FRC Team 5940 Documentation

May 13, 2019

Welcome

CHAPTER 1

Welcome to Team 5940's Documentation!

Here lives documentation of Team 5940's code, including team-specific items. Please refer to the frc-docs companion documentation for WPILibJ and WPILibC resoureces.

This repository also includes documentation of Team 5190 Green Hope Robotics' FalconLibrary and FalconDashboard as an unnoficial resourece.

CHAPTER 2

FalconLibrary

2.1 Overview

Falcon Library is the backend library that is used on all FRC Team 5190 robots. This library was written in the Kotlin JVM Language. Some features of this library include:

- Wrapped WPILib Commands and Subsystems with Kotlin Coroutines asynchronous optimzation.
- High level mathematics for path generation, tracking, custom typesafe units of measure, etc.
  - Two-dimensional parametric and functional splines.
  - Arc length of parametric quintic hermite splines evaluated using recursive arc subdivision (from Team 254).
  - Trajectory generation that respects constraints (i.e. centripetal acceleration, motor voltage).
  - Custom trajectory followers
    - Ramsete
    - Adaptive Pure Pursuit
    - Feedforward
  - Typesafe units of measure
    - Quick and easy conversions between all length, velocity, acceleration, electrical units.
    - Support for Talon SRX native unit length and rotation models.
- AHRS sensor wrapper for Pigeon IMU and NavX.
- Tank Drive Subsystem abstraction with built-in odometry and command to follow trajectories.
- Talon SRX wrapper that utilizes Kotlin properties to set configurations.
- Custom robot base with fully implemented state machine and coroutine support.
- Other WPILib wrappers for NetworkTables, etc.

2.2 Java Interoperability

FalconLibrary is written in Kotlin, a new programming language based on the Java Virtual Machine. Kotlin code is fully interoperable with Java code. One caviat: static functions and members will be split into their own class. For instance, to construct a Java Length, one would do:

```
var x = LengthKt.getFeet(10);
```

2.3 Contributing

This library is open source and we would love to have you contribute code to this repository. Please make sure that before submitting a pull request, your code is formatted according to `ktlint` (already in the project). The Gradle build will fail if all code is not formatted correctly.

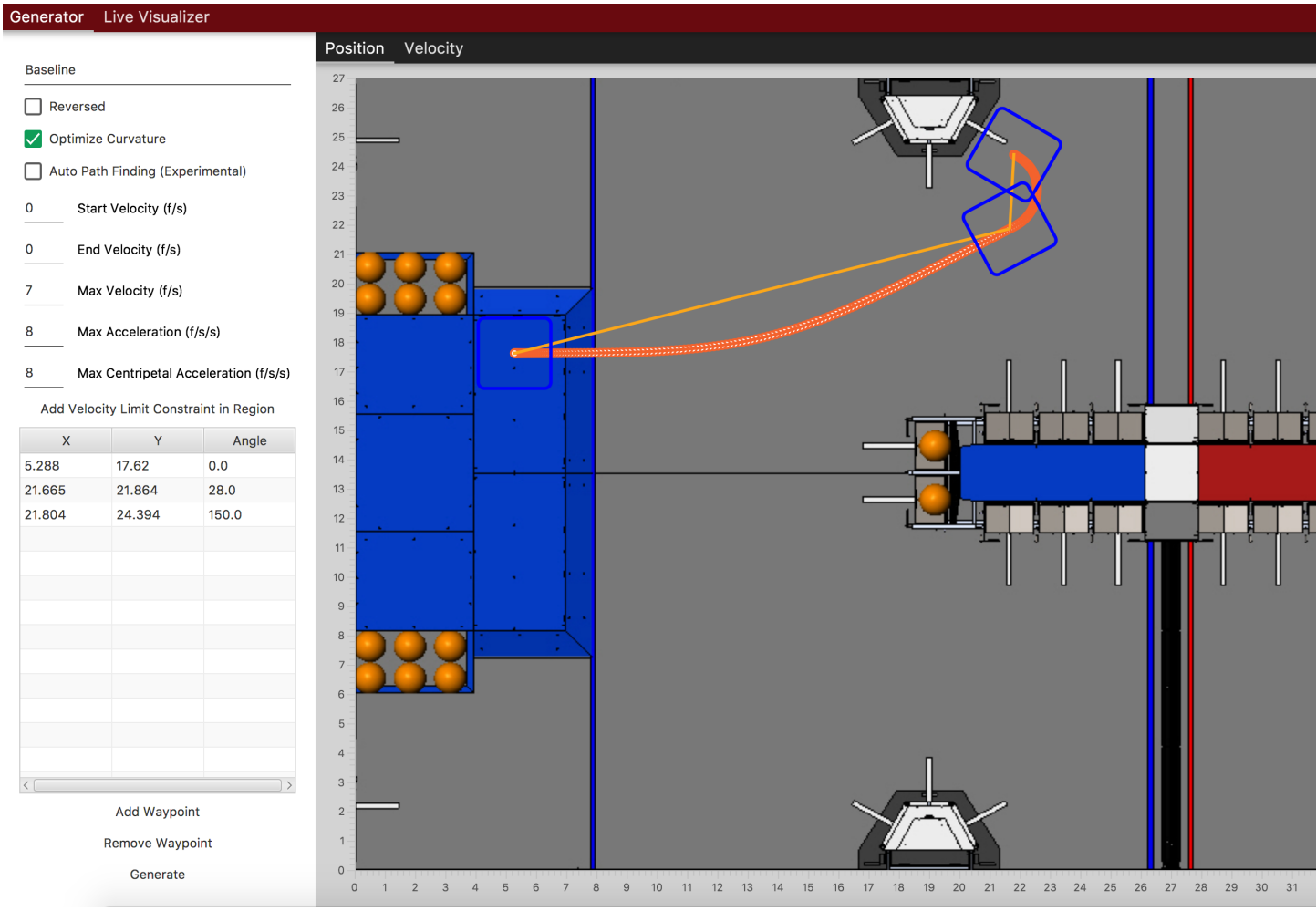
To format code automatically, run `./gradlew spotlessApply`. Please build the project locally using `./gradlew build` to make sure everything works before submitting a pull request.

