The 7 Dorks
333A

2020-2021 VRC Annotated Programming Skills Challenge

Contents

1 Header Files

1.1 include/Auton.hpp

```cpp
/**
 * Auton.hpp
 *
 * This contains the declaration of the Auton struct,
 * which is responsible for reading the sd card to determine
 * which auton is selected, and running the correct auton.
 */
#pragma once // makes sure the file is only included once
#include "main.h" // gives access to dependancies from other files
class Auton
{
  public:
    // when making autons, you must add the text to the dropdown in DisplayControl.cpp, a new value
    // to this enum, and a new case is the switch in Auton.cpp
    enum class Autons // all possible autons
    {
        none,
        test,
        prog
    } static auton; // selected auton

    static void readSettings(); // read the sd card to set the settings

    static void runAuton(); // runs the selected auton

  private:
    static void auton_task_func(void *); // separate thread for running the auton, in case a
                                         // particular auton needs control over it's thread.
};
```

1.2 include/main.h

```
/**
 * \file main.h
 *
 * Contains common definitions and header files used throughout your PROS
 * project.
 *
 * Copyright (c) 2017-2020, Purdue University ACM SIGBots.
 * All rights reserved.
 *
 * This Source Code Form is subject to the terms of the Mozilla Public
 * License, v. 2.0. If a copy of the MPL was not distributed with this
 * file, You can obtain one at http://mozilla.org/MPL/2.0/.
 */

#ifndef _PROS_MAIN_H_
#define _PROS_MAIN_H_

/**
 * If defined, some commonly used enums will have preprocessor macros which give
 * a shorter, more convenient naming pattern. If this isn't desired, simply
 * comment the following line out.
 *
 * For instance, E_CONTROLLER_MASTER has a shorter name: CONTROLLER_MASTER.
 * E_CONTROLLER_MASTER is pedantically correct within the PROS styleguide, but
 * not convienent for most student programmers.
 */
#define PROS_USE_SIMPLE_NAMES

/**
 * If defined, C++ literals will be available for use. All literals are in the
 * pros::literals namespace.
 *
 * For instance, you can do `4_mtr = 50` to set motor 4's target velocity to 50
 */
#define PROS_USE_LITERALS

#include "api.h"

/**
 * You should add more #includes here
 */
#include "okapi/api.hpp"
#include "pros/apix.h"
//#include "pros/api_legacy.h"

/**
 * If you find doing pros::Motor() to be tedious and you'd prefer just to do
 * Motor, you can use the namespace with the following commented out line.
 *
 * IMPORTANT: Only the okapi or pros namespace may be used, not both
 * concurrently! The okapi namespace will export all symbols inside the pros
 * namespace.
```

```
 */
// using namespace pros;
// using namespace pros::literals;
using namespace okapi;

#include "movement/paths/ProfileStep.hpp"
#include "movement/paths/Trajectory.hpp"

#include "gui/odomDebug.hpp"
#include "util/CustomOdometry.hpp"
#include "gui/DisplayControl.hpp"
#include "util/util.hpp"
#include "util/definitions.hpp"

#include "movement/paths/SimplePath.hpp"

#include "movement/AsyncAction.hpp"
#include "movement/Drivetrain.hpp"

#include "stateMachines/VStateMachine.hpp"
#include "stateMachines/DrivetrainStateMachine.hpp"
#include "stateMachines/BallControlStateMachine.hpp"

#include "Auton.hpp"

/**
 * Prototypes for the competition control tasks are redefined here to ensure
 * that they can be called from user code (i.e. calling autonomous from a
 * button press in opcontrol() for testing purposes).
 */
#ifdef __cplusplus
extern "C"
{
#endif
    void autonomous(void);
    void initialize(void);
    void disabled(void);
    void competition_initialize(void);
    void opcontrol(void);
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus
/**
 * You can add C++-only headers here
 */
//#include <iostream>
#endif

#endif // _PROS_MAIN_H_
```

1.3 include/gui/DisplayControl.hpp

```cpp
/**
 * DisplayControl.hpp
 *
 * This file contains the declaration of the DisplayControl class.
 * DisplayControl is the class that handles the organization of the
 * LittleV Graphics Library (LVGL) objects on the screen of the brain.
 */
#pragma once // makes sure the file is only included once
#include "main.h" // gives access to objects not declared here (LVGL objects)
class DisplayControl
{

    /* ---------------------- Tabview Elements -------------------- */
    static lv_obj_t * mtabview; // contains the whole tabview

    static lv_obj_t * mtabview_odom; // tabview page with odom debugger
    OdomDebug modom; // odom debugger

    static lv_obj_t * mtabview_auton; // tab for setting auton
    static lv_obj_t * mtabview_auton_dropdown; // autons to choose from
    static lv_res_t tabview_auton_dropdown_action(lv_obj_t * idropdown); // event
    ↪  handler

    static lv_obj_t * mtabview_graph; // tabview page with graph
    static lv_obj_t * mtabview_graph_chart; // graph
    static lv_chart_series_t * mtabview_graph_chart_series_0; // chart series...
    static lv_chart_series_t * mtabview_graph_chart_series_1;
    static lv_chart_series_t * mtabview_graph_chart_series_2;
    static lv_chart_series_t * mtabview_graph_chart_series_3;
    static lv_chart_series_t * mtabview_graph_chart_series_4;
    static lv_chart_series_t * mtabview_graph_chart_series_5;
    static lv_chart_series_t * mtabview_graph_chart_series_6;

    static lv_obj_t * mtabview_misc; // extra tabview page for anything
    static lv_obj_t * mtabview_misc_container; // container on the misc page to hold
    ↪  elements
    static lv_obj_t * mtabview_misc_label; // text box on misc page
    static lv_obj_t * mtabview_misc_label_2; // second text box on misc page

    /* ------------------------- Styles ------------------------- */
    static lv_style_t mstyle_tabview_indic; // for page indicator line
    static lv_style_t mstyle_tabview_btn; // for page header
    static lv_style_t mstyle_tabview_btn_tgl; // for selected page header
    static lv_style_t mstyle_tabview_btn_pr; // for pressed page header
    static lv_style_t mstyle_tabview_container; // for page background
    static lv_style_t mstyle_text; // for text

  public:
    DisplayControl(); // constructor that sets everything up, like styles and
    ↪  positioning

    void setOdomData(); // updates the values for OdomDebug
```

```
50      void setAutonDropdown(); // updates the dropdown to match sd card at the start of
        ↪  the program,
51                               // to ensure the auton set on the sd card is always the
                                 ↪  same as the
52                               // auton displayed on the dropdown.
53
54      void setChartData(int iseries,
55                        double ivalue); // sets the value of one of the series in the
                          ↪  chart
56
57      void setMiscData(int ilabel,
58                       std::string itext); // sets the information displayed in the misc
                         ↪  tab
59  };
60
61  namespace def
62  {
63  extern DisplayControl
64      display; // declares the display object as extern, to make sure it only gets
        ↪  constructed once
65  }
```

1.4 include/gui/odomDebug.hpp

```cpp
/**
 * odomDebug.hpp
 *
 * The contents of this file were not written by any members of 333A*.
 * This is code from the publicly availible GitHub repository, odomDebug
 * by theol0403, found here: https://github.com/theol0403/odomDebug.
 *
 * The OdomDebug class is used for the tab on the screen of the brain
 * that shows the odometry position of the robot in the form of number
 * values and a moving circle on a picture of the field representing the
 * robot.
 *
 *    * slight modifications were made to make it work with the display
 */
#pragma once
#include "main.h"

class OdomDebug
{

  public:
    /**
     * Contains robot state - x, y, theta
     * Can be initialized using QUnits or doubles
     */
    struct state_t
    {
        QLength x{0.0};
        QLength y{0.0};
        QAngle theta{0.0};

        /**
         * @param ix QLength
         * @param iy QLength
         * @param itheta QAngle
         */
        state_t(QLength ix, QLength iy, QAngle itheta);

        /**
         * @param ix inches
         * @param iy inches
         * @param itheta radians
         */
        state_t(double ix, double iy, double itheta);
    };

    /**
     * Contains encoder information - left, right, middle(optional)
     * Can be initialized using three or two sensors
     */
    struct sensors_t
    {
```

```
        double left{0.0};
        double right{0.0};
        double middle{0.0};

        /**
         * @param ileft the left encoder value
         * @param iright the right encoder value
         */
        sensors_t(double ileft, double iright);

        /**
         * @param ileft the left encoder value
         * @param iright the right encoder value
         * @param imiddle imiddle the middle encoder value
         */
        sensors_t(double ileft, double iright, double imiddle);

      private:
        bool hasMiddle{false};
        friend class OdomDebug;
    };

    /**
     * Constructs the OdomDebug object.
     * @param parent the lvgl parent, color is inherited
     */
    OdomDebug(lv_obj_t * parent);

    /**
     * Constructs the OdomDebug object.
     * @param parent the lvgl parent
     * @param mainColor The main color for the display
     */
    OdomDebug(lv_obj_t * parent, lv_color_t mainColor);

    ~OdomDebug();

    /**
     * Sets the function to be called when a tile is pressed
     * @param callback a function that sets the odometry state
     */
    void setStateCallback(std::function<void(state_t state)> callback);

    /**
     * Sets the function to be called when the reset button is pressed
     * @param callback a function that resets the odometry and sensors
     */
    void setResetCallback(std::function<void()> callback);

    /**
     * Sets the position of the robot in QUnits and puts the sensor data on the
     * display
     * @param state   robot state - x, y, theta
     * @param sensors encoder information - left, right, middle (optional)
```

```
107         */
108         void setData(state_t state, sensors_t sensors);
109 
110     private:
111         // parent container
112         lv_obj_t * container = nullptr;
113         lv_style_t cStyle;
114 
115         // field
116         lv_style_t fStyle;
117         double fieldDim = 0; // width and height of field container
118 
119         // tile styles
120         lv_style_t grey;
121         lv_style_t red;
122         lv_style_t blue;
123 
124         // robot point
125         lv_obj_t * led = nullptr;
126         lv_style_t ledStyle;
127 
128         // robot line
129         lv_obj_t * line = nullptr;
130         lv_style_t lineStyle;
131         std::vector<lv_point_t> linePoints = {{0, 0}, {0, 0}}; // line positions
132         int lineWidth = 0;
133         int lineLength = 0;
134 
135         // status label
136         lv_obj_t * statusLabel = nullptr;
137         lv_style_t textStyle;
138 
139         // reset button styles
140         lv_style_t resetRel;
141         lv_style_t resetPr;
142 
143         // external callbacks to interface with odometry
144         std::function<void(state_t state)> stateFnc = nullptr;
145         std::function<void()> resetFnc = nullptr;
146 
147         static lv_res_t tileAction(lv_obj_t *); // action when tile is pressed
148         static lv_res_t resetAction(lv_obj_t *); // action when reset button is pressed
149 };
```

1.5 include/movement/AsyncAction.hpp

```cpp
/**
 * AsyncAction.hpp
 *
 * This file contains the definition of the AsyncAction struct.
 * AsyncActions are objects that have an action (maction) and
 * a certain error where the action should be executed (merror).
 * It is used by motions in the Drivetrain class to run asynchronus
 * actions at a certain distance from the target.
 */
#pragma once // makes sure the file is only included once
#include "main.h" // gives access to objects declared elsewhere
struct AsyncAction
{
    AsyncAction(double ierror, std::function<void()> iaction)
        : merror(ierror), maction(iaction) // constructor
    {
    }

    double merror; // error value at which the loop will execute the action
    std::function<void()> maction; // action to execute
};
```

1.6 include/movement/Drivetrain.hpp

```cpp
/**
 * Drivetrain.hpp
 *
 * This file contains the declaration of the Drivetrain class.
 * The Drivetrain class handles almost everthing relating to the
 * drivetrain: motor control, settings (like max speed), basic
 * movement methods (like tank or arcade), more advanced movement
 * methods (like PID to point, path following, and motion
 * profiling), and more.
 */
#pragma once // makes sure the file is only included once
#include "main.h" // gives access to dependencies from other files
class Drivetrain // creates the class for the drivetrain
{
  private:
    /* ---------------------- Motor References --------------------- */
    static Motor & mmtrLeftFront;
    static Motor & mmtrRightFront;
    static Motor & mmtrRightBack;
    static Motor & mmtrLeftBack;

    /* ------------------------- Chassis ------------------------- */
    static std::shared_ptr<ChassisController>
        mchassis; // chassis object for using Pathfilder through okapi

  protected:
    /* ------------------------- Settings ------------------------ */
    static double mmaxSpeed;
    static bool menabled;

    /* -------------------- SimpleFollow Data -------------------- */
    static double mlastLookIndex; // index of the last lookahead point
    static double
        mlastPartialIndex; // fractional index of where the last lookahead point was on
        ↪  the segment

    /* --------------------- Odometry Accessors ------------------- */
    static OdomState getState(); // get position as OdomState
    static QLength getXPos();
    static QLength getYPos();
    static QAngle getTheta();
    static ExtendedPoint getPoint(); // get position as ExtendedPoint

    /* ------------------------- Helpers ------------------------- */
    static QAngle
    angleToPoint(const Point & itargetPoint); // calculates the field centric direction
    ↪  to the
                                              // itargetPoint from the robot's current
                                              ↪  position
    static std::optional<double> findIntersection(
        ExtendedPoint istart, ExtendedPoint iend,
```

```
49          const double & ilookDistIn); // looks for interections between the line segment
            ↪ created by
50                                       // the two points (istart and iend), and the
                                         ↪ circle around the
51                                       // robot with radius ilookDistIn (lookahead circle)
52      static ExtendedPoint
53      findLookahead(SimplePath ipath,
54                    const double & mlookDistIn); // looks for the intersection point
                      ↪ between the
55                                                 // lookahead circle and the SimplePath,
                                                   ↪ ipath
56
57    public:
58      /* ---------------------- Getters/Setters ---------------------- */
59      static double getMaxSpeed();
60      static void setMaxSpeed(const double imaxSpeed);
61
62      static bool isEnabled();
63      static void enable(); // allows movements to be startable
64      static void disable(); // stops active movements
65
66      static void
67      checkNextAsync(const double & ierror,
68                     std::vector<AsyncAction> &
69                         iactions); // checks if the next AsyncAction should execute, and
                           ↪ executes it
70                                    // (and removes it from the list) if it should
71
72      /* ---------------------- Basic Movement ---------------------- */
73      static void moveIndependant(
74          double ileftFront, double irightFront, double irightBack, double ileftBack,
75          const bool idesaturate = true); // moves each motor {lf, rf, rb, lb} in range
            ↪ [-1,1]
76      static void
77      moveTank(const double ileft, const double iright,
78               const bool idesaturate =
79                   true); // spins the left side and right side motors at certian speeds
                    ↪ [-1,1]
80      static void moveArcade(
81          const double iforward, const double istrafe, const double iturn,
82          const bool idesaturate = true); // moves the robot with arcade-style inputs
            ↪ (range[-1,1])
83
84      /* ------------------ Intermediate Movement ------------------ */
85      static void moveInDirection(
86          QAngle idirection, const bool ifieldCentric = false, double imagnitude = 1,
87          double itheta = 0,
88          const bool idesaturate = true); // moves the robot with a certain speed in a
            ↪ certain
89                                          // direction, while turning a certain amount
90
91      /* ------------------ Move to Point Methods ------------------ */
92      static void strafeToPoint(
93          ExtendedPoint iPoint, std::vector<AsyncAction> iactions = {},
```

```
94          PID imagnitudePID = PID(0.4, 0.005, 2.6, 0.5, 0.25, 0.05, 10_ms),
95          PID iturnPID = PID(0.028, 0.0, 0.08, 0.0, 1.5, 0.1, 10_ms),
96          Slew imagnitudeSlew = Slew(1000, 1000),
97          Slew iturnSlew =
98              Slew(1000, 1000)); // drives in a stright line to the point while turning
                ↪  using set
99                                 // PID/Slew gains, and executing the AsyncActions at the
                                   ↪  right times
100
101     static void straightToPoint(
102         ExtendedPoint itarget, std::vector<AsyncAction> iactions = {}, QLength
            ↪  inoTurnRange = 3_in,
103         double iturnWeight = 1.7, PID imagnitudePID = PID(0.4, 0.005, 2.6, 0.5, 0.25,
            ↪  0.05, 10_ms),
104         PID iturnPID = PID(0.028, 0.0, 0.08, 0.0, 1.5, 0.1, 10_ms),
105         Slew imagnitudeSlew = Slew(1000, 1000),
106         Slew iturnSlew = Slew(1000,
107                               1000)); // drives to the point without strafing using set
                                        ↪  PID/Slew
108                                       // gains, and executing the AsyncActions at the
                                          ↪  right times
109
110     static void arcStraightToPoint(
111         ExtendedPoint itarget, std::vector<AsyncAction> iactions = {}, double
            ↪  iweightModifier = 10,
112         QLength inoTurnRange = 3_in,
113         PID imagnitudePID = PID(0.4, 0.005, 2.6, 0.5, 0.25, 0.05, 10_ms),
114         PID iturnPID = PID(0.028, 0.0, 0.08, 0.0, 1.5, 0.1, 10_ms),
115         Slew imagnitudeSlew = Slew(1000, 1000),
116         Slew iturnSlew = Slew(
117             1000,
118             1000)); // drive in an "arc" (doesn't follow a path, just approximates an
                ↪  arc) using set
119                     // PID/Slew gains, and executing the AsyncActions at the right times
120
121     /* ------------------ Path Following Methods ----------------- */
122     static void simpleFollow(
123         SimplePath ipath, QLength ilookDist = 6_in, std::vector<AsyncAction> iactions =
            ↪  {},
124         PID imagnitudePID = PID(0.4, 0.005, 2.6, 0.5, 0.25, 0.05, 10_ms),
125         PID iturnPID = PID(0.028, 0.0, 0.08, 0.0, 1.5, 0.1, 10_ms),
126         Slew imagnitudeSlew = Slew(1000, 1000),
127         Slew iturnSlew = Slew(1000,
128                               1000)); // follows the path, ipath using set lookahead
                                        ↪  distance
129                                       // (ilookDist) and PID/Slew gains while executing
                                          ↪  the
130                                       // AsyncActions at the right times (only on the
                                          ↪  last segment)
131
132     /* --------------------- Motion Profiling -------------------- */
133     static std::shared_ptr<AsyncMotionProfileController>
134         mprofiler; // okapi motion profile controller
135     static void generatePathToPoint(
```

```
136         PathfinderPoint ipoint,
137         const std::string & iname); // use Pathfinder through okapi to make a motion
            ↪ profile
138     static void followPathfinder(const std::string & iname, bool ibackwards = false,
139                                  bool imirrored = false); // follow Pathfinder path
                                     ↪ through okapi
140     static void followTraj(Trajectory & itraj); // follow trajectory loaded from sd card
141 };
142
143 namespace def
144 {
145 extern Drivetrain drivetrain; // declares the drivetrain object as extern, to make sure
    ↪ it only gets
146                               // constructed once
147 }
```

