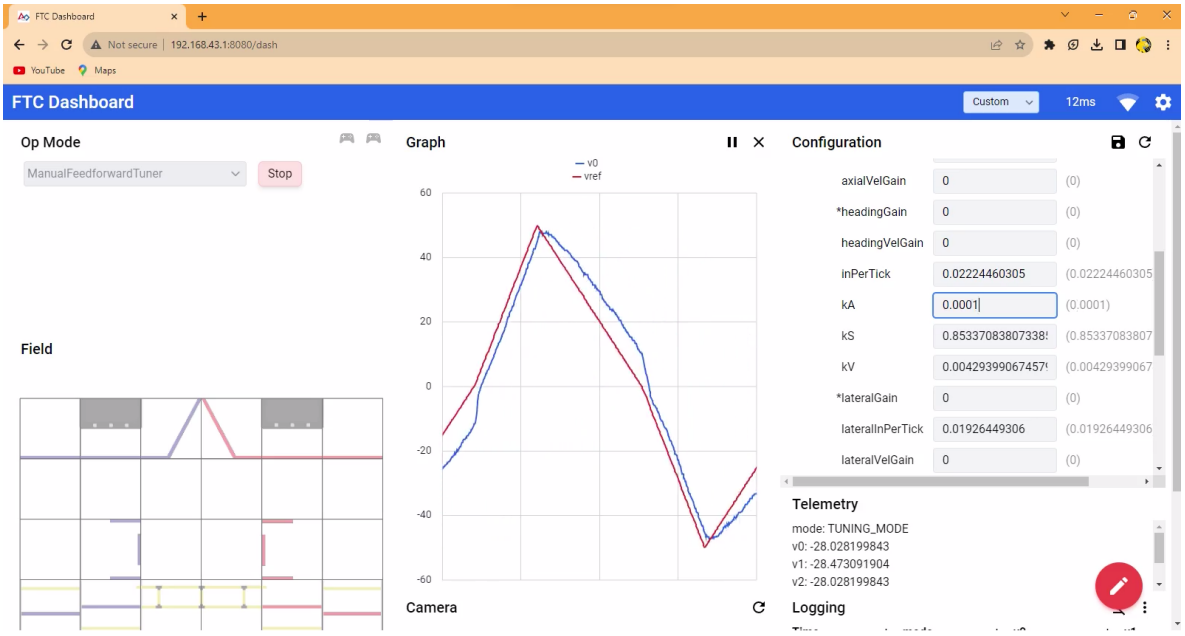
If using Thwo Dead Wheel configuration, find curve fits for `parYTicks, perpXTicks`. Go to `TwoDeadWheelLocalizer.java` and update values there

Step 12: Manual Feedforward Tuner.

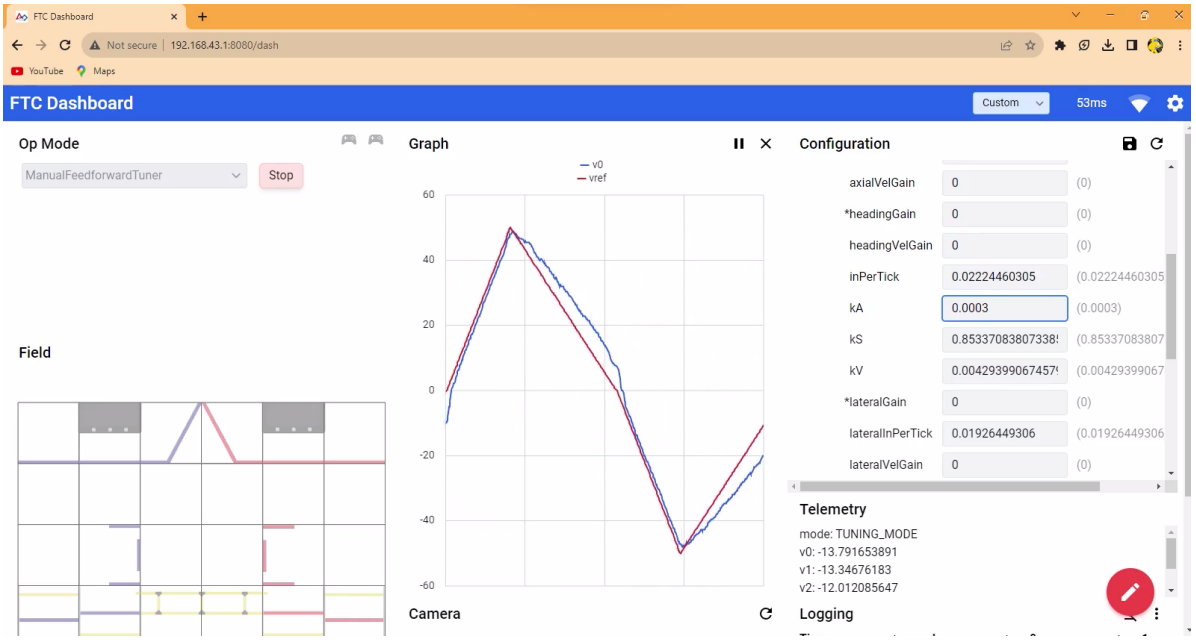
- The goal is to fine-tune kS, kV and add in kA. This routine repeatedly drives forward and back DISTANCE units, giving you an opportunity to finalize the feedforward parameters.
- For this, open FTC Dashboard on your laptop by navigating to http://192.168.43.1:8080/dash.
- Place your robot on the tiles with atleast 80 inches distance in front.
- Run the `ManualFeedforwardTuner` op mode from the pull down on the top left. The robot will start moving back and forth, each traveling around 64 inches.
- There may be a drift on the robot as the motion happens. This is expected. Pressing Y/Δ (Xbox/PS4) will pause the tuning process and enter driver override, allowing you to reset the position of the bot. You can use the joystick buttons on the gamepad to move the robot around. Pressing B/O (Xbox/PS4) will return to the tuning process.
- On the dashboard, In the graph section in the center, check the boxes for v0 and vref.and click on the graph button on the top.
- The graph for V0 and Vref will be plotted as below.



