Using Encoders to Drive Straight

From ROBOTC API Guide
< Tutorials | Arduino Projects/Mobile Robotics/VEX
Arduino → Arduino Tutorials and Guided Projects → VEX + Arduino, Mobile Robotics Platform → Tutorials/Arduino Projects/Mobile Robotics/VEX/Using encoders to drive straight

Contents

Why is the robot not going straight?

You have probably noticed by now that your robot is often having trouble driving straight. This is because the robot's wheels do not go the same speed. If each wheel does not go at exactly the same speed, the robot is not driving straight but it is in fact executing a very wide swing turn. "But I am telling them to go the same speed!" you may say. "The power value I am sending is the same for each wheel!"

Unfortunately, even if you send the same power value to each wheel, this will absolutely not guarantee that each wheel will move at the same speed. Why? Because of tiny variables such as friction resistance and manufacturing differences, one motor at power 50 will not go at the same speed as another motor at power 50.

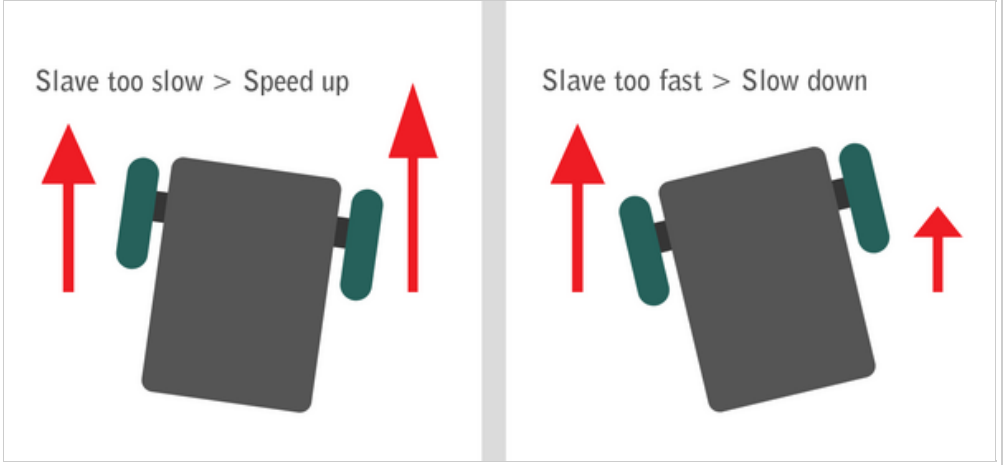
How can I fix this?

Luckily, your robot is now equipped with encoders which means that you know how much your wheels turn. If you take the amount the wheels turn within a specific timeframe, say 1 second, and find they go a certain amount of encoder ticks, say 300, you can easily determine the speed of the wheel. Now that you know the speed of each wheel, we can compare them, and apply more or less power to one wheel to catch up with/slow down to the level of the other motor. This will make your robot drive much more straight.

The Theory

This is called 'proportional control' and is one of the more useful things you will learn when programming a robot. The important thing to accept is that we will need to use one motor as a 'master motor', to which a constant power is applied, as we usually do, and a 'slave motor', whose power we will change to make sure it goes at the same speed as the master motor. It doesn't matter which side you choose, but we will use the left motor as the master motor and the right motor as the slave.

Basically, think of it this way: the master is going along at his own pace. This pace might change slightly when he gets tired or has to climb over an obstacle. The slave's job is to keep alongside the master by speeding up when he falls behind, and slowing down when he goes too far ahead.



If one side is going too fast, slow it down. If it is going too slow, speed it up.

The difference between the master's speed and the slave's speed is called the 'error'. If they are going along at exactly the same pace, the error value will be zero. If the slave is too slow, the error is positive. If the slave is going too fast, the error is negative. Error is very simple to calculate, and can be defined as:

Error = Speed of master - Speed of slave.

After the error is found, it should be added onto the power value of the slave motor, so that it will go faster by an appropriate amount. However, there is something important to do first - the error should be multiplied by a number called the *Constant of Proportionality*, usually expressed as 'kp'. What this does is converts the difference in encoder speeds (error) into something that can be used to adjust the motor power.

For example: say we find that the master encoder is ticking at 300 ticks per second, and the slave encoder is ticking at 250. We apply the error formula and find that the error = 300 - 250 = 50. However, we don't want to add 50 straight onto the motor power! It will overcompensate far too much and zoom ahead, and the next time it calculates the error it will speed backwards, overcompensating again. To fix this, we need to multiply the error by a value, which here is kp. If kp is, say, 0.2, we multiply the error value (50) by kp (0.2) and get the result: 50 * 0.2 = 10. If we were to add 10 to the slave motor power instead of 50, this would give us a much more reasonable increase in speed.

There is no magic way to determine the best value of kp. You simply need to use trial and error to find a value that will result in neither overcompensation nor under compensation. This is refereed to as 'tuning' kp.

