*Following is a guest post by Corey Clements, a student on Team 9779, PIEaters. If you (or someone you know) would like to contribute as a guest author, please*[*let us know*](mailto:team@ftc-tricks.com)*.*

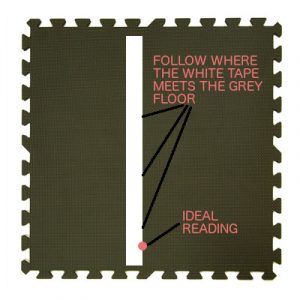
**Introduction**

When we first try to make a line follower, we generally look at a big white line and think,

That’s the line we want to follow. If our robot sees that white line, it needs to turn away from the white line. If it doesn’t see a white line, it needs to turn towards the white line.

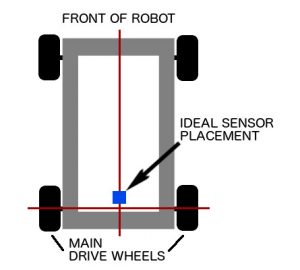
Using this method, our robot will turn back and forth as it shimmies its way down the line. One of the biggest problems with this method is that when we get to the stopping point, there’s no way to predict which way the robot will point. Also, the process of turning back and forth may have left the robot slightly left or slightly right of the line depending on where it was in the line finding process. We quickly realize that we need a more precise method. This is where the proportional line follower comes in.

**What is a Proportional Line Follower?**

The first step in making a proportional line follower is realizing that we are not trying to follow the big white line. Instead, we are trying to follow the exact point where the white line meets the grey tile. In other words, we are trying to follow the “edge” of the line. We’ll discuss this more in a moment.

The next step is to realize that the values returned by our sensors can be used to control motor power. As programmers, we do not need to be directly involved in deciding exactly how much power to give the drive motors. The motors can receive a “correction” to the power setting depending on how far the sensor reading has drifted from a “perfect” value. In other words, if out robot is **not** in the perfect spot above the edge of the line, we can compute how far from perfect it is, and use that value to adjust the drive motors. This is where we get the name *proportional* line follower. The power applied to the motors is directly proportional to how far from perfect our sensor is positioned.

**Where to Place the Color Sensor**

To get the best results out of our line follower, correct placement of the color sensor is crucial.

First, we want to place the sensor slightly in front of the main drive motor axis. The sensor must be in front of this point so that the drive motors can see what’s coming. However, if the sensor is too far forward, every small motor correction will drastically adjust the position of the color sensor.

Next, we should place the sensor pointing straight down at the floor, about 1 inch from the ground. We might need to adjust this up or down depending on the robot and the layout of the field. The height of the sensor makes a huge difference in the values our sensor will return, and it requires some experimentation to figure out what will work best.

Finally, we want to center the sensor left to right on your robot. An off-center position can still work, but it will be best if centered.

**Note**: It is also not a bad idea to shield the sensor from outside lighting leaks. Every robot will need to perform in different lighting conditions; the more we can block out the ambient light, the more consistency we can expect. A typical robot will probably provide enough protection from outside light sources, but if there is an opportunity to shield it from light, we should do it.

**What Sensor to Use and Why**

There are multiple color sensors to choose from. We will pick from the Modern Robotics color sensor, the Adafruit RGB color sensor, and the Modern Robotics optical distance sensor.

|  |  |  |
| --- | --- | --- |
| Modern Robotics ODS | Adafruit RGB | Modern Robotics Color Sensor |
| Modern Robotics Color Sensor  Returns values for Red, Green, Blue, and White  Easy to plug in and get working | Adafruit RGB Color Sensor  Returns values for Red, Green, Blue, and Alpha  You must wire it yourself | Modern Robotics Optical Distance Sensor  Returns reflected light level  Most teams already have this sensor |

First, let’s consider the Modern Robotics color sensor. This sensor returns integer values often ranging from about 1 to 40, and the range of values between the grey mat and the white line can be even less depending on calibration. This range of values is limiting, because we only get a few values between completely on grey and completely on white. This isn’t much data to make effective corrections, so we may want to use a different sensor.

