An Analysis of the RobotDrive Class *drive()* Method

by

FRC Team 4579
Jaden Bottemiller, Lead Programmer
Joe Hafner, Mentor

April 25, 2015

An Analysis of the RobotDrive Class *drive()* Method

Introduction

The RobotDrive Class in the wpi library has a wide variety of constructors and methods to handle almost any driving tasks that an FRC team needs to control their drivetrain. It handles different types of motor controllers, different numbers of motors, arcade driving with a single joystick, tank driving with two joysticks, and even the complex mecanum drive system. As a class, its designers are to be commended for the breadth of options that it provides. The class has a *drive()* method, designed for differential steering of a two motor robot. The javadoc information for the method is very brief, saying that "*outputMagnitude* is the forward component of the output magnitude to send to the motors, and that *curve* is the rate of turn, constant for different forward speeds." But specifically, what values of *outputMagnitude* and *curve* should we use to cause predicted robot behavior. In the 2015 Recycle Rush game, we needed to position our robot at 45 degrees from the baseline player station wall in order to be close to a yellow tote. Then we needed to turn right 45 degrees, pivoting just right to line up with the tote to pick it up and carry it to the Auto Zone. Clearly there wasn't enough information in the javadocs to predict what values of the parameters would let us achieve our desired motion. We looked at the comments in the code, but that didn't provide much enlightenment, except that we knew the parameters needed to be between -1 and +1. We tried a few combinations of values for the parameters, but we were unable to find the right combination because like many teams, we had very limited test time, we had a certain amount of slippage on a hard surface that reduced the accuracy of testing, and the robot was changing configuration as we progressed. In order to effectively use the *drive()* method in our control software, we needed to get a theoretical understanding of how it works.

After completing the analysis shown below, we developed the following information to share as our most important findings of how to effectively use the *drive()* method.

Findings

- Use a negative value of *curve* to turn left, positive to turn right, and zero to go straight, either forwards or backwards.
- The speed of the outside wheel in a turn is set by the input parameter *outputMagnitude*.
- The speed of the inside wheel is set by reducing *outputMagnitude* by a number calculated in the code.
- To get a large turning radius requires very small values for *curve*. For example, for a radius of 10 times the wheel base, *curve* would need to be 0.0000454.
- To get a specific turning radius, pre-calculate the value of the *curve* parameter using $curve = e^{-radius/wheelbase}$, where *radius* is the desired turn radius, and *wheelbase* is the distance between the drive wheels of your robot.
- If *curve* is set to 0.60653, your robot will spin around the inside wheel, which will be stopped.
- If *curve* is set to 1.0, your robot will pivot clockwise around the center point between the drive wheels. -1.0 will pivot the other way.
- If your robot is moving through the turn too fast or slow, adjust *outputMagnitude* only, but don't change *curve*.
- Joystick inputs can be used to directly turn the robot, but can't provide a turning radius larger than about 6.2 times the wheel base. For a 20" wheel base though, this would be a turn radius of over 10 feet, which should be more than adequate for normal play field dimensions and driver needs.
- Change the value of *m_sensitivity* to change the value of *curve* that will make the robot pivot around its inside wheel. For example, if *m_sensitivity* is set to 0.8, the robot will pivot around its inside wheel if *curve* is 0.45.

The remainder of this paper looks at robot differential steering and how the code implements it in the *drive()* method. It is somewhat mathematical, because that is the nature of differential steering. If you'd like to ignore the math, just use the Findings listed above, and check out the Conclusions on page 5. That should cover all the important points to help you use the *drive()* method to its full capabilities. In the following paragraphs, we look at the theory of differential steering, we see how the formulas or equations in the code behave and how they match the theoretical version, and we see some specific values of *curve* make your robot behave. This all leads to the Findings and Conclusions in this paper.

Robot Model
Consider Figure 1, the diagram of a robot with two motors and two wheels, and differential steering.

