Pathing: a simplified approach

Ryan Pappa

robots@rpappa.com

Abstract. A common problem in robotics applications is developing a path planning and following system that enables robots to perform pre-planned motions. In 2018, FRC Team 340 explored an approach to pathing that aimed to make the process extremely simple. It relies on two drive encoders and a gyroscope. This approach is based off parametric equations (especially Bézier curves) and simple differential calculus. The complexity of any given implementation can be adjusted as desired.

1. An introduction to parametric equations

Most high schoolers are familiar with functional relationships, where an equation takes an independent input variable and outputs a dependent value. In parametric equations, a third variable (call it *t*) is introduced that is generally independent of the coordinate plane. The *x* and *y* coordinates are found via two separate functions, each taking *t* as an input. This is seen in projectile motion, where *y* position of an object is represented by $y(t)=y_o+v_{oy}t+(½)a_yt^2$ and *x* position by $x(\mathrm{t})=x_o+v_{ox}t$, with *t* representing time. Although *x* and *y* behave independently, both are dependent on time. The (*x*,*y*) position can only be found through passing time to both equations. Parametric equations allow interesting shapes to be drawn, such as a circle ($x(t)=\cos(t)$, $y(t)=\sin(t)$), or the butterfly shape shown in figure 1.

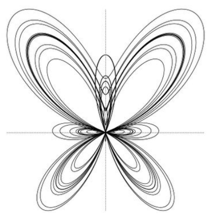

Figure 1: an example parametric

Later we will explore Bézier curves, a type of parametric equation, as they are extremely useful for defining paths for a robot to follow

2. An introduction to differential calculus

The derivative of a function $f(x)$, represented as $f'(x)$, is simply the **slope** of the **tangent** of a particular line at any given x-value.

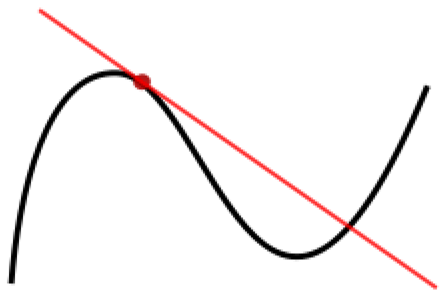

Figure 2: If the black line is a function $f(x)$, the slope of the red line equals f' evaluated at the point's x-value

There are a lot of rules for differentiating, which will not be explored in this document, but one must understand the nature of a derivative in order to understand this approach to pathing.

3. The theory behind 340's approach

The rest of this paper is based on the use of, at a minimum, one drive encoder (ideally two) measuring distance traveled and a gyroscope measuring the robot's angle. We assume a basis of a simple closed-loop controller that can drive while keeping a desired angle. This kind of system is commonly used to drive straight, where the desired angle is 0°. With error calculated by $Error = gyro - desired$, gyro output gives the error directly, since $gyro - 0^{\circ} = gyro$.

Let's define a path P as $P(in)$, where in is some input variable. Accordingly, the equation for the derivative of P is $P'(in)$.

We will start by telling our robot to drive forward. The value, in inches, our drive encoders will be defined as *d*. As we drive, we calculate *P'*(*d*). Now we have the slope of the tangent of our path. If you are following a path, your drive rails should always be parallel to its tangent, so thus by finding *P'*(*d*) you have found the slope of the lines created by your left and right drive rails. Additionally, you can find the current slope of your drive rails by finding the tangent of your gyro output, *m*=tan(*gyro*).

The angle between two slopes is found through this equation:

$$\theta = arctan\left|\frac{m_1 - m_2}{1+m_1 m_2}\right|$$

Thus, given the variables of your distance, *d*, and gyro angle, θ, you can find the error in your robot's angle:

$$Error \ = \ arctan(\frac{P'(d) - tan(\theta)}{1+P'(d) * tan(\theta)})$$

We simply pass this error back into the closed-loop system that was previously used to drive straight, and our robot follows the path *P*. In fact, we do not even have to define *P* in the code, but rather can just tell the robot the equation for *P'*, its derivative. Any path which can be modeled and differentiated can be followed. Now we simply have to figure out how to define paths that take distance traveled as an argument.

4. Using Bézier Curves

Bézier curves commonly used in computer graphics. They are defined through parametric equations where the *x* and *y* equations are polynomials that take an argument, *t*, which is the percentage (from 0.0 to 1.0) of the curve you have progressed through. They can be designed through a group of control control points.

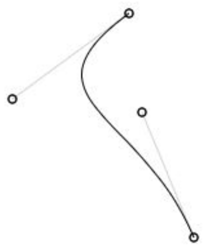
Figure 3: an example Bézier curve

Explore more about Bézier curves' construction here[1].

Simply put, Bézier curves are perfect for this application. Since we know where our robot should start and end, we can make a curve that takes in those two points, and visually adjust the two control points in the middle to produce the desired motion. Once we have the curve defined, we generate the parametric and its derivative. All we need to put in our code is the curve's derivative (P') and its length (*L*). As the robot moves, we evaluate

$$Error = arctan(\frac{P'(\frac{d}{L}) - tan(gyro)}{1 + P'(\frac{d}{L}) * tan(gyro)})$$

We divide our distance travelled by the length of the curve, since the domain of the Bézier curve, and thus also its derivative, is between 0.0 and 1.0. Passing this error back into the closed loop function allows us to follow any Bézier curve.

