Odometry for FTC
A Practical Guide

Andrew Goossen

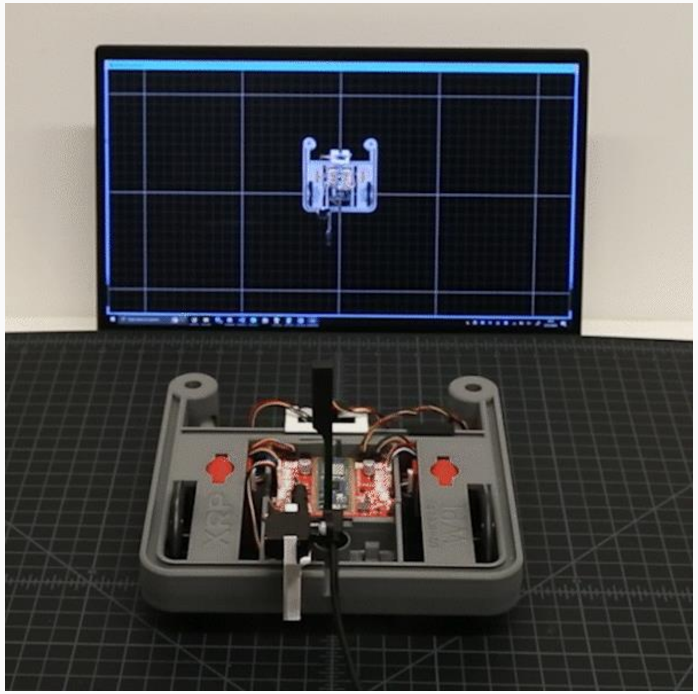
Swerve Robotics

Overview

- Different types of odometry for FTC
  - ~~Underwater odometry~~
  - Encoder odometry
  - Optical odometry
- Using odometry
  - Build-your-own trajectories
  - Road Runner
- Tips and tricks

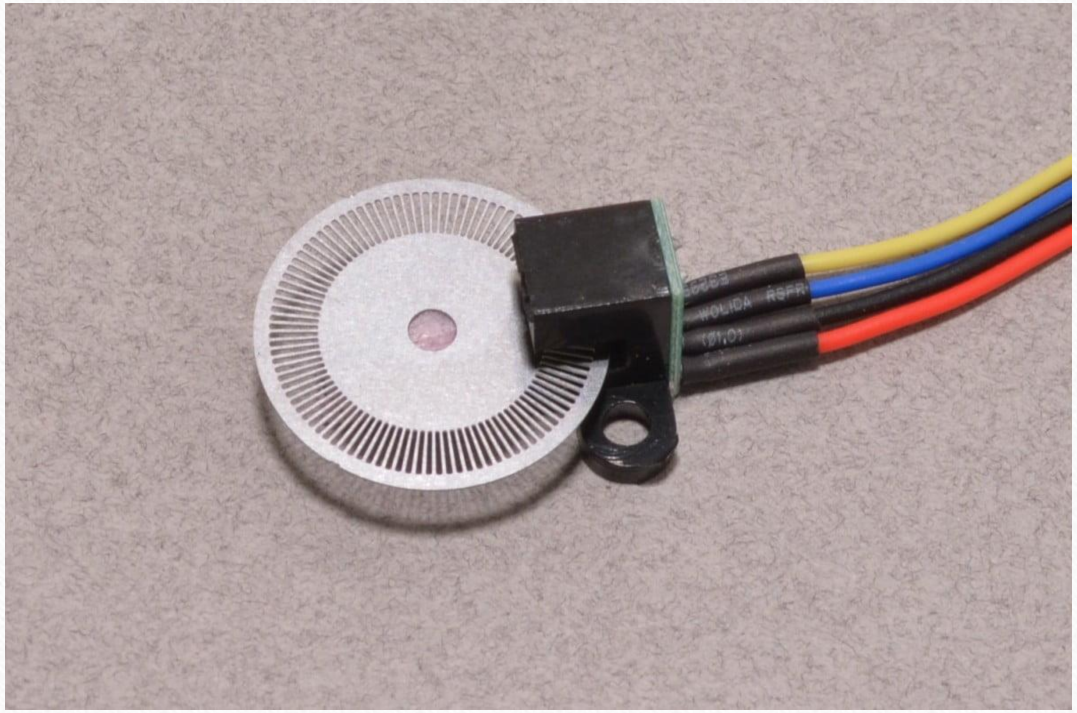
What is odometry?

- Odometry is a technique to estimate a robot's change in position over time using data from motion sensors
- It gives your code an estimate of the robot's effective position and orientation (the ***pose***) relative to a set starting point
- In FTC, a ***pose*** is 3-dimensional: it's the robot's *(x, y)* position on the field plus its heading
- Odometry is a ***relative*** tracking technique that is very accurate over short distances, but which accumulates significant error over time



Motor Encoder Odometry

- Encoders measure the rotation of a shaft
  - Commonly measured in units of ***ticks***
- Most FTC motors have encoders built-in
  - They're there so that the motor can know how far and how fast it's turning
- Encoders on the drive train's axle can estimate the robot's ***axial*** and ***angular*** velocities
  - This can tell us the change in heading and the forward or backward motion…
  - …allowing us to estimate the robot's current ***pose***
- But it turns out that most teams don't use the motor encoders for odometry!



Dead Wheel Odometry

- ***Dead wheels*** are unpowered wheels (not attached to a motor) with encoders, dedicated to odometry
  - Also known as ***tracking wheels*** or ***odometry pods***
- They offer accuracy advantages over motor-encoder odometry in important scenarios
  - Drive motor wheel slip
  - Sideways motion
  - Jostling from other robots
- You can purchase odometry pods or make your own, e.g.:
  - GoBilda's $75 swingarm odometry pod
  - OpenOdometry's open-source module

Configuring Dead Wheels in the SDK

- FTC does not make configuration of dead wheels encoders intuitive!
- A dead wheel and an unrelated motor can share the same port assignment
  - Don't do anything in the Control Hub's "Configure Robot" UI specifically for the encoder in this case
    - In your code, call **hardwareMap.get(DcMotorEx.class, …)** twice, using the name you gave the motor
    - Use one object instance for programming the motor, the other for querying the encoder
  - Make sure you never program that motor with RUN_WITH_ENCODER, RUN_TO_POSITION or STOP_AND_RESET_ENCODER
    - In other words, always use RUN_WITHOUT_ENCODER

Configuring Dead Wheels in the SDK

- A dead wheel can use a port even if there is no motor to share the port assignment
  - In this case, create a fake motor in the Control Hub's "Configure Robot" UI
  - Pick any motor type for the fake motor, it doesn't matter
  - In your code, call **hardwareMap.get(DcMotorEx.class, …)** once, using the name you gave the fake motor
  - Use this object instance to query the encoder

How Many Encoders do you Need?

