TRISONICS
4003

2018 Controls: MOTION PROFILING

FIRST®
POWER UP℠

TriSonics, FRC 4003
Motion Profiling Whitepaper

A common problem in autonomous robot operation is getting the robot to follow a potentially complicated path accurately and dependably. Many factors, such as wheel slippage and variations in battery voltage, make this a difficult problem.

This paper outlines our solution to this problem using a technique called *motion profiling*, which is being more frequently used in FRC robots. For instance, the Hall of Fame team, Cheesy Poofs, FRC 254, gave a standing-room only presentation at the 2015 World Championships (`https://www.youtube.com/watch?v=8319J1BEHwM`) on this topic. The Talon SRX motor controllers recently added a "motion profiling" mode with plenty of documentation. The Chief Delphi user "Jaci" also provides a software package called PathFinder to help teams implement motion profiling.

Given these resources and the fact that the 2018 FRC game, Power Up, requires autonomous robots to navigate across a large part of the competition field, we thought this was a good opportunity to learn some more advanced control theory, and we'll share what we've learned here. In spite of there being free resources provided for teams, we wanted to do it ourselves from scratch because it seemed like a good learning opportunity, and we wanted to understand what was happening better so that we could debug any problems.

In rough outline, motion profiling involves two big pieces. First, we have to create a path or trajectory in a form that a robot can understand. Second, we need to tell the robot how to traverse that path.

Here is a very crude implementation of motion profiling. First, create a trajectory by placing the robot on the field and pushing it along your desired path while sampling the encoders on the left and right drive motors at some regular interval. Second, devise a means to "play back" these encoder readings; that is, use these readings to determine power settings that will reproduce the encoder readings.

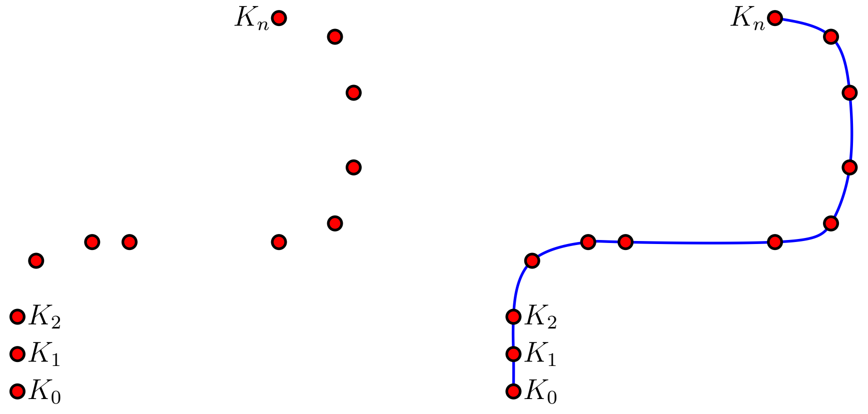
This is quite similar to what we will do. However, rather than pushing the robot along a trajectory to generate the encoder readings, we will create a mathematical algorithm that does it for us. The main idea is this:

Reliable robot performance follows from smoothness of the profile.

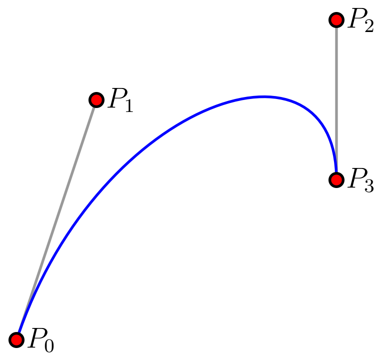
For instance, asking the robot to turn a sharp corner or to suddenly accelerate are tasks that are more difficult to perform reliably. We want to create a smooth path for the robot to move along and have the robot accelerate smoothly along this path.

Trajectory generation

We will first describe a technique for generating the path that the robot will move along by specifying a certain number of "waypoints." For instance, the figure on the left below shows a number of points that are labeled K_0, K_1, and so on. We would like to be able to generate a smooth path that passes through these points, as shown on the right.



There are several ways to tackle this problem, and none of them are perfect; we will outline one of them here. Between each pair of points, we will move along a cubic Bezier curve. Cubic Bezier curves are defined by four "control points," which are often denoted P_0, P_1, P_2, and P_3.



