Spline-Based Path Following

Qingyuan Li

Code & demonstration video found on Github.

Problem Statement

Create an algorithm that interpolates a smooth path between two *poses*. Given the robot pose relative to the path, return *differential drive* motor outputs to follow the path based on a motion profile.

Pose: (x, y, θ) combination of point and angle.
Differential drive: drivetrain model assuming two separately driven wheels on either side of the robot.

Methodology

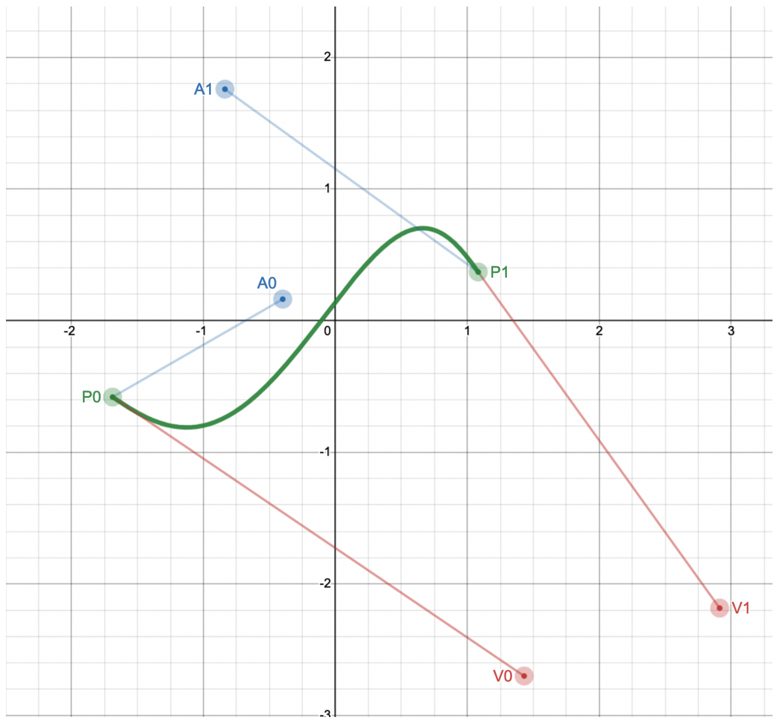
1. The Path

I decided to use Quintic Hermite splines, a type of Hermite interpolation that creates a parametric function $c(t)$ satisfying
$c(0) = p_0, c(1) = p_1, c'(0) = v_0, c'(1) = v_1, c''(0) = a_0, c''(1) = a_1$.
The most convenient definition was using six basis functions (fifth degree polynomials) and multiplying by the position, velocity, and acceleration vectors:

$$\mathbf{c}(t) = H_0^5(t)\,\mathbf{p}_0 + H_1^5(t)\,\mathbf{v}_0 + H_2^5(t)\,\mathbf{a}_0 + H_3^5(t)\,\mathbf{a}_1 + H_4^5(t)\,\mathbf{v}_1 + H_5^5(t)\,\mathbf{p}_1$$

$$H_0^5(t) = 1 - 10t^3 + 15t^4 - 6t^5$$
$$H_1^5(t) = t - 6t^3 + 8t^4 - 3t^5$$
$$H_2^5(t) = \frac{1}{2}t^2 - \frac{3}{2}t^3 + \frac{3}{2}t^4 - \frac{1}{2}t^5$$
$$H_3^5(t) = \frac{1}{2}t^3 - t^4 + \frac{1}{2}t^5$$
$$H_4^5(t) = -4t^3 + 7t^4 - 3t^5$$
$$H_5^5(t) = 10t^3 - 15t^4 + 6t^5$$

This yields a smooth parametric between the two points whose curvature and "momentum" can easily be edited via the velocity and acceleration vectors.