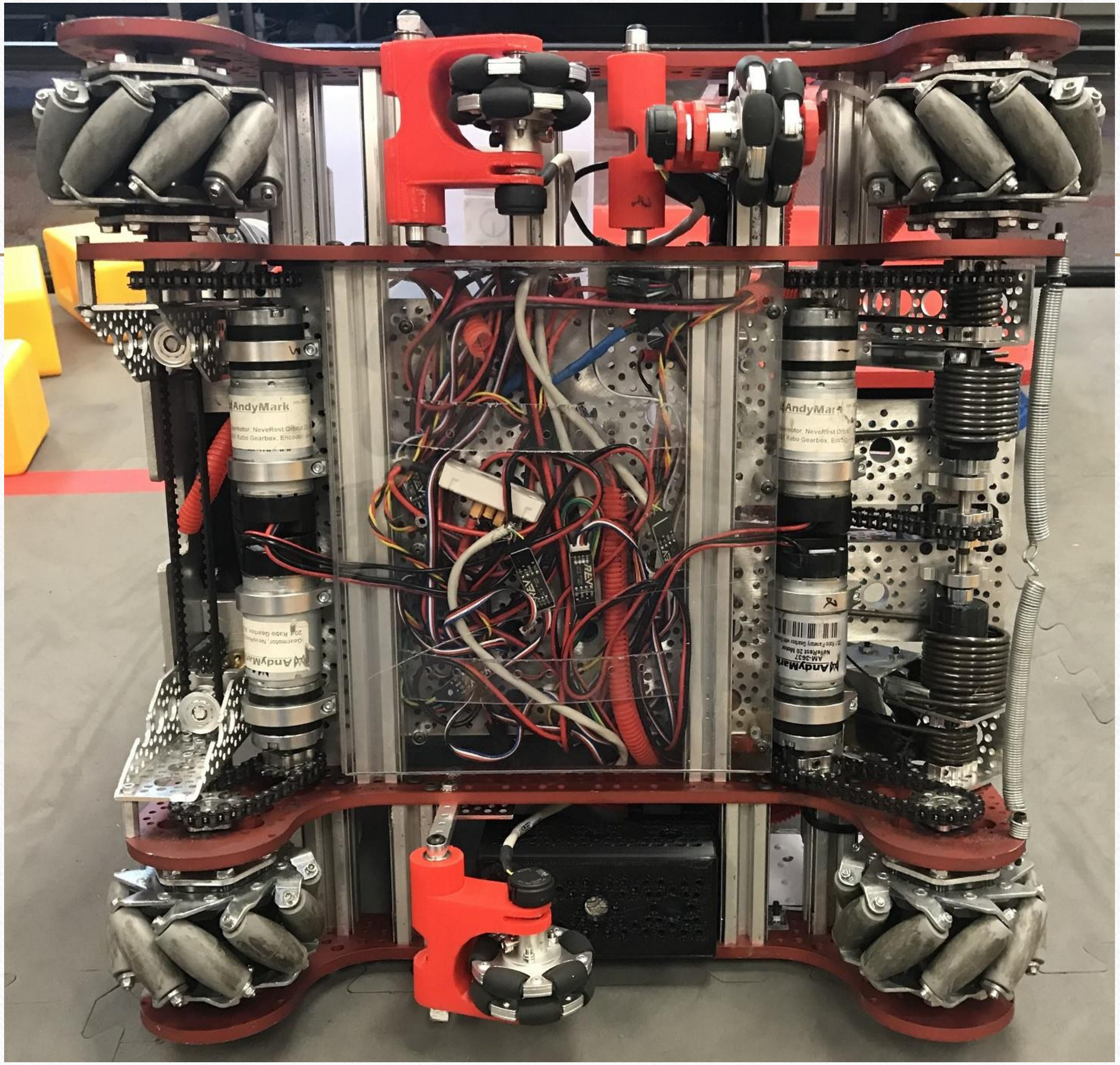
- The minimum number of encoders depends on the drive train and whether you'll use an ***IMU gyro*** for measuring change in orientation, i.e.:

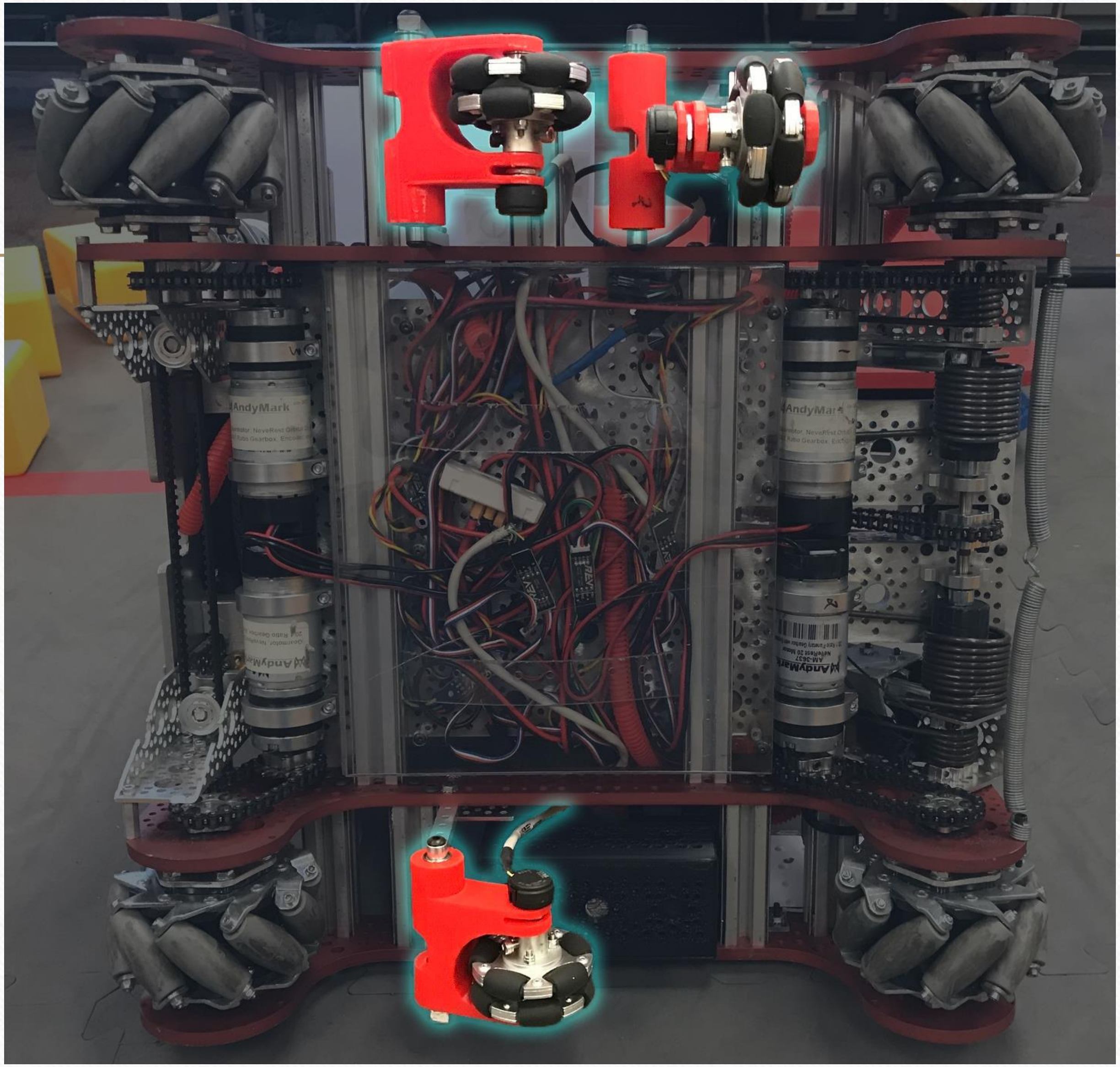
	Use IMU	No IMU
Differential (e.g., Tank)	1+	2+
Holonomic (e.g., Mecanum)	2+	3+

- More encoders than the minimum provide redundancy for potentially improved accuracy
- If you're targeting 3 dead wheels, be sure to evaluate performance compared to 2 dead wheels plus IMU
  - No hardware changes needed for evaluation
  - Simply substitute the change in gyro readings instead of the computed *delta theta*
  - We've found that 2 wheels with IMU was often more accurate than 3-wheel odometry

Where Do You Put Dead Wheels for Holonomic Drives?

