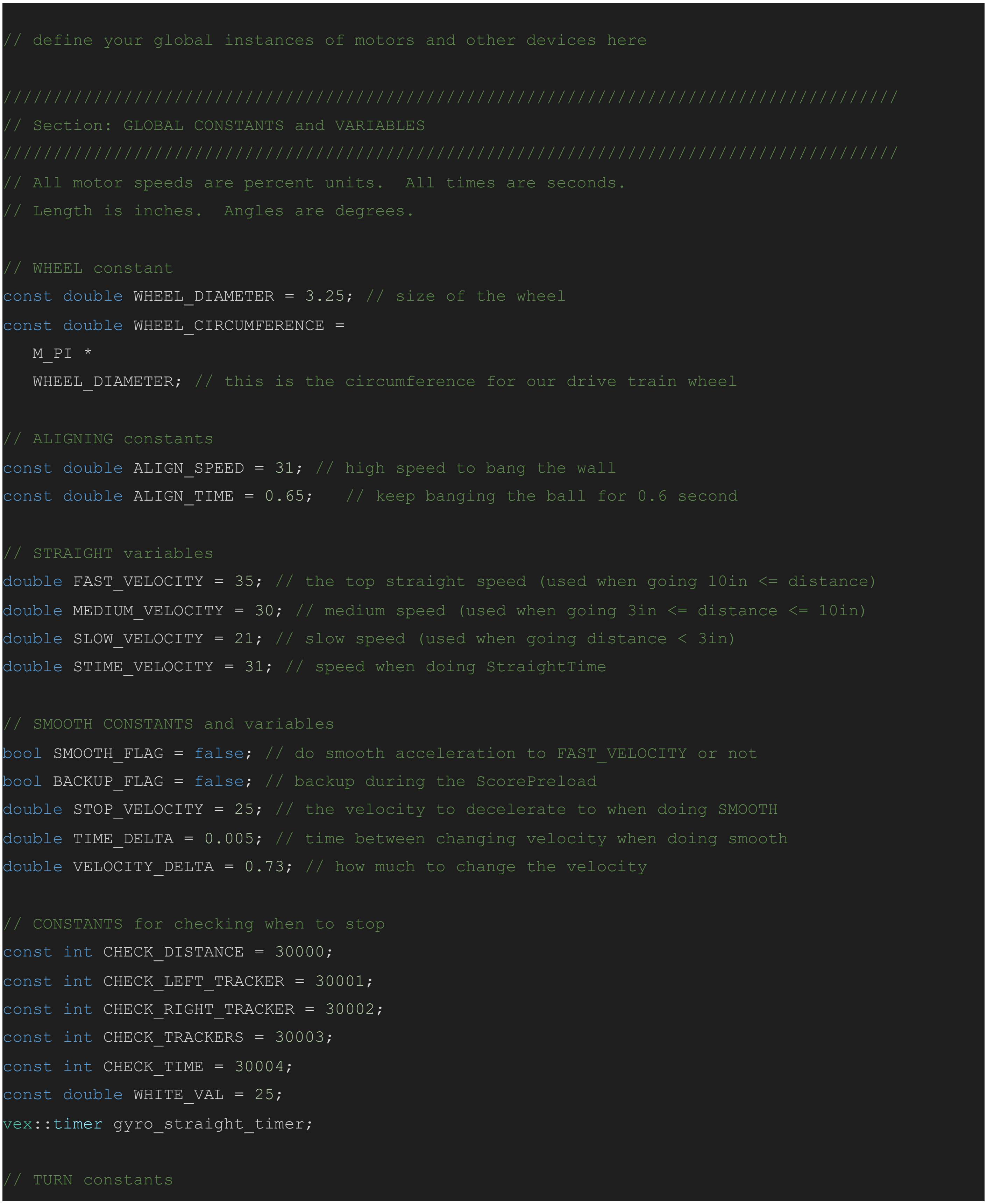
```
//////////////////////////////////////////////////////////////////////////
// 315V Annotated Programming Skills Code
//
// Please look for these functions and follow top down ….
//
// ProgrammingSkills106 - CURRENT programming skill run.
//   This was shown in last minute of video entry
// ProgrammingSkills108 - NEW route being developed.
//   In the video, we showed how we developed the CenterWallBall pattern.
//   This route reuses most of the functions used in ProgrammingSkills106
//   except for the new CenterWallBall pattern.  This demonstrates the power
//   of code reuse.
// ProgrammingSkills114 - OLD route, too slow.
//   In the video, we showed a fast-forwarded version of this to show how
//   we collect all red balls. Although unsuccessful overall, it does demonstrate more
//   features of our robot where we can collect and score two balls in
//   a programming skills.
//////////////////////////////////////////////////////////////////////////

// ---- START VEXCODE CONFIGURED DEVICES ----
// Robot Configuration:
// [Name]               [Type]        [Port(s)]
// FrontLeft            motor         3
// BackLeft             motor         5
// FrontRight           motor         8
// BackRight            motor         10
// Indexer              motor         11
// Ejector              motor         15
// LeftIntake           motor         1
// RightIntake          motor         6
// Controller1          controller
// Inertial             inertial      9
// LeftTracker          line          B
// RightTracker         line          A
// EjectorDetector      sonar         G, H
// ---- END VEXCODE CONFIGURED DEVICES ----

#include "vex.h"

using namespace vex;
// A global instance of competition
competition Competition;
```



```
// define your global instances of motors and other devices here

//////////////////////////////////////////////////////////////////////////////////////////
// Section: GLOBAL CONSTANTS and VARIABLES
//////////////////////////////////////////////////////////////////////////////////////////
// All motor speeds are percent units.  All times are seconds.
// Length is inches.  Angles are degrees.

// WHEEL constant
const double WHEEL_DIAMETER = 3.25; // size of the wheel
const double WHEEL_CIRCUMFERENCE =
   M_PI *
   WHEEL_DIAMETER; // this is the circumference for our drive train wheel

// ALIGNING constants
const double ALIGN_SPEED = 31; // high speed to bang the wall
const double ALIGN_TIME = 0.65;   // keep banging the ball for 0.6 second

// STRAIGHT variables
double FAST_VELOCITY = 35; // the top straight speed (used when going 10in <= distance)
double MEDIUM_VELOCITY = 30; // medium speed (used when going 3in <= distance <= 10in)
double SLOW_VELOCITY = 21; // slow speed (used when going distance < 3in)
double STIME_VELOCITY = 31; // speed when doing StraightTime

// SMOOTH CONSTANTS and variables
bool SMOOTH_FLAG = false; // do smooth acceleration to FAST_VELOCITY or not
bool BACKUP_FLAG = false; // backup during the ScorePreload
double STOP_VELOCITY = 25; // the velocity to decelerate to when doing SMOOTH
double TIME_DELTA = 0.005; // time between changing velocity when doing smooth
double VELOCITY_DELTA = 0.73; // how much to change the velocity

// CONSTANTS for checking when to stop
const int CHECK_DISTANCE = 30000;
const int CHECK_LEFT_TRACKER = 30001;
const int CHECK_RIGHT_TRACKER = 30002;
const int CHECK_TRACKERS = 30003;
const int CHECK_TIME = 30004;
const double WHITE_VAL = 25;
vex::timer gyro_straight_timer;

// TURN constants
```

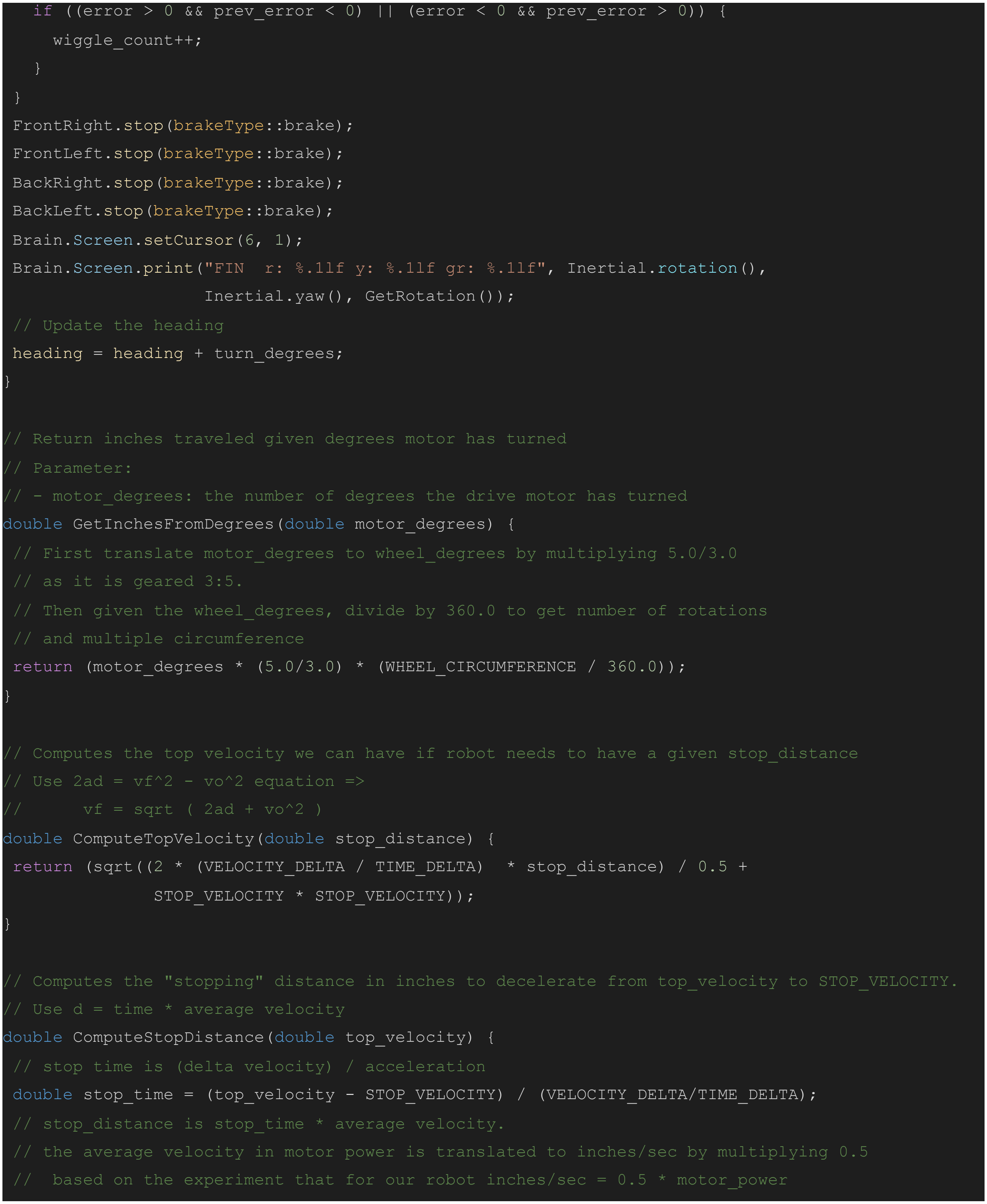
```
const double FAST_TURN_SPEED = 23;
const double SLOW_TURN_SPEED = 10; // 10->5
const double START_SLOW_TURN = 30;
const double STOP_SLOW_TURN = 5;
const double TURN_SPEED_SLOPE = (FAST_TURN_SPEED - SLOW_TURN_SPEED)/(START_SLOW_TURN -
STOP_SLOW_TURN);


// SCORING constants
double SCORE_TIME = 675;
const double CENTER_SCORE_POWER = 90;
const double CENTER_SCORE_TIME = 1000;

// GYRO stuff
// GLOBAL VARIABLE for intended heading
double heading = 0;

// The higher the correction factor, the stronger the correction.  The less,
// the smoother.