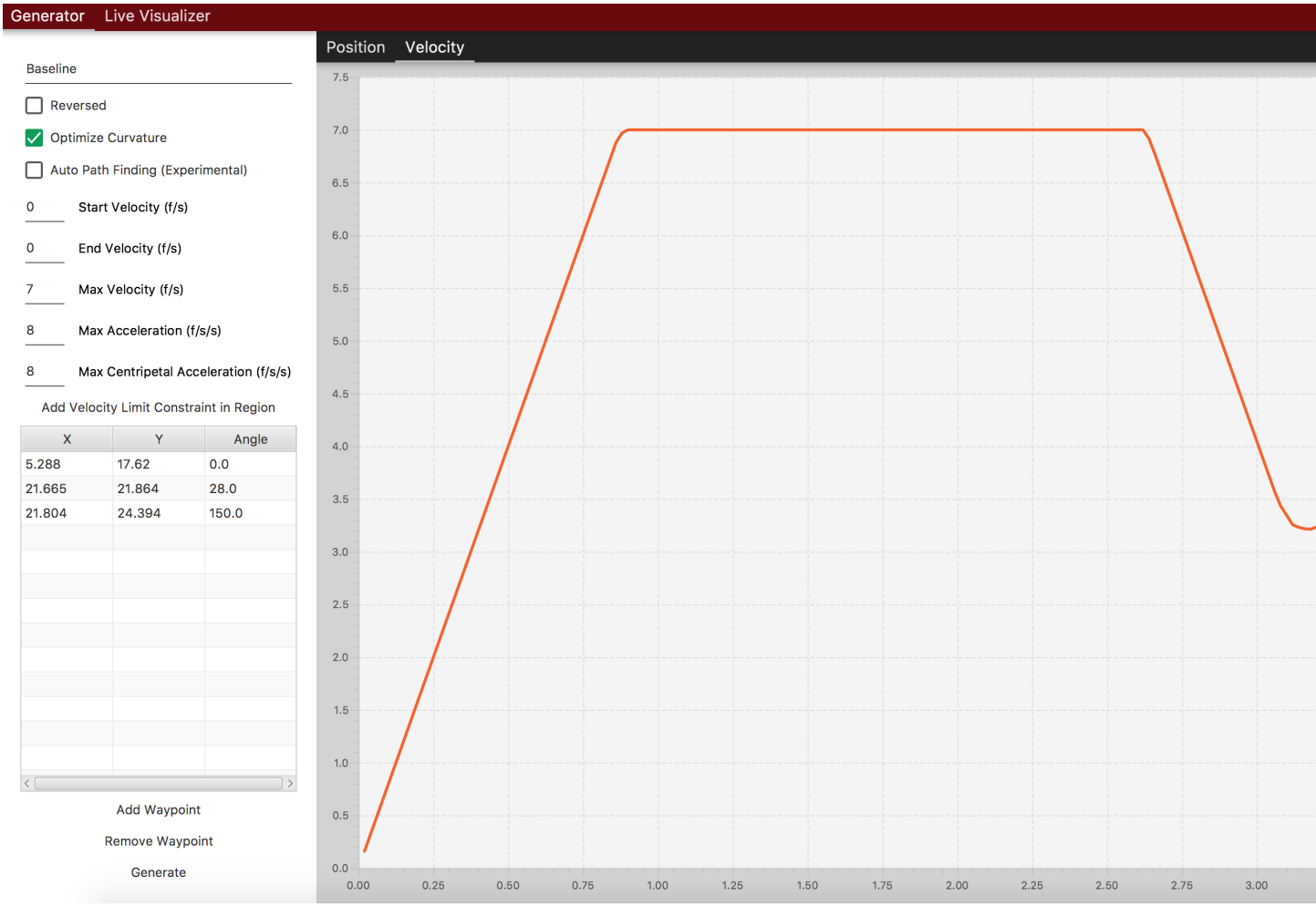
When adding new features, it is encouraged that these features be game-agnostic. This library is intended to be used for robots that play any game. Also make sure to include unit-tests for any new features.