- If 2 dead wheels, mount them orthogonally (90 degrees to each other), anywhere on the underside of the robot
- If 3 dead wheels, the parallel pair should be as far away from each other as possible
  - …but they can be anywhere front to back (i.e., positioned anywhere along their line of motion without affecting the odometry math)

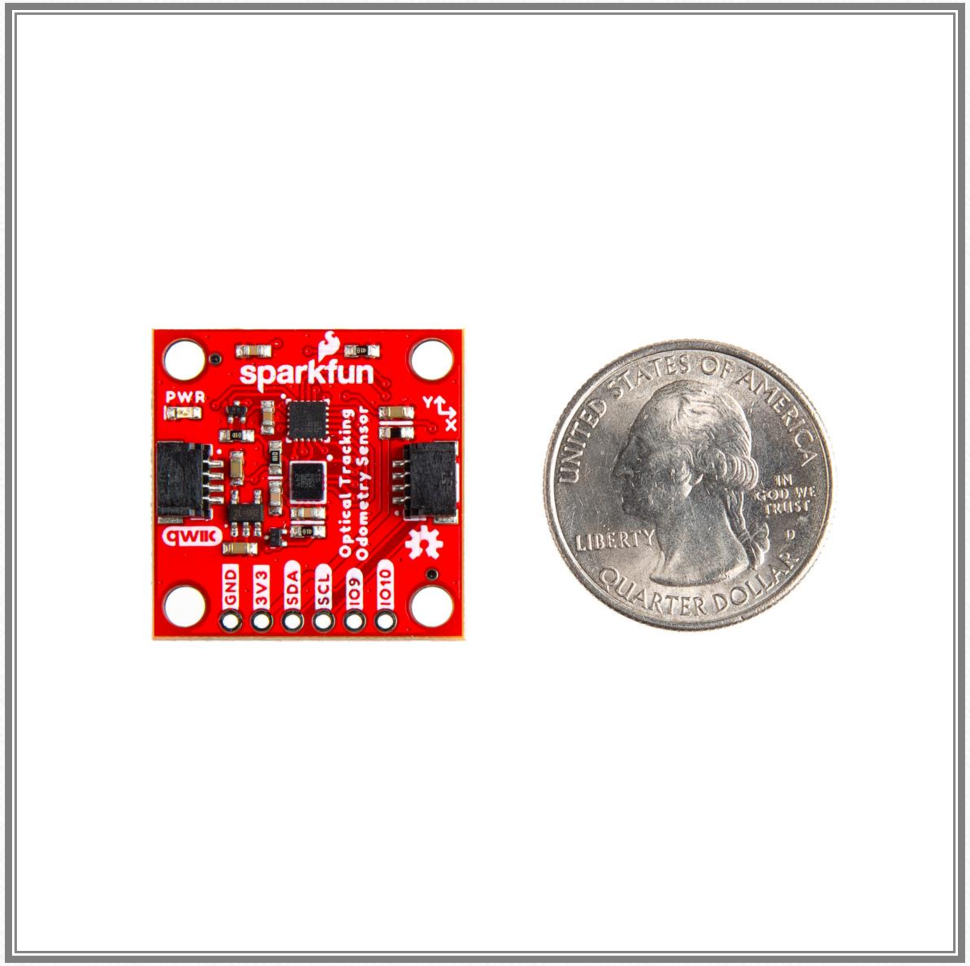

AndyMark
AndyMark


AndyMark

Encoder Odometry Resources

- For a simple derivation of the math needed to convert encoder input to change in motion, see Mecanum Wheel Odometry by FTC 9866 Virus
- For the best code to use as a starting point for your own implementation, see FTC Lib's odometry section
- If using Road Runner, see Learn Road Runner's dead wheel section
- For a mathematical proof about where you can position wheels, see Introduction to Position Tracking by Team 5225
- Game Manual 0 also has a good odometry section

New to FTC: Optical Odometry



- The $80 ***SparkFun Optical Tracking Odometry Sensor (OTOS)*** was revealed at the Worlds Competition this year
- It operates using the same *optical flow* principles as computer mice and drone navigation systems

OTOS vs. Dead Wheels

OTOS Pros vs. Dead Wheels

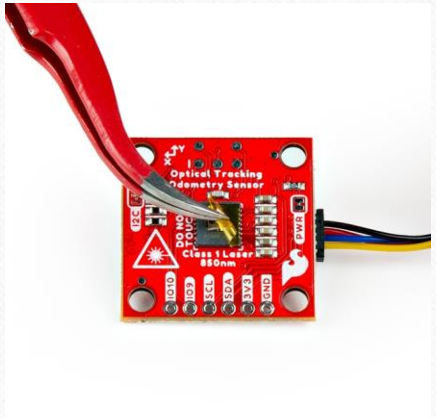
- More accurate
- Only need one (vs. 2 or 3 dead wheels)
- Small and can be located pretty much anywhere on the robot
- Less fiddly (e.g., no wheel slip worries)
- Simpler tuning
- Does all the math for you
- More accurate than an FTC software implementation could ever be

Cons

- Picky about being exactly 10mm from floor
- Road Runner doesn’t natively support it yet
- Sensor loop isn’t as fast as 3-dead wheel case