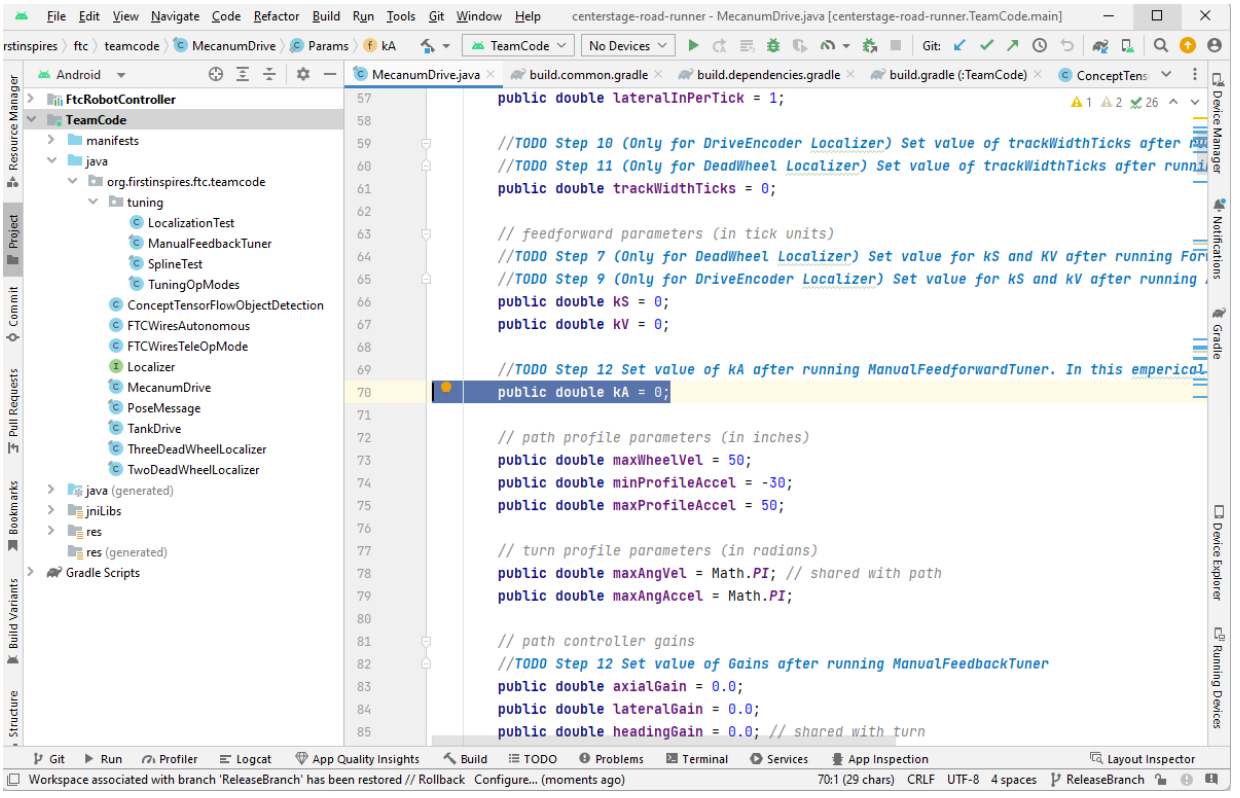
- On the Configuration section on the right, scroll to find kA and Enter value of kA in increments of 0.0001 (and hit Enter each time you change the value) to make the lines of V0 and Vref as close as possible. At some point the lines don't get any better even if you increase kA, and beyond that the robot starts to move uncontrollably.



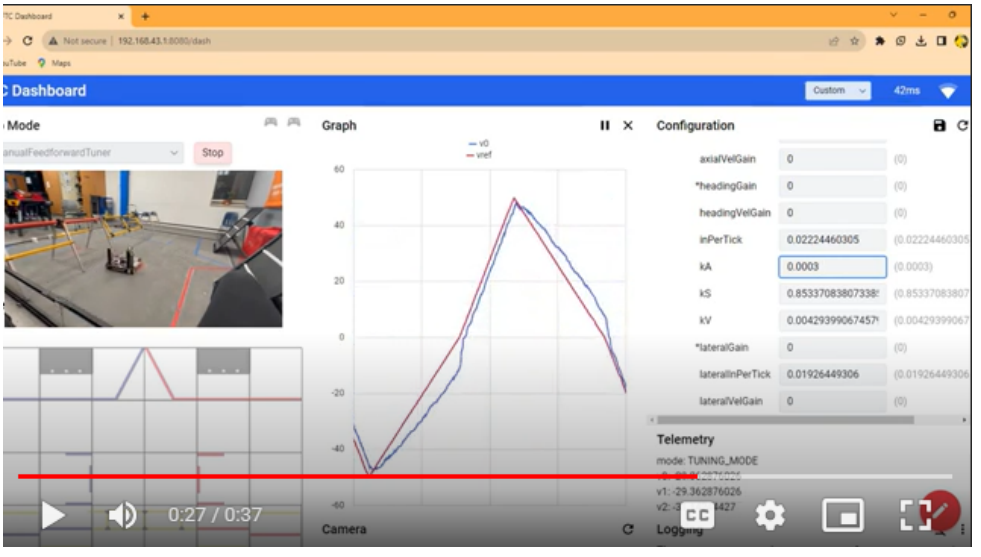
- Pick the lowest value of kA when the lines are close to each other. As an example the value we got was 0.0003.