1.7 include/movement/paths/ProfileStep.hpp

```cpp
/**
 * ProfileStep.hpp
 *
 * ProfileStep is used for organizing the information parsed from
 * motion profiles stored on the sd card, calculated by the
 * publically availible GitHub repository, TrajectoryLib by
 * Team254 (FRC Team 254, The Cheesy Poofs), found here:
 * https://github.com/Team254/TrajectoryLib. The trajectories
 * are calculated on a computer, and stored on the sd card for
 * the robot to use. Each time step of the profile is read from
 * the sd card, and stored in an instance of ProfileStep by the
 * Trajectory class.
 */
#pragma once
#include "main.h"
struct ProfileStep
{
    float pos{0.000};
    float vel{0.000};
    float acc{0.000};
    float jerk{0.000};
    float heading{0.000};
    float dt{0.000};
    float x{0.000};
    float y{0.000};

    const std::string getString(); // return the contents of the ProfileStep for
    ↪  testing purposes
};
```

1.8 include/movement/paths/SimplePath.hpp

```cpp
/**
 * SimplePath.hpp
 *
 * SimplePath is a simple struct that has a list of points
 * on a path represented by ExtendedPoints in a std::vector.
 * This is used for path following by the Drivetrain class.
 */
#pragma once // makes sure the file is only included once
#include "main.h" // gives access to objects declared elsewhere (std::vector and ExtendedPoint)
struct SimplePath
{
    std::vector<ExtendedPoint> mpoints;

    ExtendedPoint at(size_t iindex); // gets the point at iindex
    ExtendedPoint last(); // gets the last point
    int size(); // number of points
};
```

1.9 include/movement/paths/Trajectory.hpp

```cpp
/**
 * Trajectory.hpp
 *
 * This file contains the declaration of the Trajectory class.
 * The Trajectory class reads and stores motion profile information
 * from the sd card. Motion profiles stored on the sd card are
 * calculated by the publically availible GitHub repository,
 * TrajectoryLib by Team254 (FRC Team 254, The Cheesy Poofs),
 * found here: https://github.com/Team254/TrajectoryLib. The
 * trajectories are calculated on a computer, and stored on the
 * sd card for the robot to use. Each time step of the profile is
 * read from the sd card, and stored in an instance of ProfileStep
 * by the Trajectory class.
 *
 * The paths are intended to be executed by the Drivetrain class.
 */
#pragma once
#include "main.h"
class Trajectory
{
    double mkP, mkD, mkV, mkA; // control constants
    int mstepNumber; // index of current step
    double mlastErrorL, mlastErrorR; // old error values

    std::string mname; // name of the movement from the top of the file
    int mlength; // number of steps for each side to execute
    ProfileStep * mleftSteps; // steps for the left side
    ProfileStep * mrightSteps; // steps for the right side

  public:
    Trajectory(const char * ifileName, double ikP = 0.0, double ikD = 0.0, double ikV =
    ↪  0.025,
               double ikA = 0.0); // constructor that specifies control constants
    ~Trajectory(); // destructor that handles dynamically allocated arrays to prevent
    ↪  memory issues

    std::string getName();
    int getLength();
    void reset(); // sets mstepNumber back to 0
    std::pair<ProfileStep, ProfileStep>
    getStep(int istepNumber); // get the left and right values at a certain step
    void setGains(const double ikP, const double ikD, const double ikV, const double
    ↪  ikA);
    bool isDone(); // checks to see if all of the steps have been executed

    std::pair<double, double>
    iterate(const double ileftDistSoFar,
            const double irightDistSoFar); // calculate the motor vales at the next step
};

namespace def
{
```

```
50 extern Trajectory traj_test;
51 extern Trajectory TestSpline;
52 } // namespace def
```

1.10 include/stateMachines/BallControlStateMachine.hpp

```cpp
/**
 * BallControlStateMachine.hpp
 *
 * This file contains the definitions of the BallControlStateMachine class.
 * BallControlStateMachine inherits from VStateMachine, and
 * it is responsible for controlling all of the ball manipulators
 * (intake, indexer, filter, and flywheel).
 *
 * The intake, indexer, and flywheel all have their own mini
 * state machine in structs all contained in
 * BallControlStateMachine. BallControlStateMachine puts them
 * all together to make them function cohesively
 */
#pragma once // makes sure the file is only included once
#include "main.h" // gives access to dependencies from other files
class BallControlStateMachine : public VStateMachine // state machine to represent the ball
                                                     // controlers (intake/indexer/flywheel)
{
  public:
    BallControlStateMachine(); // constructor to set defaults

    void controlState() override; // sets the state based on inputs from the controller
    void update() override; // controls the robot based on the state

    void itIn(); // spins the intakes in
    void itOut(); // spins the intakes out
    void itOff(); // stops the intakes
    void ixUp(); // spins the indexer up
    void ixDown(); // spins the indexer down
    void ixOff(); // stops the indexer
    void fwShoot(); // shoots the flywheel
    void fwFilter(); // spins the flywheel backwards
    void fwOff(); // stops the flywheel

    void itInFor(double imilliseconds); // spins the intakes for specified number of miliseconds
    void ixUpFor(double imilliseconds); // spins the indexer for specified number of miliseconds
    void shoot(int ims = 300); // shoots a ball

    bool controlEnabled; // decides if the state machine should pay attention to the controller

  private:
    /* ------------------------ Controls ----------------------- */
    ControllerButton & mbtnIn;
    ControllerButton & mbtnOut;
    ControllerButton & mbtnShoot;
    ControllerButton & mbtnFilter;

```

```
48      /* ---------------------- Nested Classes --------------------- */
49      struct MIntake // controls the intakes
50      {
51          MIntake(); // constructor to set defaults
52
53          enum MStates // enumeration to organize all possible states
54          {
55              off,
56              in,
57              out
58          };
59
60          void update(); // updates the subsystem based on the state
61
62          /* -------------------------- State -------------------------- */
63          MStates mstate;
64
65          /* -------------------------- Other -------------------------- */
66          MotorGroup mmotors;
67      } mintake;
68
69      struct MIndexer // controls the indexer
70      {
71          MIndexer(); // constructor to set defaults
72
73          enum class MStates // enumeration to organize all possible states
74          {
75              off,
76              in,
77              out
78          };
79
80          void update(); // updates the subsystem based on the state
81
82          /* -------------------------- State -------------------------- */
83          MStates mstate;
84
85          /* -------------------------- Other -------------------------- */
86          Motor mmotor;
87      } mindexer;
88
89      struct MFlywheel // controls the flywheel
90      {
91          MFlywheel(); // constructor to set defaults
92
93          enum class MStates // enumeration to organize all possible states
94          {
95              off,
96              shoot, // forward
97              filter // reverse
98          };
99
100         void update(); // updates the subsystem based on the state
101
```

```
        /* ------------------------- State ------------------------- */
        MStates mstate;

        /* ------------------------- Other ------------------------- */
        MotorGroup mmotors;
    } mflywheel;
};

namespace def
{
extern BallControlStateMachine
    sm_bc; // declares the sm_bc object as extern, to make sure it only gets
    ↪  constructed once
} // namespace def
```

1.11 include/stateMachines/DrivetrainStateMachine.hpp

```cpp
/**
 * DrivetrainStateMachine.hpp
 *
 * This file contains the declaration of the DrivetrainStateMachine class.
 * DrivetrainStateMachine is a state machine that inherits from VStateMachine.
 * It has an enumeration of different possible states to make it easy for
 * the user to controll the drivetrain.
 *
 * To use the state machine in auton, you use doAutonMotion() to disable
 * the normal state machine tasks and run the specified action.
 */
#pragma once // makes sure the file is only included once
#include "main.h" // gives access to dependancies from other files
class DrivetrainStateMachine : public VStateMachine // state machine to represent the drivetrain
{
  public:
    DrivetrainStateMachine(); // constructor to set defaults
    enum class MStates // enumeration to organize possible states
    {
        off,
        busy, // doing an AutonMotion
        manual, // standard split arcade drive
        fieldCentric // standard split arcade, but with field centric turns
    };
    MStates getState();
    void setState(MStates istate);

    void
    doAutonMotion(std::function<void()> iaction); // disable manual control, and execute the action

    void controlState() override; // update the state based on controller input
    void update() override; // move the robot based on the state

  private:
    /* -------------------------- State -------------------------- */
    MStates mstate, mlastState;

    bool stateChanged(); // returns whether the last state is the same as the current one

    /* ------------------------- Controls ------------------------ */
    Controller & mcontroller; // reference to the controller to get joystick values
    ControllerButton &
        mtoggleFieldCentric; // reference to the button that toggles field centric control

    /* -------------------------- Other -------------------------- */
    Drivetrain & mdrivetrain; // reference to the drivetrain to give control of the drivetrain to
                              // the state machine
```

```
48 };
49
50 namespace def
51 {
52 extern DrivetrainStateMachine
53     sm_dt; // declares the sm_dt object as extern, to make sure it only gets
       ↪  constructed once
54 } // namespace def
```

1.12 include/stateMachines/VStateMachine.hpp

```cpp
/**
 * VStateMachine.hpp
 *
 * This file contains the parent class, VStateMachine.
 * VStateMachine is an abstract base class for all state
 * machines. It specifies that all state machines should
 * have a method that controls the state based on user
 * input, and a method that moves the robot based on the state.
 */
#pragma once // makes sure the file is only included once
class VStateMachine // abstract state machine base class
{
  public:
    virtual void controlState() = 0; // changes the state based on user input
    virtual void update() = 0; // controls the subsystem based on the current state
};
```

1.13 include/util/CustomOdometry.hpp

```hpp
/**
 * CustomOdometry.hpp
 *
 * This file contains the declaration of the CustomOdometry class.
 * CustomOdometry is resposible for doing all the math and storing
 * information about the robot's position and orientation. Everthing
 * is static, because there doesn't need to be more than one position
 * calculation.
 */
#pragma once         // makes sure the file is only included once
#include "main.h"    // give access to dependencies from other files
class CustomOdometry // class that organizes the sensors, calculations, and state variables for odometry
{

    /* ------------------------ Constants ------------------------ */
    static const double & moffFIn; // offset of forward tracking wheel in inches
    static const double & moffSIn; // offset of side tracking wheel in inches
    static const double & mcircIn; // tracking wheel circumference in inches

    /* -------------------- Sensor References -------------------- */
    static ADIEncoder & meF;  // left tracking wheel encoder
    static ADIEncoder & meS;  // right tracking wheel encoder
    static pros::Imu & mimu1; // inertial sensors
    static pros::Imu & mimu2;

    /* ------------------------ Variables ------------------------ */
    static OdomState mstate; // position of the robot
    static bool menabled;    // whether or not the loop is allowed to run

    /* ------------------------ Methods ------------------------ */
    static std::valarray<double> getSensorVals(); // returns new sensor values
    friend void odomTaskFunc(void *);             // task to be run all the time.

  public:
    static OdomState getState();                    // returns the current state of the robot
    static QLength getX();                          // returns the x value of the state
    static QLength getY();                          // returns the y value of the state
    static QAngle getTheta();                       // returns the theta value of the state
    static void setState(const OdomState & istate); // sets the state of the robot

    static void enable();  // allows the odometry thread to be started (but does not start it)
    static void disable(); // stops the odometry thread from running, prevents it from starting

    static OdomState mathStep(std::valarray<double> ivalsDiff); // does one iteration of odometry math, given sensor changes
};

```

```
47 namespace def
48 {
49 extern CustomOdometry customOdom; // declares the customOdom object as extern, to make
   ↪  sure it only gets constructed once
50 }
51
52 void odomTaskFunc(void *); // friend function to CustomOdometry to be run as a separate
   ↪  thread
```

1.14 include/util/definitions.hpp

```cpp
/**
 * definitions.hpp
 *
 * This file contains various declarations and definitions for
 * motors, sensors, controls, constants, and settings, so that
 * things that might need to be changed are all in one place.
 */
#pragma once // makes sure the file is only included once
#include "main.h" // gives access to dependancies from other files

#define DT_STATES DrivetrainStateMachine::MStates
#define IT_STATES BallControlStateMachine::MIntake::MStates
#define IX_STATES BallControlStateMachine::MIndexer::MStates
#define FW_STATES BallControlStateMachine::MFlywheel::MStates

#define makeFunc(i) [&]() i
#define cutDrive(i)                \
    {                              \
        AsyncAction(i, makeFunc({ def::drivetrain.disable(); }))
                \
    }
namespace def
{
/* ------------------------------------------------------------ */
/*                            Devices                           */
/* ------------------------------------------------------------ */

/* -------------------------- Motors ------------------------- */
extern Motor mtr_dt_left_front;
extern Motor mtr_dt_right_front;
extern Motor mtr_dt_right_back;
extern Motor mtr_dt_left_back;
/* ------------------------------------------------------------ */
extern Motor mtr_it_left;
extern Motor mtr_it_right;
/* ------------------------------------------------------------ */
extern Motor mtr_ix;
/* ------------------------------------------------------------ */
extern Motor mtr_fw1;
extern Motor mtr_fw2;

/* -------------------------- Sensors ------------------------ */
extern ADIEncoder track_encoder_forward;
extern ADIEncoder track_encoder_side;
extern pros::Imu imu_top;
extern pros::Imu imu_bottom;

/* ------------------------------------------------------------ */
/*                            Controls                          */
/* ------------------------------------------------------------ */
```

```
50  extern Controller controller;
51
52  /* ------------------------ Drivetrain ---------------------- */
53  extern ControllerButton btn_dt_tglFieldCentric;
54
55  /* ----------------------- Ball Control --------------------- */
56  extern ControllerButton btn_bc_in;
57  extern ControllerButton btn_bc_out;
58  extern ControllerButton btn_bc_shoot;
59  extern ControllerButton btn_bc_down;
60
61  /* ------------------------------------------------------------ */
62  /*                          Constants                           */
63  /* ------------------------------------------------------------ */
64
65  /* --------------------- Tracking Wheels -------------------- */
66  const QLength TRACK_WHEEL_DIAMETER = 2.847_in;
67  const QLength TRACK_WHEEL_CIRCUMFERENCE = TRACK_WHEEL_DIAMETER * M_PI;
68  const QLength TRACK_FORWARD_OFFSET = 2.3_in;
69  const QLength TRACK_SIDE_OFFSET = 7_in;
70
71  /* ------------------------ Drivetrain ---------------------- */
72  const QLength DRIVE_WHEEL_DIAMETER = 4.041_in;
73  const double DRIVE_WHEEL_DIAMETER_IN = DRIVE_WHEEL_DIAMETER.convert(inch);
74  const QLength DRIVE_WHEEL_CIRCUMFERENCE = DRIVE_WHEEL_DIAMETER * M_PI;
75  const double DRIVE_WHEEL_CIRCUMFERENCE_IN = DRIVE_WHEEL_CIRCUMFERENCE.convert(inch);
76  const QLength DRIVE_OFFSET = 42_in;
77
78  const QAcceleration DRIVE_MAX_ACCEL = 1_G; // approxamate measured linear acceleration
79  const QSpeed DRIVE_MAX_SPEED = 2.7_mps; // a measured linear velocity
80
81  /* ------------------------ Settings ------------------------ */
82  const double SET_DT_MAX = 1; // default drivetrain max speed (1 is 100%)
83  const OdomState SET_ODOM_START = {0_ft, 0_ft, 0_deg}; // starting position of the robot
    ↪  on the field
84  } // namespace def
```

1.15 include/util/util.hpp

```cpp
/**
 * util.hpp
 *
 * This file contains miscellaneous utility functions and classes
 * to help with the general organization of the rest of the code.
 */
#pragma once      // makes sure the file is only included once
#include "main.h" // gives access to dependancies from other files

/* ------------------------------------------------------------ */
/*                     ExtendedPoint Struct                     */
/* ------------------------------------------------------------  /
 * ExtendedPoint struct inherits from the built in okapi Point struct,
 * but provides additional functionality, like an orientation value
 * (theta) as well as x and y values. It also adds some vector operations.
 */
struct ExtendedPoint : Point
{
    ExtendedPoint(QLength ix, QLength iy, QAngle itheta);

    QAngle theta{0_deg}; // stores the orientation at the point, with a default of 0
    ↪  degrees

    /* ----------------------- Subtraction ----------------------- */
    ExtendedPoint operator-(const ExtendedPoint & ivec);
    ExtendedPoint sub(const ExtendedPoint & ivec);

    /* ------------------------- Addition ------------------------- */
    ExtendedPoint operator+(const ExtendedPoint & ivec);
    ExtendedPoint add(const ExtendedPoint & ivec);

    /* ---------------------- Multipliation ---------------------- */
    QLength dot(const ExtendedPoint & ivec); // dot multiply the vectors
    ExtendedPoint operator*(const double iscalar);
    ExtendedPoint scalarMult(const double iscalar);      // multiply the vectors by a
    ↪  scalar
    ExtendedPoint operator*(const ExtendedPoint & ivec); // elementwise multiplication
    ExtendedPoint eachMult(const ExtendedPoint & ivec);

    /* ----------------------- Comparative ----------------------- */
    bool operator==(const ExtendedPoint & ipoint); // checks to see of the points are
    ↪  the same

    /* ------------------------- Other ------------------------- */
    QLength dist(const ExtendedPoint & ivec); // distance between points
    QLength mag();                            // magnitude
    ExtendedPoint normalize();                // creates a vector with a length of 1
    std::string string();
};

/* ------------------------------------------------------------ */
/*                        Misc Functions                        */
```

```
/* ------------------------------------------------------------ */
void waitForImu(); // blocks execution of the code until the imu is done calibrating

/* -------------------- OdomDebug Helpers -------------------- */
void odomSetState(OdomDebug::state_t istate); // sets the state of odometry based on
↪  display inputs
void odomResetAll();                          // resets everything having to do with
↪  odometry (for "Reset" button)

/* ---------------------- Task Functions ---------------------- */
void sm_dt_task_func(void *); // state machine drivetrain task to be run independently
void sm_bc_task_func(void *); // state machine ball control task to be run independently

void display_task_func(void *); // display task to be run independently

/* -------------------------- Macros ------------------------- */
void deploy(); // deploys the robot

/* ------------------------------------------------------------ */
/*                           Control                            */
/* ------------------------------------------------------------ */

/* ------------------------ PID Class ------------------------ /
 * PID is a feedback loop that uses the difference between the goal
 * and the current position (error) of the robot to decide how much
 * power to give the motors. The "P" stands for "proportional", and
 * it adds power proportional to the error, so it gets slower and
 * slower as it gets closer to the goal to prevent it from driving
 * too fast passed it. The "D" stands for "derivative", because it
 * uses the derivative of the error (the speed of the robot) to apply
 * power. If the robot is moving too fast, the "D" term will slow
 * down, and if it is moving too slow, the "D" term will speed up.
 * The "I" stands for "integral", because it uses the integral of the
 * error (the absement of the robot) to apply power. When the robot
 * is close to the goal, sometimes the "P" and "D" terms do not
 * apply enough power to move the robot, but when the robot isn't
 * moving, the "I" term is acumulating, so it eventually builds up
 * enough to get the robot even closer to the goal. This implementation
 * of PID only enables the "I" term when the robot is close enough
 * to the goal, to prevent "integral windup", which is when the
 * integral gets too big when it's too far away from the goal.
 *
 * We have a PID controller class, because we use different PID loops
 * in many different places in the code, so we wanted to be able to
 * be able to quickly make one with constants specific to the application.
 */
class PID
{
    double msettlerError, msettlerDerivative; // target error and derivative for the
    ↪  settler
    QTime msettlerTime;                       // target time for the settler
    std::unique_ptr<SettledUtil> msettler;    // okapi settler that is used to
    ↪  determine if the PID should stop, based on error, derivative, and time