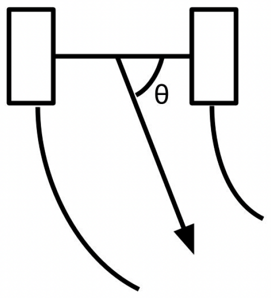


Live Demo: https://www.desmos.com/calculator/fikl0h8vvi

2. The Robot Pose

To track the field-relative robot pose in (x, y, θ) form, I used a combination of gyro, *encoder* values and computer vision pose data. To integrate these different value points, I used a *Kalman filter*. Whereas the gyro and encoder are precise in the short-term, vision data's accuracy can be used to correct any long-term error buildup. This was accounted for using the appropriate noise covariance matrices.

Encoder to pose:
Since the code is being run every 20ms, I can interpolate the arc as a line segment with the length being the average distance driven by the two motors and the direction being the gyro angle. This vector is added to the (x, y) pose every cycle. The gyro updates θ automatically.



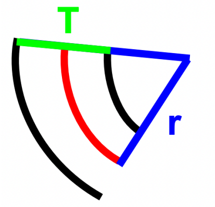
Encoder: sensor that measures how much a motor turns.
Kalman filter: common state-space algorithm that uses measurements over time to create a more accurate estimate of a state, by estimating a joint probability distribution of the state's variables.

3. Differential Drive Kinematics

Differential drive, by definition, must travel along the arc of a circle. I define it using linear velocity, *l*, and angular velocity, *w*.

From the common formula *l=wr*, I can derive the radius *r* of the differential drive circle from *l* and *w*. Then, the velocities of the left and right wheels given only depend on the distance between them, which I will call the trackwidth (*T*).

$r = l/w\,,\ v_{left} = w \cdot (r - T/2)\,,\ v_{right} = w \cdot (r + T/2)\,.$



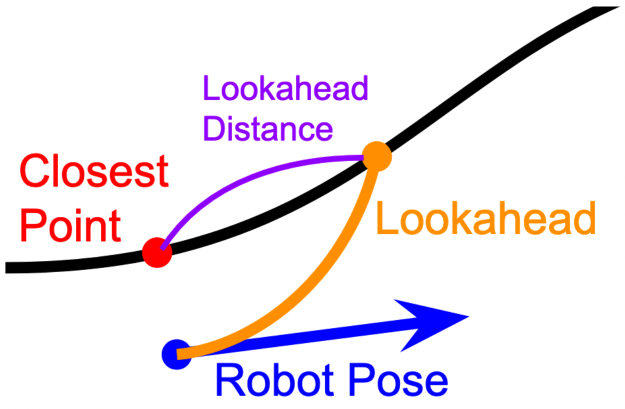
This is easily derived from *l=wr* where the radius for each wheel is different. To account for turning right versus turning left, I define turning right to be negative angular velocity. This also makes radius negative, so the two cancel, such that $v_{left} > v_{right} > 0$.

4. Following the Path

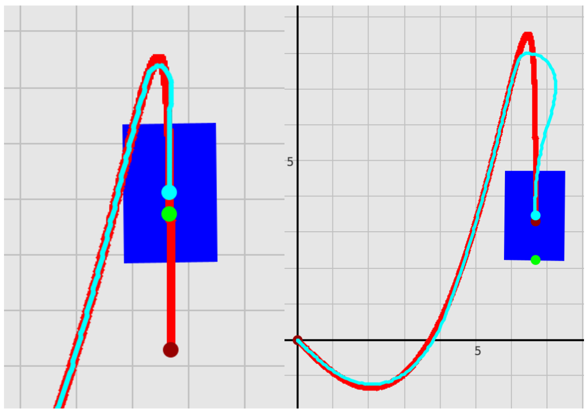
To break down the two path following algorithms I implemented:

Pure Pursuit takes in the robot pose and the spline, then finds the point on the spline closest to the robot pose. It then finds the point on the spline a certain distance in front of the closest point.