- Search for "`TODO Step 12`" in `MecanumDrive.java`.Now you can copy the values above the plot into the `kA` fields in the `static PARAMS` class in `Mecanumdrive` class

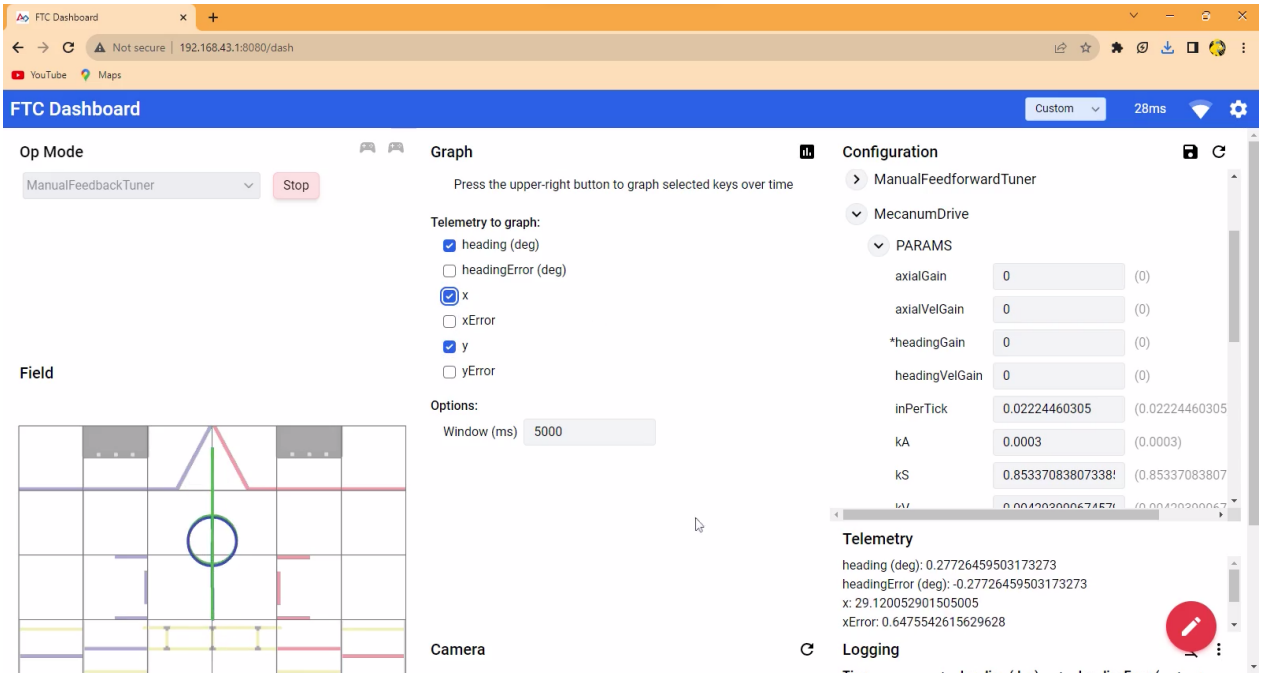


- Download the code to the Rev Control Hub
- [Click here for video of Manual Feed Forward Tuning for reference]



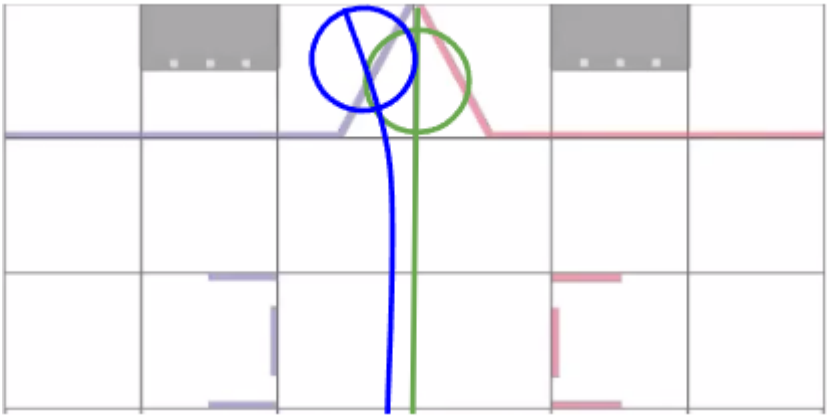
Step 13: Manual Feedback Tuner.

- In the previous steps, you would be experiencing drift in the robot over the many cycles. This is since the robot still has no feedback. The goal is now to tune the feedback parameters.
- Place your robot on the tiles with atleast 80 inches distance in front.
- For this test , open FTC Dashboard on your laptop by navigating to http://192.168.43.1:8080/dash
- Run the `ManualFeedbackTuner` op mode from the pull down on the top left. The robot will start moving back and forth, each traveling around 64 inches. The motion would be plotted on the Field view in the dashboard, The Green Circle denotes the input or target motion, and the Blue Circle denotes the robot's actual motion.
- This routine also goes back and forth DISTANCE (64 inches)units but using combined feedforward and feedback. The theory is that we are using this opportunity to tune the feedforward parameters of your trajectory following controller.

