Team 10:
Matthew Haahr (MH)
Brian Shin (BS)
Nick Hom (NH)

RBE 2002: Unified Robotics II—Lab 5: Transporting Objects Post-Lab

Contribution Statement: For this lab, work was divided well. Matt took on the bulk of the programming and testing of the robot. Brian helped debug and worked on the testing. Everyone worked on the documentation of this lab and answered the questions.

1. **Task 1**
    a. Coordinates used:

Table 1:

X position (m)	Y position (m)
0.610	0.610
1.220	0.915
1.525	0
0.610	-0.305
0	0

Table 1: Waypoint values for parkour.

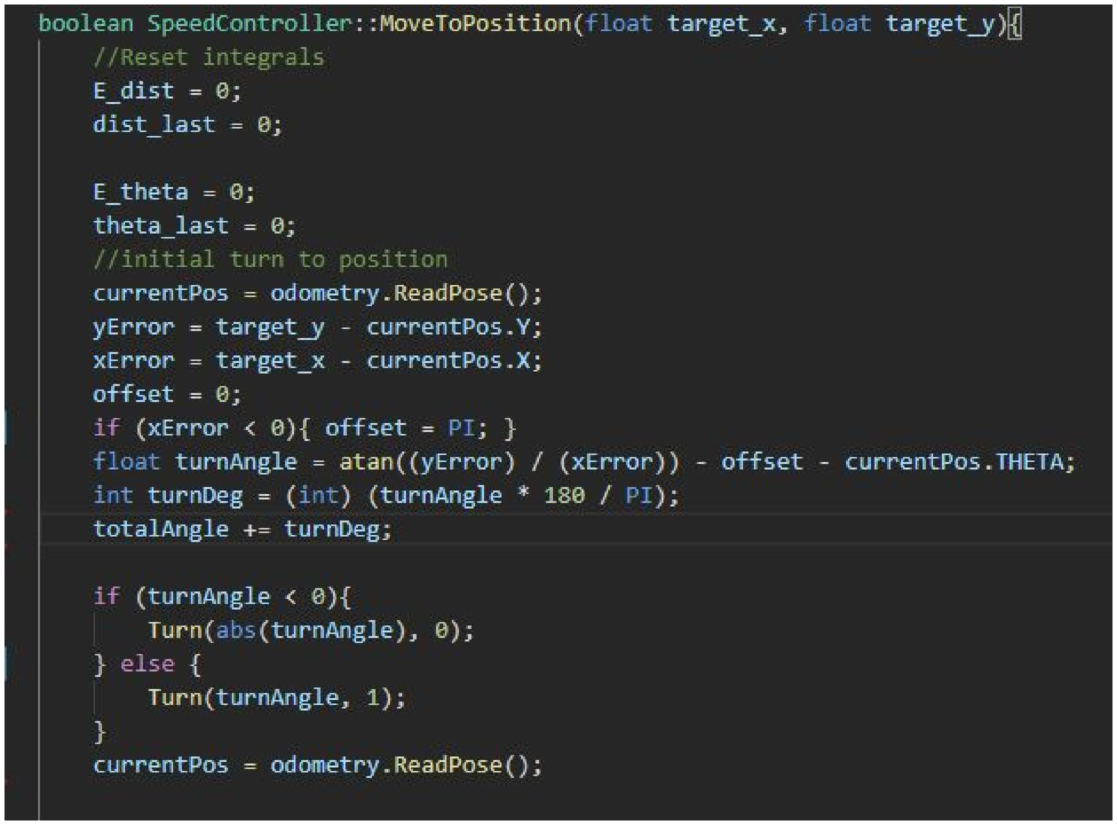
2. **Task 2**
    a. PID vs P controller speed comparison:
    A fully implemented PID controller can be significantly faster and is more robust than a simple proportional controller. A tuned proportional controller no matter the usage will either oscillate about the setpoint until a threshold is reached, or will have a consistent undershot known as steady-state error. For some uses, this may be all that is needed, but for most uses, the introduction of integral and/or derivative control will increase the robustness of the system. By adding an integral term to the controller, oftentimes the steady-state error can be eliminated. Likewise, by adding a derivative term to the controller, the system can "anticipate" error based on the rate of change of error and provide a damping effect or increased responsiveness.

