TKO Autonomous Documentation 2019-2020

Team 1351

2019

Contents

1 Overview

This is team 1351's autonomous programming documentation for a deeper understanding of what autonomous features we use on our team, what our custom libraries do, and how to implement them into your code. This covers a wide range of autonomous topics from how to plan your autonomous mode to advanced autonomous concepts such as path following and computer vision. This does not go over basic robot code, but you can refer back to our (insert link here)FRC Programming documentation for all the resources related to the essential robot code.

2 Key Terms

2.1 Control Loop

- A loop used to calculate an output (generally motor voltage)
- A fundamental concept in autonomous robot movement
- Can be open (no feedback) or closed (feedback)
- Learn more from (insert link to closed loop control on other documentation)

2.2 Error

- The difference between the expected output (i.e. position setpoint) and the current output (i.e. current position)
- For example, if the setpoint is 5 inches and the robot has moved 3 inches already, the error is 2 inches (5 in - 3 in)

2.3 PID Controller

- Proportional, Integral, Derivative control loop
- Learn more from (insert link to PID controller on other documentation)

2.4 Motion Profile

- A graph of the movement of a mechanism over time
- Used to optimize the movement of a mechanism

2.5 Trapezoidal Motion Profile

- A motion profile where the velocity over time is in the shape of a trapezoid, with an acceleration, cruise, and deceleration period

2.6 Triangular Motion Profile

- A motion profile where the velocity over time is in the shape of a triangle, with only an acceleration and deceleration period

2.7 Relative Position

- A position relative to the starting position
- For example, a relative position of 4 inches along a slider is 4 inches further than the current position of the mechanism

2.8 World (or Absolute) Position

- A position relative to the real-world position of the mechanism
- For example, a world position of 4 inches along a slider is a distance 4 inches along the slider regardless of the current position of the mechanism

2.9 Path Following

- The method of autonomous drivetrain movement that follows a 2D path (generally a curved path)

2.10 Pure Pursuit Controller

- A method of robot path following that calculates the angular velocity to reach points on a path
- More information from https://www.mathworks.com/help/robotics/ug/pure-pursuit-controller.html

2.11 Look Ahead

- A principle used in the pure pursuit controller that represents the act of the robot targeting a point in front of it
- The point that the robot is targeting is referred to as the target point or the look ahead point
- The distance between the target point and the robot is referred to as the look ahead distance

2.12 Field-Centric Location

- Similar to the concept of absolute position, field-centric location refers to the location of the robot relative to the field
- Field-centric and robot-centric location is explained more in the Autonomous Drivetrain Movement section

2.13 Robot-Centric Location

- Similar to the concept of relative position, robot-centric location refers to the location of the robot relative to the current location of the robot
- The current position is always (0,0,heading), the new location is (x,y,heading) away from the robot (Heading is always absolute, however, since it is referring to the angle of the robot given by the gyro, which calibrated to 0 when the robot is first turned on)
- Field-centric and robot-centric location is explained more in the Autonomous Drivetrain Movement section

2.14 Robot Heading

- A fancy way of saying the robot's current angle

2.15 Autonomous Sequence

- The sequence of robot actions that makeup the autonomous mode or an automated sequence of events that the robot performs
- For example, if the robot drives up 48 inches, lifts the arm 20 inches, and outtakes an object, the sequence is comprised of a command to move forward 48 inches, a command to lift the arm 20 inches, and a command to run the intake motors outward

2.16 Vision / Computer Vision

- A method of autonomous robot movement involving the detected features of vision targets

2.17 Vision Target

- An easily recognizable target that can be detected by a computer vision algorithm
- In FRC, vision targets are generally pieces of retro-reflective tape next to important field elements that can be easily detected by shining a bright light at it and filtering the color reflected back

3 What is Autonomous?

On a general level, teams create autonomous functions for their robot in response to FRC's 15 seconde autonomous control portion of the game. This 15 second portion involves no user input (in exception to 2019's game, which allowed user input but hindered viewing capabilities of the field), and it is entirely up to the

robot's pre-programmed autonomous motion to perform game actions. On a deeper scale, however, autonomous functions can be integrated into the robot's tele-operated functions to give your robot a competitive edge during match play.

4 Control Loops

The general overview of what control loops are and how to program them can be found in our general FRC programming documentation here: (insert link to control loops section of general documentation), but this will cover how control loops are specifically related to autonomous.

Control loops are the underlying principle to getting reliable movement out of your robot, and most of the more advanced autonomous concepts come down to a control loop to control the actual movement of the mechanism. For example, the final layer of path following results in an output object with just a left and right drivetrain velocity, which are sent to a control loop that calculates the voltage to apply to the motors in order achieve that velocity. Whenever you see control loops mentioned in this document, just know that they are simply referring to a function that takes in a setpoint (generally position or velocity) and calculates a motor voltage (or percent output between -1 and 1) based on sensor feedback to be applied to the motor. In other words, they are the layer that allows mechanisms to achieve units known to us such as speed or position by calculating the voltage to apply to the motors.

