Code Listing

Typeset using LaTeX.

Tele-Op:

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.OpticalDistanceSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.util.Range;

@TeleOp(name = "FinalTeleOp", group = "Competition")
//@Disabled
public class FinalTeleOp extends LinearOpMode {

    private DcMotor rightBack, leftBack, rightFront, leftFront;
    private DcMotor intake, shooter1, shooter2;

    private Servo ballFeeder;

    private OpticalDistanceSensor ods, ods2;

    //final String NORMAL = "normal", STRAIGHT = "straight";
    private final double POWER_FACTOR = 1, POSITIVE_STEP = 0.2, NEGATIVE_STEP = 0.5;
    protected static final double SHOOTER2_OFFSET = 0.04; // 0.07
    private final double INTAKE_POWER = 0.9;
    private final double SHOOT = Util.SHOOT, LOAD = Util.LOAD;
    protected static final long MILLIS_PER_NANO = 1000000;

    //String driveMode = NORMAL;
    private long shooterStart = System.nanoTime(), shooterLoadTimer = shooterStart;
    private double targetPowerR = 1, targetPowerL = 1, currentR = 1, currentL = 1;
    private boolean shooterStatus = false; //, aHasBeenPressed = false;

    private static int intakeStatus = 0;
    private boolean intakeChanged = false;
    private long oldLoopTime;

    public void runOpMode() throws InterruptedException {
        Util.colorSensors = false; Util.otherSensors = true; Util.servos = true;
        Util.init(this);
        ShooterPID.init();

        this.rightBack = Util.rightBack;
        this.leftBack = Util.leftBack;
        this.rightFront = Util.rightFront;
        this.leftFront = Util.leftFront;

        this.shooter1 = Util.shooter1;
        this.shooter2 = Util.shooter2;

        DcMotor[] temp = new DcMotor[6];
        temp[0] = this.rightBack; temp[1] = this.leftBack;
        temp[2] = this.rightFront; temp[3] = this.leftFront;
```

```
        temp[4] = this.shooter1; temp[5] = this.shooter2;

        Util.resetEncoders(this, temp);

        this.intake = Util.intake;

        this.ballFeeder = Util.ballFeeder;

        this.ods = Util.ods;
        this.ods2 = Util.ods2;

        ShooterPID.realRPMtarget = 1100;
        ShooterPID.calcuateTicsTarget(1100);

        Util.upDown.setPosition(Util.BEACON_DOWN);

        waitForStart();

        //long start = System.nanoTime();
        oldLoopTime = System.nanoTime();

        while (opModeIsActive()) {
        //for (int i = 0; i < 1000; i++) {

            handleDriveMotors();

            handleIntake();

            handleShooter();


            //Util.telemetry("shooter power", shooter1.getPower(), true);

            Thread.sleep(10);
        }

        /*/
        long end = System.nanoTime();

        Util.telemetry("average loop time (ms)", (end - start) / 1000000000, true);

        while (opModeIsActive()) Thread.sleep(20);
        /**/
    }

    private void handleDriveMotors() throws InterruptedException {
        if (gamepad1.dpad_up || gamepad1.dpad_down || gamepad1.dpad_left || gamepad1.
            ↪ dpad_right) {
            DpadDrive();
        } else if (Math.abs(gamepad1.right_stick_y) > JOYSTICK_DEADZONE_LIMIT ||
                   Math.abs(gamepad1.left_stick_y) > JOYSTICK_DEADZONE_LIMIT) {
            joystickDrive();
        } else if (gamepad1.a || gamepad1.y) {
            pressBeacon();
        } else {
            Util.setAllPowers(0);
            currentR = 1; currentL = 1;
```

```
    }

    telemetry.update();*/
}

final double DpadPower = 0.2;
final int DpadTime = 100;

private void DpadDrive() throws InterruptedException {
    if (gamepad1.dpad_up) {
        if (!gamepad1.dpad_right) Util.setLeftPowers(DpadPower);
        if (!gamepad1.dpad_left) Util.setRightPowers(DpadPower);
    } else if (gamepad1.dpad_down) {
        if (!gamepad1.dpad_right) Util.setLeftPowers(-DpadPower);
        if (!gamepad1.dpad_left) Util.setRightPowers(-DpadPower);
    } else if (gamepad1.dpad_right) {
        Util.setRightPowers(-DpadPower);
        Util.setLeftPowers(DpadPower);
    } else if (gamepad1.dpad_left) {
        Util.setRightPowers(DpadPower);
        Util.setLeftPowers(-DpadPower);
    }

    Thread.sleep(DpadTime);
    Util.setAllPowers(0);

    while (gamepad1.dpad_up || gamepad1.dpad_down || gamepad1.dpad_left || gamepad1.
        ↪ dpad_right) Thread.sleep(10);
}

private void joystickDrive() {
    double r = Util.getGamepadRightJoystickY(gamepad1);
    double l = Util.getGamepadLeftJoystickY(gamepad1);

    r = scaleDriveJoystick(r);
    l = scaleDriveJoystick(l);

    targetPowerR = r + 1;
    targetPowerL = l + 1;
        /*if (driveMode.equals(NORMAL)) {
            targetPowerL = l + 1;
        } else {
            targetPowerL = r + 1;
        }*/

    if (currentR < (targetPowerR - POSITIVE_STEP)) {
        currentR += POSITIVE_STEP;
    } else if (currentR < targetPowerR) {
        currentR = targetPowerR;
    }

    if (currentR > (targetPowerR + NEGATIVE_STEP)) {
        currentR -= NEGATIVE_STEP;
    } else if (currentR > targetPowerR) {
        currentR = targetPowerR;
    }
```

```
    if (currentL < (targetPowerL - POSITIVE_STEP)) {
        currentL += POSITIVE_STEP;
    } else if (currentL < targetPowerL) {
        currentL = targetPowerL;
    }

    if (currentL > (targetPowerL + NEGATIVE_STEP)) {
        currentL -= NEGATIVE_STEP;
    } else if (currentL > targetPowerL) {
        currentL = targetPowerL;
    }

    rightBack.setPower((currentR - 1) * POWER_FACTOR);
    leftBack.setPower((currentL - 1) * POWER_FACTOR);
    rightFront.setPower((currentR - 1) * POWER_FACTOR);
    leftFront.setPower((currentL - 1) * POWER_FACTOR);
}

private final double JOYSTICK_DEADZONE_LIMIT = 0.1;
private final double MIN_POWER = 0.1;
private final double B = 13.2699, A = 0.0684;

private double scaleDriveJoystick(double joystickValue) {
    // if the joystick is in the deadzone I defined, return 0
    if (Math.abs(joystickValue) < JOYSTICK_DEADZONE_LIMIT) return 0.0;
    // use the formula A*B^(joystickValue)
    double power = Math.signum(joystickValue) * A * Math.pow(B, Math.abs(joystickValue));
    if (Math.abs(power) < MIN_POWER) return 0.0;
    return Range.clip(power, -1.0, 1.0);
}

private void pressBeacon() throws InterruptedException {
    Util.upDown.setPosition(Util.BEACON_DOWN);
    if (gamepad1.y) {
        if (beaconForward() == -1) return;
    }
    else if (gamepad1.a && !gamepad1.start) {
        if (beaconBackward() == -1) return;
    }
}

private int beaconForward() throws InterruptedException {
    Util.setRightPowers(0.26);
    Util.setLeftPowers(0.28);
    //if (lookForLineAndCheckJoystick(0.50) == -1) return -1;
    if (lookForLineAndCheckJoystick(0.5) == -1) return -1;
    if (sleepAndCheckJoystick(250) == -1) return -1;
    Util.setAllPowers(0);
    if (sleepAndCheckJoystick(50) == -1) return -1;
    Util.setRightPowers(-0.16);
    Util.setLeftPowers(-0.14);
    if (sleepAndCheckJoystick(1000) == -1) return -1;
    return 0;
}

private int beaconBackward() throws InterruptedException {
    Util.setRightPowers(-0.28);
```

```
    Util.setLeftPowers(-0.26);
    //if (lookForLineAndCheckJoystick(0.50) == -1) return -1;
    if (lookForLineAndCheckJoystick(0.5) == -1) return -1;
    if (sleepAndCheckJoystick(50) == -1) return -1;
    Util.setAllPowers(0);
    if (sleepAndCheckJoystick(50) == -1) return -1;
    Util.setRightPowers(0.16);
    Util.setLeftPowers(0.14);
    if (sleepAndCheckJoystick(1000) == -1) return -1;
    return 0;
}

private int lookForLineAndCheckJoystick(double lightThreshold) throws
    ↪ InterruptedException {
    while ((ods.getLightDetected() < lightThreshold) && (ods2.getLightDetected() <
        ↪ lightThreshold)) {
        if (Math.abs(gamepad1.right_stick_y) > JOYSTICK_DEADZONE_LIMIT || Math.abs(
            ↪ gamepad1.left_stick_y) > JOYSTICK_DEADZONE_LIMIT) return -1;
        Thread.sleep(20);
    }
    return 0;
}