5. 340's implementation: defining paths

In the robot code, paths are defined through one or more PathSegments, where each segment is a function's derivative (defined through a lambda function[2]) and its length.

For now, I have made a relatively barebones website[3] for creating paths. Figure 3 is an example of a path generated by our website. The positive x-axis is forward

[1] https://www.desmos.com/calculator/cahqdxeshd
[2] https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html
[3] http://paths.rpappa.com/
Source code https://github.com/rpappa/path-site

relative to the robot’s position at the start of the path, and left and right correspond to negative and positive on the y-axis. The website generates PathSegments that can be pasted directly into the Java code. In the code, paths look like this:

```
new PathSegment(t ->
/*
{"start":{"x":0,"y":50},"mid1":{"x":46,"y":48},"mid2":{"x":51,"y":109},"end":{"x":
112,"y":108}} */
(-6 + 378 * t + -375 * Math.pow(t, 2))/ (138 + -246 * t + 291 * Math.pow(t, 2))
, 131));
```

The commented section is a JSON object that can be imported into the website.

6. 340’s implementation: calculating error

In RunPath.java[4], we have a method `dydx(double s)` that finds the PathSegment our robot is currently following and calculates the derivative at the given *s* value, which is the distance we have traveled across the entire Path.

We calculate error using the method `deltaAngle(double currentAngle)`, which takes gyro yaw as an argument. It first finds

```
currentSlope = Math.tan(currentAngle)
nextSlope = dydx(getDistance())
```

To find the error, in degrees, we use the previously shown equation:

```
angle = Math.atan((nextSlope - currentSlope)/(1 + currentSlope *
nextSlope))
```

This angle is what is returned by `deltaAngle`.

[4] https://github.com/Greater-Rochester-Robotics/PowerUp2018-340/blob/master/Team340PowerUp2018/src/org/usfirst/frc/team340/robot/commands/pathing/RunPath.java

7. 340's implementation: closed loop function

The closed loop we use is essentially a P loop, first implemented in our 2017 codebase[5]. At the time, it was used exclusively for driving straight, with nobody recognizing its potential application for pathing. The final implementation of it can be found in our 2018 code, in the RunPath.java file.

The loop is called in `execute` in the RunPath command. It takes in our error from `deltaAngle` and an input speed. Mathematically, it functions as follows:

$$outSpeed \;=\; inSpeed \pm \frac{error * |inSpeed|}{divisor}$$

The divisor is essentially a P constant. For the left rail, we add to the input speed, for the right rail, we subtract. Generally, both drive rails are being set to same-signed numbers, to produce arcing motions as opposed to turning on a point.

8. The future of this approach

There's a lot that could be added. The most obvious is to use more kinematics, regulating velocity and acceleration. That shouldn't be too hard to do: just pass the output of the kinematics controls into the input speed on the closed-loop control.

Speaking of closed-loop, actually using PID for closing error would probably bring benefits.

I started work on an "animations" system, where one can specify robot motions to occur at different points along the path.

5
https://github.com/Greater-Rochester-Robotics/Steamworks2017-340/blob/master/src/org/usfirst/frc/team340/robot/commands/DriveRails.java#L74

The theory behind this approach seems to be widely applicable. I would love to see ports to other robotics systems. C++ WPILib, Arduino, and possibly even Lego robots are all viable targets.

Our most immediate goal is to wrap this all into an easy-to-use, well-documented library. Abstracting just a few methods (setDriveRails, getDistance) could make this a 5 minute job to port into any robot's codebase.

9. Conclusion

Here I have explained an approach to path following that I feel is a perfect balance between simplicity and performance. I hope that teams can use this to improve their controls system, whether by using my specific implementation or by building upon the theory in their own.

If you need any help or have further questions, please feel free to contact Justin Tervay[6] or myself[7]. Furthermore, exploring 340's 2018 GitHub repo[8] as well as the pathing website[9] and its source code[10] may help you understand exactly how we used the ideas laid out in this paper to help win 2 blue banners.

This is my first time writing a paper like this, and I sincerely hope I have done a good job explaining what I personally find to be an exciting step towards raising the bar across all of FRC.

10. Acknowledgements

Justin Tervay (justin@tervay.com): assisted in implementation and testing of this approach, has helped a few other teams implement it for the 2018 season and will

[6] justin@tervay.com
[7] robots@rpappa.com
[8] https://github.com/Greater-Rochester-Robotics/PowerUp2018-340
[9] http://paths.rpappa.com/
[10] https://github.com/rpappa/path-site/

likely be helping maintain it after I graduate. He was the mentor who oversaw the majority of my work as I developed this and his help was simply irreplaceable.

FRC Team 340: provided me with 5 years of experience culminating with this pathing work. Furthermore, provided 3 different robots to test this on (one of which went on to win 2 regionals and rank 5th at champs—huge shoutout to everyone involved with that!)