2018 Software Overview

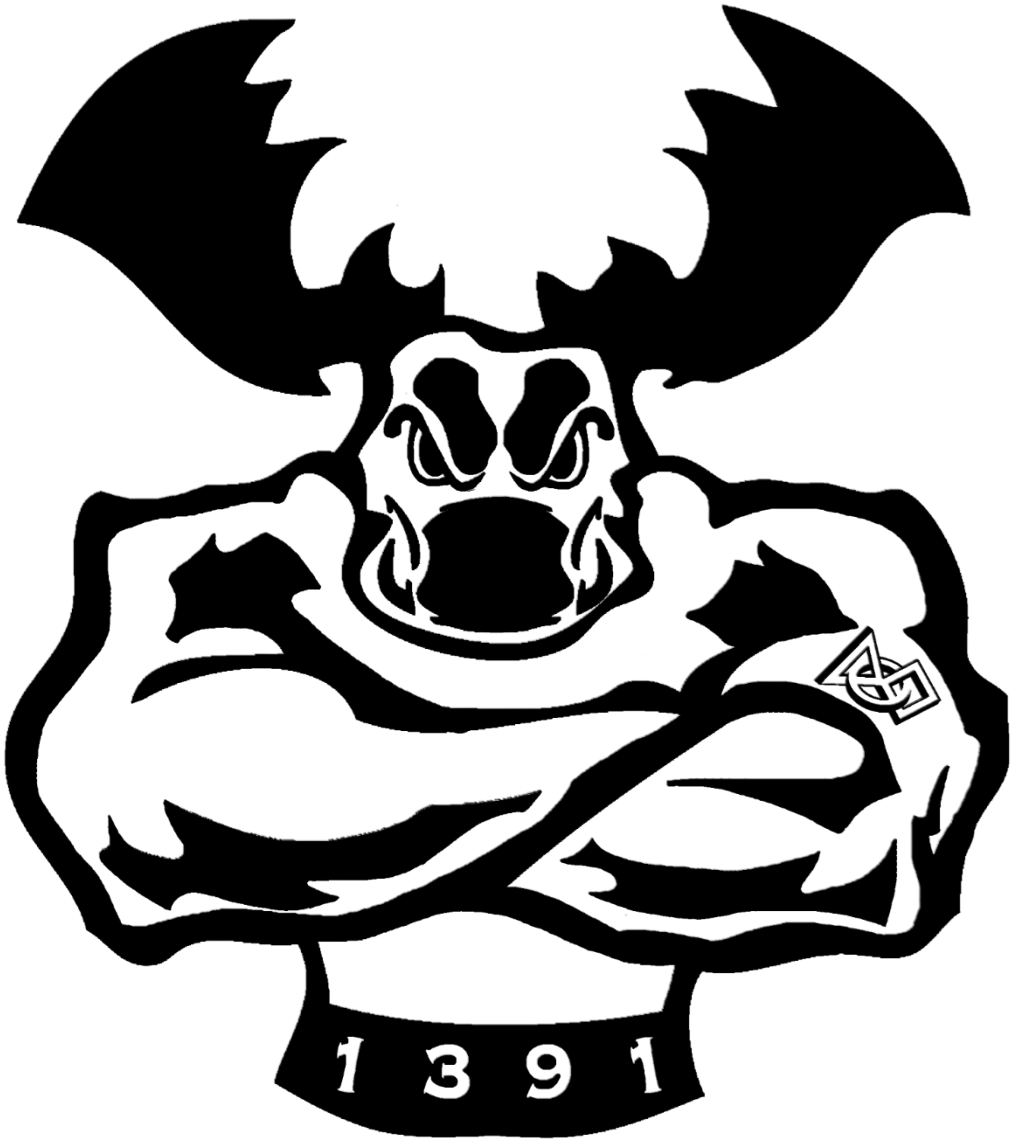


Table of contents

Introduction

This paper is a brief rundown of the various software functions of the Metal Moose's 2018 robot. Some of the reasons behind writing it include:

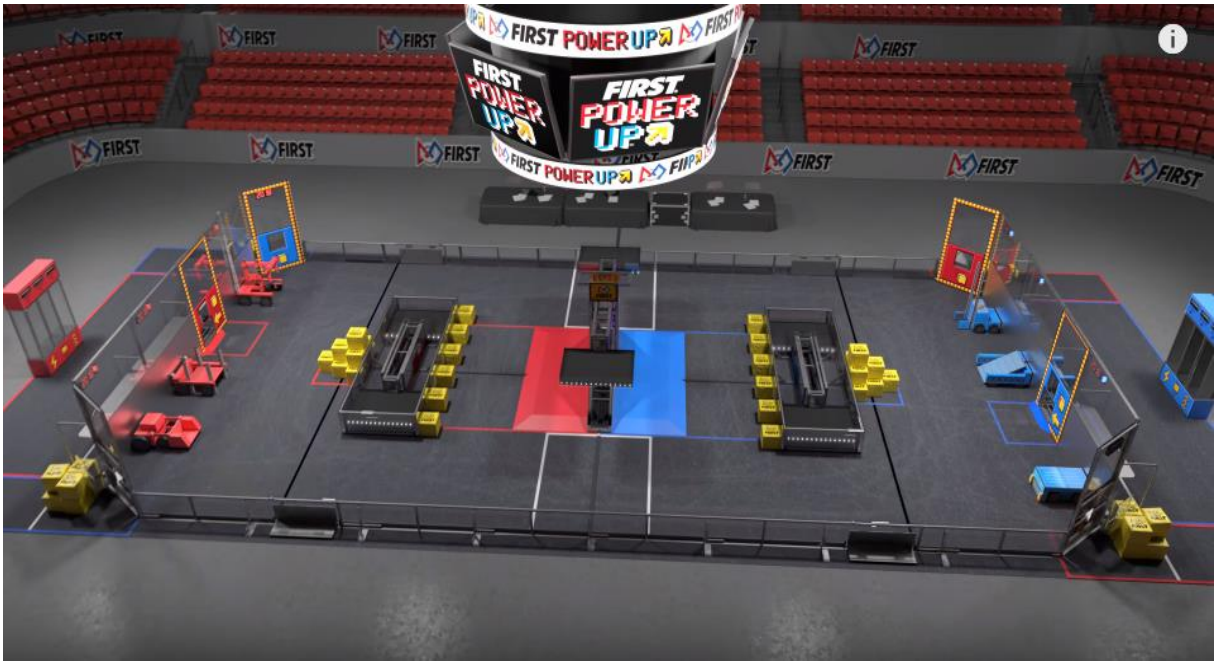
- Sharing interesting ideas with other FRC teams, looking for either inspiration or experience with various programming and real-life challenges that we faced during our 2018 FRC season.
- Serving as a quick reference/catch-up guide for our team's members.
- Having an easy way to share the interesting functionalities of our 2018 robot with the people that are interested.

It is also meant to be complemented with the Metal Moose 2018 GitHub repository, where the team stores all the code used for the robot (https://github.com/daelanr/MetalMoose2018).

FRC 2018

First Robotics Competition (FRC) is a high-school robotics competition. Every year, thousands of teams made up of high school students and their mentors build and program a robot to compete against (and with) other robots in various games that include driving, picking up objects and climbing with the robot. Getting funding and building a good robot is all about innovation and creativity – it is as close to real life engineering challenges as it gets.

The game is divided into two parts: the autonomous and the teleoperated**.** In autonomous, each of the robots operate autonomously (15 seconds). After that, the robots are operated by the drivers (2 minutes and 15 seconds). In the last 30 seconds of teleoperated, the endgame period begins: the has frequently been to elevate robot above a certain distance.



For this year's game, the robots pick up cubes to deliver on a switch and scale, with the intent of tipping them to their team's favor. They can also score points by completing tasks in the autonomous, by pushing the collected cubes into the exchange on their side of the field, and by activating power-ups using the cubes pushed into the exchange.

Software Overview

This section covers the various techniques used to easily operate our robot (both teleoperated and autonomous), some of the methods used to version control the code and to synchronize the contents of the two team's drive stations, some interesting techniques to help with operating the robot.

