```cpp
#include "includes.h"
#include <array>


//subsystem objects
Drive *drive = new Drive();
Pneumatics *fourbarpneum = new Pneumatics(FRONT_PNEUM);
Pneumatics *backclamppneum = new Pneumatics(BACK_PNEUM);
Effectors effectors;
Intake *intake = new Intake(INTAKE_PORT);
Intake *fourbar1 = new Intake(FOUR_BAR_FIRST);
Intake *fourbar2 = new Intake(FOUR_BAR_FIRST);
Button *buttons = new Button();
pros::Imu imu(IMUPORT);
pros::ADIDigitalIn but('H');
okapi::Controller controller (okapi::ControllerId::master);

double speeds[3] = {150, 150, 150};

PIDConst forwardDefault = {0.035, 0.00005, 0};
PIDConst headingDefault = {0, 0, 0};
PIDConst turnDefault = {0.02, 0.000025, 0};

int route;

/**
 * A callback function for LLEMU's center button.
 *
 * When this callback is fired, it will toggle line 2 of the LCD text between
 * "I was pressed!" and nothing.
 */
void on_center_button() {
    static bool pressed = false;
    pressed = !pressed;
    if (pressed) {
        pros::lcd::set_text(2, "I was pressed!");
    } else {
        pros::lcd::clear_line(2);
    }
}

void autonSelector() {
    while(1) {
        if(controller.getDigital(okapi::ControllerDigital::A)) {
            route = 1;
            break;
        }
        if(controller.getDigital(okapi::ControllerDigital::B)) {
            route = 2;
            break;
        }
        if(controller.getDigital(okapi::ControllerDigital::X)) {
            route = 3;
            break;
        }
        if(controller.getDigital(okapi::ControllerDigital::Y)) {
            route = 4;
```

```
            break;
        }
        pros::delay(15);
    }
}


/**
 * Runs initialization code. This occurs as soon as the program is started.
 *
 * All other competition modes are blocked by initialize; it is recommended
 * to keep execution time for this mode under a few seconds.
 */


void initialize() {
    //make sure four bar can't go higher/lower than the mechanical stops
    fourbar1->setLimits(2400, 0);
    fourbar2->setLimits(2400, 0);

    pros::lcd::initialize();
    pros::lcd::set_text(1, "Hello PROS User!");

    //autonSelector();

    //calibrate imu
    imu.reset();

    int time = pros::millis();
    int iter = 0;
    while (imu.is_calibrating())
    {
        printf("IMU calibrating... %d\n", iter);
        iter += 10;
        pros::delay(10);
    }
    imu.set_data_rate(10);
    pros::lcd::register_btn1_cb(on_center_button);
}

//x-drive math to find difference between two odomstates
OdomState transform(OdomState curr, OdomState target) {
    OdomState diff;
    diff.x = target.x - curr.x;
    diff.y = target.y - curr.y;
    diff.theta = OdomMath::constrainAngle180(target.theta-curr.theta);


    //rotate vector
    diff.x = diff.y * sin(curr.theta.convert(radian)) +
diff.x*cos(curr.theta.convert(radian));
    diff.y = diff.y*cos(curr.theta.convert(radian)) - diff.x*sin(curr.theta.convert(radian));

    return diff;
}
```

```
//slew limiter for PID
double limiter(double prevOutput, double currOutput, double step) {
    double output;
    if(currOutput > 0){ // positive rawOutput case

        output = std::clamp(currOutput, prevOutput - step, std::min(1.0, prevOutput + step));
// clamped for slew and so finalOutput does not exceed maxOutput
        output = std::max(0.2, output); //make sure output above min power

    } else if (currOutput < 0){ // negative rawOutput case

        output = std::clamp(currOutput, std::max(-1.0, prevOutput - step), prevOutput +
step); // clamped for slew and so finalOutput does not exceed -maxOutput
        output = std::min(-0.2, output); //make sure output above min power

    } else { // rawOutput is 0

        output = 0; // step will return 0
    }
    return output;
}


//overhauled move function
void pidMoveForward(OdomState target, PIDConst forwardConstants, PIDConst headingConstants,
double timeout = 5) {
    double forward, turn, prevForward, prevTurn;
    QLength magerr;
    QAngle headerr;
    QAngle targetAngle;
    OdomState currState;
    QLength xDiff, yDiff;
    prevForward = 0;
    prevTurn = 0;

    PID forwardObj = PID(forwardConstants);
    PID turnObj = PID(headingConstants);

    double startTime = pros::millis();
    do {
        currState = drive->getState();
        xDiff = target.x-currState.x;
        yDiff = target.y-currState.y;

        targetAngle = okapi::OdomMath::constrainAngle180((PI/2 - atan2(xDiff.convert(meter),
yDiff.convert(meter)))*1_rad);
        targetAngle = 1_deg * targetAngle.convert(degree);

        //calculate errors
        QAngle curr = okapi::OdomMath::constrainAngle180(imu.get_heading()*1_deg);
        headerr = okapi::OdomMath::constrainAngle180(curr-targetAngle);
        magerr = sqrt((xDiff * xDiff) + (yDiff * yDiff));

        if(abs(magerr.convert(inch))<10) {
            headerr = 0_deg;
        }

        //if overshoot point, reverse direction and target heading
```

```
        if(abs(headerr.convert(degree)) > 100) {
            headerr = okapi::OdomMath::constrainAngle180(headerr-180_deg);
            magerr*=-1;
        }

        //limit and set motors
        forward = limiter(prevForward, forwardObj.step(magerr.convert(inch)), 0.11);
        turn = limiter(prevTurn, turnObj.step(headerr.convert(degree)), 0.11);
        printf("%f %f %f\n", drive->getX(), drive->getY(), drive->getHeading());
        drive->runTankArcade(forward, turn);
        prevForward = forward;
        prevTurn = turn;
        pros::delay(10);
    } while((abs(magerr.convert(inch)) > 2) && pros::millis()-startTime > (timeout*1000) );
//timeout and check mag err
    drive->runTankArcade(0, 0);
}

//overhauled turn function
void pidTurn(QAngle targetHeading, PIDConst turnConstants, double timeout = 3) {
    double turn, prevTurn;
    QAngle headerr;
    QAngle targetAngle;
    OdomState currState;
    QLength xDiff, yDiff;
    prevTurn = 0;

    targetAngle = targetHeading;

    PID turnObj = PID(turnConstants);
    do {
        currState = drive->getState();


        //calculate errors
        QAngle curr = okapi::OdomMath::constrainAngle180(imu.get_heading()*1_deg);
        headerr = okapi::OdomMath::constrainAngle180(curr-targetAngle);


        //limit and set motors
        turn = limiter(prevTurn, turnObj.step(headerr.convert(degree)), 0.11);
        printf("%f %f %f %f\n", drive->getX(), drive->getY(), drive->getHeading(),
abs(headerr.convert(degree))>3);
        drive->runTankArcade(0, turn);
        prevTurn = turn;
        pros::delay(10);
    } while(abs(headerr.convert(degree))>3);
    drive->runTankArcade(0, 0);
}

//move to any point
void moveToPoint(OdomState target, PIDConst forwardConstants, PIDConst headingConstants,
PIDConst turnConstants, double timeoutforward = 5, double timeoutturn = 3) {
    OdomState currState = drive->getState();
    QLength xDiff = target.x-currState.x;
    QLength yDiff = target.y-currState.y;

    QAngle targetAngle = okapi::OdomMath::constrainAngle180((PI/2 -
```

```
atan2(xDiff.convert(meter), yDiff.convert(meter)))*1_rad);
    targetAngle = 1_deg * targetAngle.convert(degree);

    pidTurn(targetAngle, turnConstants, timeoutturn);
    printf("Done turning");
    pidMoveForward(target, forwardConstants, headingConstants, timeoutforward);
}

//PID move function that can handle turns and forward movements
void pidMoveTank(OdomState target, PIDConst forwardConstants = forwardDefault, PIDConst
turnConstants = headingDefault, bool turning = false) {
    double forward, turn, prevForward, prevTurn;
    QLength magerr;
    QAngle headerr;
    QAngle targetAngle;
    OdomState currState;