const double CORRECTION_FACTOR = 0.7;

/////////////////////////////////////////////////////////////////////////////////////////
// Section: Motor Setup, Inertial Setup
/////////////////////////////////////////////////////////////////////////////////////////
// This function sets the braketypes and torque for the motors, and resets their rotation.
void SetupMotors() {
 FrontLeft.setStopping(brakeType::brake);
 FrontLeft.resetRotation();
 FrontRight.setStopping(brakeType::brake);
 FrontRight.resetRotation();
 BackLeft.setStopping(brakeType::brake);
 BackLeft.resetRotation();
 BackRight.setStopping(brakeType::brake);
 BackRight.resetRotation();

 LeftIntake.setStopping(brakeType::coast);
 LeftIntake.resetRotation();
 LeftIntake.setMaxTorque(100, percentUnits::pct);
 RightIntake.setStopping(brakeType::coast);
 RightIntake.resetRotation();
 RightIntake.setMaxTorque(100, percentUnits::pct);
```

```
  Indexer.setStopping(brakeType::brake);
  Indexer.resetRotation();
  Indexer.setMaxTorque(100, percentUnits::pct);

  Ejector.setStopping(brakeType::coast);
  Ejector.resetRotation();
  Ejector.setMaxTorque(100, percentUnits::pct);
}

// This function calibrates the inertial sensor by calling the "calibrate" and
// using the "isCalibrating" method to wait for calibration to complete.
void SetupInertial() {
  // Calibrate the Inertial sensor
  Inertial.calibrate();
  Brain.Screen.print("Calibrating inertial ....");
  vex::task::sleep(4000);
  while (Inertial.isCalibrating()) {
    vex::task::sleep(100);
  }
  // Calculate the drift and show to the user
  Inertial.setHeading(0, degrees);
  Inertial.setRotation(0, degrees);
  for (int i = 0; i < 5; i++) {
    Brain.Screen.setCursor(2, 1);
    Brain.Screen.print("heading = %.3lf, rotation = %.3lf",
                       Inertial.heading(degrees), Inertial.rotation(degrees));
    vex::task::sleep(1000);
  }
  double drift = (Inertial.rotation(degrees) / 5.0);
  Brain.Screen.setCursor(4, 1);
  Brain.Screen.print("DRIFT = %.3lf", drift);
  vex::task::sleep(3000);
  Brain.Screen.clearScreen();
}

/////////////////////////////////////////////////////////////////////////////////////////
// Section: Drive Train for Auton/Skills based on the Gyro/Inertial
//   StraightTime, BackwardAlign
//   ResetHeading, GetRotation, PrintGyro
//   GyroTurn,
//   InitializeCheckStop, CheckStop
//   GyroStraight
//    (methods that call GyroStraight
//      - GyroStraightTime, GyroStraightDistance, GyroStraightUntilLeftWhite,
//        GyroStraightUntilRightWhite, GyroStraightUntilWhite)
///////////////////////////////////////////////////////////////////////////////////////
// StraightTime goes straight at a given time at a given velocity
// Parameters:
//  - velocity: specefies how fast in terms of %
//  - seconds: how long the straight goes for
void StraightTime(double velocity, double seconds) {
 // make the motors spin for some seconds
 FrontLeft.spin(directionType::fwd, velocity, velocityUnits::pct);
 FrontRight.spin(directionType::fwd, velocity, velocityUnits::pct);
 BackLeft.spin(directionType::fwd, velocity, velocityUnits::pct);
 BackRight.spin(directionType::fwd, velocity, velocityUnits::pct);

 vex::task::sleep(int(seconds * 1000));

 // stop all motors
 BackLeft.stop();
 BackRight.stop();
 FrontLeft.stop();
 FrontRight.stop();
}

// BackwardAlign assumes that the robot's back is close to the wall, and it will
// bang the wall for ALIGN_TIME seconds
void BackwardAlign() {
 StraightTime(-ALIGN_SPEED, ALIGN_TIME);
}

// Resets inertial+heading variable makes them 0
void ResetHeading() {
 heading = 0;
 Inertial.resetRotation();
}

// Scaling by 1.009
// We got this number by rotating the robot 90 degrees and checking the inertial reading
double GetRotation() {
 return 1.009 * Inertial.rotation(degrees);
}
```

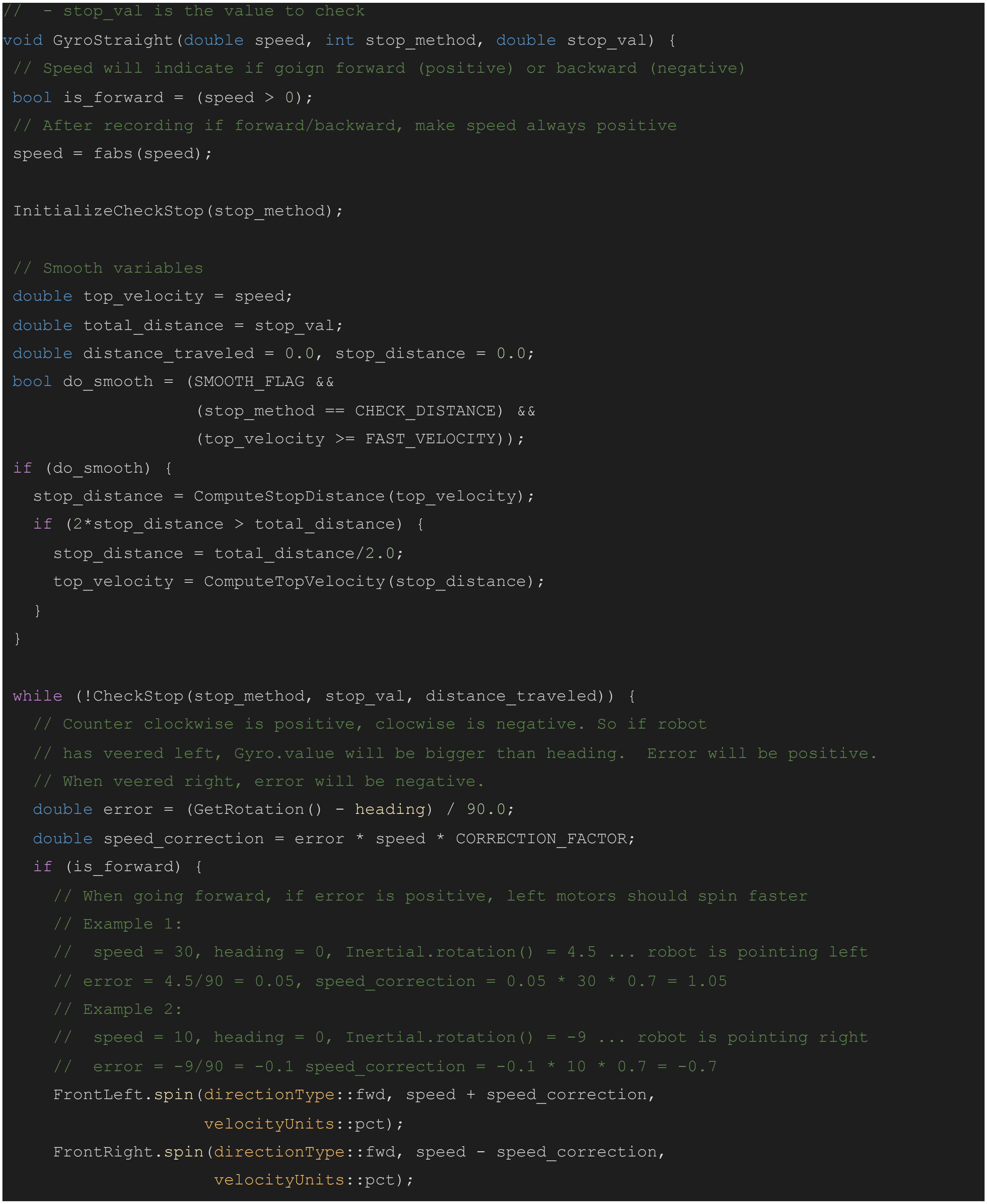
```
// Prints gyro stats for debugging
void PrintGyro() {
 Brain.Screen.print("Print gyro");
 for (int i = 0; i < 5; i++) {
   Brain.Screen.setCursor(i + 6, 1);
   Brain.Screen.print("h: %.1lf r: %.1lf y: %.1lf gr: %.lf", heading,
                      Inertial.rotation(), Inertial.yaw(), GetRotation());
   vex::task::sleep(2000);
 }
 Brain.Screen.clearScreen();
}

// Turns based on inertial sensor.
// At the end, updates the heading variable.