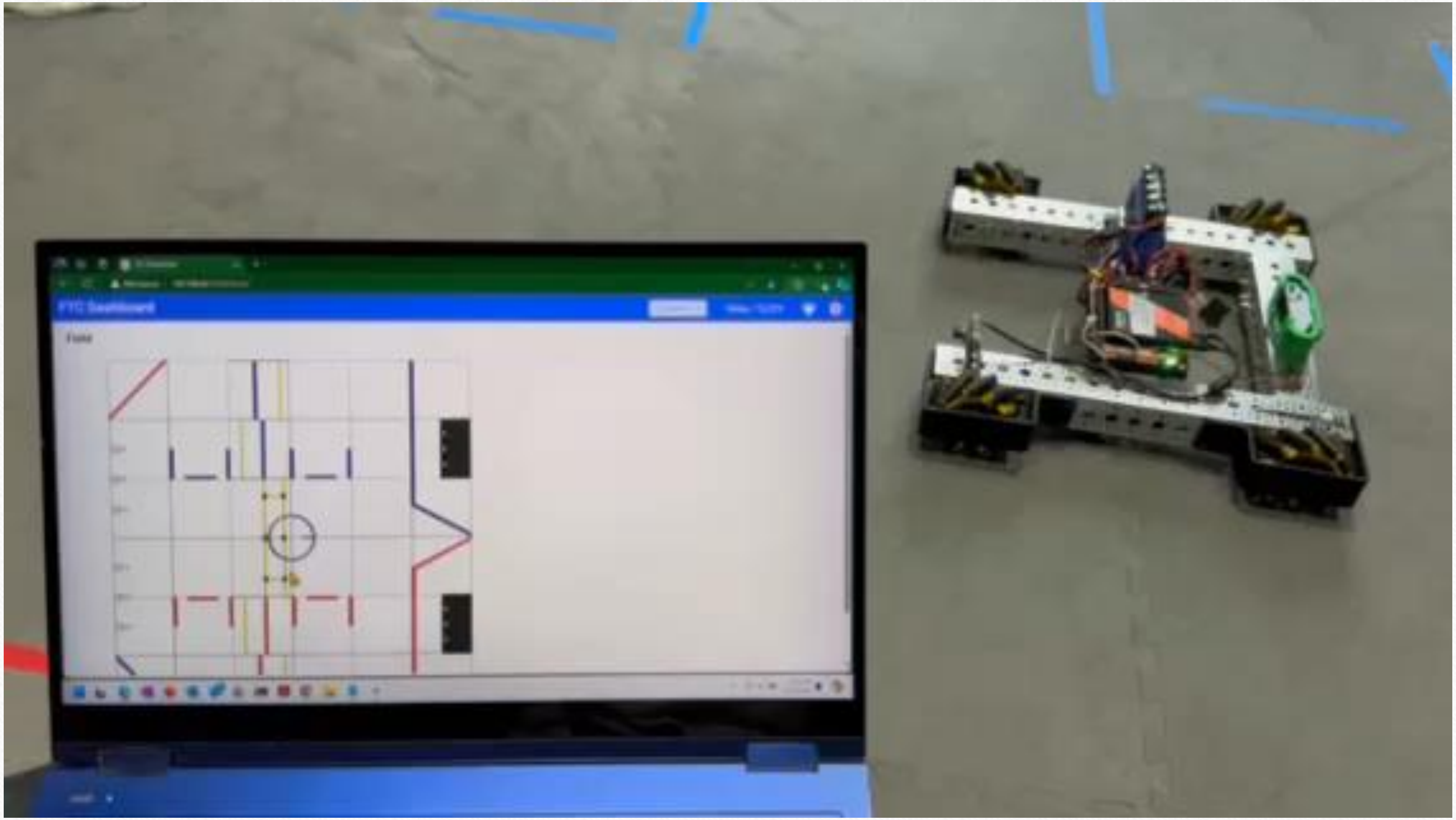
Using OTOS

- SparkFun has many resource guides available
  - There’s a good overview YouTube video
  - Their hookup guide includes Onshape mount designs for FTC robots
  - Lots of technical documentation available in their documents section
- Be sure to use the FTC 10.0 SDK when it comes out
  - The current 9.2 SDK’s driver has a bad heading bug
- And remember to remove the Kapton tape from the sensor!
  - It’s really hard to see, but it’s there!



OTOS Calibration is Important

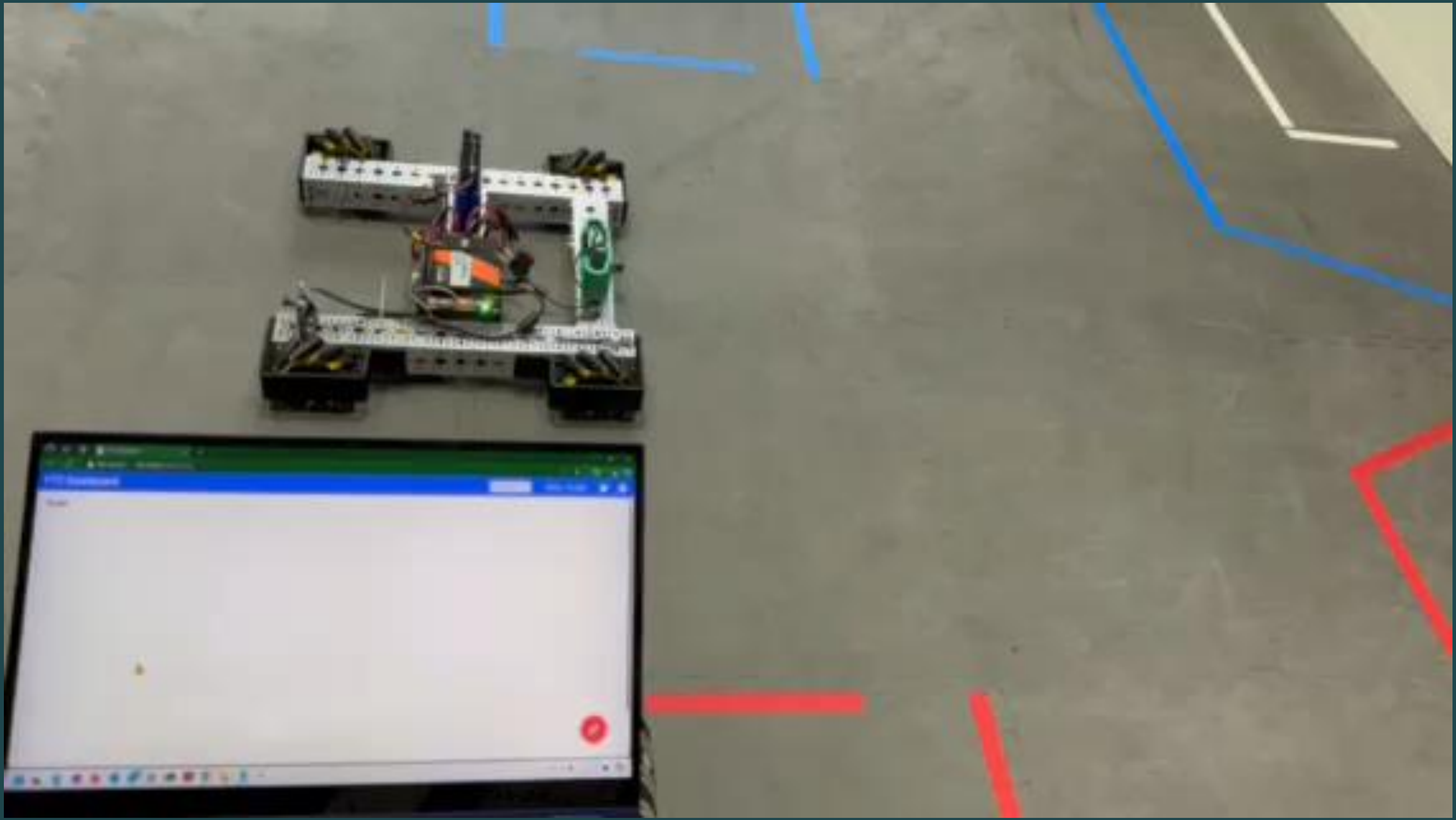
- See the OTOS Video for instructions
- Although pretty good out-of-the-box, I highly recommend calibrating *LinearScalar*
  - This tells you how far off you are from exactly 10mm!
  - *AngularScalar* has generally been pretty good
- The *offset* tells the sensor its position on the robot relative to the center of rotation
  - The offset can be measured with a ruler…
  - …or calculated programmatically from the origin of the circle traversed by an in-place rotation of the robot (an exercise left for the reader!)


