Holonomic Following of Bezier Curves in FTC

Adam R

September 2023

1 Introduction

In FTC, one of the main challenges to teams is finding a way to navigate the game field autonomously. When using fully actuated/holonomic drivetrains, it is possible to move in any direction that is desired while also moving on other axes, making the challenge of field navigation much easier. However, the question of how to generate paths and follow them accurately stands in the way of this.

The first question is mainly one of efficiency. While a straight line often is the fastest way to reach a point, when we introduce obstacles into the environment, we often have to stop at the end of the line before moving again. Curved paths, however, do not have this issue, and fit better in the FTC environment. The second question is substantially more complex, and this paper aims to offer a possible solution to it.

2 Bézier Curves

In order to generate and follow a curved path, we first need a curve. Bézier curves are commonly found in graphic and industrial design and are well documented. These parametric curves are well-suited for planning the movement of a robot and allow for easy tuning and generation.

In order to create a Bézier curve, we first need this formula, which defines an interpolation between 2 points, where t is a value between 0 and 1, and P_0 and P_{n+1} are points[1]:

$$L(t) = (1-t)P_n + tP_{n+1}$$

Next, we need to understand control points. A Bézier curve is defined by n control points that tell the interpolation equations what to do. We will be using Bézier curves with 4 control points (cubic Béziers) for this example. In order to generate the cubic curve, we will need 3 levels of interpolation. First, we interpolate between the control points. Here, $L0$ is the interpolation between points 0 and 1, L1 between points 1 and 2, and L2 between 2 and 3[2].

$$\vec{L_0}(t) = (1-t)P_0 + tP_1, L_1(t) = (1-t)P_1 + tP_2, L_2(t) = (1-t)P_2 + tP_3$$

Then, we interpolate between the points generated by the above functions:

$$\vec{Q_0}(t) = (1-t)L_0(t) + tL_1(t), Q_1(t) = (1-t)L_1(t) + tL_2(t)$$

Lastly, we interpolate between those newly generated points with this final equation:

$$\vec{C_0}(t) = (1-t)Q_0(t) + tQ_1(t)$$

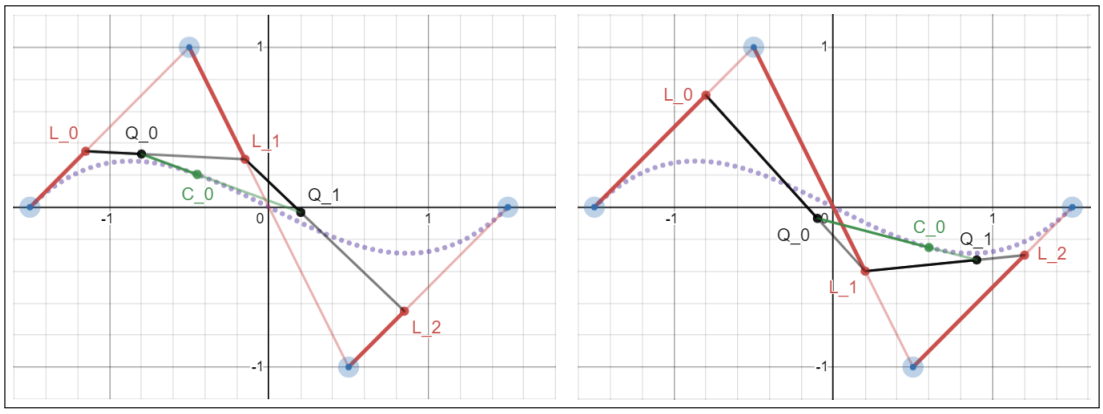

Figure 1: A 4 control point Bézier at $t = 0.35$ and $t = 0.7$.

The long series of equations shown previously can be simplified to this one-line formula:

$$\vec{B(t)} = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

In order to get x and y coordinates, we would simply rewrite this formula using the x and y coordinates of the points in the original function:

$$x(t) = (1-t)^3 x_0 + 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 x_3, y(t) = (1-t)^3 y_0 + 3(1-t)^2 t y_1 + 3(1-t)t^2 y_2 + t^3 y_3$$

We can combine the output of these functions to create the curve like this:

$$\vec{B(t)} = (x(t), y(t)) \quad 0 \leq t \leq 1$$

2.1 Working With the Curves

Before we can begin to follow a Bézier curve, we will need to gather some information about it. We need the velocity (tangent) vector of the curve, the length of the curve, and the tangent angle.

