8027

NOT THE DROIDS YOU ARE LOOKING FOR

2020 FRC Java Control System

Introductions

- We were one of 25 teams using Java invited to beta test the 2020 FRC Control System

Changes

- Documentation
- New Command Based framework
- Synchronous PID Controller
- Kinematics classes
- Path Planner
- C++/Java Simulator UI

ScreenSteps → frc-docs

- The new documentation can be found at https://frc-docs.readthedocs.io/, which has replaced the ScreenSteps documentation
- Much of the 2020 changes are in the documentation already

Command Framework

Lessons Learnt in Porting Code

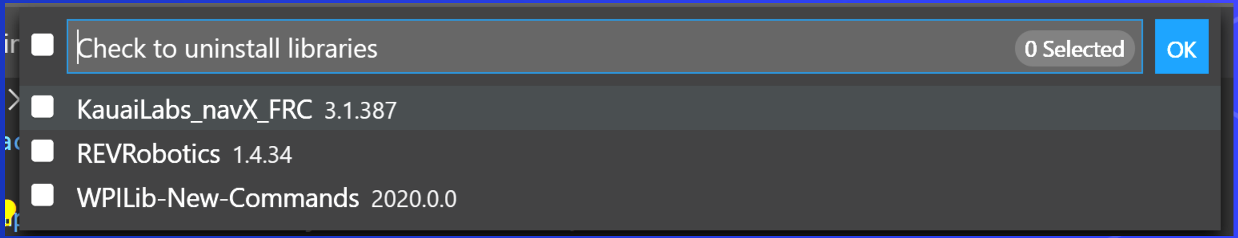
- Easier to cut and paste existing commands into a working 2020 framework than trying to patch up a 2019 code base
  - May depend on size of your existing code base
- Existing code will continue to work for 2020 (i.e. support for old command structure).
  - However, it is deprecated → so good time to start switching

Command-Based Framework

- The framework was rewritten for the following reasons:
  - Readability and maintainability
  - Encapsulation and separation of responsibilities
  - Restrictive API design
  - Clutter

Command Framework Location

- The new framework is located in the frc2 namespace for C++ and the edu.wpi.first.wpilibj2 package in Java.
- The command framework is a separate vendor library
  - Can have old or new command framework installed for a project
- Examples:
  - `import edu.wpi.first.wpilibj2.command.CommandBase;`
  - From manage vendor libraries in VScode



Commands & Subsystems

- Command (Java, C++) and Subsystem (Java, C++) are both now interfaces as opposed to abstract classes
- Recommended method is to subclass the abstract CommandBase and SubsystemBase class
  - public class ExampleCommand extends CommandBase {
  - public class ExampleSubsystem extends SubsystemBase {

New Basic Structure

The root package/directory generally will contain four classes and two directories:

- **Main**, which is the main robot application. Most users should not touch this class.
- **Robot**, which is responsible for the main control flow of the robot code.
- **RobotContainer**, which holds robot subsystems and commands, and is where most of the declarative robot setup (e.g. button bindings) is performed.
- **Constants**, which holds globally-accessible constants to be used throughout the robot.
- **Subsystems** directory
- **Commands** directory

Robot.java

- Construct RobotContainer in robotInit()
- CommandScheduler.getInstance().run() call in the robotPeriodic() to run commands
- The autonomousInit() method schedules an autonomous command returned by the RobotContainer instance. However, logic for selecting autonomous command to run can be handled inside of RobotContainer
- The teleopInit() method cancels any still-running autonomous commands. This is essentially the same as before.

RobotContainer.java

- Most of the robot setup/customization
- Create subsystems:

```
private final ExampleSubsystem m_exampleSubsystem = new
ExampleSubsystem();
```

  - Notice that subsytems are “private” unlike past years
- Must pass needed subsystems to commands (called “dependency injection”)

```
private final ExampleCommand m_autoCommand = new
  ExampleCommand(m_exampleSubsystem);
```

- Button Bindings → no more OI.java

Constants.java

- Place for useful constants such as speeds, unit conversion factors, PID gains, and sensor/motor ports
- All constants should be declared public static final so that they are globally accessible and cannot be changed
- An import static statement imports the static namespace of a class into the class in which you are working

```
import static
edu.wpi.first.wpilibj.templates.commandbased.Constants.OIConstants.*;
```

