FRC Swerve Programming with Command-Based Java WPILib - 2024

MK4i Swerve Modules (code could be modified for other hardware)

Create new subsystem, SwerveModule.java

- Inside the SwerveModule class, create driveMotor, rotationMotor, driveEncoder, rotationEncoder, canCoder, rotationPidController objects
    - Also create a double canCoderOffsetRadians (required to align wheels, will get into this later)

```java
private final CANSparkMax driveMotor;
private final CANSparkMax rotationMotor;

private final RelativeEncoder driveEncoder;
private final RelativeEncoder rotationEncoder;

private final CANcoder canCoder;
private final double canCoderOffsetRadians;

private final PIDController rotationPidController;
```

- Create the constructor for the SwerveModule class. Pass in drive, rotation, and cancoder ID's, as well as the canCoderOffsetRadians.
    - Assign correct port & motortype to each motor.
    - Grab encoders for each motor and assign to encoder objects.
    - Assign correct port to the cancoder and map the input offset to the declared offset.
    - Assign position conversion factors to both the drive and rotation encoders.
        - We are trying to convert revolutions of the wheels to radians. 2π is treated as one full revolution. One full revolution on the wheel is the gear ratio, therefore we 2π by the gear ratio.
    - Assign velocity conversion factors to both the drive and rotation encoders.
        - We are trying to convert native units of the encoder (rpm) to angular velocity units (rad/s). Work this out using dimensional analysis if needed.
    - Tell the PIDController that $-\pi$ is the same as $+\pi$. This allows us to calculate a shorter path to the setpoint.
    - Create a configuration for the CANCoder
        - Make the range of the sensor 0-1 so that radians can be calculated
        - Make turning counterclockwise positive (as in the unit circle)
        - Apply the configuration

```java
public SwerveModule(
        int driveMotorId,
        int rotationMotorId,
        int canCoderId,
        double canCoderOffsetRadians
) {
        driveMotor = new CANSparkMax(driveMotorId, MotorType.kBrushless);
        rotationMotor = new CANSparkMax(rotationMotorId, MotorType.kBrushless);

        driveEncoder = driveMotor.getEncoder();
        rotationEncoder = rotationMotor.getEncoder();

        canCoder = new CANcoder(canCoderId);
        this.canCoderOffsetRadians = canCoderOffsetRadians;

        driveEncoder.setPositionConversionFactor(2.0 * Math.PI /
SwerveConstants.driveGearRatio);
        rotationEncoder.setPositionConversionFactor(2.0 * Math.PI /
SwerveConstants.rotationGearRatio);

        driveEncoder.setVelocityConversionFactor(2.0 * Math.PI / 60 /
SwerveConstants.driveGearRatio);
        rotationEncoder.setVelocityConversionFactor(2.0 * Math.PI / 60 /
SwerveConstants.rotationGearRatio);

        rotationPidController.enableContinuousInput(-Math.PI, +Math.PI);

        CANcoderConfiguration canCoderConfig = new CANcoderConfiguration();
        canCoderConfig.MagnetSensor.AbsoluteSensorRange =
AbsoluteSensorRangeValue.Unsigned_0To1;
        canCoderConfig.MagnetSensor.SensorDirection =
SensorDirectionValue.CounterClockwise_Positive;
        canCoder.getConfigurator().apply(canCoderConfig);
}
```

- Next, some useful methods. Some are getters and setters, others serve slightly more of a purpose. See descriptions of each one below. Note: these go BELOW the constructor, but not outside of the class.
    - getDriveEncoderPosition: fetches raw position of the drive encoder for a specific module.
    - getRotationEncoderPosition: fetches raw position of the rotation encoder for a specific module. Extra code required to account for the closest angle to the raw

position. If you look on the unit circle for example, a negative angle also has an identical positive angle.