The curve begins at P_0 and ends at P_3. The relationship between P_1 and P_0 defines the initial velocity, and the relationship between P_2 and P_3 defines the final velocity. In fact, we may write the Bezier curve as

$$B(x) = (1-x)^3 P_0 + 3x(1-x)^2 P_1 + 3x^2(1-x) P_2 + x^3 P_3$$

where $0 \leq x \leq 1$. Notice that

$$B(0) = P_0 \qquad \text{and} \qquad B(1) = P_1.$$

Differentiating, we find that

$$\begin{aligned} B'(x) &= -3(1-x)^2 P_0 + (3 - 12x + 9x^2) P_1 + (6x - 9x^2) P_2 + 3x^2 P_3 \\ B''(x) &= 6(1-x) P_0 + (-12 + 18x) P_1 + (6 - 18x) P_2 + 6x P_3 \end{aligned}$$

If we evaluate the first derivative at the endpoints, we find that

$$\begin{aligned} B'(0) &= 3(P_1 - P_0) \\ B'(1) &= 3(P_3 - P_2). \end{aligned}$$

This shows how the control points P_1 and P_2 determine the velocities at the two endpoints.

Given the sequence of $n+1$ points $K_0, K_1, \ldots, K_n$, we will find n Bezier curves that form a smooth trajectory between them. We will call these curves $B_0, B_1, \ldots, B_{n-1}$ and require that

$$\begin{aligned} B_i(0) &= K_i \\ B_i(1) &= K_{i+1} \\ B_i'(1) &= B_{i+1}'(0) \\ B_i''(1) &= B_{i+1}''(0) \end{aligned}$$

In other words, each curve will begin at a waypoint and end at the next waypoint. In addition, the velocities and accelerations of the two curves that meet at a waypoint will be the same. In this way, we can create a smooth trajectory.

These equations will help us determine the control points for each curve. For the curve $B_i(x)$, we denote its control points by P_0^i, P_1^i, P_2^i, and P_3^i. The first two equations above say that

$$\begin{aligned} B_i(0) &= P_0^i = K_i \\ B_i(1) &= P_3^i = K_{i+1}. \end{aligned}$$

This says that half of the control points, namely the P_0^i and P_3^i, are determined by the waypoints.

We only need to determine the control points P_1^i and P_2^i. Here we use the requirement that the incoming and outgoing velocities and accelerations at a waypoint equal one another. Using the equations for the derivatives we found above, this gives

$$\begin{aligned} 3(P_3^i - P_2^i) &= 3(P_1^{i+1} - P_0^{i+1}) \\ 6P_1^i - 12P_2^i + 6P_3^i &= 6P_0^{i+1} - 12P_1^{i+1} + 6P_2^{i+1} \end{aligned}$$

Without too much work, we can rearrange these equations as

$$\begin{aligned} P_1^{i+1} + P_2^i &= 2K_{i+1} \\ P_1^i - 2P_2^i + 2P_1^{i+1} - P_2^{i+1} &= 0 \end{aligned}$$

The problem is that there are only $2(n-1)$ equations (two for each interior waypoint) to determine $2n$ unknowns. We need to add two more equations, which we do by requiring the acceleration at the beginning and ending waypoints to be zero:

$$\begin{aligned} B_0''(0) &= 6P_0^0 - 12P_1^0 - 12P_2^0 = 0 \\ B_{n-1}''(1) &= 6P_1^{n-1} - 12P_2^{n-1} + 6P_3^{n-1} = 0 \end{aligned}$$

We now have $2n$ equations in $2n$ unknowns. If we do a bit of algebra, we can separate the equations involving P_1 from those involving P_2 and find that

$$\begin{aligned} 2P_1^0 + P_1^1 &= K_0 + 2K_1 \\ P_1^{i-1} + 4P_1^i + P_1^{i+1} &= 4K_i + 2K_{i+1} \qquad i = 1, 2, \ldots, n-2 \\ 2P_1^{n-2} + 7P_1^{n-1} &= 8K_{n-1} + K_n. \end{aligned}$$

To find the P_1, we need to solve an $n \times n$ system of equations, but this is pretty easy because it's tridiagonal; that is, the coefficient matrix looks like

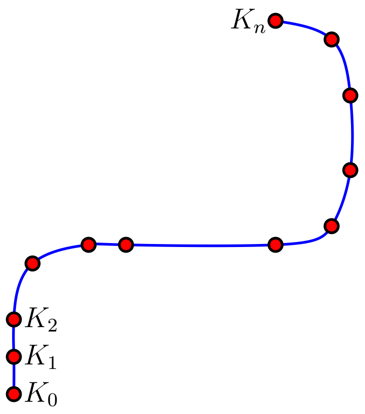
$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & 4 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 4 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & 4 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 2 & 7 \end{bmatrix}.$$

This can be solved efficiently using Thomas' algorithm, which produces the control points P_1^i.