Other Changes: Command Groups

- CommandGroup class no longer exists – replaced with:
  - SequentialCommandGroup, ParallelCommandGroup
  - ParallelRaceGroup (ends when any subcommand finishes)
  - ParallelDeadlineGroup (ends when specific subcommand finishes)
- Each implements Command interface so can be composed

Other Changes: Composing Commands

```
new SequentialCommandGroup(
  new DriveToGoal(m_drive),
  new ParallelCommandGroup(new RaiseElevator(m_elevator),
    new SetWristPosition(m_wrist)),
  new ScoreTube(m_wrist));
```

Other Changes: Inline Commands

- Simplifies “small” commands with single use

```
private void configureButtonBindings() {
  // Grab the hatch when the 'A' button is pressed.
  new JoystickButton(m_driverController, Button.kA.value)
      .whenPressed(new InstantCommand(m_hatchSubsystem::grabHatch,
                             m_hatchSubsystem));
```

- Note method reference is object::method
- Especially useful with InstantCommand

Other Changes

- requires() method has been renamed to addRequirement()
- interrupted() method has been rolled into the end() method, which now takes a parameter specifying whether the command was interrupted (false if it ended normally).

PID Controls

PID Controller

- Old PIDController Class created a separate thread that read PIDSource and wrote PIDOutput periodically.
- New PIDController runs synchronously from main robot loop
- Example:

```
public class ShooterSubsystem extends PIDSubsystem {
 public ShooterSubsystem() {
  super(new PIDController(kP, kI, kD));
  getController().setTolerance(kShooterToleranceRPS);
  m_shooterEncoder.setDistancePerPulse(
    kEncoderDistancePerPulse);
  setSetpoint(kShooterTargetRPS);
 }
```

PID Controller

- Also need to provide getMeasurement and useOutput

```
@Override
public void useOutput(double output, double setpoint) {
  m_shooterMotor.setVoltage(output +
      m_shooterFeedforward.calculate(setpoint));
}

@Override
public double getMeasurement() {
  return m_shooterEncoder.getRate();
}
```

Kinematics

Kinematics and Odometry

- Brand new
- Help convert between a universal **ChassisSpeeds** object, containing linear and angular velocities for a robot to usable speeds for each individual type of drivetrain i.e. left and right wheel speeds for a differential drive, four wheel speeds for a mecanum drive, or individual module states (speed and angle) for a swerve drive.

Chassis Speed

- **`vx`**: The velocity of the robot in the x (forward) direction (in meters/sec)
- **`vy`**: The velocity of the robot in the y (sideways) direction. (Positive values mean the robot is moving to the left) (in meters/sec)
  - Note: vy = 0 for non-holonomic drive
- **`omega`**: The angular velocity of the robot (in radians/sec)
- Can also use field relative measurements using `ChassisSpeeds.fromFieldRelativeSpeeds()`

Differential Drive Kinematics

- DifferentialDriveKinematics has one constructor argument → the track width of the robot.

```
DifferentialDriveKinematics kinematics = new
DifferentialDriveKinematics(Units.inchesToMeters(27.0));
// chassis speeds: 2 m/s speed,1 radian/s angular velocity.
var chassisSpeeds = new ChassisSpeeds(2.0, 0, 1.0);
// Convert to wheel speeds
DifferentialDriveWheelSpeeds wheelSpeeds =
kinematics.toWheelSpeeds(chassisSpeeds);
// wheel velocities
double leftVelocity = wheelSpeeds.leftMetersPerSecond;
double rightVelocity = wheelSpeeds.rightMetersPerSecond;
```

Differential Drive Odometry

- Constructor requires angle (as Rotation2d) and optionally field position (as Pose2d)
  - Facing opponent alliance = 0 degrees, turning left = positive degrees
- Example:

```
// our pose is 5 meters along the long end of field and
// center of the field along the short end, facing forward.
DifferentialDriveOdometry m_odometry = new
DifferentialDriveOdometry(
  getGyroHeading(), new Pose2d(5.0, 13.5, new Rotation2d());
```

