The Drivetrain Interface

FTC Team 4410 (Lightning)

October 2017

1 Introduction

With Mecanum wheels, the robot is able to move in 8 directions from a given point without turning. To fully take advantage of this feature, the drivetrain subsystem should be optimized and designed to work with them. The subsystem should be designed to work with both TeleOp and Autonomous logic as effortlessly as possible *(See Autonomous Coordinate System for how autonomous code may interact with the drivetrain)*; It should also provide plenty of flexibility for testing and experimentation purposes.

1.1 Use of Vectors

This interface presents movements as vectors on a 2D plane, in which the positive y direction represents forward and the positive x direction represents right. Note that the plane **is not** projected relative to the playing field, but rather to the robot, i.e. the direction in which the robot is facing equals the positive y direction, and the robot's location is $(0, 0)$, or the starting point of any vector in this context.

2 Description

2.1 Autonomous Methods

To support the 2D processing involved in ACS, the drivetrain interface accepts any arbitrary vector from the origin as a parameter. It then applies desired movements to drivetrain motors to accomplish this task without turning the robot. This feature works directly with multipoint paths that are generated by ACS for effortless navigation. An optional power multiplier parameter is provided for customization of speed.

2.2 TeleOp Methods

The methods designed for TeleOp are optimized for being controlled in a synchronous loop

3 Implementation

Algorithm 1 Find path from a to b

```
procedure A*($a, b$)
    $f \leftarrow \{[a, 0]\}$                                  ▷ Priority queue: Frontier
    $p \leftarrow \{a : \text{nil}\}$                          ▷ Key-value map: Path
    $c \leftarrow \{a : 0\}$                                   ▷ Key-value map: Cost to node
    while $\#f > 0$ do
        $s \leftarrow$ pop item from $f$                       ▷ Coordinate: Current node
        if $s = b$ then
            return BACKTRACK($s, a, p$)                        ▷ From $s$ to $a$ in $p$
        end if
        for $n \leftarrow$ NEIGHBORS($s$) do                   ▷ Coordinate: Next node
            $o \leftarrow c[s] + \text{COST}(s, n)$            ▷ Real: Cost to next node
            if $(n \notin c) \vee (o < c[n])$ then             ▷ New cheapest cost to $n$
                $c[n] \leftarrow o$
                put $n$ into $f$ with priority $o + \text{HEURISTIC}(b, n)$
                $p[n] \leftarrow s$
            end if
        end for
    end while
end procedure
```

If the algorithm above were to be used with the result directly fed to the drivetrain manager, the robot would stop itself everytime a new node is reached. To alleviate this ineffiency, the following algorithm converts the steps into a collection of points that describe when the driving direction changes.

Consequently, the results from the above algorithm are converted to calls to the Drivetrain Manager in the following algorithm.

Algorithm 2 Convert nodes P to points p

```
Require: $P$ to be an array of coordinates
 1: procedure ConvertPath($P$)
 2:     $l \leftarrow (0, 0)$                                   ▷ Last vector
 3:     $o \leftarrow \emptyset$                                ▷ Output path (ORDERED)
 4:     for $i$ in $[1, \#P)$ do                                ▷ Index
 5:         $v \leftarrow \overrightarrow{(P[i-1])(P[i])}$      ▷ Current vector
 6:         if $v \neq l$ then
 7:             $o \leftarrow o \cup \{P[i-1]\}$
 8:             $l \leftarrow v$
 9:         end if
10:     end for
11:     return $o \cup \{P[\#P - 1]\}$
12: end procedure
```

Algorithm 3 Carry out navigation of points p

```
Require: $p$ to be an array of coordinates
 1: procedure Navigate($p$)
 2:     for $i$ in $[1, \#p)$ do                                ▷ Index
 3:         Drivetrain.move($\overrightarrow{(p[i-1])(p[i])}$)
 4:     end for
 5: end procedure
```