This distance, called the lookahead, can be tuned to edit the behavior of the path. It then interpolates a circular arc between the robot pose and the lookahead that is tangent to the robot pose angle and intersects both points.



This arc can be followed using differential drive kinematics by inputting the linear velocity (from the motion profile) and radius of the arc, this time solving for the angular instead. Although the arc looks sharp here, the algorithm is updating every 20ms, so as the robot gets closer to the spline the curvature decreases. This has the effect of rapidly correcting any error while minimizing overshooting.



Examples of error correction in my simulator. Note how well the first robot follows the spline given no physical errors with a smaller lookahead. In practice, a longer lookahead distance like the one in the second image would be used to make the path more adaptable to physical error. This parameter is tuned for every path.

The ***Ramsete Unicycle Controller*** is a lot more complex. It is based on the following paper: https://www.researchgate.net/publication/225543929_Control_of_Wheeled_Mobile_Robots_An_Experimental_Overview.

It uses arbitrary functions that have associated Lyapunov functions which prove global asymptotic stability. It takes in the robot pose and desired pose (in our case, the closest point on the spline) and uses the errors, in combination with two gain constants, b and ζ, to return a linear and angular velocity that results in rapid convergence with the spline. The linear and angular velocity outputs can be converted into differential drive outputs through the aforementioned kinematics.

https://file.tavsys.net/control/controls-engineering-in-frc.pdf offers a summary of the equations:

$$\begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d - x \\ y_d - y \\ \theta_d - \theta \end{bmatrix}$$

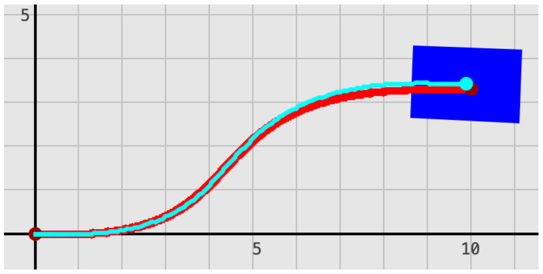
$$v = v_d \cos e_\theta + k e_x$$

$$\omega = \omega_d + k e_\theta + b v_d \operatorname{sinc}(e_\theta) e_y$$

$$k = 2\zeta\sqrt{\omega_d^2 + b v_d^2}$$

$$\operatorname{sinc}(e_\theta) = \frac{\sin e_\theta}{e_\theta}$$

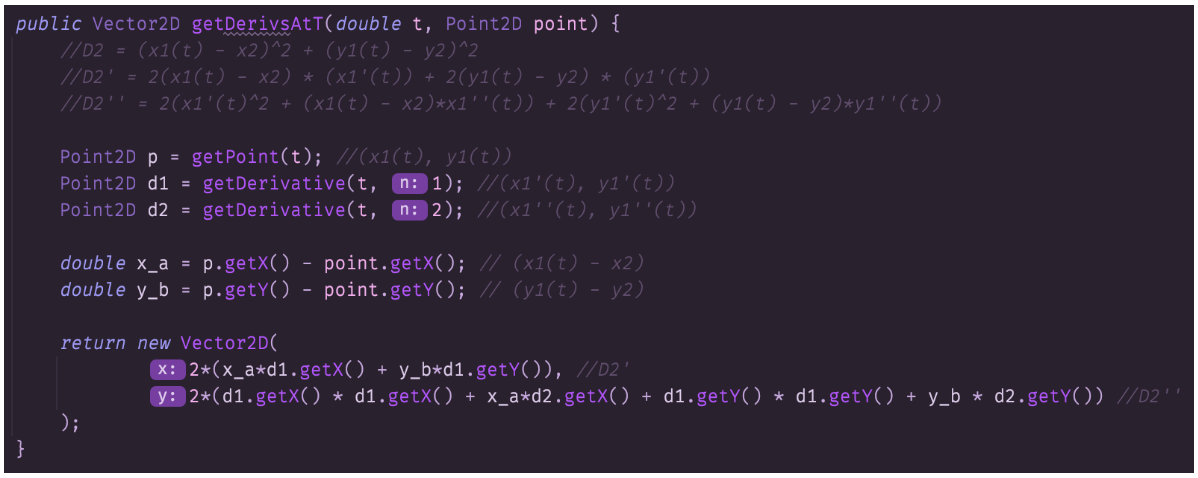
v	velocity command	v_d	desired velocity
ω	turning rate command	ω_d	desired turning rate
x	actual x position in global coordinate frame	x_d	desired x position
y	actual y position in global coordinate frame	y_d	desired y position
θ	actual angle in global coordinate frame	θ_d	desired angle