// Might not need this method in the end
private int sleepAndCheckJoystick(int sleepTimeMillis) throws InterruptedException {
    long startTime = System.nanoTime() / MILLIS_PER_NANO;
    while (((System.nanoTime() / MILLIS_PER_NANO) - startTime) < sleepTimeMillis) {
        if (Math.abs(gamepad1.right_stick_y) > JOYSTICK_DEADZONE_LIMIT || Math.abs(
            ↪ gamepad1.left_stick_y) > JOYSTICK_DEADZONE_LIMIT) return -1;
        Thread.sleep(20);
    }
    return 0;
}

// intake variables
private static final int INTAKE_OFF = 0, INTAKE = 1, OUTTAKE = 2;

private void handleIntake() {
    /*if ((gamepad1.right_bumper && gamepad1.left_bumper) && !intakeChanged) {
        /* if the intake is off, outtake
         * if the intake is intaking, outtake
         * if the intake is outtaking, do nothing
         *//*
        switch (intakeStatus) {
            case INTAKE_OFF:
            case INTAKE: outtake(); break;
            case OUTTAKE: break;
        }
        intakeChanged = true;
    }*/
    if (gamepad1.left_bumper && !intakeChanged) {
        /* if the intake is off, do nothing
         * if the intake is intaking, turn it off
         * if the intake is outtaking, turn it off
         */
        switch (intakeStatus) {
            case INTAKE_OFF: outtake(); break;
```

```
            case INTAKE:
            case OUTTAKE: intakeOff(); break;
        }
        intakeChanged = true;
    }
    if (gamepad1.right_bumper && !intakeChanged) {
        /* if the intake is off, intake
         * if the intake is intaking, do nothing
         * if the intake is outtaking, intake
         */
        switch (intakeStatus) {
            case INTAKE_OFF: intake(); break;
            case INTAKE:
            case OUTTAKE: intakeOff(); break;
            //case OUTTAKE: break;
        }
        intakeChanged = true;
    }
    // wait until the user releases all intake-related buttons before allowing the user
        ↪ to change the intake again
    else if (!gamepad1.right_bumper && !gamepad1.left_bumper) {
        intakeChanged = false;
        /*if (intakeStatus == OUTTAKE) {
            intakeOff();
        }*/
    }
}

// the three following methods standardize intaking, outtaking, and neither
private void intake() {
    this.intake.setPower(INTAKE_POWER);
    intakeStatus = INTAKE;
}

private void outtake() {
    this.intake.setPower(-INTAKE_POWER);
    intakeStatus = OUTTAKE;
}

private void intakeOff() {
    this.intake.setPower(0);
    intakeStatus = INTAKE_OFF;
}

private boolean SHOOTER_ON = true, SHOOTER_OFF = false;
private int shooterPID = 1200, shooterSpinUp = 1050 /* 1500 */, shooterLoad = 2000,
    ↪ shooterFire = 300;
private double shooter1Power = 0, shooter2Power = 0;

private void handleShooter() throws InterruptedException {
    long time = System.nanoTime() / MILLIS_PER_NANO;

    if (!shooterStatus && gamepad1.right_trigger >= 0.5) {
        double power = calculateShooterPower();
        shooter1Power = power;
        shooter2Power = power + SHOOTER2_OFFSET;
        shooter1.setPower(shooter1Power);
```

```
            shooter2.setPower(shooter2Power);
            shooterStart = time;
            shooterStatus = SHOOTER_ON;
        }
        if (gamepad1.left_trigger >= 0.5) {
            shooter1.setPower(0);
            shooter2.setPower(0);
            shooter1Power = 0;
            shooter2Power = 0;
            shooterStatus = SHOOTER_OFF;
        }

        if (shooterStatus) {
            ShooterPID.manageEncoderData(time - oldLoopTime);

            if ((time - shooterStart) > shooterPID) {
                //Util.telemetry("elapsedTime", time - oldLoopTime, false);
                double[] powers = ShooterPID.PID_calculateShooterPower(shooter1Power,
                    ↪ shooter2Power);
                shooter1Power = powers[0];
                shooter2Power = powers[1];
                shooter1.setPower(shooter1Power);
                shooter2.setPower(shooter2Power);
                /*Util.telemetry("power1", powers[0], false);
                Util.telemetry("power2", powers[1], true);*/
            }

            if (gamepad1.b && (time - shooterStart) > shooterSpinUp) { // && (time -
                ↪ shooterLoadTimer) > shooterLoad) {
                ballFeeder.setPosition(this.SHOOT);
                Thread.sleep(shooterFire);
                shooter1.setPower(0.5);
                shooter2.setPower(0.5);
                Thread.sleep(50);
                ballFeeder.setPosition(this.LOAD);
                shooter1.setPower(shooter1Power);
                shooter2.setPower(shooter2Power);
                shooterStart = time;
                //shooterLoadTimer = System.nanoTime();
            }
        } else {
            ShooterPID.clearQueue();
        }