CHAPTER 3

FalconDashboard

Falcon Dashboard is a Kotlin-based utility that can be used to generate trajectories and visualize the robot's position on the field live. This utility uses FalconLibrary as the backend trajectory generation code, and generates code that can be pasted into your robot code. The source code is publicly available at https://github.com/5190GreenHopeRobotics/FalconDashboard


Generator
Live Visualizer
Baseline
Reversed
Optimize Curvature
Auto Path Finding (Experimental)
0
Start Velocity (f/s)
0
End Velocity (f/s)
7
Max Velocity (f/s)
8
Max Acceleration (f/s/s)
8
Max Centripetal Acceleration (f/s/s)
Add Velocity Limit Constraint in Region
X
Y
Angle
5.288
17.62
0.0
21.665
21.864
28.0
21.804
24.394
150.0
Add Waypoint
Remove Waypoint
Generate
Position
Velocity

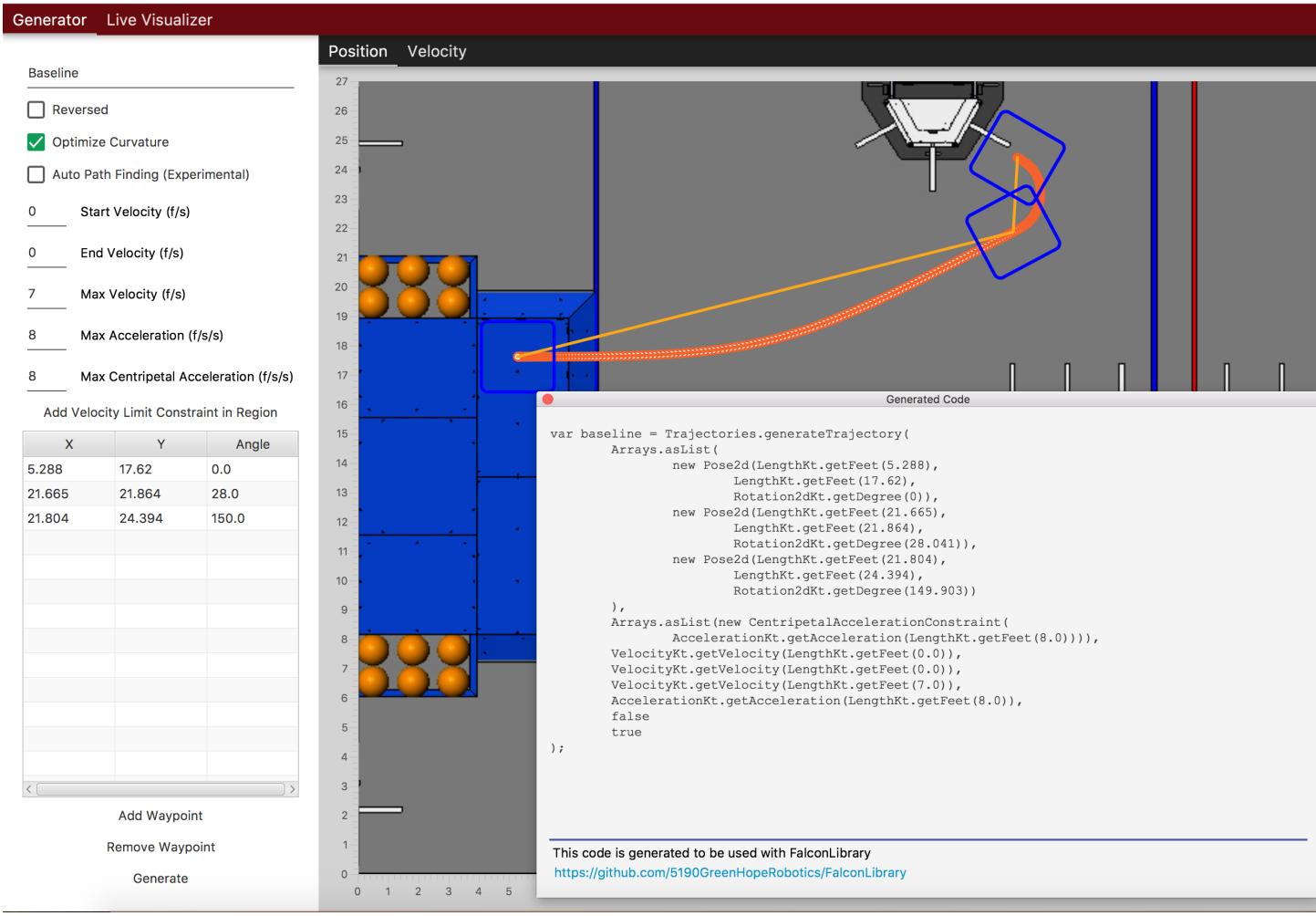


3.1 Running FalconDashboard

Clone or download the repository, and execute this command from within the project root directory:

```
./gradlew run
```

3.2 Generating code



To generate code, just click generate and paste the code into a command or command group. Specific implementation will vary user to user. See Following Trajectories (TODO Link) for more information. One example might be:

Java

```
var traject = Trajectories.generateTrajectory(
    Arrays.asList(
        new Pose2d(LengthKt.getFeet(5.634),
            LengthKt.getFeet(17.719),
            Rotation2dKt.getDegree(0)),
        new Pose2d(LengthKt.getFeet(18.36),
            LengthKt.getFeet(19.143),
            Rotation2dKt.getDegree(0)),
        new Pose2d(LengthKt.getFeet(20.822),
            LengthKt.getFeet(25.301),
            Rotation2dKt.getDegree(153.33))
    ),
    Arrays.asList(new CentripetalAccelerationConstraint(
        AccelerationKt.getAcceleration(LengthKt.getFeet(8.0)))),
    VelocityKt.getVelocity(LengthKt.getFeet(0.0)),
```