Here, it converges... somewhat. Having two somewhat arbitrary values makes Ramsete harder to tune. However, for short curves it is preferred to Pure Pursuit, which is less accurate at short distances because it follows its own arc and not the spline.

5. Implementing these Algorithms

For either of these algorithms, I need the point on the spline closest to the robot pose.
I found that the easiest way was to use Newton's method on the square of the distance function $D^2=d^2$ between the spline and the point.

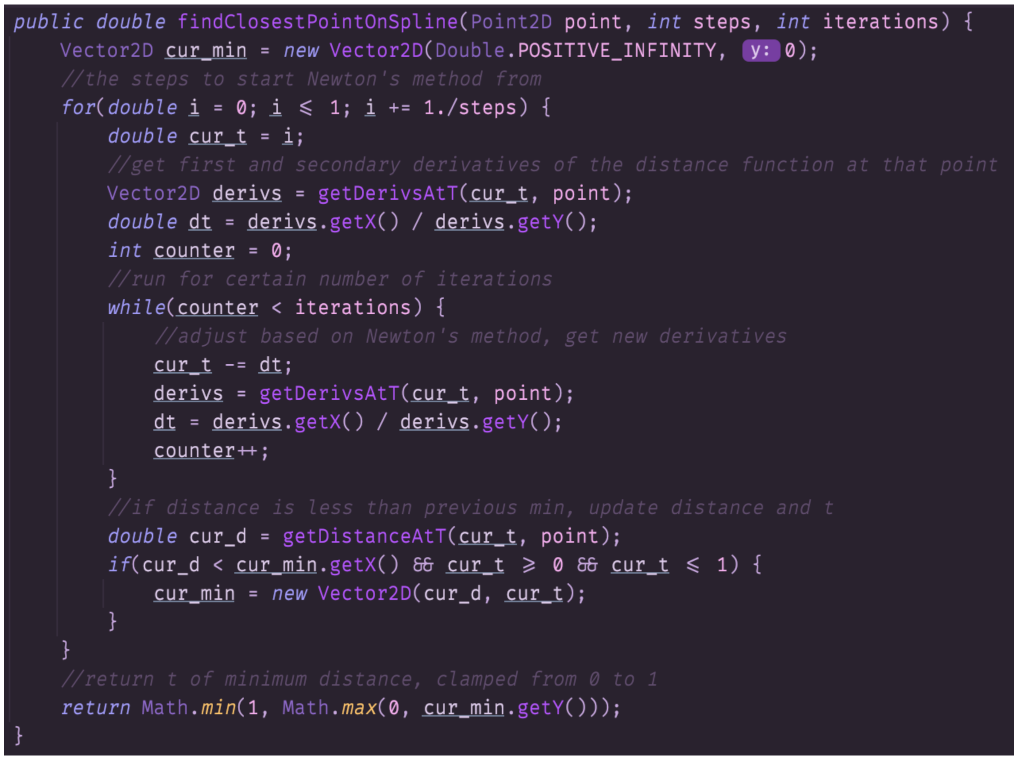


```
public Vector2D getDerivsAtT(double t, Point2D point) {
    //D2 = (x1(t) - x2)^2 + (y1(t) - y2)^2
    //D2' = 2(x1(t) - x2) * (x1'(t)) + 2(y1(t) - y2) * (y1'(t))
    //D2'' = 2(x1'(t)^2 + (x1(t) - x2)*x1''(t)) + 2(y1'(t)^2 + (y1(t) - y2)*y1''(t))

    Point2D p = getPoint(t); //(x1(t), y1(t))
    Point2D d1 = getDerivative(t, n: 1); //(x1'(t), y1'(t))
    Point2D d2 = getDerivative(t, n: 2); //(x1''(t), y1''(t))

    double x_a = p.getX() - point.getX(); // (x1(t) - x2)
    double y_b = p.getY() - point.getY(); // (y1(t) - y2)

    return new Vector2D(
            x: 2*(x_a*d1.getX() + y_b*d1.getY()), //D2'
            y: 2*(d1.getX() * d1.getX() + x_a*d2.getX() + d1.getY() * d1.getY() + y_b * d2.getY()) //D2''
    );
}
```