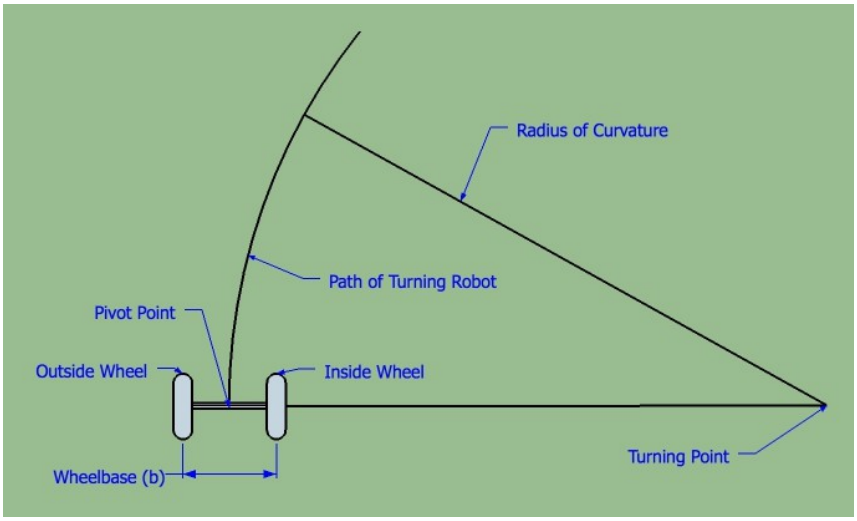

Figure 1, Differential Steering Model

If we define the speed *S* of the robot as the speed of the pivot point between the two wheels, we can see that the speed of the robot is given by the equation

$$S = R \times \omega \ , \tag{1}$$

where *R* is the Radius of Curvature or turning radius, the distance between the pivot point and turning point, and ω (omega) is the rotational speed of the robot around the Turning Point, in revolutions per second. Clearly, in a turn, the outside wheel will have to move faster than the pivot point, and the inside wheel will have to move slower. The speed of the outside wheel is equal to

$$So = (R + \frac{b}{2}) \times \omega \ , \tag{2}$$

and the inside wheel is

$$Si = (R - \frac{b}{2}) \times \omega \ . \tag{3}$$

In Equations 2 and 3, *b* represents the width of the robot wheel base, or the distance between the two drive wheels. We divide that quantity by *2* since our reference point is the center of the wheelbase, the Pivot Point.

The ratio between the two wheel speeds is given by the following relationship.

$$\frac{Si}{So} = \frac{(R - \frac{b}{2})}{(R + \frac{b}{2})} \tag{4}$$

Using this equation we can set the speed of either wheel based on the speed of the other for a specific turn radius.

Based on this understanding of how differential steering operates, we next examine how the *drive()* method in RobotDrive uses its input parameters to control steering.

Source Code
The entire source code for the RobotDrive class is available on line at https://usfirst.collab.net/sf/projects/wpilib/, and need not be replicated here. The *drive()* method is not too long, and is displayed in Appendix A. A scan of the code shows there are basically three important *if-else* clauses. One is for *curve=0,* and therefore drives straight. The other two are for *curve* less than zero and greater than zero. There are really only four lines to understand to see how the code works. This is the *if* clause for when *curve<0.*

```
1.          double value = Math.logarithm(-curve);
2.          double ratio = (value - m_sensitivity) / (value + m_sensitivity);
3.          leftOutput = outputMagnitude / ratio;
4.          rightOutput = outputMagnitude;
```

In Line 1, we see *value* is the natural logarithm of *curve*. And if *curve* is negative, we take the logarithm of -*curve* because we can't normally take the logarithm of a negative number. Here is a graph of what is happening.

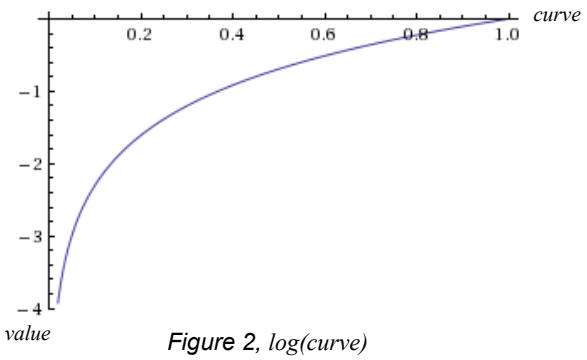

Figure 2, log(curve)

Figure 2 shows us how values of *curve* that are limited to a range of 0 to +1 are mapped to a range of 0 to -∞, which is the range necessary for a turn radius. Also from Figure 2 we see that small values of *curve* are mapped to large (negative) values of turn radius. A turn radius of infinity is the same as going straight. We also see that *value* will always be a negative number.

In line 2, the variable *ratio* is calculated as *ratio = (value - m_sensitivity) / (value + m_sensitivity)*. This statement looks suspiciously similar to Equation 4, which provides the ratio between the wheel speeds of a turning robot. Where Equation 4 has the value *b/2*, the code has the variable *m_sensitivity*. *m_sensitivity* is a class variable with a default value of 0.5 and is a settable parameter. Where Equation 4 has the turning radius *R*, the code has the variable *value*, which we already know is the turn radius from Line 1.

