Code Listing

Typeset using LaTeX.

Teleop:

```
package com.qualcomm.ftcrobotcontroller;

import android.app.Activity;
import android.graphics.Color;
import android.view.View;

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;
import com.qualcomm.robotcore.util.Range;

/* FTC Team 9899 TeleOp opmode
 * accepts input from two gamepads which control the...
 *  ...tank drive system
 *  ...arm
 *  ...doors and intake on the arm
 *  ...trigger activators
 * automatically...
 *  ...reduces and then, after 3 seconds of continuous stalling, zeroes all motor powers (
    ↪ stall protection)
 */

public class TeleOpLinear extends LinearOpMode {

    // motors and sensors
    private DcMotor rightBack, leftBack, rightFront, leftFront, arm, intake, hanger;
    private Servo leftDoor, rightDoor, rightTrigger, leftTrigger;
    private GyroSensor gyro;

    // robot status variables
    private int intakeStatus = 0;
    private boolean intakeChanged = false;
    private boolean robotTurning = false;
    private boolean robotGoingForward = false;
    private boolean robotGoingBackward = false;
    private double powerFactor = Util.POWER_LIMIT;
    private float motorRampUpTime;
    private float checkIntervalTime;
    private float hasBeenStalledTime;
    private boolean continuousStall = false;
    private int numStalls = 0;
    protected static boolean stallProtectionGloballyEnabled = false;

    // power constants
    private final double BACK_SCALE = Util.BACK_SCALE;
    private final double POWER_FLOAT = Util.POWER_FLOAT;

    // stall protection limits and thresholds
    private final double MOTOR_POWER_THRESHOLD = 0.9 * Util.POWER_LIMIT;
    private final double TIME_THRESHOLD = 0.3 * Util.SEC_TO_NSEC;
```

```
private final double GYRO_MOUNTAIN_THRESHOLD = -15;
private final double FORWARD_MIN_STALL_POWER = 0.35;
private final double BACKWARD_MIN_STALL_POWER = 0.25;
private final int HAS_BEEN_STALLED_LIMIT = 3;

//private final double GYRO_FORWARD_TIP_THRESHOLD = 5;

private View relativeLayout;

public TeleOpLinear() {
}

public void runOpMode() throws InterruptedException {
    // initialization
    Util.init(this);
    StallProtection.init();

    // get motors from shared class
    rightBack = Util.rightBack;
    leftBack = Util.leftBack;
    rightFront = Util.rightFront;
    leftFront = Util.leftFront;
    arm = Util.arm;
    intake = Util.intake;
    hanger = Util.hanger;

    // get servos from shared class
    leftDoor = Util.leftDoor;
    rightDoor = Util.rightDoor;
    rightTrigger = Util.rightTrigger;
    leftTrigger = Util.leftTrigger;

    // get sensor(s) from shared class
    gyro = Util.gyro;

    relativeLayout = ((Activity) hardwareMap.appContext).findViewById(R.id.RelativeLayout
        ↪ );

    waitForStart();

    motorRampUpTime = System.nanoTime();

    while (opModeIsActive()) {

        // drive motors, arm, and hang mechanism
        double r = Util.getGamepadRightJoystickY(gamepad1);
        double l = Util.getGamepadLeftJoystickY(gamepad1);
        double a = -Util.getGamepadRightJoystickY(gamepad2);
        double h = (Util.getGamepadLeftJoystickY(gamepad2) + (gamepad1.right_trigger -
            ↪ gamepad1.left_trigger)) / 2;

        r = scaleDriveJoystick(r);
        l = scaleDriveJoystick(l);
        a = scaleActuatorJoystick(a);

        h = scaleActuatorJoystick(h);
        if (h < 0) h *= 0.7;
```

```
        h -= a*0.4;
        h = Range.clip(h, -1, 1);

        robotTurning = (r > 0 && l < 0) || (r < 0 && l > 0);
        robotGoingForward = (r > 0 && l > 0);
        robotGoingBackward = (r < 0 && l < 0);

        adjustDrivePowers(((r + l) / 2) * Util.POWER_LIMIT); // this is mostly stall
            ↪ protection

        this.rightBack.setPower(powerFactor * BACK_SCALE * r);
        this.rightFront.setPower(powerFactor * r);
        this.leftBack.setPower(powerFactor * BACK_SCALE * l);
        this.leftFront.setPower(powerFactor * l);
        this.arm.setPower(a);
        this.hanger.setPower(h);

        // check the gamepad bumpers and set the intake accordingly
        handleIntake();

        // check the gamepad triggers and set the door servos accordingly
        handleDoorServos();

        // if a is pressed on gamepad1, move the front wheels backward and then forward
            ↪ very quickly
// this is used to shake the robot when it is on the mountain, making dumping faster
if (gamepad1.a && r == 0 && l == 0) {
  Util.setFrontPowers(0.35);
          Util.setBackPowers(1 * Util.POWER_LIMIT);
  Thread.sleep(300);
  Util.setAllPowers(0);
}

// if b is pressed on gamepad2, move the right trigger activator out