This function returns the 1st and 2nd derivatives of the squared distance function at a specific *t* value of the parametric for a given point. According to Newton's method, to get the zeroes a function, we can iterate our initial guess *x* many times:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

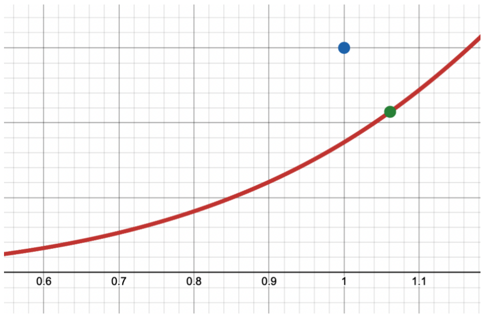
We do this with $D^2{}'$ and $D^2{}''$ instead of D^2 and $D^2{}'$ to find the zeros of $D^2{}'$, which are the local minimums/maximums of the squared distance. To account for all local minimums, I iterate through a series of equidistant initial starting points and use the one that returns the overall minimum distance.



```java
public double findClosestPointOnSpline(Point2D point, int steps, int iterations) {
    Vector2D cur_min = new Vector2D(Double.POSITIVE_INFINITY, y: 0);
    //the steps to start Newton's method from
    for(double i = 0; i <= 1; i += 1./steps) {
        double cur_t = i;
        //get first and secondary derivatives of the distance function at that point
        Vector2D derivs = getDerivsAtT(cur_t, point);
        double dt = derivs.getX() / derivs.getY();
        int counter = 0;
        //run for certain number of iterations
        while(counter < iterations) {
            //adjust based on Newton's method, get new derivatives
            cur_t -= dt;
            derivs = getDerivsAtT(cur_t, point);
            dt = derivs.getX() / derivs.getY();
            counter++;
        }
        //if distance is less than previous min, update distance and t
        double cur_d = getDistanceAtT(cur_t, point);
        if(cur_d < cur_min.getX() && cur_t >= 0 && cur_t <= 1) {
            cur_min = new Vector2D(cur_d, cur_t);
        }
    }
    //return t of minimum distance, clamped from 0 to 1
    return Math.min(1, Math.max(0, cur_min.getY()));
}
```

The returned *t* can then be plugged into *c(t)* to get the closest point. Done!

Here is the result with 25 initial starting points:



For Pure Pursuit, I also need a way to get the lookahead, a certain distance in front of the closest point on the spline.