In our circumstances, using the Arduino UNO with ROBOTC, we cannot use floating point numbers, which means that a value of, say, 0.2 for kp is not possible. However, if you consider that multiplying by 0.2 is the same as dividing by 5, then we can use this property and set kp so that the error is *divided* by a whole number instead of *multiplied* by a decimal.

So, we have added a suitable value to the power of the slave motor to compensate for the error. Now, we need to do it again, looping around a certain amount of times per second. For this application, ten times per second is plenty.

The Code

Let's write a program encompassing the concepts we have just talked about. This will make the robot drive straight indefinitely.

```
task main()
{
  //The powers we give to both motors. masterPower will remain constant while slavePower will change so that
  //the right wheel keeps the same speed as the left wheel.
  int masterPower = 30;
  int slavePower = 30;

  //Essentially the difference between the master encoder and the slave encoder. Negative if slave has
  //to slow down, positive if it has to speed up. If the motors moved at exactly the same speed, this
  //value would be 0.
  int error = 0;

  //'Constant of proportionality' which the error is divided by. Usually this is a number between 1 and 0 the
  //error is multiplied by, but we cannot use floating point numbers. Basically, it lets us choose how much
  //the difference in encoder values effects the final power change to the motor.
  int kp = 5;
```

```
  //Reset the encoders.
  SensorValue[leftEncoder] = 0;
  SensorValue[rightEncoder] = 0;

  //Repeat ten times a second.
  while(true)
  {
    //Set the motor powers to their respective variables.
    motor[leftServo] = masterPower;
    motor[rightServo] = slavePower;

    //This is where the magic happens. The error value is set as a scaled value representing the amount the slave
    //motor power needs to change. For example, if the left motor is moving faster than the right, then this will come
    //out as a positive number, meaning the right motor has to speed up.
    error = SensorValue[leftEncoder] - SensorValue[rightEncoder];

    //This adds the error to slavePower, divided by kp. The '+=' operator literally means that this expression really says
    //"slavePower = slavepower + error / kp", effectively adding on the value after the operator.
    //Dividing by kp means that the error is scaled accordingly so that the motor value does not change too much or too
    //little. You should 'tune' kp to get the best value. For us, this turned out to be around 5.
    slavePower += error / kp;

    //Reset the encoders every loop so we have a fresh value to use to calculate the error.
    SensorValue[leftEncoder] = 0;
    SensorValue[rightEncoder] = 0;

    //Makes the loop repeat ten times a second. If it repeats too much we lose accuracy due to the fact that we don't have
    //access to floating point math, however if it repeats to little the proportional algorithm will not be as effective.
    //Keep in mind that if this value is changed, kp must change accordingly.
    wait1Msec(100);
  }
}
```

This may look like a long program, but without the comments it is only 18 lines long.

Run the program and you will see your robot drive much straighter than it has before.

Driving straight for a distance

We are now going to use this knowledge and apply it to the previous lesson by writing a function that will drive straight for a certain distance - very useful, indeed.

Recall the function driveDistance from the previous lesson:

```
void driveDistance(int tenthsOfIn, int power)
{
  sensorValue[leftEncoder] = 0; // It is good practice to reset encoder values at the start of a function.

  //Calculate tenths of an inch by multiplying the ratio we determined earlier with the amount of
  //tenths of inches to go, then divide by ten as the ratio used is for an inch value.
  //Since we don't want to calculate every iteration of the loop, we will find the clicks needed
  //before we begin the loop.
  int tickGoal = (42 * tenthsOfIn) / 10;

  while(abs(SensorValue[leftEncoder]) < tickGoal)
  {
    motor[leftServo] = power;  // We can now set the power from the function's second parameter.
    motor[rightServo] = power;
  }
  motor[leftServo] = 0; // Stop the loop once the encoders have counted up the correct number of encoder ticks.
  motor[rightServo] = 0;
}
```

Remember to put this at the top of the program, below the configuration code:

```
#define abs(X) ((X < 0) ? -1 * X : X)
```

See how, within the while loop in driveDistance, we simply set the motor powers? If we merge this with our drive straight code, we will get the new function:

```
void driveStraightDistance(int tenthsOfIn, int masterPower)
{
  int tickGoal = (42 * tenthsOfIn) / 10;

  //Initialise slavePower as masterPower - 5 so we don't get huge error for the first few iterations. The
  //-5 value is based off a rough guess of how much the motors are different, which prevents the robot from
  //veering off course at the start of the function.
  int slavePower = masterPower - 5;

  int error = 0;

  int kp = 5;

  SensorValue[leftEncoder] = 0;
  SensorValue[rightEncoder] = 0;

  //We still only have to monitor only one encoder as we have made it so that they will have the same values anyway.
  while(abs(SensorValue[leftEncoder]) < tickGoal)
  {
    //Proportional algorithm to keep the robot going straight.
    motor[leftServo] = masterPower;
    motor[rightServo] = slavePower;

    error = SensorValue[leftEncoder] - SensorValue[rightEncoder];

    slavePower += error / kp;

    SensorValue[leftEncoder] = 0;
    SensorValue[rightEncoder] = 0;

    wait1Msec(100);
  }
  motor[leftServo] = 0; // Stop the loop once the encoders have counted up the correct number of encoder ticks.
  motor[rightServo] = 0;
}
```

Will this work? No. The while loop can never trigger! We reset both encoder values every iteration, so the encoder tick count never even gets near the threshold.