5 Motion Profiling

5.1 Overview

Motion profiling is a powerful tool that regulates and controls the motion of your mechanism in a control loop. Motion profiles calculate the motion of your mechanism to get from point A to point B and provides an incremental setpoint for your control loop that is in alignment with your specified maximum acceleration, deceleration, and peak velocity for your mechanism. Not only does this provide smoother movement for your mechanism, it also reduces the stress on your control loops, thus reducing the amount of tuning required to get accurate positioning.

5.2 Anatomy of a Motion Profile

Motion profiles can come in different shapes. A motion profile shape is defined by the shape of it's velocity over time graph. Generally, we use trapezoidal motion profiles due to their simplicity, which means the graph of velocity over time is in the shape of a trapezoid. **Figure 1** displays a trapezoidal motion profile. Look for the position over time graph (blue), the velocity over time graph (red), and the acceleration over time graph (green). A motion profile can

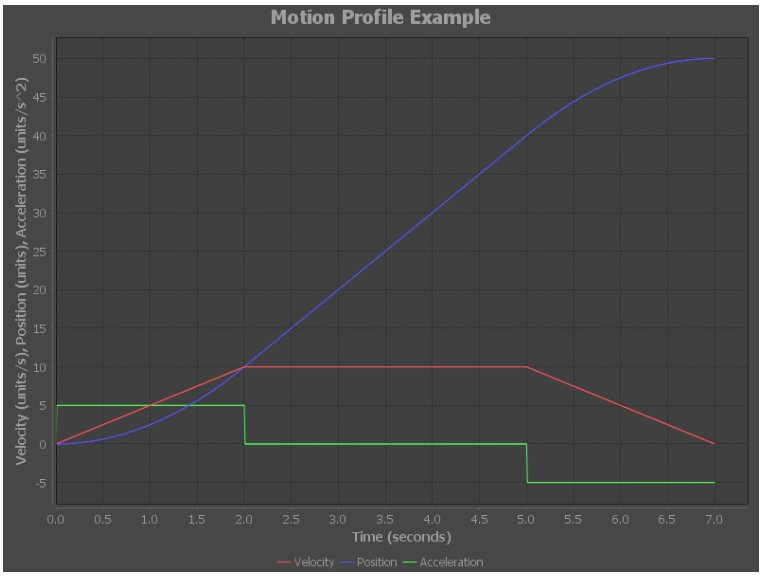

Figure 1: An example motion profile

be divided into 3 parts, an acceleration, cruise, and deceleration portion. The acceleration portion is where the mechanism is accelerating from its starting velocity (may be nonzero in some cases) to it's cruise velocity (the maximum velocity of the mechanism). The cruise portion is where the mechanism has hit its maximum velocity and is cruising at that velocity until it is time to decelerate. The deceleration portion is when the mechanism must decelerate to its ending velocity (may also be nonzero). The position is then calculated based on the velocity graph, and as you can see it results in a very smooth, curved movement.

5.3 Our Library

For motion profiling, our team uses our custom motion profile generator library that can be found on GitHub: `https://github.com/MittyRobotics/motion-profile-generator`. There are basic instructions on the usage of the motion profile library as well as how to add it to your code on the GitHub page,

but you can also continue reading more detailed information on the usage of the library in this document.

5.4 Using The Motion Profile Library

5.4.1 Inputs

The motion profile library takes in the following inputs:

- maximum acceleration
- maximum deceleration
- maximum velocity
- starting velocity
- ending velocity
- lower position bound
- upper position bound
- current position
- desired position (setpoint)

This may seem like a lot of inputs, but they are all quite straight forward to determine. If you would like explinations of any of these inputs, here is a small index explaining in depth what each input means:

- maximum acceleration
  - The maximum acceleration speed of your mechanism (units/s/s)
- maximum deceleration
  - The maximum deceleration speed of your mechanism (units/s/s)
- maximum velocity
  - The maximum velocity speed of your mechanism (units/s)
- starting velocity
  - The starting velocity speed of your mechanism (units/s). This is used if your mechanism is currently moving and you want it to smoothly transition into another movement without violating the maximum acceleration and deceleration bounds.
- ending velocity

  - The ending velocity speed of your mechanism (units/s). This is used if you want your mechanism to finish the motion profile moving at a certain speed (example: to make a flywheel slowly ramp up speed and hold it's maximum speed at the end).

- lower position bound
  - The lower or minimum position allowed for the mechanism. This is used if your mechanism has certain bounds it needs to stay within, such as a lift. For example, if a lift mechanism can go down to its lowest position 0 and up to its highest position 200, 0 would be the lower position bound. This is an input to ensure that the motion profile will never ask for an unachievable position from the mechanism. Leave this at 0 if the mechanism does not have any position bounds.