        oldLoopTime = time;
    }

    protected static double calculateShooterPower() {
        double voltage = Util.getBatteryVoltage();
        // if the collector is running, the battery appears lower
        if (intakeStatus == INTAKE) voltage += 0.1;
        if (voltage >= 13.6) return -0.033*voltage + 0.71; //0.696
        else return -0.04*voltage + 0.80; //0.784
    }
}
```

Standard Red Autonomous (Standard Blue Autonomous is very similar):

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DeviceInterfaceModule;
import com.qualcomm.robotcore.hardware.GyroSensor;
import com.qualcomm.robotcore.hardware.OpticalDistanceSensor;
import com.qualcomm.robotcore.hardware.Servo;

@Autonomous(name="RedStandardCorner", group="Competition")
//@Disabled
public class RedStandardCorner extends LinearOpMode {

    // motors
    DcMotor rightBack, leftBack, rightFront, leftFront;
    DcMotor shooter1, shooter2;
    DcMotor[] driveMotors, shooterMotors;

    // servos
    Servo ballFeeder, upDown;

    // sensors
    OpticalDistanceSensor ods;
    GyroSensor gyro;

    // autonomous constants
    final int BEACON_MOVE = 400;
    double offBeaconPower, onBeaconPower;

    // variables to hold motor powers,
    double shooter1Power, shooter2Power;

    public void runOpMode() throws InterruptedException {

        Util.colorSensors = true; Util.otherSensors = true; Util.servos = true;
        Util.init(this);

        // disable color sensors to speed up gyro (for now)
        I2C_ColorSensor.disable();

        // turn on red LED on Device Interface Module to indicate Red Auto (and make sure
            ↪ blue LED is off)
        DeviceInterfaceModule dim = hardwareMap.deviceInterfaceModule.get("Sensors");
        dim.setLED(0, false);
        dim.setLED(1, true);

        // drive motors
        this.rightBack = Util.rightBack; this.leftBack = Util.leftBack;
        this.rightFront = Util.rightFront; this.leftFront = Util.leftFront;

        driveMotors = new DcMotor[4]; driveMotors[0] = this.rightBack; driveMotors[1] = this.
            ↪ leftBack; driveMotors[2] = this.rightFront; driveMotors[3] = this.leftFront;

        // shooter motors
        this.shooter1 = Util.shooter1; this.shooter2 = Util.shooter2;
```

```
shooterMotors = new DcMotor[2]; shooterMotors[0] = this.shooter1; shooterMotors[1] =
    ↪ this.shooter2;

// servos
this.ballFeeder = Util.ballFeeder;
this.upDown = Util.upDown;

// other sensors
this.ods = Util.ods;
this.gyro = Util.gyro;
//I2C_ColorSensor.init(this);

// reset the encoders on the DC motors
Util.resetEncoders(this, driveMotors);
Util.resetEncoders(this, shooterMotors);

ShooterPID.init();

waitForStart();

AutoLoopTest.driveAndShoot(1600, 2);

offBeaconPower = AutoUtil.offBeaconPower; onBeaconPower = AutoUtil.onBeaconPower;

Util.setDriveModeBrake();

// turn toward the closer beacon and corner vortex
AutoUtil.rampEncoderTurnLeft(60, 0.4);

Thread.sleep(100);

// drive near to the closer beacon
AutoUtil.PID_Forward(2800, 0.3, false, gyro);
AutoUtil.PID_Forward(1000, 0.2, true, gyro);

Thread.sleep(100);

// turn toward far beacon
AutoUtil.rampEncoderTurnRight(50, 0.4);

Thread.sleep(100);

// move toward the wall
AutoUtil.PID_Forward(2250, 0.4, false, gyro);

// enable the color sensors 'cause we're about to use them
I2C_ColorSensor.enable();

// follow the wall...
AutoUtil.encoderSteerForward(1250, 0.3, false);

// ...find the white line...
if (AutoUtil.encoderSteerForwardLineSafe(0.5, 0.1, 2400, false) == -1) {
    //Util.telemetry("failsafe", "------FAILSAFE ENGAGED------", true);
    Util.setDriveModeFloat();
    Util.setAllPowers(0);
```

```
    while (opModeIsActive()) Thread.sleep(20);
}
//Util.telemetry("failsafe", "-----FAILSAFE DIDN'T ENGAGE-----", true);

// ...and center the robot on the beacon
AutoUtil.encoderSteerForward(150, 0.1, true);

/* based on which side is red, move to that side,
 * lower our button pusher,
 * and roll over the button
 */
boolean tryAgain = false;
int frontRed, backRed;
double frontRatio, backRatio;
final int TRUE = 1, FALSE = 0, UNKNOWN = -1;

do {
    tryAgain = !tryAgain;

    frontRatio = frontRedVal / frontBlueVal;
    backRatio = backRedVal / backBlueVal;

    if (frontRatio > 1.1) frontRed = TRUE;
    else if (frontRatio < 0.9) frontRed = FALSE;
    else frontRed = UNKNOWN;

    if (backRatio > 1.1) backRed = TRUE;
    else if (backRatio < 0.9) backRed = FALSE;
    else backRed = UNKNOWN;
} while (frontRed == UNKNOWN && backRed == UNKNOWN && tryAgain);


boolean pressFront = false, pressBack = false;
if ((frontRed == FALSE && backRed != FALSE) || (frontRed != TRUE && backRed == TRUE))
    ↪  pressBack = true;
else if ((frontRed != FALSE && backRed == FALSE) || (frontRed == TRUE && backRed !=
    ↪ TRUE)) pressFront = true;
else if (frontRed == FALSE && backRed == FALSE) {
    if (frontRatio > backRatio) pressFront = true;
    else if (backRatio > frontRatio) pressBack = true;
}

if (!pressFront && !pressBack) {
    AutoUtil.encoderSteerBackward(2800, 0.3, false);
    AutoUtil.beaconUp(upDown);
} else if (pressFront) {
    AutoUtil.encoderSteerForward(BEACON_MOVE, offBeaconPower, true);
    AutoUtil.beaconDown(upDown);
    AutoUtil.encoderSteerBackward(BEACON_MOVE, onBeaconPower, true);
    Thread.sleep(100);

    AutoUtil.encoderSteerForward(BEACON_MOVE / 2, onBeaconPower, true);
    Thread.sleep(100);
    AutoUtil.encoderSteerBackward(BEACON_MOVE / 2, onBeaconPower, true);
    Thread.sleep(100);

    AutoUtil.encoderSteerForward(BEACON_MOVE, offBeaconPower, true);
```

```
    AutoUtil.beaconUp(upDown);
    AutoUtil.encoderSteerBackward(2800 + BEACON_MOVE, 0.3, false);
} else if (pressBack) {
    AutoUtil.encoderSteerBackward(BEACON_MOVE, offBeaconPower, true);
    AutoUtil.beaconDown(upDown);
    AutoUtil.encoderSteerForward(BEACON_MOVE, onBeaconPower, true);
    Thread.sleep(100);

    AutoUtil.encoderSteerBackward(BEACON_MOVE / 2, onBeaconPower, true);
    Thread.sleep(100);
    AutoUtil.encoderSteerForward(BEACON_MOVE / 2, onBeaconPower, true);
    Thread.sleep(100);

    AutoUtil.encoderSteerBackward(BEACON_MOVE, offBeaconPower, false);
    AutoUtil.encoderSteerBackward(2800 - BEACON_MOVE, 0.3, false);
    AutoUtil.beaconUp(upDown);
} else {
    AutoUtil.encoderSteerBackward(2800, 0.3, false);
    AutoUtil.beaconUp(upDown);
}

// move to the closer beacon
if (AutoUtil.encoderSteerBackwardLineSafe(0.5, 0.1, 3700, false) == -1) {
    //Util.telemetry("failsafe", "------FAILSAFE ENGAGED------", true);
    Util.setDriveModeFloat();
    Util.setAllPowers(0);
    while (opModeIsActive()) Thread.sleep(20);
}
//Util.telemetry("failsafe", "-----FAILSAFE DIDN'T ENGAGE-----", true);

// center the robot on the beacon
AutoUtil.encoderSteerBackward(80, 0.1, true);

/* based on which side is red, move to that side,
 * lower our button pusher,
 * and roll over the button
 */
tryAgain = false;
frontRed = 0; backRed = 0;

do {
    tryAgain = !tryAgain;

    frontRatio = frontRedVal / frontBlueVal;
    backRatio = backRedVal / backBlueVal;

    if (frontRatio > 1.1) frontRed = TRUE;
    else if (frontRatio < 0.9) frontRed = FALSE;
    else frontRed = UNKNOWN;

    if (backRatio > 1.1) backRed = TRUE;
    else if (backRatio < 0.9) backRed = FALSE;
    else backRed = UNKNOWN;
} while (frontRed == UNKNOWN && backRed == UNKNOWN && tryAgain);


pressFront = false; pressBack = false;
```

```
if ((frontRed == FALSE && backRed != FALSE) || (frontRed != TRUE && backRed == TRUE))
    ↪  pressBack = true;
else if ((frontRed != FALSE && backRed == FALSE) || (frontRed == TRUE && backRed !=
    ↪ TRUE)) pressFront = true;
else if ((frontRed == FALSE) && backRed == FALSE) {
    if (frontRatio > backRatio) pressFront = true;
    else if (backRatio > frontRatio) pressBack = true;
}

if (!pressFront && !pressBack) {
    AutoUtil.encoderSteerBackward(3000, 0.3, false);
    AutoUtil.beaconUp(upDown);
} else if (pressFront) {
    AutoUtil.encoderSteerForward(BEACON_MOVE, offBeaconPower, true);
    AutoUtil.beaconDown(upDown);
    AutoUtil.encoderSteerBackward(BEACON_MOVE, onBeaconPower, true);
    Thread.sleep(100);

    AutoUtil.encoderSteerForward(BEACON_MOVE / 2, onBeaconPower, true);
    Thread.sleep(100);
    AutoUtil.encoderSteerBackward(BEACON_MOVE / 2, onBeaconPower, true);
    Thread.sleep(100);

    AutoUtil.encoderSteerForward(BEACON_MOVE, offBeaconPower, false);
    AutoUtil.beaconUp(upDown);
    AutoUtil.encoderSteerForward(BEACON_MOVE, 0.3, true);
} else if (pressBack) {
    AutoUtil.encoderSteerBackward(BEACON_MOVE, offBeaconPower, true);
    AutoUtil.beaconDown(upDown);
    AutoUtil.encoderSteerForward(BEACON_MOVE, onBeaconPower, true);
    Thread.sleep(100);

    AutoUtil.encoderSteerBackward(BEACON_MOVE / 2, onBeaconPower, true);
    Thread.sleep(100);
    AutoUtil.encoderSteerForward(BEACON_MOVE / 2, onBeaconPower, false);

    AutoUtil.encoderSteerBackward(BEACON_MOVE, offBeaconPower, true);
    AutoUtil.beaconUp(upDown);
    Thread.sleep(100);
    AutoUtil.encoderSteerForward(BEACON_MOVE * 3, 0.3, true);
    // move away from the corner vortex
    //AutoUtil.encoderForward(BEACON_MOVE * 4, onBeaconPower, false);
} else {
    AutoUtil.encoderSteerBackward(3000, 0.3, false);
    AutoUtil.beaconUp(upDown);
}

Util.setDriveModeBrake();

Thread.sleep(100);

AutoUtil.encoderSteerBackward(1200, 0.05, 1, false);

Util.setRightPowers(-0.6);
Util.setLeftPowers(-0.1);

Thread.sleep(1300);
```

```
        Util.setAllPowers(0);

        //Thread.sleep(500);
        /*Thread.sleep(100);

        //Util.setDriveModeBrake();

        AutoUtil.encoderTurnRight(85, 0.25);

        Thread.sleep(100);

        Util.setDriveModeFloat();

        AutoUtil.encoderForward(3700, 0.8, true);

        Thread.sleep(1000);*/

        Util.setDriveModeBrake();

        while(opModeIsActive()) Thread.sleep(100);
    }
}
```

Alternate Red Autonomous (Alternate Blue Autonomous is very similar):

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DeviceInterfaceModule;
import com.qualcomm.robotcore.hardware.GyroSensor;
import com.qualcomm.robotcore.hardware.OpticalDistanceSensor;
import com.qualcomm.robotcore.hardware.Servo;

@Autonomous(name="RedAlternateCorner", group = "Competition")
//@Disabled
public class RedAlternateCorner extends LinearOpMode {
    // motors
    DcMotor rightBack, leftBack, rightFront, leftFront;
    DcMotor shooter1, shooter2;
    DcMotor[] motors;

    // servos
    Servo ballFeeder, upDown;

    // sensors
    OpticalDistanceSensor ods;
    GyroSensor gyro;

    public void runOpMode() throws InterruptedException {
        Util.colorSensors = false;
        Util.otherSensors = true;
        Util.servos = true;
        Util.init(this);

        // turn on red LED on Device Interface Module to indicate Red Auto (and make sure
            ↪ blue LED is off)
        DeviceInterfaceModule dim = hardwareMap.deviceInterfaceModule.get("Sensors");
        dim.setLED(0, false);
        dim.setLED(1, true);

