FRC Guidebook for Titan Robotics 492

Abhay Deshpande

Table of Contents

[Introduction 3](#_Toc49560943)

[How We Do Math 3](#_Toc49560944)

[Tuning 4](#_Toc49560945)

[Basic Tuning 4](#_Toc49560946)

[Adding an I term 4](#_Toc49560947)

[Characterization 4](#_Toc49560948)

[Working with Vendor Libraries 5](#_Toc49560949)

[CTRE Phoenix 5](#_Toc49560950)

[Trapezoidal Profile – Motion Magic 5](#_Toc49560951)

[Offboard PID Control 5](#_Toc49560952)

[REV Robotics 6](#_Toc49560953)

[Trapezoidal Profile – Smart Motion 6](#_Toc49560954)

[Offboard PID Control 6](#_Toc49560955)

[Subsystem Ownership 7](#_Toc49560956)

[Vision Targeting 8](#_Toc49560957)

[Pixy Cam 8](#_Toc49560958)

[Raspberry Pi 8](#_Toc49560959)

[LimeLight 8](#_Toc49560960)

[Tuning 8](#_Toc49560961)

[Communication 9](#_Toc49560962)

[Depth Estimation 9](#_Toc49560963)

[LEDs 9](#_Toc49560964)

[Wiring 9](#_Toc49560965)

[Path Following 11](#_Toc49560966)

[Discrete PID Waypoints 11](#_Toc49560967)

[Talon Tank Motion Profiling 13](#_Toc49560968)

[Holonomic Pure Pursuit 14](#_Toc49560969)

[Holonomic Pure Pursuit V2 15](#_Toc49560970)

[Swerve Motion Profiling 16](#_Toc49560971)

[Shooters 18](#_Toc49560972)

[Flywheel Basics 18](#_Toc49560973)

[Flywheel Control 18](#_Toc49560974)

[Aiming 19](#_Toc49560975)

# Introduction

This is not a comprehensive guide to programming for FRC. What I hope this becomes is a collection of advice and general wisdom I’ve picked up that might help you. It should cover how to use some of the tools in the team library, and some basic techniques of other things such as tuning. I highly recommend you join the FRC discords, both the PNW and global one. The global one has 2 programming channels which are both fairly active. Although it may seem intimidating at first, it’s a very useful resource to learn more and share ideas with others. Additionally, Chief Delphi will be incredibly useful as well, so I recommend you monitor that and sign up for its weekly emailing list. Keep in mind that most other teams will use different conventions than us, so refer to the How We Do Math section when interpreting advice from other teams.

As I’ve said before, this will not be comprehensive. Fully understanding this will probably require extra research and work. In order to fully understand the math in the library, I recommend you understand a little bit of matrix math and some basic calculus. You should also have a firm understanding of reference frames.

# How We Do Math

We use what’s called the ENU (East-North-Up) coordinate system. This means that +x points to the right, and +y points forward. Additionally, our angles are measured in clockwise degrees, where 0 degrees is forward. (+y)

This is different than what wpilib and most other teams use. They use the NWU (North-West-Up) coordinate system. In this system, +x points forward, +y points left, and angles are measured in counterclockwise radians, where 0 degrees is forward. (+x)

Therefore, **make sure you check your math**. For example, atan2 has a method signature of Math.atan2(double y, double x). However, in a lot of cases, you shouldn’t pass in x for x and y for y, since that will give you the complement of the angle you want (assuming you want angle from +y). You should do Math.atan2(x, y). To verify this, check by drawing diagrams and use trig.

Additionally, if you ever use any code/math pulled from wpilib, **you** **need to convert coordinate systems.** This is because wpilib (and most other teams) use the NWU system. Check examples of system conversion in FrcSwervePathFollower and TrcDriveBase. (the updateOdometry method) To do this, you need to use what’s called a change of basis matrix, to convert from ENU to NWU, and vice versa. You should check the code for examples.