        if (gamepad2.b) {
            rightTrigger.setPosition(Util.RIGHT_TRIGGER_OUT);
        }

        // if x is pressed on gamepad2, move the left trigger activator out
        if (gamepad2.x) {
            leftTrigger.setPosition(Util.LEFT_TRIGGER_OUT);
        }

        // if a is pressed on gamepad2, move both trigger activators in
        if (gamepad2.a) {
            rightTrigger.setPosition(Util.RIGHT_TRIGGER_IN);
            leftTrigger.setPosition(Util.LEFT_TRIGGER_IN);
        }

// if y is pressed on gamepad2, reset the gyro heading
        if (gamepad2.y) {
            AutoUtil.resetGyroHeading(gyro);
        }

        if (gamepad2.guide && gamepad2.dpad_up) {
            stallProtectionGloballyEnabled = true;
            AutoUtil.resetGyroHeading(gyro);
```

```
        }

        // tip protection
        /*if (Util.SENSORS && gyro.rawZ() < -2000) {
            Util.setAllPowers(0);
            arm.setPower(0);
            while (Math.abs(gamepad1.right_stick_y) > 0.2 && Math.abs(gamepad1.
                ↪ left_stick_y) > 0.2);
        }*/



        // sleep for 2 ms
        Thread.sleep(2, 0);
    }
}

private final double JOYSTICK_DEADZONE_LIMIT = 0.2;
private final double MIN_POWER = 0.2;
private final double POWER_EXPONENT = 1.4;

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;
    /* because exponentiation can change signs,
     * store the sign of the joystick position in a separate variable */
    double posOrNeg = 1;
    if (joystickValue < 0) posOrNeg = -1;
    double power = posOrNeg * Math.pow(Math.abs(joystickValue), POWER_EXPONENT);
    if (Math.abs(power) < MIN_POWER) return 0;
    return Range.clip(power, -1, 1);
}

private double scaleActuatorJoystick(double joystickValue) {
    // if the joystick is in the deadzone I defined, return 0
    if (Math.abs(joystickValue) < JOYSTICK_DEADZONE_LIMIT) return 0;
    /* because exponentiation can change signs,
     * store the sign of the joystick position in a separate variable */
    double posOrNeg = 1;
    if (joystickValue < 0) posOrNeg = -1;
    // adjust the value
    double power = posOrNeg * Math.pow(Math.abs(joystickValue), 1 / POWER_EXPONENT);
    if (Math.abs(power) < MIN_POWER) return 0;
    return Range.clip(power, -1, 1);
}

private void handleDoorServos() {
    // get the joystick positions
    double ls = gamepad2.left_trigger;
    double rs = gamepad2.right_trigger;

    // scale the joystick input to the servo range
    ls = Util.LEFT_DOOR_MIN + (ls * (Util.LEFT_DOOR_MAX - Util.LEFT_DOOR_MIN));
    rs = Util.RIGHT_DOOR_MAX - (rs * (Util.RIGHT_DOOR_MAX - Util.RIGHT_DOOR_MIN));

    // make sure the value is neither less than the min nor greater than the max
    ls = Range.clip(ls, Util.LEFT_DOOR_MIN, Util.LEFT_DOOR_MAX);
```

```
    rs = Range.clip(rs, Util.RIGHT_DOOR_MIN, Util.RIGHT_DOOR_MAX);

    // set the servos to the appropriate position
    leftDoor.setPosition(ls);
    rightDoor.setPosition(rs);
}

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

private void handleIntake() {
    if ((gamepad1.right_bumper || gamepad2.right_bumper) && !intakeChanged) {
        /* if the intake is off, intake
         * if the intake is intaking, outtake
         * if the intake is outtaking, intake
         */
        switch (intakeStatus) {
            case OFF:
                intake();
                break;
            case INTAKE:
                outtake();
                break;
            case OUTTAKE:
                intake();
                break;
        }
        intakeChanged = true;
    }
    if ((gamepad1.left_bumper || gamepad2.left_bumper) && !intakeChanged) {
        /*
         * if the intake is off, outtake
         * if the intake is intaking or outtaking, turn if off
         */
        switch (intakeStatus) {
            case OFF:
                outtake();
                break;
            case INTAKE:
            case OUTTAKE:
                intakeOff();
                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 && !gamepad2.right_bumper &&
        ↪  !gamepad2.left_bumper)
        intakeChanged = false;
}

// the three following methods standardize intaking, outtaking, and neither
private void intake() {
    this.intake.setPower(0.6);
    intakeStatus = INTAKE;
}
```

```
private void outtake() {
    this.intake.setPower(-0.6);
    intakeStatus = OUTTAKE;
}

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

// adjust the drive powers based on the presence or absence of a stall condition
private void adjustDrivePowers(double requestedPower) {