        // drive motors
        this.rightBack = Util.rightBack;
        this.leftBack = Util.leftBack;
        this.rightFront = Util.rightFront;
        this.leftFront = Util.leftFront;

        motors = new DcMotor[4];
        motors[0] = this.rightBack;
        motors[1] = this.leftBack;
        motors[2] = this.rightFront;
        motors[3] = this.leftFront;

        // shooter motors
        this.shooter1 = Util.shooter1;
        this.shooter2 = Util.shooter2;

        // servos
        this.ballFeeder = Util.ballFeeder;
```

```
        this.upDown = Util.upDown;

        // otherSensors
        //this.ods = Util.ods;
        this.gyro = Util.gyro;
        //I2C_ColorSensor.init(this);

        Util.resetEncoders(this, motors);

        ShooterPID.init();

        waitForStart();

        telemetry.update();

        Thread.sleep(15 * 1000); // 12

        AutoLoopTest.driveAndShoot(3200, 2);

        AutoUtil.rampEncoderTurnLeft(60, 0.4);

        Thread.sleep(200);

        AutoUtil.PID_Forward(5500, 0.3, true, gyro);

        Thread.sleep(500);

        Util.setAllPowers(0);

        while (opModeIsActive()) Thread.sleep(10);
    }
}
```

Autonomous State Machine:

```
package org.firstinspires.ftc.teamcode;

//@Autonomous(name = "AutoLoopTest", group ="Test")
//@Disabled

//public class AutoLoopTest extends LinearOpMode {
public class AutoLoopTest {

    // motors
    //DcMotor rightBack, leftBack, rightFront, leftFront;
    //static DcMotor shooter1, shooter2;
    //DcMotor[] driveMotors, shooterMotors;

    /*int driveDistance = 1900;
    int shotNumber = 2;*/

    static double shooter1Power, shooter2Power;

    //static GyroSensor gyro;

    static AutoStates state = AutoStates.SHOOTER_SPIN_UP;

    static boolean firstTime = true, PIDon = false;

    /*public void runOpMode() throws InterruptedException {

        Util.colorSensors = true; Util.otherSensors = true; Util.servos = true;
        Util.init(this);

        I2C_ColorSensor.disable();

        this.rightBack = Util.rightBack; this.leftBack = Util.leftBack;
        this.rightFront = Util.rightFront; this.leftFront = Util.leftFront;

        driveMotors = new DcMotor[4]; driveMotors[0] = this.rightBack; driveMotors[1] = this.
            ↪ leftBack; driveMotors[2] = this.rightFront; driveMotors[3] = this.leftFront;

        this.shooter1 = Util.shooter1; this.shooter2 = Util.shooter2;

        shooterMotors = new DcMotor[2]; shooterMotors[0] = this.shooter1; shooterMotors[1] =
            ↪ this.shooter2;

        this.gyro = Util.gyro;

        Util.resetEncoders(this, driveMotors);
        Util.resetEncoders(this, shooterMotors);

        ShooterPID.init();

        waitForStart();
    }*/

    public static void driveAndShoot(int driveDistance, int shotNumber) throws
        ↪ InterruptedException {

        state = AutoStates.SHOOTER_SPIN_UP;
```

```
shooter1Power = 0; shooter2Power = 0;

firstTime = true; PIDon = false;

AutoUtil.onBeaconPower *= (13.6 / Util.getBatteryVoltage());

ShooterPID.realRPMtarget = 1080;
ShooterPID.calcuateTicsTarget(1080);

long start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO, currentTime, oldTime =
    ↪ start - 10;

while (state != AutoStates.END) {
    currentTime = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
    switch(state) {
        case SHOOTER_SPIN_UP:
            if (firstTime) {

                start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
                shooter1Power = FinalTeleOp.calculateShooterPower();
                shooter2Power = shooter1Power + FinalTeleOp.SHOOTER2_OFFSET; //
                    ↪ shooter 2 is slower than shooter 1
                Util.shooter1.setPower(shooter1Power); // + 0.01);
                Util.shooter2.setPower(shooter2Power); // + 0.01);

                firstTime = false;
            }
            if ((currentTime - start) > 1200) {
                state = AutoStates.DRIVE_01;
                Util.setDriveModeFloat();
                firstTime = true;
            }
            break;
        case DRIVE_01:
            if (firstTime) {

                start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
                Util.setAllPowers(0.1);

                PIDon = true;
                firstTime = false;
            }
            if ((currentTime - start) > 30) {
                state = AutoStates.DRIVE_015;
                firstTime = true;
            }
            break;
        case DRIVE_015:
            if (firstTime) {

                start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
                Util.setAllPowers(0.15);
                firstTime = false;
            }
            if ((currentTime - start) > 75) {
                state = AutoStates.DRIVE_FULL;
```

```
            firstTime = true;
        }
        break;
    case DRIVE_FULL:
        double startPos = 0;
        if (firstTime) {

            start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
            AutoUtil.resetGyroHeading(Util.gyro);
            PID.resetDriveIntegral();
            startPos = Util.rightBack.getCurrentPosition();

            firstTime = false;
        }

        PID.PIsetMotors(Util.gyro, 0.2);

        if (Util.rightBack.getCurrentPosition() > (startPos + (driveDistance *
            ↪ 0.98))) {
            state = AutoStates.DRIVE_COAST;
            firstTime = true;
            Util.telemetry("drive_time", (start - currentTime));
        }
        break;
    case DRIVE_COAST:
        if (firstTime) {

            start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
            Util.setAllPowers(0);
            firstTime = false;
        }
        if ((currentTime - start) > 500) {
            state = AutoStates.SHOOT_1;
            firstTime = true;
        }
        break;
    case SHOOT_1:
        if (firstTime) {

            start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
            Util.ballFeeder.setPosition(Util.SHOOT);
            firstTime = false;
        }
        if ((currentTime - start) > 500) {
            if (shotNumber == 1) state = AutoStates.SHOOTER_SPIN_DOWN;
            else state = AutoStates.LOAD_2;
            firstTime = true;
        }
        break;
    case LOAD_2:
        if (firstTime) {

            start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
            Util.ballFeeder.setPosition(Util.LOAD);
            firstTime = false;
        }
        if ((currentTime - start) > 1500) {
```

```
                state = AutoStates.SHOOT_2;
                firstTime = true;
            }
            break;
        case SHOOT_2:
            if (firstTime) {

                start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
                Util.ballFeeder.setPosition(Util.SHOOT);
                firstTime = false;
            }
            if ((currentTime - start) > 500) {
                state = AutoStates.SHOOTER_SPIN_DOWN;
                firstTime = true;
            }
            break;
        case SHOOTER_SPIN_DOWN:
            if (firstTime) {

                Util.shooter1.setPower(0); Util.shooter2.setPower(0);

                PIDon = false;
                firstTime = false;
                state = AutoStates.END;
            }
            break;
        case END:
            break;
    }

    ShooterPID.manageEncoderData(currentTime - oldTime);
    oldTime = currentTime;

    if (PIDon) {
        double[] powers = ShooterPID.PID_calculateShooterPower(shooter1Power,
            ↪ shooter2Power);
        shooter1Power = powers[0];
        shooter2Power = powers[1];
        Util.shooter1.setPower(shooter1Power); // + 0.005);
        Util.shooter2.setPower(shooter2Power); // + 0.005);
    }

    Thread.sleep(5);

    // oversample
    Util.shooter1.getCurrentPosition();
    Util.shooter2.getCurrentPosition();

    Thread.sleep(5);
}

Util.setDriveModeBrake();

Util.ballFeeder.setPosition(Util.LOAD);

/*telemetry.update();
```

```
    while(opModeIsActive()) Thread.sleep(100);*/
}

public static void driveAndShootFasterTest(int driveDistance, int shotNumber) throws
    ↪ InterruptedException {

    state = AutoStates.SHOOTER_SPIN_UP;

    shooter1Power = 0; shooter2Power = 0;

    firstTime = true; PIDon = false;

    AutoUtil.onBeaconPower *= (13.6 / Util.getBatteryVoltage());

    ShooterPID.realRPMtarget = 1030;
    ShooterPID.calcuateTicsTarget(1030);

    long start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO, currentTime, oldTime =
        ↪ start - 10;

    while (state != AutoStates.END) {
        currentTime = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
        switch(state) {
            case SHOOTER_SPIN_UP:
                if (firstTime) {

                    start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
                    shooter1Power = FinalTeleOp.calculateShooterPower();
                    shooter2Power = shooter1Power + FinalTeleOp.SHOOTER2_OFFSET; //
                        ↪ shooter 2 is slower than shooter 1
                    Util.shooter1.setPower(shooter1Power); // + 0.01);
                    Util.shooter2.setPower(shooter2Power); // + 0.01);

                    firstTime = false;
                }
                if ((currentTime - start) > 1200) {
                    state = AutoStates.DRIVE_01;
                    Util.setDriveModeFloat();
                    firstTime = true;
                }
                break;
            case DRIVE_01:
                if (firstTime) {

                    start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
                    Util.setAllPowers(0.1);

                    PIDon = true;
                    firstTime = false;
                }
                if ((currentTime - start) > 30) {
                    state = AutoStates.DRIVE_015;
                    firstTime = true;
                }
                break;
            case DRIVE_015:
                if (firstTime) {
```