// Parameter:
// - turn_degrees: how much the robot turns in terms of degrees
void GyroTurn(double turn_degrees) {
 double target_heading = heading + turn_degrees;
 Brain.Screen.setCursor(2, 1);
 Brain.Screen.print(
     "START h: %.1lf, d: %.1lf th: %.1lf r: %.1lf y: %.1lf gr: %.lf", heading,
     turn_degrees, target_heading, Inertial.rotation(), Inertial.yaw(),
     GetRotation());
 if (turn_degrees > 0) {
   // Turn fast until we get to within 30 degrees of target
   while (GetRotation() < target_heading - 30) {
     // Brain.Screen.setCursor(4, 1);
     // Brain.Screen.print("  r: %.1lf y: %.1lf gr: %.1lf",
     //                    Inertial.rotation(), Inertial.yaw(), GetRotation());
     FrontRight.spin(directionType::fwd, FAST_TURN_SPEED, velocityUnits::pct);
     FrontLeft.spin(directionType::rev, FAST_TURN_SPEED, velocityUnits::pct);
     BackRight.spin(directionType::fwd, FAST_TURN_SPEED, velocityUnits::pct);
     BackLeft.spin(directionType::rev, FAST_TURN_SPEED, velocityUnits::pct);
     vex::task::sleep(10);
   }
   // Overturns 3.5 degrees.  At speed 10, by the time gyro returns 90 degrees,
   // robot would've overturned 3.5 degrees.
   while (GetRotation() < target_heading - 3.5) {
     // Brain.Screen.setCursor(4, 1);
     // Brain.Screen.print("  r: %.1lf y: %.1lf gr: %.1lf",
     //                    Inertial.rotation(), Inertial.yaw(), GetRotation());
     FrontRight.spin(directionType::fwd, SLOW_TURN_SPEED, velocityUnits::pct);
```

```
      FrontLeft.spin(directionType::rev, SLOW_TURN_SPEED, velocityUnits::pct);
      BackRight.spin(directionType::fwd, SLOW_TURN_SPEED, velocityUnits::pct);
      BackLeft.spin(directionType::rev, SLOW_TURN_SPEED, velocityUnits::pct);
      vex::task::sleep(10);
    }
  } else if (turn_degrees < 0) {
    // Turn fast until we get to within 30 degrees of target
    while (GetRotation() > target_heading + 30) {
      // Brain.Screen.setCursor(4, 1);
      // Brain.Screen.print("  r: %.1lf y: %.1lf gr: %.1lf",
      //                    Inertial.rotation(), Inertial.yaw(), GetRotation());
      FrontRight.spin(directionType::rev, FAST_TURN_SPEED, velocityUnits::pct);
      FrontLeft.spin(directionType::fwd, FAST_TURN_SPEED, velocityUnits::pct);
      BackRight.spin(directionType::rev, FAST_TURN_SPEED, velocityUnits::pct);
      BackLeft.spin(directionType::fwd, FAST_TURN_SPEED, velocityUnits::pct);
      vex::task::sleep(10);
    }
    // Overturns 3.5 degrees.  At speed 10, by the time gyro returns 90 degrees,
    // robot would've overturned 3.5 degrees.
    while (GetRotation() > target_heading + 3.5) {
      // Brain.Screen.setCursor(4, 1);
      // Brain.Screen.print("  r: %.1lf y: %.1lf gr: %.1lf",
      //                    Inertial.rotation(), Inertial.yaw(), GetRotation());
      FrontRight.spin(directionType::rev, SLOW_TURN_SPEED, velocityUnits::pct);
      FrontLeft.spin(directionType::fwd, SLOW_TURN_SPEED, velocityUnits::pct);
      BackRight.spin(directionType::rev, SLOW_TURN_SPEED, velocityUnits::pct);
      BackLeft.spin(directionType::fwd, SLOW_TURN_SPEED, velocityUnits::pct);
      vex::task::sleep(10);
    }
  }
  FrontRight.stop(brakeType::brake);
  FrontLeft.stop(brakeType::brake);
  BackRight.stop(brakeType::brake);
  BackLeft.stop(brakeType::brake);
  Brain.Screen.setCursor(6, 1);
  Brain.Screen.print("FIN  r: %.1lf y: %.1lf gr: %.1lf", Inertial.rotation(),
                     Inertial.yaw(), GetRotation());

  // Update the heading
  heading = target_heading;
}
```

```
// This is an improved turn, where it tries to turn within 1 degree
// and also wiggles back and forth until 1 degree.
// Parameter:
// - turn_degrees: how much the robot turns in terms of degrees
void NewGyroTurn(double turn_degrees) {
 double target_heading = heading + turn_degrees;
 Brain.Screen.setCursor(2, 1);
 Brain.Screen.print(
     "START h: %.1lf, d: %.1lf th: %.1lf r: %.1lf y: %.1lf gr: %.lf", heading,
     turn_degrees, target_heading, Inertial.rotation(), Inertial.yaw(),
     GetRotation());
 double error = GetRotation() - target_heading;
 double prev_error = error;
 double abs_error = fabs(error);
 int wiggle_count = 0;
 while (abs_error > 1 && wiggle_count < 4) {
   double turn_speed;
   // decrease speed smoothly
   if (abs_error > START_SLOW_TURN) {
     turn_speed = FAST_TURN_SPEED;
   } else if (abs_error < STOP_SLOW_TURN) {
     turn_speed = SLOW_TURN_SPEED;
   } else {
     turn_speed = SLOW_TURN_SPEED + TURN_SPEED_SLOPE*(abs_error - STOP_SLOW_TURN);
   }
   if (error > 0) {
     // Turn left/clockwise if error > 0
     FrontRight.spin(directionType::rev, turn_speed, velocityUnits::pct);
     FrontLeft.spin(directionType::fwd, turn_speed, velocityUnits::pct);
     BackRight.spin(directionType::rev, turn_speed, velocityUnits::pct);
     BackLeft.spin(directionType::fwd, turn_speed, velocityUnits::pct);
   } else {
     // Turn right/counterclockwise if error < 0
     FrontRight.spin(directionType::fwd, turn_speed, velocityUnits::pct);
     FrontLeft.spin(directionType::rev, turn_speed, velocityUnits::pct);
     BackRight.spin(directionType::fwd, turn_speed, velocityUnits::pct);
     BackLeft.spin(directionType::rev, turn_speed, velocityUnits::pct);
   }
   vex::task::sleep(TIME_DELTA*1000);
   prev_error = error;
   error =  GetRotation() - target_heading;
   abs_error = fabs(error);
```



```
   if ((error > 0 && prev_error < 0) || (error < 0 && prev_error > 0)) {
     wiggle_count++;
   }
 }
 FrontRight.stop(brakeType::brake);
 FrontLeft.stop(brakeType::brake);
 BackRight.stop(brakeType::brake);
 BackLeft.stop(brakeType::brake);
 Brain.Screen.setCursor(6, 1);
 Brain.Screen.print("FIN  r: %.1lf y: %.1lf gr: %.1lf", Inertial.rotation(),
                    Inertial.yaw(), GetRotation());
 // Update the heading
 heading = heading + turn_degrees;
}

// Return inches traveled given degrees motor has turned
// Parameter:
// - motor_degrees: the number of degrees the drive motor has turned
double GetInchesFromDegrees(double motor_degrees) {
 // First translate motor_degrees to wheel_degrees by multiplying 5.0/3.0
 // as it is geared 3:5.
 // Then given the wheel_degrees, divide by 360.0 to get number of rotations
 // and multiple circumference
 return (motor_degrees * (5.0/3.0) * (WHEEL_CIRCUMFERENCE / 360.0));
}

// Computes the top velocity we can have if robot needs to have a given stop_distance
// Use 2ad = vf^2 - vo^2 equation =>
//      vf = sqrt ( 2ad + vo^2 )
double ComputeTopVelocity(double stop_distance) {
 return (sqrt((2 * (VELOCITY_DELTA / TIME_DELTA)  * stop_distance) / 0.5 +
               STOP_VELOCITY * STOP_VELOCITY));
}

// Computes the "stopping" distance in inches to decelerate from top_velocity to STOP_VELOCITY.
// Use d = time * average velocity
double ComputeStopDistance(double top_velocity) {
 // stop time is (delta velocity) / acceleration
 double stop_time = (top_velocity - STOP_VELOCITY) / (VELOCITY_DELTA/TIME_DELTA);
 // stop_distance is stop_time * average velocity.
 // the average velocity in motor power is translated to inches/sec by multiplying 0.5
 //  based on the experiment that for our robot inches/sec = 0.5 * motor_power
 return (((top_velocity + STOP_VELOCITY )/2.0) * 0.5 * stop_time);
}

// Initializes the objects needed for checking the stopping conditions
// Parameter:
// - stop_method: the type of stopping condition we are checking
void InitializeCheckStop(int stop_method) {
 if (stop_method == CHECK_DISTANCE) {
   FrontLeft.setRotation(0, rotationUnits::deg);
 } else if (stop_method == CHECK_TIME) {
   gyro_straight_timer.clear();
 }
}

// This code checks the stopping type of the straight
// Parameters:
//  - stop_method is the type of stopping conditions (e.g., CHECK_DISTANCE, CHECK_TRACKERS,
CHECK_TIME)
//  - val is the value to check
//  - output & distance_traveled is how far the robot has traveled
bool CheckStop(int stop_method, double val, double & distance_traveled) {
 if (stop_method == CHECK_DISTANCE) {
   distance_traveled = GetInchesFromDegrees(fabs(FrontLeft.rotation(rotationUnits::deg)));
   return (distance_traveled >= fabs(val));
 } else if (stop_method == CHECK_TIME) {
   return (gyro_straight_timer.time(timeUnits::sec) >= val);
 } else if (stop_method == CHECK_LEFT_TRACKER) {
   return (LeftTracker.value(percentUnits::pct) <= val);
 } else if (stop_method == CHECK_RIGHT_TRACKER) {
   return (RightTracker.value(percentUnits::pct) <= val);
 } else if (stop_method == CHECK_TRACKERS) {
   return ((RightTracker.value(percentUnits::pct) <= val) ||
(LeftTracker.value(percentUnits::pct) <= val));
 }
 return true;
}

// This code makes the robot go straight and autocorrects.