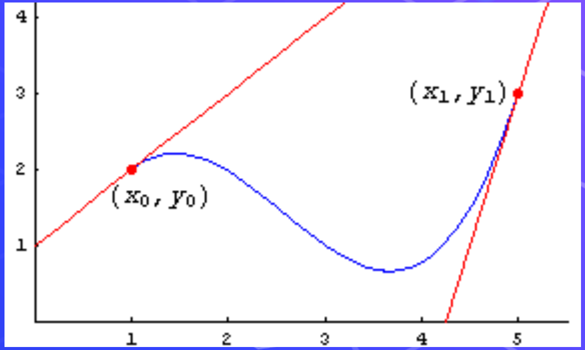
Updating Position

- Odometry update method should be called periodically (e.g. in subsystem periodic())
- Encoder distances should be in meters,

```
public void periodic() {
var gyroAngle = Rotation2d.fromDegrees(-m_gyro.getAngle());
// Update the pose
m_pose = m_odometry.update(gyroAngle,
m_leftEncoder.getDistance(), m_rightEncoder.getDistance());
```

Trajectory Generation

- Splines: curves through a set of points. WPILib supports Hermite Clamped Cubic and Hermite Quintic
- Start by creating a TrajectoryConfig object
  - Arguments: maxVelocity, maxAcceleration
  - Change startVelocity, endVelocity, reversed, constraints with set* methods



Trajectory Generation

- generateTrajectory(…) creates a trajectory given a set of Pose2d's and a TrajectoryConfig
  - Number/type of parameters determine if you use cubic/quintic

Example Trajectory Generation

```
var sideStart = new Pose2d(Units.feetToMeters(1.54),
    Units.feetToMeters(23.23), Rotation2d.fromDegrees(-180));
var crossScale = new Pose2d(Units.feetToMeters(23.7),
    Units.feetToMeters(6.8), Rotation2d.fromDegrees(-160));
var interiorWaypoints = new ArrayList<Translation2d>();
interiorWaypoints.add(new Translation2d(Units.feetToMeters(14.54), 7.0));
interiorWaypoints.add(new Translation2d(Units.feetToMeters(21.04), 6.0));
TrajectoryConfig config = new TrajectoryConfig(Units.feetToMeters(12), 4);
var trajectory = TrajectoryGenerator.generateTrajectory(
        sideStart, interiorWaypoints, crossScale, config);
```

Following a Trajectory

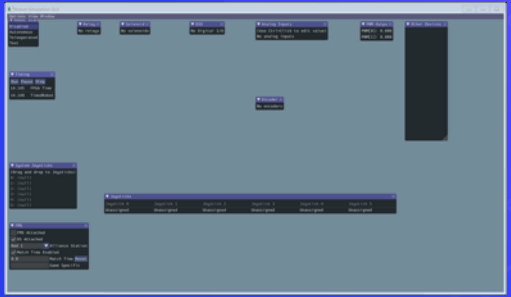
- Builtin Ramsete controller
  - calculate() method takes current position and Trajectory.state (i.e. goal) as inputs
- Example:

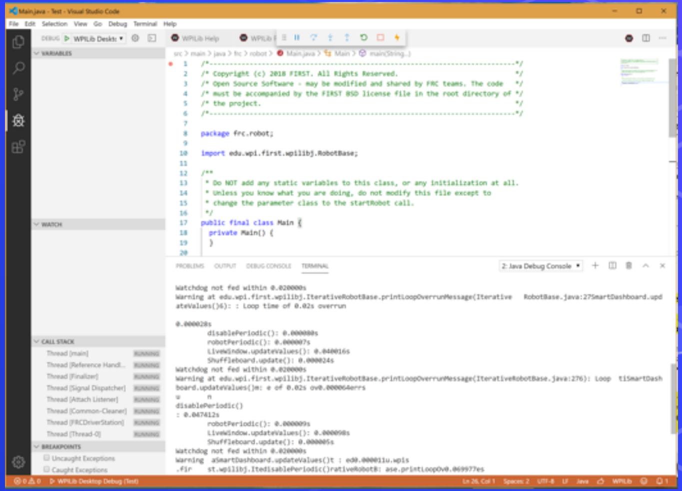
```
// sample the trajectory at 3.4 seconds from the beginning
Trajectory.State goal = trajectory.sample(3.4);
ChassisSpeeds adjustedSpeeds =
    controller.calculate(currentRobotPose, goal);
```

- Use kinematics classes to convert to wheel speeds
- Use PIDcontrol to change wheel speeds

FRC Java/C++ Simulator

- Works as a good debugger
- Allows you to view variables, threads, etc.
- Simulates motor controllers and various sensors defined





Thank you for attending