To fix this, we will need to add another variable called 'totalTicks', which will add the encoder values every time the loop iterates. Therefore, at any time, it will have a value equal to the total encoder ticks since the function was called. It is very simple to implement:

```
void driveStraightDistance(int tenthsOfIn, int masterPower)
{
  int tickGoal = (42 * tenthsOfIn) / 10;

  //This will count up the total encoder ticks despite the fact that the encoders are constantly reset.
  int totalTicks = 0;

  //Initialise slavePower as masterPower - 5 so we don't get huge error for the first few iterations. The
  //-5 value is based off a rough guess of how much the motors are different, which prevents the robot from
  //veering off course at the start of the function.
  int slavePower = masterPower - 5;

  int error = 0;

  int kp = 5;

  SensorValue[leftEncoder] = 0;
  SensorValue[rightEncoder] = 0;

  //Monitor 'totalTicks', instead of the values of the encoders which are constantly reset.
  while(abs(totalTicks) < tickGoal)
  {
    //Proportional algorithm to keep the robot going straight.
    motor[leftServo] = masterPower;
    motor[rightServo] = slavePower;

    error = SensorValue[leftEncoder] - SensorValue[rightEncoder];

    slavePower += error / kp;
```

```
    SensorValue[leftEncoder] = 0;
    SensorValue[rightEncoder] = 0;

    wait1Msec(100);

    //Add this iteration's encoder values to totalTicks.
    totalTicks+= SensorValue[leftEncoder];
  }
  motor[leftServo] = 0; // Stop the loop once the encoders have counted up the correct number of encoder ticks.
  motor[rightServo] = 0;
}
```

And that is our function! We now have the ability to easily drive any number of tenths of an inch, and the robot will drive straight throughout the distance. Let's use this in a program, where the robot should eventually end up back at its starting point.

```
#pragma config(CircuitBoardType, typeCktBoardUNO)
#pragma config(UART_Usage, UART0, uartSystemCommPort, baudRate200000, IOPins, dgtl1, dgtl0)
#pragma config(Sensor, dgtl2,  rightEncoder,    sensorQuadEncoder)
#pragma config(Sensor, dgtl7,  leftEncoder,     sensorQuadEncoder)
#pragma config(Motor,  servo_10,        rightServo,    tmotorServoContinuousRotation, openLoop, reversed, IOPins, dgtl10, None)
#pragma config(Motor,  motor_11,        leftServo,     tmotorServoContinuousRotation, openLoop, IOPins, dgtl11, None)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#define abs(X) ((X < 0) ? -1 * X : X)

void driveStraightDistance(int tenthsOfIn, int masterPower)
{
  int tickGoal = (42 * tenthsOfIn) / 10;

  //This will count up the total encoder ticks despite the fact that the encoders are constantly reset.
  int totalTicks = 0;

  //Initialise slavePower as masterPower - 5 so we don't get huge error for the first few iterations. The
  //-5 value is based off a rough guess of how much the motors are different, which prevents the robot from
  //veering off course at the start of the function.
  int slavePower = masterPower - 5;

  int error = 0;

  int kp = 5;

  SensorValue[leftEncoder] = 0;
  SensorValue[rightEncoder] = 0;

  //Monitor 'totalTicks', instead of the values of the encoders which are constantly reset.
  while(abs(totalTicks) < tickGoal)
  {
    //Proportional algorithm to keep the robot going straight.
    motor[leftServo] = masterPower;
    motor[rightServo] = slavePower;

    error = SensorValue[leftEncoder] - SensorValue[rightEncoder];

    slavePower += error / kp;

    SensorValue[leftEncoder] = 0;
    SensorValue[rightEncoder] = 0;

    wait1Msec(100);

    //Add this iteration's encoder values to totalTicks.
    totalTicks+= SensorValue[leftEncoder];
  }
  motor[leftServo] = 0; // Stop the loop once the encoders have counted up the correct number of encoder ticks.
  motor[rightServo] = 0;
}


task main()
{
  //Distances specified in tenths of an inch.

  driveStraightDistance(62,30);
  wait1Msec(500);              //Stop in between to prevent momentum causing wheel skid.
  driveStraightDistance(54,-30);
  wait1Msec(500);
  driveStraightDistance(87,30);
  wait1Msec(500);
  driveStraightDistance(95,-30);
}
```

Beautiful, isn't it? Extremely accurate and consistent, regardless of surface friction or other variables. If you are using the robot on a smooth surface and/or decide to increase the speed, you may notice a bit of inaccuracy as your tires may skid after the sudden stop. Overall, however, this is a superior method for driving accurately.