```
            start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
            Util.setAllPowers(0.15);
            firstTime = false;
        }
        if ((currentTime - start) > 75) {
            state = AutoStates.DRIVE_FULL;
            firstTime = true;
        }
        break;
    case DRIVE_FULL:
        double startPos = 0;
        if (firstTime) {

            start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
            AutoUtil.resetGyroHeading(Util.gyro);
            PID.resetDriveIntegral();
            startPos = Util.rightBack.getCurrentPosition();

            firstTime = false;
        }

        PID.PIsetMotors(Util.gyro, 0.2);

        if (Util.rightBack.getCurrentPosition() > (startPos + (driveDistance *
            ↪ 0.98))) {
            state = AutoStates.DRIVE_COAST;
            firstTime = true;
            Util.telemetry("drive_time", (start - currentTime));
        }
        break;
    case DRIVE_COAST:
        if (firstTime) {

            start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
            Util.setAllPowers(0);
            firstTime = false;
        }
        if ((currentTime - start) > 500) {
            state = AutoStates.SHOOT_1;
            firstTime = true;
        }
        break;
    case SHOOT_1:
        if (firstTime) {

            start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
            Util.ballFeeder.setPosition(Util.SHOOT);
            firstTime = false;
        }
        if ((currentTime - start) > 350) {
            if (shotNumber == 1) state = AutoStates.SHOOTER_SPIN_DOWN;
            else state = AutoStates.LOAD_2;
            firstTime = true;
        }
        break;
    case LOAD_2:
```

```
            if (firstTime) {

                start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
                Util.ballFeeder.setPosition(Util.LOAD);
                firstTime = false;
            }
            if ((currentTime - start) > 600) {
                state = AutoStates.SHOOT_2;
                firstTime = true;
            }
            break;
        case SHOOT_2:
            if (firstTime) {

                start = System.nanoTime() / FinalTeleOp.MILLIS_PER_NANO;
                Util.ballFeeder.setPosition(Util.SHOOT);
                firstTime = false;
            }
            if ((currentTime - start) > 350) {
                state = AutoStates.SHOOTER_SPIN_DOWN;
                firstTime = true;
            }
            break;
        case SHOOTER_SPIN_DOWN:
            if (firstTime) {

                Util.shooter1.setPower(0); Util.shooter2.setPower(0);

                PIDon = false;
                firstTime = false;
                state = AutoStates.END;
            }
            break;
        case END:
            break;
    }

    ShooterPID.manageEncoderData(currentTime - oldTime);
    oldTime = currentTime;

    if (PIDon) {
        double[] powers = ShooterPID.PID_calculateShooterPower(shooter1Power,
            ↪ shooter2Power);
        shooter1Power = powers[0];
        shooter2Power = powers[1];
        Util.shooter1.setPower(shooter1Power); // + 0.005);
        Util.shooter2.setPower(shooter2Power); // + 0.005);
    }

    Thread.sleep(5);

    // oversample
    Util.shooter1.getCurrentPosition();
    Util.shooter2.getCurrentPosition();

    Thread.sleep(5);
}
```

```
        Util.setDriveModeBrake();

        Util.ballFeeder.setPosition(Util.LOAD);

        /*telemetry.update();

        while(opModeIsActive()) Thread.sleep(100);*/
    }
}
```

Util class:

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
import com.qualcomm.robotcore.hardware.Gamepad;
import com.qualcomm.robotcore.hardware.GyroSensor;
import com.qualcomm.robotcore.hardware.OpticalDistanceSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.HardwareMap;
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.util.RobotLog;

import org.firstinspires.ftc.robotcontroller.internal.FtcRobotControllerActivity;

public final class Util {

    protected static boolean init = false;
    //protected static boolean gyroEnabled = false;

    protected static DcMotor rightBack, leftBack, rightFront, leftFront;
    protected static DcMotor shooter1, shooter2, intake, led;

    protected static Servo ballFeeder, upDown;

    protected static OpticalDistanceSensor ods, ods2;

    protected static GyroSensor gyro;

    protected static boolean colorSensors = false, otherSensors = true, servos = true;

    protected final static double SEC_TO_NSEC = 1000000000, NEVEREST_37_TICS_PER_ROTATION =
        ↪ 103.6;
    protected final static double POWER_LIMIT = 1;
    protected static final double SHOOT = 0.5, LOAD = 1; // 0.95
    protected static final double BEACON_UP = 0.6, BEACON_DOWN = 0.9; // was 0.95

    //private static LinearOpMode linearOpMode;
    protected static LinearOpMode linearOpMode;
    private static DcMotor[] /*motors,*/ motorsWithEncoders;

    private Util() throws Exception {
        throw new Exception();
    }

    public static void init(LinearOpMode opMode) throws InterruptedException {
        linearOpMode = opMode;

        DcMotor[] temp;
        DcMotor[] tempWithEncoders;

        // drive motors
        rightBack = opMode.hardwareMap.dcMotor.get("rightBack"); rightBack.setDirection(
            ↪ DcMotor.Direction.REVERSE);
        leftBack = opMode.hardwareMap.dcMotor.get("leftBack");
        rightFront = opMode.hardwareMap.dcMotor.get("rightFront"); rightFront.setDirection(
            ↪ DcMotor.Direction.REVERSE);
```

```
        leftFront = opMode.hardwareMap.dcMotor.get("leftFront");

        rightBack.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
        leftBack.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
        rightFront.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
        leftFront.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);

        AutoUtil.r = rightFront; AutoUtil.l = leftFront;


        temp = new DcMotor[4]; temp[0] = rightBack; temp[1] = leftBack; temp[2] = rightFront;
            ↪  temp[3] = leftFront;
        tempWithEncoders = temp;

        //motors = temp;
        motorsWithEncoders = tempWithEncoders;

        // shooter motors
        shooter1 = getMotor("shooter1");
        shooter2 = getMotor("shooter2"); shooter2.setDirection(DcMotorSimple.Direction.
            ↪ REVERSE);
        shooter1.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);
        shooter2.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);

        // intake motor
        intake = getMotor("intake");

        // shooter indicator
        led = getMotor("led");
        led.setPower(0);

        // servos
        if (servos) {
            ballFeeder = getServo("ballFeeder"); ballFeeder.setPosition(LOAD);
            upDown = getServo("upDown"); upDown.setPosition(BEACON_UP);
        }

        // color sensors
        if (colorSensors) I2C_ColorSensor.init(opMode);

        // other sensors
        if (otherSensors) {
            ods = opMode.hardwareMap.opticalDistanceSensor.get("ods");
            ods2 = opMode.hardwareMap.opticalDistanceSensor.get("ods2");
            gyro = opMode.hardwareMap.gyroSensor.get("gyro");
        }

        //resetEncoders();

        init = true;
    }

    public static DcMotor getMotor(HardwareMap map, String deviceName) {
        return map.dcMotor.get(deviceName);
    }

    public static DcMotor getMotor(String deviceName) {
```

```
    return linearOpMode.hardwareMap.dcMotor.get(deviceName);
}

public static Servo getServo(HardwareMap map, String deviceName) {
    return map.servo.get(deviceName);
}

public static Servo getServo(String deviceName) {
    return linearOpMode.hardwareMap.servo.get(deviceName);
}

public static void resetEncoders(LinearOpMode opMode, DcMotor[] motorList) throws
    ↪ InterruptedException {
    for (DcMotor motor : motorList) motor.setMode(DcMotor.RunMode.RESET_ENCODERS);
    Thread.sleep(200);
    //while (motorList[0].getMode() != DcMotorController.RunMode.RESET_ENCODERS);
    for (DcMotor motor : motorList) motor.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODERS);
    Thread.sleep(200);
}

public static void resetEncoders(LinearOpMode opMode) throws InterruptedException {
    resetEncoders(opMode, motorsWithEncoders);
}

public static void resetEncoders(DcMotor[] motorList) throws InterruptedException {
    resetEncoders(linearOpMode, motorList);
}

public static void resetEncoders() throws InterruptedException {
    resetEncoders(linearOpMode, motorsWithEncoders);
}

public static double getBatteryVoltage() {
    return linearOpMode.hardwareMap.voltageSensor.iterator().next().getVoltage();
}

public static double getGamepadRightJoystickY(Gamepad gamepad) {
    double joystick;
    joystick = gamepad.right_stick_y;
    if (joystick != 0) return -joystick;
    return joystick;
}

public static double getGamepadLeftJoystickY(Gamepad gamepad) {
    double joystick;
    joystick = gamepad.left_stick_y;
    if (joystick != 0) return -joystick;
    return joystick;
}

public static void setRightPowers(double p) {
    rightBack.setPower(p);
    rightFront.setPower(p);
}

public static void setLeftPowers(double p) {
    leftBack.setPower(p);
```

```
    leftFront.setPower(p);
}

public static void setFrontPowers(double p) {
    rightFront.setPower(p);
    leftFront.setPower(p);
}

public static void setBackPowers(double p) {
    rightBack.setPower(p);
    leftBack.setPower(p);
}

public static void setAllPowers(double p) {
    rightBack.setPower(p);
    leftBack.setPower(p);
    rightFront.setPower(p);
    leftFront.setPower(p);
}

public static void setDriveModeFloat() {
    rightBack.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);
    leftBack.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);
    rightFront.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);
    leftFront.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);
}

public static void setDriveModeBrake() {
    rightBack.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
    leftBack.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
    rightFront.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
    leftFront.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
}

public static void setMotorsPowers(DcMotor[] motors, double p) {
    for (DcMotor motor : motors) {
        motor.setPower(p);
    }
}

public static void log(String message) {
    if (!FtcRobotControllerActivity.LOG) return;
    RobotLog.i(message);
}

public static void telemetry(String key, String data) {
    Util.linearOpMode.telemetry.update();
}

public static void telemetry(String key, int data) {
    Util.linearOpMode.telemetry.update();
}

public static void telemetry(String key, double data) {
    Util.linearOpMode.telemetry.update();
}
```

```
    public static void telemetry(String key, String data, boolean update) {
        if (update) Util.linearOpMode.telemetry.update();
    }

    public static void telemetry(String key, int data, boolean update) {
        if (update) Util.linearOpMode.telemetry.update();
    }

    public static void telemetry(String key, double data, boolean update) {
        if (update) Util.linearOpMode.telemetry.update();
    }
}
```

