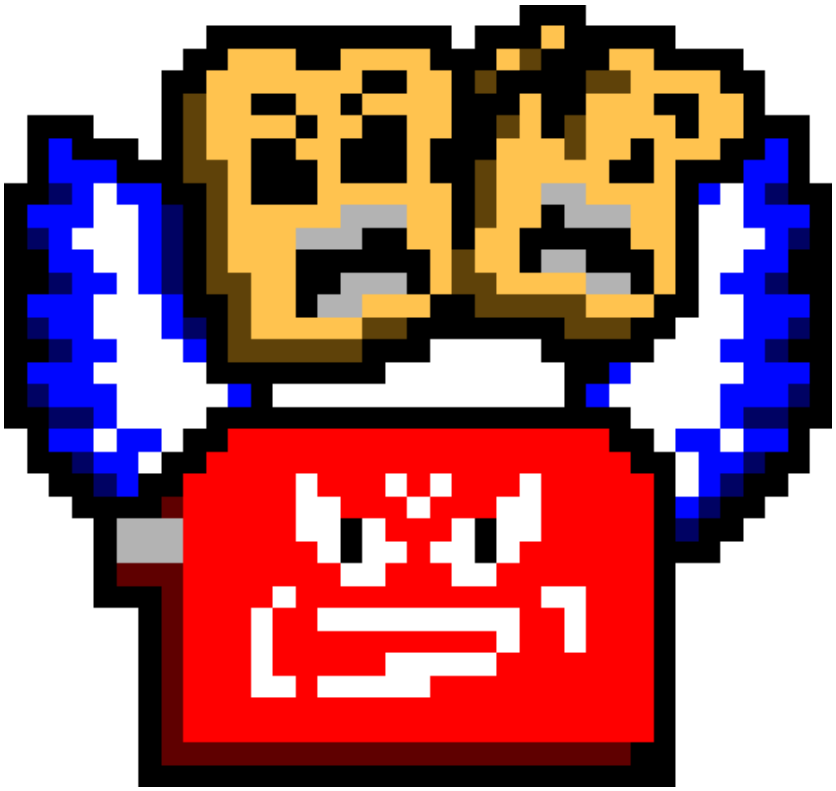
Motion Profile Generation & Following

Jack LeFevre
March 2018



Team 3641 - The Flying Toasters

Contents

Background

This paper's purpose is to describe the process to generate and follow motion profiles, which are smooth curved paths with acceleration/deceleration profiles that the robot can follow. This will allow a robot to drive consistently and quickly during autonomous routines.

Profile generation

There are two distinct steps to generating a motion profile: Generating the curve for the robot to follow, and calculating the robot's position, velocity, and acceleration along the curve.

Curve generation

The path of a motion profile is expressed as a series of points with angles. Since each path can have as many as a few hundred points, it is necessary to generate the paths programmatically. For this paper, I will describe how this is done with cubic Bezier curves, which are created from a start, an end, and two guide points. This type of curve creates a smooth and continuous path defined by a cubic function.