// Parameters:
//  - speed is how fast the robot moves
//  - stop_method is the type of stopping condition to check (e.g., CHECK_DISTANCE,
CHECK_TRACKERS, CHECK_TIME)
```



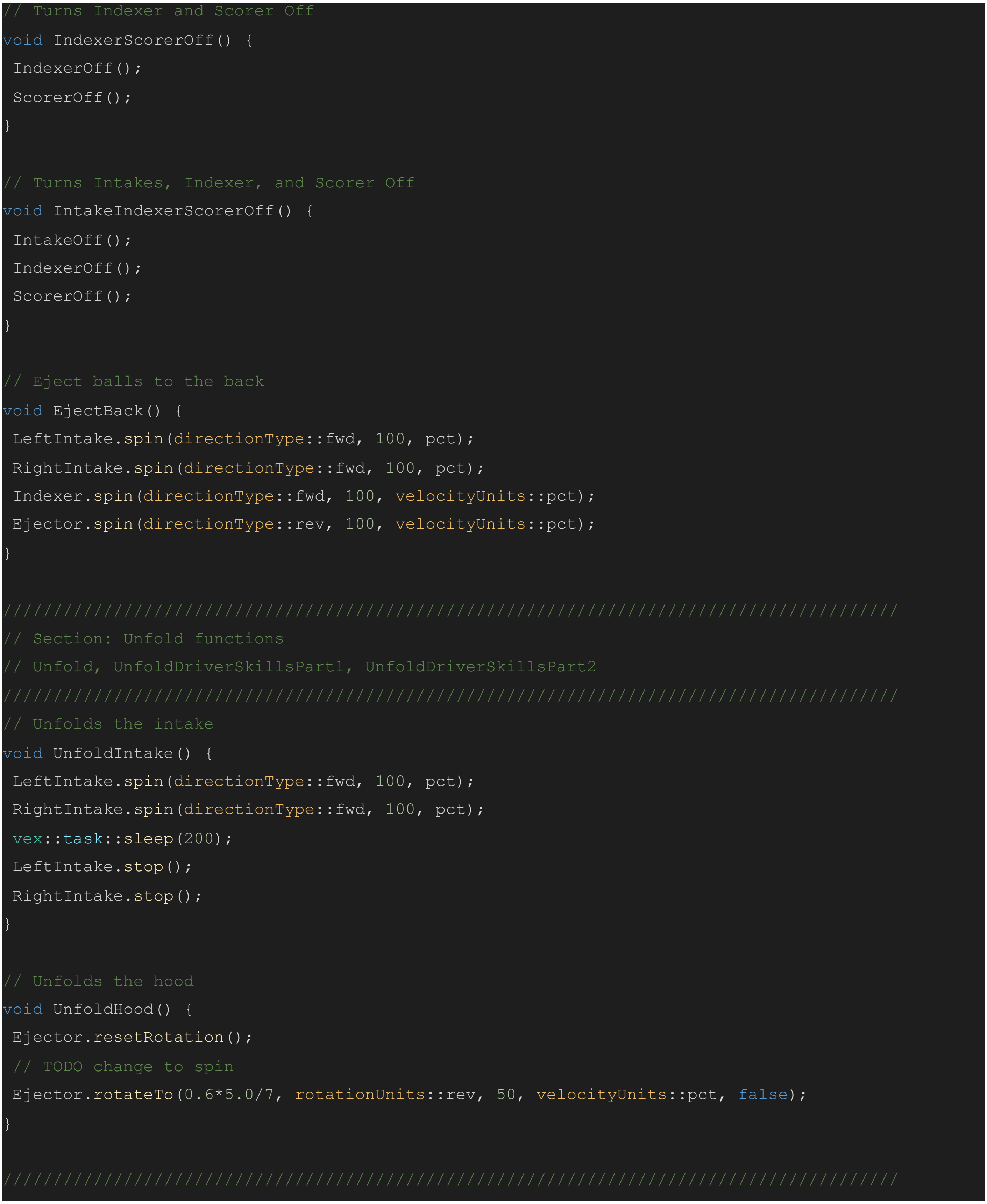
```
//  - stop_val is the value to check
void GyroStraight(double speed, int stop_method, double stop_val) {
 // Speed will indicate if goign forward (positive) or backward (negative)
 bool is_forward = (speed > 0);
 // After recording if forward/backward, make speed always positive
 speed = fabs(speed);

 InitializeCheckStop(stop_method);

 // Smooth variables
 double top_velocity = speed;
 double total_distance = stop_val;
 double distance_traveled = 0.0, stop_distance = 0.0;
 bool do_smooth = (SMOOTH_FLAG &&
                   (stop_method == CHECK_DISTANCE) &&
                   (top_velocity >= FAST_VELOCITY));
 if (do_smooth) {
   stop_distance = ComputeStopDistance(top_velocity);
   if (2*stop_distance > total_distance) {
     stop_distance = total_distance/2.0;
     top_velocity = ComputeTopVelocity(stop_distance);
   }
 }

 while (!CheckStop(stop_method, stop_val, distance_traveled)) {
   // Counter clockwise is positive, clocwise is negative. So if robot
   // has veered left, Gyro.value will be bigger than heading.  Error will be positive.
   // When veered right, error will be negative.
   double error = (GetRotation() - heading) / 90.0;
   double speed_correction = error * speed * CORRECTION_FACTOR;
   if (is_forward) {
     // When going forward, if error is positive, left motors should spin faster
     // Example 1:
     //  speed = 30, heading = 0, Inertial.rotation() = 4.5 ... robot is pointing left
     // error = 4.5/90 = 0.05, speed_correction = 0.05 * 30 * 0.7 = 1.05
     // Example 2:
     //  speed = 10, heading = 0, Inertial.rotation() = -9 ... robot is pointing right
     //  error = -9/90 = -0.1 speed_correction = -0.1 * 10 * 0.7 = -0.7
     FrontLeft.spin(directionType::fwd, speed + speed_correction,
                    velocityUnits::pct);
     FrontRight.spin(directionType::fwd, speed - speed_correction,
                     velocityUnits::pct);
```

```
      BackLeft.spin(directionType::fwd, speed + speed_correction,
                    velocityUnits::pct);
      BackRight.spin(directionType::fwd, speed - speed_correction,
                     velocityUnits::pct);
    } else {
      // When going backward, if error is positive (front is pointing to the
      // left) right motors should spin faster
      // Example 3:
      //  speed = 30, heading = 0, Inertial.rotation() = 4.5 ... robot is pointing left
      //  error = 4.5/90 = 0.05 speed_correction = 0.05 * 30 * 0.7 = 1.05
      // Example 4:
      //  speed = 10, heading = 0, Inertial.rotation() = -9 ... robot is pointing right
      // error = -9/90 = -0.1 speed_correction = -0.1 * 10 * 0.7 = -0.7
      FrontLeft.spin(directionType::rev, speed - speed_correction,
                     velocityUnits::pct);
      FrontRight.spin(directionType::rev, speed + speed_correction,
                      velocityUnits::pct);
      BackLeft.spin(directionType::rev, speed - speed_correction,
                    velocityUnits::pct);
      BackRight.spin(directionType::rev, speed + speed_correction,
                     velocityUnits::pct);
    }

    // smooth deceleration and acceleration code
    if (do_smooth) {
      if (((total_distance - distance_traveled) <= stop_distance) && (speed > STOP_VELOCITY)) {
        // If we are stop_distance within the end, start slowing down
        speed -= VELOCITY_DELTA;
      } else if (speed < top_velocity) {
        // Speed up if we havent achieved top_velocity
        speed += VELOCITY_DELTA;
      }
    }
    task::sleep(TIME_DELTA*1000);
  }
  if (do_smooth) {
    FrontLeft.stop(brakeType::coast);
    FrontRight.stop(brakeType::coast);
    BackRight.stop(brakeType::coast);
    BackLeft.stop(brakeType::coast);
  } else {
    FrontLeft.stop();
```

```
   FrontRight.stop();
   BackRight.stop();
   BackLeft.stop();
 }

}

// This function makes the robot go straight for a specified amount of time
// Parameters:
// - speed: how fast the robot goes in terms if %
// - time_constraint: how long it goes straight for
void GyroStraightTime(double speed, double time_constraint) {
 GyroStraight(speed, CHECK_TIME, time_constraint);
}

// This function makes the robot go straight until the left line sensor sees a white line
// Parameters:
// - speed: how fast the robot goes in terms of %
void GyroStraightUntilLeftWhite(double speed) {
 GyroStraight(speed, CHECK_LEFT_TRACKER, WHITE_VAL);
}

// This function makes the robot go straight until the right line sensor sees a white line
// Parameters:
// - speed: how fast the robot goes in terms of %
void GyroStraightUntilRightWhite(double speed) {
 GyroStraight(speed, CHECK_RIGHT_TRACKER, WHITE_VAL);
}

// This function makes the robot go straight until both sensors see a white line
// Parameters:
// - speed: how fast the robot goes in terms of %
void GyroStraightUntilWhite(double speed) {
 GyroStraight(speed, CHECK_TRACKERS, WHITE_VAL);
}

// This function makes the robot go straight until it reaches a specified distance
// Parameters:
// - speed: how fast the robot goes in terms of %
// - inches: how far the robot goes in terms of inches
void GyroStraightDistance(double speed, double inches) {
 GyroStraight(speed, CHECK_DISTANCE, inches);
```

```
}

/////////////////////////////////////////////////////////////////////////////////////////
// Section: Methods for turning on/off the Intake, Indexer, Scorer/Ejector
//  and various combinations.