    float currentTime = System.nanoTime();
    // store encoder positions from all drive motors
    StallProtection.storeEncoderData();
    // only check for a stall every 0.01 seconds
    if ((currentTime - checkIntervalTime) > (0.01 * Util.SEC_TO_NSEC)) {
        /* check for a stall when stall protection is globally enabled,
         * the driver is requesting enough power,
         * and the robot is not on the mountain, respectively */
        boolean enabled = stallProtectionGloballyEnabled &&
                (Math.abs(requestedPower) > MOTOR_POWER_THRESHOLD) &&
                ((PID.heading(gyro)) > GYRO_MOUNTAIN_THRESHOLD);
        if (enabled) {
            // store how much each drive encoder moved in the last 0.01 seconds
            StallProtection.manageEncoderData();
            // check for a stall when the drive motors should be at speed
            if ((currentTime - motorRampUpTime) > TIME_THRESHOLD) {
                // check for a stall
                int isStalled = StallProtection.stalled();
                int tempColor = Color.WHITE;
                // if the robot is stalled
                if (isStalled == 1) {
                    //numStalls++;
                    // if the robot did not stall the last time through the loop
                    if (!continuousStall) {
                        // reset the continuous stall timer
                        hasBeenStalledTime = currentTime;
                        // the robot DID stall THIS time through the loop, so...
                        continuousStall = true;
                    }
                    // decrease the power factor
                    powerFactor -= 0.05;
                    tempColor = Color.YELLOW;
                    // if the robot has been continuously stalling for 3 seconds, shut
                        ↪ down the motors
                    if (((currentTime - hasBeenStalledTime) / Util.SEC_TO_NSEC) >
                        ↪ HAS_BEEN_STALLED_LIMIT) {
                        powerFactor = 0;
                        tempColor = Color.RED;
                    }
                    // keep the drive motors going at the minimum power if it hasn't
                        ↪ stalled for three seconds yet
                    else {
                        if (robotGoingForward && (powerFactor < FORWARD_MIN_STALL_POWER))
                            ↪  {
```

```
                    powerFactor = FORWARD_MIN_STALL_POWER;
                } else if (powerFactor < BACKWARD_MIN_STALL_POWER) {
                    powerFactor = BACKWARD_MIN_STALL_POWER;
                }
            }
        }
        // if the robot is not stalled
        else if (isStalled == 0) {
            // increase the power factor
            powerFactor += 0.02;
            // keep the drive motors from going over the power limit
            if (powerFactor > Util.POWER_LIMIT) powerFactor = Util.POWER_LIMIT;
            // reset the continuous stall timer and boolean, respectively
            hasBeenStalledTime = currentTime;
            continuousStall = false;
        }

        // change the backround color of the app to reflect stall status
        /*final int color = tempColor;
        relativeLayout.post(new Runnable() {
            public void run() {
                relativeLayout.setBackgroundColor(color);
            }
        });*/

        // change the background color of the app based on the stall status for
            ↪ debugging purposes
        switch (tempColor) {
            case Color.RED:
                break;
            case Color.YELLOW:
                break;
            case Color.WHITE:
                break;
        }
    }
} else {
    StallProtection.reset();
    // reset the timer that allows the robot to get up to speed before enabling
        ↪ stall protection
    motorRampUpTime = currentTime;
    // if the robot is turning or the driver is requesting enough power, increase
        ↪  the power factor
    if (robotTurning || (Math.abs(requestedPower) > MOTOR_POWER_THRESHOLD)) {
        powerFactor += 0.02;
        // keep the drive motors from going over the power limit
        if (powerFactor > Util.POWER_LIMIT) powerFactor = Util.POWER_LIMIT;
    }
    else {
        // give the driver a decent amount of power to start with
        powerFactor = Util.STARTING_POWER;
    }
    // reset the continuous stall timer and boolean, respectively
    hasBeenStalledTime = currentTime;
    continuousStall = false;
}
// reset the timer that checks for a stall every 0.01 seconds
```

```
            checkIntervalTime = System.nanoTime();
        }
    }
}
```