- upper position bound
  - The upper or maximum position allowed for the mechanism. This is used if your mechanism has certain bounds it needs to stay within, such as a lift. For example, if a lift mechanism can go down to its lowest position 0 and up to its highest position 200, 200 would be the upper position bound. This is an input to ensure that the motion profile will never ask for an unachievable position from the mechanism. Leave this at 0 if the mechanism does not have any position bounds.

- current position
  - The current position of the mechanism. This is where the mechanism is right now, and is taken into account when calculating the ending position. Using the current position input makes your motion profile work in absolute space rather than relative space, meaning the mechanism will move to the absolute position of the setpoint rather than the setpoint of units away from the current position. In order to work in relative space instead, simply leave the current position value at 0. For example, this would be your point A if you want to get from point A to point B.

- desired position (setpoint)
  - Finally, this is the setpoint of your motion profile. This where you want the mechanism to move to, or the point B part of point A to point B.

All of these inputs are in no specific unit, so whatever units you like working in the best is what you can input as long as they are consistent. The output will then be in the same units as the input. This means you could work in inches, encoder ticks, meters, etc.

5.4.2 Initialization

To initialize the motion profile, you first start by defining all of your inputs as well as passing them into their respective objects. The motion profile itself is organized into a TrapezoidalMotionProfile object, which takes in a setpoint, a VelocityConstraints object (containing all velocity related inputs), and a MechanismBounds object (containing all mechanism position related inputs).

```
double acceleration = 5;                    //units/sec^2
double deceleration = 5;                    //units/sec^2
double maxVelocity = 10;                    //units/sec
double startVelocity = 0;                   //units/sec
double endVelocity = 0;                     //units/sec
double lowerPositionBound = 0;              //units
double upperPositionBound = 124.5;          //units
double currentPosition = 22.4;              //units
double setpoint = 80;                       //units

VelocityConstraints velocityConstraints = new
    VelocityConstraints(
acceleration,
deceleration,
maxVelocity,
startVelocity,
endVelocity
); //Creates the VelocityConstraints object with respective
     inputs

MechanismBounds mechanismBounds = new MechanismBounds(
currentPosition,
lowerPositionBound,
upperPositionBound
);  //Creates the MechanismBounds object with respective
    inputs

TrapezoidalMotionProfile motionProfile = new
    TrapezoidalMotionProfile(
setpoint,
velocityConstraints,
mechanismBounds
); //Creates the main TrapezoidalMotionProfile object with
    respective inputs
```

5.4.3 Accessing Data From Motion Profiles

Now that the motion profile is generated, you can get the acceleration, velocity, and position value for an input timestamp using:

```
double t = 2.43; //seconds

MotionFrame frame = motionProfile.getFrameAtTime(t); //
    Motion frame object that contains the acceleration,
    velocity, and position at timestamp t

double position = frame.getPosition();         //units
double velocity = frame.getVelocity();         //units/sec
double acceleration = frame.getAcceleration();//units/sec^2
```

Normally, to follow the motion profile you want to loop through the motion profile periodically and get the current position value for the time since you began following the motion profile.

FRC Commands and Subsystems tip: If you are using the FRC commands and subsystems format, commands have a function to get the time since the command began, which comes in handy while following a motion profile:

```
double t = timeSinceInitialized(); //Time in seconds since
    the command has been initialized, this is usually the
    time you would want to get a motion frame from
```

5.4.4 Visualizing Motion Profiles

The library has a built in class to allow you to graph a motion profile. This uses JFreeChart to graph the values of the motion profile over time. Simply create a new GraphMotionProfile object and input the desired motion profile to graph as a parameter:

```
new GraphMotionProfile(motionProfile); //Creates a new
    JFrame window containing a JFreeChart graph of the
    motion profile
```

Figure 2 displays the output graph of the example code.