Polynomial Equations

One of the problems that we faced for this year was the functionality of the elevator – we would like to throttle its speed, so it both doesn't come crashing down (or flying up).

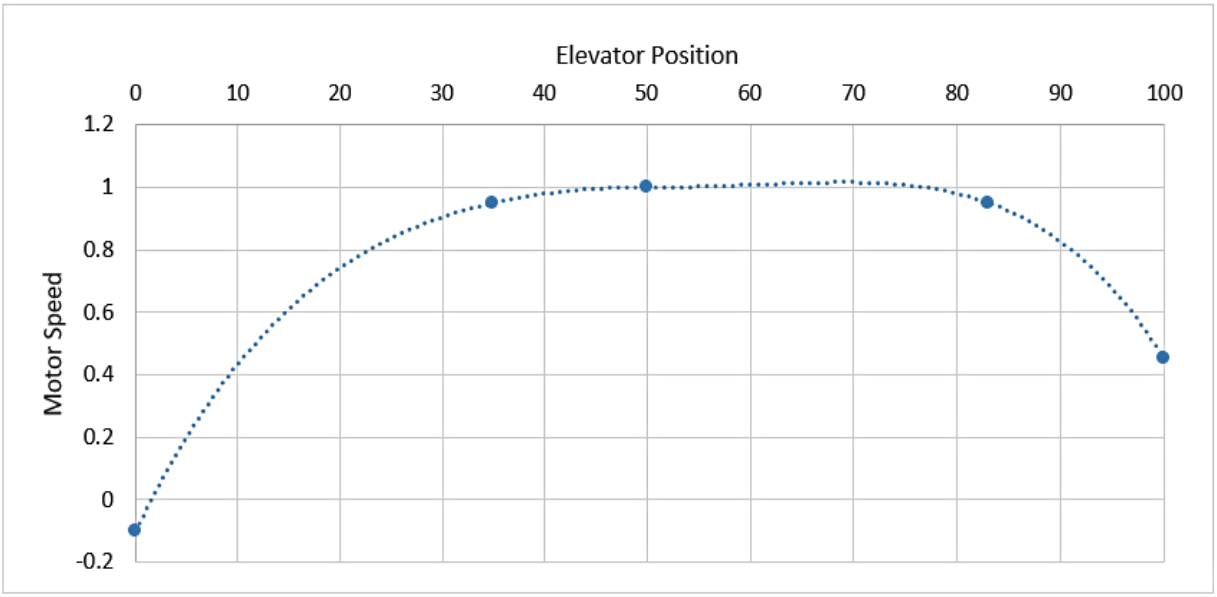
After putting an encoder on the elevator, we had a few options. Restricting the movement would be a possibility (not going above and below a certain threshold, or simply throttling the speed by a coefficient). Another way would be using PID. However, after realizing that the range of the values from the encoder is restricted (by the physical reach of the elevator), we found a more sound solution that allows us to use the full mechanical potential of the robot with elegancy: a polynomial function.

Elevator Polynomial Function

We selected 5 points to model the equation of the elevator and, using Excel's trend analysis, we generated a 4th degree polynomial function going through all the selected points. Combining them into a piecewise function (the function is different when going down – the elevator needs to slow down more):

$$f(x) = \begin{cases} -0.0000000957x^4 + 0.0000191x^3 - 0.00163x^2 + 0.0679x - 0.1, & 0 \le x < 50 \\ -0.000000123x^4 + 0.0000246x^3 - 0.00176x^2 + 0.0528x + 0.45, & 50 \le x \le 100 \end{cases}$$

Plotting the equation gives us:



The elevator subsystem now includes a "throttled" function, that automatically adjusts the speed of the elevator depending on the position.