```

```
    double mkP, mkI, mkD, mkIRange; // constants

    double merror, mlastError, mtotalError;
    double mderivative; // used for storing derivative before lastError is overwritten

  public:
    PID(double ikP, double ikI, double ikD, double ikIRange, double isettlerError,
    ↪  double isettlerDerivative, QTime isettlerTime); // constructor

    PID(const PID & iother); // copy constructor

    double getLastError();
    double getTotalError();

    void setGains(double ikP, double ikI, double ikD);

    double getP();
    double getI();
    double getD();

    double iterate(double ierror); // goes through one iteration of the PID loop

    bool isSettled(); // returns whether or not the controller is settled at the target
};

/* ------------------------ Slew Class ------------------------ /
 * Slew rate control is a system that limits the change in speed to
 * prevent wheel slip. If the robot changes speed to fast, the wheels
 * can slip, and make the robot's motion less fluid. When the target
 * speed changes by a lot, the slew rate controller slowly increases
 * it's output to eventually get to the target speed.
 *
 * This Slew rate controller is also intended to be used with PID, but
 * sometimes when slew is used with PID, it interferes with the settling
 * of the PID. To prevent this, the slew rate controller is only active
 * when there are large changes in the target input value, making it only
 * really affect the beginning of the motion. For example, if the motors
 * aren't moving, and the target value suddenly jumps to 100%, the slew
 * controller might gradually increase by increments of 5% until it
 * reaches 100%, but if the target value jumps to from0% to 20%, the
 * slew controller might not engage (actual values depend on constants
 * "mincrement" and "mactiveDifference").
 */
class Slew
{
    double mincrement;        // amount to change between each iteration
    double mactiveDifference; // threshold to activate slew
    double mlastValue;        // previous value

  public:
    Slew(double iincrement, double iactiveDifference); // constructor

    double getIncrement();
    double getActiveDifference();
```

```
153     double getLastValue();
154 
155     double iterate(double ivalue); // limits the input value to maximum changes
        ↪  described by constants when run in a loop
156 };
157 
158 /* ------------------------------------------------------------ */
159 /*                             Util                             */
160 /* ------------------------------------------------------------  /
161  * The util namespace is used to organaize basic functions that don't
162  * necessarily need to be used for robotics.
163  */
164 namespace util
165 {
166 
167 /* ----------------------- DEMA Filter -----------------------  /
168  * DEMA is short for Double Exponential Moving Average. It is a method
169  * is a type of filter that smooths data and gives more weight to
170  * more recent values.
171  *
172  * A Simple Moving Average (SMA) takes the mean of a certain number
173  * of values over a specified period of time. An Exponential Moving
174  * Average (EMA) is similar, but it gives more weight to newer values,
175  * so it more closly tracks the actual value. A DEMA is the EMA of
176  * an EMA. More specifically, it is calculated by 2EMA - EMA(EMA),
177  * and it gives even more weight to newer values.
178  *
179  * The DEMAFilter class was originally added as an easy way to improve
180  * the quality of angle measurements from the inertial sensor. It was
181  * needed because the odometry loop updates at 100hz, and the inertial
182  * sensors used to only update at 50hz. The DEMA filter did improve
183  * the position calculation a small amount, but now the inertial
184  * sensor can update at 100hz (maybe more; it's unclear), and the
185  * filter is no longer useful.
186  */
187 template <int N> // the DEMA filter can be set to use the previous N values, changing
    ↪  how significant newer values are
188 class DEMAFilter
189 {
190     const double mk;              // weighting constant
191     double mlastEMA, mlastEMAEMA; // previous EMA values
192 
193     double EMACalc(double & inext, double & iold) { return (inext - iold) * mk + iold;
        ↪  } // EMA calculation
194 
195   public:
196     DEMAFilter(std::array<double, 2 * N - 1> ifirstVals) : mk(2.0 / (N + 1)) // to
        ↪  start filtering values, the DEMA filter needs to have pre-filtered values. The
        ↪  constructor calculates the "last" values of the EMA and EMA(EMA)
197     {
198         for (int i = 0; i < N; i++) // calc sum of the first N numbers
199         {
200             mlastEMA += ifirstVals[i];
201         }
```

```
202         mlastEMA /= N; // store the SMA (mean) of the first N numbers
203 
204         mlastEMAEMA = mlastEMA;
205         for (int i = 0; i < N - 1; i++)
206         {
207             mlastEMA = EMACalc(ifirstVals[i + N], mlastEMA); // put the next values
                ↪  from the input through EMA filter
208             mlastEMAEMA += mlastEMA;                         // store sum of these
                ↪  values
209         }
210         mlastEMAEMA /= N; // store the SMA (mean) of the first N values from the EMA
211     }
212 
213     double filter(double iinput) // filters the input value by doing DEMA calculation
214     {
215         double EMA = EMACalc(iinput, mlastEMA);    // first EMA
216         double EMAEMA = EMACalc(EMA, mlastEMAEMA); // EMA of first EMA (EMA(EMA))
217         mlastEMA = EMA;                            // store previous EMA
218         mlastEMAEMA = EMAEMA;                      // store previous EMA(EMA)
219 
220         return 2 * EMA - EMAEMA; // 2EMA - EMA(EMA)
221     }
222 };
223 
224 /* ---------------------- Angle Wrappers ---------------------  /
225  * All of these functions take an angle as an input, and return an
226  * angle fitting into a certain range. For example, wrapDeg(370) would
227  * return 10, and wrapDeg180(200) would return -160
228  */
229 double wrapDeg(double iangle);       // [0, 360)
230 double wrapDeg180(double iangle);    // [-180, 180)
231 double wrapRad(double iangle);       // [0, 2pi)
232 double wrapRadPI(double iangle);     // [-pi, pi)
233 QAngle wrapQAngle(QAngle iangle);    // [0_deg, 360_deg)
234 QAngle wrapQAngle180(QAngle iangle); // [-180_deg, 180_deg)
235 
236 /* ------------------------ Find Max ------------------------  /
237  * these functions all find the maximum value of a few different types
238  * of inputs. They use templates so they can be used on different types,
239  * and on arrays of different lengths.
240  */
241 template <class T, std::size_t N>
242 T findMax(const std::array<T, N> && iarray) // returns the max value in iarray
243 {
244     T largest = iarray.at(0);    // gives largest a value to compare with
245     for (const T & val : iarray) // loops through all values
246         if (val > largest)
247             largest = val; // stores the largest value
248     return largest;
249 }
250 template <class T, std::size_t N>
251 T findMax(const std::array<T, N> & iarray) // returns the max value in iarray
252 {
253     T largest = iarray.at(0);    // gives largest a value to compare with
```

```
254     for (const T & val : iarray) // loops through all values
255         if (val > largest)
256             largest = val; // stores the largest value
257     return largest;
258 }
259 template <class T, std::size_t N>
260 T findAbsMax(const std::array<T, N> && iarray) // returns the max absolute value in
    ↪  iarray
261 {
262     T largest = iarray.at(0);    // gives largest a value to compare with
263     for (const T & val : iarray) // loops through all values
264         if (abs(val) > largest)
265             largest = abs(val); // stores the largest value
266     return largest;
267 }
268 template <class T, std::size_t N>
269 T findAbsMax(const std::array<T, N> & iarray) // returns the max absolute value in
    ↪  iarray
270 {
271     T largest = iarray.at(0);    // gives largest a value to compare with
272     for (const T & val : iarray) // loops through all values
273         if (abs(val) > largest)
274             largest = abs(val); // stores the largest value
275     return largest;
276 }
277
278 /* ------------------------ Fitters ------------------------  /
279  * These functions modfify the input to fit in a specified range
280  */
281 template <std::size_t N>
282 std::array<double, N> scaleToFit(double imagnitude, std::array<double, N> && iarray) //
    ↪  scales all elements in iarray to fit within [-imagnitude, imagnitude]
283 {
284     double largest = findAbsMax<double, N>(iarray);
285     if (largest > imagnitude) // if anything is out of range
286     {
287         largest = std::abs(largest);
288         for (double & val : iarray) // scales everything down to fit in the range
289             val = val / largest * imagnitude;
290     }
291     return iarray;
292 }
293 template <std::size_t N>
294 void scaleToFit(double imagnitude, std::array<double, N> & iarray) // scales all
    ↪  elements in iarray to fit within [-imagnitude, imagnitude]
295 {
296     double largest = findAbsMax<double, N>(iarray);
297     if (largest > imagnitude) // if anything is out of range
298     {
299         largest = std::abs(largest);
300         for (double & val : iarray) // scales everything down to fit in the range
301             val = val / largest * imagnitude;
302     }
303 }
```

```
304
305 template <class T, std::size_t N>
306 void chop(T imin, T imax, std::array<T, N> & iarray) // if any values in iarray are out
    ↪ of range, they are set to the limit
307 {
308     for (double & val : iarray)
309     {
310         if (val > imax)
311             val = imax;
312         else if (val < imin)
313             val = imin;
314     }
315 }
316 template <class T>
317 void chop(T imin, T imax, T & inum) // if the value is out of range, it is set to the
    ↪ limit
318 {
319     if (inum > imax)
320         inum = imax;
321     else if (inum < imin)
322         inum = imin;
323 }
324 } // namespace util
```

2 Source Files

2.1 src/Auton.cpp

```cpp
/**
 * Auton.cpp
 *
 * This contains the definitions of the Auton struct,
 * which is responsible for reading the sd card to determine
 * which auton is selected, and running the correct auton.
 */
#include "main.h" // gives access to Auton and other dependencies

Auton::Autons Auton::auton =
    Auton::Autons::none; // default auton is none, if the sd card is not installed

void Auton::readSettings() // read the sd card to set the settings
{
    FILE * file; // cpp a file object to be used later
    if (pros::usd::is_installed()) // checks if the sd card is installed before trying
    ↪  to read it
    {
        file = fopen("/usd/auton_settings.txt", "r"); // open the auton settings
        if (file) // check to see if the file opened correctly
        {
            fscanf(file, "%i", &auton);
        }
        else
        {
            std::cout << "/usd/auton_settings.txt is null."
                      << std::endl; // if the file didn't open right, tell the terminal
        }
        fclose(file); // close the file
    }
}

void Auton::runAuton() // runs the selected auton
{
    pros::Task auton_task(auton_task_func);
}
/*
  ____
 /    \
|   1  |
 \____/

*/

/* ------------------------------------------------------------ */
/*                     Private Information                      */
/* ------------------------------------------------------------ */
void Auton::auton_task_func(void *) // separate thread for running the auton, in case a
↪  particular
                                    // auton needs control over its thread
```

```
{
    // when making autons, you must add the text to the dropdown in DisplayControl.cpp, a new enum
    // value in Auton.hpp, and a new case is this switch
    switch (auton)
    {
        case Autons::none:
            break;
        case Autons::test:
            def::sm_dt.doAutonMotion(makeFunc({
                def::drivetrain.strafeToPoint({4_ft, 0_ft, 0_deg});
                def::drivetrain.strafeToPoint({4_ft, -3.8_ft, 180_deg});
                def::drivetrain.strafeToPoint({4_in, -3.8_ft, 0_deg});
                def::drivetrain.strafeToPoint({4_in, -4_in, 0_deg});
            }));
            break;
        case Autons::prog:
            /**
             * `makeFunc()` is a preprocessor macro that takes the contents and turns them into a
             * lambda function, [](){}. It is frequently used to specify the actions in
             * AcyncActions.
             *
             * `cutDrive()` is a preprocessor macro that adds an AsyncAction to the motion that
             * disables the motion at a certain error (in inches) from the target. This is used
             * frequently because many motions do not need to go exactly to the target. When
             * possible, stopping the motion before it reaches the target is faster, because it
             * doesn't need to use PID to settle.
             */
            CustomOdometry::setState({8_in, 37_in, 0_deg}); // set the starting position
            def::sm_dt.doAutonMotion(makeFunc({
                deploy(); // deploys the hood
                /* -------------------------- Goal 1 ------------------------- */
                def::sm_bc.ixUp(); // start the indexer to get the first ball
                def::drivetrain.strafeToPoint(
                    {17_in, 36_in, 0_deg}, // get the first ball (now 2 balls)
                    cutDrive(2));
                def::drivetrain.strafeToPoint({15_in, 16_in, 225_deg}, // line up with goal #1
                                              cutDrive(1.5));
                def::sm_bc.ixOff(); // turns off the indexer
                pros::delay(300);
                def::sm_bc.shoot(); // now 1 ball

                /* -------------------------- Goal 2 ------------------------- */
                def::drivetrain.strafeToPoint(
                    {35_in, 24_in, -90_deg}, // line up with the second ball
                    cutDrive(2));
                def::sm_bc.ixUp(); // get ready for the next ball by starting the indexer
```

```
95                  def::drivetrain.strafeToPoint({35_in, 10_in, -90_deg}); // get the next
                    ↪  ball (now 2)
96                  def::drivetrain.strafeToPoint({73_in, 26_in, 0_deg}, // get the next
                    ↪  ball (now 3)
97                                          cutDrive(2));
98                  def::drivetrain.strafeToPoint(
99                      {73_in, 29_in, -90_deg}, {}, PID(0.4, 0.005, 2.6, 0.5, 0.5, 0.5,
                        ↪  1_ms),
100                     PID(0.028, 0.0, 0.08, 0.0, 5, 2,
101                         1_ms)); // turn to face the goal with custom PID gains, becuase
                            ↪  for some
102                                 // reason, this specific motion frequently settled
                                    ↪  inconsistantly
103                 def::drivetrain.strafeToPoint({71_in, 20_in, -90_deg}, // drive to the
                    ↪  next goal
104                                         cutDrive(1));
105                 def::sm_bc.ixOff(); // turns off the indexer
106                 def::sm_bc.shoot(); // now 2 balls
107
108                 /* ------------------------- Goal 3 ------------------------ */
109                 def::drivetrain.strafeToPoint(
110                     {108_in, 34_in, 0_deg},
111                     {AsyncAction(10,
112                                  makeFunc({ def::sm_bc.ixUp(); })), // start the
                                    ↪  indexer mid-motion
113                      AsyncAction(2, makeFunc({ def::drivetrain.disable(); }))}); //
                        ↪  same cutDrive(2)
114                 def::drivetrain.strafeToPoint({118_in, 34_in, 0_deg}, cutDrive(1));
115                 def::drivetrain.strafeToPoint({126_in, 14_in, -36_deg}, // line up with
                    ↪  goal #3
116                                         cutDrive(1));
117                 def::sm_bc.ixOff(); // turn off the indexer
118                 def::sm_bc.shoot(); // now 2 balls
119
120                 /* ------------------------- Goal 4 ------------------------ */
121                 def::drivetrain.strafeToPoint({114_in, 72_in, 0_deg}, // move towards
                    ↪  goal #4
122                                         cutDrive(3));
123                 def::sm_bc.ixUp();
124                 def::drivetrain.strafeToPoint({125_in, 72_in, -2_deg}, // line up with
                    ↪  goal #4
125                                         cutDrive(1));
126                 def::sm_bc.ixOff(); // turs the indexer off
127                 def::sm_bc.shoot(); // now 1 ball
128                 def::sm_bc.ixUp(); // get ready to shoot the next ball
129                 pros::delay(700);
130                 def::sm_bc.shoot(); // shoot now 0 balls
131                 def::sm_bc.ixOff(); // turns the indexer off
132
133                 /* ------------------------- Goal 5 ------------------------ */
134                 def::drivetrain.strafeToPoint({122_in, 90_in, 90_deg}, // move towards
                    ↪  the next ball
135                                         cutDrive(2));
136                 def::sm_bc.ixUp(); // turns the indexer on
```

```
137                 def::drivetrain.strafeToPoint(
138                     {123_in, 108_in, 90_deg}, // get the next ball (now 1 ball)
139                     cutDrive(1));
140                 def::drivetrain.strafeToPoint({120_in, 124_in, 45_deg}, // move towards
                    ↪  goal #5
141                                               cutDrive(3));
142                 def::drivetrain.strafeToPoint({130_in, 126_in, 43_deg}, // line up with
                    ↪  goal #5
143                                               cutDrive(1));
144                 def::sm_bc.shoot(500); // now 0 balls
145                 def::sm_bc.ixOff(); // turns the indexer off
146
147                 /* ------------------------- Goal 6 ------------------------ */
148                 def::sm_bc.ixUp(); // turns the indexer on
149                 def::drivetrain.strafeToPoint(
150                     {85_in, 120_in, 180_deg}, // move towards the next ball
151                     cutDrive(2));
152                 def::drivetrain.strafeToPoint({70_in, 118_in, 180_deg}, // get the next
                    ↪  ball (now 1)
153                                               cutDrive(1));
154                 def::drivetrain.strafeToPoint({77_in, 123_in, 90_deg},
155                                               cutDrive(0.25)); // line up with the goal
156                 def::sm_bc.shoot(600); // now 0
157                 def::sm_bc.ixOff(); // turns off the indexer
158                 pros::delay(100); // pause to make sure the shot works
159
160                 /* ------------------------- Goal 7 ------------------------ */
161                 def::sm_bc.ixUp(); // get ready for the next ball
162                 def::drivetrain.strafeToPoint({37_in, 124_in, 90_deg}, // line up with
                    ↪  the next ball
163                                               cutDrive(3));
164                 def::drivetrain.strafeToPoint({38_in, 136_in, 90_deg}); // get the next
                    ↪  ball (now 1)
165                 def::drivetrain.strafeToPoint({17.5_in, 125.5_in, 119_deg}, // line up
                    ↪  with goal #7
166                                               cutDrive(1));
167                 def::sm_bc.shoot(600); // now 0 balls
168                 def::sm_bc.ixOff(); // turns off the indexer
169
170                 /* ------------------------- Goal 8 ------------------------ */
171                 def::drivetrain.strafeToPoint({28_in, 129_in, -90_deg}); // turn around
172                 def::sm_bc.ixUp(); // turns the indexer on
173                 def::drivetrain.strafeToPoint(
174                     {28_in, 119_in, -90_deg}, // gets the next ball (now 1)
175                     cutDrive(2));
176                 def::drivetrain.strafeToPoint({28_in, 69_in, 180_deg}, // move towards
                    ↪  goal #8
177                                               cutDrive(3));
178                 def::drivetrain.strafeToPoint({21_in, 72_in, 180_deg}, // line up with
                    ↪  goal #8
179                                               cutDrive(1));
180                 def::sm_bc.shoot(600); // now 0
181                 def::sm_bc.ixOff();
182
```

```
                /* ------------------------- Goal 9 ------------------------- */
                def::drivetrain.strafeToPoint({23_in, 72_in, 0_deg}, cutDrive(0.5)); // turn around
                def::sm_bc.ixUp(); // get ready to get the ball by turning the indexer on
                def::drivetrain.strafeToPoint({48_in, 72_in, 0_deg}, // get the next ball (now 1)
                                              cutDrive(1));
                def::drivetrain.strafeToPoint(
                    {36_in, 72_in, 0_deg}, // back up to make sure the descorer doesn't hit the goal
                    cutDrive(2));
                def::sm_bc.itOut(); // deploy
                pros::delay(600); //
                def::sm_bc.itOff(); // stop the descorer
                def::drivetrain.strafeToPoint({56_in, 69_in, 0_deg}); // descore
                pros::delay(500);
                def::sm_bc.ixOff(); // stop the indexer
                def::drivetrain.strafeToPoint({30_in, 76_in, 0_deg}, cutDrive(2)); // back up
                def::drivetrain.strafeToPoint({57_in, 79_in, -10_deg}, // go to the goal
                                              cutDrive(1));
                def::sm_bc.shoot(); // now 0
            }));
            break;
    }
}
```