Autonomous:

```
// 1 encodercount is about 18.3 inches
package com.qualcomm.ftcrobotcontroller;



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

//import com.qualcomm.robotcore.hardware.TouchSensor;
public class Auto_RachelTurn extends LinearOpMode {

    DcMotor motorRight, motorLeft, motorRightFront, motorLeftFront, arm, hanger;
    GyroSensor gyro;
   // int counter=0;
    int count=1024;
    float timer,timer1;
   //TouchSensor touch;

    public Auto_RachelTurn() {}

    public void runOpMode() throws InterruptedException {
        Util.init(this);
        gyro = hardwareMap.gyroSensor.get("gyro");
        AutoUtil.init(this, gyro);

        motorRight = Util.rightBack;
        motorLeft = Util.leftBack;
        motorRightFront = Util.rightFront;
        motorLeftFront = Util.leftFront;
        arm = Util.arm;
        hanger = Util.hanger;

        //touch = hardwareMap.touchSensor.get("Tsensor");

        /*motorRight.setMode(DcMotorController.RunMode.RUN_WITHOUT_ENCODERS);
        motorLeft.setMode(DcMotorController.RunMode.RUN_WITHOUT_ENCODERS);
        arm.setMode(DcMotorController.RunMode.RUN_WITHOUT_ENCODERS);*/

        waitForStart();

        //AutoUtil.moveForward(5.57 * count, 0.5f, gyro);

        //Thread.sleep(500);
        /*timer1=System.nanoTime();
        while ((System.nanoTime()-timer1)<(10 *Util.SEC_TO_NSEC)) {
            AutoUtil.moveBackward(5.37 * count, .5f, gyro);
            //AutoUtil.turnLeft(1 * count, .5f);
            AutoUtil.moveBackward(5.37 * count, .5f, gyro);
            Thread.sleep(500);
        }

        //AutoUtil.turnRight(1 * count, .5f);*/

        // Thread.sleep(500);
```

```
        timer=System.nanoTime();
        arm.setPower(-0.5);
        hanger.setPower(0.25);
        Thread.sleep(2500);
        //while ((System.nanoTime()-timer)<(2 *Util.SEC_TO_NSEC)) {

            //counter++;
       // }



        arm.setPower(0);
        hanger.setPower(0);
        Thread.sleep(2000);
        Util.setAllPowers(0);
        waitOneFullHardwareCycle();
        ;
       // }

        gyro.getHeading();


        //if (gyro.getHeading()==4)
        //{
            //AutoUtil.moveForward(6*1024,.7,gyro);
        //}




    }
}
```