# Tuning

To do basically anything, you will have to tune controllers. PID/PIDF is the most common, and I’ll briefly cover a few tips/tricks on how to tune them.

## Basic Tuning

Setting initial value of kP

You can choose an initial kP by thinking about it as "output per error". It's more of an art than a science, but you can ask the question "at what error should my robot respond with x% power?". Then .

For example, if you're designing a PID controller to control heading, you can say that you want the robot to respond with 50% power when the robot is 90 degrees away from the target. Then your initial kP is .

Tuning a position PD controller:

1. Zero all PID constants and set P to some initial value, as described above.
2. Decrease P until the mechanism doesn’t oscillate. If it already doesn’t oscillate, go to step 3.
3. Increase P until the mechanism oscillates a little too much.
4. Start D at P/10, adjust so that the mechanism overshoots by a *tiny* bit, and then settles fairly quickly.
5. Increase P again until the mechanism oscillates a little too much.
6. Increase D until the desired behavior is found.
7. Repeat steps 4 and 5 until the settling time is to your liking.

Finding top speed:

1. If possible, run at full power and graph speed
2. The limit is the top speed

OR

1. Run at partial power and graph speed
2. The limit is the measured speed
3. Calculate top speed as topSpeed = measuredSpeed / commandedPower.

## Adding an I term

If you want to quickly settle to a value and precision is incredibly important (swerve modules, shooter arms, etc) then using an I term will help in reducing steady state error. In position controllers they’re somewhat unwieldy to work with, however. First, if your controller requires precision then you shouldn’t be using a straight PID controller anyway. You should be using high-resolution control, using the equivalent of a trapezoidal profile. (Motion Magic in CTRE, Smart Motion in REV, etc) This allows you to use a PIDF controller, and you can get very accurate and tightly controlled loops.

In order to effectively use an I term, you basically **need** an integral zone. Essentially, the integral zone is the error threshold at which the I term is enabled. If the absolute error exceeds the I zone, the accumulator is zeroed, and the I term stays at 0. The I term only accumulates when the error stays below the I zone. This makes sure that the I term only helps with steady state error and doesn’t destabilize the rest of the system.

## Characterization

The Robot Characterization Tool is pretty useful too. It’s a mostly automated tool that automatically determines the real-world characteristics of your system and spits out useful constants, including kF. It also approximates PID constants, but that’s a little more finnicky. It’s changing all the time, so just check the wpilib docs for the latest information. When you use it, make sure you correctly account for gear ratios, encoder constants, etc. Sanity check everything.

# Working with Vendor Libraries

We use a lot of 3rd party libraries, mostly for obvious reasons. Usually they’re abstracted away with trclib and frclib, however you still need to know about them, especially if you want to offboard control loops for better precision. I’m only going to cover specific features of motor controllers, since figuring out most other stuff shouldn’t be that bad.

## CTRE Phoenix

CTRE makes Talons and Falcons, and they share a mostly common software interface. You should generally be controlling these via CAN. Newport sometimes doesn’t, but they’re not exactly role models. Check the CTRE Phoenix docs, they’re very useful. Generally, just make sure you keep the weird hijinks in mind. As of 2020, CTRE measured PID output on a scale from [0, 1023] instead of [0,1]. This affects the scaling of your PID constants. As of 2020, CTRE provides no unit scaling functionality, so “ticks” refers to raw encoder ticks. Velocity is measured as ticks/100ms. Also, acceleration is measured in ticks/100ms/second. The unit scaling and the PID output scale is definitely subject to change, so refer to the docs.

CTRE has a very mature codebase, and although it has some weird gotchas, it’s reliable. Device management is super easy with Phoenix tuner.

Additionally, always use current limits, they will prevent brownouts from excessive current draw. If using a Falcon, I recommend putting a ramp rate as well, since brushless motors tend to draw a lot of current.