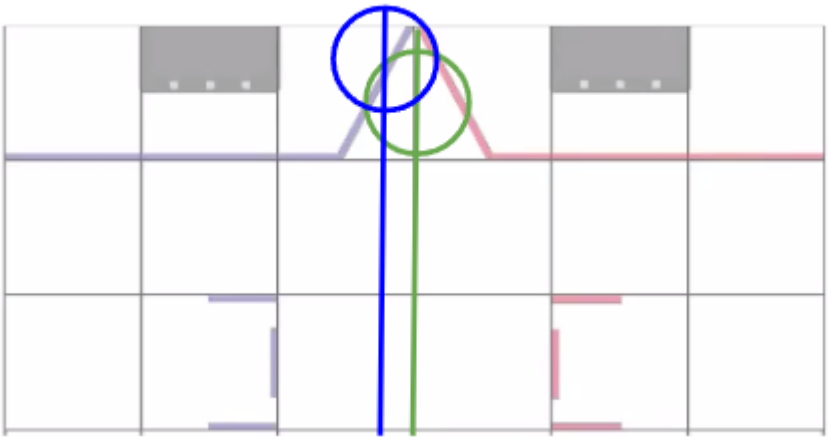


- On the dashboard, In the graph section in the center, check the boxes for heading, x and y and click on the graph button on the top.
- There are two gains for each travel direction (forward, strafe, rotate). The first ("position gain") is like the proportional term of a position PID, and the second ("velocity gain") is like the derivative term of a position PID. We would need to only tune the proportional gains mostly.
- The configuration values we are going to tune are the headingGain, axialGain and lateralGain.

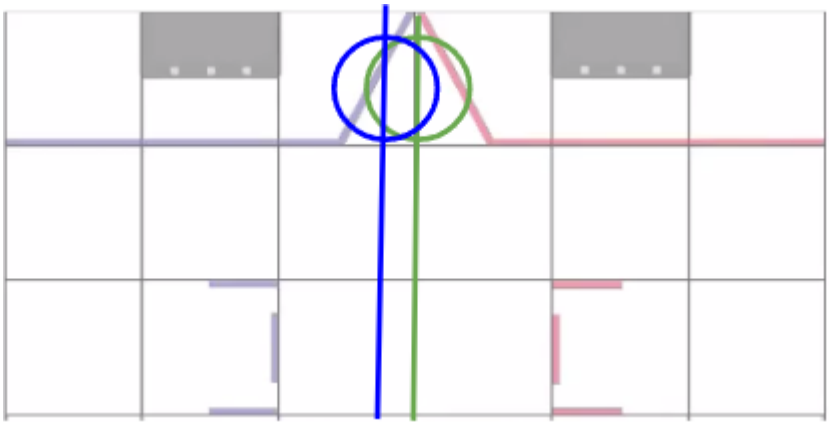
- In the worst case, the starting point would be where the blue circle is drifting away from the green circle in each cycle, The blue circle moves forward and backward more than the green circle and the blue circle motion does not overlap the green circle line, and goes in parallel.



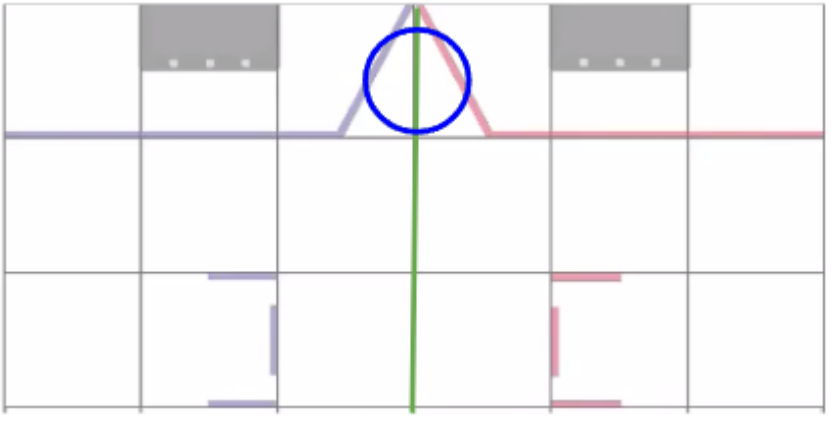
- Start with tuning the headingGain. Increment value of headingGain in steps of 0.5. (Hit Enter each time you change the value). The drifting of the blue circle would start reducing and become parallel to the green circle motion.