2.2 src/definitions.cpp

```cpp
/**
 * definitions.cpp
 *
 * This file contains various declarations and definitions for
 * motors, sensors, controls, constants, and settings, so that
 * things that might need to be changed are all in one place.
 */
#include "main.h" // gives access to definition.hpp and other dependencies

namespace def
{
/* ------------------------------------------------------------ */
/*                           Devices                            */
/* ------------------------------------------------------------ */

/* ------------------------- Motors ------------------------- */
Motor mtr_dt_left_front(16);
Motor mtr_dt_right_front(-1);
Motor mtr_dt_right_back(-19);
Motor mtr_dt_left_back(20);
/* ------------------------------------------------------------ */
Motor mtr_it_left(18);
Motor mtr_it_right(-3);
/* ------------------------------------------------------------ */
Motor mtr_ix(2);
/* ------------------------------------------------------------ */
Motor mtr_fw1(-17);
Motor mtr_fw2(7);

/* ------------------------- Sensors ------------------------- */
ADIEncoder track_encoder_forward('G', 'H', true);
ADIEncoder track_encoder_side('E', 'F', true);
pros::Imu imu_top(4);
pros::Imu imu_bottom(5);

/* ------------------------------------------------------------ */
/*                           Controls                           */
/* ------------------------------------------------------------ */
Controller controller = Controller();

/* ------------------------ Drivetrain ----------------------- */
ControllerButton btn_dt_tglFieldCentric = ControllerDigital::A;

/* ----------------------- Ball Control ---------------------- */
ControllerButton btn_bc_in = ControllerDigital::R1;
ControllerButton btn_bc_out = ControllerDigital::R2;
ControllerButton btn_bc_shoot = ControllerDigital::L1;
ControllerButton btn_bc_down = ControllerDigital::L2;

/* ------------------------------------------------------------ */
/*                           Constructs                         */
/* ------------------------------------------------------------ */
```

```
53 CustomOdometry customOdom = CustomOdometry(); // object that calculates position
54
55 Drivetrain drivetrain = Drivetrain(); // used by DrivetrainStateMachine for drivetrain
   ↪  control
56
57 DrivetrainStateMachine sm_dt = DrivetrainStateMachine(); // state machine to control
   ↪  the drivetrain
58 BallControlStateMachine sm_bc = BallControlStateMachine(); // state machine for ball
   ↪  manipulators
59 } // namespace def
```

2.3 src/main.cpp

```cpp
/**
 * main.cpp
 *
 * This file contains the orchestration of all the compenents. It
 * starts all of the separate tasks that are needed for controlling
 * the robot, and has all the functions called by the competition
 * switch.
 */
#include "main.h" // gives access to dependencies from other files

DisplayControl def::display = DisplayControl();
pros::Task sm_dt_task(sm_dt_task_func);
pros::Task sm_bc_task(sm_bc_task_func);
pros::Task odomTask(odomTaskFunc);
pros::Task display_task(display_task_func);

/**
 * 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()
{
    Auton::readSettings(); // read sd card to remeber the auton selected when the brain
    ↪  was run last
    def::display.setAutonDropdown(); // update auton dropdown to match the sd card

    def::mtr_dt_left_front.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
    def::mtr_dt_right_front.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
    def::mtr_dt_right_back.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
    def::mtr_dt_left_back.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
    def::mtr_it_left.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
    def::mtr_it_right.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
    def::mtr_ix.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
    def::mtr_fw1.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
    def::mtr_fw2.setEncoderUnits(AbstractMotor::encoderUnits::degrees);
}

/**
 * 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.
 */
void autonomous()
{
    Auton::runAuton(); // uses the auton class to run the slected auton
}

/**
 * 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 opcontrol()
{

    def::sm_dt.setState(
        DT_STATES::manual); // set the drivetrain to basic controls during drivercontrol

    // there is no need for a loop in opcontrol(), because there are already other
    ↪  tasks running
    // that control all of the movement
    // while (true)
    // {
    //     pros::delay(20);
    // }
}
```

2.4 src/gui/DisplayControl.cpp

```cpp
/**
 * DisplayControl.cpp
 *
 * This file cointains the definitions for the DisplayControl class.
 * DisplayControl is the class that handles the organization of the
 * LittleV Graphics Library (LVGL) objects on the screen of the brain.
 */
#include "main.h" // gives access to the DisplayControl declaration and other dependencies

/* ---------------------- Tabview Elements -------------------- */
lv_obj_t * DisplayControl::mtabview = lv_tabview_create(lv_scr_act(), NULL); // creates the tabview

lv_obj_t * DisplayControl::mtabview_odom = lv_tabview_add_tab(
    DisplayControl::mtabview,
    "Odom"); // creates the tab on the screen that shows the calculated robot position

lv_obj_t * DisplayControl::mtabview_auton = lv_tabview_add_tab(
    DisplayControl::mtabview, "Auton"); // creates the tab with the auton selection dropdown
lv_obj_t * DisplayControl::mtabview_auton_dropdown =
    lv_ddlist_create(mtabview_auton, NULL); // creates the auton selection dropdown
lv_res_t DisplayControl::tabview_auton_dropdown_action(
    lv_obj_t * idropdown) // specifies the code to be executed when the auton dropdown is changed
{
    FILE * file; // creates an object that will be used to reference the file containing the
                 // selected auton
    if (pros::usd::is_installed()) // makes sure the sd card is installed before trying to access
                                   // its contents
    {
        file = fopen("/usd/auton_settings.txt", "w"); // opens the auton settings file
        if (file) // makes sure the file was accessed correctly
        {
            fprintf(file, "%i",
                    lv_ddlist_get_selected(idropdown)); // update sd card based on new value
        }
        else
        {
            std::cout
                << "/usd/auton_settings.txt is null"
                << std::endl; // output to the terminal if the sd card was not accessed correctly
        }
        fclose(file);
        Auton::readSettings(); // update auton based on new sd card values
    }

```

```
45      return LV_RES_OK; // required for dropdown callback
46  }
47
48  lv_obj_t * DisplayControl::mtabview_graph = lv_tabview_add_tab(
49      DisplayControl::mtabview, "Graph"); // creates the tab with the graph for debugging
50  lv_obj_t * DisplayControl::mtabview_graph_chart =
51      lv_chart_create(DisplayControl::mtabview_graph, NULL); // create the graph
52
53  // create 7 series of different color, so it is easy to make a graph with any color
54  lv_chart_series_t * DisplayControl::mtabview_graph_chart_series_0 =
55      lv_chart_add_series(DisplayControl::mtabview_graph_chart, LV_COLOR_RED);
56  lv_chart_series_t * DisplayControl::mtabview_graph_chart_series_1 =
57      lv_chart_add_series(DisplayControl::mtabview_graph_chart, LV_COLOR_ORANGE);
58  lv_chart_series_t * DisplayControl::mtabview_graph_chart_series_2 =
59      lv_chart_add_series(DisplayControl::mtabview_graph_chart, LV_COLOR_YELLOW);
60  lv_chart_series_t * DisplayControl::mtabview_graph_chart_series_3 =
61      lv_chart_add_series(DisplayControl::mtabview_graph_chart, LV_COLOR_GREEN);
62  lv_chart_series_t * DisplayControl::mtabview_graph_chart_series_4 =
63      lv_chart_add_series(DisplayControl::mtabview_graph_chart, LV_COLOR_BLUE);
64  lv_chart_series_t * DisplayControl::mtabview_graph_chart_series_5 =
65      lv_chart_add_series(DisplayControl::mtabview_graph_chart, LV_COLOR_PURPLE);
66  lv_chart_series_t * DisplayControl::mtabview_graph_chart_series_6 =
67      lv_chart_add_series(DisplayControl::mtabview_graph_chart, LV_COLOR_MAGENTA);
68
69  lv_obj_t * DisplayControl::mtabview_misc =
70      lv_tabview_add_tab(DisplayControl::mtabview, "Misc"); // creates the miscelaneus
        ↪  debugging tab
71  lv_obj_t * DisplayControl::mtabview_misc_container =
72      lv_obj_create(DisplayControl::mtabview_misc, NULL);
73  lv_obj_t * DisplayControl::mtabview_misc_label =
74      lv_label_create(mtabview_misc_container, NULL); // creates the left text box
75  lv_obj_t * DisplayControl::mtabview_misc_label_2 =
76      lv_label_create(mtabview_misc_container, NULL); // creates the right text box
77
78  /* ------------------------- Styles ------------------------ */
79  lv_style_t DisplayControl::mstyle_tabview_indic;
80  lv_style_t DisplayControl::mstyle_tabview_btn;
81  lv_style_t DisplayControl::mstyle_tabview_btn_tgl;
82  lv_style_t DisplayControl::mstyle_tabview_btn_pr;
83  lv_style_t DisplayControl::mstyle_tabview_container;
84  lv_style_t DisplayControl::mstyle_text;
85
86  /* ------------------------------------------------------------ */
87  /*                      Public Information                      */
88  /* ------------------------------------------------------------ */
89  DisplayControl::DisplayControl() : modom(mtabview_odom, LV_COLOR_PURPLE)
90  {
91      /* ----------------------- Style Setup ----------------------- /
92       * Specifies what each style should look like when they are used.
93       */
94      lv_style_copy(&mstyle_tabview_indic, &lv_style_plain);
95      mstyle_tabview_indic.body.padding.inner = 5;
96
97      lv_style_copy(&mstyle_tabview_btn, &lv_style_plain);
```

```
    mstyle_tabview_btn.body.main_color = LV_COLOR_PURPLE;
    mstyle_tabview_btn.body.grad_color = LV_COLOR_PURPLE;
    mstyle_tabview_btn.text.color = LV_COLOR_WHITE;
    mstyle_tabview_btn.body.border.part = LV_BORDER_BOTTOM;
    mstyle_tabview_btn.body.border.color = LV_COLOR_WHITE;
    mstyle_tabview_btn.body.border.width = 1;
    mstyle_tabview_btn.body.padding.ver = 4;

    lv_style_copy(&mstyle_tabview_btn_tgl, &mstyle_tabview_btn);
    mstyle_tabview_btn_tgl.body.border.part = LV_BORDER_FULL;
    mstyle_tabview_btn_tgl.body.border.width = 2;

    lv_style_copy(&mstyle_tabview_btn_pr, &lv_style_plain);
    mstyle_tabview_btn_pr.body.main_color = LV_COLOR_WHITE;
    mstyle_tabview_btn_pr.body.grad_color = LV_COLOR_WHITE;
    mstyle_tabview_btn_pr.text.color = LV_COLOR_WHITE;

    lv_style_copy(&mstyle_tabview_container, &lv_style_plain_color);
    mstyle_tabview_container.body.main_color = LV_COLOR_PURPLE;
    mstyle_tabview_container.body.grad_color = LV_COLOR_PURPLE;
    mstyle_tabview_container.body.border.width = 0;
    mstyle_tabview_container.body.radius = 0;
    mstyle_tabview_container.body.padding.inner = 0;
    mstyle_tabview_container.body.padding.hor = 0;
    mstyle_tabview_container.body.padding.ver = 0;

    lv_style_copy(&mstyle_text, &lv_style_plain);
    mstyle_text.text.color = LV_COLOR_WHITE;
    mstyle_text.text.opa = LV_OPA_100;

    lv_tabview_set_style(mtabview, LV_TABVIEW_STYLE_INDIC,
                         &mstyle_tabview_indic); // set tabview styles
    lv_tabview_set_style(mtabview, LV_TABVIEW_STYLE_BTN_REL, &mstyle_tabview_btn);
    lv_tabview_set_style(mtabview, LV_TABVIEW_STYLE_BTN_PR, &mstyle_tabview_btn_pr);
    lv_tabview_set_style(mtabview, LV_TABVIEW_STYLE_BTN_TGL_REL,
    ↪  &mstyle_tabview_btn_tgl);
    lv_tabview_set_style(mtabview, LV_TABVIEW_STYLE_BTN_TGL_PR, &mstyle_tabview_btn_pr);

    /* ------------------------ Auton Tab ------------------------  /
     * When making autons, you must add the text this dropdown, a new
     * enum value in Auton.hpp, and a new case in the switch in Auton.cpp.
     */
    lv_ddlist_set_options(mtabview_auton_dropdown, "none\n"
                                                   "test\n"
                                                   "prog\n"); // auton types in
                                                   ↪  selection dropdown
    lv_ddlist_set_action(mtabview_auton_dropdown,
                         tabview_auton_dropdown_action); // set the dropdown callback to
                                                         // tabview_auton_dropdown_acti
                                                         ↪  on()
    lv_obj_align(mtabview_auton_dropdown, NULL, LV_ALIGN_IN_TOP_LEFT, 0,
                 0); // align the dropdown in the top left

    lv_obj_set_style(mtabview_auton, &mstyle_tabview_container); // set styles
```

```
149
150     /* ------------------------ Graph Tab ------------------------ */
151     lv_obj_set_style(mtabview_graph, &mstyle_tabview_container); // set styles
152     lv_obj_set_style(mtabview_graph_chart, &mstyle_tabview_btn_pr);
153
154     lv_page_set_sb_mode(mtabview_graph, LV_SB_MODE_OFF); // hide scrollbar
155
156     lv_chart_set_type(mtabview_graph_chart, LV_CHART_TYPE_LINE); // make chart graph
        ↪ lines
157     lv_chart_set_point_count(mtabview_graph_chart,
158                              lv_obj_get_width(mtabview_graph_chart) * 2); // set number
                                 ↪ of points
159     lv_chart_set_div_line_count(mtabview_graph_chart, 9, 5); // set the number of chart
        ↪ lines
160     lv_obj_set_size(mtabview_graph_chart, lv_obj_get_width(mtabview_graph),
161                     lv_obj_get_height(mtabview_graph)); // set the graph to fill the
                        ↪ screen
162     lv_obj_align(mtabview_graph_chart, NULL, LV_ALIGN_CENTER, 0, -10); // center chart
163
164     /* ------------------------ Misc Tab ------------------------ */
165     lv_page_set_sb_mode(mtabview_misc, LV_SB_MODE_OFF); // hide scrollbar
166
167     lv_obj_set_style(mtabview_misc, &mstyle_tabview_container); // set styles
168     lv_obj_set_style(mtabview_misc_container, &mstyle_tabview_container);
169     lv_obj_set_size(mtabview_misc_container, lv_obj_get_width(mtabview_misc),
170                     lv_obj_get_height(mtabview_misc)); // set up the background
171     lv_obj_align(mtabview_misc_container, NULL, LV_ALIGN_CENTER, 0, 0);
172     lv_obj_set_style(mtabview_misc_label, &mstyle_text); // set up text boxes (labels)
173     lv_obj_set_style(mtabview_misc_label_2, &mstyle_text);
174
175     lv_label_set_text(mtabview_misc_label, "No data provided."); // set default text
        ↪ for labels
176     lv_label_set_text(mtabview_misc_label_2, "No data provided.");
177
178     lv_obj_align(mtabview_misc_label, mtabview_misc_container, LV_ALIGN_IN_TOP_LEFT, 0,
179                  0); // align labels
180     lv_obj_align(mtabview_misc_label_2, mtabview_misc_container, LV_ALIGN_IN_TOP_RIGHT,
        ↪ -70, 0);
181
182     modom.setStateCallback(
183         odomSetState); // set callbacks for odomDebug to make it interactive on the
            ↪ screen
184     modom.setResetCallback(odomResetAll);
185 }
186
187 void DisplayControl::setOdomData() // sets the information on the OdomDebug window to
    ↪ the new
188                                    // calculated odom data
189 {
190     modom.setData({CustomOdometry::getX(), CustomOdometry::getY(),
        ↪ CustomOdometry::getTheta()},
191                   {def::track_encoder_forward.get(), def::track_encoder_side.get(),
                      ↪ 0.0});
192 }
```

```
193 void DisplayControl::setAutonDropdown() // update the auton dropdown to match the sd
    ↪  card
194 {
195     lv_ddlist_set_selected(mtabview_auton_dropdown, (int)Auton::auton);
196 }
197
198 void DisplayControl::setChartData(int iseries,
199                                   double ivalue) // inputs new values to a specific
                                      ↪  chart series
200 {
201     switch (iseries) // updates the correct series with the new value
202     {
203         case 0:
204             lv_chart_set_next(mtabview_graph_chart, mtabview_graph_chart_series_0,
                ↪  ivalue);
205             break;
206         case 1:
207             lv_chart_set_next(mtabview_graph_chart, mtabview_graph_chart_series_1,
                ↪  ivalue);
208             break;
209         case 2:
210             lv_chart_set_next(mtabview_graph_chart, mtabview_graph_chart_series_2,
                ↪  ivalue);
211             break;
212         case 3:
213             lv_chart_set_next(mtabview_graph_chart, mtabview_graph_chart_series_3,
                ↪  ivalue);
214             break;
215         case 4:
216             lv_chart_set_next(mtabview_graph_chart, mtabview_graph_chart_series_4,
                ↪  ivalue);
217             break;
218         case 5:
219             lv_chart_set_next(mtabview_graph_chart, mtabview_graph_chart_series_5,
                ↪  ivalue);
220             break;
221         case 6:
222             lv_chart_set_next(mtabview_graph_chart, mtabview_graph_chart_series_6,
                ↪  ivalue);
223             break;
224     }
225 }
226
227 void DisplayControl::setMiscData(int ilabel,
228                                  std::string itext) // set the text on text box (label)
                                     ↪  1 or 2
229 {
230     if (ilabel == 1)
231     {
232         lv_label_set_text(mtabview_misc_label, itext.c_str());
233     }
234     else if (ilabel == 2)
235     {
236         lv_label_set_text(mtabview_misc_label_2, itext.c_str());
```

```
237         }
238 }
```