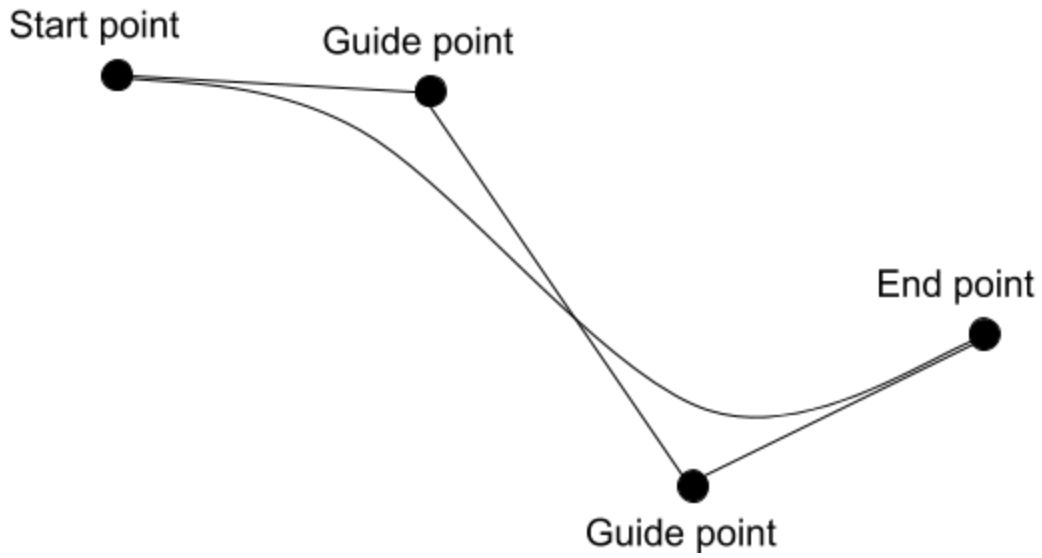


Fig. 1: An example of a cubic Bezier spline from a start, end, and two guide points.

There are two methods of generating such a curve. The relatively slow but easier to understand method involves linear interpolation between points. Linear interpolation finds a point in between two other points at any fraction of the way between them. Here is an example of a linear interpolation method:

```
static double lerp1D(double p1, double p2, double alpha){
      return alpha * (p2 - p1) + p1;
}
static Point lerp2D(Point p1, Point p2, double alpha){
      return new Point(lerp1D(p1.x, p2.x, alpha), lerp1D(p1.y, p2.y, alpha);
}
```

The points along the curve are defined by interpolating between each set of adjacent points to get three points instead of four, then repeating this process until only 1 point is left. Different ‘alpha’ values for the interpolation function produces different points along the curve. To calculate a curve, simply use many (We used 100) points evenly spaced from 0 to 1.

A faster (but equivalent) method is to use basis functions, which are functions multiplied by the curve points to find the position of each spline point.

For longer paths, multiple curves can be linked start to end. However, this causes a problem with Bezier curves because the derivative of the curvature does not match, and this causes problems later on. A solution to this problem is to use quintic Hermite splines that have acceleration set to 0 at each end instead. This will not be discussed in depth in this paper because we haven't finished implementing it yet.

See here for more information and algorithms regarding Bezier curves: https://pomax.github.io/bezierinfo/

Information about quintic Hermite splines: https://www.rose-hulman.edu/~finn/CCLI/Notes/day09.pdf

Velocity profiles

The profiles are tuned to a specific acceleration and maximum velocity, and the generated profile follows a trapezoidal velocity profile. This means the robot accelerates at a fixed acceleration until it reaches a maximum velocity, drives at that maximum velocity, then decelerates at the same acceleration until it reaches its endpoint at a stop.