2.2 Velocity Vector and Derivatives

The velocity/tangent vector is quite possibly the most useful piece of information we can glean from the curve. We will use it for calculating length, finding our target heading, and calculating our translational velocities. The velocity vector is defined by the following

$$\vec{B'(t)} = (x'(t), y'(t))$$

where $x'(t)$ and $y'(t)$ are the derivatives of $x(t)$ and $y(t)$. This vector can be thought of as the rate of change in the positon of the curve at a step t, or literally as a velocity vector. Both interpretations will be useful to us, so remember them both for now.

[1] This formula is parametric, meaning you need a version of the formula taking the x and y coordinates of P_0 and P_{n+1} to get a point.

[2] When labeling Bézier control points, we start counting at 0, meaning the start points is p0, the second point is p1, and so on.

2.2.1 Length

The other piece of information we need to be able to follow the curve is its length. Since, of course, this is a curve and not a line, we cannot simply measure the distance from the start point to the endpoint. We must use the formula for the arc length of parametric[3] curves to do this:

$$l = \int_0^1 \sqrt{(x'(t))^2 + (y'(t))^2}\, dt$$

At first, this looks a bit daunting. However, if we use the rate of change interpretation of the velocity vector, the workings of it become clear. This integral just applies the distance formula to the change in x and the change in y of the curve in order to find the length of a very small step. Summing all of these steps gives us the total length of the curve.

2.3 Tangent Angle and Heading

It may also be of use to us to know the tangent angle of the curve at a given point in the event that we want to end up facing the same angle as the curve when we finish following. This can be done by applying the arc tangent function to the tangent vector like so:

$$\arctan \frac{y'(t)}{x'(t)}$$

3 Pose

The pose is an important prerequisite to following the curve. A pose describes the position of our robot in 3D space as a 3x1 matrix or as a 3-dimensional point.

$$P = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix},\ m = (x, y, \theta)$$

It is also important to note that there are 2 main reference frames for our robot's pose: local and global. The local frame seese pose changes relative to the robot, while the global frame sees poses relative to the field. We can convert between these frames with the rotation matrix

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

With this, we can rotate any global information to local, and when we want to apply our local changes, use it again to go back.

4 Motion Profiling

Now that we have all the necessary information from the curves, we can move on to the next key step in the following process: motion profiling. Motion profiles allow us to find a desired displacement, velocity, and acceleration at a given time step. This is useful in computing our directional velocities, indexing the curve, and computing our motor power commands.

[3] As noted earlier, Béziers are parametric curves.

4.1 Parameters and Time

For our purposes, we will be making use of the trapezoidal motion profile. This profile takes the max acceleration rate, max velocity, and desired displacement, and outputs a set of equations that we can index to gather information about our motion. The first thing we need to do to create our

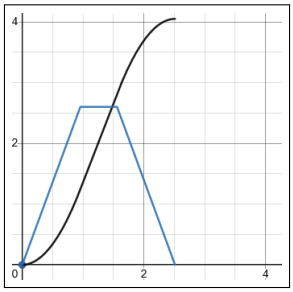

Figure 2: A sample trapezoidal motion profile. Here, the x-axis is time. The y-axis of the blue graph is velocity and the y-axis of the black graph is displacement

profile is calculate how long we will it will take us to reach the desired distance. First, we must check if we will reach our maximum velocity during the profile, or if the distance is too short and we will decelerate before max velocity. This can simply be done by calculating the distance would travel in the acceleration and deceleration phases if we did hit max velocity, $\frac{v^2}{a}$, and subtracting it from the total distance. If the difference is 0 or negative, we will not reach max velocity. If it is nonzero and positive, we will reach max velocity. Then, we can compute time as follows:

$$t = \begin{array}{ll} \frac{2v}{a} + \frac{d - \frac{v^2}{a}}{v} & \text{if we reach max velocity} \\ 2\sqrt{\frac{d}{a}} & \text{if we do not reach max velocity} \end{array}$$

4.2 Profile Equations

After we have computed time, it is necessary to find the time we will spend accelerating/decelerating (since this is a trapezoid profile, the time spent in each will be equal) and the time at which we begin decelerating. These time steps can be represented as t_a and t_d respectively.