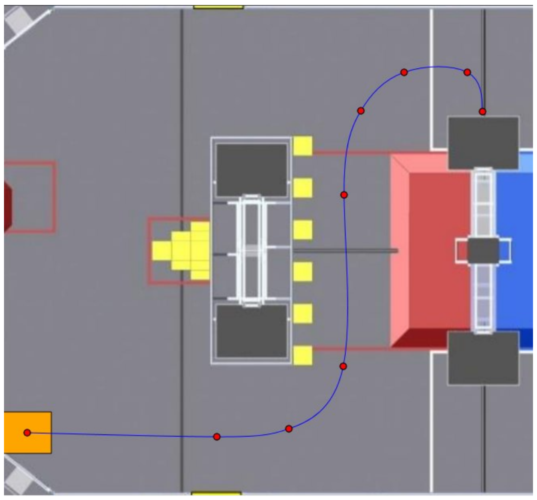
The equations for P_2^i are relatively straightforward once we know the P_1^i:

$$\begin{aligned} P_2^i &= 2K_{i+1} - P_1^{i+1} \qquad i = 0, 1, \dots, n-2 \\ P_2^{n-1} &= \frac{1}{2}(K_n + P_1^{n-1}) \end{aligned}$$

At this point, we know all the control points so we completely know the trajectory on which the robot will move.



We created a Java program that allows us to move the waypoints and see the resulting trajectory. This allows us to easily design trajectories to accomplish various autonomous actions. Once the trajectory has been determined, we save the waypoints in a `.csv` file.



Velocity profiles

If we just turn full power to the robot on and off, we can see that the robot's motion is not very reliable. It will usually jerk at the beginning and end of its motion. This means that, now that we have smooth trajectories, we want to move along them with a smoothly changing velocity.

We can create smooth velocity profiles using some ideas from calculus. We will be periodically updating the power settings on the drive train at regular time intervals. We will update every $\Delta t = 10$ milliseconds so we would like to determine our robot's velocity every Δt milliseconds.

Calculus tells us how to update the position, velocity, and acceleration. At some time, our position is s, our velocity is v, our acceleration is a, and the jerk (rate of change of the acceleration) is j. Then we update at the next time with

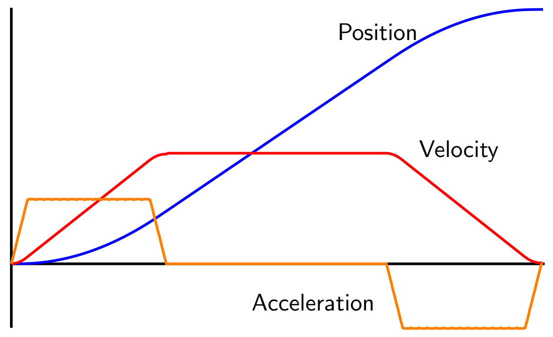
$$\begin{aligned} s &= s + v\Delta t \\ v &= v + a\Delta t \\ a &= a + j\Delta t \end{aligned}$$

We did some experiments with our robot by applying maximum power to the drive train to measure its maximum velocity v_{max}, its maximum acceleration a_{max}, and its maximum jerk j_{max}.

To begin, suppose that our velocity is 0 and that we would like to accelerate up to v_{max}. We start with $s = 0$, $v = 0$, and $a = 0$. To accelerate, we set the jerk $j = j_{max}$, the largest possible jerk. This gives us the following table of values, assuming $\Delta t = 10$, which can be continued as long as we need.

t	s	v	a	j
0	0	0	0	j_{max}
10	0	0	$10j_{max}$	j_{max}
20	0	$100j_{max}$	$20j_{max}$	j_{max}
30	$1000j_{max}$	$300j_{max}$	$30j_{max}$	j_{max}

Once the acceleration reaches a_{max}, we set the jerk $j = 0$ so that the acceleration continues to be a_{max}. Once the velocity is near its maximum v_{max}, we set $j = -j_{max}$, which causes the acceleration to decrease to zero and the velocity to smoothly arrive at v_{max}. Here is the resulting graph of position, velocity, and acceleration.

