VEXCODE VR SKILLS CODE EXPLANATION - 2496X:

Our code for the VEXcode VR Skills Online Challenge was created with efficiency and functionality in mind. To satisfy the criteria of being consistent, high-scoring, and efficient, our code utilizes the PID control loop. This control loop allows the robot to accelerate when further from the target, and decelerate slowly as the robot approaches the target. This control loop is defined at the beginning of our program in the turnPID and drivePID functions.

```
20  def drivePID(desiredValue):
21
22      enableDrivePID = true
23      prevError = 0
24      totalError = 0
25      count = 0
26
27      kP = 1
28      kI = 1
29      kD = 1
30
31      maxI = 500
32
33      integralThreshold = 150
34
35      drivetrain.set_rotation(0,DEGREES)
36
```

THE PID FUNCTION:

This is where the drivePID function is initialized. Inside of the function, before the while loop, the necessary variables are initialized, including the tuning variables (kP, kI, and kD). The drivetrain's rotation sensors are also reset before the motion is run.

In the next section of the PID loop, we write the while loop that allows the values of P, I, and D to refresh, keeping the robot on track to finish its movement accurately and quickly. Inside of the loop, we read the rotation sensor values of the drivetrain, and save it in a variable. This is used to calculate P, which contributes acceleration and deceleration to the movements of the robot depending on how far the robot is from the target. This is done by saving the result of the target distance minus the current value in a variable named error. Moving on to the derivative part of the control loop, this is calculated by subtracting the error of the previous time the loop was run from the current error. Since error reduces over time in an ideal situation, this value should be negative. Hence, this part of the control loop counteracts the proportional, allowing the robot to slow down as needed when approaching the target. The last part of the control loop is the integral. This is calculated by adding the sum of the integral values over the entire movement to the current error. This causes the integral to build up over time, and when it is added to the speed, it ensures the robot is traveling at the optimal speed, so that the movements are as quick as possible. We have implemented some safety measures through the integral threshold and maxI variables. These variables make sure that the

```
38  while (enableDrivePID):
39
40
41      #get current value of chassis motors
42      currentValue = drivetrain.rotation(DEGREES)
43
44      #proportional
45      error = desiredValue - currentValue
46
47      #derivative
48      derivative = error - prevError
49
50      #integral
51      if (abs(error) < integralThreshold):
52          totalError += error
53
54      if (error > 0):
55          totalError = min(totalError, maxI)
56
57      else:
58          totalError = max(totalError, -maxI)
59
60
61      speed = (error * kP + derivative * kD + totalError * kI)
62
```

integral only builds up at times when it is needed and ensures that its value isn't so large that it overshoots the target. All of these parts of the control loop multiplied by their tuning constants (which is calculated manually through trial and error) added together results in the final speed value for the robot.

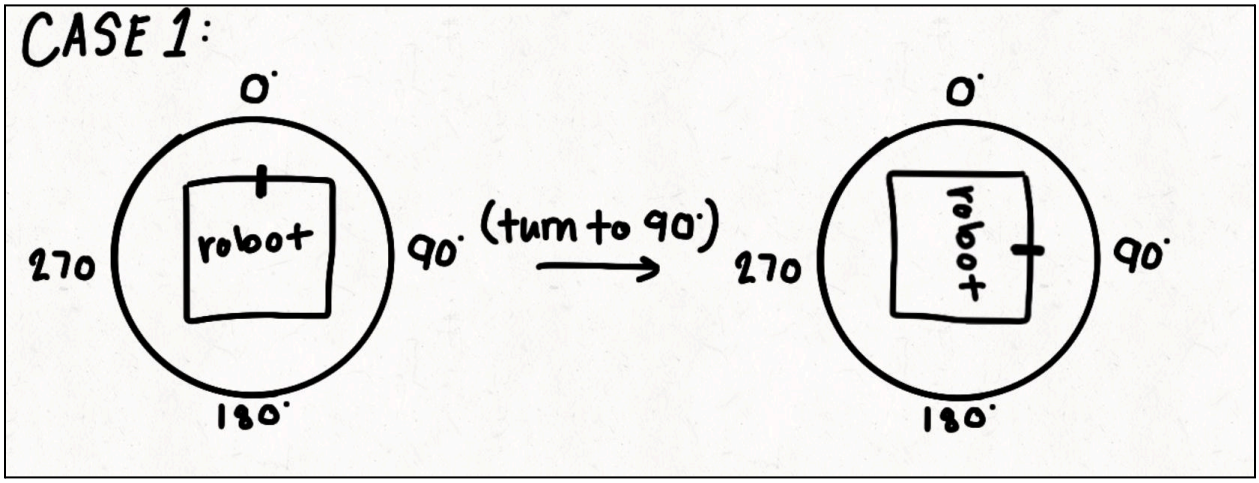
```
63          drivetrain.set_drive_velocity(speed,PERCENT)
64          drivetrain.drive(FORWARD)
65
66          prevError = error
67
68          if (abs(error) < 20):
69              count++
70
71
72          if (count > 35):
73              enableDrivePID = false
74
75
76          delay(20);
77
78
79
80      drivetrain.stop()
81
```