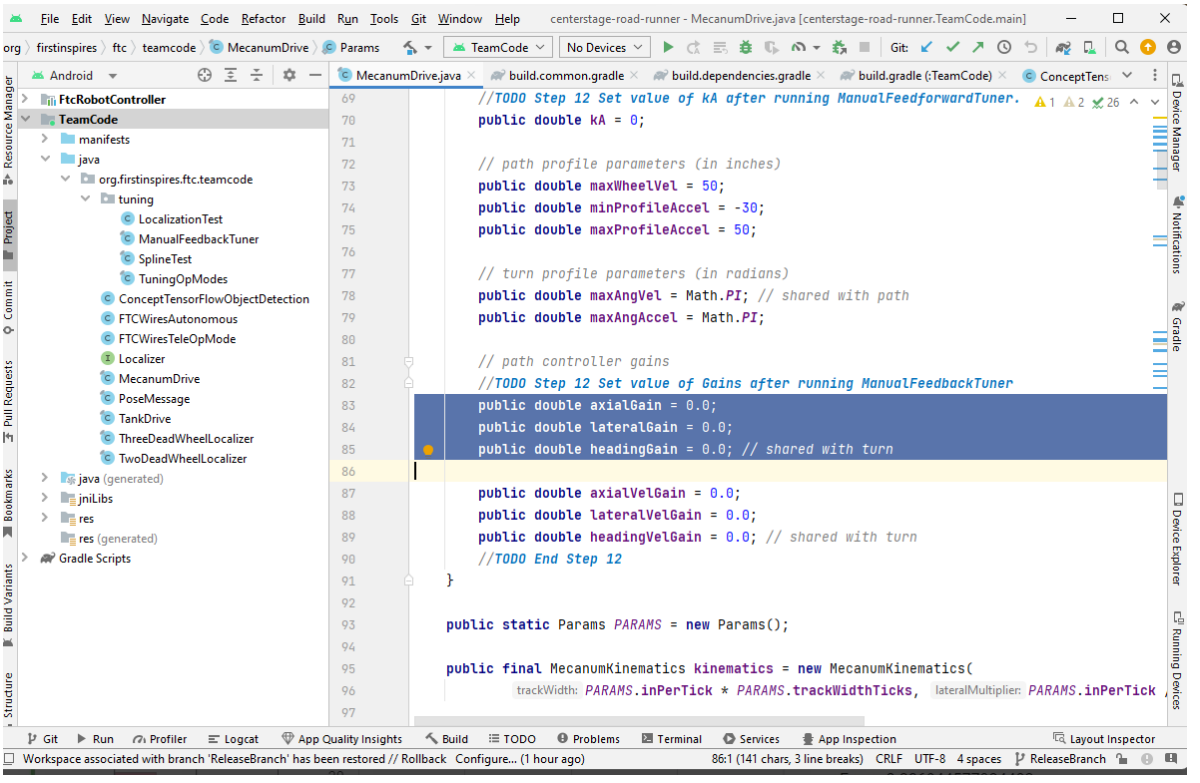
- Then tune the axialGain. Increment value of axialGain in steps of 0.5. The overshoot of the blue circle will reduce and it would match the extent of the green circle.



- Lastly, tune the lateralGain. Increment value of lateralGain in steps of 0.5. The blue line and green line would overlap at the right value. At this time, you can push the robot on the side during motion, and it should still track back to the right position.



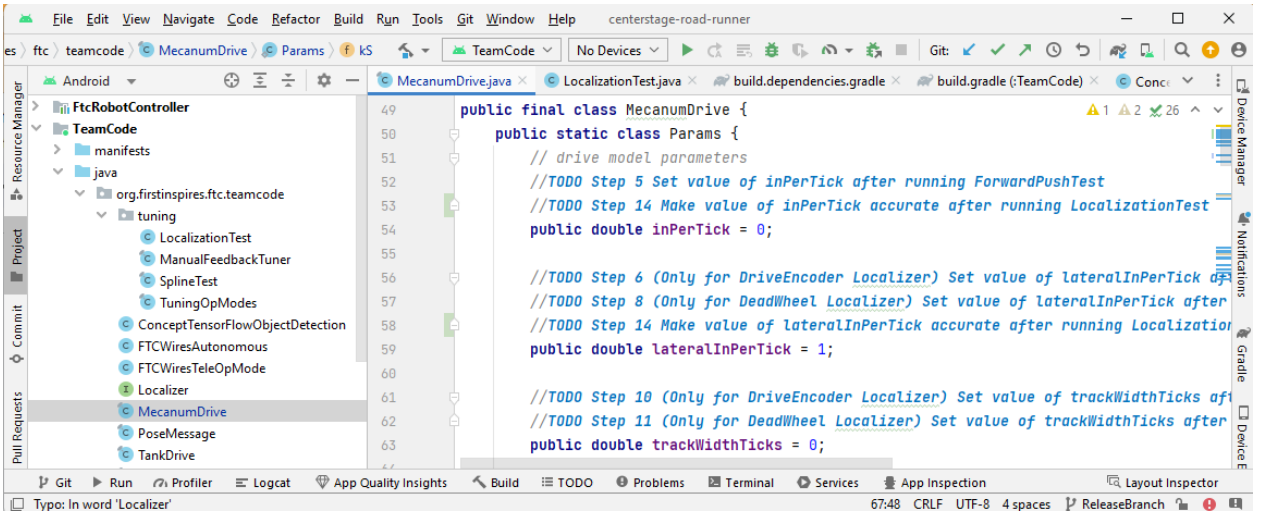
- As an example, the value we got for headingGain = 3.0, axialGain = 2.0, lateralGain =2.0.
- Even if there is no variance, suggest keeping the minimum values of the Gains as 1.0.
- Save the values of gains and update the code.
- Search for "`TODO Step 13`" in `MecanumDrive.java`.Now you can copy the values above the plot into the `kA` fields in the `static PARAMS` class in `Mecanumdrive` class
- Download the code to the Rev Control Hub