/////////////////////////////////////////////////////////////////////////////////////////
// Turn on intake motors to bring up balls
void IntakeOn(double speed) {
 // Turn indexer slowly so ball goes to bottom of ramp
 RightIntake.spin(directionType::fwd, speed, velocityUnits::pct);
 LeftIntake.spin(directionType::fwd, speed, velocityUnits::pct);
}

// Turns on Intake and Indexer motors
// - intake_speed: speed of the intake motors in %
// - indexer_speed: speed of indexer motors in %
void IntakeIndexerOn(double intake_speed, double indexer_speed) {
 RightIntake.spin(directionType::fwd, intake_speed, velocityUnits::pct);
 LeftIntake.spin(directionType::fwd, intake_speed, velocityUnits::pct);
 Indexer.spin(directionType::fwd, indexer_speed, velocityUnits::pct);
}

// Turns on Intake motors at full speed and indexer at slow speed, thus leaving the ball at the
indexer
void IntakeToIndexer() {
 IntakeIndexerOn(100, 15);
}

// Turns on Intake and Indexer and Scorer motors
// - intake_speed: speed of the intake motors in %
// - indexer_speed: speed of indexer motors in %
// - scorer_speed: speed of scorer motors in %
void IntakeIndexerScorerOn(double intake_speed, double indexer_speed, double scorer_speed) {
 RightIntake.spin(directionType::fwd, intake_speed, velocityUnits::pct);
 LeftIntake.spin(directionType::fwd, intake_speed, velocityUnits::pct);
 Indexer.spin(directionType::fwd, indexer_speed, velocityUnits::pct);
 Ejector.spin(directionType::fwd, scorer_speed, velocityUnits::pct);
}
// Turns on Intake and Indexer motors at full speed and scorer motor at slow speed, thus leaving
the ball at the scorer
void IntakeToScorer() {
 IntakeIndexerScorerOn(100, 100, 6);
```

```
}

// Turns on Indexer and Scorer motors
void IndexerScorerOn() {
 Indexer.spin(directionType::fwd, 100, velocityUnits::pct);
 Ejector.spin(directionType::fwd, 100, velocityUnits::pct);
}

// Turns on Indexer and Scorer motors
// - scorer_power: speed of scorer motors in %
void IndexerScorerOn(double scorer_power) {
 Indexer.spin(directionType::fwd, scorer_power, velocityUnits::pct);
 Ejector.spin(directionType::fwd, scorer_power, velocityUnits::pct);
}

// Used for auton before a tower, so that the ball is not high up
void BackwardIndexer() {
 Indexer.spin(directionType::rev, 50, velocityUnits::pct);
}

// Turns Intakes Off
void IntakeOff() {
 RightIntake.stop();
 LeftIntake.stop();
}

// Turns Indexer Off
void IndexerOff() {
 Indexer.stop();
}

// Turns Scorer Off
void ScorerOff() {
 Ejector.stop();
}

// Turns Intakes and Indexer Off
void IntakeIndexerOff() {
 IntakeOff();
 IndexerOff();
}
```

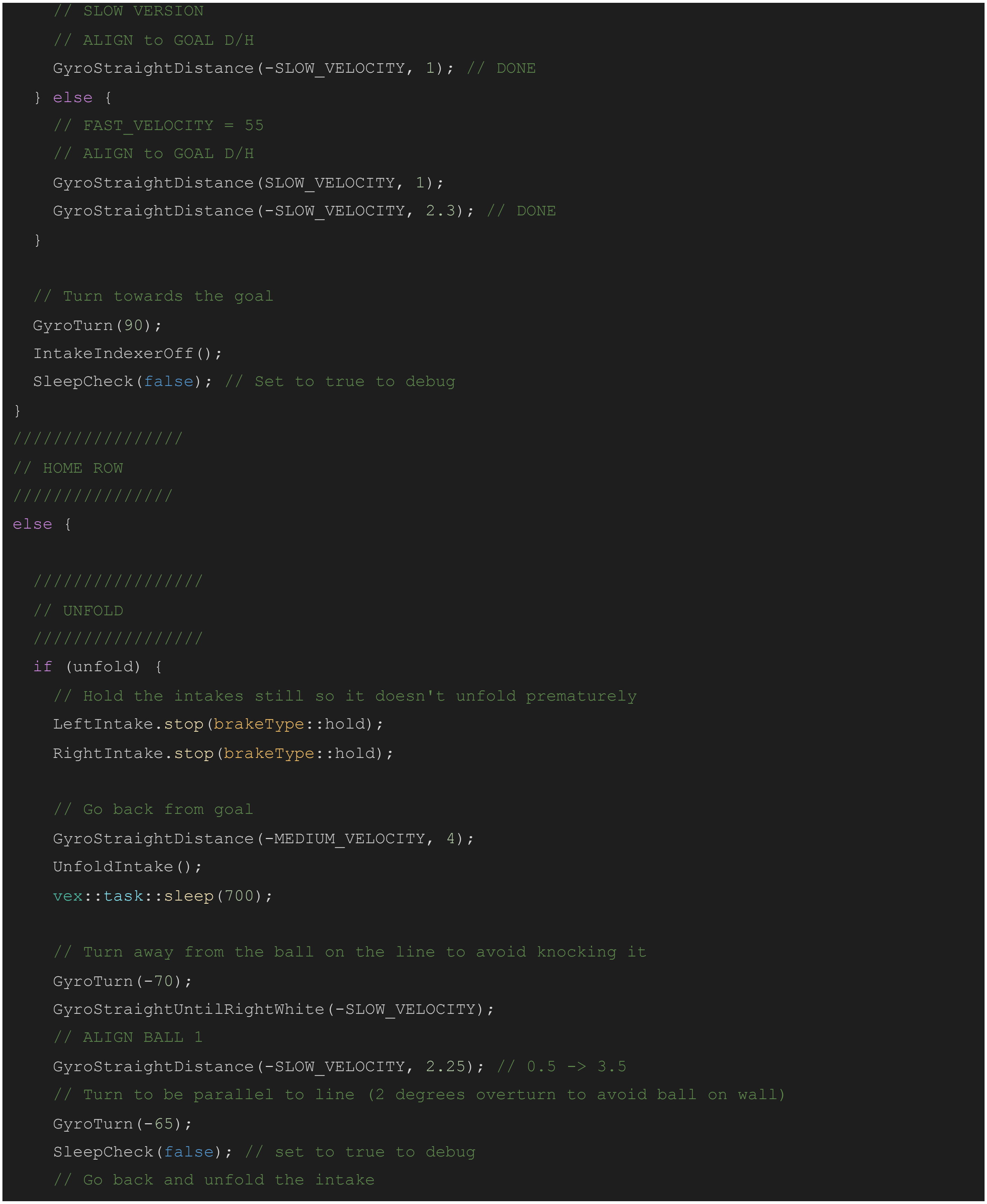


```
// Turns Indexer and Scorer Off
void IndexerScorerOff() {
 IndexerOff();
 ScorerOff();
}

// Turns Intakes, Indexer, and Scorer Off
void IntakeIndexerScorerOff() {
 IntakeOff();
 IndexerOff();
 ScorerOff();
}

// Eject balls to the back
void EjectBack() {
 LeftIntake.spin(directionType::fwd, 100, pct);
 RightIntake.spin(directionType::fwd, 100, pct);
 Indexer.spin(directionType::fwd, 100, velocityUnits::pct);
 Ejector.spin(directionType::rev, 100, velocityUnits::pct);
}

////////////////////////////////////////////////////////////////////////////////////////
// Section: Unfold functions
// Unfold, UnfoldDriverSkillsPart1, UnfoldDriverSkillsPart2
////////////////////////////////////////////////////////////////////////////////////////
// Unfolds the intake
void UnfoldIntake() {
 LeftIntake.spin(directionType::fwd, 100, pct);
 RightIntake.spin(directionType::fwd, 100, pct);
 vex::task::sleep(200);
 LeftIntake.stop();
 RightIntake.stop();
}

// Unfolds the hood
void UnfoldHood() {
 Ejector.resetRotation();
 // TODO change to spin
 Ejector.rotateTo(0.6*5.0/7, rotationUnits::rev, 50, velocityUnits::pct, false);
}

////////////////////////////////////////////////////////////////////////////////////////
```

```
// Section: ProgrammingSkills
// ProgrammingSkills
////////////////////////////////////////////////////////////////////////////////////////
// Sleeps if doSleep is true
void SleepCheck(bool doSleep) {
 if (doSleep) {
   vex::task::sleep(6000);
 }
}

// Sleeps
void SleepCheck() {
 SleepCheck(true);
}

/ Turns on the Indexer and Scorer motors for however many balls there are in terms of seconds
// Parameters:
// - numBalls: number of balls to score, the motors will be on for this many seconds
void FastScoreBalls(int numBalls) {
 IndexerScorerOn();
 vex::task::sleep(SCORE_TIME*numBalls);
 IndexerScorerOff();
}

// Turns on the Intake, Indexer and Scorer motors for however many balls there are in terms of
seconds
// Parameters:
// - numBalls: number of balls to score, the motors will be on for this many seconds
void FastScoreIntakeBalls(int numBalls) {
 IntakeIndexerScorerOn(100, 100, 100);
 vex::task::sleep(SCORE_TIME*numBalls);
 IntakeIndexerScorerOff();
}

// Turns on the Indexer and Scorer motors for however many balls there are in terms of seconds
// Parameters:
// - numBalls: number of balls to score, the motors will be on for this many seconds
// - scorer_power: power of scorer motor
void FastScoreBalls(int numBalls, double scorer_power) {
 IndexerScorerOn(scorer_power);
 vex::task::sleep(SCORE_TIME*numBalls);
 IndexerScorerOff();
```

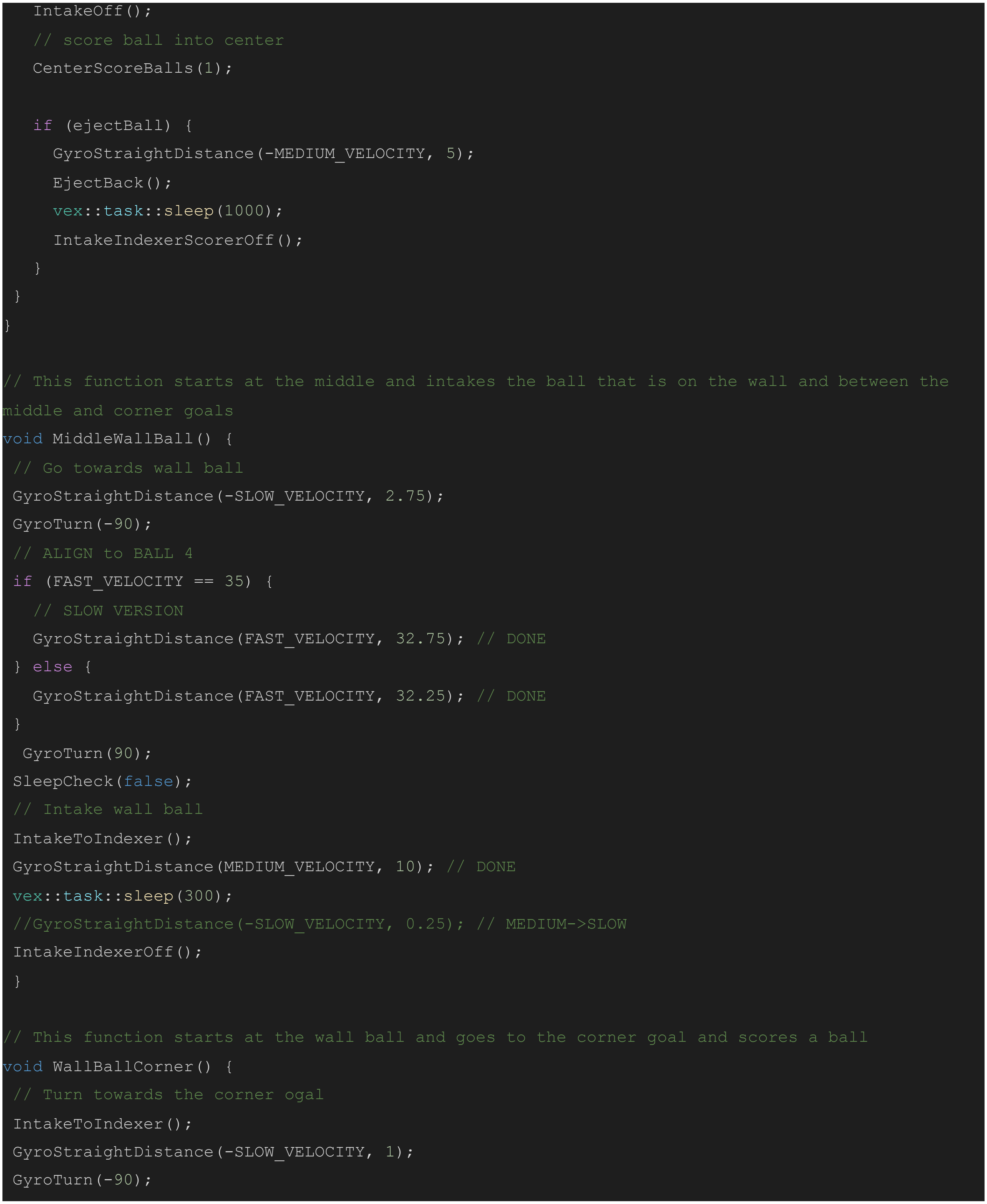
```
}

// Turns on the Indexer and Scorer motors for however many balls there are in terms of seconds
// This function will make the scorer motor go slower to score the center goal
// Parameters:
// - numBalls: number of balls to score, the motors will be on for this many seconds
void CenterScoreBalls(int numBalls) {
 IndexerScorerOn(CENTER_SCORE_POWER);
 vex::task::sleep(CENTER_SCORE_TIME*numBalls);
 IndexerScorerOff();
}