ArmToUp class:

```
package com.qualcomm.ftcrobotcontroller;

public class ArmToUp  {

    public static void armMoveUp(double distance, double power) {

        double start = Util.arm.getCurrentPosition();
        Util.arm.setPower(power);
        while (Util.arm.getCurrentPosition() < (start + (distance * 0.98)));
        Util.setAllPowers(0);
    }

    public static void armMoveDown(double distance, double power) {

        double start = Util.arm.getCurrentPosition();
        Util.arm.setPower(-power);
        while (Util.arm.getCurrentPosition() > (start + (distance * 0.98)));
        Util.setAllPowers(0);
    }
}
```

Util class:

```
package com.qualcomm.ftcrobotcontroller;

import com.qualcomm.ftcrobotcontroller.opmodes.MRRGBExample;
import com.qualcomm.hardware.modernrobotics.ModernRoboticsUsbDcMotorController;
import com.qualcomm.robotcore.eventloop.opmode.OpModeManager;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorController;
import com.qualcomm.robotcore.hardware.Gamepad;
import com.qualcomm.robotcore.hardware.GyroSensor;
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;

public final class Util {

    protected static DcMotor rightBack, rightFront, leftBack, leftFront, arm, intake, hanger;
    protected static Servo rightDoor, leftDoor, rightTrigger, leftTrigger;
    protected static GyroSensor gyro;
    protected static ColorSensor color;
    protected static boolean init = false;
    protected static ModernRoboticsUsbDcMotorController c;
    //protected static boolean gyroEnabled = false;

    protected static final double LEFT_DOOR_MIN = 0.02 /*CLOSED*/, LEFT_DOOR_MAX = 0.4; //
        ↪ OPEN
    protected static final double LEFT_DOOR_CLOSED = LEFT_DOOR_MIN;
    protected static final double RIGHT_DOOR_MIN = 0.5 /*OPEN*/, RIGHT_DOOR_MAX = 0.89; //
        ↪ CLOSED
    protected static final double RIGHT_DOOR_CLOSED = RIGHT_DOOR_MAX;
    protected static final double POWER_LIMIT = 0.7, BACK_SCALE = 1.3;
    protected static final double STARTING_POWER = 0.5;
    protected static final double LEFT_TRIGGER_OUT = 0, LEFT_TRIGGER_IN = 0.6,
        ↪ LEFT_TRIGGER_STOW = 0.65;
    protected static final double RIGHT_TRIGGER_OUT = 0.67, RIGHT_TRIGGER_IN = 0.06,
        ↪ RIGHT_TRIGGER_STOW = 0.04;
    protected static final boolean SENSORS = true, SERVOS = true;

    protected final static double SEC_TO_NSEC = 1000000000, POWER_FLOAT = 100;

    //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;

        // motors
        rightBack = getMotor("right");
        rightBack.setDirection(DcMotor.Direction.REVERSE);
        leftBack = getMotor("left");
```

```
        rightFront = getMotor("rightFront");
        rightFront.setDirection(DcMotor.Direction.REVERSE);
        leftFront = getMotor("leftFront");
        arm = getMotor("arm");
        intake = getMotor("intake");
        hanger = getMotor("hanger");

        c = (ModernRoboticsUsbDcMotorController) rightBack.getController();

        DcMotor[] temp = {rightBack, leftBack, rightFront, leftFront, arm, intake};
        DcMotor[] tempWithEncoders = {rightBack, leftBack, rightFront, leftFront};//, arm};

        motors = temp;
        motorsWithEncoders = tempWithEncoders;

        // servos
        if (SERVOS) {
            leftDoor = getServo("leftDoor");
            leftDoor.setPosition(LEFT_DOOR_CLOSED);
            rightDoor = getServo("rightDoor");
            rightDoor.setPosition(RIGHT_DOOR_CLOSED);
            rightTrigger = getServo("rightTrigger");
            rightTrigger.setPosition(RIGHT_TRIGGER_STOW);
            leftTrigger = getServo("leftTrigger");
            leftTrigger.setPosition(LEFT_TRIGGER_STOW);
        }

        // sensors
        if (SENSORS) {
            gyro = linearOpMode.hardwareMap.gyroSensor.get("gyro");
            //color = linearOpMode.hardwareMap.colorSensor.get("colorSensor1");

            AutoUtil.calibrateGyro(gyro);
            AutoUtil.resetGyroHeading(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(DcMotorController.RunMode.
        ↪ RESET_ENCODERS);
    for (int i = 0; i < 11; i++) opMode.waitOneFullHardwareCycle();
    //while (motorList[0].getMode() != DcMotorController.RunMode.RESET_ENCODERS);
    for (DcMotor motor : motorList) motor.setMode(DcMotorController.RunMode.
        ↪ RUN_WITHOUT_ENCODERS);
    for (int i = 0; i < 11; i++) opMode.waitOneFullHardwareCycle();
}

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 c.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) {
    if (p == POWER_FLOAT) {
        rightBack.setPowerFloat();
        rightFront.setPowerFloat();
        return;
    }
    rightBack.setPower(p);
    rightFront.setPower(p);
}

public static void setLeftPowers(double p) {
    if (p == POWER_FLOAT) {
        leftBack.setPowerFloat();
        leftFront.setPowerFloat();
        return;
    }
```

```
        leftBack.setPower(p);
        leftFront.setPower(p);
    }

    public static void setFrontPowers(double p) {
        if (p == POWER_FLOAT) {
            rightFront.setPowerFloat();
            leftFront.setPowerFloat();
            return;
        }
        rightFront.setPower(p);
        leftFront.setPower(p);
    }

    public static void setBackPowers(double p) {
        if (p == POWER_FLOAT) {
            rightBack.setPowerFloat();
            leftBack.setPowerFloat();
            return;
        }
        rightBack.setPower(p);
        leftBack.setPower(p);
    }

    public static void setAllPowers(double p) {
        if (p == POWER_FLOAT) {
            rightBack.setPowerFloat();
            rightFront.setPowerFloat();
            leftBack.setPowerFloat();
            leftFront.setPowerFloat();
            return;
        }
        rightBack.setPower(p);
        rightFront.setPower(p);
        leftBack.setPower(p);
        leftFront.setPower(p);
    }

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

    public static void registerOpModes(OpModeManager m) {
        //m.register("Auto", Auto.class);
        m.register("TeleOpLinear", TeleOpLinear.class);
        m.register("Auto_RachelTurn", Auto_RachelTurn.class);

        //m.register("____TestingAuto", ____TestingAuto.class);
        //m.register("____TestingTeleOp", ____TestingTeleOp.class);
```

```
        //m.register("____TestingTiming", ____TestingTiming.class);

        m.register("ArmLower", ArmLower.class);
        m.register("ArmRaise", ArmRaise.class);

        m.register("MRRGBExample", MRRGBExample.class);
        //m.register("EncoderTest", EncoderTest.class);
        //m.register("ServoUtility", ServoUtility.class);
    }

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

AutoUtil class:

```
package com.qualcomm.ftcrobotcontroller;

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

public final class AutoUtil {

    protected static boolean init = false;
    protected static LinearOpMode linearOpMode;
    private static float stallEnabledTime;
    private static double powerFactor = Util.POWER_LIMIT;
    private static final double MOTOR_POWER_THRESHOLD = 0.8 * Util.POWER_LIMIT,
        ↪ TIME_THRESHOLD = 0.3 * Util.SEC_TO_NSEC;

    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 moveForward(double distance, double power, GyroSensor gyro) throws
        ↪ InterruptedException {
        resetGyroHeading(gyro);
        double start = Util.rightBack.getCurrentPosition();
        stallEnabledTime = System.nanoTime();
        while (Util.rightBack.getCurrentPosition() < (start + (distance * 0.98))) {
            float currentTime = System.nanoTime();
            if (TeleOpLinear.stallProtectionGloballyEnabled && (Math.abs(power) >
                ↪ MOTOR_POWER_THRESHOLD) && ((currentTime - stallEnabledTime) >
                ↪ TIME_THRESHOLD)) {
                int isStalled = StallProtection.stalled();
                if (isStalled == 1) {
                    powerFactor -= 0.05;
                    if (powerFactor < 0) powerFactor = 0;
                }
                else if (isStalled == 0) {
                    powerFactor += 0.02;
                    if (powerFactor > Util.POWER_LIMIT) powerFactor = Util.POWER_LIMIT;
                }
            }
            //PID.PIsetMotors(gyro, powerFactor * power);
            Util.setFrontPowers(powerFactor * power);
            Util.setBackPowers(powerFactor * Util.BACK_SCALE * power);
            Thread.sleep(10);
        }
        Util.setAllPowers(0);
    }
```

```
public static void moveBackward(double distance, double power, GyroSensor gyro) throws
    ↪ InterruptedException {
    resetGyroHeading(gyro);
    double start = Util.rightBack.getCurrentPosition();
    stallEnabledTime = System.nanoTime();
    while (Util.rightBack.getCurrentPosition() > (start - (distance * 0.98))) {
        float currentTime = System.nanoTime();
        if ((Math.abs(power) > MOTOR_POWER_THRESHOLD) && ((currentTime - stallEnabledTime
            ↪ ) > TIME_THRESHOLD)) {
            int isStalled = StallProtection.stalled();
            if (isStalled == 1) {
                powerFactor -= 0.05;
                if (powerFactor < 0) powerFactor = 0;
            }
            else if (isStalled == 0) {
                powerFactor += 0.02;
                if (powerFactor > Util.POWER_LIMIT) powerFactor = Util.POWER_LIMIT;
            }
        }
        //PID.PIsetMotors(gyro, powerFactor * -power);
        Util.setFrontPowers(powerFactor * -power);
        Util.setBackPowers(powerFactor * Util.BACK_SCALE * -power);
        Thread.sleep(10);
    }
    Util.setAllPowers(0);
}

