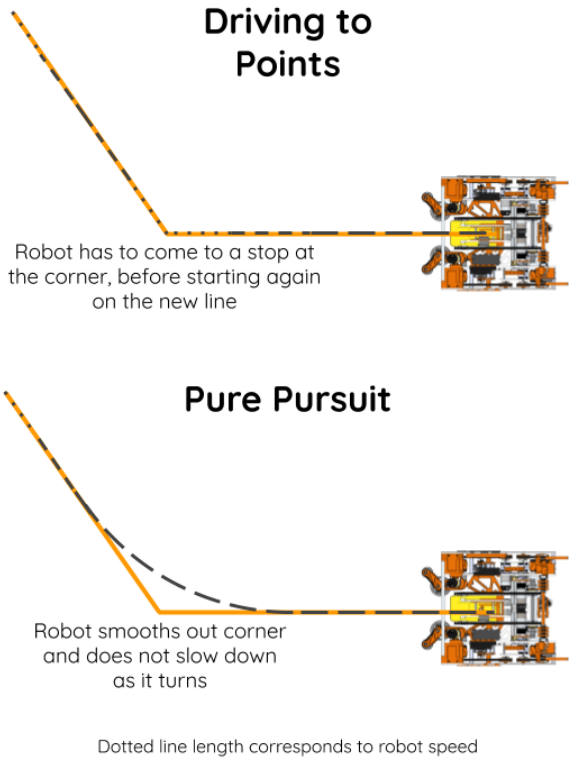
Pure Pursuit

Keertik Bacon - FTC 9866 Virus

July 25, 2020

Introduction

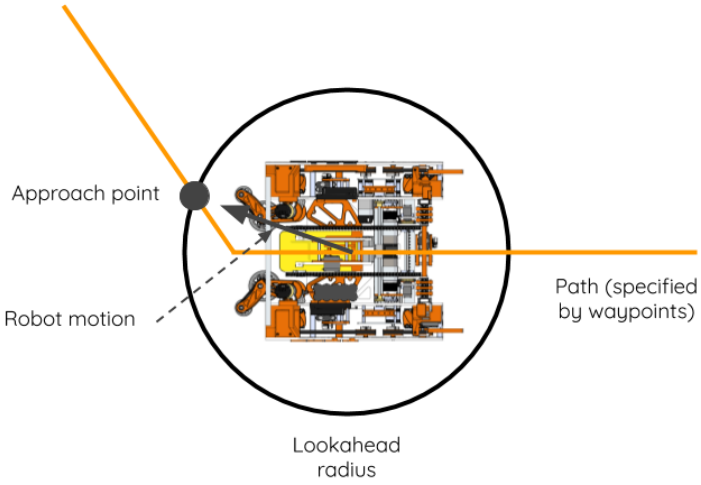
Pure pursuit is a system to make robot movement more efficient, by letting the robot turn corners without having to stop. It saves valuable time during autonomous, and allows more complex paths to be done.



Prerequisites

The two prerequisites are an odometry system, to keep track of the robot's location and heading, as well as a field-centric "go-to-position" algorithm. This algorithm should accept a target location and heading, and move the robot to this position, using position and heading PID controllers.

Overview



The general logic of pure pursuit is that the we are dangling a carrot on a stick in front of the robot; the robot approaches not a set endpoint, but instead a "look-ahead" point that is a set distance further along the path than it. As the look-ahead point reaches and turns corners before the robot does, the robot begins turns early and thus rounds those corners.

To achieve this effect, we place a circle around the robot. Where this circle intersects the path will determine the current look-ahead point. The radius of this circle is the look-ahead radius, and will affect how much the corners are rounded. A large radius will result in the robot loosely rounding corners and deviating somewhat from the path as a result. A small radius will result in the robot tightly rounding corners, running the risk of accidentally losing the path as a result of the sudden direction changes.

Example Implementation

Now that we have an understanding of how pure pursuit works, let's look at an example of how to implement it, with the system we used this past season. Note that there is no definitive way to do pure pursuit; this section is only an example, and should be used as a jumping-off point.

Defining the Path

The path is defined in the code as a list of Waypoint objects. Each Waypoint contains an x-coordinate, y-coordinate, and a heading; these are the location of the waypoint and the direction that the robot should face upon arriving at that location.

This list of waypoints is then used to generate a list of Line objects that are all the lines in the path. Each line is defined parametrically, with three equations generated for x, y, and heading, respectively.

$$x = x_0 + (x_1 - x_0)t$$
$$y = y_0 + (y_1 - y_0)t$$
$$\theta = \theta_0 + (\theta_1 - \theta_0)t$$

In this case, the first waypoint is at (x_0, y_0), with a heading of θ_0, while the second waypoint is at (x_1, y_1), with a heading of θ_1. For simplicity, the parametric variable t is defined such that $t = 0$ is at the first waypoint, while $t = 1$ is at the second waypoint.