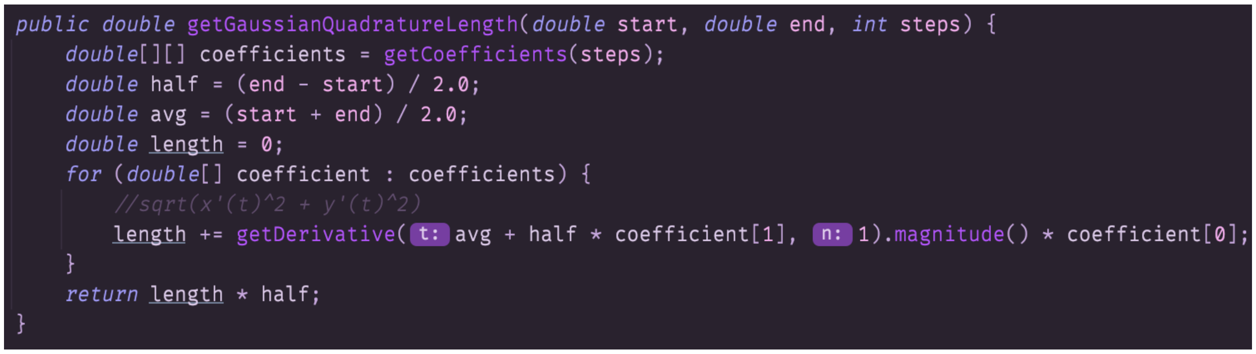
Any approach will need a way to efficiently get the length of the spline, or a portion of it. Because a Quintic Hermite spline is defined by polynomials, Gaussian quadrature is the easiest method to efficiently get the length. From Wikipedia:

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

This is on the interval [-1, 1]. For other intervals, we can use the change of interval formula:

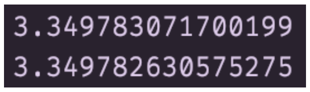
$$\int_{a}^{b} f(x)\,dx \approx \frac{b-a}{2} \sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}\xi_i + \frac{a+b}{2}\right)$$

I implemented this in a function:



```java
public double getGaussianQuadratureLength(double start, double end, int steps) {
    double[][] coefficients = getCoefficients(steps);
    double half = (end - start) / 2.0;
    double avg = (start + end) / 2.0;
    double length = 0;
    for (double[] coefficient : coefficients) {
        //sqrt(x'(t)^2 + y'(t)^2)
        length += getDerivative( t: avg + half * coefficient[1],  n: 1).magnitude() * coefficient[0];
    }
    return length * half;
}
```

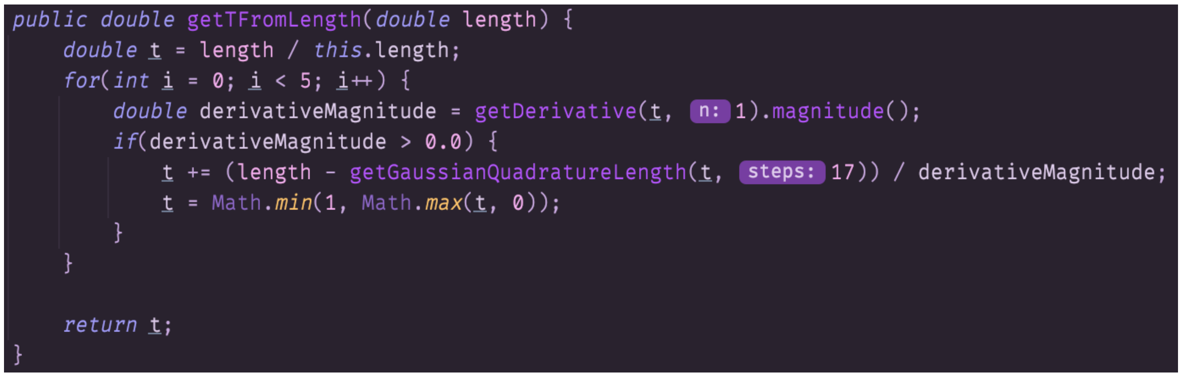
In practice, I almost always use 17-th degree weights and abscissae to balance accuracy and speed.



```
3.349783071700199
3.349782630575275
```

First value is using 17-th degree Gaussian quadrature, second is by manually adding up the distance between 1000 equidistant points on the spline (brute force approximation). The accuracy of quadrature with polynomials is frankly uncanny.