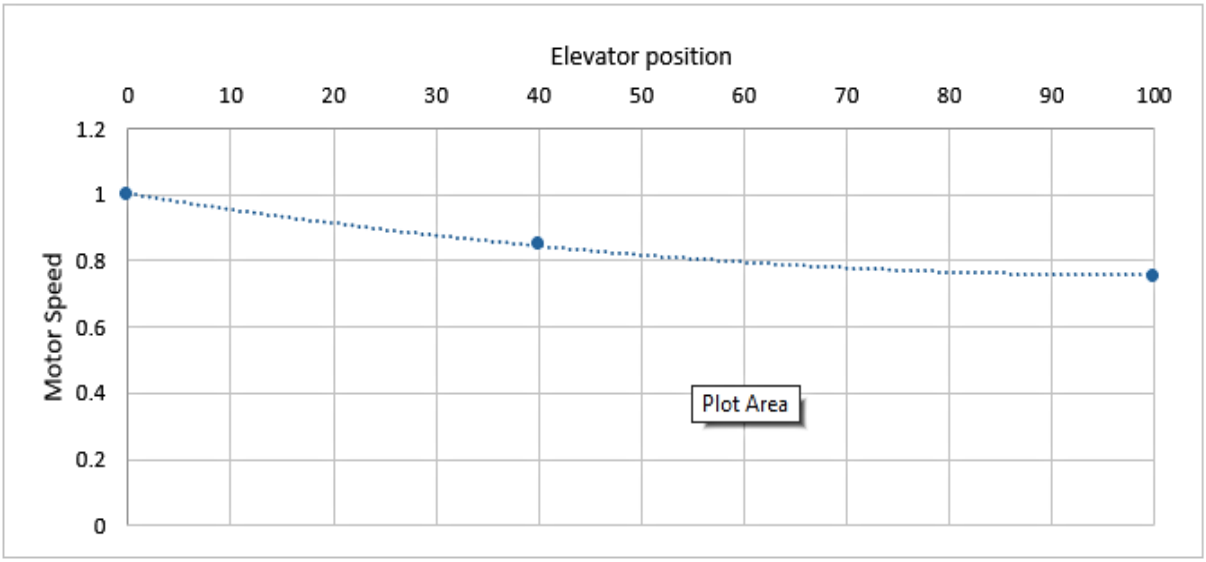
A note to be made is that going up, the elevator is throttled only when above 50%. Similarly, when going down, the elevator is only throttled when below 50% (there is no reason to be slowing the elevator down when it is in no danger of hitting either of the limits).

Drivebase Polynomial Function

Another problem that we faced during the first competition was the tipping of the robot when the elevator was up. As this problem had to be solved immediately (since we discovered the problem during the first event), we had to come up with another polynomial equation on-the-go, so over a lunch break, we added this 2nd degree polynomial function that throttles the speed of the robot depending on the position of the elevator, effectively finding the solution in a high-stress situation:

$$f(x) = 0.0000245x^2 - 0.00490x + 1$$

Plotting the equation gives us:



Simplifying Evaluation

A neat little trick that we used to evaluate respective values of the polynomial is algebraically change the form, as shown on this example:

$$5x^3 + 3x^2 - 2x + 1 = x(x(x(5) + 3) - 2) + 1$$

This both speeds up the computation and simplifies the code, since no exponentiation is required.

Autonomous Programming Language

This year's game posed a significant challenge for the autonomous systems, for the reason of the randomness in the field layout. We wanted to come up with a way to both easily define sets of movements for the robot, change them on the go, and easily mirror them over the half-line.

Our solution was creating a language with an intuitive syntax, simple adjustments of parameters and, most importantly, easy mirroring of a sequence when starting on the opposite side of the field. This way, we can easily change any part of our autonomous strategy from SmartDashboard at any time.

Syntax

A syntax is best shown on a sample sequence of commands:

```
DriveTurn(45) DriveDistance(80) DriveTurn(-45) DriveTime(2) Out(0.5)
```

This short sequence scores the cube on the right switch when starting from the middle. The robot starts turning and driving at the same time by 45°, drives 80", drive-turns again by -45° (now facing the switch), drives forward for 2 seconds (to cross the line and touch the switch), and outtakes the cube for 0.5 seconds.

Generalizing, the default syntax of a command looks like such:

$$\texttt{Command}(p_1, p_2, \ldots, p_n)$$

Command is the name of the command, and p_n is the value of the parameter of the command (usually a number. The name of the command can be either the full name (DriveDistance, Elevate…), or the capital letters (DD, E…).