2.5 src/gui/odomDebug.cpp

```cpp
/**
 * odomDebug.cpp
 *
 * The contents of this file were not written by any members of 333A*.
 * This is code from the publicly availible GitHub repository, odomDebug
 * by theol0403, found here: https://github.com/theol0403/odomDebug.
 *
 * The OdomDebug class is used for the tab on the screen of the brain
 * that shows the odometry position of the robot in the form of number
 * values and a moving circle on a picture of the field representing the
 * robot.
 *
 *    *slight modifications were made to make it work with the display
 */
#include "main.h"

/**
 * @param ix QLength
 * @param iy QLength
 * @param itheta QAngle
 */
OdomDebug::state_t::state_t(QLength ix, QLength iy, QAngle itheta) : x(ix), y(iy),
↪  theta(itheta) {}

/**
 * @param ix inches
 * @param iy inches
 * @param itheta radians
 */
OdomDebug::state_t::state_t(double ix, double iy, double itheta)
    : x(ix * inch), y(iy * inch), theta(itheta * radian)
{
}

/**
 * @param ileft the left encoder value
 * @param iright the right encoder value
 */
OdomDebug::sensors_t::sensors_t(double ileft, double iright)
    : left(ileft), right(iright), hasMiddle(false)
{
}

/**
 * @param ileft the left encoder value
 * @param iright the right encoder value
 * @param imiddle imiddle the middle encoder value
 */
OdomDebug::sensors_t::sensors_t(double ileft, double iright, double imiddle)
    : left(ileft), right(iright), middle(imiddle), hasMiddle(true)
{
}
```

```
52
53  /**
54   * Okapi units that represent a tile (2ft) and a court(12ft)
55   * Literals are `_tl` and `_crt`, respectivly
56   */
57  namespace okapi
58  {
59  constexpr QLength tile = 2 * foot;
60  constexpr QLength court = 12 * foot;
61  inline namespace literals
62  {
63  constexpr QLength operator"" _tl(long double x) { return static_cast<double>(x) * tile;
    ↪  }
64  constexpr QLength operator"" _crt(long double x) { return static_cast<double>(x) *
    ↪  court; }
65  constexpr QLength operator"" _tl(unsigned long long int x) { return
    ↪  static_cast<double>(x) * tile; }
66  constexpr QLength operator"" _crt(unsigned long long int x)
67  {
68      return static_cast<double>(x) * court;
69  }
70  } // namespace literals
71  } // namespace okapi
72
73  /**
74   * Constructs the OdomDebug object.
75   * @param parent the lvgl parent, color is inherited
76   */
77  OdomDebug::OdomDebug(lv_obj_t * parent)
78      : OdomDebug(parent, lv_obj_get_style(parent)->body.main_color)
79  {
80  }
81
82  /**
83   * Constructs the OdomDebug object.
84   * @param parent the lvgl parent
85   * @param mainColor The main color for the display
86   */
87  OdomDebug::OdomDebug(lv_obj_t * parent, lv_color_t mainColor)
88      : container(lv_obj_create(parent, NULL))
89  {
90      /**
91       * Container Style
92       */
93      lv_style_copy(&cStyle, &lv_style_plain_color);
94      cStyle.body.main_color = mainColor;
95      cStyle.body.grad_color = mainColor;
96      cStyle.body.border.width = 0;
97      cStyle.body.radius = 0;
98      cStyle.body.padding.inner = 0;
99      cStyle.body.padding.hor = 0;
100     cStyle.body.padding.ver = 0;
101
102     lv_obj_set_style(parent, &cStyle);
```

```
103
104     lv_obj_set_size(container, lv_obj_get_width(parent), lv_obj_get_height(parent));
105     lv_obj_align(container, NULL, LV_ALIGN_CENTER, 0, 0);
106
107     lv_obj_set_style(container, &cStyle);
108
109     /**
110      * Field
111      */
112     lv_obj_t * field = lv_obj_create(container, NULL);
113     fieldDim = std::min(lv_obj_get_width(container), lv_obj_get_height(container));
114     lv_obj_set_size(field, fieldDim, fieldDim);
115     lv_obj_align(field, NULL, LV_ALIGN_IN_RIGHT_MID, 0, 0);
116
117     /**
118      * Field Style
119      */
120     lv_style_copy(&fStyle, &cStyle);
121     fStyle.body.main_color = LV_COLOR_WHITE;
122     fStyle.body.grad_color = LV_COLOR_WHITE;
123     lv_obj_set_style(field, &fStyle);
124
125     /**
126      * Tile Styles
127      */
128     lv_style_copy(&grey, &lv_style_plain);
129     grey.body.main_color = LV_COLOR_HEX(0x828F8F);
130     grey.body.grad_color = LV_COLOR_HEX(0x828F8F);
131     grey.body.border.width = 1;
132     grey.body.radius = 0;
133     grey.body.border.color = LV_COLOR_WHITE;
134
135     lv_style_copy(&red, &grey);
136     red.body.main_color = LV_COLOR_HEX(0xD42630);
137     red.body.grad_color = LV_COLOR_HEX(0xD42630);
138     lv_style_copy(&blue, &grey);
139     blue.body.main_color = LV_COLOR_HEX(0x0077C9);
140     blue.body.grad_color = LV_COLOR_HEX(0x0077C9);
141
142     /**
143      * Tile Layout
144      */
145     std::vector<std::vector<lv_style_t *>> tileData = {
146         {&grey, &red, &grey, &grey, &blue, &grey},  {&red, &grey, &grey, &grey, &grey,
        ↪  &blue},
147         {&grey, &grey, &grey, &grey, &grey, &grey}, {&grey, &grey, &grey, &grey, &grey,
        ↪  &grey},
148         {&grey, &grey, &grey, &grey, &grey, &grey}, {&grey, &grey, &grey, &grey, &grey,
        ↪  &grey}};
149
150     double tileDim = fieldDim / tileData.size(); // tile dimention
151
152     /**
153      * Create tile matrix, register callbacks, assign each tile an ID
```

```
     */
    for (size_t y = 0; y < 6; y++)
    {
        for (size_t x = 0; x < 6; x++)
        {
            lv_obj_t * tileObj = lv_btn_create(field, NULL);
            lv_obj_set_pos(tileObj, x * tileDim, y * tileDim);
            lv_obj_set_size(tileObj, tileDim, tileDim);
            lv_btn_set_action(tileObj, LV_BTN_ACTION_CLICK, tileAction);
            lv_obj_set_free_num(tileObj, y * 6 + x);
            lv_obj_set_free_ptr(tileObj, this);
            lv_btn_set_toggle(tileObj, false);
            lv_btn_set_style(tileObj, LV_BTN_STYLE_PR, tileData[y][x]);
            lv_btn_set_style(tileObj, LV_BTN_STYLE_REL, tileData[y][x]);
        }
    }

    /**
     * Robot point using lvgl led
     */
    led = lv_led_create(field, NULL);
    lv_led_on(led);
    lv_obj_set_size(led, fieldDim / 15 * 2.5, fieldDim / 15 * 2.5);

    lv_style_copy(&ledStyle, &lv_style_plain);
    ledStyle.body.radius = LV_RADIUS_CIRCLE;
    ledStyle.body.main_color = LV_COLOR_MAKE(0, 255, 0);
    ledStyle.body.grad_color = LV_COLOR_MAKE(0, 255, 0);
    ledStyle.body.border.color = LV_COLOR_WHITE;
    ledStyle.body.border.width = 2;
    ledStyle.body.border.opa = LV_OPA_100;
    lv_obj_set_style(led, &ledStyle);

    /**
     * Robot line
     */
    line = lv_line_create(field, NULL);
    lv_line_set_points(line, linePoints.data(), linePoints.size());
    lv_obj_set_pos(line, 0, 0);

    lineWidth = 3;
    lineLength = fieldDim / 4;

    lv_style_copy(&lineStyle, &lv_style_plain);
    lineStyle.line.width = 6;
    lineStyle.line.opa = LV_OPA_100;
    lineStyle.line.color = LV_COLOR_MAKE(0, 255, 0);
    lv_obj_set_style(line, &lineStyle);

    /**
     * Status Label
     */
    statusLabel = lv_label_create(container, NULL);
    lv_style_copy(&textStyle, &lv_style_plain);
```

```
208     textStyle.text.color = LV_COLOR_WHITE;
209     textStyle.text.opa = LV_OPA_100;
210     lv_obj_set_style(statusLabel, &textStyle);
211     lv_label_set_text(statusLabel, "No Odom Data Provided");
212     lv_obj_align(statusLabel, container, LV_ALIGN_CENTER,
213                  -lv_obj_get_width(container) / 2 + (lv_obj_get_width(container) -
                     ↪  fieldDim) / 2,
214                  0);
215
216     /**
217      * Reset Button
218      */
219     {
220         lv_obj_t * btn = lv_btn_create(container, NULL);
221         lv_obj_set_size(btn, 100, 40);
222         lv_obj_align(
223             btn, NULL, LV_ALIGN_IN_TOP_MID,
224             -lv_obj_get_width(container) / 2 + (lv_obj_get_width(container) - fieldDim)
                 ↪  / 2, 0);
225         lv_obj_set_free_ptr(btn, this);
226         lv_btn_set_action(btn, LV_BTN_ACTION_PR, resetAction);
227
228         /**
229          * Button Style
230          */
231         lv_style_copy(&resetRel, &lv_style_btn_tgl_rel);
232         resetRel.body.main_color = mainColor;
233         resetRel.body.grad_color = mainColor;
234         resetRel.body.border.color = LV_COLOR_WHITE;
235         resetRel.body.border.width = 2;
236         resetRel.body.border.opa = LV_OPA_100;
237         resetRel.body.radius = 2;
238         resetRel.text.color = LV_COLOR_WHITE;
239
240         lv_style_copy(&resetPr, &resetRel);
241         resetPr.body.main_color = LV_COLOR_WHITE;
242         resetPr.body.grad_color = LV_COLOR_WHITE;
243         resetPr.text.color = mainColor;
244
245         lv_btn_set_style(btn, LV_BTN_STYLE_REL, &resetRel);
246         lv_btn_set_style(btn, LV_BTN_STYLE_PR, &resetPr);
247
248         /**
249          * Reset Button Label
250          */
251         lv_obj_t * label = lv_label_create(btn, NULL);
252         lv_obj_set_style(label, &textStyle);
253         lv_label_set_text(label, "Reset");
254     }
255     printf("Made an OdomDebug\n");
256 }
257
258 OdomDebug::~OdomDebug() { lv_obj_del(container); }
259
```

```cpp
/**
 * Sets the function to be called when a tile is pressed
 * @param callback a function that sets the odometry state
 */
void OdomDebug::setStateCallback(std::function<void(state_t state)> callback)
{
    stateFnc = callback;
}

/**
 * Sets the function to be called when the reset button is pressed
 * @param callback a function that resets the odometry and sensors
 */
void OdomDebug::setResetCallback(std::function<void()> callback) { resetFnc = callback; }

/**
 * Sets the position of the robot in QUnits and puts the sensor data on the
 * display
 * @param state   robot state - x, y, theta
 * @param sensors encoder information - left, right, middle (optional)
 */
void OdomDebug::setData(state_t state, sensors_t sensors)
{

    // position in court units
    double c_x = state.x.convert(court);
    double c_y = state.y.convert(court);
    double c_theta = state.theta.convert(radian);

    // place point on field
    lv_obj_set_pos(led, (c_x * fieldDim) - lv_obj_get_width(led) / 2,
                   (c_y * fieldDim) - lv_obj_get_height(led) / 2 - 1);

    // move start and end of line
    linePoints[0] = {(int16_t)((c_x * fieldDim)), (int16_t)((c_y * fieldDim) -
    ↪  (lineWidth / 2))};
    double newY = lineLength * sin(c_theta);
    double newX = lineLength * cos(c_theta);
    linePoints[1] = {(int16_t)(newX + linePoints[0].x), (int16_t)(newY +
    ↪  linePoints[0].y)};

    lv_line_set_points(line, linePoints.data(), linePoints.size());
    lv_obj_invalidate(line);

    std::string text =
        "X: " + std::to_string(std::round(state.x.convert(inch) * 1000) /
        ↪  1000).substr(0, 5) +
        " in\n" +
        "Y: " + std::to_string(std::round(state.y.convert(inch) * 1000) /
        ↪  1000).substr(0, 5) +
        " in\n" + "Theta: " +
        std::to_string(std::round(state.theta.convert(degree) * 1000) / 1000).substr(0,
        ↪  5) +
```

```
308         " deg\n" + "Left: " + std::to_string(sensors.left).substr(0, 4) + " deg\n" +
309         "Right: " + std::to_string(sensors.right).substr(0, 4) + " deg\n";
310     if (sensors.hasMiddle)
311     {
312         text = text + "Middle: " + std::to_string(sensors.middle).substr(0, 4) + " deg";
313     }
314 
315     lv_label_set_text(statusLabel, text.c_str());
316     lv_obj_align(statusLabel, container, LV_ALIGN_CENTER,
317                  -lv_obj_get_width(container) / 2 + (lv_obj_get_width(container) -
                     ↪  fieldDim) / 2,
318                  25);
319 }
320 
321 /**
322  * Sets odom state when tile is pressed
323  * Decodes tile ID to find position
324  */
325 lv_res_t OdomDebug::tileAction(lv_obj_t * tileObj)
326 {
327     OdomDebug * that = static_cast<OdomDebug *>(lv_obj_get_free_ptr(tileObj));
328     int num = lv_obj_get_free_num(tileObj);
329     int y = num / 6;
330     int x = num - y * 6;
331     if (that->stateFnc)
332         that->stateFnc({x * tile + 0.5_tl, y * tile + 0.5_tl, 0_deg});
333     else
334         std::cout << "OdomDebug: No tile action callback provided";
335     return LV_RES_OK;
336 }
337 
338 /**
339  * Reset Sensors and Position
340  */
341 lv_res_t OdomDebug::resetAction(lv_obj_t * btn)
342 {
343     OdomDebug * that = static_cast<OdomDebug *>(lv_obj_get_free_ptr(btn));
344     if (that->resetFnc)
345         that->resetFnc();
346     else
347         std::cout << "OdomDebug: No reset action callback provided";
348     return LV_RES_OK;
349 }
```