    The PID controller completed the parkour circuit in: 2:11 min
    The Proportional controller completed the circuit in: 19:02 min (KpDist: 20; KpTheta: 11)

b. `SpeedController::MoveToPosition(float,float)`

   i. Code Screenshot:

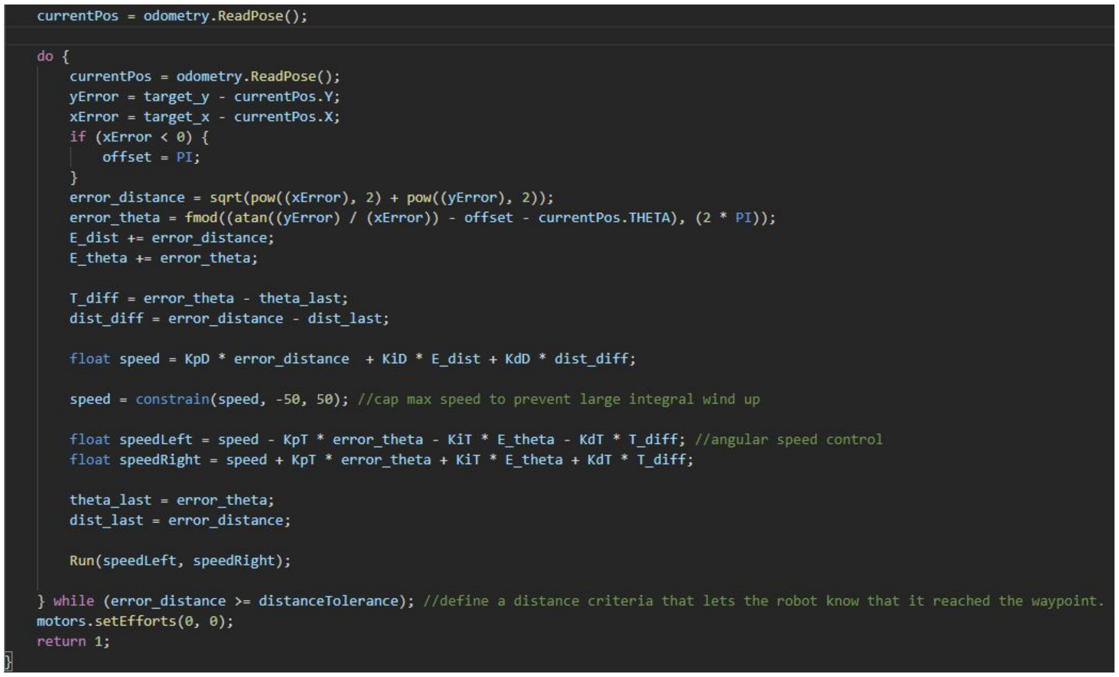
Figure 1:



```
boolean SpeedController::MoveToPosition(float target_x, float target_y){
    //Reset integrals
    E_dist = 0;
    dist_last = 0;

    E_theta = 0;
    theta_last = 0;
    //initial turn to position
    currentPos = odometry.ReadPose();
    yError = target_y - currentPos.Y;
    xError = target_x - currentPos.X;
    offset = 0;
    if (xError < 0){ offset = PI; }
    float turnAngle = atan((yError) / (xError)) - offset - currentPos.THETA;
    int turnDeg = (int) (turnAngle * 180 / PI);
    totalAngle += turnDeg;

    if (turnAngle < 0){
        Turn(abs(turnAngle), 0);
    } else {
        Turn(turnAngle, 1);
    }
    currentPos = odometry.ReadPose();
```



```
    currentPos = odometry.ReadPose();

    do {
        currentPos = odometry.ReadPose();
        yError = target_y - currentPos.Y;
        xError = target_x - currentPos.X;
        if (xError < 0) {
            offset = PI;
        }
        error_distance = sqrt(pow((xError), 2) + pow((yError), 2));
        error_theta = fmod((atan((yError) / (xError)) - offset - currentPos.THETA), (2 * PI));
        E_dist += error_distance;
        E_theta += error_theta;

        T_diff = error_theta - theta_last;
        dist_diff = error_distance - dist_last;

        float speed = KpD * error_distance  + KiD * E_dist + KdD * dist_diff;

        speed = constrain(speed, -50, 50); //cap max speed to prevent large integral wind up

        float speedLeft = speed - KpT * error_theta - KiT * E_theta - KdT * T_diff; //angular speed control
        float speedRight = speed + KpT * error_theta + KiT * E_theta + KdT * T_diff;

        theta_last = error_theta;
        dist_last = error_distance;

        Run(speedLeft, speedRight);

    } while (error_distance >= distanceTolerance); //define a distance criteria that lets the robot know that it reached the waypoint.
    motors.setEfforts(0, 0);
    return 1;
}
```