FTC Dashboard

Techniques Common to any Type of FTC Odometry

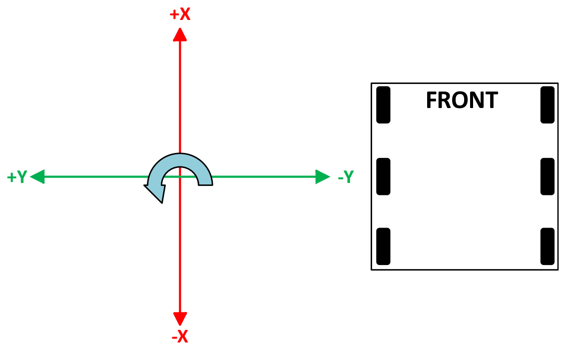
FTC Dashboard

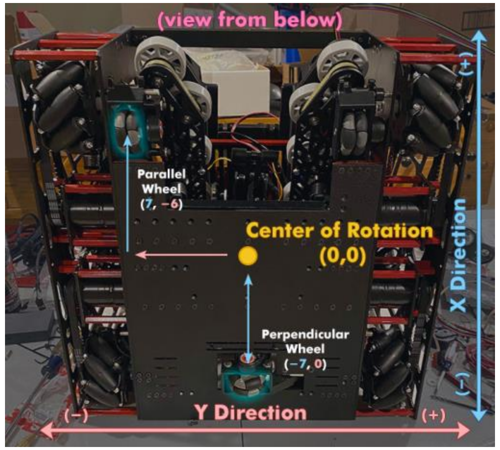
- FTC Dashboard should be considered a basic requirement for doing odometry on FTC
  - …whether you're using Road Runner, FTC Lib or writing all your own algorithms
- The live field view it provides is invaluable for visualization and debugging
  - It has other useful features too, but none as imperative as the field view



The Pose

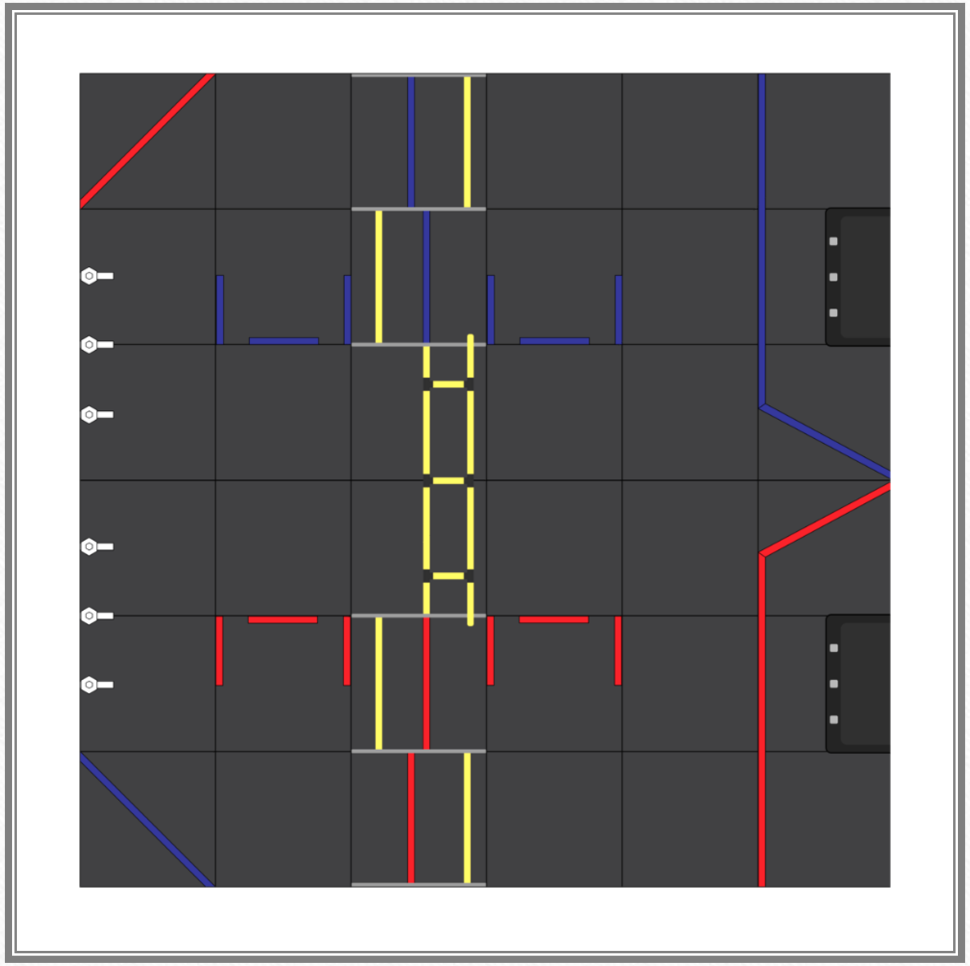
- Recall that odometry computes a *pose* estimate that is an (x, y) position plus heading
  - In FTC, it's best for the units to be inches and radians
- When a robot turns in place, its *heading* should change, but not its *position*
  - Therefore, a robot's **pose** is defined as representing its center of rotation
  - The center-of-rotation is not necessarily the same as the geometric center!





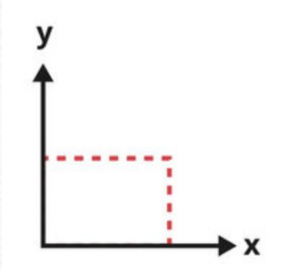
The Robot Coordinate Space

- Every odometry algorithm needs to know where sensors are on your robot…
- …so, there must be a *coordinate space* to describe those locations
- It might be a bit surprising, but when looking at the robot from above, the positive x axis goes straight out from the front of the robot…
- …and the positive y axis goes out the left side of the robot
- When looking from below, the y axis get flipped