public static void turnRight(double distance, float power) {//, GyroSensor gyro) {
    //resetGyroHeading(gyro);
    double start = Util.leftBack.getCurrentPosition();
    Util.setRightPowers(-power);
    Util.setLeftPowers(power);
    while (Util.leftBack.getCurrentPosition() < (start + distance * 0.98));
    Util.setAllPowers(0);
}

public static void turnLeft(double distance, float power) {
    double start = Util.rightBack.getCurrentPosition();
    Util.setRightPowers(power);
    Util.setLeftPowers(-power);
    while (Util.rightBack.getCurrentPosition() < (start + distance * 0.98));
    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 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;
    }*/
}
```

StallProtection class:

```
package com.qualcomm.ftcrobotcontroller;

import com.qualcomm.robotcore.hardware.DcMotor;

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

public class StallProtection {

    private static double timeDiff, backAvg, frontAvg, avgDiff, ratio;
    private static int rightBackPos, leftBackPos, rightFrontPos, leftFrontPos;
    private static int lastRightBackPos = 0, lastLeftBackPos = 0, lastRightFrontPos = 0,
        ↪ lastLeftFrontPos = 0;
    private static int rightBackDiff, leftBackDiff, rightFrontDiff, leftFrontDiff;
    private static int rightBackSum = 0, leftBackSum = 0, rightFrontSum = 0, leftFrontSum =
        ↪ 0;
    //protected static boolean init = false;
    private static boolean telemetry = true;
    //private static double ratioMax = 0, ratioMin = 100;

    private static Queue<Integer> rightBackQueue, leftBackQueue, rightFrontQueue,
        ↪ leftFrontQueue;

    private static DcMotor right = Util.rightBack;
    private static DcMotor left = Util.leftBack;
    private static DcMotor rightFront = Util.rightFront;
    private static DcMotor leftFront = Util.leftFront;

    private final static int MOVING_AVERAGE_LENGTH = 10;          // these settings are
        ↪ decent (we think)
    private final static double STALL_RATIO_THRESHOLD_MAX = 1.5;  //1.2
    private final static double STALL_RATIO_THRESHOLD_MIN = 0.4;  //0.65
    private final static double STALL_DIFF_THRESHOLD = 20;        //30

    final static double SEC_TO_NSEC = Util.SEC_TO_NSEC;

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

    public static void init() {
        rightBackQueue = new LinkedList<Integer>();
        leftBackQueue = new LinkedList<Integer>();
        rightFrontQueue = new LinkedList<Integer>();
        leftFrontQueue = new LinkedList<Integer>();

        for (int i = 0; i < MOVING_AVERAGE_LENGTH; i++) {
            rightBackQueue.add(0);
            leftBackQueue.add(0);
            rightFrontQueue.add(0);
            leftFrontQueue.add(0);
        }

        //init = true;
    }
```

```
public static int stalled() {
    /*int result = -1;
    if (!init) return result;*/
    int result = 0;
    if (backAvg == 0) {
        ratio = 10;
    } else {
        ratio = frontAvg / backAvg;
        /*if (ratio > ratioMax) {
            ratioMax = ratio;
        }
        if (ratio < ratioMin) {
            ratioMin = ratio;
        }*/
    }
    if ((ratio > STALL_RATIO_THRESHOLD_MAX) ||
            (ratio < STALL_RATIO_THRESHOLD_MIN) ||
            (avgDiff < STALL_DIFF_THRESHOLD)) {
        result = 1;
        if (telemetry && (ratio > STALL_RATIO_THRESHOLD_MAX)) {
        }
        if (telemetry && (ratio < STALL_RATIO_THRESHOLD_MIN)) {
        }
        if (telemetry && (avgDiff < STALL_DIFF_THRESHOLD)) {
        }
        telemetry = false;
    }
    return result;
}

public static void reset() {
    lastRightBackPos = right.getCurrentPosition();
    lastLeftBackPos = left.getCurrentPosition();
    lastRightFrontPos = rightFront.getCurrentPosition();
    lastLeftFrontPos = leftFront.getCurrentPosition();

    for (int i = 0; i < MOVING_AVERAGE_LENGTH; i++) {
        rightBackQueue.poll();
        rightBackQueue.add(0);
        leftBackQueue.poll();
        leftBackQueue.add(0);
        rightFrontQueue.poll();
        rightFrontQueue.add(0);
        leftFrontQueue.poll();
        leftFrontQueue.add(0);
    }

    rightBackSum = 0;
    leftBackSum = 0;
    rightFrontSum = 0;
    leftFrontSum = 0;