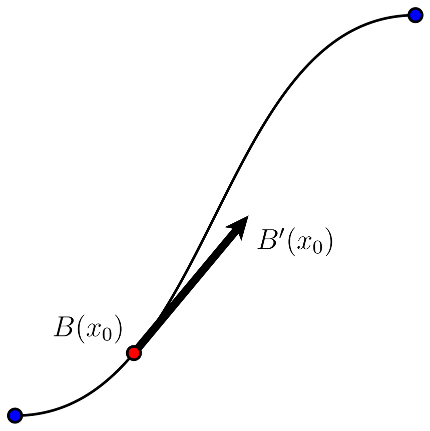


Putting it all together

We are now able to make a smooth trajectory and a smooth velocity profile. Next, we have to put everything together by determining where we are on the trajectory at any given time.

To do this, we know that our trajectory is built from Bezier curves so at any instant, we are moving along a Bezier curve $B(x)$. Suppose that we are at the point $B(x_0)$ and moving with velocity v, which is determined by our velocity profile. We need to update the position on the Bezier curve to find $B(x_1)$ after Δt milliseconds have passed.

Knowing the Bezier curve $B(x)$, we can compute its derivative $B'(x)$ and its length $|B'(x)|$. In this way, we are able to keep track of where we are on the Bezier curve at any time.



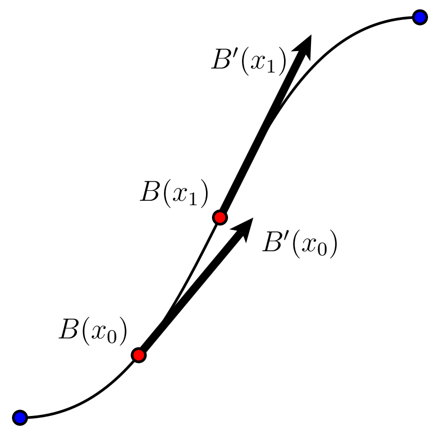
The distance traveled during the time interval is $v\Delta t = |B'(x_0)|\Delta x$, which gives

$$\Delta x = \frac{v\Delta t}{|B'(x_0)|}$$

so that

$$x_1 = x_0 + \frac{v\Delta t}{|B'(x_0)|}.$$

This is useful because we can keep track of our heading at every time and measure the amount that the robot has rotated over each time interval. This tells us the robot's angular velocity ω at every time.



Our last step is to determine the velocities and positions for the left and right drive motors, which we will call v_l and v_r. We know the robot's velocity v and its angular velocity ω at every time.

Once we had the robot, we measured its wheelbase, which is the distance between the left and right wheels. We did this in two ways: once with a tape measure and once by spinning the robot in place and looking at the encoder readings on the drive motors. In this way, we found that the distance was $2r = 25.5$", and we can use r as the radius of the robot as it rotates. If we call the differential in the the right and left velocities

$$v_{\text{diff}} = v_r - v_l,$$

then we know that v_{diff} is related to the angular velocity ω by

$$v_{\text{diff}} = r\omega.$$

We then set

$$v_l = v - v_{\text{diff}}$$
$$v_r = v + v_{\text{diff}}$$

to determine the velocities of the left and right drive motors at every time. If s_l and s_r are the positions of the left and right drive motors, we update these positions by

$$s_l = s_l + v_l \Delta t$$
$$s_r = s_r + v_r \Delta t.$$

We now save all this information in a profile `.csv` file with five entries per line, which are left position, left velocity, right position, right velocity, and heading. Each row corresponds to one instance in time. A typical portion of a profile `.csv` file looks like this:

```
206.3249,0.5758,189.7543,0.2882,-38.44
206.8938,0.5620,190.0494,0.3020,-39.09
207.4663,0.5832,190.3409,0.2808,-39.76
```

```
208.0420,0.5682,190.6292,0.2958,-40.45
208.6204,0.5884,190.9148,0.2756,-41.14
209.2009,0.5726,191.1983,0.2914,-41.85
209.7831,0.5919,191.4801,0.2721,-42.57
210.3667,0.5752,191.7605,0.2888,-43.29
210.9512,0.5939,192.0400,0.2701,-44.02
```

We put all this information together with a Python program we created. This program takes the waypoints and velocity information and generates the profile `.csv` file using the approach described above.

Controlling the robot

Finally, we need to use all this information to control the robot. We first store all the `.csv` files on the Roborio so they can be read into the robot project when needed. To execute the profile, we use a `Notifier` that fires every $\Delta t = 10$ milliseconds and processes one line of our `.csv` file.

At every time, we know the left and right positions and velocities s_l, v_l, s_r, and v_r, and θ, the heading.