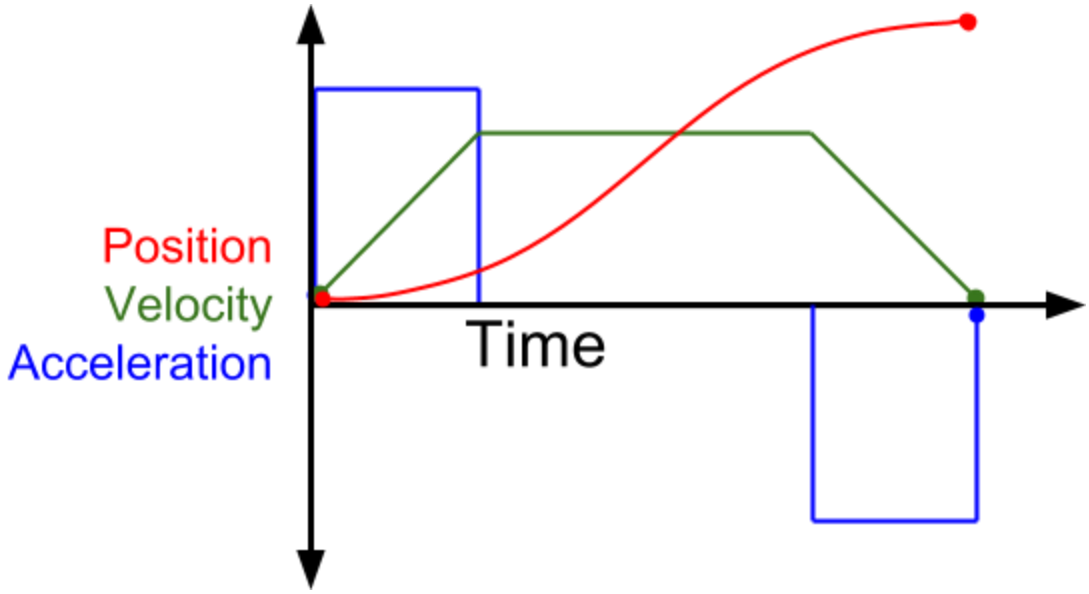


Fig. 2: A robot's position, velocity, and acceleration over time as it follows a trapezoidal velocity profile.

This is the simpler of the two popular types of velocity profiles. The other type is a jerk-minimizing profile, which is much more complex to compute. A jerk minimizing profile prevents the acceleration from changing instantaneously like in a trapezoidal profile. This creates slightly smoother profiles, but we found trapezoidal profiles to be *Good Enough™.*

Later in the season we did have to make one modification to the velocity profiles to account for the inside and outside wheels in a turn, which will be higher or lower than the velocity determined by the trapezoidal profile. When we tried to speed up our motion profiles to the maximum speed, we were severely limited by this. The exact algorithm to correct for this is discussed in the next section.

Velocity calculation

The following section was the most difficult part of this whole process to get right. Due to this, this section will be rather technical.

Calculating the velocity at each point in the motion profile requires knowing the distance of each point along the curve. There are a few fancy mathematical

techniques for this, but it's easiest to approximate it. The approximation technique we used was to calculate the distance from each point to the next and keep a running total. As long as each curve has enough points along it, this is precise enough. The distance from the end is needed as well, but this is fairly trivial to figure out by subtracting the distance of a point from the distance of the end point.

Now that the distance of each point has been determined, there are three possibilities to consider for the velocity of the robot at that point:

1. The robot is accelerating
2. The robot is at max speed
3. The robot is decelerating

If you are feeling confident in your calculus/physics skills, you can calculate the time at which the robot transitions between these states. At first, we tried to do this, but it is impossible when using turning velocity correction. The approach we used instead was to find the velocity assuming each of these cases were true and use the minimum of the three velocities.

Calculating acceleration/deceleration velocity:

We used the following kinematic equation to determine the velocity for the acceleration/deceleration case:

$$V_F^2 = V_I^2 + 2ad$$

Where V_F is the velocity at the given distance (What we're looking for), V_I is the initial velocity of the motion profile, a is the robot's maximum acceleration, and d is the distance from the starting or ending point. Solving this equation for V_F gives

$$V_F = \sqrt{V_I^2 + 2ad}$$

Which can be used to calculate the acceleration/deceleration velocity at each point.

Calculating maximum turning velocity:

If only the maximum velocity is used for velocity calculation, the wheel on the outside edge of a turn can exceed the robot's maximum velocity. This is undesirable when traveling fast is necessary.

To correct for this issue, the following steps are taken:

1. The angular velocity of the robot at the given point is multiplied by half of the distance between the wheels to calculate the velocity the wheels would be spinning if the robot was turning in place.
2. The absolute value of the rotational wheel velocity is added to the max robot velocity to find the estimated speed of the outside wheel.
3. The max speed for the turn is set to the max velocity squared divided by the estimated outside wheel velocity. This means that if the robot is not turning, the max velocity will not be affected, but if the robot is turning it will slow down.