(continues on next page)

(continued from previous page)

```
    VelocityKt.getVelocity(LengthKt.getFeet(0.0)),
    VelocityKt.getVelocity(LengthKt.getFeet(7.0)),
    AccelerationKt.getAcceleration(LengthKt.getFeet(8.0)),
    false
    true
);

addSequential(new FollowPathCommand(traject, true, m_DriveBase);
```

Kotlin

```
// coming soon, coz i don't know Kotlin at all
```

CHAPTER 4

FalconLibrary Units and Math

4.1 Summery

FalconLibrary includes all common SI units and derived units as typesafe objects. This includes base units such as Length, Time and Voltage, as well as derived units such as Velocity and Acceleration.

All typesafe units include common mathematical operators such as unary plus, unary minus, equivalency checks, multiplication and division.

Note: To use these functions in Java, one must call `Rotation2dKt.getDegree(10).div(TimeUnitsKt.getSecond(10))` rather than `Rotation2dKt.getDegree(10) / TimeUnitsKt.getSecond(10)`. This does not apply to Kotlin users.

Note: These functions do not modify the class on which they are called, but instead return a new instance of it. So if I construct a new Length of 5 inches and call `.plus(x)` on it, that original length is still 5 inches.

All the base units contain the basic SI increments, such as milli-, micro-, nano, as well as kilo- or even yotta- and exa-. Additionally, units include (in general) the imperial equivalent, such as inches and feet or miles for Length or pounds for Mass. See your autocomplete for a full list of types.

4.2 Time

Time represents a time, and implements SIUnit. Use this to represent a passing time, a duration, or to construct derived units such as *Velocity* and *Acceleration*.'

Java

```
Time aTime = TimeUnitsKt.getSecond(10);
```

Kotlin

```
val aTime = 10.second

val anotherTime = 10.millisecond
```

4.3 Length

A Length represents a displacement in 1D space, and can be either positive or negative. The function `aLength.getValue()` will return the value in meters, as it is the base unit of Length in the SI system.

Java

```
// creating a Length
Length aLength = LengthKt.getFeet(10);
Length anotherLen = LengthKt.getMeter(3);

// getting a Length
double aLengthInInches = aLength.getInch();
double miles = anotherLen.getMile();
```

Kotlin

```
val aLen = 10.feet
val anotherLen = 3.meter

val inches = aLen.inch
val miles = anotherLen.mile
```

4.4 Rotation2d

A Rotation2d represents a rotation in 2d space. Think of it like an angle on a unit circle - it can represent the angle of the triangle's hypotenuse, and can be converted into a *Translation2d* with X and Y components correlated to the angle's sine and cosine components.

Java

```
Rotation2d rotation = Rotation2dkt.getDegree(45);

// returns true
var isParallel = rotation.isParallel(rotation);

Rotation2d aMultiple = rotation.times(4);

Rotation2d mMinus = rotation.minus(Rotation2dKt.getDegree(30);

Rotation2d mPlus = rotation.plus(Rotation2dKt.getDegree(-10);
```

Kotlin

```
val rotation = 45.degree

val isParallel = rotation.isParallel(rotation)
```

(continues on next page)

(continued from previous page)

```
val aMultiple = rotation.times(4)

val mMinus = rotation.minus(30.degree)

val mPlus = rotation.plus((-10).radian)
```

4.5 NativeUnit

NativeUnit, an SIValue, are often used on motor controllers with feedback sensors, such as TalonSRXes with Quadrature encoders. These encoders output a fixed number of pulses for every rotation of the shaft they are connected to, and a NativeUnit represents these pulses distinct from information encoding actual, real-life position measurements such as distance or angles. Because NativeUnit is a SIValue, it inherits the same common operators as Length and Rotation2d. Conversion between NativeUnits and physical units are done using the Native Unit Model abstract class. Included in FalconLibrary are `NativeUnitLengthModel` and `NativeUnitRotationModel` models, which covers both linear applications (for example, an elevator or slide) and angular applications (such as an arm). These models include methods to convert between native and physical unit positions, velocities, accelerations, and error, among other things.

Warning: The TalonSRX encodes velocity as ticks per 100ms, however other motor controllers such as the Spark Max encode rpm by default. Furthermore, most motor controllers will let you multiply their measured ticks by an arbitrary constant, so even if the Spark MAX says RPM, you may have it configured for RPS. Keep this in mind when using Length and Rotation models!

Java

```
NativeUnit someUnits = NativeUnitKt.getNativeUnits(10);
```

Kotlin

```
val nativeUnits = 10.nativeUnits
```

4.6 Velocity

Velocity, a derived unit, is often used to represent a linear or angular speed. However it is possible to make a Velocity of any type that impalements SIValue. The type of Velocity represented can be parameterized by any class that implements SIValue - for instance, a `Velocity<Length>`, or `Velocity<Rotation2d>`, or even `Velocity<NativeUnit>`.