From the formula in the code, and Figure 2 above, we see that to specify a particular turning radius, we need to pre-calculate a value for *curve* using the negative radius as an exponent of the natural number *e*, because the code uses the natural logarithm. This can be done on a scientific calculator.

$$curve = e^{-radius} \tag{5}$$

This will yield a number for curve between 0 and 1. Then we set the answer to a positive or negative value depending on whether we want to turn right or left. It turns out that if *curve* is equal to $\sqrt{1/e}$, approximately 0.60653, then *value* will be exactly -0.5, and the denominator in the *ratio* formula will be zero. There should be code to avoid the case where *value* equals *m_sensitivity*, so there won't be a division by zero in the *ratio* calculation, but the designers probably felt that it would be very rare for an Autonomous Command to set *curve* to precisely $\sqrt{1/e}$,and a joystick can't do it.

Since we know that *ratio* is calculated from *value*, and *value* is calculated from *curve,* we can look at the relationship between *ratio* and *curve.* Figure 3 shows how *ratio* is calculated from *curve*.

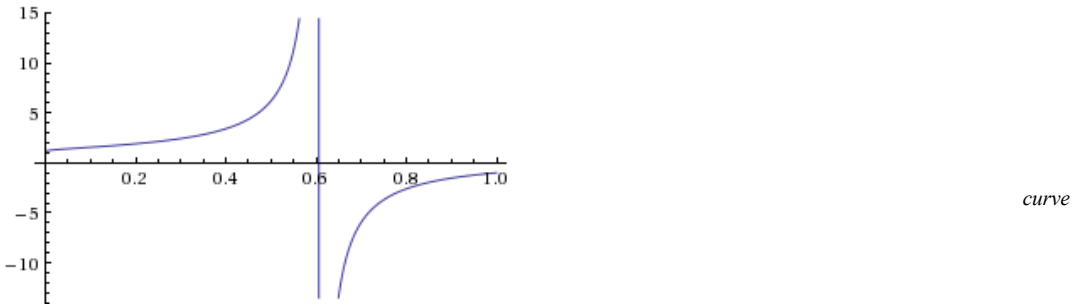

Figure 3, ratio vs curve

Lines 3 and 4 of the code use *outputMagnitude* to set the wheel speed, and use *ratio* as a divisor to adjust the speed of the inside wheel. From Figure 3, if *curve* is 0, then *ratio* is 1. Both motors are going the same speed. As *curve* increases, *ratio* also increases. Since *ratio* is used as a divisor on the motor speed in Line 3, we know that larger values of *ratio* will make the affected wheel slower, therefore it must be the inside wheel. We see that larger values of *curve* cause larger values of *ratio*, thus sharper turning. The vertical line at 0.60653 is because *m_sensitivity* is set to 0.5 as a default, and it's where *ratio* switches from positive to negative, literally wrapping around from +∞ to -∞. In terms of the robot, when *curve* is set to 0.60653, the inside wheel will be stopped because *ratio* will be near ±∞, and the robot will pivot around that wheel. Further increases in *curve* move the turning radius inside the robot wheel base and have the inside wheel going backwards as indicated by the sudden negative values of *ratio*. From Figure 3, when *curve* is 1, *ratio* is -1, indicating that the two wheels will be spinning in opposite directions at the same speed.

m_sensitivity is a settable parameter in the RobotDrive class. Increasing *m_sensitivity* makes the robot turn more sharply with the same value of *curve*. Using this, we can predetermine what value of *curve* we want to use to cause the robot to pivot about it's inside wheel. For example, if *m_sensitivity* is set to 0.8, the robot will pivot on the inside wheel if *curve* is equal to 0.45.