2.6 src/movement/Drivetrain.cpp

```cpp
/**
 * Drivetrain.cpp
 *
 * This file contains the definitions of the Drivetrain class.
 * The Drivetrain class handles almost everthing relating to the
 * drivetrain: motor control, settings (like max speed), basic
 * movement methods (like tank or arcade), more advanced movement
 * methods (like PID to point, path following, and motion
 * profiling), and more.
 */
#include "main.h" // gives access to Drivetrain and other dependancies

/* ------------------------------------------------------------ */
/*                      Private Information                     */
/* ------------------------------------------------------------ */

/* --------------------- Motor References --------------------- */
Motor & Drivetrain::mmtrLeftFront = def::mtr_dt_left_front;
Motor & Drivetrain::mmtrRightFront = def::mtr_dt_right_front;
Motor & Drivetrain::mmtrRightBack = def::mtr_dt_right_back;
Motor & Drivetrain::mmtrLeftBack = def::mtr_dt_left_back;

/* ---------------------- Okapi Chassis ---------------------- */
std::shared_ptr<ChassisController> Drivetrain::mchassis =
    ChassisControllerBuilder()
        .withMotors({Drivetrain::mmtrLeftFront, Drivetrain::mmtrLeftBack},
                    {Drivetrain::mmtrRightFront, Drivetrain::mmtrRightBack})
        .withDimensions(AbstractMotor::gearset::green,
                        {{def::DRIVE_WHEEL_DIAMETER, def::DRIVE_OFFSET}, imev5GreenTPR})
        .build(); // chassis object for using Pathfilder through okapi

/* ------------------------------------------------------------ */
/*                     Protected Information                    */
/* ------------------------------------------------------------ */

/* --------------------- Initial Settings --------------------- */
double Drivetrain::mmaxSpeed = def::SET_DT_MAX;
bool Drivetrain::menabled = true;

/* -------------------- Simple Follow Data -------------------- */
double Drivetrain::mlastLookIndex = 0; // index of the last lookahead point
double Drivetrain::mlastPartialIndex =
    0; // fractional index of where the last lookahead point was on the segment

/* -------------------- Odometry Accessors -------------------- */
OdomState Drivetrain::getState() // get position as OdomState
{
    return CustomOdometry::getState();
}
QLength Drivetrain::getXPos() { return CustomOdometry::getX(); }
QLength Drivetrain::getYPos() { return CustomOdometry::getY(); }
QAngle Drivetrain::getTheta() { return CustomOdometry::getTheta(); }
```

```cpp
ExtendedPoint Drivetrain::getPoint() // get position as ExtendedPoint
{
    return ExtendedPoint(getXPos(), getYPos(), getTheta());
}

/* ------------------------- Helpers ------------------------- */
QAngle Drivetrain::angleToPoint(
    const Point & itargetPoint) // calculates the field centric direction to the itargetPoint from
                                // the robot's current position
{
    return (atan((getYPos().convert(inch) - itargetPoint.y.convert(inch)) /
                 (getXPos().convert(inch) - itargetPoint.x.convert(inch))) +
            (getXPos() > itargetPoint.x ? M_PI : 0)) *
           radian;
}
std::optional<double> Drivetrain::findIntersection(
    ExtendedPoint istart, ExtendedPoint iend,
    const double & ilookDistIn) // looks for interections between the line segment created by the
                                // two points (istart and iend), and the circle around the robot
                                // with radius ilookDistIn (lookahead circle)
{
    ExtendedPoint d = iend - istart; // differece vector
    ExtendedPoint f = istart - getPoint(); // robot position relative to the start of the segment

    double a = d.dot(d).convert(inch); // set up quadratic
    double b = 2 * f.dot(d).convert(inch);
    double c = f.dot(f).convert(inch) - ilookDistIn * ilookDistIn;

    double discriminant = b * b - 4 * a * c; // used to make sure it doesn't sqrt(a negative number)
    if (discriminant >= 0)
    {
        discriminant = sqrt(discriminant);

        double t1 = (-b - discriminant) / (2 * a); // solution 1
        double t2 = (-b + discriminant) / (2 * a); // solution 2

        if (t2 >= 0 && t2 <= 1) // t2 is always farther along the segment, so return t2 first
        {
            return t2;
        }
        else if (t1 >= 0 && t1 <= 1) // then t1
        {
            return t1;
        }
    }

    return {}; // no intersections
}
```

```
ExtendedPoint Drivetrain::findLookahead(
    SimplePath ipath, const double & ilookDistIn) // looks for the intersection point between the
                                                  // lookahead circle and the SimplePath, ipath
{
    ExtendedPoint currentPos = getPoint();
    int lastIntersectIndex = 0;

    if (currentPos.dist(ipath.last()).convert(inch) <=
        ilookDistIn) // if the last point is within range, return
    {
        return ipath.last();
    }

    for (int i = mlastLookIndex; i < ipath.size() - 1;
         i++) // searchs through the whole path starting at the index of the previous lookhead point
    {
        std::optional<double> t_partialIndex = findIntersection(
            ipath.at(i), ipath.at(i + 1),
            ilookDistIn); // finds the partial index of the intersection in the range [0, 1)
        if (t_partialIndex.has_value() &&
            (i > mlastLookIndex ||
             t_partialIndex > mlastPartialIndex)) // if there is an intersection farther along the
                                                  // path than the last point
        {
            mlastLookIndex = i;
            mlastPartialIndex = t_partialIndex.value();

            if (lastIntersectIndex > 0) // if this is the second intersection, the loop can exit
            {
                break;
            }

            lastIntersectIndex = i; // if this is the first intersection found, record it
        }

        if (lastIntersectIndex > 0 &&
            ipath.at(i).dist(ipath.at(lastIntersectIndex)).convert(inch) >=
                ilookDistIn *
                    2) // if it is searching for intersections farther than the diameter of a
                       // lookahead circle, and it has alread found a point, exit the loop. It is
                       // impossible for there to be a second lookahead point more than 2 *
                       // (lookhead distance) away from the first lookahead point
        {
            break;
```

```
145         }
146     }
147 
148     ExtendedPoint segmentStart = ipath.at(mlastLookIndex);
149     def::display.setMiscData(
150         1, "start: " + segmentStart.string() +
151                "\nvec: " + (ipath.at(mlastLookIndex + 1) - segmentStart).string() +
152                "\npartlIndx: " + std::to_string(mlastPartialIndex) + "\nscaled: " +
153                ((ipath.at(mlastLookIndex + 1) - segmentStart) *
                 ↪  mlastPartialIndex).string());
154     return segmentStart +
155            (ipath.at(mlastLookIndex + 1) - segmentStart) *
156                mlastPartialIndex; // calculates the location of the lookahead point by
                 ↪  getting the
157                                   // vector from the start of the segment to the end of
                                    ↪  the segment,
158                                   // multiplying that by the fractional index of the
                                    ↪  lookahead
159                                   // point, and adding that the the starting point of
                                    ↪  the segment
160 }
161 
162 /* ------------------------------------------------------------ */
163 /*                      Public Information                      */
164 /* ------------------------------------------------------------ */
165 std::shared_ptr<AsyncMotionProfileController> Drivetrain::mprofiler =
166     AsyncMotionProfileControllerBuilder()
167         .withLimits({def::DRIVE_MAX_SPEED.convert(mps),
            ↪  def::DRIVE_MAX_ACCEL.convert(mps2), 8.0})
168         .withOutput(Drivetrain::mchassis)
169         .buildMotionProfileController(); // okapi motion profile controller with
            ↪  measured constants
170 
171 /* --------------------- Getters/Setters --------------------- */
172 double Drivetrain::getMaxSpeed() { return mmaxSpeed; }
173 void Drivetrain::setMaxSpeed(double imaxSpeed) { mmaxSpeed = imaxSpeed; }
174 
175 bool Drivetrain::isEnabled() { return menabled; }
176 void Drivetrain::enable() // allows movements to be startable
177 {
178     menabled = true;
179 }
180 void Drivetrain::disable() // stops active movemnts
181 {
182     menabled = false;
183     moveTank(0, 0, false);
184 }
185 
186 void Drivetrain::checkNextAsync(
187     const double & ierror,
188     std::vector<AsyncAction> & iactions) // checks if the next AsyncAction should
        ↪  execute, and
189                                          // executes it (and removes it from the list)
                                           ↪  if it should
```

```
{
    if (iactions.size()) // if there is at least one action to execute
    {
        const AsyncAction & nextAction = iactions.at(0);
        if (ierror < nextAction.merror) // if the robot is close enough to the target
        {
            nextAction.maction(); // execute the action
            iactions.erase(iactions.begin()); // remove the action, having already
            ↪  executed it
        }
    }
}

/* ---------------------- Basic Movement ---------------------  /
 * These "basic" motions are lower level fuctions mostly just intended
 * to prevent the call to each motor individually in more advanced
 * motions, to keep the code cleaner.
 *
 * "Saturation" is when the motor inputs are higher than their max
 * speed, which makes the motor go at max speed. This can cause problems,
 * however, when the motors are all working together to follow a
 * specific motion, because one motor might be going at exactly the
 * intended speed, but another motor might be saturated, so it doesn't
 * go at the right speed, making the robot follow the wrong motion.
 * To account for this, the basic movment methods have a variable,
 * idesaturate, that, when true, scales all motor values down so they
 * fit within the motors capabilities.
 */
void Drivetrain::moveIndependant(
    double ileftFront, double irightFront, double irightBack, double ileftBack,
    const bool idesaturate) // moves each motor {lf, rf, rb, lb} in range [-1,1]
{
    if (idesaturate) // desaturates values
    {
        std::array<double, 4> motor_values =
            util::scaleToFit<4>(mmaxSpeed, {ileftFront, irightFront, irightBack,
            ↪  ileftBack});
        ileftFront = motor_values[0];
        irightFront = motor_values[1];
        irightBack = motor_values[2];
        ileftBack = motor_values[3];
    }
    // moves all of the motors by voltage
    mmtrLeftFront.moveVoltage(12000 * ileftFront);
    mmtrRightFront.moveVoltage(12000 * irightFront);
    mmtrRightBack.moveVoltage(12000 * irightBack);
    mmtrLeftBack.moveVoltage(12000 * ileftBack);
}
void Drivetrain::moveTank(double ileft, double iright,
                          const bool idesaturate) // spins the left side and right side
                          ↪  motors at
                                                  // certian speeds in range [-1,1]
{
    if (idesaturate) // desaturates values
```

```
241     {
242         std::array<double, 2> motor_values = util::scaleToFit<2>(mmaxSpeed, {ileft,
            ↪  iright});
243         ileft = motor_values[0];
244         iright = motor_values[1];
245     }
246     Drivetrain::moveIndependant(
247         ileft, iright, iright, ileft,
248         false); // don't try to desaturate, because the values have already been
            ↪  desaturated
249 }
250 void Drivetrain::moveArcade(
251     double iforward, double istrafe, double iturn,
252     const bool idesaturate) // moves the robot with arcade-style inputs in range [-1,1]
253 {
254     if (idesaturate) // desaturates values
255     {
256         std::array<double, 4> motor_values = {
257             iforward + istrafe + iturn, iforward - istrafe - iturn, iforward + istrafe
                ↪  - iturn,
258             iforward - istrafe + iturn};
259         util::scaleToFit<4>(mmaxSpeed, motor_values); // modifies reference to
            ↪  motor_values
260         Drivetrain::moveIndependant(motor_values[0], motor_values[1], motor_values[2],
261                                     motor_values[3]); // moves the motors from within
                                     ↪  the if to
262                                                       // prevent the need to copy values
263     }
264     else
265     {
266         Drivetrain::moveIndependant(iforward + istrafe + iturn, iforward - istrafe -
            ↪  iturn,
267                                     iforward + istrafe - iturn, iforward - istrafe +
                                     ↪  iturn,
268                                     false); // don't desaturate
269     }
270 }
271
272 /* ------------------ Intermediate Movement ----------------- */
273 void Drivetrain::moveInDirection(
274     QAngle idirection, const bool ifieldCentric, double imagnitude, double itheta,
275     const bool idesaturate) // moves the robot with a certain speed in a certain
        ↪  direction, while
276                             // turning a certain amount
277 {
278     if (ifieldCentric) // if the direction is in reference to the field
279     {
280         idirection -= Drivetrain::getTheta(); // changes the direction the robot should
            ↪  go in based
281                                               // on its field centric rotation
282     }
283     idirection = util::wrapQAngle(idirection); // fits idirection into [0, 360)
284     util::chop<double>(0, 1, imagnitude); // caps magnitude
285     util::chop<double>(-1, 1, itheta); // caps itheta
```

```
286
287     Drivetrain::moveArcade(
288         imagnitude * cos(idirection.convert(radian)), imagnitude *
            ↪ sin(idirection.convert(radian)),
289         itheta, idesaturate); // move in the direction of the vector, and turn the
            ↪ specified amount
290 }
291
292 /* ------------------ Move to Point Methods ------------------ /
293  * Because these methods have a target, they need to be run in a loop
294  * to constantly re-evaluate how fast the robot should be going. To
295  * do this, there are PID and Slew calculations being done for driving
296  * straight and turning in each of the methods. This is easy to do
297  * because of the PID and Slew classes.
298  *
299  * Each method has custom tuned default PID/Slew values, but they
300  * can be modified on a per-motion basis when they are called.
301  */
302 void Drivetrain::strafeToPoint(
303     ExtendedPoint itarget, std::vector<AsyncAction> iactions, PID imagnitudePID, PID
        ↪ iturnPID,
304     Slew imagnitudeSlew,
305     Slew iturnSlew) // drives in a stright line to the point while turning using set
        ↪ PID/Slew gains,
306                     // and executing the AsyncActions at the right times
307 {
308     enable(); // make sure the action can run
309     while (menabled && (!imagnitudePID.isSettled() || !iturnPID.isSettled()))
310     {
311         double inToPoint =
312             OdomMath::computeDistanceToPoint(itarget, Drivetrain::getState())
313                 .convert(inch); // calc inches to target point. itarget can be passed as
314                                 // okapi::Point, because ExtendedPoint inherits from
                                    ↪ okapi::Point
315         double degToPoint = util::wrapDeg180(
316             (itarget.theta - Drivetrain::getTheta())
317                 .convert(degree)); // calc the angle to the point in the range [-180,
                    ↪ 180) to always
318                                    // turn the right direction
319         def::display.setMiscData(1, std::to_string(degToPoint));
320
321         Drivetrain::moveInDirection(Drivetrain::angleToPoint(itarget), true,
322                                     imagnitudeSlew.iterate(imagnitudePID.iterate(inToPo
                                        ↪ int)),
323                                     iturnSlew.iterate(iturnPID.iterate(degToPoint)),
                                        ↪ true);
324
325         Drivetrain::checkNextAsync(
326             inToPoint,
327             iactions); // executes the next action if availible, and removes it from
                ↪ the list
328
329         pros::delay(20);
330     }
```

```
}

void Drivetrain::straightToPoint(
    ExtendedPoint itarget, std::vector<AsyncAction> iactions, QLength inoTurnRange,
    double iturnWeight, PID imagnitudePID, PID iturnPID, Slew imagnitudeSlew,
    Slew iturnSlew) // drives to the point without strafing using set PID/Slew gains,
    ↪  and executing
                    // the AsyncActions at the right times
{
    const double noTurnRangeIn = inoTurnRange.convert(inch);

    enable(); // make sure the action can run
    while (menabled && !imagnitudePID.isSettled())
    {
        QAngle angleToPoint = util::wrapQAngle180(
            Drivetrain::angleToPoint(itarget) -
            Drivetrain::getTheta()); // how much the robot needs to turn to face the
            ↪  point
        double inToPoint = OdomMath::computeDistanceToPoint(itarget,
        ↪  Drivetrain::getState())
                               .convert(inch); // how far the robot is away from the
                               ↪  target
        double inForward =
            inToPoint *
            cos(angleToPoint.convert(radian)); // how far the robot needs to drive
            ↪  straight (no
                                               // turning) to get as close to the
                                               ↪  target as possible

        double forward = imagnitudeSlew.iterate(
            imagnitudePID.iterate(inForward)); // calculates value from PID fed into
            ↪  Slew
        util::chop<double>(-mmaxSpeed, mmaxSpeed,
                           forward); // limits the values in [-mmaxSpeed, mmaxSpeed]

        double turn;
        if (inToPoint > noTurnRangeIn) // if the robot is far enough away from the
        ↪  target
        {
            turn = iturnSlew.iterate(iturnPID.iterate(
                angleToPoint.convert(degree))); // calculates value from PID fed into
                ↪  Slew
            util::chop<double>(-mmaxSpeed, mmaxSpeed, turn);
        }
        else
        {
            turn = 0; // don't turn when too close to the target
        }

        if (abs(turn) == mmaxSpeed) // if the robot is turning at max speed (which
        ↪  means it must be
                                    // far off target)
        {
```

```
374             turn *= iturnWeight; // increase the amount to turn, so that it turns
                ↪  faster as a result
375                                  // of forward getting scaled down in moveArcade
376         }
377
378         Drivetrain::checkNextAsync(
379             inToPoint,
380             iactions); // executes the next action if availible, and removes it from
                ↪  the list
381
382         Drivetrain::moveArcade(forward, 0, turn, true);
383
384         pros::delay(20);
385     }
386 }
387
388 void Drivetrain::arcStraightToPoint(
389     ExtendedPoint itarget, std::vector<AsyncAction> iactions, double iweightModifier,
390     QLength inoTurnRange, PID imagnitudePID, PID iturnPID, Slew imagnitudeSlew,
391     Slew iturnSlew) // drive in an "arc" (doesn't follow a path, just approximates an
    ↪  arc) using set
392                     // PID/Slew gains, and executing the AsyncActions at the right times
393 {
394     const double noTurnRangeIn = inoTurnRange.convert(inch);
395
396     enable(); // make sure the action can run
397     while (menabled && !imagnitudePID.isSettled())
398     {
399         double theta = util::wrapRad(
400             2 * (Drivetrain::getTheta().convert(radian) -
401                  abs(atan2(
402                      (Drivetrain::getYPos() - itarget.y).convert(inch),
403                      (Drivetrain::getXPos() - itarget.x)
404                          .convert(inch))))); // calculates how much the robot should
                        ↪  end up turning
405         double radius = abs(
406             hypot((getXPos() - itarget.x).convert(inch), (getYPos() -
                ↪  itarget.y).convert(inch)) /
407             2 / sin(theta / 2)); // calculates the radius of the arc
408         double targetIn = theta * radius; // how far the robot needs to go (arc length)
409         double turnWeight =
410             iweightModifier /
411             radius; // how aggressively the robot needs to turn to approximate the arc
412
413         QAngle angleToPoint =
414             Drivetrain::angleToPoint(itarget) - Drivetrain::getTheta(); // direction of
                ↪  target
415         double inToPoint = OdomMath::computeDistanceToPoint(itarget, getState())
416                                .convert(inch); // distance of target
417         double inForward =
418             inToPoint *
419             cos((angleToPoint)
420                     .convert(radian)); // distance to perpendicular line intersecting
                    ↪  target
```

```
421
422         double forward = imagnitudeSlew.iterate(
423             imagnitudePID.iterate(inForward)); // calculates value from PID fed into
                ↪  Slew
424         util::chop<double>(-mmaxSpeed, mmaxSpeed,
425                            forward); // limits the values in [-mmaxSpeed, mmaxSpeed]
426
427         double turn;
428         if (inToPoint > noTurnRangeIn) // if the robot is far enough away from the
            ↪  target
429         {
430             turn = iturnSlew.iterate(iturnPID.iterate(
431                 angleToPoint.convert(degree))); // calculates value from PID fed into
                    ↪  Slew
432             util::chop<double>(-mmaxSpeed, mmaxSpeed, turn);
433         }
434         else
435         {
436             turn = 0; // don't turn when too close to the target
437         }
438
439         if (abs(turn) == mmaxSpeed) // if the robot is turning at max speed (which
            ↪  means it must be
440                                     // far off target)
441         {
442             turn *= turnWeight; // increase the amount to turn, so that it turns faster
                ↪  as a result
443                                 // of forward getting scaled down in moveArcade
444         }
445
446         Drivetrain::checkNextAsync(
447             inToPoint,
448             iactions); // executes the next action if availible, and removes it from
                ↪  the list
449
450         Drivetrain::moveArcade(forward, 0, turn, true);
451
452         pros::delay(20);
453     }
454 }
455
456 /* ------------------ Path Following Methods -----------------  /
457  * simpleFollow uses the concept from "Pure Pursuit" of using a "lookahead circle" to
    ↪  follow a path.
458  * The idea is that, when given a line, the robot will figure out how to follow it
    ↪  smoothly. It does
459  * this by checking for points on the line that are a certain distance away from the
    ↪  robot, and
460  * moving in the direction of whichever point it sees that is farthest on the line.
    ↪  Another way  to
461  * picture this, is the robot has a circle (the lookahead circle) around it with the
    ↪  radius being
462  * the "lookahead distance". The robot is always trying to drive to intersections
    ↪  between this
```

```
463  * circle, and the path (a.k.a. the lookahead point).
464  *
465  * The robot goes at full speed to the lookahead point until it gets to the end, where
     ↪  it settles
466  * with PID.
467  */
468 void Drivetrain::simpleFollow(
469     SimplePath ipath, QLength ilookDist, std::vector<AsyncAction> iactions, PID
         ↪  imagnitudePID,
470     PID iturnPID, Slew imagnitudeSlew,
471     Slew iturnSlew) // follows the path, ipath using set lookahead distance (ilookDist)
         ↪  and PID/Slew
472                     // gains while executing the AsyncActions at the right times (only
                        ↪  on the last
473                     // segment)
474 {
475     double lookDistIn = ilookDist.convert(inch);
476     ExtendedPoint lookPoint = ipath.at(0);
477
478     double magnitude = 1; // the robot will always go full speed until the end is near
479     bool reachedEnd = false;
480
481     mlastLookIndex = 0;
482     mlastPartialIndex = 0;
483
484     enable(); // make sure the action can run
485     while (menabled && (!imagnitudePID.isSettled() || !iturnPID.isSettled()) ||
         ↪  !reachedEnd)
486     {
487         if (!reachedEnd &&
488             mlastLookIndex ==
489                 ipath.size() - 2) // detects if the robot should be going to the last
                     ↪  point
490         {
491             reachedEnd = true;
492             lookPoint = ipath.last();
493         }
494
495         if (!reachedEnd)
496         {
497             lookPoint = findLookahead(ipath, lookDistIn); // find the next lookahead
498         }
499         else
500         {
501             double inToPoint =
502                 OdomMath::computeDistanceToPoint(lookPoint, Drivetrain::getState())
503                     .convert(
504                         inch); // calc inches to target point. itarget can be passed as
505                                // okapi::Point, because ExtendedPoint inherits from
                                   ↪  okapi::Point
506
507             Drivetrain::checkNextAsync(
508                 inToPoint,
```

```
509                 iactions); // executes the next action if availible, and removes it
                    ↪ from the list
510
511             magnitude = imagnitudePID.iterate(inToPoint); // how fast the robot should
                ↪ be moving
512         }
513
514         double degToPoint = util::wrapDeg180(
515             (lookPoint.theta - Drivetrain::getTheta())
516                 .convert(degree)); // calc the angle to the point in the range [-180,
                    ↪ 180) to always
517                                    // turn the right direction
518
519         moveInDirection(Drivetrain::angleToPoint(lookPoint), true,
520                         imagnitudeSlew.iterate(magnitude),
521                         iturnSlew.iterate(iturnPID.iterate(degToPoint)), true);
522
523         pros::delay(20);
524     }
525 }
526
527 /* ---------------------- Motion Profiling -------------------- */
528 void Drivetrain::generatePathToPoint(
529     PathfinderPoint ipoint,
530     const std::string & iname) // use Pathfinder through okapi to make a motion profile
531 {
532     ipoint.y = -ipoint.y;
533     ipoint.theta = -ipoint.theta;
534     mprofiler->generatePath({{0_ft, 0_ft, 0_deg}, ipoint}, iname);
535 }
536 void Drivetrain::followPathfinder(const std::string & iname, bool ibackwards,
537                                   bool imirrored) // follow Pathfinder path through
                                      ↪ okapi
538 {
539     mprofiler->setTarget(iname, ibackwards, imirrored);
540 }
541 void Drivetrain::followTraj(Trajectory & itraj) // follow trajectory loaded from sd card
542 {
543     const double startLeft = mmtrLeftFront.getPosition();
544     const double startRight = mmtrRightFront.getPosition();
545
546     if (itraj.isDone()) // if the path is done before execution, reset
547     {
548         itraj.reset();
549     }
550     while (!itraj.isDone()) // execute until the path is done
551     {
552         std::pair<double, double> values = itraj.iterate(
553             (mmtrLeftFront.getPosition() - startLeft) *
                ↪ def::DRIVE_WHEEL_CIRCUMFERENCE_IN / 360,
554             (mmtrRightFront.getPosition() - startRight) *
                ↪ def::DRIVE_WHEEL_CIRCUMFERENCE_IN /
555                 360); // iterate through the profile passing the distance each side has
                    ↪ gone so far
```

```
556
557         moveTank(values.first, values.second, false);
558
559         pros::delay(10);
560     }
561 }
```

2.7 src/movement/paths/ProfileStep.cpp

```cpp
/**
 * ProfileStep.cpp
 *
 * ProfileStep is used for organizing the information parsed from motion profiles stored on the sd
 * card, calculated by the publically availible GitHub repository, TrajectoryLib by Team254 (FRC
 * Team 254, The Cheesy Poofs), found here: https://github.com/Team254/TrajectoryLib. The
 * trajectories are calculated on a computer, and stored on the sd card for the robot to use. Each
 * time step of the profile is read from the sd card, and stored in an instance of ProfileStep by
 * the Trajectory class.
 */
#include "main.h" // gives access to ProfileStep declaration and other dependencies

const std::string ProfileStep::getString() // returns the ProfileStep formatted as a std::string
                                           // without changing anything (const)
{
    return std::to_string(pos) + " " + std::to_string(vel) + " " + std::to_string(acc) + " " +
           std::to_string(jerk) + " " + std::to_string(heading) + " " + std::to_string(dt) + " " +
           std::to_string(x) + " " + std::to_string(y);
}
```