Step 14: LocalizationTest

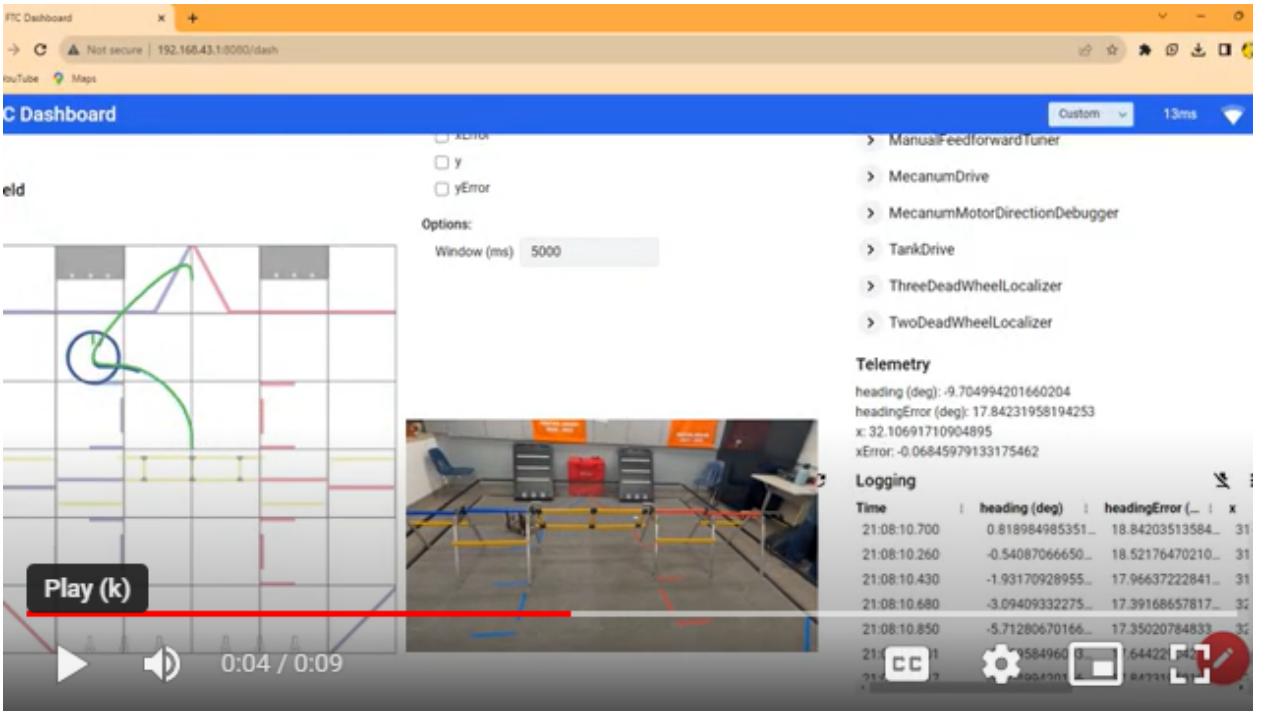
- In the Localization Test, you could drive the robot around with the joystick and the robot would be moving without drifts. However, there may be variation in the physical distance traveled vs what is shown as x, y and heading in the drive station.
- The goal of this test is to fix the motion such that the physical distances match the displayed values of the coordinates
- Straight Test : Fix forward motion measurement (`inPerTick`):
    - Place the robot on the tiles with plenty of room in front. Square the robot up with the grid and make note of its starting position.
    - Run LocalizationTest Opmode on the Driver Hub and then use the joystick to move the robot straight until the end of the tiles.
    - Record the x value from display on the Driver Hub before stopping the op mode. And measure the actual distance traveled with a measuring tape.
    - Calculate the product of the current inPerTick and the ratio of actual distance traveled over the displayed value of x.
    - Assign the calculated value as the new `inPerTick`.
    - Download code and repeat the same, till the actual and measured distance are almost equal.

- Strafe Test: Fix lateral motion measurement (`lateralInPerTick`):
    - Set the robot up as before with space towards the right of the robot, Square the robot up with the grid and make note of its starting position.
    - Run LocalizationTest Opmode on the Driver Hub and then use the joystick to strafe the robot right until the end of the tiles.
    - Record the y value from display on the Driver Hub before stopping the op mode. And measure the actual distance traveled with a measuring tape.
    - Calculate the product of the current lateralInPerTick and the ratio of actual distance traveled over the displayed value of y.
    - Assign the calculated value as the new `lateralInPerTick` .
    - Download code and repeat the same, till the actual and measured distance are almost equal.



Step 15 : SplineTest.

- The goal is to validate the calibration done by running a spline motion.
- Set the robot in the center of the field.
- For this test , open FTC Dashboard on your laptop by navigating to http://192.168.43.1:8080/dash
- Run the `SplineTest` op mode from the pull down on the top left. The robot will start moving in a spline motion denoted in the screenshot below. If tuned well, the Blue circle and the robot will trace the green circle accurately.
- Click here for video of how it is done



If you see variance in the physical motion, or on the display, retrace your steps and fix any errors in the process.

Once you are satisfied with calibration, remove the TODO comments from the code, to mark them as completed.

The calibration should be repeated occasionally, and particularly where there is a significant change physically on the robot, that affects its weight, center of gravity, motors or wheels.