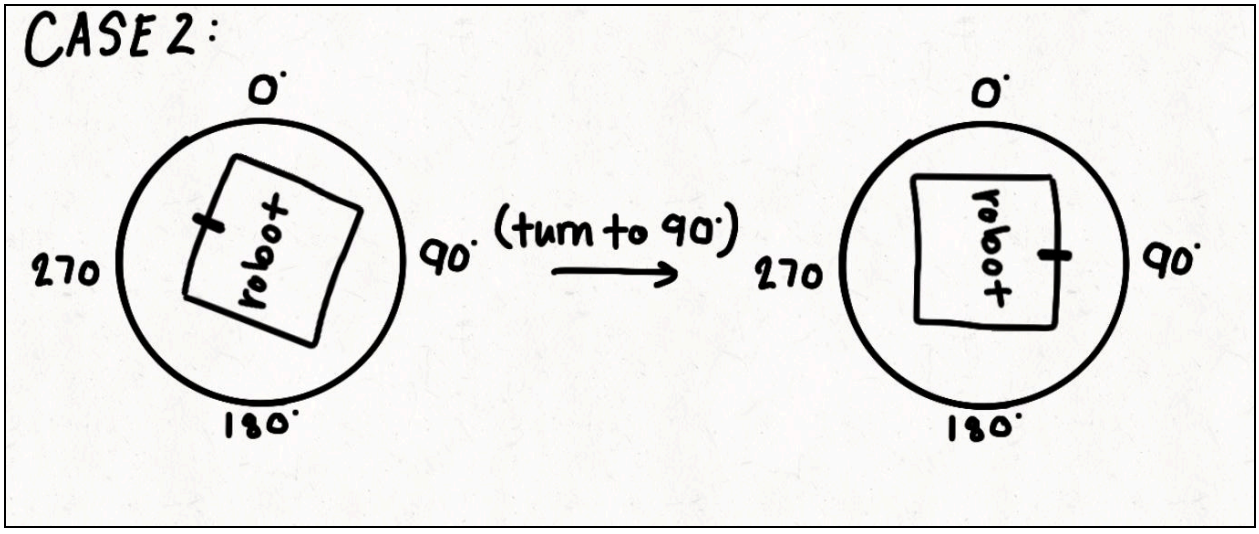
At the end of the while loop, the values are updated for the next loop around. The error, since it has already been used in calculated speed, is passed to the previous error variable. The count variable, used for timing out of the control loop once the movement is complete is updated, and some delay is added at the end of the loop to ensure that the timing of the control loop continues smoothly and does not ruin the rest of the autonomous code. Finally, after the robot breaks out of the PID loop, the drivetrain is stopped to ensure the robot does not move away from the target that we have reached.