Next, the Adafruit RGB color sensor. This sensor will return a much wider range of values, an integer value from 0 to 65,535. That would give us more detail to create correction values. The only problem with this sensor is that by default, its refresh rate (the number of times it takes in a new value) is set to about 600 milliseconds. It takes some programming work to get this value lower, so we won’t use the Adafruit RGB color sensor in our example.

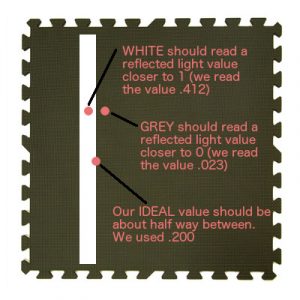
Lastly, let’s look at the Modern Robotics optical distance sensor. It may seem odd that we would use a distance sensor to read color values, but this sensor actually reads the reflected light value. White returns a greater reflected light value, and grey returns a lower reflected light value. These values are represented as a double (decimal) value from 0 to 1. Our example sensor returns 0.015 for grey and 0.400 for white, so that gives us even more detailed values to work with. Also, it is much easier to use our correction since the motors also run with decimal values. As a bonus, the optical distance sensor is included in the default electronics kit from Modern Robotics. The sensor you use is your own personal choice, but we will use the optical distance sensor for our examples.

**Declaring Hardware**

Here are some of the variable declarations we will be using in our example. We will use two motors, the optical distance sensor, double variables to hold the values of the left & right powers, a double variable to hold the value of the correction, and a constantdouble to hold the value of the perfect color value (which we will discuss in the next paragraph).

|  |
| --- |
| DcMotor leftWheel, rightWheel;  double leftPower, rightPower, correction;  final double PERFECT\_COLOR\_VALUE = 0.2;  OpticalDistanceSensor lightSensor; |

**What to do First**

First, we will determine the color sensor value that represents all grey, all white, and “perfect” (the point where grey and white meet). To accomplish this, we will display the color sensor values on the driver station phone using telemetry. Take note of the color sensor readings while the sensor is over each section. We will use these values in our program.

|  |
| --- |
| // Code for the telemetry, which displays the light detected by  // the sensor on the Driver Station phone.  telemetry.addData("Color Value", lightSensor.getLightDetected()); |

**Programming the Line Follower**

First, assuming we are creating a Linear OpMode, let’s create a while loop. We need a loop so we can constantly check our light sensor readings and adjust accordingly. In our example, we will have the loop run forever.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | while (true) {      // Line follower code here  } | |

To create our line follower, we will need to get a “correction”. The correction is basically how far from the PERFECT\_COLOR\_VALUE our current sensor reading is. The correction will be added onto the appropriate motor’s power to steer the robot towards the perfect value. The closer the sensor reading is to the perfect value, the closer the correction is to zero, so less power will be added to the motor, and the robot will steer less. When the color sensor reading is equal to the perfect value, there should be no correction at all, and the robot will go straight.

|  |
| --- |
| correction = (PERFECT\_COLOR\_VALUE - lightSensor.getLightDetected()); |

Here we have set our correction variable to the PERFECT\_COLOR\_VALUE minus the amount of light detected. In this example the perfect value is 0.2, but your value will be whatever you saw from the telemetry in the previous step. The optical distance sensor returns a value closer to 0.0 when it is over grey, and a value closer to 0.4 when it is over white.

So let’s say our program is going to follow the point where the right side of the white line meets the grey mat. If the robot is over grey, we want it to turn left. If it is over white, we want it to turn right. If the optical distance sensor returns a value of 0.01, we know we are over grey, and want to turn more aggressively to the left. The PERFECT\_COLOR\_VALUE of 0.2, minus the sensor reading of 0.01, equals 0.19, a large correction. When we add this correction to the right motor’s base power, the robot will steer aggressively to the left.

If the robot is somewhere near the center of the line, let’s say 0.15, we can tell we are getting close. We are still in the grey, so we still want to turn left to get closer to that perfect value of 0.2. But we need a less dramatic turn. Our correction would be the PERFECT\_COLOR\_VALUE of 0.2, minus the color sensor reading of 0.15. This gives us a correction of 0.05, which is a lower correction added to our right motor’s base power.