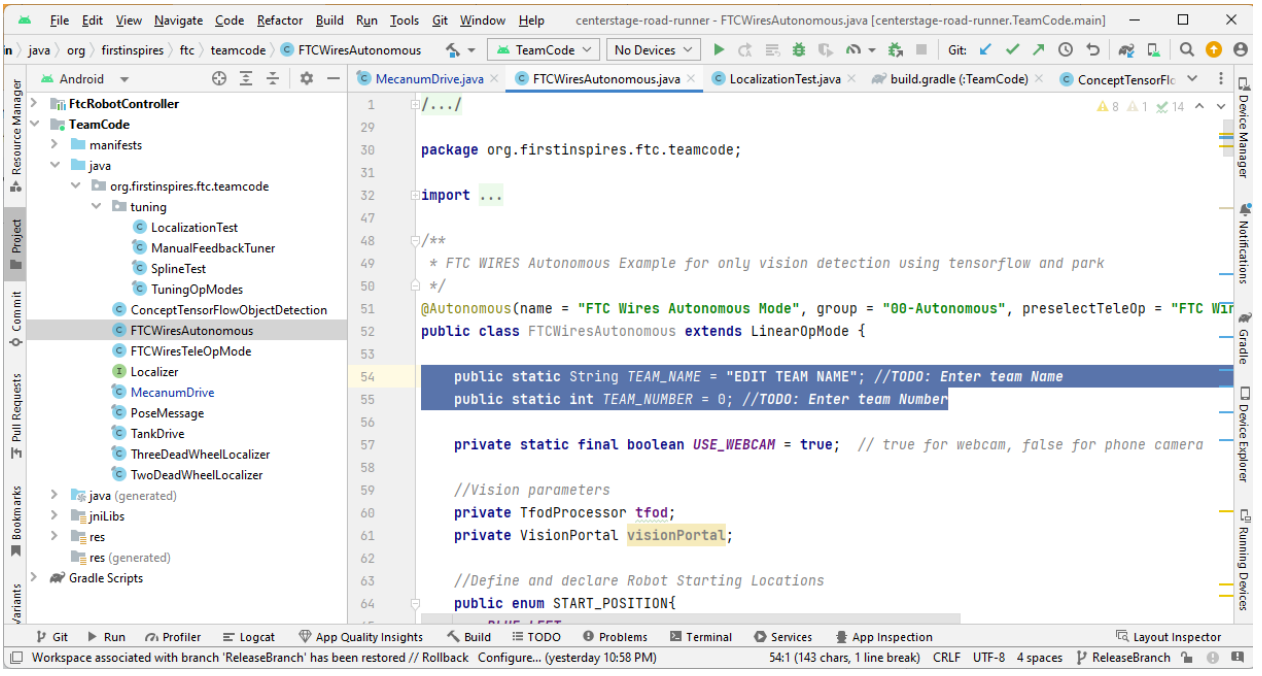
This concludes the calibration process!

Autonomous Op Mode Code

Now that the setup is done, let's get to the real deal - Autonomous mode. The set up and operation is explained in the "How will the FTCWires Autonomous Mode work" section earlier in this document.

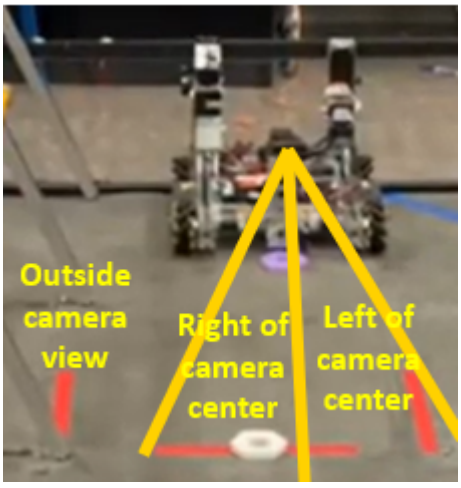
Open the Autonomous mode code in `teamcode/FTCWiresAutonomous.java`

Edit the team name and team number to reflect your credentials.



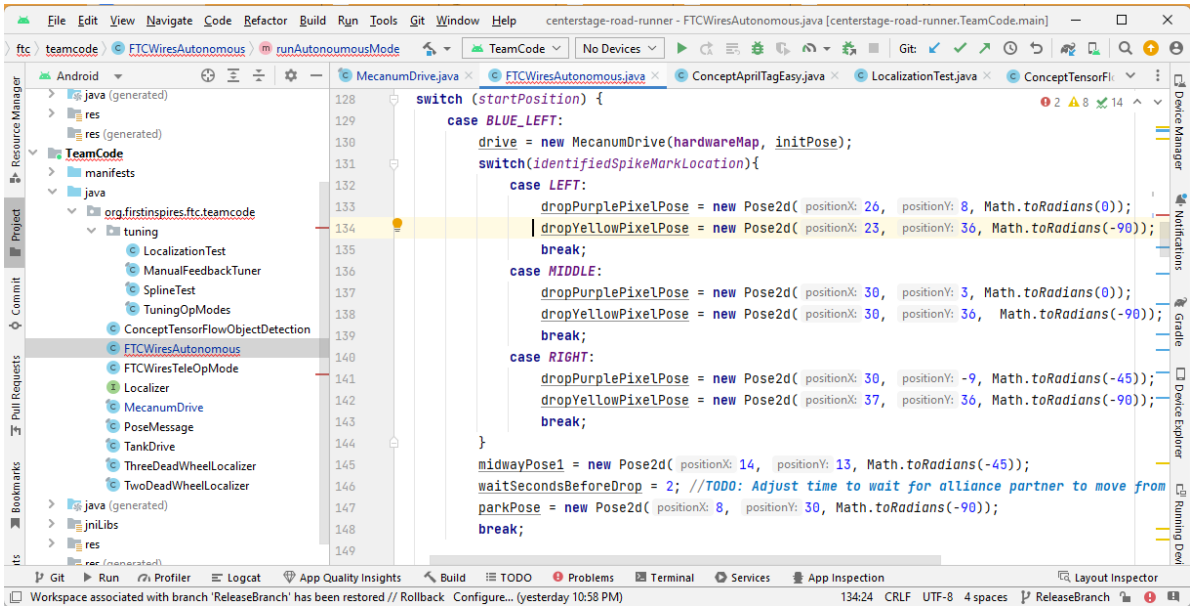
The Opmode code has the following methods:

- `runOpMode()` : This where the Opmode is run. Assume the robot is placed on the field. Once Init is pressed (runOpMode() starts)
- `selectStartingPosition()` function is used to take inputs from gamepad 1 such that pressing X will initiate Blue Left, Y - Blue Right, B - Red Left and A - Red Right paths.
- Vision portal is initiated with `initTfod()`. This method and `runTfodTensorFlow()` are functions to run the visionPortal. This is adapted from `FtcRobotController/java//external.samples/ConceptAprilTagEasy.java`
    - We added the command line to adjust the confidence threshold for recognizing the Pixel on the field.: `tfod.setMinResultConfidence(0.095f);`
    - In `runTfodTensorFlow()`, we also added the logic to determine where the pixel is located on the spike mark. When aligning the robot to the edge farther front he truss, the camera would be in a way that it can "see" only one of the side spike marks and the central spike mark. The logic to determine is to check the x coordinate of the identified pixel to be to the left or right of the camera and if in neither, assume it is on the spikemark that is not seen.