// Turns on the Indexer and Scorer motors for however many balls there are in terms of seconds
// This function will make the scorer motor turn backwards for a short time to make sure the ball isn't on top of the scorer roller
// Parameters:
// - numBalls: number of balls to score, the motors will be on for this many seconds
void ScoreBalls(int numBalls) {
 BackwardIndexer();
 vex::task::sleep(400);
 IndexerOff();
 FastScoreBalls(numBalls);
}

////////////////////////////////////////////////////////////////////////////////////////
// ProgrammingSkills 106 Helper Functions
////////////////////////////////////////////////////////////////////////////////////////
// This function will unfold the hood and score the preload
void ScorePreload() {
 // Unfold the hood
 UnfoldHood();
 // Backup from wall
 if (BACKUP_FLAG) {
   // ALIGN Goal A
   GyroStraightDistance(-SLOW_VELOCITY, 1.6);
 }
 // Hold intakes steady so it doesn't accidentally
 LeftIntake.stop(brakeType::hold);
 RightIntake.stop(brakeType::hold);
  // Turn towards goal
 GyroTurn(45);
 SleepCheck(false);
```

```
  // Score the pre load
 GyroStraightTime(STIME_VELOCITY, 0.49);
 FastScoreBalls(1);
}

// This pattern takes the robot from a corner goal to the middle goal.
// Set isHomeRow to true if the path is along a home row.  In case of a home
// row, because the next ball is very close to the corner, the robot has to make
// two turns (90 degree, then 45 degree).  It then intakes the ball, slows down
// to check for the white line indicating a middle goal and turns towards it.
//
// If it is not in the home row, the intakes are turned on just before the white line
// as the ball in this case is on the white line.
// Parameters:
// isHomeRow: boolean that checks if it is a homerow or not
// unfold: if the robot should unfold or not
void CornerMiddle(bool isHomeRow, bool unfold) {
 /////////////////////
 // NOT A HOMEROW
 /////////////////////
 if (!isHomeRow) {
   // If not a home row, just go back and turn 135 degrees towards middle
   if (FAST_VELOCITY <= 35) {
     // SLOW VERSION
     // ALIGN to ball 3 and 7
     GyroStraightDistance(-MEDIUM_VELOCITY, 15.25); // DONE
   } else {
     // FAST_VELOCITY = 55
     // ALIGN to ball 3 and 7
     GyroStraightDistance(-MEDIUM_VELOCITY, 15.25); // DONE
   }
   GyroTurn(-135);
   EjectBack();
   SleepCheck(false); // Set to true to debug
   //IntakeToIndexer();
   GyroStraightDistance(FAST_VELOCITY, 25);
   IntakeIndexerScorerOff();

   IntakeToIndexer();
   // Look for white line marking middle goal, and back up after finding it
   GyroStraightUntilLeftWhite(SLOW_VELOCITY);
   if (FAST_VELOCITY <= 35) {
```

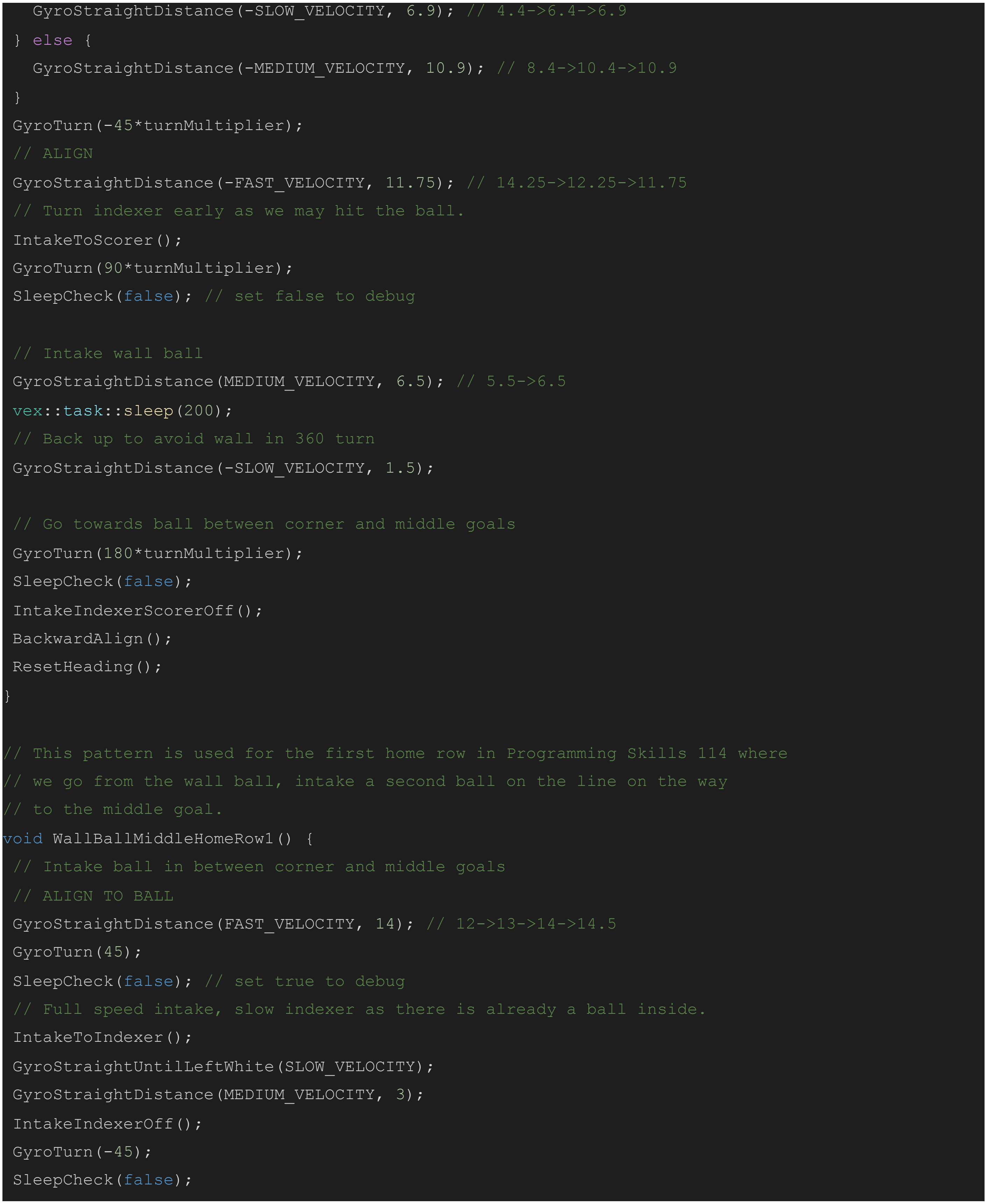


```
    // SLOW VERSION
    // ALIGN to GOAL D/H
    GyroStraightDistance(-SLOW_VELOCITY, 1); // DONE
  } else {
    // FAST_VELOCITY = 55
    // ALIGN to GOAL D/H
    GyroStraightDistance(SLOW_VELOCITY, 1);
    GyroStraightDistance(-SLOW_VELOCITY, 2.3); // DONE
  }

  // Turn towards the goal
  GyroTurn(90);
  IntakeIndexerOff();
  SleepCheck(false); // Set to true to debug
}
/////////////////
// HOME ROW
////////////////
else {

  /////////////////
  // UNFOLD
  /////////////////
  if (unfold) {
    // Hold the intakes still so it doesn't unfold prematurely
    LeftIntake.stop(brakeType::hold);
    RightIntake.stop(brakeType::hold);

    // Go back from goal
    GyroStraightDistance(-MEDIUM_VELOCITY, 4);
    UnfoldIntake();
    vex::task::sleep(700);

    // Turn away from the ball on the line to avoid knocking it
    GyroTurn(-70);
    GyroStraightUntilRightWhite(-SLOW_VELOCITY);
    // ALIGN BALL 1
    GyroStraightDistance(-SLOW_VELOCITY, 2.25); // 0.5 -> 3.5
    // Turn to be parallel to line (2 degrees overturn to avoid ball on wall)
    GyroTurn(-65);
    SleepCheck(false); // set to true to debug
    // Go back and unfold the intake
    //GyroStraightDistance(-MEDIUM_VELOCITY, 2);

  }
  /////////////////
  // NOT UNFOLD
  /////////////////
  else {
    // Go back from goal
    GyroStraightDistance(-MEDIUM_VELOCITY, 4);

    // Angles are slightly different, as robot turns differently when unfolded
    // Turn away from the ball on the line to avoid knocking it
    GyroTurn(-70);
    GyroStraightUntilRightWhite(-SLOW_VELOCITY);
    // ALIGN BALL 5
    GyroStraightDistance(-SLOW_VELOCITY, 3.5); // DONE
    // Turn to be parallel to line
    GyroTurn(-65);
    SleepCheck(false); // set to true to debug
  }
  // Backward align and reset heading
  GyroStraightTime(-STIME_VELOCITY, 1);
  //BackwardAlign();
  ResetHeading();
  SleepCheck(false); // set to true to debug

  // Intake ball along the line
  IntakeToIndexer();
  GyroStraightDistance(FAST_VELOCITY, 53);


  // Look for white line marking middle goal, and back up after finding it
  GyroStraightUntilLeftWhite(SLOW_VELOCITY);

  if (unfold) {
    if (FAST_VELOCITY == 35) {
      // SLOW VERSION
      // ALIGN GOAL B/F
      GyroStraightDistance(-SLOW_VELOCITY, 1.15); // DONE
    } else {
      // FAST_VELOCITY = 55
      // ALIGN GOAL B/F
```

```
       GyroStraightDistance(-SLOW_VELOCITY, 0.9); // 1.3->0.8->1->0.9 DONE
     }
   } else {
     if (FAST_VELOCITY == 35) {
       // SLOW VERSION
       // ALIGN GOAL B/F
       GyroStraightDistance(-SLOW_VELOCITY, 0.87); // DONE
     } else {
       // FAST_VELOCITY = 55
       // ALIGN GOAL B/F
       GyroStraightDistance(-SLOW_VELOCITY, 0.825); // 0.9->0.8->0.85->0.825 DONE
     }
   }
   IntakeIndexerOff();

   // Turn towards the goal
   GyroTurn(90);
   SleepCheck(false); // set to true to debug
 }

 // Score the ball
 GyroStraightTime(STIME_VELOCITY, 0.5);
 FastScoreBalls(1);
}

// This function starts out at the middle goal, intakes a ball, and scores it in the corner goal
void MiddleCorner() {
 // Go towards ball along the line.