Java

```
// a linear velocity
Velocity<Length> tenFeetPerSec = LengthKt.getFeet(10).div(TimeUnitsKt.getSecond(1));

// an angular velocity
Velocity<Rotation2d> tenDegPerSec = Rotation2dKt.getDegree(10).div(TimeUnitsKt.
↪getSecond(1));

double radPerSec = tenDegPerSec.getType$FalconLibrary().getRadian();
```

(continues on next page)

(continued from previous page)

```
Velocity<NativeUnit> ticksPerSec = NativeUnitKt.getNativeUnits(10).div(TimeUnitsKt.
↪getSecond(1));
```

Kotlin

```
val tenFeetPerSec = 10.feet / 1.second

val tenDegPerSec = 10.degree / 1.second

val ticksPerSec = 10.nativeUnits / 1.second

// TODO make this actually work in kotlin
val inRadiansPerSec = aVel.getType$FalconLibrary().getRadian();
```

4.7 Acceleration

Acceleration, a derived unit of Velocity, is used to represent either a linear or angular acceleration. Similar to Length, the type can be parameterized by any class that implements SIValue. Similar to Length, Acceleration must be parameterized by a class with inherits SIValue.

Java

```
Velocity<Length> tenFeetPerSecSquared = LengthKt.getFeet(10).div(TimeUnitsKt.
↪getSecond(1)).div(TimeUnitsKt.getSecond(1));

Velocity<Rotation2d> nyooooomAccel = Rotation2dKt.getDegree(10000).div(TimeUnitsKt.
↪getSecond(1)).div(TimeUnitsKt.getSecond(1));

Velocity<NativeUnit> fastNativeUnitNyoom = NativeUnitKt.getNativeUnits(10000).
↪div(TimeUnitsKt.getSecond(1)).div(TimeUnitsKt.getSecond(1));
```

Kotlin

```
val tenFeetPerSecSquared = 10.feet / 1.second / 1.second

val angularAccel = 10000.degree / 1.second / 1.second

val fastNativeUnitNyoom = 1000000.nativeUnits / 1.second / 1.second
```

4.8 Translation2d

A Translation2d is similar to a 2d vector. It can be constructed either with a typesafe magnitude and direction, or from x and y components, or from the displacement between two other Translation2ds. Translation2d is also special because it implements VaryInterpolatable, which means that you can linearly interpolate between two Translation2ds. This is very useful for path following.

Java

```
// This is assumed to be meters
Translation2d tran = new Translation2d(
      4, 5
```

(continues on next page)

(continued from previous page)

```
);

// This is a typesafe translation
tran = new Translation2d(
      LengthKt.getInch(4),
      LengthKt.getFeet(10)
);

// make a Translation2d out of essentially a vector
tran = new Translation2d(
      LengthKt.getFeet(20),
      Rotation2dKt.getDegree(21)
);

// This will have a "norm" of 1 meter
Translation2d anotherTran = Translation2dKt.fromRotation(Rotation2dkt.getDegree(45));

// return the point interpolated half way between these two points
var interpolated = tran.interpolate(anotherTran, 0.5);

// get the Length of the hypotenuse of this
var hypotenuseLength = tran.norm();
```

Kotlin

```
// This is assumed to be meters
val tran = Translation2d(4, 5);
val tran = Translation2d(4.feet, 10.meter)

// make a Translation2d out of essentially a vector
val tran = Translation2d(5.feet, 21.degree)

// This will have a "norm" of 1 meter
val anotherTran = Translation2d.fromRotation(45.degree)

// return the point interpolated half way between these two points
val interpolated = tran.interpolate(anotherTran, 0.5);

// get the Length of the hypotenuse of this
val hypotenuseLength = tran.norm()
```

4.9 Pose2d

Pose2d is a composition of Translation2d and Rotation2d. It represents a point in 2 dimensional space with an associated heading, for example,

Java

```
var pose = new Pose2d(LengthKt.getInch(5), LengthKt.getInch(5), Rotation2dKt.
↪getDegree(45);
```

Kotlin

```
val pose = Pose2d(Translation2d(5.feet, 2.inch), 45.degree)
```

This unit is also really useful for path following, and is used to represent a robot's 2d position on the field and a heading. The type also includes methods such as `.mirror()`, which mirrors the Pose2d about the middle of the field (left/right, relative to the alliance wall), and the usual plus/minus functions, and interpolation methods. For more advanced functions such as `inFrameOfReferenceOf()` or `twist()`, teams are encourage to Read the github source.

4.10 Twist2d

Coming soon, i'm confused.

Twist2d holds a dx, dy and dtheta component to represent a robot "twist." More docs coming soon.

4.11 Pose2dWithCurvature

Pose2dWithCurvature, similar to Twist2d, holds `Pose2d` and curvature components. Curvature is defined as one over the radius of a circle, and curvature can be positive or negative depending on the direction that the pose twists - left or right.

4.12 Other Units

Other SI Units of FalconLibrary not men sioned here include Mass, Ohms, Volts and Amps.

CHAPTER 5

FalconLibrary's Command-based Implememntation

FalconLibrary implements a Command Based framework, similar to WPILib. This command based implementation is based upon wrapping WPILib's Command based Commands and Subsystems, but is scheduled based upon Kotlin Coroutines and includes syntactic sugar for command group building. This example from Team 5190's 2018 offseason code demonstrates building CommandGroups with both parallel and sequential commands.