2.8 src/movement/paths/SimplePath.cpp

```cpp
/**
 * SimplePath.cpp
 *
 * SimplePath is a simple struct that has a list of points
 * on a path represented by ExtendedPoints in a std::vector.
 * This is used for path following by the Drivetrain class.
 */
#include "main.h" // gives access to SimplePath and other dependencies

ExtendedPoint SimplePath::at(size_t iindex) // returns the point at the index
{
    return mpoints.at(iindex);
}
ExtendedPoint SimplePath::last() // returns the point at the end
{
    return mpoints.back();
}
int SimplePath::size() // returns the length of the path
{
    return mpoints.size();
}
```

2.9 src/movement/paths/Trajectory.cpp

```cpp
/**
 * Trajectory.cpp
 *
 * This file contains the definitions of the Trajectory class. The Trajectory class reads and stores
 * motion profile information from the sd card. Motion profiles stored on the sd card are calculated
 * by the publically availible GitHub repository, TrajectoryLib by Team254 (FRC Team 254, The Cheesy
 * Poofs), found here: https://github.com/Team254/TrajectoryLib. The trajectories are calculated on
 * a computer, and stored on the sd card for the robot to use. Each time step of the profile is read
 * from the sd card, and stored in an instance of ProfileStep by the Trajectory class.
 *
 * The paths are intended to be executed by the Drivetrain class, but are not used in programming
 * skills.
 */
#include "main.h" // gives access to Trajectory and other dependencies

/* ------------------------------------------------------------ */
/*                      Public Information                      */
/* ------------------------------------------------------------ */
Trajectory::Trajectory(const char * ifileName, double ikP, double ikD, double ikV, double ikA)
    : mkP(ikP), mkD(ikD), mkV(ikV), mkA(ikA), mstepNumber(0), mlastErrorL(0.0),
      mlastErrorR(0.0) // constructor that specifies the file with the trajectory, and the gains for
                       // following the trajectory
{
    FILE * file; // creates a file object to be used later
    if (pros::usd::is_installed()) // checks if the sd card is installed before trying to access it
    {
        file = fopen(ifileName, "r"); // open the file
        if (file) // makes sure the file was opened correctly
        {
            char name[256];
            fgets(name, 255, file); // put the name of the trajectory in a char array
            mname = name;
            mname = mname.substr(0, mname.length() - 1); // chop off \n

            fscanf(file, "%i", &mlength); // store the number of steps

            mleftSteps = new ProfileStep[mlength]; // dynamically allocate left and right profiles
                                                   // based on the length of the trajectory
            mrightSteps = new ProfileStep[mlength];

            for (int i = 0; i < mlength; i++) // fill left profile array from sd card
```

```
            {
                float pos, vel, acc, jerk, heading, dt, x, y;
                fscanf(file, "%f %f %f %f %f %f %f %f", &pos, &vel, &acc, &jerk,
                ↪  &heading, &dt, &x,
                       &y);
                mleftSteps[i] = {pos, vel, acc, jerk, heading, dt, x, y};
            }
            for (int i = 0; i < mlength; i++) // fill right profile array from sd card
            {
                float pos, vel, acc, jerk, heading, dt, x, y;
                fscanf(file, "%f %f %f %f %f %f %f %f", &pos, &vel, &acc, &jerk,
                ↪  &heading, &dt, &x,
                       &y);
                mrightSteps[i] = {pos, vel, acc, jerk, heading, dt, x, y};
            }
        }
        else
        {
            std::cout << "\"" << ifileName << "\" file is null"
                      << std::endl; // output to the terminal if the file is null
        }

        fclose(file);
    }
    else // if the sd card is not installed, create empty arrays and send a message to
    ↪  the terminal
    {
        mleftSteps = new ProfileStep[1];
        mrightSteps = new ProfileStep[1];
        std::cout << "no sd card inserted" << std::endl;
    }
}
Trajectory::~Trajectory() // destructor to clean up heap variables
{
    delete[] mleftSteps;
    delete[] mrightSteps;
}

int Trajectory::getLength() { return mlength; }
std::string Trajectory::getName() { return mname; }
void Trajectory::reset() { mstepNumber = 0; }
std::pair<ProfileStep, ProfileStep>
Trajectory::getStep(int istepNumber) // return the left and right ProfileSteps at a
↪  given point
{
    if (istepNumber >
        sizeof(mleftSteps) / sizeof(ProfileStep *)) // if the stepnumber is out of range
    {
        std::cout << "index out of bounds" << std::endl;
    }
    return std::pair<ProfileStep, ProfileStep>(mleftSteps[istepNumber],
    ↪  mrightSteps[istepNumber]);
}
```

```
90  void Trajectory::setGains(const double ikP, const double ikD, const double ikV, const
    ↪  double ikA)
91  {
92      mkP = ikP;
93      mkD = ikD;
94      mkV = ikV;
95      mkA = ikA;
96  }
97  bool Trajectory::isDone() // checks to see if the trajectory is done
98  {
99      return mlength <= mstepNumber;
100 }
101
102 std::pair<double, double>
103 Trajectory::iterate(double ileftDistSoFar,
104                     double irightDistSoFar) // goes through one iteration of the
                        ↪  FEEDFORWARD loop
105                                             // based on how far the left and right
                                                ↪  wheels have gone
106 {
107     if (mstepNumber < mlength)
108     {
109         double errorL = mleftSteps[mstepNumber].pos -
110                         ileftDistSoFar; // diference between where the wheels should
                            ↪  be, and where
111                                         // they are for feedback control
112         double errorR = mrightSteps[mstepNumber].pos - irightDistSoFar;
113         double errorVelL = ((errorL - mlastErrorL) / mleftSteps[mstepNumber].dt -
114                             mleftSteps[mstepNumber].vel); // velocity error
115         double errorVelR =
116             ((errorR - mlastErrorR) / mrightSteps[mstepNumber].dt -
                ↪  mrightSteps[mstepNumber].vel);
117
118         double resultL =
119             mkP * errorL + mkD * errorVelL + mkV * mleftSteps[mstepNumber].vel +
120             mkA * mleftSteps[mstepNumber].acc; // Kp*ep(t) + Kd*ev(t) + Kv*rv(t) +
                ↪  Ka*ra(t)
121         double resultR = mkP * errorR + mkD * errorVelR + mkV *
            ↪  mrightSteps[mstepNumber].vel +
122                          mkA * mrightSteps[mstepNumber].acc;
123
124         mlastErrorL = errorL; // store error for derivative calculation next time
125         mlastErrorR = errorR;
126
127         mstepNumber++;
128
129         return {resultL, resultR}; // return the necessary motor movements
130     }
131     else
132     {
133         return {0, 0}; // return 0 power for both motors, because the path has finished
            ↪  executing
134     }
135 }
```

2.10 src/stateMachines/BallControlStateMachine.cpp

```cpp
/**
 * BallControlStateMachine.cpp
 *
 * This file contains the definitions of the BallControlStateMachine class.
 * BallControlStateMachine inherits from VStateMachine, and
 * it is responsible for controlling all of the ball manipulators
 * (intake, indexer, filter, and flywheel).
 *
 * The intake, indexer, and flywheel all have their own mini
 * state machine in structs all contained in
 * BallControlStateMachine. BallControlStateMachine puts them
 * all together to make them function cohesively
 */
#include "main.h" // gives access to BallControlStateMachine and other dependencies

/* ------------------------------------------------------------ */
/*                      Public Information                      */
/* ------------------------------------------------------------ */
BallControlStateMachine::BallControlStateMachine()
    : controlEnabled(true), mbtnIn(def::btn_bc_in), mbtnOut(def::btn_bc_out),
      mbtnShoot(def::btn_bc_shoot), mbtnFilter(def::btn_bc_down)
{
} // constructor to set defaults

void BallControlStateMachine::controlState() // sets the mini states based on inputs from the
                                             // controller
{
    if (mbtnOut.changedToPressed()) // when the  out button is pressed
    {
        // spin out
        mintake.mstate = IT_STATES::out;
        mindexer.mstate = IX_STATES::out;
    }
    else if (mbtnOut.changedToReleased()) // when the out button is released
    {
        if (mbtnIn.isPressed()) // if the in button is also being pressed
        {
            // spin in
            mintake.mstate = IT_STATES::in;
            mindexer.mstate = IX_STATES::in;
        }
        else
        {
            // stop spinning
            mintake.mstate = IT_STATES::off;
            mindexer.mstate = IX_STATES::off;
        }
    }

    if (mbtnIn.changedToPressed()) // if the in button is pressed
    {
```

```
        // spin in
        mintake.mstate = IT_STATES::in;
        mindexer.mstate = IX_STATES::in;
    }
    else if (mbtnIn.changedToReleased()) // if the in button is released
    {
        if (mbtnOut.isPressed()) // is the out button is also being pressed
        {
            // spin out
            mintake.mstate = IT_STATES::out;
            mindexer.mstate = IX_STATES::out;
        }
        else
        {
            // stop
            mintake.mstate = IT_STATES::off;
            mindexer.mstate = IX_STATES::off;
        }
    }

    if (mbtnFilter.changedToPressed()) // if the filter button is pressed
    {
        mflywheel.mstate = FW_STATES::filter; // spin the filter
    }
    if (mbtnFilter.changedToReleased()) // is the filter button is released
    {
        if (mbtnShoot.isPressed()) // if the shoot button is also being pressed
        {
            mflywheel.mstate = FW_STATES::shoot; // shoot
        }
        else
        {
            mflywheel.mstate = FW_STATES::off; // stop spinning
        }
    }

    if (mbtnShoot.changedToPressed()) // if the shoot button is pressed
    {
        mflywheel.mstate = FW_STATES::shoot; // shoot
    }
    else if (mbtnShoot.changedToReleased()) // if the shoot button is released
    {
        if (mbtnFilter.isPressed()) // if the filter button is also being pressed
        {
            mflywheel.mstate = FW_STATES::filter; // spin the filter
        }
        else
        {
            mflywheel.mstate = FW_STATES::off; // stop shooting
        }
    }
}
void BallControlStateMachine::update() // controls the robot based on the state by
↪  updating each
```

```
                                        // mini state machine independantly
{
    mintake.update();
    mindexer.update();
    mflywheel.update();
}

void BallControlStateMachine::itIn() // spins the intakes in
{
    mintake.mstate = IT_STATES::in;
}
void BallControlStateMachine::itOut() // spins the intakes out
{
    mintake.mstate = IT_STATES::out;
}
void BallControlStateMachine::itOff() // stops the intakes
{
    mintake.mstate = IT_STATES::off;
}
void BallControlStateMachine::ixUp() // spins the indexer up
{
    mindexer.mstate = IX_STATES::in;
}
void BallControlStateMachine::ixDown() // spins the indexer down
{
    mindexer.mstate = IX_STATES::out;
}
void BallControlStateMachine::ixOff() // stops the indexer
{
    mindexer.mstate = IX_STATES::off;
}
void BallControlStateMachine::fwShoot() // shoots the flywheel
{
    mflywheel.mstate = FW_STATES::shoot;
}
void BallControlStateMachine::fwFilter() // spins the flywheel backwards
{
    mflywheel.mstate = FW_STATES::filter;
}
void BallControlStateMachine::fwOff() // stops the flywheel
{
    mflywheel.mstate = FW_STATES::off;
}

void BallControlStateMachine::itInFor(
    double imilliseconds) // spins the intakes for specified number of miliseconds
{
    mintake.mstate = IT_STATES::in;
    pros::delay(imilliseconds);
    mintake.mstate = IT_STATES::off;
}
void BallControlStateMachine::ixUpFor(
    double imilliseconds) // spins the indexer for specified number of miliseconds
{
```

```
159     mindexer.mstate = IX_STATES::in;
160     pros::delay(imilliseconds);
161     mindexer.mstate = IX_STATES::off;
162 }
163 void BallControlStateMachine::shoot(int ims) // shoots a ball
164 {
165     mflywheel.mstate = FW_STATES::shoot;
166     pros::delay(ims);
167     mflywheel.mstate = FW_STATES::off;
168 }
169 /* ------------------------------------------------------------ */
170 /*                      Private Information                     */
171 /* ------------------------------------------------------------ */
172
173 /* ---------------------- Nested Classes ---------------------- */
174 BallControlStateMachine::MIntake::MIntake()
175     : mstate(IT_STATES::off), mmotors({def::mtr_it_left, def::mtr_it_right})
176 {
177 } // constructor to set defaults
178 void BallControlStateMachine::MIntake::update() // updates the subsystem based on the
    ↪  state
179 {
180     switch (mstate)
181     {
182         case IT_STATES::off:
183             mmotors.moveVoltage(0);
184             break;
185         case IT_STATES::in:
186             mmotors.moveVoltage(12000);
187             break;
188         case IT_STATES::out:
189             mmotors.moveVoltage(-12000);
190             break;
191     }
192 }
193 /* ------------------------------------------------------------ */
194 BallControlStateMachine::MIndexer::MIndexer()
195     : mstate(IX_STATES::off), mmotor(def::mtr_ix) {} // constructor to set defaults
196 void BallControlStateMachine::MIndexer::update() // updates the subsystem based on the
    ↪  state
197 {
198     switch (mstate)
199     {
200         case IX_STATES::off:
201             mmotor.moveVoltage(0);
202             break;
203         case IX_STATES::in:
204             mmotor.moveVoltage(12000);
205             break;
206         case IX_STATES::out:
207             mmotor.moveVoltage(-12000);
208             break;
209     }
210 }
```

```
211
212 BallControlStateMachine::MFlywheel::MFlywheel()
213     : mstate(FW_STATES::off), mmotors({def::mtr_fw1, def::mtr_fw2})
214 {
215 } // constructor to set defaults
216 void BallControlStateMachine::MFlywheel::update() // updates the subsystem based on the
    ↪  state
217 {
218     switch (mstate)
219     {
220         case FW_STATES::off:
221             mmotors.moveVoltage(0);
222             break;
223         case FW_STATES::shoot:
224             mmotors.moveVoltage(12000);
225             break;
226         case FW_STATES::filter:
227             mmotors.moveVoltage(-12000);
228             break;
229     }
230 }
```

2.11 src/stateMachines/DrivetrainStateMachine.cpp

```cpp
/**
 * DrivetrainStateMachine.cpp
 *
 * This file contains the definitions of the DrivetrainStateMachine class.
 * DrivetrainStateMachine is a state machine that inherits from VStateMachine.
 * It has an enumeration of different possible states to make it easy for
 * the user to controll the drivetrain.
 *
 * To use the state machine in auton, you use doAutonMotion() to disable
 * the normal state machine tasks and run the specified action.
 */
#include "main.h" // gives access to DrivetrainStateMachine and other dependencies

/* ------------------------------------------------------------ */
/*                      Private Information                     */
/* ------------------------------------------------------------ */

/* --------------------------- State -------------------------- */
bool DrivetrainStateMachine::stateChanged() // returns whether the last state is the same as the current one
{
    if (mstate != mlastState)
    {
        return true;
    }
    return false;
}

/* ------------------------------------------------------------ */
/*                       Public Information                     */
/* ------------------------------------------------------------ */
DrivetrainStateMachine::DrivetrainStateMachine() : mstate(DT_STATES::off), mlastState(mstate), mcontroller(def::controller), mtoggleFieldCentric(def::btn_dt_tglFieldCentric), mdrivetrain(def::drivetrain) {} // constructor to set defaults

DT_STATES DrivetrainStateMachine::getState() { return mstate; }
void DrivetrainStateMachine::setState(DT_STATES istate)
{
    mlastState = mstate;
    mstate = istate;
}

void DrivetrainStateMachine::doAutonMotion(std::function<void()> iaction) // disable manual control, and execute the action
{
    DT_STATES oldState = mstate;
    setState(DT_STATES::busy);
    iaction();
    setState(oldState);
}

```

```
48 void DrivetrainStateMachine::controlState() // update the state based on controller
   ↪ input
49 {
50     if (mtoggleFieldCentric.changedToPressed()) // toggle field centric
51     {
52         if (mstate == DT_STATES::manual)
53         {
54             mstate = DT_STATES::fieldCentric;
55         }
56         else
57         {
58             mstate = DT_STATES::manual;
59         }
60     }
61 }
62
63 void DrivetrainStateMachine::update() // move the robot based on the state
64 {
65     switch (mstate)
66     {
67         case DT_STATES::off:
68             break;
69         case DT_STATES::busy:
70             break;
71         case DT_STATES::manual: // normal, arcade control
72             mdrivetrain.moveArcade(mcontroller.getAnalog(ControllerAnalog::leftY),
                ↪ mcontroller.getAnalog(ControllerAnalog::leftX),
                ↪ mcontroller.getAnalog(ControllerAnalog::rightX), false);
73             break;
74         case DT_STATES::fieldCentric: // field centric arcade control
75             QAngle direction = 90_deg -
                ↪ atan2(mcontroller.getAnalog(ControllerAnalog::leftY),
                ↪ mcontroller.getAnalog(ControllerAnalog::leftX)) * radian;
76             double magnitude = hypot(mcontroller.getAnalog(ControllerAnalog::leftX),
                ↪ mcontroller.getAnalog(ControllerAnalog::leftY));
77             mdrivetrain.moveInDirection(direction, true, magnitude,
                ↪ mcontroller.getAnalog(ControllerAnalog::rightX), true);
78             break;
79     }
80 }
```