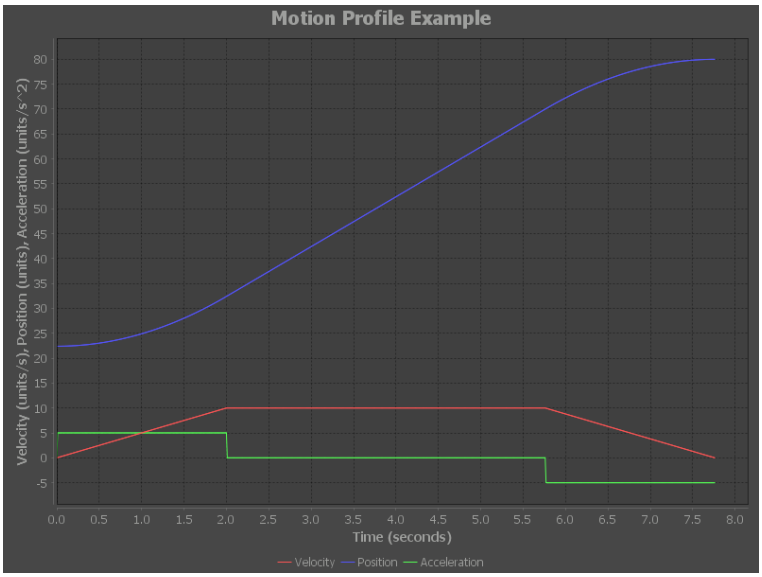

Figure 2: The generated motion profile given the example code

5.5 How can this be used?

Since the TrapezoidalMotionProfile can be used to get a position and/or velocity value at a specified time, all you need to do is create a loop that iterates at a specified frequency, keep track of the total time since the loop began, and get the MotionFrame object created by the motion profile's getFrameAtTime() function to get the value. For each iteration of the loop, you can then pass in the position and/or velocity value for that time into a control loop as the setpoint.

6 Autonomous Drivetrain Movement

6.1 Overview

In all autonomous modes, the robot generally must move to some location. There are many ways to achieve robot movement depending on the needs of your autonomous sequence.

DISCLAIMER: Our team currently uses a differential drive drivetrain, and will therefore base all of our autonomous drivetrain movement off of the differential drivetrain model. If you are using any other type of drivetrain, many of the same principles can be adapted to your drivetrain type.

6.2 Field-centric and Robot-centric location

Before we get into autonomous drivetrain movement, we must first establish our standards for what coordinate system we are working in.

6.2.1 Field-centric location

Field-centric location is the location of the robot relative to a specified point on the field. However you wish to define the point that all robot locations are based around is fine, but our team generally uses the bottom-left corner of the field as (0,0) and facing towards the opposite driver station is a heading of 0. We like to view the field with x position as the width of the field and y position as the length of the field because it is easier to visualize the paths from the perspective of the driver station. **Figure 3** displays a diagram of our team's standardized field-centric coordinate system on a half field with some 2019 field elements drawn for a since of direction.

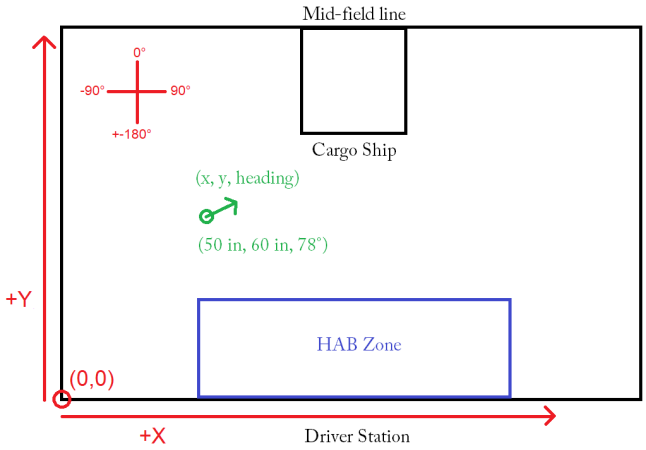

Figure 3: Field-centric location diagram

Notice the compass at the top left shows the heading coordinate system, the red circled point at (0,0) is the point we chose to orient the robot around, and the green vector represents a sample robot orientation at (50, 60, 78 degrees). We choose the units to be in inches and we represent a robot location using a coordinate object containing an x, y, and heading value. Normally Gyros read their rotation from 0-360 degrees, but we map that value between -180 to 180 degrees so we can represent a right rotation as + and a left rotation as - for simplicity.

If you are ever confused about where a field-centric coordinate is on the field, refer to the diagram to help plot the point. This standardized coordinate system was chosen by our team to be easy to visualize robot coordinates in your head without having to plot it on graph.

6.2.2 Robot-centric location

Robot-centric location is more simple. We maintain the same coordinate standard, but the origin point (0,0) is now always at the robot's current location. For example, the coordinate (24, 60, 0) is 24 inches to the right of the robot and 60 inches in front of it, while still facing 0 degrees.

6.2.3 Working with angles

Remember, heading is always absolute, and 0 is the heading of the robot when you first turn it on or initiate your Gyro, since that is when the gyro is recalibrated and set to 0. With that in mind, always position your robot in the correct angle that you want to be 0 degrees when you turn it on or initialize the robot code, or you can manually set the angle to 0 after it is initialized when you move the robot to face the correct angle. Also, as mentioned above, the heading ranges from -180 to 180 degrees, so we have to adjust our final angle value to fall in the same system (normally gyros read from 0 to 360, also make sure that a right turn is positive angle and a left turn is negative angle, or else you may need to reverse your value).

6.3 Figuring Out the Robot's Location (Odometry)

In order to find out where the robot is, whether it is field-centric or robot-centric, you need two elements on your drivetrain: encoders on the left and right wheels and a gyro to measure the robot's yaw angle.