 GyroStraightUntilWhite(-SLOW_VELOCITY);
 // ALIGN to ball 2/6
 GyroStraightDistance(-SLOW_VELOCITY, 2.9); // MEDIUM->SLOW DONE
 GyroTurn(-90);
 SleepCheck(false);

 // Intake the ball
 IntakeToIndexer();
 // ALIGN to goal C/G
 GyroStraightDistance(FAST_VELOCITY, 47.25); // DONE

 // Turn towards the corner
 GyroTurn(45);
 SleepCheck(false); // set to true to debug
 IntakeIndexerOff();
 GyroStraightTime(STIME_VELOCITY, 1.2);
 // Score the ball into the corner goal
 //FastScoreIntakeBalls(1);
 FastScoreBalls(1);
}



// This function starts out at the middle goal (on the side) and can either score the ball and
eject a blue ball or just intake a ball
// Parameters:
// scoreBall: If this is true, will intake and score a ball; if it is false, will just intake
ball
// ejectBall: If this is true, ejects blue ball that it has intaken
void MiddleCenter(bool scoreBall, bool ejectBall) {
 // Go towards ball near center
 GyroStraightDistance(-MEDIUM_VELOCITY, 9);
  // Intake ball near center
 if (!scoreBall) {
   GyroTurn(180);
   IntakeToScorer();
   GyroStraightDistance(MEDIUM_VELOCITY, 23);
   IntakeIndexerScorerOff();
 } else {
   // ALIGN BALL 8
   GyroTurn(187);
   SleepCheck(false); // set to true to debug
   IntakeToIndexer();
   // ALIGN to CENTER
   GyroStraightDistance(MEDIUM_VELOCITY, 12);
   GyroTurn(-3.5);
   // ALIGN to CENTER
   GyroStraightDistance(MEDIUM_VELOCITY, 14);

   // Go to center
   // Turn on intake to help line up with center
   GyroStraightTime(FAST_VELOCITY, 0.84);
   IntakeIndexerOff();

   // Intake a blue ball
   IntakeOn(100);
   vex::task::sleep(1100);
```

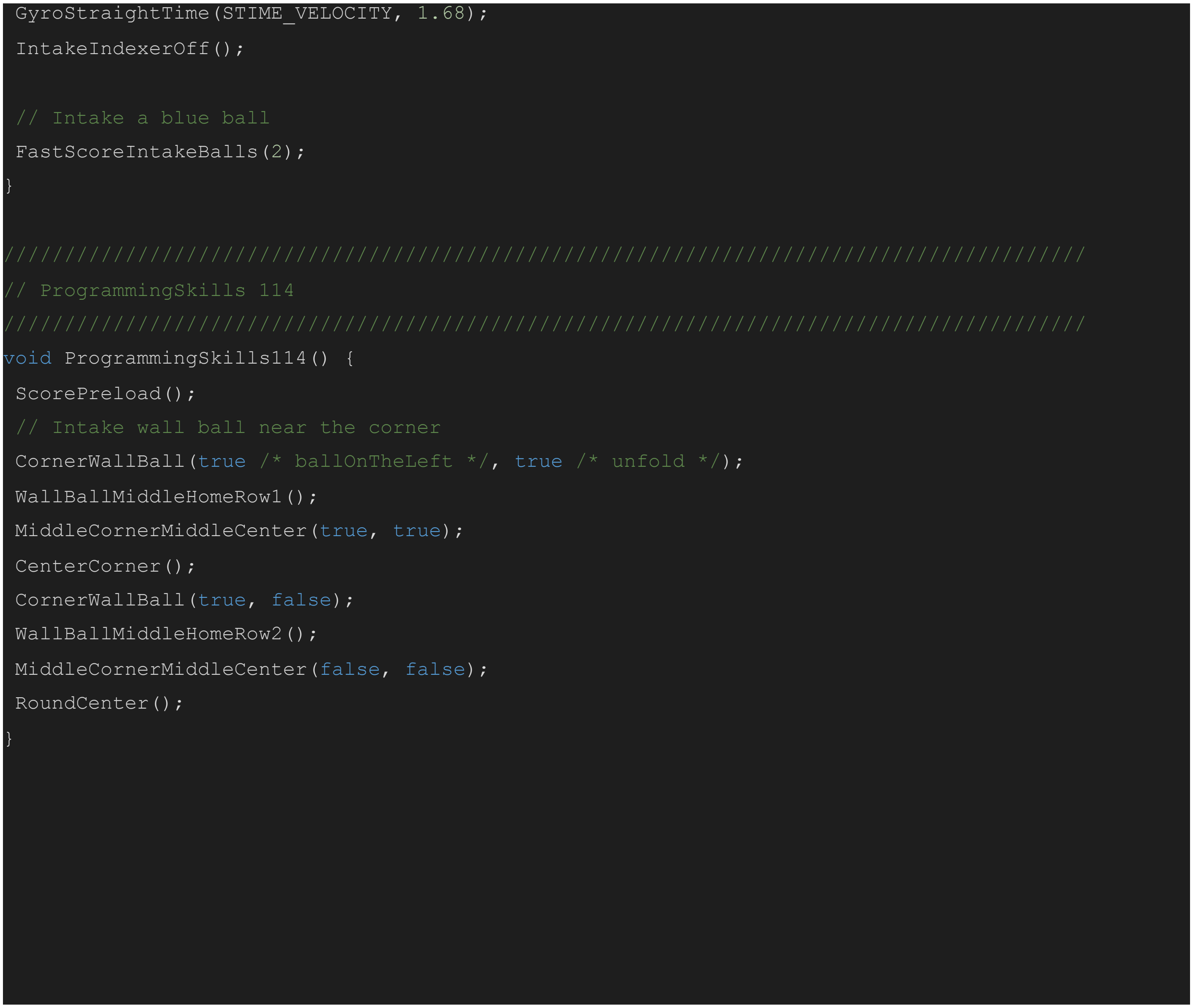


```
   IntakeOff();
   // score ball into center
   CenterScoreBalls(1);

   if (ejectBall) {
     GyroStraightDistance(-MEDIUM_VELOCITY, 5);
     EjectBack();
     vex::task::sleep(1000);
     IntakeIndexerScorerOff();
   }
 }
}

// This function starts at the middle and intakes the ball that is on the wall and between the
middle and corner goals
void MiddleWallBall() {
 // Go towards wall ball
 GyroStraightDistance(-SLOW_VELOCITY, 2.75);
 GyroTurn(-90);
 // ALIGN to BALL 4
 if (FAST_VELOCITY == 35) {
   // SLOW VERSION
   GyroStraightDistance(FAST_VELOCITY, 32.75); // DONE
 } else {
   GyroStraightDistance(FAST_VELOCITY, 32.25); // DONE
 }
  GyroTurn(90);
 SleepCheck(false);
 // Intake wall ball
 IntakeToIndexer();
 GyroStraightDistance(MEDIUM_VELOCITY, 10); // DONE
 vex::task::sleep(300);
 //GyroStraightDistance(-SLOW_VELOCITY, 0.25); // MEDIUM->SLOW
 IntakeIndexerOff();
 }

// This function starts at the wall ball and goes to the corner goal and scores a ball
void WallBallCorner() {
 // Turn towards the corner ogal
 IntakeToIndexer();
 GyroStraightDistance(-SLOW_VELOCITY, 1);
 GyroTurn(-90);
 IntakeIndexerOff();
 GyroStraightUntilLeftWhite(MEDIUM_VELOCITY);
 // ALIGN to GOAL E
 // GyroStraightDistance(SLOW_VELOCITY, 1); // DONE
 GyroTurn(45);
 SleepCheck(false);
 // Score ball into corner goal
 GyroStraightTime(STIME_VELOCITY, 0.5);
 FastScoreBalls(1);
}

// This
void CornerMiddleCorner(bool unfold) {
 CornerMiddle(true /* isHomeRow */, unfold);
 MiddleCorner();
}

/////////////////////////////////////////////////////////////////////////////////////////
// ProgrammingSkills 106
/////////////////////////////////////////////////////////////////////////////////////////
// Programming Skills 106 scores 106 points by putting
// 9 redballs into 9 goals forming 8 rows, and taking out 1 blue ball
// from the center.  It follows a circular path starting from
// a corner goal->middle goal->corner goal -> and so on
// After scoring on the 4th middle goal, it goes to the center to score
// the last red ball.