    QLength xDiff, yDiff;
    prevForward = 0;
    prevTurn = 0;

    //forward and turn objects for PID
    PID forwardObj = PID(forwardConstants);
    PID turnObj = PID(turnConstants);

    do {
        currState = drive->getState();
        xDiff = target.x-currState.x;
        yDiff = target.y-currState.y;

        //calculate target thetas differently depending on forward or turn
        if(!turning) {
            targetAngle = okapi::OdomMath::constrainAngle180((PI/2 -
atan2(xDiff.convert(meter), yDiff.convert(meter)))*1_rad);
            targetAngle = 1_deg * targetAngle.convert(degree);
        }
        else {
            targetAngle = target.theta;
        }

        //calculate errors
        QAngle curr = okapi::OdomMath::constrainAngle180(imu.get_heading()*1_deg);
        headerr = okapi::OdomMath::constrainAngle180(curr-targetAngle);
        magerr = sqrt((xDiff * xDiff) + (yDiff * yDiff));

        //if overshoot point, reverse direction and target heading
        if(abs(headerr.convert(degree)) > 100 && forwardConstants.kp != 0) {
            headerr = okapi::OdomMath::constrainAngle180(headerr-180_deg);
            magerr*=-1;
        }

        //limit and set motors
        forward = limiter(prevForward, forwardObj.step(magerr.convert(inch)), 0.11);
        turn = limiter(prevTurn, turnObj.step(headerr.convert(degree)), 0.11);
        printf("%f %f %f\n", drive->getX(), drive->getY(), drive->getHeading());
        drive->runTankArcade(forward, turn);
        prevForward = forward;
        prevTurn = turn;
        pros::delay(10);
```

```
278     } while((abs(magerr.convert(inch)) > 3 && !turning) || (abs(headerr.convert(degree))>3 &&
turning)); //tolerances checked differently depending on turning or forward
279     drive->runTankArcade(0, 0);
280 }
281
282
283 //use odometry magnitude error to move a set distance
284 void distanceMove(double distance, double speed) {
285     OdomState initial = drive->getState();
286     double error = 0;
287     // double start = drive->getEncoder();
288     drive->runTankArcade(speed, 0);
289     double start = pros::millis();
290     do {
291         OdomState temp = drive->getState();
292         QLength xdiff = temp.x-initial.x;
293         QLength ydiff = temp.y-initial.y;
294         printf("Odom: %f %f %f\n", temp.x.convert(inch), temp.y.convert(inch),
temp.theta.convert(degree));
295         //  error = ((drive->getEncoder() - start) / 360*(7.0/5)) * 4 * PI ;
296         error = okapi::sqrt((xdiff*xdiff) + (ydiff*ydiff)).convert(inch);
297         pros::delay(30);
298     } while(error<distance && pros::millis()-start<6000);
299     drive->runTankArcade(0, 0);
300 }
301
302 void distancePID(double distance, PIDConst gains) {
303     OdomState initial = drive->getState();
304     double error = 0;
305     double speed = 0;
306     double prevSpeed = 0;
307     PID obj = PID(gains);
308     do {
309         OdomState temp = drive->getState();
310         QLength xdiff = temp.x-initial.x;
311         QLength ydiff = temp.y-initial.y;
312         printf("Odom: %f %f %f\n", temp.x.convert(inch), temp.y.convert(inch),
temp.theta.convert(degree));
313         error = okapi::sqrt((xdiff*xdiff) + (ydiff*ydiff)).convert(inch);
314         speed = limiter(prevSpeed, obj.step(distance>0 ? distance-error: distance+error),
0.11);
315         prevSpeed = speed;
316         drive->runTankArcade(speed, 0);
317         pros::delay(30);
318     } while(error<abs(distance));
319     drive->runTankArcade(0, 0);
320 }
321
322 void speedMove(double time, double speed) {
323
324     double start = pros::millis();
325     printf("%f\n", start);
326     drive->runTankArcade(speed, 0);
327     while(pros::millis()-start<time) {
328         pros::delay(10);
329     }
330     double end = pros::millis();
331     printf("%f\n", end);
```

```
    drive->runTankArcade(0, 0);
}

void setEffectorPositions() {
    //set all effector positions
    effectors.addPosition();
}

void turnTest() {
    OdomState goal = drive->getState();
  while(1) {
    goal.theta = 90_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000005, 0}, true);
    pros::delay(1000);
    goal = drive->getState();
    goal.theta = 180_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000005, 0}, true);
    pros::delay(1000);
    goal = drive->getState();
    goal.theta = 270_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000005, 0}, true);
  }
}


//drag turn function
void dragTurn(double heading, double direction, double side) {
    while(abs(heading-imu.get_heading())>2) {
        //handle side and direction logic
        if(side == 0) {
            drive->runTank(0.5*direction, 0.1*direction*-1);
        }
        if(side == 1) {
            drive->runTank(0.1*direction*-1, 0.5*direction);
        }
        pros::delay(10);
    }
    drive->runTank(0, 0);
}


//experimental autonSelector function


void autobalancer(double tolerance) {
    drive->runTankArcade(-1, 0);
    pros::delay(2000);
    double curr_pitch = imu.get_pitch();
    double last_pitch = curr_pitch;
    while(curr_pitch > tolerance || curr_pitch-last_pitch>=0) {
        pros::delay(5);
        printf("Current pitch: %f\n", curr_pitch);
        last_pitch = curr_pitch;
        curr_pitch = imu.get_pitch();
    }
    drive->runTankArcade(0, 0);
    drive->setMode(okapi::AbstractMotor::brakeMode::hold);
```

```
    pros::delay(10000);
}

void testBalancing() {
    setEffectorPositions();

    fourbarpneum->turnOn();
    fourbar1->moveTarget(2400);
    fourbar2->moveTarget(2400);
    pros::delay(2000);
    fourbar1->moveTarget(0);
    fourbar2->moveTarget(0);

    pros::delay(2000);
    autobalancer(22);
}

void moveUntilButton(double speed) {
    drive->runTankArcade(speed, 0);
    while(!but.get_value()) {
        pros::delay(10);
    }
    drive->runTankArcade(0, 0);
}
/**
 * Runs while the robot is in the disabled state of Field Management System or
 * the VEX Competition Switch, following either autonomous or opcontrol. When
 * the robot is enabled, this task will exit.
 */
void disabled() {}

/**
 * Runs after initialize(), and before autonomous when connected to the Field
 * Management System or the VEX Competition Switch. This is intended for
 * competition-specific initialization routines, such as an autonomous selector
 * on the LCD.
 *
 * This task will exit when the robot is enabled and autonomous or opcontrol
 * starts.
 */
void competition_initialize() {}

/**
 * Runs the user autonomous code. This function will be started in its own task
 * with the default priority and stack size whenever the robot is enabled via
 * the Field Management System or the VEX Competition Switch in the autonomous
 * mode. Alternatively, this function may be called in initialize or opcontrol
 * for non-competition testing purposes.
 *
 * If the robot is disabled or communications is lost, the autonomous task
 * will be stopped. Re-enabling the robot will restart the task, not re-start it
 * from where it left off.
 */

/**
 * Runs the operator control code. This function will be started in its own task
 * with the default priority and stack size whenever the robot is enabled via
```

```
 * the Field Management System or the VEX Competition Switch in the operator
 * control mode.
 *
 * If no competition control is connected, this function will run immediately
 * following initialize().
 *
 * If the robot is disabled or communications is lost, the
 * operator control task will be stopped. Re-enabling the robot will restart the
 * task, not resume it from where it left off.