The Missing Element

There is one last consideration that isn't obvious from just looking at the code. The designers of the *drive()* method did not know the dimensions of your physical robot wheel base. They used a mathematical concept called “normalization” to eliminate the need to know about every robot that might use the *drive()* method. In this case, they assumed that the wheel base is a value of 1.0. It doesn't matter if the wheel base is 1 inch or 1 foot or 1 meter. The cleverness of the code is that it assumes that the wheel base is the normalizing value. So for example, if your robot has a wheel base of 20”, then in effect, 20” is a unit of 1 for the equations in the code. That means, for example, that *m_sensitivity*, with a normalized value of 0.5, is really referring to 10” on your actual 20” robot. It also means that *value*, the apparent turning radius, should really be multiplied by your actual wheel base, 20”, to see what your actual turning radius will be for your robot. Equation 5 needs to be adjusted for this as well.

$$curve = e^{-radius/wheelbase} \tag{6}$$

Conclusions

Because we couldn't figure out how to use *drive()* to control our differential steering in a predictable way, we were in the process of creating a new method that would implement Equations 1 through 4 directly. However, with our new understanding of the existing *drive()* method, we can just use it, and correctly set the parameters to achieve what we want. Knowing the turn radius and the speed of the outside motor allows us to correctly calculate the expected rotational speed around the turning point in revolutions per second, and use that to compare with the rate value from a gyro on the robot. That provides a new level of control for both autonomous and manual driving where we control the rate of turn to match an expected value, so we can move more precisely even in the face of wheel slippage.

We would like to see the *ratio* calculations inverted and used as a multiplier to avoid the divide by zero issue, but it doesn't seem to be a particularly nagging problem.

Mostly, we would like to see the javadocs provide better documentation for the method. Here is an example:

```
/*
 * Drive the motors at "outputMagnitude" and "curve".
 * Both outputMagnitude and curve are -1.0 to +1.0 values, where 0.0 represents stopped and not turning.
 * Curve < 0 will turn left.
 * The algorithm for steering provides a constant turn radius for any normal speed range, both forward and
 * backward. Increasing m_sensitivity causes sharper turns for fixed values of curve.
 *
 * This function will most likely be used in an autonomous routine.
 *
 * @param outputMagnitude The speed setting for the outside wheel in a turn, forward or backwards, +1 to -1.
 * @param curve The rate of turn, constant for different forward speeds. Set curve<0 for left turn.
 *   Set curve = e^(-r/w) to get a turn radius r for wheelbase w of your robot.
 *   Conversely, turn radius r = -ln(curve)*w for a given value of curve and wheelbase w.
 */
```

This has been an interesting exercise in understanding very clearly how to use the RobotDrive Class *drive()* method. We developed a theoretical model for differential steering of a robot. We analyzed how the *drive()* method implemented its formulas to achieve differential steering, and how all the parameters and variables are used in the code. We now understand precisely what values to use to achieve predicted robot behaviors in Autonomous mode, and we understand the limits of the method so we don't try to make it do something beyond its capabilities.

The graphs of Figures 2 and 3 were taken from plots using Wolfram Alpha at wolframalpha.com.
The drawing of Figure 1 was done with SketchUp.

Appendix A – The *drive()* method

This is a listing of the complete *drive(double outputMagnitude, double curve)* method of the RobotDrive Class. Lines that were part of the analysis in the article are underlined.

```
    /**
     * Drive the motors at "speed" and "curve".
     *
     * The speed and curve are -1.0 to +1.0 values where 0.0 represents stopped and
     * not turning. The algorithm for adding in the direction attempts to provide a constant
     * turn radius for differing speeds.
     *
     * This function will most likely be used in an autonomous routine.
     *
     * @param outputMagnitude The forward component of the output magnitude to send to the motors.
     * @param curve The rate of turn, constant for different forward speeds.
     */
    public void drive(double outputMagnitude, double curve) {
        double leftOutput, rightOutput;

        if(!kArcadeRatioCurve_Reported) {
            UsageReporting.report(tResourceType.kResourceType_RobotDrive, getNumMotors(),
tInstances.kRobotDrive_ArcadeRatioCurve);
            kArcadeRatioCurve_Reported = true;
        }
        if (curve < 0) {
            double value = Math.log(-curve);
            double ratio = (value - m_sensitivity) / (value + m_sensitivity);
            if (ratio == 0) {
                ratio = .0000000001;
            }
            leftOutput = outputMagnitude / ratio;
            rightOutput = outputMagnitude;
        } else if (curve > 0) {
            double value = Math.log(curve);
            double ratio = (value - m_sensitivity) / (value + m_sensitivity);
            if (ratio == 0) {
                ratio = .0000000001;
            }
            leftOutput = outputMagnitude;
            rightOutput = outputMagnitude / ratio;
        } else {
            leftOutput = outputMagnitude;
            rightOutput = outputMagnitude;
        }
        setLeftRightMotorOutputs(leftOutput, rightOutput);
    }
```