This code is mostly duplicated for the turnPID function, with the exception that the robot is turning, moving both sides of the chassis in opposite directions. The final speed value and components of the PID loop are calculated in the same ways. The turnPID function uses the robot's heading to calculate the PID values instead of the rotation sensors that measure distance traveled.

However, the most essential difference between the turn function and drive functions is the reading and wrapping of the robot heading value. The unique reading and processing of the robot's heading value allows the robot to turn to a global heading value. This means that no matter the current heading of the robot, the robot will always end up facing the same direction after the movement is completed.

Here are some examples of this in action:





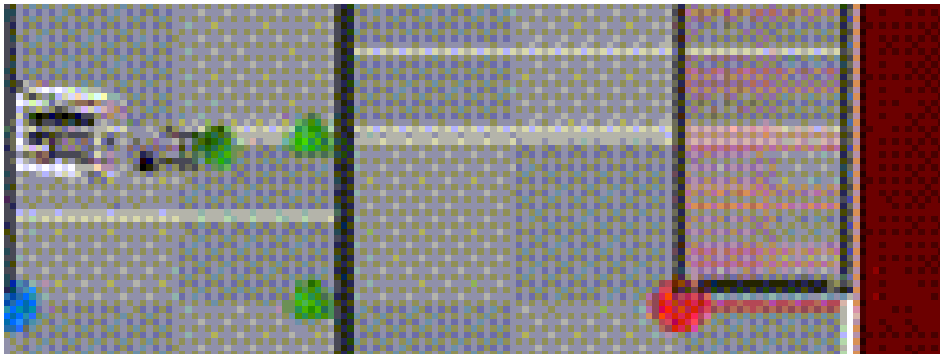
This movement algorithm allows the robot's turns to be ultraprecise, aiding in making the autonomous program as accurate and consistent as possible. Employing this kind of code also is a failsafe for our code. In the event that one turn has messed up and the robot is no longer facing the right direction, the next turn of the program will correct the robot's heading to where it should be and allows the rest of the program to continue and does not interfere with the success of other parts of the code.

This functionality of the robot's turn PID is achieved with the block of code shown at right. These lines of code read the robot's heading and wrap it from -180 to 180 degrees. The heading of the robot is first saved inside of a variable called position. This value is then passed through a variety of checks in the form of conditional statements. In summary, these lines of code calculate the distance needed to travel in each direction to reach the target and then tell the robot to choose the most optimal path possible to ensure the movement is completed quickly and accurately. This information is saved in variables which are then read at the end of the PID loop to help calculate the correct speed which needs to be assigned to the drivetrain motors.

```
124         #get position of inertial and wrap angle
125         position = drivetrain.heading(DEGREES)
126         if (position > 180):
127             position = ((360-position) * -1)
128
129
130         if ((desiredValue < 0) && (position > 0)):
131             if ((position - desiredValue) >= 180):
132                 desiredValue = desiredValue + 360
133                 position = inertial.get_heading()
134                 turnV = (position + desiredValue)
135
136             else:
137                 turnV = (abs(position) + abs(desiredValue))
138
139
140         else if ((desiredValue > 0) && (position < 0)):
141             if ((desiredValue - position) >= 180):
142                 position = inertial.get_heading()
143                 turnV = 360 - desiredValue - abs(position)
144
145             else:
146                 turnV = (position + desiredValue);
147
148
```

THE SKILLS PROGRAM:

Using the PID functions defined at the beginning of the file, we created a highly-efficient and high-scoring autonomous program. The robot in the virtual environment follows a well-planned route to score a large amount of points. We created another function, called whenStarted, which would be called when the program is started, and sent to the robot to be executed. Inside of the whenStarted function, we call the PID functions and pass a distance to be driven accurately. Using these functions for robot movements, the robot executes the pre-planned autonomous route. One of the most unique components of our autonomous strategy is the half court shots that can be seen in our program. By outtaking the triball as the front of the chassis bumps up onto the barrier, we can allow the triball to gain enough speed and height to shoot across the field and go into the goal. This strategy saves us a lot of time in the program and allows us to score a large number of points.