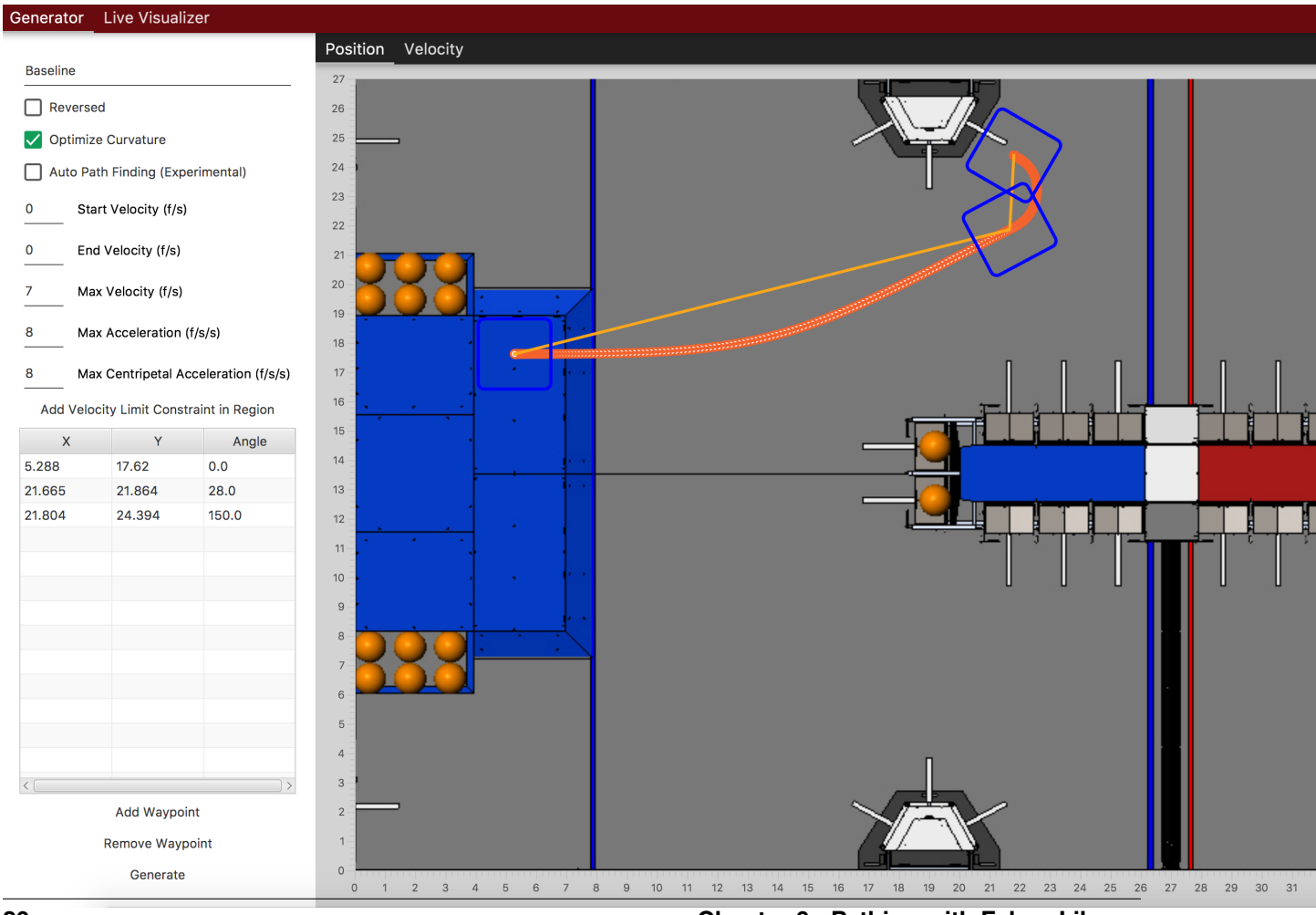
```
// Place third cube in scale
+parallel { // run all these commands in parallel
    +DriveSubsystem.followTrajectory(cube2ToScale, shouldMirrorPath)
        .withExit(stopScalePathCondition)
    +sequential { // run first the DelayCommand, then move the arm back
        +DelayCommand(cube2ToScale.lastState.t - 2.7.second)
        +SubsystemPreset.BEHIND.command
    }
    +sequential { // wait for the arm, then outtake a cube.
        +ConditionCommand { ArmSubsystem.armPosition > Constants.kArmBehindPosition -␣
↪Constants.kArmAutoTolerance }
        +IntakeCommand(IntakeSubsystem.Direction.OUT, 0.4).withTimeout(500.
↪millisecond)
    }
}
```

Note: To use FalconLibrary's built-in tank-drive drivetrains a team will have to be completely Falcon-command-based, and teams using WPI's command-based cannot use any FalconCommands or FalconSubsystems.

CHAPTER 6

Pathing with FalconLibrary

6.1 Summery



To get from point A to B in the fastest way possible, FalconLibrary generates a spline between the two points. (iirc) Team 254 teh Chezy Pofs were the first to use splines, back in 2014. FalconLibrary generates splines using user provided waypoints, initial and ending velocities, maximum speeds and accelerations, and additional constraints including velocity limiting regions and drivetrain models (see the article on these). These trajectories are then uploaded to the robot, which follows them using a selection of available followers. To do this, though, the robot needs to know where it is on the field and update it in real-time using a technique called Odometry, meaning that your robot needs encoders and a form of gyroscope such as NavX or Pigeon IMU.