Now let’s say the robot is over white, and our sensor is returning a reading of 0.35, which is pretty heavy into white. Our correction would be the PERFECT\_COLOR\_VALUE of 0.2 minus the color sensor reading of 0.35. This would give us a correction of  -0.15, which is a pretty large correction, but negative. A negative number means that we need to turn right, so we’ll add our correction to the left motor (we’ll deal with the negative in a second).

We need a way of checking if our correction value is positive (left turn) or negative (right turn). We are going to use an if else statement, checking if the correction is negative. If it is, we know we need to turn right. In this case, we can just subtract the correction from the left motor’s base power, because remember when you subtract a negative, it’s like adding a positive. Else, we know we need to turn left by adding the correction to the right motor’s base power.

|  |
| --- |
| // Sets the powers so they are no less than .075 and apply to correction  if (correction <= 0) {     leftPower = .075d - correction;     rightPower = .075d;  } else {     leftPower = .075d;     rightPower = .075d + correction;  } |

Here, we apply the correction to the power variable. First off, we don’t want our robot stopping, so we will have a minimum base power of 0.075. The correction is added to this power. One more check, what if the correction was zero? In the if statement, we ask if the correction is less than or equal to zero, and it is zero, so we go into that statement. No correction is applied, so we go forward without turning at all.

The final step is to actually set the power to the motors. We will give the motors the variables of leftPower and rightPower that we just stored in the previous step.

|  |
| --- |
| // Sets the powers to the motors  leftMotor.setPower(leftPower);  rightMotor.setPower(rightPower); |

**Putting it Together**

Here is the loop with all of the pieces put together:

|  |
| --- |
| while (true) {      // Get a correction      correction = (PERFECT\_COLOR\_VALUE - lightSensor.getLightDetected());        // Sets the powers so they are no less than .075 and apply to correction      if (correction <= 0) {          leftPower = .075d - correction;          lightPower = .075d;      } else {          leftPower = .075d;          rightPower = .075d + correction;      }        // Sets the powers to the motors      leftMotor.setPower(leftPower);      rightMotor.setPower(rightPower);  } |

**Final Thoughts**

There are just a few differences between the example code here and the proportional line follower we’ve used in competition.

In our example, we just put everything in a while loop that runs forever. For our real code, we created a method with the line following code inside of it. In our runOpMode() code block, we created a while loop that calls the method. We can end that while loop with whatever condition we want (time, finding a different color on the mat, ultrasonic sensor reading, etc.).

Also, in our example, we didn’t create a variable for the base power of our motors. In our real code, we added a constant to keep track of the base power, called MOTOR\_BASE\_POWER. This allows us to change the base power constant in our variable declaration statements, without having to find it in the program and change all the instances of the base power.

When our team first created our proportional line follower, we actually used all three sensors mentioned (Adafruit RGB color sensor, Modern Robotics color sensor, and Modern Robotics optical distance sensor). We ran into problems with the Adafruit RGB and the Modern Robotics color sensor. As we mentioned earlier, the Modern Robotics color sensor doesn’t return a wide range of values. It is our understanding that the raw data of this sensor returns values closer to the Adafruit’s range, but it is difficult to go through the steps to obtain the raw data. The Adafruit RGB color sensor returns a larger range of values, but it had a different problem. After testing this sensor, it seemed that we weren’t getting readings frequent enough. After some research, we found out that by default, it’s refresh rate is about 600 milliseconds. You can change this as well, but there are extensive steps involved. We found the information we needed to fix the refresh rate at [Ollie’s Workshop](http://olliesworkshops.blogspot.com/2015/10/i2c-color-sensor-for-ftc-on-qualcomm.html). Lastly, the Optical Distance Sensor works great as a reflected light sensor. For a line follower, reflected light works great.

We hope this gets your robot following the line, and that you have a successful season!

<https://ftc-tricks.com/proportional-line-follower/>