 */

 void driverMovementTrack() {
    //initialize variables and set effector positions
    setEffectorPositions();
    int parking = 0;
    double forward;
    double turn;
    double strafe;
    int i = 0;
    bool fourbarpneumstate = true;
    bool backclampstate = false;
    ADIEncoder righttrack = ADIEncoder('A', 'B', false);        //encoders because i don't know how to get values
    ADIEncoder lefttrack = ADIEncoder('C', 'D', true);
    righttrack.reset();
    lefttrack.reset();
    okapi::Motor fourbar(FOUR_BAR_FIRST);

    double max = 1;
    drive->setMode(okapi::AbstractMotor::brakeMode::hold);
    fourbarpneum->turnOn();
    backclamppneum->turnOff();
    while(true) {
        //toggle between coast and hold brake modes
        //get controller and drive chassis base
        // printf("%f %f\n", righttrack.get(), lefttrack.get());
        // printf("%f %f %d\n", drive->getX(), drive->getY(), (int)drive->getHeading()%360);
        //update all button values
        buttons->handleButtons(controller);
        int buttonCounts[9];
        for(int i = 0; i < 9; i++) {
            buttonCounts[i] = buttons->getCount(buttons->buttonList[i]);
        }

        if(buttonCounts[7]%2) {
            max = 5.0/7;
            drive->setMode(okapi::AbstractMotor::brakeMode::hold);
        }
        else {
            max = 1;
            drive->setMode(okapi::AbstractMotor::brakeMode::coast);
        }

        forward = controller.getAnalog(okapi::ControllerAnalog::leftY);
        turn = controller.getAnalog(okapi::ControllerAnalog::rightX);
        if(forward>=0) {
            drive->runTankArcade(std::max(forward*-(6.0/7), max*-1), turn*-0.6);
```

```
        }
        else {
            drive->runTankArcade(std::min(forward*-(6.0/7), max), turn*-0.6);
        }

        // effectors.step(buttonCounts, speeds); //handle two bar

        //intake->run(false, buttons->getPressed(okapi::ControllerDigital::right), 150);
//handle intake
        //runs the intake backwards
        if (buttonCounts[8]%2 == 1) {
            intake->handle(buttonCounts[8], -180);
        }
        else {
            intake->handle(buttonCounts[3], 180); //handle intake (toggle)
        }

        //handle four bar
        fourbar1->run(buttons->getPressed(okapi::ControllerDigital::R1), buttons-
>getPressed(okapi::ControllerDigital::R2), 175);
        //fourbar2->run(buttons->getPressed(okapi::ControllerDigital::R1), buttons-
>getPressed(okapi::ControllerDigital::R2), 175);

        //handle clamp
        fourbarpneum->handle(buttonCounts[5]);
        backclamppneum->handle(buttonCounts[0]);
        // parking = buttonCounts[7] % 2;
        // if (parking == 1) {
        //  drive->setMode(okapi::AbstractMotor::brakeMode::hold);
        // } else {
        //  drive->setMode(okapi::AbstractMotor::brakeMode::coast);
        // }
        //printf("%d\n", parking);

        if (buttonCounts[1]%2) {
            printf("Heading: %f    Distance from last point: %f inches     Four Bar position:
%f\n\n", imu.get_heading(), ((righttrack.get() + lefttrack.get())/2 * 2.81665 * PI / 360),
fourbar.getEncoder());
            righttrack.reset();
            lefttrack.reset();
        }

        pros::delay(60);
        pros::lcd::clear_line(2);
    }
 }

void opcontrol() {



    //initialize variables and set effector positions
    setEffectorPositions();
    int parking = 0;
    double forward;
    double turn;
    double strafe;
    int i = 0;
```

```
555     bool fourbarpneumstate = true;
556     bool auxilclampstate = false;
557     ADIEncoder righttrack = ADIEncoder('A', 'B', false);        //encoders because i don't
know how to get values
558     ADIEncoder lefttrack = ADIEncoder('C', 'D', true);
559
560   double max = 1;
561   drive->setMode(okapi::AbstractMotor::brakeMode::hold);
562     while(true) {
563         //toggle between coast and hold brake modes
564         //get controller and drive chassis base
565         printf("%f %f\n", righttrack.get(), lefttrack.get());
566         // printf("%f %f %d\n", drive->getX(), drive->getY(), (int)drive->getHeading()%360);
567         //update all button values
568         buttons->handleButtons(controller);
569         int buttonCounts[9];
570         for(int i = 0; i < 9; i++) {
571             buttonCounts[i] = buttons->getCount(buttons->buttonList[i]);
572         }
573
574     if(buttonCounts[7]%2) {
575       max = 5.0/7;
576     }
577     else {
578       max = 1;
579     }
580
581     forward = controller.getAnalog(okapi::ControllerAnalog::leftY);
582         turn = controller.getAnalog(okapi::ControllerAnalog::rightX);
583     if(forward>=0) {
584             drive->runTankArcade(std::max(forward*-(6.0/7), max*-1), turn*-0.6);
585     }
586     else {
587         drive->runTankArcade(std::min(forward*-(6.0/7), max), turn*-0.6);
588     }
589
590         // effectors.step(buttonCounts, speeds); //handle two bar
591
592         //intake->run(false, buttons->getPressed(okapi::ControllerDigital::right), 150);
//handle intake
593         if (buttonCounts[8]%2 == 1) {
594             intake->handle(buttonCounts[8], -200);
595         }
596         else {
597             intake->handle(buttonCounts[3], 200); //handle intake (toggle)
598         }
599
600         //handle four bar
601         fourbar1->run(buttons->getPressed(okapi::ControllerDigital::R1), buttons-
>getPressed(okapi::ControllerDigital::R2), 175);
602         //fourbar2->run(buttons->getPressed(okapi::ControllerDigital::R1), buttons-
>getPressed(okapi::ControllerDigital::R2), 175);
603
604         //handle clamp
605         fourbarpneum->handle(buttonCounts[5]);
606         backclamppneum->handle(buttonCounts[0]);
607         parking = buttonCounts[7] % 2;
608         if (parking == 1) {
```

```
            drive->setMode(okapi::AbstractMotor::brakeMode::hold);
        } else {
            drive->setMode(okapi::AbstractMotor::brakeMode::coast);
        }
        //printf("%d\n", parking);
        pros::delay(60);
        pros::lcd::clear_line(2);
    }
}

void right() {
  setEffectorPositions();
    printf("done\n");
    distanceMove(39, -1);   //move towards side neutral at full speed
    fourbarpneum->turnOn(); //clamp it
    pros::delay(100);
    printf("Finished\n");
    distanceMove(10, 1); //move back
    distanceMove(4, 0.6);


    pidTurn(270_deg, {0.01, 0.000008, 0});  //make the turn

    drive->runTankArcade(0.5, 0); //move towards alliance goal
    pros::delay(1400);
    backclamppneum->turnOn();
    drive->runTankArcade(-0.5, 0);
    pros::delay(1000);
    drive->runTankArcade(0, 0);
    intake->run(true, false, -180);



}

void leftfast() {
    setEffectorPositions();

    distanceMove(42, -1);   //move towards side neutral at full speed
    fourbarpneum->turnOn(); //clamp it
    printf("Finished\n");
    distanceMove(30, 1); //move back
}

void thenewnewskills() {

    //first alliance pickup
    setEffectorPositions();
    distanceMove(10, -0.5);
    pidTurn(270_deg, {0.007, 0.000008, 0});
    drive->runTankArcade(0.5, 0);
    pros::delay(1000);
    backclamppneum->turnOn();
    pidTurn(270_deg, {0.007, 0.000008, 0});
    drive->runTankArcade(0, 0);
    distanceMove(15, -0.5);
    // drive->runTankArcade(0, 0);
```

```
    intake->run(true, false, -180); //start intake
    pidTurn(0_deg, {0.007, 0.000008, 0});


    //first neutral and to goal
    // distanceMove(15, -0.8);
    // distancePID(-15, {0.01, 0.0000008, 0});
    // pidTurn(0_deg, {0.006, 0.000008, 0});
    distanceMove(34, -0.6);
    fourbarpneum->turnOn(); //clamp
    pros::delay(100);
    fourbar1->moveTarget(2400);
    pidTurn(331_deg, {0.020, 0.000009, 0}); //turn to seesaw
    distanceMove(51, -0.3); //move to seesaw
    // distancePID(-27, {0.01, 0.0000008, 0});
    pros::delay(300);
    fourbar1->moveTarget(1900);
    pros::delay(500);
    fourbarpneum->turnOff();
    pros::delay(200);
    // fourbar1->moveTarget(2400);

    //alliance currently in two bar
    distanceMove(18, 0.4);
    fourbar1->moveTarget(0);
    backclamppneum->turnOff();
    //move forwards and turn 180
    pros::delay(1000);
    distanceMove(9, -0.4);
    double curr = imu.get_heading();
    pidTurn((curr+175) * 1_deg, {0.005, 0.000008, 0});
    //move forwards and clamp on goal
    distanceMove(20, -0.4);
    fourbarpneum->turnOn();
    pros::delay(100);
    //raise four bar
    fourbar1->moveTarget(2100);
    //turn back towards seesaw
    pidTurn(345_deg, {0.006, 0.000008, 0});
    //forward to seesaw
    distanceMove(30, -0.6);
    //drop goal
    fourbar1->moveTarget(1900);
    pros::delay(700);
    fourbarpneum->turnOff();