6.2 Odometry

Odometry is the process of deriving robot position using Dead Reckoning. Using information about driven distance and heading, a robot can be "localized" on the field. The process of relocalization is deriving an absolute robot position by way of known robot pose or a vision target. FalconLibrary implements for users a build in Tank Drive Odometry class - all that users need to do is in give it Suppliers for drivetrain distances and robot heading. Review this for more information on Functional Interfaces in Java.

Java

```
/* Create a localization object because lambda expressions are fun */
localization = new TankEncoderLocalization(
        // the gyro needs to be positive counter-clockwise
        () -> Rotation2dKt.getDegree(getGyro(true)),
        // and these need to return a Length
        () -> getLeft().getDistance(),
        () -> getRight().getDistance());

/* set the robot pose to 0,0,0 */
localization.reset(new Pose2d());

// the update() method must be called periodically,
// as fast as possible. 100hz is ideal, but 20 will work.
Notifier localizationNotifier = new Notifier(
        () -> {localization.update();}
);
localizationNotifier.startPeriodic(1d / 100d);
```

Kotlin

```
// coming soon, coz i don't know Kotlin at all
```

6.3 Following paths

Paths should be generated with /docs/unnoficial-libraries/team5190/falcon-dash (TODO fix that link) Paths are stored as a `TimedTrajectory<Pose2dWithCurvature>`, which can be followed using:

- Feedforward, using no pose feedback
- **Pure Pursuit, which uses a lookahead point and angle to follow** a path. This should be phased out for tank drive in favor of
- RAMSETE, non-linear feedback based on robot pose.

It is recommended that teams make their drivetrains implement `DifferentialTrackerDriveBase` and convert their motors to `FalconMotor<Length>`, or it's subclasses, such as `FalconSRX<Length>`.

6.3.1 The DifferentialTrackerDriveBase

`DifferentialTrackerDriveBase` is an interface for teams to quickly make their drivetrains integrate with FalconLibrary path following. The interface requires you to have characterized and modeled your drivetrain, to have drive motors which subclass FalconMotor, and have implemented a form of localization. The method contains and inherits methods for utilizing the feedforward models of your drivetrain to estimate the voltage required for a (velocity, acceleration) command and setting motor output to a PID setpoint + feedforward voltage. See (TODO LINK) characterizing your drivetrain for information on the DifferentialDrive class. A bare-bones example of a DifferentialTrackerDriveBase can be found (TODO LINK) HERE.

6.3.2 An example path following command

Java

```
public class TrajectoryTrackerCommand extends SendableCommandBase {
        private TrajectoryTracker trajectoryTracker;
        private Supplier<TimedTrajectory<Pose2dWithCurvature>> trajectorySource;
        private DriveTrain driveBase;
        private boolean reset;
        private TrajectoryTrackerOutput output;
        Length mDesiredLeft;
        Length mDesiredRight;
        double mCurrentLeft;
        double mCurrentRight;

        Notifier mUpdateNotifier;

        public TrajectoryTrackerCommand(DriveTrain driveBase, Supplier<TimedTrajectory
↪<Pose2dWithCurvature>> trajectorySource) {
                this(driveBase, trajectorySource, false);
        }

        public TrajectoryTrackerCommand(DriveTrain driveBase, Supplier<TimedTrajectory
↪<Pose2dWithCurvature>> trajectorySource, boolean reset) {
                this(driveBase, Robot.drivetrain.getTrajectoryTracker(),␣
↪trajectorySource, reset);
        }

        public TrajectoryTrackerCommand(DriveTrain driveBase, TrajectoryTracker␣
↪trajectoryTracker, Supplier<TimedTrajectory<Pose2dWithCurvature>> trajectorySource,␣
↪boolean reset) {
                addRequirements(driveBase);
                this.driveBase = driveBase;
                this.trajectoryTracker = trajectoryTracker;
                this.trajectorySource = trajectorySource;
                this.reset = reset;
        }

        @Override
        public void initialize() {
                LiveDashboard.INSTANCE.setFollowingPath(false);

                if (trajectorySource == null) {
                        Logger.log("Sadly the trajectories are not generated. the␣
↪person responsible for the trajectories has been sacked.");
                        Trajectories.generateAllTrajectories();
```

(continues on next page)