### Trapezoidal Profile – Motion Magic

When you need precise control, Motion Magic is what you want. Whenever you need tight control, (swerve steering, shooter aiming) motion magic will serve you best. It’s more work, so if it’s not super necessary you can opt for a regular position loop instead.

The PID constants will be the same as a regular position controller. Tune the FF like normal. You will also need to determine a max acceleration and max speed. These should **not** be your hard limitations. They should be comfortably less than your limits, to allow for headroom and consistent operation across battery voltages. These will be used to construct a trapezoidal velocity profile and will use the FF to follow it and use PID to follow the interpolated position setpoint.

### Offboard PID Control

Phoenix provides a lot of closed loop control and I’m not going to explain them all. But they are all very high-resolution control. Velocity and position control are generally what you’d use.

## REV Robotics

REV Robotics makes the Neo and Neo 550. One benefit is that the API provides support for unit scaling, which is nice. Additionally, it’s a little more intuitive than CTRE. A major downside is that device management is crap. Flashing the firmware is slow and clunky, and I’ve had to hard reset faulty firmware installations way too many times. It also requires physically plugging into the motor controller with a USB-C cable, which is unwieldy. All these may have changed in the future, and this is as of 2020. Another downside is that the integrated encoders have a low resolution, which makes them less useful for high-precision control, unless you have a high gear ratio. Both Neo motors clock in at 42 CPR, which is a lot lower compared to quad/mag encoders (4096 CPR) or the Falcon integrated encoder. (2048 CPR) It’s worth noting that sometimes you don’t need that much precision, especially if your gear ratio makes up for it, in which case the Neo motors are great. For example, in 2019 and 2020 we used the Neos as drive motors (both in the mecanum base and in the swerve modules) and they worked fine, since they offered an appropriate amount of precision.

Since both Neos are brushless, I highly recommend setting current limits AND ramp rates. This will prevent excessive current draw and brownouts. We’ve lost matches because of this.

### Trapezoidal Profile – Smart Motion

Smart motion is basically the REV equivalent of Motion Magic. That is, it uses FF to follow a trapezoidal velocity profile and uses PID to track a moving position setpoint. A cruise velocity and max acceleration must be specified in order to construct the trapezoidal profile. Again, these should be comfortably below the physical limits of the robot.

### Offboard PID Control

REV Robotics also provides a lot of onboard PID controls. They’re high frequency control loops, and they will typically outperform onboard control loops.

# Subsystem Ownership

In the past few years, we’ve started bringing autonomous routines into tele-op. Often times, the robot will autonomously control part of the robot, while the driver and operator will control the rest. This means that the code and the drivers may have mappings to control the same parts of the robot at the same time, which is obviously not a good idea.

For example, in 2020 we had autonomous shooting mode. In this mode, the robot automatically controlled the heading of the robot to point at the target, aimed the shooter correctly, and set the flywheels to the correct speed. The humans were responsible for moving the robot and triggering the actual shot. Coordinating this meant that the humans shouldn’t be able to accidentally provide input to the subsystems being controlled autonomously during this routine.

The solution to this was Subsystem Ownership. Subsystem Ownership is a system by which specific routines will own specific subsystems. If another routing tries to operate that subsystem, it will either no-op or throw an exception. A subsystem that is able to be owned is called an Exclusive Subsystem. Handling the ownership of a subsystem must be written into the specific subsystem code, but boilerplate code is taken care of by implementing the TrcExclusiveSubsystem interface. This interface only has default methods, none of which should be overridden. It provides code to handle checking, claiming, and releasing ownership of a subsystem. The subsystem class should use these methods to appropriately handle its ownership. An example of an exclusive subsystem is Conveyor.java in the 2020 code.