We know that full power corresponds to a velocity of v_{max} so our first guess for the left and right powers P_l and P_r is

$$P_l = \frac{v_l}{v_{max}}$$
$$P_r = \frac{v_r}{v_{max}}.$$

This is called a feed-forward term. For instance, if we want to travel at $0.5v_{max}$, we set the power to 0.5. We are assuming that there is a direct proportionality between the power and the velocity, which is not usually true. We can use feedback, however, to correct the error.

We know that the expected position on the left is s_l and we can read the actual position e_l from the left encoder. The error is $s_l - e_l$. If $s_l - e_l > 0$, then we are running behind where we should be so we need to increase the power. We use a constant K_p, which is really the P term of a PID controller, and set

$$P_l = \frac{v_l}{v_{max}} + K_P(s_l - e_l)$$
$$P_r = \frac{v_r}{v_{max}} + K_P(s_r - e_r).$$

In fact, we have more information from the heading. We can compare the desired heading to the observed heading read from a gyro and measure an angular error $\Delta\theta$. We use another proportional controller and define a constant K_θ so that we have, at last,

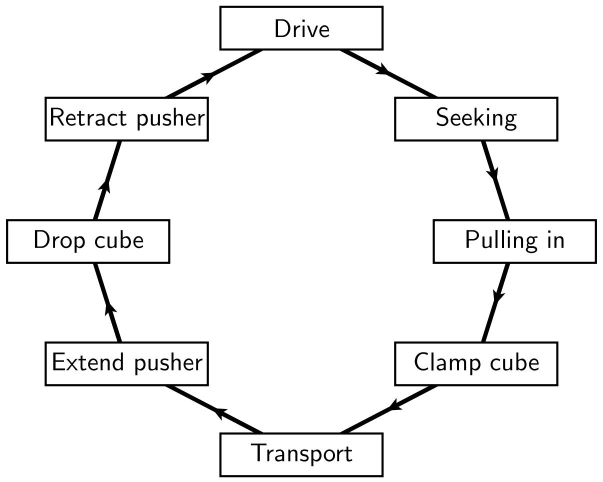
$$P_l = \frac{v_l}{v_{max}} + K_P(s_l - e_l) - K_\theta\Delta\theta$$
$$P_r = \frac{v_r}{v_{max}} + K_P(s_r - e_r) + K_\theta\Delta\theta.$$

We have found this approach to produce very reliable autonomous trajectories that can be easily implemented and maintained. To create this implementation, we have followed some of the available resources, but we have written all our own code so that we understand the ideas better.

Additional controls

State machine

The subsystem we use to manipulate cubes has several parts whose actions need to be coordinated. We use a state machine to make sure that everything stays in sync. The state machine looks like this:



Here are the details of the states:

Drive The intake is in, rollers are in, clamp is closed, and pusher retracted. The intake motors are off.

Seeking The intake is out, rollers are out, clamp is open, and pusher retracted. The intake motors are on.

Pulling in The intake is out, rollers are in, clamp is open, and pusher retracted. The intake motors are on.

Clamp cube The intake is in, rollers are in, clamp is closed, and pusher retracted. The intake motors are off.

Transport The intake is in, rollers are in, clamp is closed, and pusher retracted. The intake motors are off, and the lift is raised to driving height.

Extend pusher The intake is in, rollers are in, clamp is closed, and pusher extended. The intake motors are off.

Drop cube The intake is in, rollers are in, clamp is open, and pusher extended. The intake motors are off.

Retract pusher The intake is in, rollers are in, clamp is open, and pusher retracted. The intake motors are off.

The operator moves between these states with a button press. There are also button presses to return to the Seeking state and the Drive state.

Autonomous selector

We have had problems in the past with using the Smart Dashboard to select our autonomous mode because network tables sometimes does not transmit the data at the right time. Because of this, we use a DIP switch to choose our autonomous mode. We use nine switches:

1 Select whether we begin in the center position.

2 Select whether we begin in the left or right position.

3 Select whether we go to the scale or switch if the game data begins with "LL"

4 Select whether we go to the scale or switch if the game data begins with "LR"

5 Select whether we go to the scale or switch if the game data begins with "RL"

6 Select whether we go to the scale or switch if the game data begins with "RR"

7 Select whether we go to the left scale from the side or null zone in case we have a partner who is also going to the scale.

8 Select whether we go to the right scale from the side or null zone in case we have a partner who is also going to the scale.

9 Select whether to place second cube in switch or scale.