6.3.1 Encoder measurements

Encoders will always output their value in encoder ticks. Read more about how to use encoders and interpret encoder tick outputs here: (insert link to encoders section of FRC notes documentation). In summary, you need to find the amount of encoder ticks that are read per inch of robot movement. Then

you can measure the distance the robot moves forward or backward. Later, we will explain how you can use this distance combined with the gyro angle to get the 2D position of the robot instead of a linear position.

6.3.2 Gyro measurements

Gyros will generally output their value in degrees. Read more about how to use gyros here: (insert link to gyros section of FRC notes documentation). Gyros alone will allow you to measure the angle of your robot for performing in-place turns to a specific angle, but later we will explain how gyros can be used in combination with encoder position to get the 2D position of the robot.

6.3.3 Odometry

Odometry is the use of sensors to find the robot's position. This is how we will get the full 2D location of the robot, either field-centric or robot-centric. In most cases in FRC, odometry is achieved with just the encoder and gyro measurements. The way we achieve 2D location measurements is by using trigonometry with the gyro angle and the encoder position at a given time, and iterating through this calculation many times a second.

The process is as follows:

- Determine which axis is forward
  - In our case, we defined the forward axis as the positive Y axis
- Determine which axis is right
  - In our case, we defined the right axis as the positive X axis
- Find the change in encoder position from the last calculation
  - Keep track of the left and right encoder measurements from the previous time you calculated your odometry measurement and subtract it from the current measurements
  - The final change in encoder position should be averaged from the change in the left and right's encoder position and be in whatever units you want to measure the location in, in our case inches
- Keep a global variable for the 2D x and y value of the robot as well as the heading (angle) of the robot.
- Get the forward position by taking the cosine of the gyro angle and multiplying it by the average change in encoder position and then add it to the global x or y value (depending on which axis you chose as the forward axis)
  - If you choose the forward axis to be negative x or y, subtract it instead of adding it

- Get the right position by taking the sine of the gyro angle and multiplying it by the average change in encoder position and then add it to the global x or y value (depending on which axis you chose as the forward axis)
  - If you choose the right axis to be negative x or y, subtract it instead of adding it
- Lastly, map the gyro angle to your standardized heading system, in our case -180 to 180, and set your global heading value to that

This results in the x, y, and heading value of the robot in the standardized coordinate system we chose. Here is some example code:

```
double x, y, heading; //robot location variables
double previousLeftEncoder, previousRightEncoder; //
    previous encoder positions

//Updates the odometry measurements. Call updateOdometry()
    at a cetain frequency, we just use the period time of
    our robot (somewhere around 0.02 seconds per loop)
public void updateOdometry(){
    double gyro = Gyro.getAngle(); //get gyro angle, make
        sure your gyro reads the angle between 0 and 360 or
        else you may need to adjust the value to read that.
        Also make sure a right turn is a positive angle on
        the gyro, otherwise you need to do -Gyro.getAngle();

    double changeLeftPos = getLeftEncoderInches() -
        previousLeftEncoder; //change in left encoder
        position (inches)
    double changeRightPos = getRightEncoderInches() -
        previousRightEncoder; //change in right encoder
        position (inches)

    double avgEncoderChange = (changeLeftPos +
        changeRightPos)/2; //average change in encoder
        position

    double forward = avgEncoderChange * Math.cos(gyro); //
        forward change in position
    double right = avgEncoderChange * Math.sin(gyro); //
        right change in position

    y += forward; //add forward to Y (the forward axis)
    x += right; //add right to X (the right axis)

    robotHeading = gyro-180; //make robotHeading between
        -180 and 180

    previousLeftEncoder = getLeftEncoderInches(); //set the
         previous left encoder position to the current
        encoder position
    previousRightEncoder = getRightEncoderInches(); //set
        the previous right encoder position to the current
        encoder position
}
```

6.4 Methods of Autonomous Drivetrain Movement

There are many ways to achieve autonomous drivetrain movement. The simplest method of movement is to construct a path for the robot using linear forward/backward movement and in-place turns. We like to call this 1D movement, since the robot can only move in one dimension at a time: either forward, backward, turning left, or turning right. Another, more complex method of drivetrain movement is called Path Following, or 2D movement. This method involves the drivetrain to be both driving forward/backward and turning at the same time, allowing it to follow generated 2D curved paths. 2D path following will be described later on in the next section. **Figure 4** and **Figure 5** show a 1D vs 2D path.