void ProgrammingSkills106() {
 ScorePreload();
 CornerMiddleCorner(true /* unfold */);
 CornerMiddle(false /* isHomeRow */, false /* unfold */);
 MiddleWallBall();
 WallBallCorner();
 CornerMiddleCorner(false /* unfold */);
 CornerMiddle(false /* isHomeRow */, false /* unfold */);
 MiddleCenter(true /* scoreBall */, true /* ejectBall */);
}

/////////////////////////////////////////////////////////////////////////////////////////
// ProgrammingSkills 108 and Helper Function CenterWallBall
/////////////////////////////////////////////////////////////////////////////////////////
// This function starts at the center goal and intakes the wall ball. Used in PS108
void CenterWallBall() {
 GyroStraightDistance(-MEDIUM_VELOCITY, 8);
 GyroTurn(130);
 GyroStraightDistance(MEDIUM_VELOCITY, 43.5);
 GyroTurn(50);
 SleepCheck(true);
 IntakeIndexerOn(100, 100);
 GyroStraightDistance(MEDIUM_VELOCITY, 11);
 IntakeIndexerOff();
}

// This program calls the correct functions to follow the PS108 route
void ProgrammingSkills108() {
 ScorePreload();
 CornerMiddleCorner(true /* unfold */);
 CornerMiddle(false /* isHomeRow */, false /* unfold */);
 MiddleCenter(true, false);
 CenterWallBall();
 WallBallCorner();
 CornerMiddleCorner(false /* unfold */);
 CornerMiddle(false /* isHomeRow */, false /* unfold */);
 MiddleCenter(true /* scoreBall */, true);
}

//////////////////////////////////////////////////////////////////////////////////////////
// ProgrammingSkills 114 Helper Functions
//////////////////////////////////////////////////////////////////////////////////////////
// This pattern is used after scoring at a corner goal, and the nearest wall
// ball is intaken.  Set ballOnTheLeft to true if after the robot backs up from
// the corner, the wall ball is on the left of the robot.
//
// This is used by programming skills route 114.
void CornerWallBall(bool ballOnTheLeft, bool unfold) {
 // Go to the wall ball
 int turnMultiplier = 1;
 // if the ball is on the right, negate all the turns
 if (!ballOnTheLeft) {
   turnMultiplier = -1;
 }
 if (unfold) {
   GyroStraightDistance(-SLOW_VELOCITY, 4);
   UnfoldIntake();
   vex::task::sleep(700);
```



```
   GyroStraightDistance(-SLOW_VELOCITY, 6.9); // 4.4->6.4->6.9
 } else {
   GyroStraightDistance(-MEDIUM_VELOCITY, 10.9); // 8.4->10.4->10.9
 }
 GyroTurn(-45*turnMultiplier);
 // ALIGN
 GyroStraightDistance(-FAST_VELOCITY, 11.75); // 14.25->12.25->11.75
 // Turn indexer early as we may hit the ball.
 IntakeToScorer();
 GyroTurn(90*turnMultiplier);
 SleepCheck(false); // set false to debug

 // Intake wall ball
 GyroStraightDistance(MEDIUM_VELOCITY, 6.5); // 5.5->6.5
 vex::task::sleep(200);
 // Back up to avoid wall in 360 turn
 GyroStraightDistance(-SLOW_VELOCITY, 1.5);

 // Go towards ball between corner and middle goals
 GyroTurn(180*turnMultiplier);
 SleepCheck(false);
 IntakeIndexerScorerOff();
 BackwardAlign();
 ResetHeading();
}

// This pattern is used for the first home row in Programming Skills 114 where
// we go from the wall ball, intake a second ball on the line on the way
// to the middle goal.
void WallBallMiddleHomeRow1() {
 // Intake ball in between corner and middle goals
 // ALIGN TO BALL
 GyroStraightDistance(FAST_VELOCITY, 14); // 12->13->14->14.5
 GyroTurn(45);
 SleepCheck(false); // set true to debug
 // Full speed intake, slow indexer as there is already a ball inside.
 IntakeToIndexer();
 GyroStraightUntilLeftWhite(SLOW_VELOCITY);
 GyroStraightDistance(MEDIUM_VELOCITY, 3);
 IntakeIndexerOff();
 GyroTurn(-45);
 SleepCheck(false);
 // Go to middle goal
 GyroStraightDistance(FAST_VELOCITY, 31.5);
 GyroStraightUntilLeftWhite(SLOW_VELOCITY);
 // ALIGN TO GOAL
 GyroStraightDistance(-SLOW_VELOCITY, 3);  // 2.1->2.5 MEDIUM->SLOW
 GyroTurn(90);
 SleepCheck(false);
 GyroStraightTime(STIME_VELOCITY, 0.67);

 // Score two balls into middle goal
 ScoreBalls(2);
}

// This function starts at the wall ball, intakes a ball, and scores 2 balls in the goal
void WallBallMiddle() {
 // ALIGN TO BALL
 GyroStraightDistance(MEDIUM_VELOCITY, 12); // 11->11.5-12
 GyroTurn(90);
 SleepCheck(false); // set true to debug
 // - Intake ball
 GyroStraightDistance(MEDIUM_VELOCITY, 6);
 // Full speed intake, slow indexer as there is already a ball inside.
 IntakeToIndexer();
 GyroStraightUntilWhite(SLOW_VELOCITY);
 // ALIGN TO GOAL
 GyroStraightDistance(-SLOW_VELOCITY, 1.25);  // MEDIUM->SLOW, 1.0->1.5->1.25
  // Go to middle goal
 GyroTurn(90);
 IntakeIndexerOff();
 SleepCheck(false); // set true to debug
 GyroStraightTime(STIME_VELOCITY, 0.67);

 // Score two balls into middle goal
 ScoreBalls(2);
}

// Pattern used to go from middle goal (e.g., B) to other middle goal (e.g., D)
//  - goes from middle goal, intakes ball along home row line, scores it at corner
//  - from the corner, goes to the ball near the middle goal (along the double line)
//    and scores that ball in the middle goal
//  - then turns around 180 degrees, intakes ball near center, and scores that ball
//    into the center goal
void MiddleCornerMiddleCenter(bool scoreBall, bool ejectBall) {
 MiddleCorner();
 CornerWallBall(false /* ballOnTheLeft */, false /* unfold */);
 WallBallMiddle();
 MiddleCenter(scoreBall, ejectBall);
}

// Goes from the Center Goal, picks it up, and scores it in a Corner Goal
void CenterCorner() {
 //  Go towards ball between corner and center goals
 // ALIGN TO BALL
 IntakeToIndexer();
 vex::task::sleep(1000);
 IntakeIndexerOff();
 GyroStraightDistance(-FAST_VELOCITY, 24); // 25->24
 GyroTurn(90);
 SleepCheck(false);
 EjectBack();
 GyroStraightDistance(FAST_VELOCITY, 25);
 ScorerOff();

 // Intake ball
 IntakeToIndexer();
 GyroStraightUntilWhite(SLOW_VELOCITY);
 GyroStraightDistance(SLOW_VELOCITY, 1);
 IntakeIndexerOff();

 // Go to corner goal
 GyroTurn(90);
 // ALIGN TO GOAL
 GyroStraightDistance(FAST_VELOCITY, 16.7);
 GyroTurn(-45);
 SleepCheck(false);
 GyroStraightTime(STIME_VELOCITY, 1.38);

 // Score ball into corner goal
 FastScoreBalls(1);
}

// This function starts at the Wall Ball, gets a ball near the center, and scores 2 balls in the
center goal
```

```
void WallBallMiddleHomeRow2 () {
 // Go towards goal between center and middle goal
 // ALIGN TO BALL
 GyroStraightDistance(FAST_VELOCITY, 8.8); // 10.8->9.8->8.8
 GyroTurn(-11.5); // change 12->11
 SleepCheck(false);

 // Intake ball
 // Full speed intake, slow indexer as there is already a ball inside.
 IntakeToIndexer();
 // ALIGN TO GOAL
 GyroStraightDistance(FAST_VELOCITY, 51);
 IntakeIndexerScorerOff();

 // Go to middle goal
 GyroTurn(11.5+90);
 SleepCheck(false);
 GyroStraightDistance(FAST_VELOCITY, 16.7);
 GyroStraightUntilWhite(SLOW_VELOCITY);
 GyroStraightTime(STIME_VELOCITY, 0.67);
 // Score ball into middle goal
 ScoreBalls(2);
}

// This function goes to a ball near the center and scores 2 balls while intaking blue balls from
the center
void RoundCenter() {
 // - Go to ball near center
 // ALIGN TO BALL
 GyroTurn(48);
 SleepCheck(false);

 // Intake ball
 IntakeToScorer();
 // ALIGN TO GOAL
 GyroStraightDistance(FAST_VELOCITY, 31.7);

 // Go towards center
 GyroTurn(-48-90);
 IntakeIndexerScorerOff();
 SleepCheck(false);
 IntakeToIndexer();
```



```
 GyroStraightTime(STIME_VELOCITY, 1.68);
 IntakeIndexerOff();

 // Intake a blue ball
 FastScoreIntakeBalls(2);
}

/////////////////////////////////////////////////////////////////////////////////////////
// ProgrammingSkills 114
/////////////////////////////////////////////////////////////////////////////////////////
void ProgrammingSkills114() {
 ScorePreload();
 // Intake wall ball near the corner
 CornerWallBall(true /* ballOnTheLeft */, true /* unfold */);
 WallBallMiddleHomeRow1();
 MiddleCornerMiddleCenter(true, true);
 CenterCorner();
 CornerWallBall(true, false);
 WallBallMiddleHomeRow2();
 MiddleCornerMiddleCenter(false, false);
 RoundCenter();
}
```