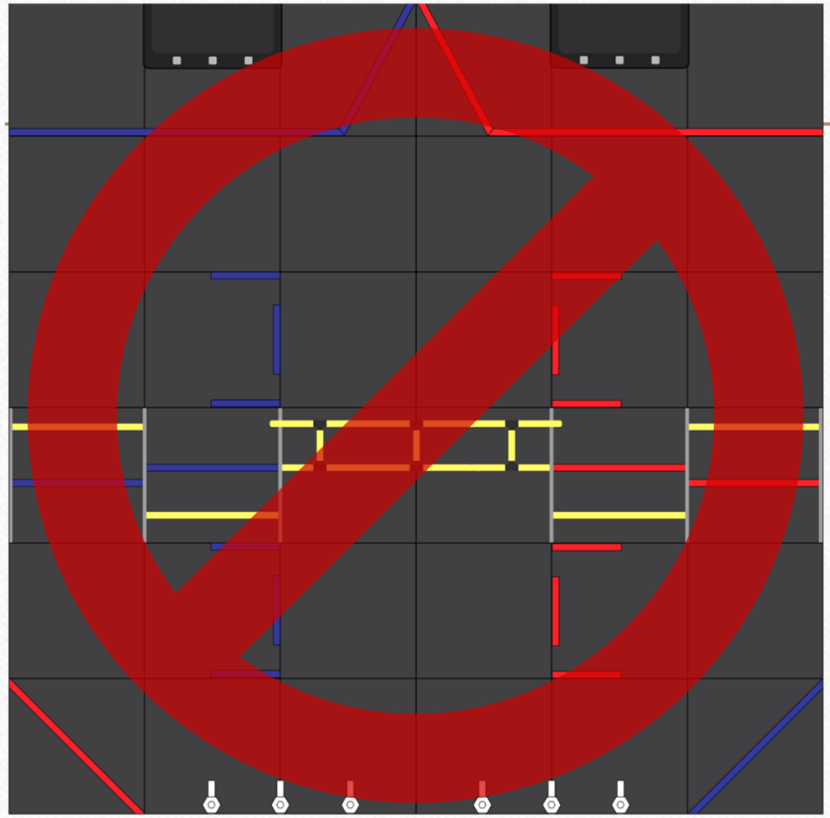


The Field Coordinate Space

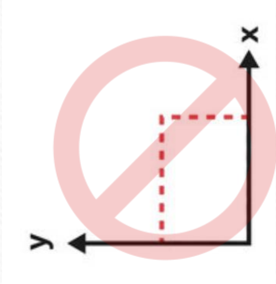
- The ***field*** (or ***world***) coordinate space is used to define the location and orientation of the robot on the field
- The red alliance station is clearly defined for every game so serves as the frame of reference for the field's orientation
- The (0, 0) origin is at the field's center
- The field extends from (-72, -72) to (72, 72), in inches
- The x axis goes left-right, the y axis goes bottom-up, as usual



FTC Dashboard's Field Representation



- FTC Dashboard's field view is wonderful
  - …except for the fact that in its default view, it rotates the field left by 90 degrees
- So, positive x is up, and positive y is to the left



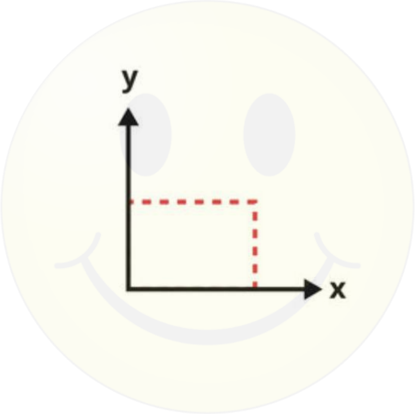
- This is always confusing when estimating positions and angles
- Just say no!

How to Fix FTC Dashboard's Field

- Apply this code at the beginning of any Dashboard canvas drawing to make the world right again!

```
TelemetryPacket packet = new TelemetryPacket();

// Prepare the packet for drawing.
//
// By default, Road Runner draws the field so positive y goes left, positive x
// goes up. Rotate the field clockwise so that positive y goes up, positive x
// goes right. This rotation is 90 (rather than -90) degrees in page-frame space.
// Then draw the grid on top and finally set the transform to rotate all subsequent
// rendering.
Canvas canvas = packet.fieldOverlay();
canvas.drawImage("/dash/centerstage.webp", 0, 0, 144, 144, Math.toRadians(90), 0, 144, true);
canvas.drawGrid(0, 0, 144, 144, 7, 7);
canvas.setRotation(Math.toRadians(-90));
```

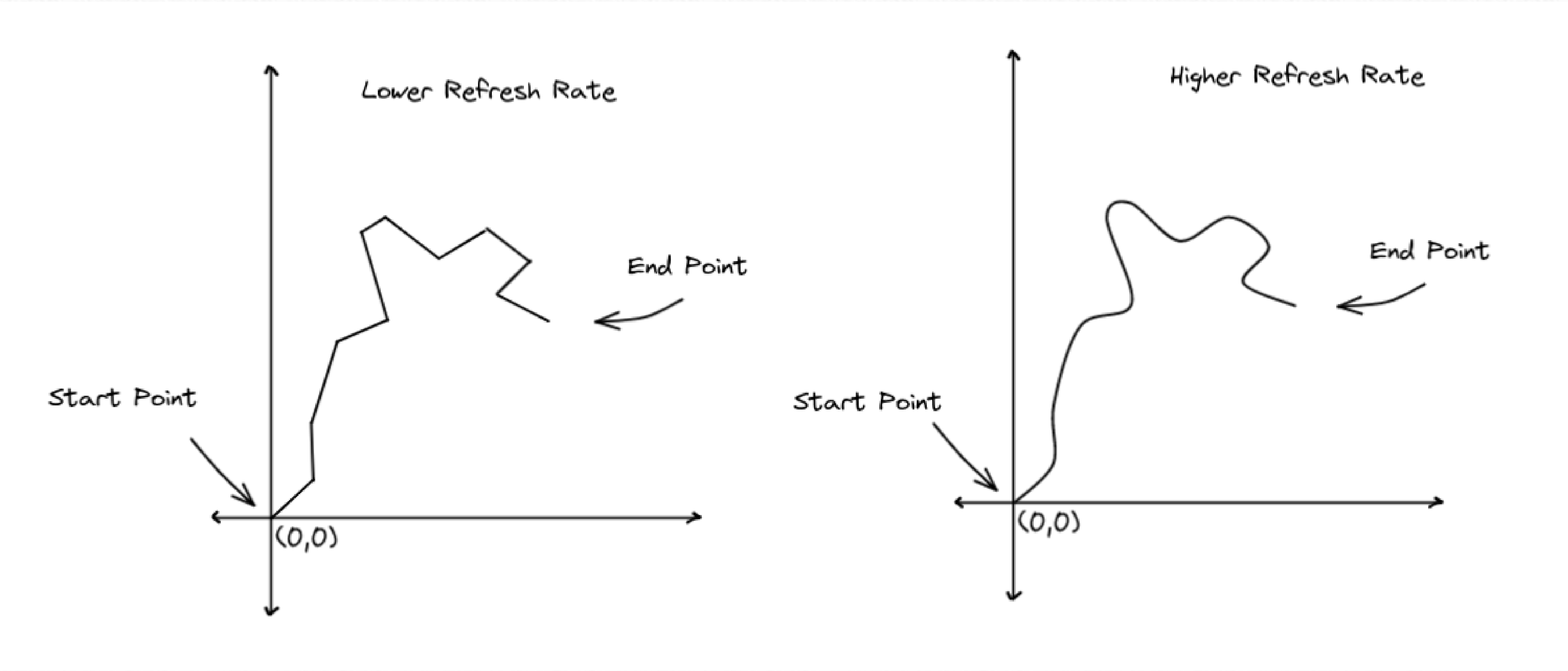


- You'll need to substitute "**intothedeep.webp**" (or possibly "**intothedeep.png**") for **centerstage.webp** once FTC Dashboard gets updated to support this year's game

Main Loop Performance

- At a high level, the main loop of your code looks like this when using odometry:
  1. Read sensors
  2. Update the motors
  3. Go to 1
- The performance of this loop greatly affects the accuracy of your odometry and traversal
  - You want as high a *refresh rate* as possible (just like in a video game!)…
  - …or equivalently, as low a *loop time* as possible

Low vs. High Refresh Rates



Measure Your Loop Time!