Figure 1: Screenshot of our `SpeedController::MoveToPosition(float,float)` function.

ii. PID Parameters for distance:
   1. Kp: 2.0
   2. Ki: 0.01
   3. Kd: 0.01

iii. PID Parameters for heading:
   1. Kp: 11.0
   2. Ki: 0.0001
   3. Kd: 3.0

3. **Task 3**
   a. Time for robot to finish parkour at different error distances

Table 2:

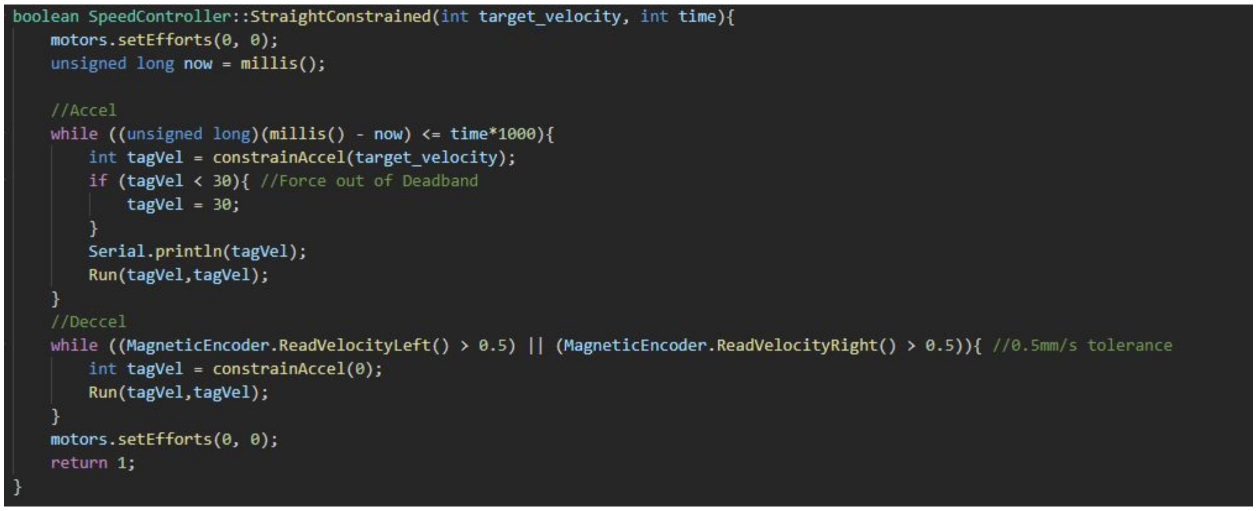
error_distance = 0.02m time	error_distance = 0.03m time
16:19 min	14:27 min

Table 2: Time values in seconds for the robot to finish the parkour, with distance Kp = 20 and theta Kp = 11.

   b. Advantages/disadvantages of loosening/tightening distance criterion
   When the distance criterion is loosened (the error tolerance is greater), the circuit will take less time to complete, however, there will be more accumulated error as each waypoint was not as accurately achieved. Conversely, a tighter distance criterion will take more time to complete but each waypoint will be more accurate, and as a whole, the total error will be less than a loosened criterion.

4. **Extra Point Task**
   a. `SpeedController::Straight(int, int)`
      i. Code Screenshot:

Figure 2:



```
boolean SpeedController::StraightConstrained(int target_velocity, int time){
    motors.setEfforts(0, 0);
    unsigned long now = millis();

    //Accel
    while ((unsigned long)(millis() - now) <= time*1000){
        int tagVel = constrainAccel(target_velocity);
        if (tagVel < 30){ //Force out of Deadband
            tagVel = 30;
        }
        Serial.println(tagVel);
        Run(tagVel,tagVel);
    }
    //Deccel
    while ((MagneticEncoder.ReadVelocityLeft() > 0.5) || (MagneticEncoder.ReadVelocityRight() > 0.5)){ //0.5mm/s tolerance
        int tagVel = constrainAccel(0);
        Run(tagVel,tagVel);
    }
    motors.setEfforts(0, 0);
    return 1;
}
```

Figure 2: Screenshot of our `SpeedController::StraightConstrained(int, int)` function.