    /*ratioMax = 0;
    ratioMin = 100;*/
    avgDiff = 0;

    telemetry = true;
```

```
    }

    public static void manageEncoderData() {
        rightBackDiff = Math.abs(rightBackPos - lastRightBackPos);
        leftBackDiff = Math.abs(leftBackPos - lastLeftBackPos);
        rightFrontDiff = Math.abs(rightFrontPos - lastRightFrontPos);
        leftFrontDiff = Math.abs(leftFrontPos - lastLeftFrontPos);

        rightBackSum = rightBackSum + rightBackDiff - rightBackQueue.poll();
        rightBackQueue.add(rightBackDiff);
        leftBackSum = leftBackSum + leftBackDiff - leftBackQueue.poll();
        leftBackQueue.add(leftBackDiff);
        rightFrontSum = rightFrontSum + rightFrontDiff - rightFrontQueue.poll();
        rightFrontQueue.add(rightFrontDiff);
        leftFrontSum = leftFrontSum + leftFrontDiff - leftFrontQueue.poll();
        leftFrontQueue.add(leftFrontDiff);

        backAvg = (rightBackSum + leftBackSum) / (2 * MOVING_AVERAGE_LENGTH);
        frontAvg = (rightFrontSum + leftFrontSum) / (2 * MOVING_AVERAGE_LENGTH);
        avgDiff = (backAvg + frontAvg) / 2;

        lastRightBackPos = rightBackPos;
        lastLeftBackPos = leftBackPos;
        lastRightFrontPos = rightFrontPos;
        lastLeftFrontPos = leftFrontPos;
    }

    public static void storeEncoderData() {
        rightBackPos = right.getCurrentPosition();
        leftBackPos = left.getCurrentPosition();
        rightFrontPos = rightFront.getCurrentPosition();
        leftFrontPos = leftFront.getCurrentPosition();
    }
}
```

PID class:

```
package com.qualcomm.ftcrobotcontroller;

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

public final class PID {

    private static final float Ku = 0.15f;    //ultimate gain         //TODO tune
    private static final float Tu = 0f;       //oscillation period    //TODO tune
    //private static final float Tp = 50;    //average turn power    //TODO tune
    private static final float Kp = 0.45f * Ku; //proportional constant //TODO tune
    private static final float Ki = Tu / 1.2f;  //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 = 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 * 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("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, float Tp) {
        double[] motors = P(gyro, Tp);
        Util.setRightPowers(motors[0]);
        Util.setLeftPowers(motors[1]);

    }

    public static void PIsetMotors(GyroSensor gyro, float 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
    }
}
```

configuration file:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>

<Robot>
    <!-- drive motors -->
    <MotorController name="FrontMotors" serialNumber="AL00UW4Z">
        <Motor name="leftFront" port="1" />
        <Motor name="rightFront" port="2" />
    </MotorController>
    <MotorController name="BackMotors" serialNumber="AL00VUTY">
        <Motor name="left" port="1" />
        <Motor name="right" port="2" />
    </MotorController>
    <!-- actuators -->
    <MotorController name="Actuators1" serialNumber="AL00VLUU">
        <Motor name="updown" port="1" />
    </MotorController>
    <MotorController name="Actuators2" serialNumber="AL00VG65">
        <Motor name="intake" port="1" />
    </MotorController>
    <!-- servos -->
    <ServoController name="Servos" serialNumber="AL00VSDY">
        <Servo name="leftDoor" port="1" />
        <Servo name="rightDoor" port="2" />
        <Servo name="rightTrigger" port="3" />
        <Servo name="leftTrigger" port="4" />
    </ServoController>
    <!-- sensors -->
    <DeviceInterfaceModule name="Sensors" serialNumber="AL00VFK6">
        <Gyro name="gyro" port="0" />
    </DeviceInterfaceModule>
</Robot>
```