- Something like this:

```
// Update the loop time, in milliseconds, and show it on FTC Dashboard:
double lastLoopTime; // Seconds
void updateLoopTimeStatistic(TelemetryPacket p) {
   double currentTime = nanoTime() * 1e-9; // Seconds
   double loopTime = currentTime - lastLoopTime;
   lastLoopTime = currentTime;
   p.put("Loop time", loopTime * 1000.0); // Milliseconds
}
void resetLoopTimeStatistic() {
   lastLoopTime = nanoTime() * 1e-9;
}
```

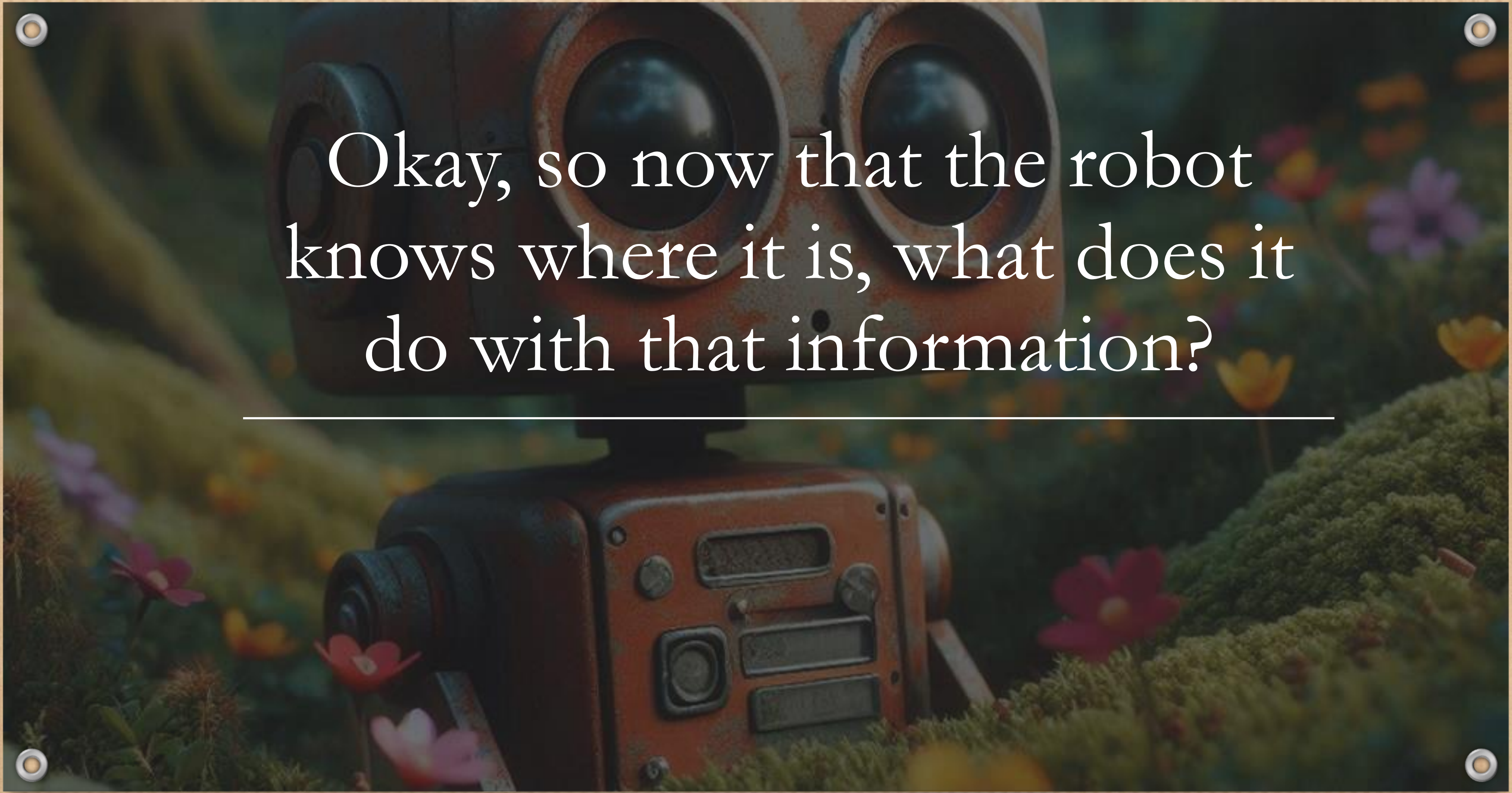
- You can view and graph the result in FTC Dashboard

Loop Time Targets

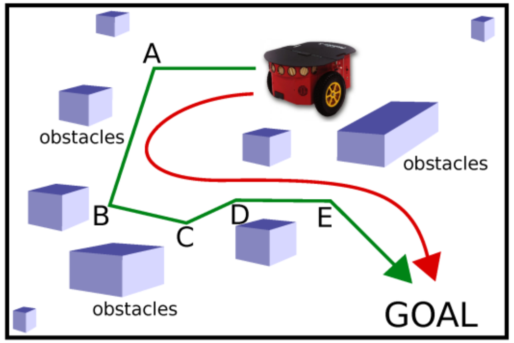
- The following should be your performance target:

3 dead wheels	2.5ms
2 dead wheels plus IMU/ wheel encoders plus IMU	8.4ms
OTOS	5.1ms

- These were measured in a loop reading the sensors and calling **SetPower()** to 4 motors
  - FTC run-time performance is highly variable so expect significant fluctuation in all your performance measurements
- If using Road Runner, approaching these times requires reducing the cost of Road Runner's call to **voltageSensor.getVoltage()**
  - This can be done by reading the sensor only every 100ms or so



Okay, so now that the robot knows where it is, what does it do with that information?



Trajectory Traversal!

- Use odometry to follow *trajectories*
- A ***trajectory*** is a **path** that is defined by a sequence of positions and orientations that the robot must achieve at specific times
- A ***path*** is a sequence of positions represented by lines, curves, and turns that define how a robot moves from a start to a goal
- A ***motion profile*** defines how a robot’s position, velocity and acceleration change over time to achieve smooth and controlled movement
- Writing your own trajectory traversal code is great fun..
- …or you can use Road Runner for FTC for a pre-built version that supports curved paths, motion profiles and more

Road Runner for FTC

- Road Runner is a library for FTC whose claim to fame is that it supports curved paths for trajectories
  - It has other useful features, such as support for concurrency (in the form of *Actions)*
- It supports tank drives and Mecanum drives
  - …and has odometry support for 0, 2 or 3 dead wheels
- It implements a Ramsete controller to smoothly correct for deviations from the path (like when your robot gets bumped)
- Road Runner was recently updated to version 1.0 last year
  - It's now very different from the now-deprecated version 0.5.x which was widely used in FTC
  - When looking at resources for Road Runner, be very careful to check what version they're referring to!