- getCANcoder: simple getter method that returns the canCoder object for a specific swerve module
- getDriveVelocity & getRotationVelocity do the same thing, simple getter methods which return the velocity from the encoders, with the conversion factor applied
- getRotationMotor & getDriveMotor do the same thing, simple getter methods which return the rotationMotor or driveMotor object depending on the method called
- getCANcoderRad: fetches the actual amount of rotation from each motor by getting the CANCoder position offset (from center) and subtracting from the current rotation.
- resetEncoders: setter method which sets the drive encoder position to 0, and the rotation encoder position to the accounted offset position
- getState: getter method which fetches the current SwerveModuleState, the SwerveModuleState class does calculations on what the motors need to do (hence why it is a subclass of the kinematics class). Will come in use throughout the rest of the swerve drivetrain program.
- setDesiredStates: actually applies a SwerveModuleState
    - Optimizes to find the closest path to the target angle
    - Scales driveMotor speed to the max speed
    - Use PID for the rotation motor setpoint to avoid overshooting

```java
public double getDriveEncoderPosition() {
	return driveEncoder.getPosition();
}

public Rotation2d getRotationEncoderPosition() {
	double unsignedAngle = rotationEncoder.getPosition() % (2 * Math.PI);
	if (unsignedAngle < 0) unsignedAngle += 2 * Math.PI;
	return new Rotation2d(unsignedAngle);
}

public CANcoder getCANcoder() {
	return canCoder;
}

public double getDriveVelocity() {
	return driveEncoder.getVelocity();
}
```

```
public double getRotationVelocity() {
	return rotationEncoder.getVelocity();
}

public CANSparkMax getDriveMotor() {
	return driveMotor;
}

public CANSparkMax getRotationMotor() {
	return rotationMotor;
}

public Rotation2d getCANcoderRad() {
	double canCoderRad = (Math.PI * 2 *
canCoder.getAbsolutePosition().getValueAsDouble()) - canCoderOffsetRadians % (2
* Math.PI);

	return new Rotation2d(canCoderRad);
}

public void resetEncoder() {
	driveEncoder.setPosition(0);
	rotationEncoder.setPosition(getCANcoderRad().getRadians());
}

public SwerveModuleState getState() {
	return new SwerveModuleState(getDriveVelocity(), getCANcoderRad());
}

public void setDesiredStates(SwerveModuleState state) {
	state = SwerveModuleState.optimize(state.getRotationEncoderPosition());
	driveMotor.set(state.speedMetersPerSecond /
SwerveConstants.maxMetersPerSecond);

	rotationMotor.set(rotationPidController.calculate(getRotationEncoderPosit
	ion().getRadians(), state.angle.getRadians()));
}
```

Create another subsystem, SwerveSubsystem.java

- Declare all 4 swerve modules using your SwerveModule class
- Declare NavX object
- Within the constructor:
    - Set inversions as necessary

    - Reset encoders using that setter method that we created earlier
- Other methods within the subsystem class:
    - zeroHeading: resets the navX gyro heading
    - getHeading: returns rotation2d of navX heading
    - setModuleStates: set all modules at once
        - Makes sure speed never goes above max speed in constants
        - Set speed and rotation of each module

Java

```java
public class SwerveSubsystem extends SubsystemBase {
 /** Creates a new SwerveSubsystem. */
 private final SwerveModule frontLeft = new SwerveModule(
   SwerveConstants.frontLeftDriveMotorId,
   SwerveConstants.frontLeftRotationMotorId,
   SwerveConstants.frontLeftCanCoderId,
   SwerveConstants.frontLeftOffsetRad);

 private final SwerveModule frontRight = new SwerveModule(
   SwerveConstants.frontRightDriveMotorId,
   SwerveConstants.frontRightRotationMotorId,
   SwerveConstants.frontRightCanCoderId,
   SwerveConstants.frontRightOffsetRad);

 private final SwerveModule backLeft = new SwerveModule(
   SwerveConstants.backLeftDriveMotorId,
   SwerveConstants.backLeftRotationMotorId,
   SwerveConstants.backLeftCanCoderId,
   SwerveConstants.backLeftOffsetRad);

 private final SwerveModule backRight = new SwerveModule(
   SwerveConstants.backRightDriveMotorId,
   SwerveConstants.backRightRotationMotorId,
   SwerveConstants.backRightCanCoderId,
   SwerveConstants.backRightOffsetRad);

 private final AHRS navX = new AHRS(SPI.Port.kMXP);
```

```
public SwerveSubsystem() {
  // inversions -- need to figure these out
  frontLeft.getDriveMotor().setInverted(false);
  frontRight.getDriveMotor().setInverted(false);
  backLeft.getDriveMotor().setInverted(false);
  backRight.getDriveMotor().setInverted(false);

  // reset encoders upon each start
  frontLeft.resetEncoders();
  frontRight.resetEncoders();
  backLeft.resetEncoders();
  backRight.resetEncoders();
}

public void zeroHeading() {
  navX.reset();
}

public Rotation2d getHeading() {
  return Rotation2d.fromDegrees(navX.getYaw());
}

public void setModuleStates(SwerveModuleState[] desiredStates) {
  // makes it never go above 5 m/s
  SwerveDriveKinematics.desaturateWheelSpeeds(desiredStates,
SwerveConstants.maxMetersPerSecond);
  // Sets the speed and rotation of each module
  frontLeft.setDesiredStates(desiredStates[0]);
  frontRight.setDesiredStates(desiredStates[1]);
  backLeft.setDesiredStates(desiredStates[2]);
  backRight.setDesiredStates(desiredStates[3]);
}
```