    pros::delay(200);

    //alliance under first seesaw
    printf("Moving back\n");
    distanceMove(12, 0.4); //move back from seesaw
    fourbar1->moveTarget(0); //lower four bar
    pidTurn(272_deg, {0.009, 0.000008, 0}); //turn to the wall
    //distanceMove(33, 0.4); //forward
    drive->runTankArcade(0.4, 0); //run into wall
    pros::delay(3500);
    drive->runTankArcade(0, 0);
    distanceMove(1, -0.5);
```

```
    pidTurn(318_deg, {0.011, 0.000008, 0}); //turn towards goal under seesaw
    distanceMove(31, -0.4); //forwards to that goal
    fourbarpneum->turnOn(); //clamp
    pros::delay(200);
    distanceMove(20, 0.4); //get back out
    fourbar1->moveTarget(2400); //raise four bar
    pidTurn(226_deg, {0.010, 0.00001, 0}); //turn towards other side seesaw
    distanceMove(84, -0.6); //beeline there
    pros::delay(400);
    fourbar1->moveTarget(1400); //lower four bar
    pros::delay(1000);
    fourbarpneum->turnOff(); //drop clamp

    fourbar1->moveTarget(2400); //raise four bar

    //tall neutral
    distanceMove(13, 0.6);//back from seesaw
    fourbar1->moveTarget(0);
    pidTurn(305_deg, {0.014, 0.000008, 0}); //turn towards tall neutral
    distanceMove(30, -0.6); //beeline to tall
    fourbarpneum->turnOn();
    pros::delay(400);
    fourbar1->moveTarget(300);
    distanceMove(50, -0.6); //beeline to tall
    fourbarpneum->turnOff(); // drop tall
    pros::delay(500);
    distanceMove(9, 0.6);

    //side alliance with back clamp
    pidTurn(89_deg, {0.006, 0.000008, 0}); //turn towards side alliance
    drive->runTankArcade(0.5, 0);
    pros::delay(1700);
    backclamppneum->turnOn();
    drive->runTankArcade(0, 0);
    distanceMove(15, -0.5);
    fourbar1->moveTarget(0);
    pidTurn(180_deg, {0.007, 0.000008, 0});

    //last neutral goal
    distanceMove(38, -0.8);
    fourbarpneum->turnOn(); //clamp
    pros::delay(100);
    fourbar1->moveTarget(2400);
    pidTurn(146_deg, {0.01, 0.000009, 0}); //turn to seesaw
    distanceMove(40, -0.5); //move to seesaw
    pros::delay(300);
    fourbar1->moveTarget(2100);//drop four bar
    pros::delay(500);
    fourbarpneum->turnOff(); //drop neutral goal
    pros::delay(200);
    fourbar1->moveTarget(2400); //four bar back up

    //deposit alliance on seesaw
    distanceMove(18, 0.4); // away from seesaw
    fourbar1->moveTarget(0);//lower four bar
    backclamppneum->turnOff(); //let go of alliance goal
    //move forwards and turn 180
```

```
    distanceMove(8, -0.4); //go forward to lose the goal
    curr = imu.get_heading();
    pidTurn((curr+170)*1_deg, {0.008, 0.000008, 0}); //turn back towards the goal
    distanceMove(13, -0.4); //move forwards and clamp on goal
    fourbarpneum->turnOn();
    pros::delay(100);
    //raise four bar
    fourbar1->moveTarget(2100);
    //turn back towards seesaw
    pidTurn(155_deg, {0.009, 0.000008, 0});
    //forward to seesaw
    distanceMove(40, -0.6);
    //drop goal
    fourbarpneum->turnOff();
    distanceMove(15, 0.6);
    pros::delay(200);
}

void skills() {
    setEffectorPositions();
    printf("done\n");
    effectors.runOne(GOAL_LIFT, 1); //lower two-bar

    distanceMove(43, -1);   //move towards side neutral at full speed
    fourbarpneum->turnOn(); //clamp it
    pros::delay(300); //delay 300 ms
    printf("Finished\n");
    distanceMove(9, 1); //move back



    OdomState goal = drive->getState();
    goal.theta = 310_deg;  //turn backside towards alliance goal
    pidMoveTank(goal, {0, 0, 0}, turnDefault, true);  //make the turn

    pros::delay(300);
    distanceMove(12, 1); //move towards alliance goal
    effectors.runOne(GOAL_LIFT, 0);


    goal = drive->getState();
    goal.theta = 180_deg; //
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true); //turn to dump goal
    intake->run(true, false, -200); //start intake
    fourbarpneum->turnOff(); //dump goal
    pros::delay(750); //wait
    goal = drive->getState();
    goal.theta = 295_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true); //turn towards center goal
    pros::delay(200);
    intake->run(true, false, 0); //end intake
    fourbar1->moveTarget(0); //lower four bar
    fourbar2->moveTarget(0); //lower four bar
    distanceMove(54, -1); //move towards center goal
    fourbarpneum->turnOn(); //clamp
    pros::delay(300); //wait
    //goal.theta = 225_deg;
```

```
    //pidMoveTank(goal, {0, 0, 0}, {0.02, 0.000005, 0}, true); //turn slightly to be able to get back into home zone
    distanceMove(30, -1); //move goal forwards
    // goal.theta = 225_deg;
    // pidMoveTank(goal, {0, 0, 0}, {0.02, 0.000005, 0}, true); //drop neutral
    fourbarpneum->turnOff();
    distanceMove(5, 1);
    goal.theta = 225_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.02, 0.000005, 0}, true); //turn towards other neutral
    distanceMove(30, -1);
    fourbarpneum->turnOn();
    pros::delay(600);
    // goal.theta = 0_deg;
    // pidMoveTank(goal, {0, 0, 0}, {0.02, 0.000005, 0}, true); //turn towards other zone
    distanceMove(35, 1);
}

void newSkills() {
    setEffectorPositions();
    effectors.runOne(GOAL_LIFT, 1); //two bar down
    pros::delay(750);               //wait
    distanceMove(16, 0.7);          //Move backwards to grab alliance goal
    effectors.runOne(GOAL_LIFT, 0); //two bar up
    distanceMove(2, -0.4);          //moving away from seesaw
    OdomState goal = drive->getState();
    goal.theta = 270_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000005, 0}, true);   //turn to 90 deg
//-----------------------------------------------------------------
//FIRST NEUTRAL

    distanceMove(24, 1);            //moving towards goal
    goal.theta = 90_deg;            //stop and turn
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000005, 0}, true);   //turn 180 to face goal with four bar
    distanceMove(30, -1);           //move the rest of the distance to the goal
    fourbarpneum->turnOn();         //clamp
    fourbar1->moveTarget(500);      //raise four bar
    fourbar2->moveTarget(500);      //raise four bar
    intake->run(true, false, -150); //start intake
    distanceMove(30, -0.9);         //move the rest of the way towards the ring line
//-----------------------------------------------------------------
//RINGS ON ALLIANCE GOAL
    goal.theta = 180_deg;           //make the turn towards 180 deg
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    distanceMove(24, -1);           //move quickly to the rings
    distanceMove(48, -0.5);         //then move slowly to intake the rings
//-----------------------------------------------------------------
//FIRST NEUTRAL ON PLATFORM

    distanceMove(36, 0.8);          //backwards to orient for the platform
    goal.theta = 90_deg;            //turn towards platform
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    fourbar1->moveTarget(2400);     //four bar up
    fourbar2->moveTarget(2400);
    pros::delay(300);
    distanceMove(12, -0.4);         //move towards platform
    fourbar1->moveTarget(2000);
    fourbar2->moveTarget(2000);
```

```
    fourbarpneum->turnOff();        //release clamp
    distanceMove(12, 0.4);          //back away from seesaw
    fourbar1->moveTarget(0);        //four bar down
    fourbar2->moveTarget(0);

// ------------------------------------------------------------------------
// ALLIANCE GOAL MANIPULATION

    goal.theta = 0_deg;             //turn towards forwards
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000005, 0}, true);
    effectors.runOne(GOAL_LIFT, 1); //drop two bar
    distanceMove(12, 0.8);          //move forwards
    effectors.runOne(GOAL_LIFT, 2); //bring two bar all the way up
    pros::delay(1000);              //wait for that to happen before turning
    goal.theta = 180_deg;           //turn around
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    distanceMove(12, -1);           //Move forwards back towards the alliance goal
    fourbarpneum->turnOn();         //clamp
    fourbar1->moveTarget(2400);     //four bar up
    fourbar2->moveTarget(2400);
    pros::delay(1000);
    goal.theta = 90_deg;            //turn towards platform
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    fourbar1->moveTarget(2000);     //four bar slightly down
    fourbar2->moveTarget(2000);
    distanceMove(12, -0.4);         //move towards platform
    fourbarpneum->turnOff();        //release clamp
    distanceMove(12, 0.4);          //backwards movement
    fourbar1->moveTarget(0);        //four bar down
    fourbar2->moveTarget(0);

//------------------------------------------------------------------------
// SECOND SIDE NEUTRAL MANIPULATION