Finding the Approach Point

To find the point that the robot should approach, we need to see where the look-ahead radius circle intersects the path. First, the code picks out the line that the robot is currently on (the line with the previous approach point, or the first line if this is the first loop), and adds on the next two lines. If the path is nearing the end, this list of lines will be truncated as necessary.

Each line in the list is checked individually for intersections with the look-ahead radius circle, and these intersection points are compiled into a list. Let's look at how to derive the equation we use for this.

First, we create an equation for the circle around the robot. (h, k) is the current location of the robot, and r is the look-ahead radius.

$$(x - h)^2 + (y - k)^2 = r^2$$

Next, we substitute in the line's parametric equations.

$$[x_0 + (x_1 - x_0)t - h]^2 + [y_0 + (y_1 - y_0)t - k]^2 = r^2$$

To find the intersection point, and properly match it up with the correct heading, we need to solve for t. That first means expanding out this equation.

$$\begin{aligned} &x_0^2 + x_0 t\,(x_1 - x_0) - h x_0 + x_0 t\,(x_1 - x_0) + t^2\,(x_1 - x_0)^2 - h t\,(x_1 - x_0) - h x_0 - h t\,(x_1 - x_0) + h^2 \\ &+ y_0^2 + y_0 t\,(y_1 - y_0) - k y_0 + y_0 t\,(y_1 - y_0) + t^2\,(y_1 - y_0)^2 - k t\,(y_1 - y_0) - k y_0 - k t\,(y_1 - y_0) + k^2 = r^2 \end{aligned}$$

Now we can simplify, bringing like terms together.

$$\begin{aligned} &x_0^2 + 2x_0 t\,(x_1 - x_0) - 2h x_0 + t^2\,(x_1 - x_0)^2 - 2h t\,(x_1 - x_0) + h^2 \\ &\qquad + y_0^2 + 2y_0 t\,(y_1 - y_0) - 2k y_0 + t^2\,(y_1 - y_0)^2 - 2k t\,(y_1 - y_0) + k^2 = r^2 \end{aligned}$$

$$\begin{aligned} &\left[(x_1 - x_0)^2 + (y_1 - y_0)^2\right] t^2 + \left[2x_0\,(x_1 - x_0) - 2h\,(x_1 - x_0) + 2y_0\,(y_1 - y_0) - 2k\,(y_1 - y_0)\right] t \\ &\qquad + \left[x_0^2 - 2h x_0 + h^2 + y_0^2 - 2k y_0 + k^2 - r^2\right] = 0 \end{aligned}$$

$$\begin{aligned} &\left[(x_1 - x_0)^2 + (y_1 - y_0)^2\right] t^2 + 2\left[(x_0 - h)\,(x_1 - x_0) + (y_0 - k)\,(y_1 - y_0)\right] t \\ &\qquad + \left[x_0^2 - 2h x_0 + h^2 + y_0^2 - 2k y_0 + k^2 - r^2\right] = 0 \end{aligned}$$

Now that the equation is in the form of a quadratic, we can use the quadratic formula to solve for t. To recap, (h, k) is the current position of the robot, r is the look-ahead radius, and (x_0, y_0) and (x_1, y_1) are the start and end points of the line currently being looked at, respectively.

$$a = (x_1 - x_0)^2 + (y_1 - y_0)^2$$

$$b = 2\left[(x_0 - h)\,(x_1 - x_0) + (y_0 - k)\,(y_1 - y_0)\right]$$

$$c = x_0^2 - 2h x_0 + h^2 + y_0^2 - 2k y_0 + k^2 - r^2$$

$$t_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$$t_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

To ensure that the code does not have to deal with imaginary numbers, we first evaluate the discriminant $(b^2 - 4ac)$ to determine how many solutions we are looking for. If the discriminant is negative, a blank list of intersections is returned for this line, and the code moves on to the next line in the path. Otherwise, the code finds the values of t for which the line intersects with the circle. These values are bounded between 0 and 1—so that any out-of-bounds point is brought to the line segment—and then they are used to calculate the x, y, and θ values that they correspond to. These points are then added onto the list of intersection points. To ensure that there are no duplicate points, we specifically use a LinkedHashSet to hold these points, as it automatically removes duplicates.