The Good and Not So Good of Road Runner

The Good

- Powerful feature set
- Robust tuning
- Good visualizer
- Popular in FTC
- Very responsive author

The Not So Good

- Documentation is sparse
- Few source code comments
- Kotlin language use can be arcane
- Not much error handling
- Plenty of bugs and weird behaviors
- Precomputes everything, leaving performance on the table and consuming significant memory
- Not usable in TeleOp
- Not as fun as writing it yourself
- OTOS support is a hack

OTOS Hack for Road Runner
Change **updatePoseEstimate()** like this:

```java
public PoseVelocity2d updatePoseEstimate() {
   PoseVelocity2d poseVelocity;
   if (opticalTracker != null) {
      // Get the current pose and current pose velocity from the optical tracking sensor.
      SparkFunOTOS.Pose2D position = new SparkFunOTOS.Pose2D(0, 0, 0);
      SparkFunOTOS.Pose2D velocity = new SparkFunOTOS.Pose2D(0, 0, 0);
      SparkFunOTOS.Pose2D acceleration = new SparkFunOTOS.Pose2D(0, 0, 0);

      // Note that this single call is faster than separate calls to getPosition()
      // and getVelocity(), even if we don't use the acceleration:
      opticalTracker.getPosVelAcc(position, velocity, acceleration);

      // Road Runner requires the pose to be field-relative while the velocity has to be
      // robot-relative, but the optical tracking sensor reports everything as field-
      // relative. As such, convert the velocity to be robot-relative by rotating it
      // by the negative of the robot's current heading:
      double rotation = -position.h;
      poseVelocity = new PoseVelocity2d(
            new Vector2d(Math.cos(rotation) * velocity.x - Math.sin(rotation) * velocity.y,
                      Math.sin(rotation) * velocity.x + Math.cos(rotation) * velocity.y),
         velocity.h);

      pose = new Pose2d(position.x, position.y, position.h);
```

```java
   } else {
      // Use the wheel odometry to update the pose:
      Twist2dDual<Time> twist = WilyWorks.localizerUpdate();
      if (twist == null) {
         twist = localizer.update();
      }

      pose = pose.plus(twist.value());
      poseVelocity = twist.velocity().value();
   }

   poseHistory.add(pose);
   while (poseHistory.size() > 100) {
      poseHistory.removeFirst();
   }

   estimatedPoseWriter.write(new PoseMessage(pose));

   return poseVelocity;
}
```

OTOS Hack, Part 2
Add these to MecanumDrive/TankDrive

```
// Override the current pose for Road Runner and the optical tracking sensor:
public void setPose(Pose2d pose) {
   // Set the Road Runner pose:
   this.pose = pose;
   this.targetPose = pose;

   // Set the pose on the optical tracking sensor:
   if (opticalTracker != null) {
      opticalTracker.setPosition(new SparkFunOTOS.Pose2D(
            pose.position.x, pose.position.y, pose.heading.toDouble()));
   }
}
```

```
public SparkFunOTOS opticalTracker = null; // Can be null which means no optical tracking sensor
```

OTOS Hack, Part 3

- In MecanumDrive/TankDrive's **configure()** method:
  1. Create the optical tracker object and assign it to the new **opticalTracker** field, e.g.:

     ```
     opticalTracker = hardwareMap.get(SparkFunOTOS.class, "optical");
     ```

  2. Initialize the OTOS like the **configureOtos()** method does in the SensorSparkFunOTOS sample that can be found in the SDK's external.samples section
  3. Call the new **setPose()** method with the current pose to put Road Runner and OTOS into agreement

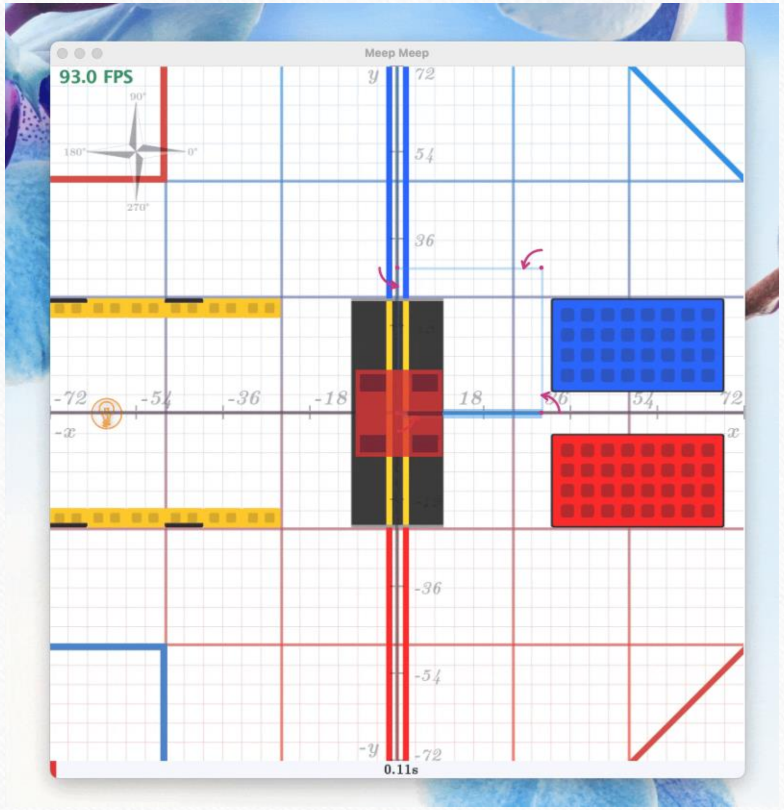
     ```
     setPose(pose);
     ```

OTOS Hack, The Big Caveat

- With this hack, you'll still have to tune Road Runner using encoders!
- The Road Runner tuning routines are hard wired to use encoders and will need to be rewritten at some point for OTOS
- The hack still needs Road Runner to have been tuned for all its parameters such as **kS, kV, kA, lateralInPerTick, trackWidthTicks** and the gains
- So, for now, use the motor encoders when using the Road Runner tuning utilities and use OTOS when using the Road Runner APIs
  - No need for dead wheels
  - Tuning doesn't suffer from any of the motor encoder vs. dead wheel drawbacks
  - This solution still provides all the advantages of OTOS for actual Road Runner operation

Visualizing Road Runner

- MeepMeep is an essential tool for Road Runner
- It allows you to visualize and debug your usage of Road Runner on your PC rather than with your robot
  - Super valuable to make software progress when the hardware folks are monopolizing the robot…
  - …and prevents a lot of potential damage to your robot!
- If you factor your code right, you can run a lot of your game's Road Runner logic directly on the PC



Road Runner Resources

- https://rr.brott.dev/docs/v1-0/tuning/ has all the official Road Runner documentation
  - It’s terse and insufficient, but correct
- www.learnroadrunner.com applies to the old, very different version, but is still the best place to learn the ideas behind Road Runner
  - Just don’t expect the descriptions of APIs or tuning to be remotely like the current version

Mistakes and Challenges

Common Odometry Pitfalls

- Not using FTC Dashboard
- Not understanding the difference between the *target* pose and the *actual* pose, and why there are two robots in the FTC Dashboard view
- Not rerunning Road Runner tuning after major changes to the robot
- Not benchmarking main loop performance
- Not remembering that the robot's "position" is its center-of-rotation
- Current pose and trajectory's start pose don't match in Road Runner

Odometry Challenges

- Can you combine odometry with other tracking techniques, such as April Tags or distance sensors?
- Can you use odometry for assists in TeleOp for *Into the Deep*?


That's all Folks!

Wait, don't forget…

- Check out Spark Fun's optical odometry sensor
- Always use FTC Dashboard when doing odometry
- Think about writing your own trajectory code
  - …but if you opt for Road Runner, be sure to use MeepMeep
- Benchmark the performance of your sensor loop
- Consider ways you could use odometry in TeleOp
- Find these slides at //github.com/SwerveRobotics/shared

Questions?

http://github.com/SwerveRobotics/shared