    goal.theta = 180_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    distanceMove(36, -0.9);         //moving towards the side neutral
    goal.theta = 270_deg;           //turn towards it
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    distanceMove(36, -1);           //move quickly towards it
    fourbarpneum->turnOn();         //clamp
    fourbar1->moveTarget(500);      //four bar up
    fourbar1->moveTarget(500);
    distanceMove(36, 0.9);          //move backwards to retrace steps
    goal.theta = 0_deg;             //turn to 0
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    fourbar1->moveTarget(2100);
    fourbar2->moveTarget(2100);
    distanceMove(36, -1);           //moving forwards to platform
    goal.theta = 90_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    distanceMove(12, -0.4);
    fourbarpneum->turnOff();
    distanceMove(12, 0.4);          //backwards from seesaw
    fourbar1->moveTarget(0);        //four bar down
    fourbar2->moveTarget(0);

```

```
//---------------------------------------------------------------------
//MOVEMENT FOR FIRST RED ALLIANCE GOAL

    goal.theta = 180_deg;           //turning towards back wall
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    distanceMove(48, -1);           //moving towards the turn location
    goal.theta = 48_deg;            //turning towards alliance goal
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    distanceMove(30, -0.6);         //move towards the goal
    fourbarpneum->turnOn();         //clamp
    distanceMove(30, 0.6);          //move away from the goal
    fourbar1->moveTarget(500);      //four bar slightly up
    fourbar2->moveTarget(500);      //four bar slightly up
    pros::delay(500);
    goal.theta = 0_deg;             //turning towards other alliance goal
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
//---------------------------------------------------------------------
//MOVEMENT FOR SECOND RED ALLIANCE GOAL

    distanceMove(80, -1);
    effectors.runOne(GOAL_LIFT, 1); //lower two bar
    pros::delay(1500);
    distanceMove(15, -0.7);         //move the rest of the distance
    pros::delay(500);
    effectors.runOne(GOAL_LIFT, 0); // two bar up
    distanceMove(1, 0.8);
    goal.theta = 270_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);

//---------------------------------------------------------------------
//RINGS
    fourbar1->moveTarget(2400);
    fourbar2->moveTarget(2400);
    distanceMove(96, -0.7);
    goal.theta = 180_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
    distanceMove(3, -0.6);
    fourbar1->moveTarget(100);
    fourbar2->moveTarget(100);
    autobalancer(23.5);
}

void leftskills() {
    OdomState goal = drive->getState();
    setEffectorPositions();
    effectors.runOne(GOAL_LIFT, 1);             //lower goal lift
    pros::delay(1500);
    distanceMove(15, 0.5);                      // move forwards and get goal
    effectors.runOne(GOAL_LIFT, 0);             // raise goal lift
    pros::delay(500);
    goal = drive->getState();
    goal.theta = 270_deg;
    pidMoveTank(goal, {0, 0, 0}, {0.006, 0.000005, 0}, true);   //turn to 90 deg
    distanceMove(24, 0.7);                          //move towards side neutral mogo
    pros::delay(500);
    goal = drive->getState();
    goal.theta = 110_deg;
```

```
1007    pidMoveTank(goal, {0, 0, 0}, {0.005, 0.000005, 0}, true);   //turn to 90 deg
1008    distanceMove(32, -1);                       //move towards side neutral mogo
1009
1010    fourbarpneum->turnOn();                     //clamp
1011    pros::delay(300);
1012    fourbar1->moveTarget(2400);                 //four bar up
1013    fourbar2->moveTarget(2400);
1014
1015    goal = drive->getState();
1016    goal.theta = 90_deg;                    //turn towards center
1017    pidMoveTank(goal, {0, 0, 0}, {0.01, 0.000005, 0}, true);
1018    distanceMove(10, -0.5);                     //move towards ring cross line
1019    goal = drive->getState();
1020    goal.theta = 180_deg;                       //turn towards ring crosses
1021    pidMoveTank(goal, {0, 0, 0}, {0.008, 0.000005, 0}, true);
1022    intake->run(true, false, -150);
1023    // effectors.runOne(GOAL_LIFT, 1);
1024    distanceMove(40, -0.5);                     //move through rings
1025    distanceMove(3, 0.8);                       //move back to platform
1026    goal = drive->getState();
1027  goal.theta = 92_deg;                          //turn towards platform
1028    pidMoveTank(goal, {0, 0, 0}, {0.008, 0.000005, 0}, true);
1029    distanceMove(14, -0.5);
1030    fourbar1->moveTarget(1800);                 //four bar down to balance platform
1031    fourbar2->moveTarget(1800);
1032    pros::delay(500);
1033    fourbarpneum->turnOff();                    //release clamp
1034    pros::delay(500);
1035    fourbar1->moveTarget(2400);
1036    fourbar2->moveTarget(2400);
1037    pros::delay(500);
1038    distanceMove(2, 0.5);                       //move away from platform
1039    goal = drive->getState();
1040    goal.theta = 180_deg;                       //turn towards other side neutral
1041    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000005, 0}, true);
1042    intake->run(true, false, -150);
1043    fourbar1->moveTarget(0);                    //four bar down
1044    fourbar2->moveTarget(0);
1045    distanceMove(35, -0.65);                    //move towards side neutral
1046    goal = drive->getState();
1047    goal.theta = 272_deg;                       //turn towards other side neutral
1048    pidMoveTank(goal, {0, 0, 0}, {0.008, 0.000005, 0}, true);
1049    distanceMove(35, -0.65);
1050    fourbarpneum->turnOn();                     //clamp
1051
1052    distanceMove(30, 0.65);                     //move backwards to line
1053    fourbar1->moveTarget(2400);
1054    fourbar2->moveTarget(2400);
1055    goal = drive->getState();
1056    goal.theta = 359_deg;
1057    pidMoveTank(goal, {0, 0, 0}, {0.008, 0.000005, 0}, true);
1058    distanceMove(36, -0.65);  //move backwards to goal
1059    pros::delay(500);
1060    fourbar1->moveTarget(2150);                 //lower four bar
1061    fourbar2->moveTarget(2150);
1062    goal = drive->getState();
1063    goal.theta = 86_deg;                        //turn towards platform
```

```
1064    pidMoveTank(goal, {0, 0, 0}, {0.0085, 0.000008, 0}, true);
1065    distanceMove(4, -0.8);                  //move forwards to platform
1066
1067    fourbar1->moveTarget(2000);             //lower four bar
1068    fourbar2->moveTarget(2000);
1069    pros::delay(750);
1070
1071    fourbarpneum->turnOff();                //release clamp
1072    pros::delay(500);
1073    fourbar1->moveTarget(2200);             //lower four bar
1074    fourbar2->moveTarget(2200);
1075
1076    pros::delay(500);
1077    distanceMove(3, 0.6);                   //backwards from platform
1078
1079
1080    goal = drive->getState();
1081    goal.theta = 180_deg;
1082    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.00000, 0}, true);
1083    effectors.runOne(GOAL_LIFT, 1);         //drop two bar
1084    fourbar1->moveTarget(0);                //four bar down
1085    fourbar2->moveTarget(0);
1086    distanceMove(8, -0.6);                  //move forwards
1087    pros::delay(500);
1088    effectors.runOne(GOAL_LIFT, 2);         //two bar up
1089    goal.theta = (imu.get_heading() + 180) * 1_deg;                 //turn around to go
back towards the dropped alliance goal
1090    pidMoveTank(goal, {0, 0, 0}, {0.005, 0.00000, 0}, true);
1091    distanceMove(14, -0.8);                 //move towards alliance goal
1092    pros::delay(500);
1093    fourbarpneum->turnOn();                 //clamp
1094    pros::delay(500);
1095    fourbar1->moveTarget(2400);             //four bar up
1096    fourbar2->moveTarget(2400);
1097    distanceMove(6, 0.8);                   //move towards alliance goal
1098    goal = drive->getState();
1099    goal.theta = 87_deg;                    //turn back towards platform with alliance
goal
1100    pidMoveTank(goal, {0, 0, 0}, {0.008, 0.000008, 0}, true);
1101    // fourbar1->moveTarget(2400);          //four bar up
1102    // fourbar2->moveTarget(2400);
1103    pros::delay(500);
1104    fourbar1->moveTarget(2000);
1105    fourbar2->moveTarget(2000);
1106    distanceMove(4, -0.6);                  //forwards to platform
1107    // fourbar1->moveTarget(2000);          //four bar down to deposit
1108    // fourbar2->moveTarget(2000);
1109    fourbarpneum->turnOff();                //release clamp
1110    pros::delay(500);
1111    distanceMove(3, 0.8);                   //backwards from platform