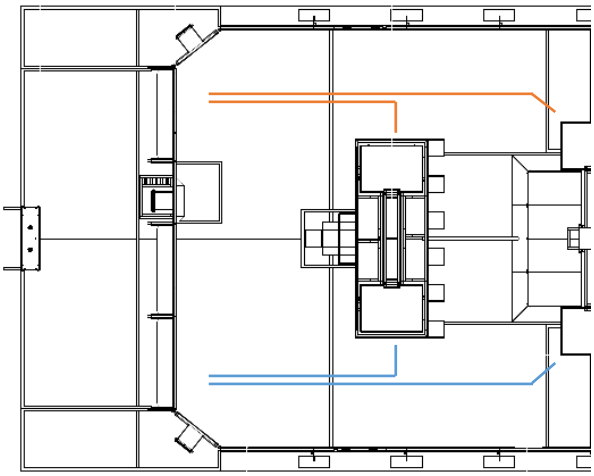
The full documentation of the parsing language can be found on the 2018 Metal Moose GitHub repository (https://github.com/Metal-Moose/MetalMoose2018).

Optional Parameters

Besides the required parameters of each of the command (DriveDistance needs the distance, TurnBy needs the angle…), a multitude of optional parameters are available. These do not completely change the functionality of a command, but simply adjust some of the possible variables of the commands:

```
TurnBy(45, Speed=0.6, Mode=Parallel)
```

In this example, the speed of the turning is adjusted to 0.6, and the command is executed in parallel with other currently running commands.



Chunks

The autonomous sequences are stored in the code as so called “chunks.” The main feature of chunks is that they can be easily mirrored, so creating the autonomous for a starting position on other side of the field is very simple.

Take starting on the right as an example. For field configurations RLR and RRR, we declared that `Chunk(1)` (scoring on the switch) and `Chunk(2)` (scoring on the scale) are going to be used.

When starting on the left, the inverses of these two chunks will be used for configurations LRL and LLL.

Inversing a chunk negates the values of all turn-based commands, so the path is perfectly symmetrical over the half-field line. The illustration bellow shows the two chunks when starting from the right (blue), and their reversed counterparts when starting from the left (red). This greatly simplifies autonomous, since there is always only one sequence that we need to keep track off, and there is no duplication of the code.

Language Synthetization

To allow the parsing of the language to be as smooth as possible, our code deploys multiple regular expressions to synthesize the input, making it less prone to typos and mistakes of similar sort. The language can pass parses incorrect inputs into their correct forms (if the syntactic structure is correct).

The parser converts the input from the first line to the input of the second line:

```
"    TrnB (  57° )DrveDisttance   ( 80in, Sped= 0.7 )"

"TurnBy(57) DriveDistance(80, S=0.7)"
```

Turing Completeness

Before the removal of the trigonometric position system, the language that we developed was a step away from being Turing complete (that it was computationally universal), meaning that you could program anything in it – an operating system, a word processing program, an equation solver…

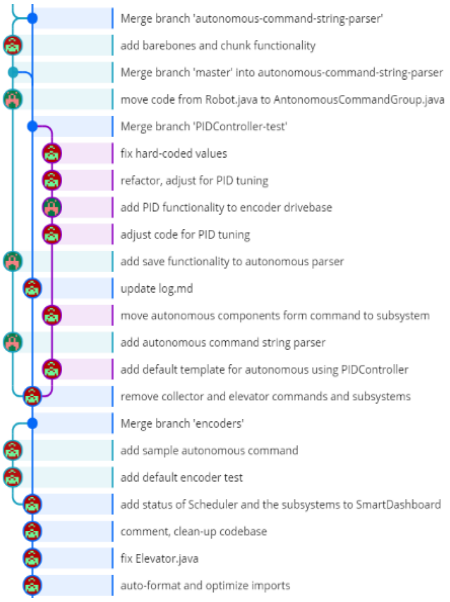
The way it would be done is taking the coordinates of the robot as the tape of the Turing machine, interpreting the commands as the states of the machine and adding a "jump" command that would jump between the states.

This is just a side note, because the programming team thought it was cool.