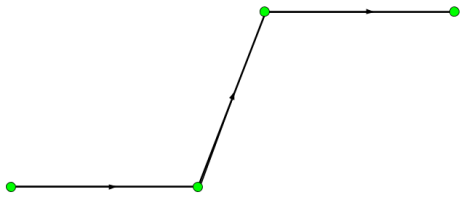

Figure 4: 1D path sequence

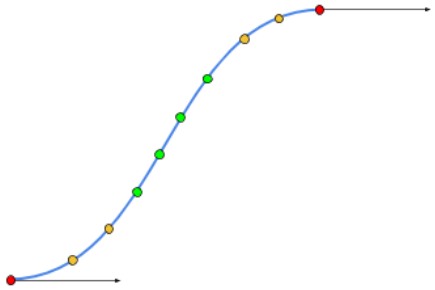

Figure 5: 2D path

6.5 1D movement

As mentioned above, 1D movement is the action of moving either forward, backward, turning left, or turning right. This results in a sequence of straight lines and in-place turns to get to a destination.

We will not go too much in dept with 1D movement here since it is quite basic compared to path following, but 1D movement basically involves a control loop on your chassis motors to rotate the drivetrain wheels in a direction until you reach the destination distance or angle. This can be done with a PID loop, and more information on how to create this PID loop can be found here: (insert link to PID section of general documentation).

6.6 Which drive method to use?

Out of the two methods, 2D driving is definitely faster to get to the destination due to the fact that you are essentially driving and turning at the same time rather than driving and then stopping to turn. There are some situations where you would still want to use 1D driving, however, such as if a greater level of precision is a requirement in your autonomous sequence. 2D driving, especially with the pure pursuit controller algorithm that we use on our team (more on this later), tends to be less accurate at reaching the destination point. In general, however, 2D movement will usually provide you with accurate enough results required in the autonomous sequence, especially if you can pair it with a method of accuracy correction such as computer vision alignment (more on this later as well). Overall, it is up to you to decide what method of driving you would like to use.

7 Path Following

7.1 Overview

Path following is the method of autonomous drivetrain movement that follows a 2D path, generally one that is curved. As previously mentioned, this is much faster than 1D movement, but is a little less precise at exactly ending at the destination point.

7.2 Methods of Path Following

There are different methods to accomplish path following, but two of the most common methods in FRC are following a pre-generated trajectory of velocity points for each wheel, or following a Pure Pursuit Controller. We are going to go more in dept with the Pure Pursuit Controller since it is the method that we use on our team.

8 Principles of the Pure Pursuit Controller

8.1 Overview

The pure pursuit controller is a method of path following that involves the robot driving towards a target point on the 2D path a certain distance ahead of the robot.

8.2 References

The pure pursuit controller can be a difficult concept to grasp, so here are some documents and papers that helped us understand it and implement it into our code:

The concept of the pure pursuit controller: https://www.mathworks.com/help/robotics/ug/pure-pursuit-controller.html

Pure pursuit controllers in the FRC competition and how to implement it into code: https://www.chiefdelphi.com/t/paper-implementation-of-the-adaptive-pure-pursuit-contro 166552

8.3 Concept

This document is going to describe the pure pursuit controller at a very high and basic level so you can understand the concept of how it works when going into the usage of our library, but if you are interested in more of the lower level details of the pure pursuit controller, we highly recommend checking out this paper by FRC team 1712: https://www.chiefdelphi.com/t/paper-implementation-of-the-adaptive-pure-pursuit-controller/166552.

8.3.1 Overview

The basic concept of the pure pursuit controller is to drive towards a point in front of the robot, called the target or look ahead point, that is continuously updating to new positions along a 2D path as the robot follows it.

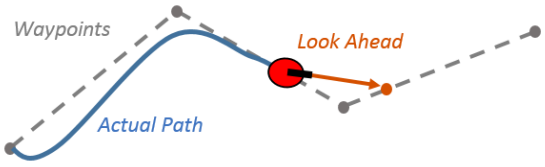

Figure 6: Diagram of the pure pursuit controller (https://www.mathworks.com/help/robotics/ug/pure-pursuit-controller.html)

In **Figure 6** displaying the pure pursuit controller in action, the robot is the red

circle and the look ahead point is the orange circle. When the robot is following a constantly moving look ahead point, it results in the robot moving in a very smooth path. Notice how the distance between the robot and the look ahead point is called the look ahead distance. Look ahead is the term used to describe the robot's following of a point in front of it.

8.3.2 Finding the look ahead point

The look ahead point is determined by finding a point on the 2D path that you want the robot to follow that is a certain specified distance away from the robot at that current time. You know the robot's location using odometry, and you give the pure pursuit controller a 2D path containing a list of points that make it up, so the look ahead point is whatever point on that path is the closest to the look ahead distance away from the robot and is in front of the robot at that current time.