(continued from previous page)

```
                }

                trajectoryTracker.reset(this.trajectorySource.get());

                if (reset == true) {
                        Robot.drivetrain.getLocalization().reset(trajectorySource.
↪get().getFirstState().getState().getPose());
                }

                LiveDashboard.INSTANCE.setFollowingPath(true);

                mUpdateNotifier = new Notifier(() -> {
                        output = trajectoryTracker.nextState(driveBase.
↪getRobotPosition(), TimeUnitsKt.getSecond(Timer.getFPGATimestamp()));

                        TrajectorySamplePoint<TimedEntry<Pose2dWithCurvature>>␣
↪referencePoint = trajectoryTracker.getReferencePoint();
                        if (referencePoint != null) {
                                Pose2d referencePose = referencePoint.getState().
↪getState().getPose();

                                LiveDashboard.INSTANCE.setPathX(referencePose.
↪getTranslation().getX().getFeet());
                                LiveDashboard.INSTANCE.setPathY(referencePose.
↪getTranslation().getY().getFeet());
                                LiveDashboard.INSTANCE.setPathHeading(referencePose.
↪getRotation().getRadian());

                        }

                        driveBase.setOutput(output);

                });
                mUpdateNotifier.startPeriodic(0.01);
        }

        @Override
        public void end(boolean interrupted) {
                mUpdateNotifier.stop();
                driveBase.stop();
                LiveDashboard.INSTANCE.setFollowingPath(false);
        }

        @Override
        public boolean isFinished() {
                return trajectoryTracker.isFinished();
        }

        public TimedTrajectory<Pose2dWithCurvature> getTrajectory() {
                return this.trajectorySource.get();
        }

}
```

Kotlin

```
// coming soon, coz i don't know Kotlin at all
```

CHAPTER 7

Example path following drivebases

Some example drivebases to get you started on path following quickly. For path following Commands, see :/falconlib-pathing

7.1 An example TalonSRX drivetrain with a NavX AHRS gyro

Java

```
public class DriveTrain extends SendableSubsystemBase /* or Subsystem */ implements␣
↪DifferentialTrackerDriveBase {

    public static final double kRobotMass = 50 /* Robot, kg */ + 5f /* Battery, kg */␣
↪+ 2f /* Bumpers, kg */;
    public static final double kRobotMomentOfInertia = 10.0; // kg m^2 // TODO Tune
    public static final double kRobotAngularDrag = 12.0; // N*m / (rad/sec)

    public static final double kWheelRadius = Util.toMeters(2f / 12f);// meters. TODO␣
↪tune
    public static final double kTrackWidth = Util.toMeters(26f / 12f);// meters

    private static final double kVDriveLeftLow = 0.274 * 1d; // Volts per radians per␣
↪second - Calculated emperically
    private static final double kADriveLeftLow = 0.032 * 1d; // Volts per radians per␣
↪second per second TODO tune
    private static final double kVInterceptLeftLow = 1.05 * 1d; // Volts - tuned!

    private static final double kVDriveRightLow = 0.265 * 1d; // Volts per radians␣
↪per second - Calculated emperically
    private static final double kADriveRightLow = 0.031 * 1d; // Volts per radians␣
↪per second per second TODO tune
    private static final double kVInterceptRightLow = 1.02 * 1d; // Volts - tuned!

    public static final DCMotorTransmission kLeftTransmissionModelLowGear = new␣
↪DCMotorTransmission(1 / kVDriveLeftLow,
```

(continues on next page)

(continued from previous page)

```
            kWheelRadius * kWheelRadius * kRobotMass / (2.0 * kADriveLeftLow),
            kVInterceptLeftLow);

    public static final DCMotorTransmission kRightTransmissionModelLowGear = new␣
↪DCMotorTransmission(1 / kVDriveRightLow,
            kWheelRadius * kWheelRadius * kRobotMass / (2.0 * kADriveRightLow),
            kVInterceptRightLow);

    private FalconSRX<Length> left, right;

    private Localization localization;

    private RamseteTracker ramseteTracker;

    private AHRS gyro; // a NavX

    private Notifier localizationNotifier;

    /* Ramsete constants */
    public static final double kDriveBeta = 2 * 1d; // Inverse meters squared
    public static final double kDriveZeta = 0.7 * 1d; // Unitless dampening co-
↪efficient

    public mlem() {
        var nativeUnitModel = new NativeUnitLengthModel(NativeUnitKt.
↪getNativeUnits(4096), LengthKt.getInch(2));
        left = new FalconSRX<Length>(0, nativeUnitModel, TimeUnitsKt.
↪getMillisecond(10));
        right = new FalconSRX<Length>(0, nativeUnitModel, TimeUnitsKt.
↪getMillisecond(10));

        gyro = new AHRS(Port.kMXP);

        /* Create a localization object because lamda expressions are fun */
        localization = new TankEncoderLocalization(() -> Rotation2dKt.
↪getDegree(getGyro(true)),
                () -> getLeftMotor().getSensorPosition(), () -> getRightMotor().
↪getSensorPosition());
        /* set the robot pose to 0,0,0 */
        localization.reset(new Pose2d());

        ramseteTracker = new RamseteTracker(kDriveBeta, kDriveZeta);

        localizationNotifier = new Notifier(() -> {
            this.getLocalization().update();
        });
        localizationNotifier.startPeriodic(1d / 100d);

    }

    private Localization getLocalization() {
        return localization;
    }

    private double getGyro(boolean isReversed) {
        return gyro.getFusedHeading() * ((isReversed) ? -1 : 1);
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    public FalconSRX<Length> getLeftMotor() {
        return left;
    }

    @Override
    public FalconSRX<Length> getRightMotor() {
        return right;
    }

    @Override
    public Pose2d getRobotPosition() {
        return null;
    }

    @Override
    public TrajectoryTracker getTrajectoryTracker() {
        return null;
    }

    @Override
    public DifferentialDrive getDifferentialDrive() {
        return null;
    }

}
```

Kotlin

```
// coming soon, coz i don't know Kotlin at all
```