AutoUtil class:

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.GyroSensor;
import com.qualcomm.robotcore.hardware.Servo;

public final class AutoUtil {

    protected static boolean init = false;
    protected static LinearOpMode linearOpMode;
    private static float stallEnabledTime;
    private static double powerFactor = Util.POWER_LIMIT;
    protected static DcMotor r, l;
    private static final double MOTOR_POWER_THRESHOLD = 0.8 * Util.POWER_LIMIT,
        ↪ TIME_THRESHOLD = 0.3 * Util.SEC_TO_NSEC;
    private static final double MIN_POWER = 0.25;
    public static double offBeaconPower = 0.15, onBeaconPower = 0.11;

    private AutoUtil() throws Exception {
        throw new Exception();
    }

    public static void init(LinearOpMode opmode, GyroSensor gyro) throws InterruptedException
        ↪  {
        Util.init(opmode);
        StallProtection.init();
        linearOpMode = Util.linearOpMode;
        resetEncoders();
        calibrateGyro(gyro);
        init = true;
    }

    public static void encoderForward(int dist, double power, boolean stop) throws
        ↪ InterruptedException {
        int pos = (r.getCurrentPosition() + l.getCurrentPosition()) / 2;

        Util.setAllPowers(power);

        while (((r.getCurrentPosition() + l.getCurrentPosition()) / 2) < (pos + dist)) Thread
            ↪ .sleep(20);

        if (stop) Util.setAllPowers(0);
    }

    public static void encoderBackward(int dist, double power, boolean stop) throws
        ↪ InterruptedException {
        int pos = (r.getCurrentPosition() + l.getCurrentPosition()) / 2;

        Util.setAllPowers(-power);

        while (((r.getCurrentPosition() + l.getCurrentPosition()) / 2) > (pos - dist)) Thread
            ↪ .sleep(20);

        if (stop) Util.setAllPowers(0);
    }
```

```
private static double FORWARD_STEER = 1.15, BACKWARD_STEER = 1.17;

public static void encoderSteerForward(int dist, double powerR, double powerL, boolean
    ↪ stop) throws InterruptedException {
    int pos = (r.getCurrentPosition() + l.getCurrentPosition()) / 2;

    Util.setRightPowers(powerR);
    Util.setLeftPowers(powerL);

    while (((r.getCurrentPosition() + l.getCurrentPosition()) / 2) < (pos + dist)) Thread
        ↪ .sleep(20);

    if (stop) Util.setAllPowers(0);
}

public static void encoderSteerForward(int dist, double power, boolean stop) throws
    ↪ InterruptedException {

    int pos = (r.getCurrentPosition() + l.getCurrentPosition()) / 2;

    Util.setRightPowers(power * FORWARD_STEER);
    Util.setLeftPowers(power / FORWARD_STEER);

    while (((r.getCurrentPosition() + l.getCurrentPosition()) / 2) < (pos + dist)) Thread
        ↪ .sleep(20);

    if (stop) Util.setAllPowers(0);
}

public static void encoderSteerBackward(int dist, double powerR, double powerL, boolean
    ↪ stop) throws InterruptedException {
    int pos = (r.getCurrentPosition() + l.getCurrentPosition()) / 2;

    Util.setRightPowers(-powerR);
    Util.setLeftPowers(-powerL);

    while (((r.getCurrentPosition() + l.getCurrentPosition()) / 2) > (pos - dist)) Thread
        ↪ .sleep(20);

    if (stop) Util.setAllPowers(0);
}

public static void encoderSteerBackward(int dist, double power, boolean stop) throws
    ↪ InterruptedException {
    int pos = (r.getCurrentPosition() + l.getCurrentPosition()) / 2;

    Util.setRightPowers(-power * BACKWARD_STEER);
    Util.setLeftPowers(-power / BACKWARD_STEER);

    while (((r.getCurrentPosition() + l.getCurrentPosition()) / 2) > (pos - dist)) Thread
        ↪ .sleep(20);

    if (stop) Util.setAllPowers(0);
}

public static void encoderSteerForwardLine(double threshold, double power, boolean stop)
```

```
    ↪ throws InterruptedException {
    Util.setRightPowers(power * FORWARD_STEER);
    Util.setLeftPowers(power / FORWARD_STEER);

    while (Util.ods.getLightDetected() < threshold) Thread.sleep(20);

    if (stop) Util.setAllPowers(0);
}

public static int encoderSteerForwardLineSafe(double threshold, double power, int maxDist
    ↪ , boolean stop) throws InterruptedException {
    int start = Util.rightFront.getCurrentPosition();

    Util.setRightPowers(power * FORWARD_STEER);
    Util.setLeftPowers(power / FORWARD_STEER);

    while ((Util.ods.getLightDetected() < threshold) && (Util.rightFront.
        ↪ getCurrentPosition() - start) < maxDist) Thread.sleep(20);

    if (stop) Util.setAllPowers(0);

    if ((Util.rightFront.getCurrentPosition() - start) > maxDist) return -1;
    return 0;
}

public static void encoderSteerBackwardLine(double threshold, double power, boolean stop)
    ↪  throws InterruptedException {
    Util.setRightPowers(-power * BACKWARD_STEER);
    Util.setLeftPowers(-power / BACKWARD_STEER);

    while (Util.ods.getLightDetected() < threshold) Thread.sleep(20);

    if (stop) Util.setAllPowers(0);
}

public static int encoderSteerBackwardLineSafe(double threshold, double power, int
    ↪ maxDist, boolean stop) throws InterruptedException {
    int start = Util.rightFront.getCurrentPosition();

    Util.setRightPowers(-power * FORWARD_STEER);
    Util.setLeftPowers(-power / FORWARD_STEER);

    while ((Util.ods2.getLightDetected() < threshold) && (start - Util.rightFront.
        ↪ getCurrentPosition()) < maxDist) Thread.sleep(20);

    if (stop) Util.setAllPowers(0);

    if (start - (Util.rightFront.getCurrentPosition()) > maxDist) return -1;
    return 0;
}

public static void PID_Forward(double distance, double power, boolean stop, GyroSensor
    ↪ gyro) throws InterruptedException {
    resetGyroHeading(gyro);
    PID.resetDriveIntegral();
    double start = Util.rightBack.getCurrentPosition();
    Util.setAllPowers(0.1);
```

```
        Thread.sleep(30);
        Util.setAllPowers(0.15);
        Thread.sleep(75);
        while (Util.rightBack.getCurrentPosition() < (start + (distance * 0.98))) {
            PID.PIsetMotors(gyro, powerFactor * power);
            Thread.sleep(10);
        }
        if (stop) Util.setAllPowers(0);
    }

    public static void PID_Backward(double distance, double power, boolean stop, GyroSensor
        ↪ gyro) throws InterruptedException {
        resetGyroHeading(gyro);
        PID.resetDriveIntegral();
        double start = Util.rightBack.getCurrentPosition();
        Util.setAllPowers(-0.1);
        Thread.sleep(30);
        while (Util.rightBack.getCurrentPosition() > (start - (distance * 0.98))) {
            PID.PIsetMotors(gyro, powerFactor * -power);
            Thread.sleep(10);
        }
        if (stop) Util.setAllPowers(0);
    }

    public static void encoderTurnRight(double degrees, double power) throws
        ↪ InterruptedException {
        Util.resetEncoders();

        double dist = degrees / 360;
        dist = dist * 15 / 4 * 1120;

        Util.setRightPowers(-power);
        Util.setLeftPowers(power);

        while (((Math.abs(r.getCurrentPosition()) + Math.abs(l.getCurrentPosition())) / 2) <
            ↪ dist) {
            /*Util.telemetry("rf", Util.rightFront.getCurrentPosition(), false);
            Util.telemetry("lf", Util.leftFront.getCurrentPosition(), false);
            Util.telemetry("rb", Util.rightBack.getCurrentPosition(), false);
            Util.telemetry("lb", Util.leftBack.getCurrentPosition(), true);*/
            Thread.sleep(20);
        }