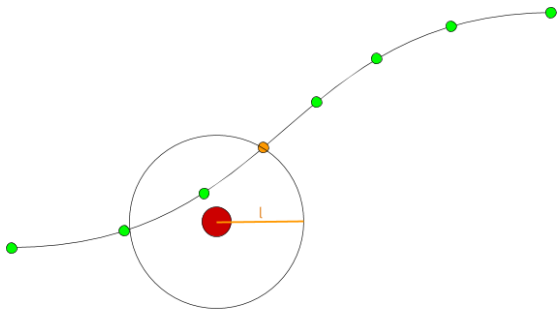

Figure 7: Look ahead point diagram

In **Figure 7**, the robot is the red circle, the orange line is the look ahead distance, and the highlighted orange point is the calculated look ahead point for the robot's position since it is the closest to the look ahead distance away from the robot and is in front of the robot. Each green point represents a point that makes up the path, but this diagram is simplified a lot. Normally there would be about 200 points in a path like this.

8.3.3 Following the look ahead point

Once the look ahead point is found, the robot needs to actually move to that point. This can be done by creating an arc from the robot to the point. The arc is a circle that is tangent to the robot's direction vector and intersecting with the look ahead point.

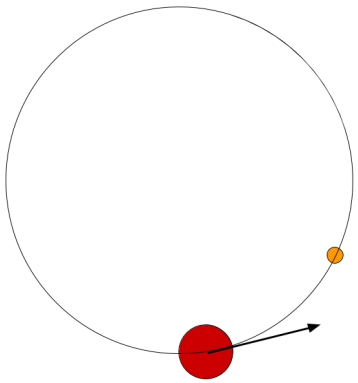

Figure 8: Arc to look ahead point diagram

In **Figure 8**, the robot is the red circle, the robot's direction is represented by the arrow, the look ahead point is the orange circle, and the arc that the robot needs to follow is represented in the diagram.

Once the arc to the look ahead point is calculated, you need the radius of that arc. Using differential drive kinematics (http://www.cs.columbia.edu/~allen/F17/NOTES/icckinematics.pdf) that will be described in the next section, you can then calculate the velocity needed for the robot to follow that arc.

8.3.4 Differential drive kinematics

Differential drive kinematics can describe the calculations for determining the velocity that the left and right wheels of the drivetrain must move at to follow a certain arc.

First we can define a few given variables: We can specify the base velocity v_b we want the robot to be driving around the arc at, we know the distance from the left and right wheel on the chassis (known as the track width of the chassis), Tw, and we know the radius r of the arc from the robot to the look ahead point. Using these, the equation for the left and right wheel velocities are the following:

$$v_a = v_b/r$$

$$v_l = v_a(r - (Tw/2))$$

$$v_r = v_a(r + (Tw/2))$$

where v_a is the angular velocity of the robot, v_l is the left wheel velocity, and v_r is the right wheel velocity.

Once you have the left and right wheel velocities, you can simply pass these as the setpoint for a velocity control loop on your left and right drivetrain motors to drive towards the look ahead point.

8.3.5 Putting it all together

At this point we are able to find a look ahead point and calculate the left and right wheel velocity to reach that look ahead point. Those are the main components in the pure pursuit controller, and in order to follow a full path you can simply put these calculations into a loop that is periodically called (similarly to the motion profile, a period of about 0.02 seconds per loop generally works well) and it will update the wheel velocities based on the look ahead point as the robot moves along the path. The result is the robot following a new arc every loop that is adjusted based on where the robot is relative to the look ahead point, and it will follow the path quite well.

In the end, the actual motion of the robot is not calculated from the path but instead from a series of arcs to get the robot from its current position to the look ahead point on the path.

9 Our Pure Pursuit Controller Library

9.1 Overview

Our team's pure pursuit controller library can be found on GitHub: https://github.com/MittyRobotics/pure-pursuit-controller. Basic instructions on the usage of the library as well as how to add it to your code can be found on the GitHub page, but you can continue reading in this document for more detailed information.

9.2 Path objects

The paths that you want the robot to follow are held in Path objects. Path objects contain an array of point objects that make up the path. The path can be generated using one of the algorithms available in our library, currently only the Cubic Hermite Spline (https://en.wikipedia.org/wiki/Cubic_Hermite_spline) algorithm. In order to create a new path, you must first create an array of defining waypoints for the path to pass through held in the Coordinate object. Coordinate objects take in an x, y, and angle value (following the same coordinate system described in the Autonomous Movement section).

```
Coordinate[] coordinates = new Coordinate[]{
    new Coordinate(0, 0, 0),
    new Coordinate(50, 100, 0)
};
```

Next, similarly to the motion profiles, you need to define velocity constraints for the path to follow. Velocity constraints are held in the VelocityConstraints object and take in a maximum acceleration, maximum deceleration, maximum velocity, a starting velocity, an ending velocity, and a gain that makes the velocity slow down around turns.

```
double maxAcceleration = 40; //max acceleration units/s^2
double maxDeceleration = 40; //max deceleration units/s^2
double maxVelocity = 100; //maximum velocity units/s
double startVelocity = 0; //starting velocity units/s
double endVelocity = 0; //ending velocity units/s
double kCurvature = 0.8; //gain that slows down the robot
    when it turns corners, generally a value between 0.8 and
     2 works well depending on how slow you want the robot
    to turn corners. The lower the value, the more it slows
    down at turns
VelocityConstraints velocityConstraints = new
    VelocityConstraints(40, 40, 100, 0, 0, 1);
```

The path takes in the velocity constraints because each point on the path has a velocity associated with it calculated by the specified constraints. As the robot follows along the path, the base average velocity for the differential drive kinematics equations are then input as the velocity associated with the point closest to the robot. This results in an easy and efficient way to determine the robot's base velocity at a given time.