1112    goal.theta = 170_deg;                   //turn towards alliance goal that fell off
the seesaw
1113    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000008, 0}, true);
1114    fourbar1->moveTarget(0);                //lower four bar
1115    fourbar2->moveTarget(0);
1116    distanceMove(37, -1);                   //moving forwards towards the first alliance
goal that came off the seesaw
1117    goal.theta = 50_deg;                    //turn towards alliance goal
```

```
    pros::delay(500);
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000008, 0}, true);
    distanceMove(28, -0.5);                     //move towards alliance goal
    fourbarpneum->turnOn();                     //clamp
    distanceMove(24, 0.7);                      //Move away from alliance goal
    fourbar1->moveTarget(2400);                 //four bar slightly up
    fourbar2->moveTarget(2400);
    // goal.theta = 85_deg;
    // pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000008, 0}, true);
    // distanceMove(6, 1);
    // pros::delay(250);
    goal.theta = 141_deg;                       //turn so that we can move backwards to the
other alliance goal
    pidMoveTank(goal, {0, 0, 0}, {0.005, 0.000008, 0}, true);
    effectors.runOne(GOAL_LIFT, 1);             //drop two bar
    pros::delay(500);
    distanceMove(110, 0.8);                     //move quickly to the alliance goal
    pros::delay(750);
    effectors.runOne(GOAL_LIFT, 0);             //raise two bar
    pros::delay(1000);
    goal.theta = 270_deg;                       //turn to face the rings
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000008, 0}, true);
    fourbar1->moveTarget(2400);                 //four bar up a bit
    fourbar2->moveTarget(2400);
    intake->run(true, false, -150);             //start intake


    speedMove(1000, -0.7);
    distanceMove(1, 0.5);
    //distanceMove(86, -0.6);                   //move along the rings
    goal.theta = 175_deg;                       //turn to climb on the seesaw
    pidMoveTank(goal, {0, 0, 0}, {0.007, 0.000008, 0}, true);
    // fourbar1->moveTarget(2000);              //four bar up to put the seesaw down
    // fourbar2->moveTarget(2000);
    distanceMove(1, -0.5);
    fourbar1->moveTarget(0);                    //four bar down to put the seesaw down
    fourbar2->moveTarget(0);
    pros::delay(2000);
    fourbar1->moveTarget(150);
    fourbar2->moveTarget(150);
    // distanceMove(40, -1.0);                  //autobalance
    autobalancer(24);
    pros::delay(10000);
}

void rightrings() {
    setEffectorPositions();
    fourbar1->moveTarget(500); //lift four bar to intake
    fourbar2->moveTarget(500);
    effectors.runOne(GOAL_LIFT, 1); //lower two bar
    pros::delay(1500);
    distanceMove(20, 1); //pick up alliance
    effectors.runOne(GOAL_LIFT, 0); //raise two bar
    distanceMove(12, -1); //move back
    intake->run(true, false, -175); //turn on intake
    while(1) {
        //oscillate to pick up rings
```

```
        distanceMove(7, -0.5);
        pros::delay(500);
        distanceMove(7, 0.5);
    }


}

void rightMiddle() {
    distanceMove(53, -1);   //move towards side neutral at full speed
    fourbarpneum->turnOn(); //clamp it
    pros::delay(300); //delay 300 ms
    printf("Finished\n");
    distanceMove(53, 1); //move back
}

void left() {
    setEffectorPositions();
    printf("done\n");
    distanceMove(43, -1);   //move towards side neutral at full speed
    fourbarpneum->turnOn(); //clamp it
    pros::delay(200);
    printf("Finished\n");
    distanceMove(10, 1); //move back
    pidTurn(20_deg, {0.02, 0.000008, 0});
    distanceMove(7, 0.6);
}

void middle() {
    setEffectorPositions();
    printf("done\n");
    distanceMove(50, -1);   //move towards side neutral at full speed
    fourbarpneum->turnOn(); //clamp it
    pros::delay(100);
    printf("Finished\n");
    distanceMove(10, 1); //move back
    distanceMove(14, 0.6);
}

void rightthenmiddle() {
    setEffectorPositions();
    printf("done\n");
    distanceMove(39, -1);   //move towards side neutral at full speed
    fourbarpneum->turnOn(); //clamp it
    pros::delay(100);
    printf("Finished\n");
    distanceMove(10, 1); //move back
    pidTurn(270_deg, {0.01, 0.000008, 0});
    fourbarpneum->turnOff();

}

void leftmiddle() {
    setEffectorPositions();
    intake->run(true, false, -180);
    printf("done\n");
    distanceMove(69, -0.7); //move towards side neutral at full speed
```

```
1231    fourbarpneum->turnOn(); //clamp it
1232    pros::delay(100);
1233    printf("Finished\n");
1234    distanceMove(10, 1); //move back
1235    distanceMove(20, 0.6);
1236 }
1237
1238
1239
1240 void esbensOdom() {
1241    // jank, Esben-coded odom that involves taking the current angle at the middle of an
       interval of 50 ms and after 50 ms,
1242    // calculating a linear distance in x and y even if the movement is a curve
1243    // essentially a linear approximation of the movement, 20 times per second
1244
1245    float x = 0;        //variables for tracking
1246    float y = 0;
1247
1248    ADIEncoder righttrack = ADIEncoder('A', 'B', false);        //encoders because i don't
       know how to get values
1249    ADIEncoder lefttrack = ADIEncoder('C', 'D', true);
1250
1251    righttrack.reset(); //reset encoders to zero
1252    lefttrack.reset();
1253
1254    float currangle = imu.get_heading();
1255    float right = 0;
1256    float left = 0;
1257
1258    while (true)
1259    {
1260        pros::delay(25);
1261        currangle = imu.get_heading();
1262        pros::delay(25);
1263        int currright = righttrack.get();
1264        int currleft = lefttrack.get();
1265        int rightencchange = currright - right;
1266        int leftencchange = currleft - left;
1267        float average = (rightencchange + leftencchange)/2 / 360 * 2.75 * PI;
1268        x += average * cos(currangle);
1269        y += average * sin(currangle);
1270        pros::c::lcd_print(0, "OdomX: %f\n", x);        //display on lcd screen
1271        pros::c::lcd_print(1, "OdomY: %f\n", y);
1272        pros::c::lcd_print(2, "OdomH: %d\n", currangle);
1273        right = currright;
1274        left = currleft;
1275    }
1276
1277 }
1278
1279
1280
1281
1282
1283 void autonomous() {
1284
1285    //okapi::Controller controller (okapi::ControllerId::master);
1286    drive->setMode(okapi::AbstractMotor::brakeMode::hold);
```

```
    left();
    //right();
    //middle();
    //left();
    //leftfast();
    //drive->setMode(okapi::AbstractMotor::brakeMode::coast);
}

//experimental pure pursuit handler
void PurePursuitHandler() {
    /*
    std::vector<point> points;
    points.push_back({0, 0, 0, 0, 0});
    points.push_back({0, 24, 0, 0, 0});
    points.push_back({-15, 40, 0, 0, 0});
    PurePursuitPathGen path = PurePursuitPathGen(3, 0.25, 0.75,0.001, points,10.0, 10.0, 2);
    path.interpolate();
  path.smooth();
  path.calc_distances();
    path.calc_curvature();
    path.calc_velocities();
    path.print_path();
    PurePursuitFollower follow = PurePursuitFollower(8);
    follow.read(path);
    std::array<double, 4> vels = {0, 0, 0, 0};
    double x, y, theta;
    do {
        theta = 90-imu.get_heading();
        x = drive->getX();
        y = drive->getY();
        vels = follow.follow(y, x, theta);
        printf("POS: %f %f %f\n", y, x, theta);
        printf("%f %f %f %f\n", vels[0], vels[1], vels[2], vels[3]);
        drive->runTankArcade(vels[0], vels[1]);
        pros::delay(30);

    } while(vels[0] != 0 && vels[1] != 0 &&vels[2] != 0 &&vels[3] != 0);
    drive->runTank(0, 0);
*/
}
```

```cpp
#include "main.h"
#include "pid.h"
#include "drive.h"

//testing push
```

```cpp
#include "button.h"

Button::Button() {
  But but;
  for(okapi::ControllerDigital x : buttonList) {
    but = {false, 0};
    std::pair<okapi::ControllerDigital , But> myBut (x,but);
    buttons.insert(myBut);
  }
}

//handle button counts and states
void Button::handleButtons(Controller controller) {
  for (auto& [key, value]: buttons) {
    if(controller.getDigital(key) && !value.state) {
      value.state = true;
      value.count++;
    }
    else if(!controller.getDigital(key) && value.state) {
      value.state = false;
    }
  }
}

//return counts
int Button::getCount(okapi::ControllerDigital id) {
  return buttons[id].count;
}

//return states
bool Button::getPressed(okapi::ControllerDigital id) {
  return buttons[id].state;
}
```

```cpp
#include "drive.h"


//creates okapi chassis object
Drive::Drive() {
  chassis = ChassisControllerBuilder()
            .withMotors(