Version Control, Data Manipulation

Git Version Controlling

Version controlling the code was an absolute must, so we decided to use the Git version control system to keep track of the changes that we made to the code. This allows for an easy collaboration between the members of our team, and other teams. We share not just our codebase, but also the layout of our SmartDashboard. The diagram on the right shows the interactions of our programming team and the easy collaboration that this version control system allows.



Offline Event Data Synchronization

Without WiFi, it is sometimes quite impractical to copy the contents of one of our drivestations to the other one using the file browser. That is why we automated the process using a Python script – we automatically copy the code of one drivestation onto a flashdrive to a folder with the current date as the name, to simplify and speed up the work amid the competition, where every second counts.

Trigonometric Position System

One of the functionalities of our autonomous was a coordinate-based position system. A goal that we had in mind was that the robot remembers its exact position during autonomous, so whether it overshoots or overturns, it could correctly calculate its next movement depending on the circumstances.

The Goto Command

The main feature of this system was the $goto$ command.

The entire field is interpreted as a large cartesian coordinate plane. The robot gets instructions $goto(x, y)$ and first generates a turn command to turn to the pair of coordinates and a drive command to drive the distance to the pair of coordinates.

The angle is calculated using $\arctan\left(\frac{y}{x}\right)$. One needs to be quite carful when using arctan, since it only works in quadrants I and II. We solved this by following this logic: if both x and y are negative, the resulting fraction $\frac{y}{x}$ will be positive, so the result lays in quadrant I, but the coordinates lay in quadrant III. That is why we subtract 180° from the result to get to quadrant III. Similarly, if only y is negative, the resulting quadrant is quadrant II, but the coordinates lay in quadrant IV, so we add 180°.

Distance to the coordinates are calculated using the Pythagorean theorem: $\sqrt{(x-x_0)^2+(y-y_0)^2}$, where x_0, y_0 are the initial coordinates and x, y are the destination coordinates.

The position of the robot on the plane is calculated using $\mathrm{x} = \sin\theta * d$ and $y = \cos\theta * d$, where θ is the angle that the robot was in before the movement started and d is the distance that it traveled. Both values are read from the sensors on the robot (an encoder and a gyro on the drivebase).

This system allows easy mirroring of the commands depending on different field layouts, as going for the coordinate on the other side is simply negating the x value and provides a really easy way to program the autonomous sequences.

Problems and Solutions

One of the problems was always calculating the correct angle that the robot had to turn, since the robot could be in any angular position (even over ± 360) and turning more than $\pm 180°$ would be suboptimal.

The solution was using the modulo operator: $\theta \equiv \theta_0 \ (mod\ 360)$, bringing the range of possible values to $\pm 360°$. If the values were over $\pm 180°$, the optimal solution for a negative angle was $x = 180 - (\theta\ mod\ 180)$, and $x = -180 + (\theta\ mod\ 180)$ for a positive angle (we are essentially calculating an angle adding up to $\pm 180°$).

We ultimately decided against using this system (although it was fully implemented and tested, as seen in the older versions of the code), for the sole reason of speed. Although the system worked very well, the amount of precision required from the PIDs of our sensors was too slow for actual usage.

With that being said: with a well-tuned PID, a system like this would be invaluable for games with randomly generated field layouts.

Robot Control

Reverse Button

Our robot must drive backwards onto the platform to successfully climb, so the primary button of the joystick reverses the driving so that forward is now backward and vice versa. This makes driving backwards incredibly simple and intuitive, making the entire process of hanging the robot on the rung as smooth as possible.

Sensors

The robot is equipped with a navX-MPX navigation sensor that allows for smooth turning during the autonomous period. It is also equipped with and encoder on one side of the drivebase to precisely measure distance driven (for autonomous), and one on the elevator motor, to throttle its speed.

Double PID Control

The autonomous heavily relies on driving a correct distance in a straight path. This is achieved by simultaneously using two PID systems: one to control the encoder (measuring the distance driven) and one for the gyro (to always drive in a straight line).

A neat trick that we used is that both PID outputs can be fed directly to the arcadeDrive function: the encoder PID output as the first parameter, to alter the speed of driving, and the gyro PID output as the second parameter, to alter the turning of the robot.