Next, the Path object can be made by creating a new CubicHermitePath object:

```
Path path = new CubicHermitePath(coordinates,
    velocityConstraints);
```

9.3 PurePursuitController object

Now we can create the main PurePursuitController object, which will contain the function to update the controller and get the left and right wheel velocities. The PurePursuitController object takes in the path, a default look ahead distance, and a minimum look ahead distance (the look ahead distance is adapted

based on the velocity of the robot, the default distance is the distance when it is at full velocity and the minimum distance is the distance at minimum velocity). Generally good look ahead distances are a default of 20 inches and a minimum of 15 inches.

```
double defaultLookaheadDistance = 20; //inches
double minimumLookaheadDistance = 15; //inches
boolean reversed = false; //whether or not the robot should
    be running backwards

PurePursuitController controller = new
    PurePursuitController(path, defaultLookaheadDistance,
    minimumLookaheadDistance, reversed);
```

9.4 PathFollowerPosition object

The PathFollowerPosition object is a singleton that interfaces your odometry object with the PurePursuitController object. This must be updated with your odometry measurements using the object's update() function. Update takes in 5 parameters: x, y, heading, current left wheel velocity, and current right wheel velocity. Remember, the PathFollowerPosition object takes in coordinates in the standardized coordinate system described in the Autonomous Movement section.

```
void updatePathFollowerPosition(){
    double x = Odometry.getInstance().getX(); //The x value
        of your odometry position
    double y = Odometry.getInstance().getY(); //The y value
        of your odometry position
    double heading = Odometry.getInstance().getHeading();
        //The heading value of your odometry position
    double leftVelocity = getLeftWheelVelocity(); //the
        velocity of your left drivetrain wheels
    double rightVelocity = getRightWheelVelocity(); //the
        velocity of your right drivetrain wheels

    PathFollowerPosition.getInstance().update(x, y, heading
        , leftVelocity, rightVelocity); //Updates the
        PathFollowerPosition object with your odometry
        values
}
```

9.5 Following the pure pursuit controller

Just like the TrapezoidalMotionProfile object in the motion profile library, the PurePursuitController object has a very easy update() function that calculates all the values and returns a PurePursuitOutput object. PurePursuitOutput objects contain a left and right wheel velocity that can be accessed by output.getLeftVelocity() and output.getRightVelociuty().

In order to follow the pure pursuit controller, it is almost exactly the same

as the motion profile. Simply create a loop that iterates at a certain frequency, call the PurePursuitController.update() function each loop, and pass the left and right wheel velocities into a control loop. The update function also takes in the time since the loop began, so keep track of that and pass it into the update function each loop.

```
//Call this function at a certain frequency (generally
    around every 0.02 seconds). If you are using the FRC
    Commands and Subsystems structure and you put this in
    the execute function of a command, it will automatically
     get called every robot period.
void execute(){
    double t = timeSinceInitialized(); //Time since you
        started following the pure pusuit controller.

    updatePathFollowerPosition(); //Update
        PathFollowerPosition object with odometry values

    PurePursuitOutput output =
    purePursuitController.update(t); //Update the pure
        pursuit controller and get the PurePursuitOutput
        object

    setLeftVelocitySetpoint(output.getLeftVelocity());//set
         the setpoint of a velocity control loop on the left
         drivetrain motors.
    setRightVelocitySetpoint(output.getRightVelocity());//
        set the setpoint of a velocity control loop on the
        right drivetrain motors.
}
```

9.6 Reversed mode

In order for the robot to follow a path backwards, simple set the reversed boolean to true in the PurePursuitController constructor parameters. Also, make sure to reverse all of your path coordinates (reverse the angles: 0 turns into 180, 90 turns into -90, etc. Reverse the position: +50 turns into -50, 100 turns into -100, etc). Think of reversing the coordinates as the robot is starting at the end of the path and working its way backwards.

```
//Forward coordinates
Coordinate[] forwardCoordinates = new Coordinate[]{
    new Coordinate(0, 0, 0),
    new Coordinate(50, 100, 0)
};
//Backward coordinates
Coordinate[] backwardCoordinates = new Coordinate[]{
    new Coordinate(0, 0, 180),
    new Coordinate(-50, -100, 180)
};
```

9.7 Visualizing paths

10 Vision

10.1 Overview