              {-TOP_LEFT_MOTOR, -LEFT_MIDDLE_MOTOR, -BOTTOM_LEFT_MOTOR},
              {TOP_RIGHT_MOTOR, RIGHT_MIDDLE_MOTOR, BOTTOM_RIGHT_MOTOR})
            .withDimensions(
              AbstractMotor::gearset::green,
                    ChassisScales({WHEELDIM, WHEELTRACK}, imev5GreenTPR))
            .withSensors(
              ADIEncoder( // left encoder
                  LEFT_TRACKING_WHEEL_TOP,
                  LEFT_TRACKING_WHEEL_BOTTOM

              ),
              ADIEncoder( // right encoder
                  RIGHT_TRACKING_WHEEL_TOP,
                  RIGHT_TRACKING_WHEEL_BOTTOM,
                  true
              )
          )
          .withOdometry(
            ChassisScales({ODOMWHEELDIM, ODOMTRACK}, quadEncoderTPR)
          )

        .buildOdometry();
        speedfactor = 1;
}




//returns X of odometry
double Drive::getX() {
  return chassis->getState().x.convert(inch);
}

//returns Y of odometry
double Drive::getY() {
  return chassis->getState().y.convert(inch);
}

//returns odometry heading
double Drive::getHeading() {
  return chassis->getState().theta.convert(degree);
}


//arcade move function for X drive (old)
void Drive::run(double forward, double strafe, double heading) {
  std::shared_ptr<okapi::XDriveModel> xDrive = std::static_pointer_cast<okapi::XDriveModel>
(chassis->getModel());
  if(forward+strafe+heading>1) {
    forward/=(forward+strafe+heading);
```

```
    strafe/=(forward+strafe+heading);
    heading/=(forward+strafe+heading);
  }
  printf("%f %f %f\n", strafe, forward, heading);
  xDrive->xArcade(strafe, forward, heading);
}

//arcade move function for tank drive
void Drive::runTankArcade(double forward, double turn) {
  chassis->getModel()->arcade(forward, turn);
}

//tank move function for tank drive
void Drive::runTank(double left, double right) {
  chassis->getModel()->tank(left, right);
}


//returns all of odometry state (x, y, and theta)
okapi::OdomState Drive::getState() {
  return chassis->getState();
}

//reverses orientation for driver
void Drive::reverseOrientation(int ori) {
  if(ori%2 == 1) {
    printf("REVERSED\n");
    speedfactor = -1;
  }
  else {
    speedfactor = 1;
  }
}

//sets brake mode of drive mode (if need to coast or hold)
void Drive::setMode(okapi::AbstractMotor::brakeMode brakeMode) {
  chassis->getModel()->setBrakeMode(brakeMode);
}

// double Drive::getEncoder() {
//   return enc.get();
// }
```

```cpp
#include "effectors.h"

//reset encoders for effectors
Effectors::Effectors() {
  for(int i = 0; i < 1; i++) {
    motors[i].getEncoder().reset();
  }
}

//set all encoder positions for two bar
void Effectors::addPosition() {
  //900 difference between upper and lower position
  encPositions[0][0] = 1530; // Two bar upper position
  encPositions[0][1] = 2350; // Two bar lower position
  encPositions[0][2] = 0;     //two bar starting position
  prevCounts[0] = 0;
  prevCounts[1] = 0;
  prevCounts[2] = 0;

}

//handle two bar in opcontrol
void Effectors::step(int buttons[3], double speeds[3]) {

  buttons[0] = buttons[0] % 2;
  buttons[1] = buttons[1] % 2;

    for(int i = 0; i < 1; i++) {
      //printf("Enc position: %f", motors[i].getPosition());
      if(buttons[i] != prevCounts[i]) {
        motors[i].moveAbsolute(encPositions[i][buttons[i]], speeds[i]);
    }

  }
  for(int i = 0; i < 1; i++) {
    prevCounts[i] = buttons[i];
  }
}

//move two bar to preset position
void Effectors::runOne(int lift, int pos) {
  motors[lift].moveAbsolute(encPositions[lift][pos], 200);
}

//move two bar to any position
void Effectors::runOneToPosition(int lift, int pos) {
  motors[lift].moveAbsolute(pos, 200);
}
```

```cpp
#include "intake.h"

Intake::Intake(double port) : m(port)
{
  m.setBrakeMode(okapi::AbstractMotor::brakeMode::hold);
  m.getEncoder().reset();
}

void Intake::addPosition(int pos) {
  encPositions.push_back(pos);
}


//move four bar at full speed to position
void Intake::moveTarget(double enc) {
  m.moveAbsolute(enc, 200);
}


//set limits of four bar
void Intake::setLimits(int upper, int lower) {
  this->upper = upper;
  this->lower = lower;
  limits = true;
}


//run intake at speed while obeying limits
void Intake::run(bool left, bool right, double speed) {
  if(limits && ((m.getPosition()>upper && left) || (m.getPosition()<lower && right))) {
    m.moveVelocity(0);
  }
  else if(left) {
    m.moveVelocity(speed);
  }
  else if(right) {
    m.moveVelocity(-speed);
  }
  else if((!left && !right)) {
    m.moveVelocity(0);
  }
}

void Intake::handle(int count, double speed) {
  if(count%2 == 1 && count!= prevCount) {
    m.moveVelocity(-speed);
  }
  if(count%2 == 0 && count!= prevCount) {
    m.moveVelocity(0);
  }
}

void Intake::stepAbsolute(int count, double speed) {
  printf("count: %d\n", count % encPositions.size());
  if(prevCount != count) {
    double target = encPositions[count % encPositions.size()];
    m.moveAbsolute(target, speed);
```

```
  }
  prevCount = count;
}
```

```cpp
#include "odometry.h"

/*
Odometry::Odometry(ADIEncoder left, ADIEncoder right, ADIEncoder back) {
  this->left = left;
  this->right = right;
  this->back = back;
}

OdomState Odometry::step() {
  int left = left.get();
  int right = right.get();
  int back = back.get();

  int leftchange = left-prevLeft;
  int rightchange = right-prevRight;
  int backchange = back-prevBack;

  double leftDistance = (leftchange / 360) * PI * ODOMWHEELDIM;
  double rightDistance = (rightchange / 360) * PI * ODOMWHEELDIM;
  double backDistance = (backchange / 360) * PI * ODOMWHEELDIM;

  prevLeft = left;
  prevRight = right;
  prevBack = back;

  double currHeading = 90-imu.get_heading();

  double anglediff = currHeading - prevHeading;
  prevHeading = currHeading;

  double localXOffset;
  double localYOffset;

  if (anglediff == 0_deg){
    localXOffset = backDistance;
    localYOffset = rightDistance;
  }
  else {
    localXOffset = 2.0 * sin(angleDiff/2.0) * ( ( backDistance/angleDiff ) + backDistance );
    localYOffset = 2.0 * sin(anglediff/2.0) * ( ( rightDistance/angleDiff ) + rightDistance );
  }
  double averageOrientation = currHeading + (deltaHeading/2.0);
  double r = sqrt( (localXOffset * localXOffset) + (localYOffset * localYOffset) );
  double theta = atan2(localYOffset , localXOffset);
  theta *= (180 / PI);

  theta -= averageOrientation;

  OdomState currState = {prevState.x + (localXOffset*1_in), prevState.y + (localYOffset*1_in),
}

}
*/
```

```cpp
#include "pid.h"

//constructor sets constants

PID::PID(PIDConst constants) {
  this->kp = constants.kp;
  this->ki = constants.ki;
  this->kd = constants.kd;
}

//steps PI algo with error passed in (no D)
double PID::step(double err) {
  totalerr+=err;
  double val = kp*err + totalerr*ki;
  return val;
}
```

```cpp
#include "pneumatics.h"

//constructor
Pneumatics::Pneumatics(uint8_t port) : piston(port) {
    // turnOff();
}

//handle function for buttons
void Pneumatics::handle(int count) {
  if(count%2 == 0 && count!= prevCount) {
    turnOn();
  }
  if(count%2 == 1 && count!= prevCount) {
    turnOff();
  }
}


//actuates pneums down
void Pneumatics::turnOn() {
    printf("pleaseturnon");
    piston.set_value(4095);
    state = true;
}

//actuates pneums up
void Pneumatics::turnOff() {
    printf("isthisrunning\n");
    piston.set_value(0);
    printf("off\n");
    state = false;
}

//testing function
void Pneumatics::onThenOff(int delay) {
    turnOn();
    pros::delay(1000);
    printf("hi");
    turnOff();
}

void Pneumatics::offThenOn(uint32_t delay) {
    turnOff();
    pros::delay(delay);
    turnOn();
}
```

```cpp
#include "../include/PurePursuitFollower.h"
#include "PurePursuitPathGen.h"

#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <algorithm>
#include <math.h>
#include <array>

#define PI 3.14159265

PurePursuitFollower::PurePursuitFollower(double lookahead) {
    this->lookahead = lookahead;
    this->prev_time = timer.millis().convert(second);
}

void PurePursuitFollower::read_from_file(std::string filename) {
    std::ifstream fin;
    fin.open(filename);
    points.clear();
    followPoint temp;
    while (!fin.eof()) {
        fin >> temp.x >> temp.y >> temp.vel;
        points.push_back(temp);
    }
}

void PurePursuitFollower::read(PurePursuitPathGen obj) {
    std::vector<point> temppoints;
    temppoints = obj.get_points();
    followPoint temp;
    printf("READING\n");
    for(int i = 0; i < temppoints.size(); i++) {
        temp.x = temppoints[i].x;
        temp.y = temppoints[i].y;
        temp.vel = temppoints[i].vel;
        points.push_back(temp);
    }
    for(followPoint x: points) {
        printf("%f %f %f\n", x.x, x.y, x.vel);
    }
}