$$t_a = \min(\frac{t}{2}, \frac{v_x}{a}),\ t_d = t - t_a$$

Now that we have our constants and time values, we can create the equations for our profile. Here, v is the velocity reference, p is the displacement reference, and a is the acceleration reference:

$$v(x) = \begin{array}{ll} ax & 0 \le x \le t_a \\ a(t_a) & t_a \le x \le t_d \\ a(t_a) - a(x - t_d) & t_d \le x \le t \\ 0 & t \le x \end{array} , p(x) = \begin{array}{ll} \frac{a(x)^2}{2} & 0 \le x \le t_a \\ \frac{a(t_a)^2}{2} + (x - t_a)a(t_a) & t_a \le x \le t_d \\ \frac{a(t_a)^2}{2} + (t_d - t_a)a(t_a) + \frac{t_a(a(t_a)) - a(t-x)}{2} & t_d \le x \le t \\ d & t \le x \end{array}$$

$$a(x) = \begin{array}{ll} a & 0 \le x \le t_a \\ 0 & t_a \le x \le t_d \\ -a & t_d \le x \le t \\ 0 & t \le x \end{array}$$

The first set of equations defines the trapezoid itself and gives us the velocity reference. The second set integrates the first and gives us displacement. The third gives us our current rate of acceleration.

5 Indexing

In order to, gather all the information we need to follow our curve, we first need to perform an index.

5.1 Prep

It is first necessary to create 2 motion profiles: a translation profile based on the length of the curve, and an angle profile based on an arbitrary angle. The angle profile can effectively be ignored until the end of the entire process, but we will tackle it first here.

5.1.1 Angle Profile

First, we take the desired end angle and subtract from it our starting angle. If we wish to end facing the same direction as the curve, we can use the arctangent method described earlier to do so. It is also necessary to normalize this angle to the [180, -180] range so that the robot takes a more efficient turning path. Regardless of what angle we choose to use, we will input the difference as the d value in our profile, and our maximum angular velocity and acceleration as v and a respectively. Let's define this profile as

$$v_\theta(x), p_\theta(x), a_\theta(x)$$

Now, we can put this to the side and deal with translation.

5.2 Translation Profile

We also need to create our translational profile. This is quite possibly the most important thing we will use in the follower. If you recall section 2.2.1, you remember that the length of a Bézier is given by this equation:

$$l = \int_0^1 \sqrt{(x'(t))^2 + (y'(t))^2}\, dt$$

We can create our profile based on l so that our target displacement is the length of the curve. Then, we simply need to give the profile our maximum translational velocity and acceleration and the equations will be satisfied. Let's define this profile as

$$v_t(x), p_t(x), a_t(x)$$

5.3 Completing the Index

Now that we have all the preparation out of the way, we can perform an index of the curve. First, we need to obtain the real-world time step, t_i. Usually, this is from some form of timer that starts at the beginning of the following. We plug this number into our motion profile equations to get a "state" with a target velocity, displacement, and acceleration. The next step is to somehow get a t value for our curve. This can be done like so:

$$d_c = p(t_i),\ t = \frac{d_c}{l}$$

As stated in section 3.2, $p(t)$ gives our current displacement. Since $p(t)$ has a range of $(0, l)$, dividing it by l gives us a t value between 0 and 1.

5.4 Target Point, Velocity Vector, and Normalization

Next, we need to index the curve itself. First, we plug the t value into the curve equations to get the point $p_t = (x(t), y(t))$. Then, we acquire the velocity vector $\vec{v} = (x'(t), y'(t))$. Finally, we need to normalize the velocity vector to acquire the directional unit vector. To do this, we divide the vector by its magnitude.

$$\vec{u} = \frac{\vec{v}}{||\vec{v}||}$$

6 Follwing the Curve

We have arrived at the final step: finding the desired directional velocities, getting positional feedback, and applying it all to the robot.

