Universal Layered Model for Robot Drivetrain Algorithms

Carlos Gross Jones

April 19, 2021

Revision 1.0

1 Rationale

The goal of this whitepaper is to define a common model for robot drive algorithms, conceptually similar to the OSI model for computer networking. This model is specified as several layers, which accept data from the previous layer (and possibly other sources), perform some mathematical transform, and then pass the results to the next layer. There are two main advantages to enforcing a hierarchical structure on drivetrain code:

1. **Separate different kinds of transforms** to allow easy modification and reuse. It is not uncommon for FRC teams, especially rookie teams or those without much software development experience, to have a single complex equation in the code that calculates drivetrain motor throttle from joystick axis values. The problem with this is that it is easy to forget what different parts of the equation do: was that constant factor of 0.75 multiplied in to correct for joystick travel limits, or to compensate for more friction on one side of the drivetrain? This makes it difficult to swap in a new joystick with different characteristics, or to reuse the drive code on next year's robot. Furthermore, to improve reusability, the layers of this model may be used to scope logical code elements, such as functions or ROS nodes.
2. **Enforce clean interfaces** and provide obvious places to implement different kinds of control. Many teams, when implementing autonomous driving for the first time, simply use "fake" joystick commands; that is, using all the normal driving code, but getting a value from a computer vision camera in place of the x-axis on a gamepad. The problem with this is that certain parts of the algorithm, such as axis deadbanding, should *not* be applied when the input is coming from something other than the driver. When the drivetrain control algorithms are structured in clean layers, it becomes clear where to inject different types of control; autonomous commands would be fed directly to layer 3, bypassing layers 1 and 2.

No drive algorithms will be shown in this paper, except as brief examples. The purpose is not to discuss actual algorithms, but to define a conceptual model onto which *any* drivetrain control algorithm, from two-stick tank drive to hybrid-autonomous field-relative swerve control, can be mapped. Furthermore, this model does not attempt to constrain data *format*, merely data *meaning*. For example, while layer 3 introduces a concept of robot state, it does not define what the state variables are.

2 Layers

2.1 Layer 1: HID Correction

This layer is responsible for correcting any errors in the HID device itself, and computing the actual position of the axes. For example, many gamepads have one or more axes inverted (pushing the stick up decreases the reported number). On some older analog devices, such as the CH FlightStick, it may be necessary to correct for nonlinearity or scaling of the potentiometers.

Inputs	Outputs
Raw DNs from the input device	Where the driver actually positioned the axis

2.2 Layer 2: Axis Shaping

This layer applies transforms to axis positions in order to provide a better "throttle response" from the driver's perspective. Examples of operations at this layer include deadbanding and input squaring.

Inputs	Outputs
Where the driver positioned the axis	The input that the driver intended to apply to the robot

2.3 Layer 3: State Control

This layer takes all of the axes that the driver is manipulating, and determines how the robot should move. A simple example is arcade mixing. The robot control variable can be modeled as a vector $u = \{\dot{x}, \dot{\theta}\}$, in which case the joystick axes map directly to the variables: the Y stick controls $\dot{x}$ (speed), and the X stick controls $\dot{\theta}$ (turn).

A more complex example would be control of a swerve drive robot. The control variable might be a vector $u = \{\dot{x}, \dot{y}, \dot{\theta}\}$ relative to the field. Layer 3 is responsible for taking the axis inputs and converting them to a state control variable in the robot's coordinate frame.

Robot state *estimation*, such as odometry, is not a part of this layer, since it's not strictly in the drivetrain "stack"; however, it would provide additional input to this layer.

Inputs	**Outputs**
Axis inputs from driver	Robot state control variable ("chassis speeds")

2.4 Layer 4: Downmixing

Generally referred to as "kinematics" in the context of WPIlib, this layer determines what motion is required from each drivetrain motor in order to achieve a commanded robot state. The motor command will generally be in terms of speed, although it may be useful to control motor position in certain circumstances.

Inputs	**Outputs**
Robot state control variable	Motor commands

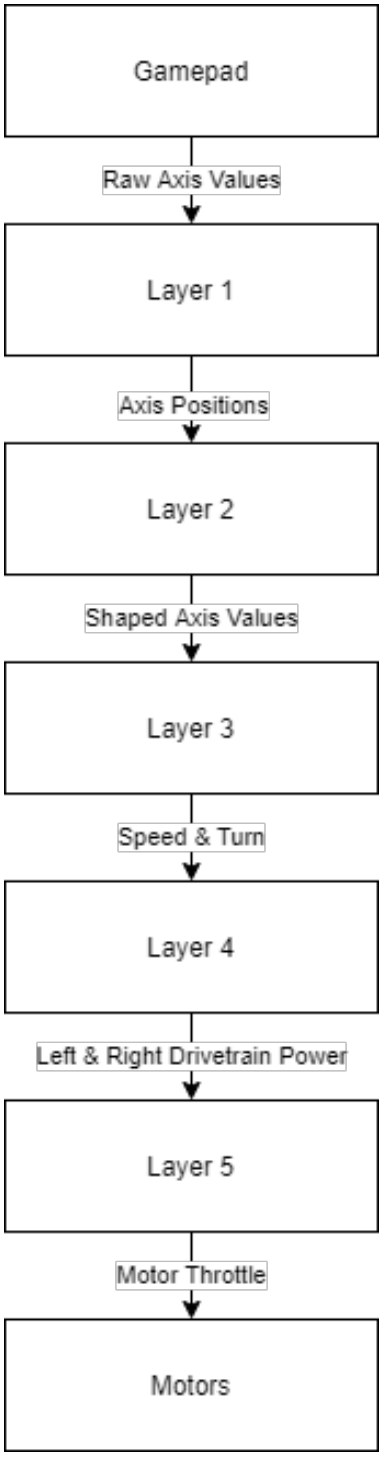
2.5 Layer 5: Motor Control

This layer converts motor commands to actual motor throttle. In the simplest case, open-loop driving, this may be an identity transform. However, more commonly, a PID controller (either in the roboRIO or onboard a motor controller) is used to adjust the throttle to maintain a commanded speed.

Inputs	**Outputs**
Motor commands	Motor throttle

3 Examples

3.1 Open-Loop Arcade Drive with Pseudocode




```
//Get DNs from joystick and
//corrects for joystick idiosyncracies
def Layer1(void):
    speed = driverJS.get(leftHandY)
    speed *= -1 //Y-axis is inverted
    turn = driverJS.get(leftHandX)
    return speed, turn

//Implement deadbanding & squaring
def Layer2(speed, turn):
    speed = deadband(speed, 0.1)
    speed = speed**2
    turn = deadband(turn, 0.1)
    turn = turn**2
    return speed, turn

//Layer 3 is an identity transform since
//joystick axes are 1:1 with state variables
def Layer3(speed, turn):
    return speed, turn

//Get motor speeds
def Layer4(speed, turn):
    leftSpeed = speed + turn
    rightSpeed = speed - turn
    //Limit to [-1, 1]
    leftSpeed = coerce(leftSpeed, -1, 1)
    rightSpeed = coerce(rightSpeed, -1, 1)
    return leftSpeed, rightSpeed

//Speeds map directly to motor throttle
//(Identity transform, since speeds
//are already on [-1, 1])
def Layer5(leftSpeed, rightSpeed):
    leftTalon.set(leftSpeed)
    rightTalon.set(rightSpeed)
```

3.2 Field-Relative Swerve Drive on ROS