void PurePursuitFollower::calc_closest_point(double x, double y) {
    double min = 1E7;
    for (int i = last_closest_point; i < points.size(); i++) {
        double dist = sqrt(((x - points[i].x) * (x - points[i].x)) + ((y - points[i].y) * (y - points[i].y)));
        if (dist < min) {
            min = dist;
            last_closest_point = i;
            closest_point = points[i];
        }
    }
```

```
}

void PurePursuitFollower::calc_lookahead(double x, double y) {
    std::pair<double, double> d, f;
    double a, b, c, discriminant, t1, t2;
    for (int i = last_closest_point + 1; i < points.size(); i++) {
        d.first = points[i].x - points[i - 1].x;
        d.second = points[i].y - points[i - 1].y;
        f.first = points[i].x - x;
        f.second = points[i].y - y;
        a = d.first * d.first + d.second * d.second;
        b = 2 * (f.first * d.first + f.second * d.second);
        c = (f.first * f.first + f.second * f.second) - lookahead * lookahead;
        discriminant = b * b - (4 * a * c);
        if (discriminant >= 0) {
            discriminant = sqrt(discriminant);
            t1 = (-b - discriminant) / (2 * a);
            t2 = (-b + discriminant) / (2 * a);
            if (t1 >= 0 && t1 <= 1 && t1 + i - 1 > last_fractional_index) {
                lookahead_point.first = points[i - 1].x + (t1 * d.first);
                lookahead_point.second = points[i - 1].y + (t1 * d.second);
                last_lookahead_point = lookahead_point;
                break;
            }
            if (t2 >= 0 && t2 <= 1 && t2 + i - 1 > last_fractional_index) {
                lookahead_point.first = points[i - 1].x + (t2 * d.first);
                lookahead_point.second = points[i - 1].y + (t2 * d.second);
                last_lookahead_point = lookahead_point;
                break;
            }
        }
    }
    lookahead_point = last_lookahead_point;
    last_lookahead_point = lookahead_point;
}

void PurePursuitFollower::calc_curvature_at_point(double x, double y, double theta) {
    double xtemp;
    double a, b, c;
    a = -tan((theta));
    b = 1;
    c = (tan((theta)) * x) - y;
    double temp = (sin((theta)) * (lookahead_point.first - x)) - (cos((theta)) *
(lookahead_point.second - y));
    int sign = (temp > 0) ? 1 : ((temp < 0) ? -1 : 0);
    xtemp = abs((a * lookahead_point.first) + (b * lookahead_point.second) + c) / sqrt((a * a)
+ (b * b));
    this->curvature = ((2 * xtemp) / (lookahead * lookahead));
    this->curvature *= sign;
}

std::array<double, 4> PurePursuitFollower::follow_sim(double x, double y, double theta) {
    calc_closest_point(x, y);
    calc_lookahead(x, y);
    calc_curvature_at_point(x, y, theta);
    std::array<double, 4> vels;
    if (closest_point.x == points[points.size() - 1].x && closest_point.y ==
points[points.size() - 1].y) {
```

```
        vels[0] = 0;
        vels[1] = 0;
        vels[2] = 0;
        vels[3] = 0;
        return vels;
    }
    vels[0] = closest_point.vel;
    vels[1] = vels[0] * curvature;
    vels[2] = 0;
    vels[3] = 0;
    return vels;
}

std::array<double, 4> PurePursuitFollower::follow(double x, double y, double theta) {
    calc_closest_point(x, y);
    calc_lookahead(x, y);
    calc_curvature_at_point(x, y, theta);
    std::array<double, 4> vels;
    if (closest_point.x == points[points.size() - 1].x && closest_point.y ==
points[points.size() - 1].y) {
        vels[0] = 0;
        vels[1] = 0;
        vels[2] = 0;
        vels[3] = 0;
        return vels;
    }
    double time = (timer.millis().convert(second)-prev_time);
    double vel, ang;
    vel = (closest_point.vel);
    vel = prev_vel+(std::clamp(vel-prev_vel, -(time*max_accel), (time*max_accel)));
    printf("vel: %f curvature: %f\n", vel, curvature);
    vel = vel/10;
    ang = vel*curvature;
    ang = ang/(10/(2*PI));
    vels[0] = vel;
    vels[1] = ang;
    vels[2] = vel;
    vels[3] = ang;
    prev_time = timer.millis().convert(second);
    prev_vel = vel;
    return vels;
}
```

```cpp
#include "PurePursuitPathGen.h"
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <algorithm>
#include <math.h>

PurePursuitPathGen::PurePursuitPathGen(double spacing, double a, double b, double tolerance, std::vector<point> points, double max_vel, double max_accel, int k) {
    this->spacing = spacing;
    this->a = a;
    this->b = b;
    this->tolerance = tolerance;
    this->max_vel = max_vel;
    this->max_accel = max_accel;
    this->k = k;
    this->initial_points = points;
}

void PurePursuitPathGen::interpolate() {
    point vec;
    int mag;
    int num_points;
    final_points.push_back(initial_points[0]);
    for(int i = 1; i < initial_points.size(); i++) {


        vec.x = initial_points[i].x-initial_points[i-1].x;
        vec.y = initial_points[i].y-initial_points[i-1].y;

        mag = sqrt((vec.x*vec.x)+(vec.y*vec.y));

        num_points = ceil(mag/spacing);

        vec.x = (vec.x/mag)*spacing;
        vec.y = (vec.y/mag)*spacing;
        point new_vec;
        for(int j = 1; j < num_points; j++) {
            new_vec.x = (initial_points[i-1].x+(vec.x*j));
            new_vec.y = (initial_points[i-1].y+(vec.y*j));
            final_points.push_back(new_vec);
        }
        final_points.push_back(initial_points[i]);
    }
}
void PurePursuitPathGen::calc_distances() {
    final_points[0].distance = 0;
    for(int i = 1; i < final_points.size(); i++) {
        final_points[i].distance = final_points[i-1].distance + sqrt(pow(final_points[i].x-final_points[i-1].x, 2)+pow(final_points[i].y-final_points[i-1].y, 2));
    }
}
void PurePursuitPathGen::calc_curvature() {
    final_points[0].curve = 0;
    double k1, k2, center1, center2, r, x1, x2, x3, y1, y2, y3;
    for(int i = 1; i < final_points.size()-1; i++) {
```

```
        x1 = final_points[i].x+0.001;
        x2 = final_points[i-1].x;
        x3 = final_points[i+1].x;

        y1 = final_points[i].y;
        y2 = final_points[i-1].y;
        y3 = final_points[i+1].y;

        k1=0.5*((x1*x1)+(y2*y2)-(x2*x2)-(y2*y2))/(x1-x2);
        k2 = (y1-y2)/(x1-x2);
        center2 = 0.5*((x2*x2)-(2*x2*k1)+(y2*y2)-(x3*x3)+(2*x3*k1)-(y3*y3))/(((x3*k2)-y3+y2-(x2*k2)));
        center1 = k1-k2*center2;
        r = sqrt((x1-center1)*(x1-center1) + (y1-center1)*(y1-center1));
        final_points[i].curve = 1/r;
    }
    final_points[final_points.size()-1].curve = 0;
}
void PurePursuitPathGen::smooth() {
    std::vector<point> copy;
    copy = final_points;
    double change = tolerance;
    while(change>=tolerance) {
        change = 0.0;
        for(int i = 1; i < final_points.size()-1; i++) {
            double aux = copy[i].x;
            copy[i].x += a*(final_points[i].x-copy[i].x) + b*(copy[i-1].x + copy[i+1].x-(2.0*(copy[i].x)));
            change+=abs(aux-copy[i].x);
            aux = copy[i].y;
            copy[i].y += a*(final_points[i].y-copy[i].y) + b*(copy[i-1].y + copy[i+1].y-(2.0*(copy[i].y)));
            change+=abs(aux-copy[i].y);
        }
    }
    final_points = copy;
}

void PurePursuitPathGen::calc_velocities() {
    for(int i = 0; i < final_points.size(); i++) {
        //std::cout << "Max vels" << k/final_points[i].curve << "\n";
        final_points[i].vel = std::min(max_vel, k/final_points[i].curve);
    }
    final_points[final_points.size()-1].vel = 0;
    for(int i = final_points.size()-2; i >=0; i--) {
        final_points[i].vel = std::min(final_points[i].vel, sqrt(pow(final_points[i+1].vel, 2)+2*max_accel*(final_points[i+1].distance-final_points[i].distance)));
    }
}

void PurePursuitPathGen::write_to_file() {
    std::ofstream fout;
    fout.open("path.txt");
    for(int i = 0; i < final_points.size(); i++) {
        fout << final_points[i].x << " "<< final_points[i].y << " " << final_points[i].vel << "\n";
    }
    fout.close();
```

```
}
void PurePursuitPathGen::print_path() {
    printf("INITIAL\n");
    for(int i = 0; i < initial_points.size(); i++) {
        printf("%f %f\n", initial_points[i].x, initial_points[i].y);
    }
    printf("FINAL\n");
    for(int i = 0; i < final_points.size(); i++) {
        printf("%f %f\n", final_points[i].x, final_points[i].y);
    }
    printf("DISTANCE\n");
    for(int i = 0; i < final_points.size(); i++) {
      printf("%f\n", final_points[i].distance);
    }
    printf("CURVATURE\n");
    for(int i = 0; i < final_points.size(); i++) {
        printf("%f\n", final_points[i].curve);
    }
    printf("VELOCITIES\n");
    for(int i = 0; i < final_points.size(); i++) {
        printf("%f\n", final_points[i].vel);
    }

}

std::vector<point> PurePursuitPathGen::get_points() {
  return final_points;
}
```