        Util.setAllPowers(0);
    }

    public static void encoderTurnLeft(int degrees, double power) throws InterruptedException
        ↪  { encoderTurnRight(degrees, -power); }

    final static double RAMP_UP_DELTA = 0.02, RAMP_DOWN_DELTA = 0.025;
    final static int EXTRA_DEGREES = 3; // 1

    public static void rampEncoderTurnRight(double targetDegrees, double targetPower) throws
        ↪ InterruptedException {
        Util.resetEncoders();

        double dist = targetDegrees / 360;
```

```
        dist = dist * 15 / 4 * 1120;

        double power = MIN_POWER - RAMP_UP_DELTA;
        boolean reachedTargetPower = false;
        while (((Math.abs(r.getCurrentPosition()) + Math.abs(l.getCurrentPosition())) / 2) <
            ↪ (dist / 2)) {
            if (!reachedTargetPower) power += RAMP_UP_DELTA;
            if (power > targetPower) {
                power = targetPower;
                reachedTargetPower = true;
            }
            Util.setRightPowers(-power);
            Util.setLeftPowers(power);
            Thread.sleep(10);
        }
        power = targetPower;
        boolean reachedMinPower = false;
        while (((Math.abs(r.getCurrentPosition()) + Math.abs(l.getCurrentPosition())) / 2) <
            ↪ dist) {
            if (!reachedMinPower) power -= RAMP_DOWN_DELTA;
            if (power < MIN_POWER) {
                power = MIN_POWER;
                reachedMinPower = true;
            }
            Util.setRightPowers(-power);
            Util.setLeftPowers(power);
            Thread.sleep(10);
        }
        Util.setAllPowers(0);
    }

    public static void rampEncoderTurnLeft(double targetDegrees, double targetPower) throws
        ↪ InterruptedException {
        Util.resetEncoders();

        double dist = targetDegrees / 360;
        dist = dist * 15 / 4 * 1120;

        double power = MIN_POWER - RAMP_UP_DELTA;
        boolean reachedTargetPower = false;
        while (((Math.abs(r.getCurrentPosition()) + Math.abs(l.getCurrentPosition())) / 2) <
            ↪ (dist / 2)) {
            if (!reachedTargetPower) power += RAMP_UP_DELTA;
            if (power > targetPower) {
                power = targetPower;
                reachedTargetPower = true;
            }
            Util.setRightPowers(power);
            Util.setLeftPowers(-power);
            Thread.sleep(10);
        }
        power = targetPower;
        boolean reachedMinPower = false;
        while (((Math.abs(r.getCurrentPosition()) + Math.abs(l.getCurrentPosition())) / 2) <
            ↪ dist) {
            if (!reachedMinPower) power -= RAMP_DOWN_DELTA;
            if (power < MIN_POWER) {
```

```
            power = MIN_POWER;
            reachedMinPower = true;
        }
        Util.setRightPowers(power);
        Util.setLeftPowers(-power);
        Thread.sleep(10);
    }
    Util.setAllPowers(0);
}

public static void gyroTurnRight(double degreeTarget, double targetPower, GyroSensor gyro
    ↪ ) throws InterruptedException {
    resetGyroHeading(gyro);
    double power = MIN_POWER;
    boolean reachedMinPower = false;
    while (PID.heading(gyro) < (degreeTarget / 2)) {
        power += RAMP_UP_DELTA;
        if (power > targetPower) {
            Util.setRightPowers(-targetPower);
            Util.setLeftPowers(targetPower);
            break;
        }
        Util.setRightPowers(-power);
        Util.setLeftPowers(power);
        Thread.sleep(10);
    }
    power = targetPower;
    double rampUpDegrees = PID.heading(gyro);
    while (degreeTarget - PID.heading(gyro) > rampUpDegrees * 2) Thread.sleep(10);
    while (PID.heading(gyro) - degreeTarget > EXTRA_DEGREES) {
        power -= RAMP_DOWN_DELTA;
        if (power < MIN_POWER) {
            Util.setRightPowers(-MIN_POWER);
            Util.setLeftPowers(MIN_POWER);
        } else {
            Util.setRightPowers(-power);
            Util.setLeftPowers(power);
        }
        Thread.sleep(10);
    }
    Util.setAllPowers(0);
}

public static void gyroTurnLeft(double degreeTarget, double targetPower, GyroSensor gyro)
    ↪  throws InterruptedException {
    degreeTarget = -degreeTarget;
    resetGyroHeading(gyro);
    double power = MIN_POWER;
    while (PID.heading(gyro) > (degreeTarget / 2)) {
        power += RAMP_UP_DELTA;
        if (power > targetPower) {
            Util.setRightPowers(targetPower);
            Util.setLeftPowers(-targetPower);
            break;
        }
        Util.setRightPowers(power);
        Util.setLeftPowers(-power);
```

```
            Thread.sleep(10);
        }
        power = targetPower;
        double rampUpDegrees = PID.heading(gyro);
        while (degreeTarget - PID.heading(gyro) < rampUpDegrees) Thread.sleep(10);
        while (PID.heading(gyro) - degreeTarget > EXTRA_DEGREES) {
            power -= RAMP_DOWN_DELTA;
            if (power < MIN_POWER) {
                Util.setRightPowers(MIN_POWER);
                Util.setLeftPowers(-MIN_POWER);
            } else {
                Util.setRightPowers(power);
                Util.setLeftPowers(-power);
            }
            Thread.sleep(10);
        }
        Util.setAllPowers(0);
    }

    public static void resetEncoders(DcMotor[] motors) throws InterruptedException {
        Util.resetEncoders(motors);
    }

    public static void resetEncoders() throws InterruptedException {
        Util.resetEncoders();
    }

    public static void calibrateGyro(GyroSensor gyro) throws InterruptedException {
        gyro.calibrate();
        while (gyro.isCalibrating()) Thread.sleep(50);
    }

    public static void resetGyroHeading(GyroSensor gyro) {
        gyro.resetZAxisIntegrator();
    }

    public static void beaconUp(Servo servo) throws InterruptedException {
        servo.setPosition(Util.BEACON_UP);
        Thread.sleep(100);
    }

    public static void beaconDown(Servo servo) throws InterruptedException {
        servo.setPosition(Util.BEACON_DOWN);
        Thread.sleep(100);
    }

    public static void intake(double p) { Util.intake.setPower(Math.abs(p)); }

    public static void outtake(double p) {
        Util.intake.setPower(-Math.abs(p));
    }

    /*public static double gyroDrift(GyroSensor gyro) throws InterruptedException {
        resetGyro(gyro);
        Thread.sleep(15000);
        int heading = gyro.getHeading();
        double error = heading;
```

```
        if (heading > 180) error = 360 - heading;
        return error / 15000;
    }*/
}
```

PID class:

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.hardware.GyroSensor;
import com.qualcomm.robotcore.util.Range;
import com.qualcomm.robotcore.util.RobotLog;

public final class PID {

    // drive
    private static final float Kp = 0.01f;   //proportional constant      //TODO final tune
    private static final float Ki = 0.0f;    //integral constant     //TODO tune
    private static final int offset = 0;          //value that <gyroHeading> should be
    private static double integral = 0;      //variable to hold integral value (accumulated
        ↪ error)

    //testing
    private static boolean log = false;// FtcRobotControllerActivity.LOG;

    private PID() throws Exception { throw new Exception(); }

    public static double[] P(GyroSensor gyro, double Tp) {
        //TODO factor in battery power
        int heading = heading(gyro);
        int error = heading - offset;
        double turn = Kp * error;
        double[] toReturn = {Range.clip(Tp - turn, -1, 1), Range.clip(Tp + turn, -1, 1)};

        if (!log) {
            return toReturn;
        }

        //logging
        RobotLog.i("-----------P␣start-----------");
        RobotLog.i("Tp␣(power)␣" + Tp);
        RobotLog.i("gyro␣heading:␣" + gyro.getHeading());
        RobotLog.i("scaled␣heading:␣" + heading);
        RobotLog.i("error:␣" + error);
        RobotLog.i("turn:␣" + turn);
        RobotLog.i("right␣power:␣" + toReturn[0]);
        RobotLog.i("left␣power:␣" + toReturn[1]);
        RobotLog.i("-----------P␣end-----------");

        return toReturn;
    }

    public static double[] PI(GyroSensor gyro, double Tp) {
        //TODO factor in battery power
        int heading = heading(gyro);
        int error = heading - offset;
        integral += error;
        double turn = Kp * error + Ki * integral;
        double[] toReturn = {Range.clip(Tp + turn, -1, 1), Range.clip(Tp - turn, -1, 1)};

        if (!log) {
            return toReturn;
        }
```

```
        //logging
        RobotLog.i("-----------P_start-----------");
        RobotLog.i("Tp_(power)_" + Tp);
        RobotLog.i("gyro_heading:_" + gyro.getHeading());
        RobotLog.i("scaled_heading:_" + heading);
        RobotLog.i("error:_" + error);
        RobotLog.i("integral:_" + integral);
        RobotLog.i("turn:_" + turn);RobotLog.i("right_power:_" + toReturn[0]);
        RobotLog.i("left_power:_" + toReturn[1]);
        RobotLog.i("-----------P_end-----------");

        return toReturn;
    }

    public static void PsetMotors(GyroSensor gyro, double Tp) {
        double[] motors = P(gyro, Tp);
        Util.setRightPowers(motors[0]);
        Util.setLeftPowers(motors[1]);
    }

    public static void PIsetMotors(GyroSensor gyro, double Tp) {
        double motors[] = PI(gyro, Tp);
        Util.setRightPowers(motors[0]);
        Util.setLeftPowers(motors[1]);
    }

    public static int heading(GyroSensor gyro) {
        int heading = gyro.getHeading();
        if (heading > 180) return heading - 360;
        return heading;
        //-179 - 180
    }

    public static void resetDriveIntegral() {
        integral = 0;
    }
}
```