6.1 Velocity

The first step in controlling our velocity is scaling the directional unit vector at our current step by the current desired linear velocity given by the translation profile.

$$\vec{s} = v_t(t_i) * \vec{u},\ v_x = \vec{s} \text{ x component},\ v_y = \vec{s} \text{ y component}$$

We also grab the desired acceleration from the profile.

$$a = a_t(x)$$

Then we acquire the desired angular velocity and acceleration from the angle profile we created in section 5.1.1.

$$v_z = v_\theta(t_i), a_z = a_\theta(t_i)$$

Now we have all of our desired velocities, but they still are in a form unusable to our drive motors. To convert them to a usable power format, we will use a feedforward model, where $k_v, k_a,\ \&\ k_s$ are tuned constants:

$$f_n = k_v * v_n + k_a * a_n + k_s$$

We can apply this to each velocity to get

$$f_x = k_v * v_x + k_a * a + k_s,\ f_y = k_v * v_y + k_a * a + k_s,\ f_\theta = k_v * v_z + k_a * a_z + k_s$$

6.2 Feedback

In the event that we get knocked off of our path, we want to have some form of controller to correct it. For our purposes, a simple PID controller will suffice. A PID loop is usually better defined in code, so I will define it in that form here.

```
Kp = num, kI = otherNum, Kd = otherOtherNum;

integralSum = 0, lastError = 0;
timer = someFormOfATimer;

double output(error) {
    derivative = (error - lastError) / timer.seconds();
    integralSum = integralSum + (error * timer.seconds());

    lastError = error;
    timer.reset();
    return (Kp * error) + (Ki * integralSum) + (Kd * derivative);
}
```

In order to use this controller, we need some form of error to pass in. In this case, we want feedback on our position, so we need our pose error. We need our pose error in the local frame as well, since the robot movement calculations are local. First, we need to obtain our desired point and angle as a matrix, which we can do by simply taking p_t and $p_\theta(t_i)$ and constructing a 3x1 matrix from them. The final pre-calculation step is to get our global pose, P_G, where $x_g, y_g,$ & θ_g are our global coordinates. Then we can calculate the error as:

$$E = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} p_x \\ p_y \\ p_\theta(t_i) \end{bmatrix} - \begin{bmatrix} x_g \\ y_g \\ \theta_g \end{bmatrix} \right)$$

We can then pass this on to our controller to get the following output:

$$o_x = \text{output}(E_{1,1}),\ o_y = \text{output}(E_{2,1}),\ o_\theta = \text{output}(E_{3,1})$$

6.3 Combining the Outputs

In the event that we are using a mecanum drive, a relatively common drive in FTC, breaking everything down to powers is relatively easy. The drive has 4 powered wheels that each create their own vector, as shown in the figure.

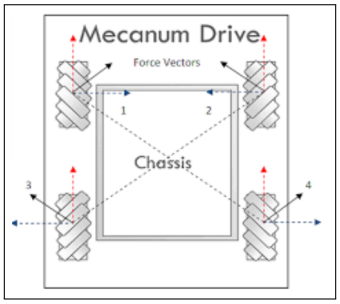

Figure 3: The force vectors of a Mecanum Drive

First, we simply sum the velocity commands and feedback commands.

$$C = \begin{bmatrix} c_x \\ c_y \\ c_\theta \end{bmatrix} = \begin{bmatrix} o_x \\ o_y \\ o_\theta(t_i) \end{bmatrix} + \begin{bmatrix} f_x \\ f_y \\ f_\theta \end{bmatrix}$$

Once we have our command matrix, we can use the values it contains to set the power to our wheels. Our wheels are defined by their positions: f_l is the front left wheel, b_r is the back right wheel, and so on. Usually, in order to get the proper y movement, you would reverse the y power before the equations, but here we will modify the equations themselves so that pre-modification is not necessary.

$$f_l = (-c_y + c_x + c_\theta),\ b_r = -c_y + c_x - c_\theta$$
$$f_r = (c_y - c_x - c_\theta),\ b_l = c_y - c_x + c_\theta$$

Then, since FTC motors only accept a power range of [-1,1], we need to create a scale factor to fit the power commands within the range.

$$s_f = \max(|c_x| + |c_y| + |c_\theta|, 1)$$

Now we just divide all the powers by the scale factor s_f and apply them to our motors.

7 Conclusion

Now we have covered the whole process of following a Bézier curve. We started with generation and $\vec{B(t)}$, found the velocity vector and length of the curve, created motion profiles to find references at real-world time steps, and broke it all down to usable motor powers with feedforward models and PID controllers. Special thanks to Arnav and Sriram for making this possible for me and helping me figure out new math every step of the way. If you have any questions or feedback for this paper, feel free to email me at 310areed@gmail.com or message me on Discord @*doublebubble*. Thank you for reading, and good luck on your following journey.