With a way to get the length of a portion of the spline, getting the lookahead is relatively easy. I can get the length of the spline up to the closest point, add the lookahead distance, and compute the corresponding *t* value that corresponds to that length. I wrote a short length-to-t algorithm:

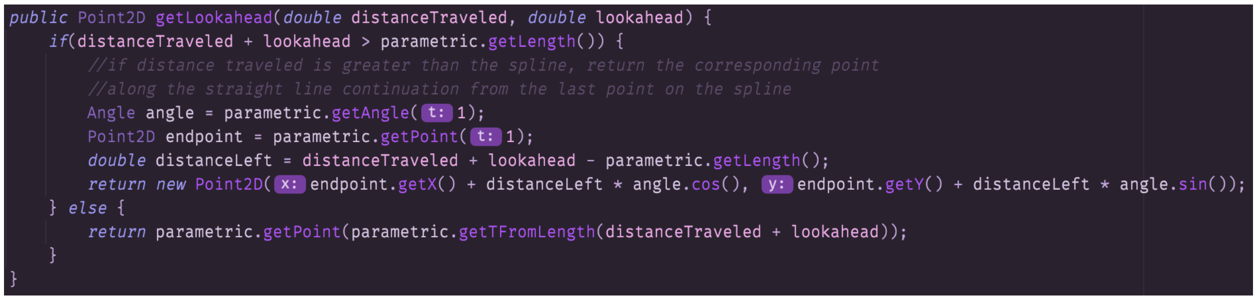


```java
public double getTFromLength(double length) {
    double t = length / this.length;
    for(int i = 0; i < 5; i++) {
        double derivativeMagnitude = getDerivative(t, n: 1).magnitude();
        if(derivativeMagnitude > 0.0) {
            t += (length - getGaussianQuadratureLength(t, steps: 17)) / derivativeMagnitude;
            t = Math.min(1, Math.max(t, 0));
        }
    }

    return t;
}
```

This first approximates *t* by dividing the desired length by the total length of the spline. Then, for five iterations, I assume that the path follows its derivative at that point perfectly, and add (remaining error / magnitude of derivative) to t.

For example, if we were at 4 inches with a desired length of 5 inches, and the magnitude of the derivative is 20 inches/*t*, we would add (5-4)/20 = 0.05 to *t*. Assuming that the path continued in a straight line at a rate of 20 inches/*t* at that point, we would be at the perfect *t*. Because paths are curves, we must iterate this process to increase accuracy. After five iterations, the *t* is accurate enough to be used, and we plug it into *c(t)* to get the lookahead point.

All I need to account for still is if the lookahead distance is greater than the distance left in the spline. I added a case for this scenario: if distance traveled plus lookahead is greater than spline length, I can "extend" the path in a straight line beyond the end pose with the same angle as the end pose.



```java
public Point2D getLookahead(double distanceTraveled, double lookahead) {
    if(distanceTraveled + lookahead > parametric.getLength()) {
        //if distance traveled is greater than the spline, return the corresponding point
        //along the straight line continuation from the last point on the spline
        Angle angle = parametric.getAngle( t: 1);
        Point2D endpoint = parametric.getPoint( t: 1);
        double distanceLeft = distanceTraveled + lookahead - parametric.getLength();
        return new Point2D( x: endpoint.getX() + distanceLeft * angle.cos(), y: endpoint.getY() + distanceLeft * angle.sin());
    } else {
        return parametric.getPoint(parametric.getTFromLength(distanceTraveled + lookahead));
    }
}
```

6. Motion Profile

The trapezoidal motion profile is easy to implement. To limit acceleration and velocity, the upper bound of the current velocity is:

$$vel_{current} = min(vel_{previous} + accel_{max} \cdot dt,\ vel_{max})$$

where dt is time passed since the last iteration.