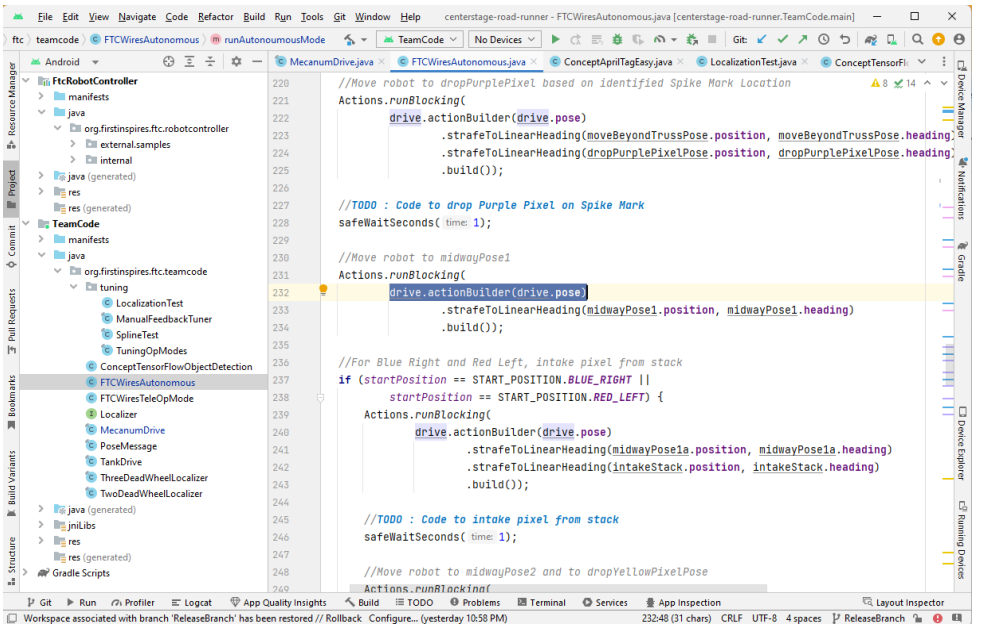


- The `runTfodTensorFlow()` is run in the while loop till the Start button is pressed, the last value it records determines the last identified spike mark with the white pixel
- More details on TensorFlow and how you could use your own team element is at https://ftc-docs.firstinspires.org/en/latest/programming_resources/vision/tensorflow_cs_2023/tensorflow-cs-2023.html

- Once the Start button is pressed opModeIsActive() becomes true and the autonomous code in `runAutonoumousMode()` is executed. This method creates the trajectories and runs the automated motion of the robot
- The method starts with setting coordinates for the different positions of the robot based on the start location selected and the identified spike mark. The coordinate system used here is based on the starting position of the robot. The front of the robot is +x and to the right is +y. The robot starts at x=0, y=0, heading = 0 degrees (Heading is the angle the robot is rotated). The position of robot is indicated by the Pose2d object



- The trajectories are build and run in the next section in the `Actions.runBlocking(drive.actionBuilder(drive.pose),[motion].build()` methods



- There are sections marked to add code for the other subsystem actions of the robot.
- Also we have added a `safeWaitSeconds()` to add waits in the motion. Use this instead of the blocking sleep() call.

The autonomous mode program provided does accomplish moving to the basic scoring operations required in the competition. However, you should modify it as needed to suit your strategy of scoring and how you plan to avoid running to your alliance partner's robot.

To program your own autonomous mode, use this provided code as reference and try out other motion primitives in https://rr.brott.dev/docs/v1-0/builder-ref/

Note: The autonomous mode example programmed here would not display correctly on the FTC Dashboard. This is since the coordinate system assumed here is robot centric, while the dashboard assumes the field centric coordinate system with Red to Blue direction as +x.

TeleOp Mode Code

`FTCWiresTeleOpMode.java` provides basic TeleOp to run the robot using roadrunner functionality. This is adapted from the `localizerTest.java`. You could add your TeleOp code for other subsystems here to utilize it.

Also, use this teleOpMode to determine the “Poses” of the robot to be used in autonomous mode. To do this, start the robot in the start location for the autonomous mode you are trying to program. Note the x, y, heading coordinates on the driver hub. It would be 0,0,0deg. Drive to the design position using your joystick. Note the x, y, heading coordinates from the driver hub and add in your autonomous program. Quite simple!

That’s it : Enjoy programming Autonomous mode and score well!

Incase of any comments, questions or errors you found, or just need help - send a mail to ftcwires@gmail.com or DM on Instagram @ftcwires

Or raise issues at https://github.com/ftcwires/centerstage-road-runner/issues

Acknowledgement :

Creators of Roadrunner: Acmerobotics, Ryan Brott

Learnroadrunner.com, Noah Bres

and

Team Hazmat 13201 (www.13201hazmat.org)

who created the FTCWires software platform.