After all the intersection points have been found on the designated lines, we need to pick the point farthest along the path to set as the new approach point. If you evaluate each line in its order along the path, and for each line you add t_1's point onto the intersection point list before t_2's point, the list will be in order, and the point furthest along the line will simply be the final point in the list.

Moving Along the Path

Now that we can determine the point that we are approaching, the next step is to have the robot actually approach that point.

Go to Position

We need a special version of the go-to-position code to work with pure pursuit. Firstly, the code should not loop. The approach point changes as the robot moves, and so we don't want the robot to target a stationary point. Thus, we need a go-to-position method that will execute once, calculating a power for each of the drive motors with the current target location data, and then exit, to be called again later with new data.

This method should also return a boolean that indicates whether the robot has reached the point it is currently approaching. This will help us in knowing when to terminate the pure pursuit loop.

With both of these considerations, this is how the general structure of the go-to-position code should look:

```
public boolean goToPosition(Point newPosition){
    // Update the current position and heading
    updatePosition();

    // Calculate the motor powers
    double[] motorPowers = calcMotorPowers(currentPosition, newPosition);

    // Set the motor powers
    Robot.runMotors(motorPowers);

    // See whether robot has reached the target point
    boolean hasReached = currentPosition.equals(targetPosition);
    return hasReached;
}
```

Path-following

The structure of the path-following is as follows:

```
public void followPath(Path path){
    boolean reachedEnd = false;
    while(!reachedEnd){
        // Update the current position and heading
        updatePosition();
```

```
        // Find the approach point
        Point approachPoint = findApproachPoint();

        /* Move towards the approach point and also check if the robot
        has reached the end of the path */
        reachedEnd = Robot.goToPosition(approachPoint);
    }

}
```

You'll notice that we use the boolean from goToPosition() (which states whether the robot has reached the target position) for determining whether we have reached the end of the path. This works because of the fact that the robot is only able to actually reach the approach point at the end of the path.

For most of the path, the robot will be chasing the approach point, but once the approach point reaches the final waypoint in the path, it will become stationary. This is because after the look-ahead radius circle goes past the final waypoint, out of the intersections calculated, the one farthest along the line will have a t-value greater than 1. As a result, that t-value will be brought down to 1, placing the approach point at the final waypoint.

Robot Speed

Because of how position PID algorithms slow the robot down the closer it is to the target point, the distance between the robot and the approach point—the look-ahead radius—essentially serves as a cap to the robot's speed. This means that with a small or even medium-sized look-ahead radius, the robot will navigate through the path really slowly.

We can fix this problem one of two ways:

1. Disable the PID algorithm determining speed until the robot reaches the final line in the path, at which point the algorithm is reenabled.
2. Create a displacement vector between the current position and target position, and replace the magnitude of the vector with the remaining distance along the path all added up. In other words, essentially straighten out the rest of the path, and place it in the direction of the current target position. Use the vector's new endpoint as the modified target position, so that the robot moves in the same direction, but at a much higher speed.

In our pure pursuit system, we used Option #2.

Conclusion

Potential Improvements

Pure pursuit is hardly a settled science. The method described above is not the definitive way to implement it, and improvements to it can surely be made. Let's look at a few:

1. **Add other variables to waypoints:** We could allow things like the look-ahead radius or the robot's maximum speed to vary throughout the path. All that is required to implement this is adding another parameter to the Waypoint class, creating a new parametric equation in the Line class in the format $a = a_0 + (a_1 - a_0)t$, and calculating the approach point's value for this variable using this equation and the t-value of that point.
2. **Have the robot always face the direction of motion:** Mecanum drivetrains move the fastest when going forwards/backwards, as there is no cancellation in the wheels' movement vectors. To always move in this direction, we can create a version of the followPath() method

that, rather than giving to goToPosition() the approach point's calculated heading, instead gives to it the direction of the displacement vector between the current and target positions, to make the robot always face towards the target position.

3. **Add other types of path components:** We could expand the types of path components beyond just lines, to include perhaps circular, or even quadratic paths. This would involve creating new classes similar to the Line class, which would all fall under an umbrella PathComponent class. For each different path type, we would have to make new parametric equations, and derive a new equation for finding intersection points.

Other Methods

There are other teams who have implemented pure pursuit in slightly different manners. Gluten Free (11115), for example, uses the same basic concept of determining the approach point by placing a look-ahead radius circle around the robot and seeing where it intersects with the lines in the path, but their implementation of the specifics of the algorithm, such as with how they calculate the robot's speed, or even with how they store the path data, is different. If you want to see an alternate approach to pure pursuit, watch their YouTube tutorial series, the first video of which is at: https://youtu.be/3l7ZNJ21wMo.