# Vision Targeting

## Pixy Cam

Before 2019, the main vision targeting used on the robot was the PixyCam. This did basic HSV filtration and blob detection, and outputted a bounding box, which the robot used to inform its decisions. However, while it was good enough in some cases, it wasn’t great. Tuning the PixyCam required dedicated software, and it also required physically plugging into it. Additionally, the algorithm for target detection wasn’t super accurate, and didn’t offer the flexibility we needed. In 2019, we decided to move away from the PixyCam to more advanced vision targeting systems.

## Raspberry Pi

The first attempt at better vision targeting was something I coded entirely from scratch, built using OpenCV and running on a Raspberry Pi. It was coded specifically for the 2019 game, and in testing performed very well. It used the EPnP algorithm to localize the detected target in 3D space, which meant it could tell you where it was, and how it was oriented. However, while it worked well during testing, it wasn’t reliable on the robot. Static electricity buildup was very common, and this corrupted the SD cards left and right, and even caused some damage to the board itself.

It’s also worth noting that the math behind it caused me to confuse myself quite often, and the performance was a little black magic in that it’s not always clear if things are working or not. In short, it was a good academic exercise, but it’s not really practical.

## LimeLight

This is the current iteration of what we use for vision targeting, and it works very well. It’s significantly more expensive than either the PixyCam or the Raspberry Pi, but it’s reliable and accurate, so it’s worth it. Since it’s the current system we use, I’ll go more in depth on it.

All handling of the LL happens in FrcLimeLightVisionProcessor.

More recently, the LL supports hardware zoom and panning, which is pretty useful for long range vision targeting, such as in 2020. It should be pretty easy to make a new pipeline using a different zoom and programmatically change the pipeline. There may be a new way to programmatically change the zoom without changing the pipeline, you’ll have to check the manual.

Additionally, while the LL supports mapping a hostname, I highly recommend you give it a static ip. The robot radio is infamously unreliable with hostnames, so just don’t bother.

### Tuning

Tuning the LimeLight is pretty easy. Go to the ip or domain of the device, and you can mess with a bunch of sliders. It also supports different types of targets, such as single target or double target, as well as intersection patterns. (are the targets pointing towards each other, away from each other, or neither) In some cases it may not be readily apparent what something does, so you should consult the documentation.

Every time you tune the LL, you should download the configuration, and optionally check it into the repo. This ensures that you have a backup in case you need to revert, or if you need to swap in a new LL.

### Communication

Communication with the LL happens via Network Tables. In most cases, this should be abstracted away. You might want to manually edit the LL Table, in which case you can boot up OutlineViewer.

### Depth Estimation

In order to calculate the position of the target, the camera needs to calculate the target depth. That is, the distance between the target and the camera. We can then use the angle to the target to calculate the relative x and y position of the target. Since the depth estimation sometimes varies on the application, any LL feature can be mapped to a target depth using the setDepthApproximator method. For example, in 2020 the vertical angle to the target was used to calculate the distance. Check the code in VisionTargeting.java in the 2020 repository for an example. (it’s basically geometry)

To know what LL features are available, check the LL NetworkTables documentation.

### LEDs

The LL is infamous for its LEDs being absolutely blinding. I can confirm this. I would see spots in my vision for the next day or so after the competition. Therefore, you should write code to turn off the LEDs when they’re not in use. (especially in disabled mode) Additionally, the LL supports dimming the LEDs, so that’s an option to consider.

If the LEDs are too obnoxious, the judges may rule that they’re disrupting the other teams, and they may make you change it, either by moving the location of your LL or removing it entirely. These situations should obviously be avoided, so be careful with your LL.

### Wiring

The LL has 2 wires coming out of it: power and ethernet. It supports PoE, so you may decide you don’t need to have a power cable. There’s something to be said for redundancy, so make an informed decision. The LL takes unregulated 12V (it has a built-in regulator), so you should plug it directly into the PDP. Since the LL is usually far from the electronics, you should take care to make the wiring clean.