To limit deceleration, I can get the distance to the end with Gaussian quadrature $length(t_{current}, 1)$ and then use the physics equation $v_f^2 = v_i^2 + 2ad$. With $decel_{max}$ as the positive maximum deceleration, I can solve for the maximum possible current velocity v_i given v_f:

$$vel_{current} = min(vel_{current}, \sqrt{v_f^2 + 2 \cdot decel_{max} \cdot dist_{end}})$$

This concludes the trapezoidal motion profile. However, I also need to limit angular velocity (a fast, heavy robot should not be turning rapidly). Using *l=wr*, solving for the maximum possible *l*, we get $l_{max} = w_{max} \cdot r$. To get *r*, I used the fact that the curvature *k* of a spline is equal to 1/*r*, where *r* is the radius of the circle that approximates the curve exactly at that point (ideally the arc that will be followed there).

$$k = \frac{x'y'' - y'x''}{\left(x'^2 + y'^2\right)^{\frac{3}{2}}}$$

for a parametric *c(t)=<x(t), y(t)>* from Wikipedia.
Then, the maximum linear velocity at a point is $w_{max} \cdot (1/k)$ where *k* is the curvature at that point.

To make sure that the robot decelerates so that it is always under the maximum angular velocity, I again utilize the 20ms cycle time. Every cycle, I use the previous equation $v_f^2 = v_i^2 + 2ad$ but with v_i as the current velocity and $v_f = 0$ to see the minimum distance the robot needs to stop completely:

$dist_{min} = v_i^2/(2 * decel_{max})$.

$d_i = dist_{closest} + dist_{min}$ (total distance to calculated point)

This makes sure that the robot always has time to decelerate, even as l_{max} at d_i approaches 0.
I can get l_{max} at d_i by using the length-to-t algorithm from earlier:

$l_{max} = w_{max} \cdot (1/k(t))$ where $length(0, t) = d_i$

I add this value of l_{max} to an array of $(l_{max,i}, d_i)$, removing any old values from the array where d_i is less than current distance traveled (i.e. the robot has passed that point). For every element in the array, I again apply $v_f^2 = v_i^2 + 2ad$ to make sure the robot has time to decelerate.

$$vel_{current} = min(vel_{current}, \sqrt{l_{max,i}^2 + 2 \cdot decel_{max} \cdot (d_i - dist_{closest}})$$

for all $(l_{max,i}, d_i)$.

7. Combined Algorithm

After creating all the necessary functions, creating the algorithm is easy. Every 20ms cycle, I can get the closest point on the spline from the current robot pose (updated through odometry).

For Pure Pursuit, I can get the lookahead using this point, and I pass the robot pose and lookahead point to the controller, which returns the Pure Pursuit radius r. I then pass $(vel_{current}, r)$, where $vel_{current}$ is the desired velocity from the motion controller, to differential drive kinematics.

For Ramsete, I pass the robot pose and closest point directly to the controller, along with the current desired velocity $vel_{current}$ and current desired angular velocity $w = vel_{current} / r = vel_{current} \cdot k$. It returns $(vel_{Ramsete}, w_{Ramsete})$ which can be passed to differential drive kinematics.

Using velocity *PID* with *feed forward*, differential drive kinematics moves the two powered wheels of the robot at the desired velocities. Odometry tracks the movement, and the algorithm updates accordingly the next cycle. I end the path when the robot pose is within a certain threshold of the end pose.

PID: Proportional-integral-derivative controller, a type of feedback control loop that uses the error value between the current state and the setpoint (desired state).

Feed forward: a type of open-loop control. Here, I multiply the feed forward constant (percent output to motor / velocity), which can be measured, by the desired velocity, getting a percent output that approximately reaches the desired velocity. PID corrects for the remaining error. By decreasing the error that PID has to work with, the response speed of the algorithm is improved significantly.