ShooterPID class:

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.util.Range;

import java.util.LinkedList;
import java.util.Queue;

public final class ShooterPID {

    private static final float shooterKp = 0.00005f; // Ku = 0.0001
    private static final float shooterKi = 0.0f;

    static final double MOVING_AVERAGE_LENGTH = 50, MEASURING_INTERVAL = 10;
    static final double RPM_TARGET = 1050.0; // 1050.0
    static final double TICS_PER_ROTATION = Util.NEVEREST_37_TICS_PER_ROTATION;
    static final double TICS_TARGET = TICS_PER_ROTATION * (RPM_TARGET / 60.0) * (
        ↪ MEASURING_INTERVAL / 1000.0); // tics per MEASURING_INTERVAL, is 46.25 if target
        ↪ is 1250

    private static double shooterIntegral1 = 0, shooterIntegral2 = 0;

    public static double realRPMtarget = RPM_TARGET;
    private static double realTicsTarget = TICS_PER_ROTATION * (realRPMtarget / 60.0) * (
        ↪ MEASURING_INTERVAL / 1000.0);

    private static Queue<Double> shooter1Queue, shooter2Queue;
    private static Queue<Long> elapsedTimeQueue;

    private static boolean queueClear, ledOn = false;

    private ShooterPID() throws Exception { throw new Exception(); }

    public static void init() {
        clearQueue();
        ledOn = false;
    }

    public static double[] PID_calculateShooterPower(double power1, double power2) {
        double delta1 = shooter1Sum / MOVING_AVERAGE_LENGTH;
        double delta2 = shooter2Sum / MOVING_AVERAGE_LENGTH;
        //long deltat = timeSum / MOVING_AVERAGE_LENGTH;

        //double ticsTarget = ((RPM_TARGET / 60.0) / elapsedTime) * TICS_PER_ROTATION;

        //                  tics per rotation * rotations per second * seconds
        /*double ticsTarget = TICS_PER_ROTATION * (RPM_TARGET / 60.0) * (deltat / 1000.0);
        Util.telemetry("ticsTarget", ticsTarget, false);*/
        /*Util.telemetry("delta1", delta1, false);
        Util.telemetry("delta2", delta2, false);
        Util.telemetry("shooter1Sum", shooter1Sum, false);*/

        return PI_Shooter(delta1, delta2, realTicsTarget, power1, power2);
    }

    public static double[] PI_Shooter(double tics1, double tics2, double tics_target, double
        ↪ power1, double power2) {
```

```
    double error1 = tics_target - tics1, error2 = tics_target - tics2;
    shooterIntegral1 += error1; shooterIntegral2 += error2;
    double adjust1 = shooterKp * error1 + shooterKi * shooterIntegral1;
    double adjust2 = shooterKp * error2 + shooterKi * shooterIntegral2;
    double[] toReturn = {Range.clip(power1 + adjust1, 0, 1), Range.clip(power2 + adjust2,
        ↪  0, 1)};
    return toReturn;
}

public static void resetShooterIntegrals() { shooterIntegral1 = 0; shooterIntegral2 = 0;
    ↪ }

private static double shooter1Diff, shooter2Diff;
private static double shooter1Sum, shooter2Sum;
private static int shooter1Pos, shooter2Pos;
private static int lastShooter1Pos = 0, lastShooter2Pos = 0;

private static long timeSum;

public static void manageEncoderData(double elapsedTime) {
    shooter1Pos = Util.shooter1.getCurrentPosition();
    shooter2Pos = Util.shooter2.getCurrentPosition();

    shooter1Diff = Math.abs(shooter1Pos - lastShooter1Pos);
    shooter2Diff = Math.abs(shooter2Pos - lastShooter2Pos);

    shooter1Diff = (MEASURING_INTERVAL / elapsedTime) * shooter1Diff;
    shooter2Diff = (MEASURING_INTERVAL / elapsedTime) * shooter2Diff;

    if (shooter1Diff > (realTicsTarget * 2)) shooter1Diff = realTicsTarget * 2;
    if (shooter2Diff > (realTicsTarget * 2)) shooter2Diff = realTicsTarget * 2;

    shooter1Sum = shooter1Sum + shooter1Diff - shooter1Queue.poll();
    shooter1Queue.add(shooter1Diff);
    shooter2Sum = shooter2Sum + shooter2Diff - shooter2Queue.poll();
    shooter2Queue.add(shooter2Diff);
    /*timeSum = timeSum + elapsedTime - elapsedTimeQueue.poll();
    elapsedTimeQueue.add(elapsedTime);*/

    lastShooter1Pos = shooter1Pos;
    lastShooter2Pos = shooter2Pos;


    if (!ledOn && (shooter1Sum / MOVING_AVERAGE_LENGTH) > (realTicsTarget)) {
        Util.led.setPower(1);
        ledOn = true;
    }
    else if (ledOn && (shooter1Sum / MOVING_AVERAGE_LENGTH ) < (realTicsTarget * 0.9)) {
        Util.led.setPower(0);
        ledOn = false;
    }
    /*else {
        Util.led.setPower(0);
        ledOn = false;
    }*/

    queueClear = false;
```

```
    }

    public static void clearQueue() {
        if (queueClear) return;

        shooter1Queue = new LinkedList<>();
        shooter2Queue = new LinkedList<>();
        //elapsedTimeQueue = new LinkedList<>();

        shooter1Sum = 0;
        shooter2Sum = 0;
        timeSum = 0;

        for (int i = 0; i < MOVING_AVERAGE_LENGTH; i++) {
            shooter1Queue.add(0.0);
            shooter2Queue.add(0.0);
            //elapsedTimeQueue.add((long)0);
        }

        Util.led.setPower(0);
        ledOn = false;

        queueClear = true;
    }

    public static void calcuateTicsTarget(double realRPM) {
        realTicsTarget = TICS_PER_ROTATION * (realRPM / 60.0) * (MEASURING_INTERVAL / 1000.0)
            ↪ ;
    }

    /*public static void fillQueue() throws InterruptedException {
        shooter1Sum = TICS_TARGET * MOVING_AVERAGE_LENGTH;
        shooter2Sum = TICS_TARGET * MOVING_AVERAGE_LENGTH;

        shooter1Queue.clear();
        shooter2Queue.clear();

        for (int i = 0; i < MOVING_AVERAGE_LENGTH; i++) {
            //shooter1Queue.poll();
            shooter1Queue.add(TICS_TARGET);
            //shooter2Queue.poll();
            shooter2Queue.add(TICS_TARGET);
            //elapsedTimeQueue.add((long)0);
        }

        lastShooter1Pos = Util.shooter1.getCurrentPosition();
        lastShooter2Pos = Util.shooter2.getCurrentPosition();

        Thread.sleep(10);
    }*/

    public static void printQueue() {
        Util.telemetry("q", shooter1Queue.toString(), true);
    }
}
```

Our robot configuration file:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<Robot type="FirstInspires-FTC">
    <ServoController name="Servos" serialNumber="AL00VST0">
        <Servo name="upDown" port="1" />
        <Servo name="ballFeeder" port="2" />
    </ServoController>
    <DeviceInterfaceModule name="Sensors" serialNumber="AI02RHDJ">
        <I2cDevice name="colorFront" port="2" />
        <Gyro name="gyro" port="3" />
        <I2cDevice name="colorBack" port="4" />
        <OpticalDistanceSensor name="ods" port="7" />
  <OpticalDistanceSensor name="ods2" port="6" />
    </DeviceInterfaceModule>
    <MotorController name="Back_Motors" serialNumber="A104OQ1P">
        <Motor name="rightBack" port="1" />
        <Motor name="leftBack" port="2" />
    </MotorController>
    <MotorController name="Front_Motors" serialNumber="AL00VWH0">
        <Motor name="rightFront" port="1" />
        <Motor name="leftFront" port="2" />
    </MotorController>
    <MotorController name="Shooter" serialNumber="AI049QOP">
        <Motor name="shooter1" port="1" />
        <Motor name="shooter2" port="2" />
    </MotorController>
    <MotorController name="Intake" serialNumber="AL00VBIF">
        <Motor name="intake" port="1" />
        <Motor name="led" port="2" />
    </MotorController>
</Robot>
```