2.12 src/util/CustomOdometry.cpp

```cpp
/**
 * CustomOdometry.cpp
 *
 * This file contains the declaration of the CustomOdometry class.
 * CustomOdometry is resposible for doing all the math and storing
 * information about the robot's position and orientation. Everthing
 * is static, because there doesn't need to be more than one position
 * calculation.
 */
#include "main.h" // gives access to CustomOdometry and other dependencies

/* ------------------------------------------------------------ */
/*                      Private Information                     */
/* ------------------------------------------------------------ */

/* ------------------------ Constants ------------------------- */
const double & CustomOdometry::moffFIn =
    def::TRACK_FORWARD_OFFSET.convert(inch); // offset of forward tracking wheel in
    ↪  inches
const double & CustomOdometry::moffSIn =
    def::TRACK_SIDE_OFFSET.convert(inch); // offset of side tracking wheel in inches
const double & CustomOdometry::mcircIn =
    def::TRACK_WHEEL_CIRCUMFERENCE.convert(inch); // tracking wheel circumference in
    ↪  inches

/* --------------------- Sensor References -------------------- */
ADIEncoder & CustomOdometry::meF = def::track_encoder_forward; // left tracking wheel
↪  encoder
ADIEncoder & CustomOdometry::meS = def::track_encoder_side; // right tracking wheel
↪  encoder
pros::Imu & CustomOdometry::mimu1 = def::imu_bottom; // inertial sensors
pros::Imu & CustomOdometry::mimu2 = def::imu_top;

/* ---------------------- Starting Values --------------------- */
OdomState CustomOdometry::mstate = {0_in, 0_in, 0_rad}; // position of the robot
bool CustomOdometry::menabled = true; // whether or not the loop is allowed to run

/* ------------------------- Methods -------------------------- */
std::valarray<double> CustomOdometry::getSensorVals() // returns new sensor values
{
    return {meF.get(), meS.get(),
            ((isinf(mimu1.get_rotation()) ? 0 : mimu1.get_rotation()) +
             (isinf(mimu2.get_rotation()) ? 0 : mimu2.get_rotation())) *
                M_PI / 180 / 2};
}

/* ------------------------------------------------------------ */
/*                       Public Information                     */
/* ------------------------------------------------------------ */
OdomState CustomOdometry::getState() { return mstate; } // returns the current state of
↪  the robot
QLength CustomOdometry::getX() { return mstate.x; } // returns the x value of the state
```

```
48 QLength CustomOdometry::getY() { return mstate.y; } // returns the y value of the state
49 QAngle CustomOdometry::getTheta() { return mstate.theta; } // returns the theta value
   ↪ of the state
50 void CustomOdometry::setState(const OdomState & istate) // sets the state of the robot
51 {
52     mstate = istate;
53 }
54
55 void CustomOdometry::enable() // allows the odometry thread to be started (but does not
   ↪ start it)
56 {
57     menabled = true;
58 }
59 void CustomOdometry::disable() // stops the odometry thread from running, prevents it
   ↪ from starting
60 {
61     menabled = false;
62 }
63
64 OdomState
65 CustomOdometry::mathStep(const std::valarray<double>
66                              ivalsDiff) // does one iteration of odometry math, given
                                ↪ sensor changes
67 {
68     const double df =
69         ivalsDiff[0] * mcircIn / 360; // stores the change of all tracking wheels in
            ↪ inches
70     const double ds = ivalsDiff[1] * mcircIn / 360;
71
72     double vectorLx, vectorLy; // declares local offset x and y
73     if (ivalsDiff[2]) // if the robot turned
74     {
75         vectorLx = 2 * sin(ivalsDiff[2] / 2) *
76                    (ds / ivalsDiff[2] + moffSIn); // sideways translation based on arc
77         vectorLy = 2 * sin(ivalsDiff[2] / 2) *
78                    (df / ivalsDiff[2] + moffFIn); // forward translation based on arc
79     }
80     else
81     {
82         vectorLx = ds; // sideways translation (without turning)
83         vectorLy = df; // forward translation (without turning)
84     }
85
86     if (isnan(vectorLy)) // makes sure the local offsets exist
87         vectorLy = 0;
88     if (isnan(vectorLx))
89         vectorLx = 0;
90
91     double avgT =
92         mstate.theta.convert(radian) + ivalsDiff[2] / 2; // calculates the direction
            ↪ the robot moved
93
94     double polarR = hypot(vectorLx, vectorLy); // calculates polar cordinate, r
95     double polarT =
```

```
96          atan2(vectorLy, vectorLx) - avgT; // calculates polar cordinate, theta, and
            ↪ rotates
97
98      double dx = sin(polarT) * polarR; // converts new polar coordinates back to
        ↪ cartesian
99      double dy = cos(polarT) * polarR;
100
101     if (isnan(dx)) // makes sure the cartesian coordinates exist
102         dx = 0;
103     if (isnan(dx))
104         dy = 0;
105
106     return {dx * inch, dy * inch}; // return the change in position
107 }
108
109 /* ------------------------------------------------------------ */
110 /*                        Friend Method                         */
111 /* ------------------------------------------------------------ */
112 void odomTaskFunc(void *) // friend function to CustomOdometry to be run as a separate
    ↪ thread
113 {
114     std::valarray<double> lastVals{0, 0, 0},
115         newVals{0, 0, 0}; // arrays used for storing sensor values
116     OdomState newState; // used to store the change in state
117     waitForImu(); // wait for the inertial sensors to calibrate
118     CustomOdometry::mstate = def::SET_ODOM_START; // set the starting position to the
        ↪ origin
119
120     while (CustomOdometry::menabled)
121     {
122         newVals = CustomOdometry::getSensorVals(); // provides new sensor values and
            ↪ saves them
123         newState = CustomOdometry::mathStep(
124             newVals -
125             lastVals); // runs odometry math on sensor value change to calulate change
                ↪ in state
126         lastVals = newVals; // stores sensor values for the next iteration
127
128         // updates state based on change
129         CustomOdometry::mstate.x += newState.x;
130         CustomOdometry::mstate.y += newState.y;
131         CustomOdometry::mstate.theta = newVals[2] * radian;
132
133         pros::delay(10); // run odometry at 100hz (every 10 ms)
134     }
135 }
```

2.13 src/util/util.cpp

```cpp
/**
 * util.cpp
 *
 * This file contains miscellaneous utility functions and classes
 * to help with the general organization of the rest of the code.
 */
#include "main.h" // gives access to util.hpp and other dependencies

/* ------------------------------------------------------------ */
/*                      ExtendedPoint Struct                    */
/* ------------------------------------------------------------  /
 * ExtendedPoint struct inherits from the built in okapi Point struct,
 * but provides additional functionality, like an orientation value
 * (theta) as well as x and y values. It also adds some vector operations that are used
 ↪  for path
 * following in the drivetrain class.
 */
ExtendedPoint::ExtendedPoint(QLength ix, QLength iy, QAngle itheta) : theta(itheta)
{
    x = ix;
    y = iy;
}

/* ----------------------- Subtraction ----------------------- */
ExtendedPoint ExtendedPoint::operator-(const ExtendedPoint & ivec) // overloaded
 ↪  subtraction
{
    return ExtendedPoint(x - ivec.x, y - ivec.y, theta - ivec.theta);
}
ExtendedPoint ExtendedPoint::sub(const ExtendedPoint & ivec) // subtraction method
{
    return *this - ivec;
}

/* ------------------------- Addition ----------------------- */
ExtendedPoint ExtendedPoint::operator+(const ExtendedPoint & ivec) // overloaded
 ↪  addition
{
    return ExtendedPoint(x + ivec.x, y + ivec.y, theta + ivec.theta);
}
ExtendedPoint ExtendedPoint::add(const ExtendedPoint & ivec) // addition method
{
    return *this + ivec;
}

/* ---------------------- Multiplication ---------------------- */
QLength ExtendedPoint::dot(const ExtendedPoint & ivec) // dot multiply vectors
{
    return (x.convert(inch) * ivec.x.convert(inch) + y.convert(inch) *
     ↪  ivec.y.convert(inch)) * inch;
}
```

```
48 ExtendedPoint ExtendedPoint::operator*(const double iscalar) // overloaded scalar
   ↪ multiplication
49 {
50     return ExtendedPoint(x * iscalar, y * iscalar, theta * iscalar);
51 }
52 ExtendedPoint ExtendedPoint::scalarMult(const double iscalar) // multiply the vectors
   ↪ by a scalar
53 {
54     return *this * iscalar;
55 }
56 ExtendedPoint ExtendedPoint::operator*(const ExtendedPoint & ivec) // elementwise
   ↪ multiplication
57 {
58     return ExtendedPoint((x.convert(inch) * ivec.x.convert(inch)) * inch,
59                          (y.convert(inch) * ivec.y.convert(inch)) * inch,
60                          (theta.convert(degree) * ivec.theta.convert(degree)) * degree);
61 }
62 ExtendedPoint ExtendedPoint::eachMult(const ExtendedPoint & ivec) // elementwise
   ↪ multiplication
63 {
64     return *this * ivec;
65 }
66
67 /* ----------------------- Comparative ----------------------- */
68 bool ExtendedPoint::operator==(const ExtendedPoint & ipoint) // overloaded equivalence
   ↪ check
69 {
70     return x == ipoint.x && y == ipoint.y && theta == ipoint.theta;
71 }
72
73 /* -------------------------- Other -------------------------- */
74 QLength ExtendedPoint::dist(const ExtendedPoint & ivec) // distance between points
75 {
76     return sqrt((x - ivec.x).convert(inch) * (x - ivec.x).convert(inch) +
77                 (y - ivec.y).convert(inch) * (y - ivec.y).convert(inch)) *
78            inch;
79 }
80 QLength ExtendedPoint::mag() // magnitude
81 {
82     return sqrt(x.convert(inch) * x.convert(inch) + y.convert(inch) * y.convert(inch))
   ↪  * inch;
83 }
84 ExtendedPoint ExtendedPoint::normalize() // creates a vector with a length of 1
85 {
86     return ExtendedPoint((x.convert(inch) / mag().convert(inch)) * inch,
87                          (y.convert(inch) / mag().convert(inch)) * inch, theta);
88 }
89 std::string ExtendedPoint::string() // returns the point in string form for testing
90 {
91     return "{" + std::to_string(x.convert(inch)).substr(0, 3) + ", " +
92            std::to_string(y.convert(inch)).substr(0, 3) + ", " +
93            std::to_string(theta.convert(degree)).substr(0, 3) + "}";
94 }
95
```

```
/* ------------------------------------------------------------ */
/*                         Misc Functions                       */
/* ------------------------------------------------------------ */
void waitForImu() // blocks the execution of the code until the imu is done calibrating
{
    while (def::imu_top.is_calibrating() || def::imu_bottom.is_calibrating())
        pros::delay(100);
}

/* -------------------- OdomDebug Helpers --------------------- */
void odomSetState(OdomDebug::state_t istate) // sets the state of odometry based on
↪  display inputs
{
    CustomOdometry::setState({istate.x, istate.y, istate.theta});
}
void odomResetAll() // resets everything having to do with odometry (for "Reset" button)
{
    CustomOdometry::setState({0_ft, 0_ft, 0_deg}); // sets the robot's positinon to 0
    def::imu_top.reset(); // resets the imu
    def::imu_bottom.reset(); // resets the imu
    // resets the ecoders
    def::track_encoder_forward.reset();
    def::track_encoder_side.reset();
    waitForImu(); // waits for the imu
}

/* ---------------------- Task Functions ---------------------- */
void sm_dt_task_func(void *) // state machine drivetrain task to be run independently
{
    while (true)
    {
        def::sm_dt.controlState(); // update the state from controller input
        def::sm_dt.update(); // moves the robot based on the state
        pros::delay(20);
    }
}

void sm_bc_task_func(void *) // state machine ball control task to be run independently
{
    while (true)
    {
        if (def::sm_bc.controlEnabled)
        {
            def::sm_bc.controlState(); // update the state from controller input if it
            ↪  is enabled
        }
        def::sm_bc.update(); // moves the robot based on the state
        pros::delay(20);
    }
}

void display_task_func(void *) // display task to be run independently
{
    while (true)
```

```
148     {
149         def::display.setOdomData(); // update the odometry information
150 
151         // room for any other miscellaneous debugging
152 
153         pros::delay(20);
154     }
155 }
156 
157 /* -------------------------- Macros ------------------------- */
158 void deploy() // deploys the robot
159 {
160     def::sm_bc.fwShoot(); // deploys the hood
161     pros::delay(250);
162     def::sm_bc.fwOff();
163 }
164 
165 /* ---------------------------------------------------------- */
166 /*                           Control                          */
167 /* ---------------------------------------------------------- */
168 
169 /* ------------------------ PID Class ----------------------- /
170  * PID is a feedback loop that uses the difference between the goal and the current
    ↪  position (error)
171  * of the robot to decide how much power to give the motors. The "P" stands for
    ↪  "proportional", and
172  * it adds power proportional to the error, so it gets slower and slower as it gets
    ↪  closer to the
173  * goal to prevent it from driving too fast past it. The "D" stands for "derivative",
    ↪  because it
174  * uses the derivative of the error (the speed of the robot) to apply power. The faster
    ↪  the robot
175  * goes, the more the d term works to slow it down. The "I" stands for "integral",
    ↪  because it uses
176  * the integral of the error (the absement of the robot) to apply power. When the robot
    ↪  is close to
177  * the goal, sometimes the "P" and "D" terms do not apply enough power to move the
    ↪  robot, but when
178  * the robot isn't moving (and when it is), the "I" term is acumulating, so it
    ↪  eventually builds up
179  * enough to move the robot even closer to the goal. This implementation of PID only
    ↪  enables the "I"
180  * term when the robot is close enough to the goal, to prevent "integral windup", which
    ↪  is when the
181  * integral gets too big when it's too far away from the goal.
182  *
183  * We have a PID controller class, because we use different PID loops in many different
    ↪  places in
184  * the code, so we wanted to be able to be able to quickly make one with constants
    ↪  specific to the
185  * application.
186  */
187 PID::PID(double ikP, double ikI, double ikD, double ikIRange, double isettlerError,
188          double isettlerDerivative,
```

```
189           QTime isettlerTime) // constructor that sets constants, and initializes
              ↪ variables
190     : msettlerError(isettlerError), msettlerDerivative(isettlerDerivative),
191       msettlerTime(isettlerTime), msettler(TimeUtilFactory::withSettledUtilParams(
192                                                 msettlerError, msettlerDerivative,
                                                  ↪ msettlerTime)
193                                                 .getSettledUtil()),
194       mkP(ikP), mkI(ikI), mkD(ikD), mkIRange(ikIRange), merror(0), mlastError(0),
          ↪ mtotalError(0),
195       mderivative(0)
196 {
197 }
198
199 PID::PID(const PID & iother) // copy constructor for duplicating PID objects behind the
    ↪ scenes
200 {
201     msettlerError = iother.msettlerError;
202     msettlerDerivative = iother.msettlerDerivative;
203     msettlerTime = iother.msettlerTime;
204     msettler =
205         TimeUtilFactory::withSettledUtilParams(msettlerError, msettlerDerivative,
              ↪ msettlerTime)
206             .getSettledUtil();
207     mkP = iother.mkP;
208     mkI = iother.mkI;
209     mkD = iother.mkD;
210     mkIRange = iother.mkIRange;
211     mlastError = iother.mlastError;
212     mtotalError = iother.mtotalError;
213     mderivative = iother.mderivative;
214 }
215
216 double PID::getLastError() { return mlastError; }
217 double PID::getTotalError() { return mtotalError; }
218
219 void PID::setGains(double ikP, double ikI, double ikD) // used only for changing
    ↪ constants later
220 {
221     mkP = ikP;
222     mkI = ikI;
223     mkD = ikD;
224 }
225
226 double PID::getP() { return mkP; }
227 double PID::getI() { return mkI; }
228 double PID::getD() { return mkD; }
229
230 double PID::iterate(double ierror) // goes through one iteration of the PID loop
231 {
232     merror = ierror;
233     if (mkI != 0) // regulate integral term
234     {
235         if (abs(merror) < mkIRange && merror != 0) // if in range, update mtotalError
236         {
```

```
237             mtotalError += merror;
238         }
239         else
240         {
241             mtotalError = 0;
242         }
243         util::chop<double>(-50 / mkI, 50 / mkI,
244                            mtotalError); // limit mtotalError to prevent integral windup
245     }
246 
247     mderivative = merror - mlastError; // calculate the derivative before lastError is
    ↪  overwritten
248     mlastError = merror; // save the current error for the next cycle
249 
250     return merror * mkP + mtotalError * mkI + mderivative * mkD;
251 }
252 
253 bool PID::isSettled() // returns whether or not the controller is settled at the target
254 {
255     return msettler->isSettled(merror);
256 }
257 
258 /* ------------------------ Slew Class ------------------------ /
259  * Slew rate control is a system that limits the change in speed to prevent wheel slip.
    ↪  If the robot
260  * changes speed too fast, the wheels can slip, and make the robot's motion less fluid.
    ↪  When the
261  * target speed changes by a lot, the slew rate controller slowly increases it's output
    ↪  to
262  * eventually get to the target speed.
263  *
264  * This Slew rate controller is also intended to be used with PID, but sometimes when
    ↪  slew is used
265  * with PID, it interferes with the settling of the PID. To prevent this, the slew rate
    ↪  controller
266  * is only active when there are large changes in the target input value, making it
    ↪  only really
267  * affect the beginning of the motion. For example, if the motors aren't moving, and
    ↪  the target
268  * value suddenly jumps to 100%, the slew controller might gradually increase by
    ↪  increments of 5%
269  * until it reaches 100%, but if the target value jumps to from0% to 20%, the slew
    ↪  controller might
270  * not engage (actual values depend on constants "mincrement" and "mactiveDifference").
271  */
272 Slew::Slew(double iincrement, double iactiveDifference)
273     : mincrement(iincrement), mactiveDifference(iactiveDifference), mlastValue(0) //
    ↪  constructor
274 {
275 }
276 
277 double Slew::getIncrement() { return mincrement; }
278 double Slew::getActiveDifference() { return mactiveDifference; }
279 double Slew::getLastValue() { return mlastValue; }
```

```cpp

double Slew::iterate(double ivalue) // limits the input value to maximum changes described by
                                    // constants when run in a loop
{
    if (abs(ivalue - mlastValue) >
        mactiveDifference) // only activate if the value difference is over the threshold
    {
        if (ivalue >
            mlastValue +
                mincrement) // if the input is too big, only let it increase by a maximum amount
        {
            mlastValue = mlastValue + mincrement;
            return mlastValue;
        }
        else if (ivalue < mlastValue - mincrement) // if the input is too small, only let it
                                                   // decrease by a maximum amount
        {
            mlastValue = mlastValue - mincrement;
            return mlastValue;
        }
    }
    mlastValue = ivalue;
    return ivalue; // this only happens if nothing is wrong
}

/* ------------------------------------------------------------ */
/*                             Util                             */
/* ------------------------------------------------------------  /
 * The util namespace is used to organaize basic functions that don't
 * necessarily need to be used for robotics.
 */

/* ---------------------- Angle Wrappers ---------------------  /
 * These methods take any angle, and return an angle representing the same position in a specific
 * range. For example, wrapDeg(370) returns 10, because 370 is out of the range [0, 360).
 */
double util::wrapDeg(double iangle) // range [0, 360)
{
    iangle = fmod(iangle, 360);
    if (iangle < 0)
        iangle += 360;
    return iangle;
}
double util::wrapDeg180(double iangle) // range [-180, 180]
{
    iangle = fmod(iangle, 360);
    if (iangle < -180)
        iangle += 360;
```

```
328     else if (iangle > 180)
329         iangle -= 360;
330     return iangle;
331 }
332 double util::wrapRad(double iangle) // range [0, 2pi)
333 {
334     iangle = fmod(iangle, 2 * 3.14159265358979323846);
335     if (iangle < 0)
336         iangle += 2 * 3.14159265358979323846;
337     return iangle;
338 }
339 double util::wrapRadPI(double iangle) // range [-pi, pi]
340 {
341     iangle = fmod(iangle, 2 * 3.14159265358979323846);
342     if (iangle < -3.14159265358979323846)
343         iangle += 2 * 3.14159265358979323846;
344     else if (iangle > 3.14159265358979323846)
345         iangle -= 2 * 3.14159265358979323846;
346     return iangle;
347 }
348 QAngle util::wrapQAngle(QAngle iangle) // range [0, 360) for QAngles
349 {
350     iangle = fmod(iangle.convert(degree), 360) * degree;
351     if (iangle < 0_deg)
352         iangle += 360_deg;
353     return iangle;
354 }
355 QAngle util::wrapQAngle180(QAngle iangle) // range [-180, 180] for QAngles
356 {
357     iangle = fmod(iangle.convert(degree), 360) * degree;
358     if (iangle < -180_deg)
359         iangle += 360_deg;
360     else if (iangle > 180_deg)
361         iangle -= 180_deg;
362     return iangle;
363 }
```