Profile following

There are two steps required to follow the paths: Generating the motion profile for each wheel, and using feedback from sensors to follow the profile.

Generating wheel profiles

To drive the profiles, the robot generates a path offset to each side of the center for each wheel to drive. This is done by offsetting each individual point along the curve, then calculating distance between them to determine their position, velocity, and acceleration. The output of this is a set of points with a specified time, position, velocity, and acceleration which will be fed to the feedback controller that allows the robot to follow the points.

Generating consistently timed control points

All algorithms up to this point generate inconsistently spaced and timed points. To follow the profile, either points must be generated on the fly every time feedback control is run, or generated in advance to send to a motor controller like a Talon SRX.

Either method relies on being able to calculate the values of a control point at a given time. This is done by a interpolating between already-calculated control points. When a time is passed to the method to get the point, it searches for the previously generated control points directly before and after the given time. The method finds the proportion of the way between the two control points the given time is, then uses the linear interpolation method shown above to generate the values in between the two control points.

This step can be quite expensive if done all at once, so the Flying Toasters found it advantageous to do it as the motion profile was followed. When the feedback controller (Described below) is run, the control point for the current time is generated and passed to the feedback controller. This optimization allows the RoboRio to generate motion profiles in a matter of milliseconds rather than seconds.

Following profiles using feedback controls

Once a motion profile has been generated for each wheel, they are each followed independently.

In order to follow the profile, a motor controller has to account for positional error, velocity, and acceleration. The velocity and acceleration of each point along the curve has already been calculated. The output of the motor controller is set to a constant times the acceleration plus a constant times the velocity. Since this is independent of the robot's position, this is called Feed Forward Control. The output of the feed forward control is:

$$Output = K_{Velocity} * V + K_{Acceleration} * A$$

Where V is the velocity target and A is the acceleration target. After some testing, it became clear that the acceleration term was not nearly as important as

the velocity term, since the derivative term of the PID controller handles acceleration quite well.

The positional error is corrected with a PID controller. A PID controller tries to minimize the error of the robot's position, the integral of the error, and the derivative of the error, which is represented by the following:

$$Output = K_p * error + K_i * \int (error)\, dt + K_d * \frac{d}{dt}[error]$$

Where error is the difference between the target position and the actual position, and K_p, K_i, and K_d are the constants for the proportional, integral, and derivative terms.

The final output of the motor controller is the sum of the feed forward output and the PID output.

Good tuning of the PID and feedforward values is essential to smoothly follow motion profiles. The best method that we have tried for PID tuning is the Ziegler-Nichols method. You can learn more about the method here: https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method

The feedforward value is simply equal to $\frac{1}{max\ speed}$. This is the value needed to sustain a constant velocity with no correction. The acceleration constant should be tuned last by increasing the value until the error throughout the profile remains the lowest. We found that a precise acceleration constant is not nearly as important as a well-tuned PID and feedforward setup.

During the 2018 season, the Flying Toasters used our own PID/feedforward/acceleration system in our own code. Alternatively, Talon SRX motor controllers have the same type of control built in and only need to have control points periodically pushed to them.

Results

The result of this project is that our robot can generate a path to follow and then follow it smoothly and consistently. The error is within about 2-4 cm each time the profile is driven. The motion of the robot is still slightly jerky at the start and end of the profile, but this is to be expected without a jerk-minimizing S-Curve profile.

Here are some videos of the Flying Toasters' 2017 robot following the profiles (This is a very early result, before the turn speed correction was implemented):

https://www.youtube.com/edit?o=U&video_id=6r4NarlMbEg

https://www.youtube.com/edit?o=U&video_id=9m0POZ0W8fM

Newer video of autonomous motion profiles from the Flying Toasters' 2018 bot during the Michigan State Championship (Our robot is the farthest red robot):

https://www.youtube.com/watch?v=88TymDbz8MM

In the future, we are looking to implement jerk-minimizing S-Curve velocity profiles and use more optimized Quintic Hermite splines for path generation.