Another creative part of our code is our use of variables, loops, and conditional statements to simplify the code and make its function more efficient and consistent. This is seen in the part of the autonomous program in which the robot takes triballs from the match loading zone and shoots them across the field into the goal for extra points.

As seen in line 279 and 280, some variables are initialized before the loop is started. These are the loop counter and turn distance variables. The loop counter variable is sequenced every time the loop runs once. This allows us to know how many times the loop has run, and break out of the loop according to this value. The other value is the turn distance. This is what is passed to the turnPID function, telling it how much it needs to turn before shooting the triball. Next, we have the loop itself, which will keep on running until the loop counter variable reaches 4. Inside of the loop itself, the PID functions are called to tell the robot to repeat functions. The loop counter variable is sequenced and the turn distance variable is changed every loop around. The turn distance is increased every loop so that the robot is always facing an open spot in the goal before shooting the triball there. This avoids clumps of triballs and increases the chances the triball lands inside of the goal. We have also added an if statement inside of the loop, which checks if the loop has run more than 3 times, and breaks out of the loop before the robot returns to the match loading bar, saving us some time in the program.

```
279    loopcount = 0
280    dist = 90
281    while loopcount<4:
282        drivePID(-1974)
283        turnPID(dist)
284        intake_motor.spin(REVERSE)
285        wait(0.05,SECONDS)
286        drivePID(775)
287        drivePID(-775)
288        intake_motor.spin(FORWARD)
289        loopcount = loopcount+1
290        dist = dist+5
291        if loopcount>3:
292            break
293        turnPID(317)
294        drivePID(1974)
```

WHAT MAKES OUR CODE SPECIAL:

There are many parts of our autonomous program that make it unique from the code of other teams partaking in this challenge. These special parts of our code allow it to function faster and better, giving our team's submission the competitive advantage over other teams.

```
20  def drivePID(desiredValue):
21
22      enableDrivePID = true
23      prevError = 0
24      totalError = 0
25      count = 0
26
27      kP = 1
28      kI = 1
29      kD = 1
30
31      maxI = 500
```

One of these features are the variables. Variables are storage for information in a computer program. By saving some information under a name, we can retrieve this information later in our code and use it for a variety of things.

Variables can be modified at any time. This simplifies the coding process greatly and allows for some extra function in the code. Our code utilizes many boolean and integer variables. We use boolean variables in our code to tell the multiple parts of the PID control loop and robot's subsystems to be on or off. With this, we can properly manage the functions of the autonomous program and ensure that it does not break due to the incoordination of elements. Integer variables are utilized in many parts of the code, such as in saving distance driven, calculating the different parts of the PID, and sequencing variables.

```
281     while loopcount<4:
282         drivePID(-1974)
283         turnPID(dist)
284         intake_motor.spin(REVERSE)
285         wait(0.05,SECONDS)
286         drivePID(775)
287         drivePID(-775)
288         intake_motor.spin(FORWARD)
289         loopcount = loopcount+1
290         dist = dist+5
291         if loopcount>3:
292             break
293         turnPID(317)
294         drivePID(1974)
```

Our code also uses loops. These are built in functions of python that allow for a few lines of code to be repeated. Using this functionality, we can simplify our code and by introducing other features, such as variables and conditional statements, we can make

the code function better as a whole. We also use conditional statements, which are lines of code that only allow a specific block of code to run if the current program meets certain requirements. Our program is centered around these statements, from reading variables, to initiating loops, and other elements of our code. Finally, our code uses functions, which are multiple lines of code that store lines of code to be called whenever necessary. When paired with parameters (which function as guidelines for the function to run), these functions greatly improve the function and readability of the code.

In conclusion, our autonomous program makes use of the industry-level PID control loop and innovative coding solutions to allow our team's VEXcode VR Skills runs to be high-scoring, efficient and accurate. By implementing unique strategies and programming to allow our code to gain a competitive edge over other teams, our team was able to create an autonomous program that places us amongst the top 20 teams in the Global Virtual Skills Standings.