Since the LL communicates via ethernet, it will need to plug into the radio. Unfortunately, the radio only has 2 ethernet ports, one of which is taken by the RoboRio. This means that when both the RoboRio and the LL are connected, physically tethering the robot via ethernet becomes impossible. There are a few different ways to remedy this.

In 2019, we put an ethernet switch on the robot. When we needed to tether, we would plug the tether and the LL into the switch, and then plug the switch into the radio. Originally the plan was to wire the RoboRio and the LL to the switch, but we decided against it, since we didn’t want a switch failure to cost us a match.

In 2020, we didn’t want to deal with the extra hardware of the switch, so we plugged both RoboRio and the LL into the radio. Then, tethering was done via USB. This works pretty well, but it has one downside. Since the tether is via USB, the laptop doesn’t have access to the radio subnet. This means that neither the camera feeds nor the LL page could be accessed, which wasn’t very ideal. If the camera feeds are critical (as it was in 2020) this isn’t a great option for driver practice. For basic testing and maintenance, it works fine.

We never got around to implementing it, but wpilib gives you a way to forward traffic from USB to devices on the radio subnet. Therefore, it’s an option to stick with USB tethering, and use this to access the camera feeds and LL page on the radio subnet. For more information check out [the WPILib docs about roborio port forwarding.](https://docs.wpilib.org/en/stable/docs/networking/networking-utilities/portforwarding.html)

# Path Following

Path following is a very important task in FRC auto, and some of our implementations are pretty untested. After the 2020 season (or lack thereof) we have a few different methods of path following.

* Discrete PID Waypoints
* Talon Tank Motion Profiling
* Holonomic Pure Pursuit
* Holonomic Pure Pursuit V2
* Swerve Motion Profiling

I’ll cover what each does.

## Discrete PID Waypoints

This was our original path following system, pre-2019, but it’s evolved a bit in recent years as well. Essentially, it uses a PID (or PD, or P, or whatever) controller operating on position to move to a setpoint. This works for both holonomic and non-holonomic drivetrains.

Pre-2020, this was incredibly slow, as the robot stops at each intermediate waypoint, since it can only move in straight lines. Additionally, it was made worse because each intermediate waypoint was controlled for with unnecessary accuracy. This was slightly improved during the 2020 FTC season, where the waypoints were adjusted using the new field oriented odometry system, so accumulated error in prior PID operations could be corrected for in subsequent PID operations. For example, say it’s commanded to go forward 60 inches, then left 60 inches. If it goes forward 58 inches instead of 60, and then goes left 60, it’ll end at (-60,58) instead of (-60,60). The new system fixes this, since it’s aware of the residual error from the first PID operation, and then corrects for it in the second operation.

In order to use this, you **must** mess with no-oscillate. The no-oscillate setting on the PID controller (TrcPidController, not TrcPidDrive) is a flag that says that as soon as the sign of the error changes, the PID controller is marked as on target, without waiting for it to settle. This makes it much less accurate, but for intermediate waypoints, nobody really cares. Therefore, enable no-oscillate on the x and y PID controllers for all intermediate PID operations, and then you **must** re-enable it for the last PID operation, or else it will not be accurate. **Never use no-oscillate for the heading controller.** This will screw up the reference frames and cause the path to be very inaccurate.

How to tune:

Tune position and heading like normal, outlined in the basic tuning section.

This approach has some benefits:

* It’s incredibly simple. Getting the robot moving has basically zero boilerplate code, you only have to create a TrcPidDrive instance, and call the setTarget method.
  + Compared to other techniques where you have to create a path and a velocity profile, this is dead simple.
* It’s very useful for simple and time-insensitive movements. For example, in 2020, in one of the auto options, after shooting the balls, the robot just had to back up a few feet. For such a simple movement, PID fit the bill.

There are some important drawbacks to keep in mind.

* Heading cannot be controlled at the same time as position
  + This applies for both holonomic and non-holonomic robots
  + The method may exist that allows you to set heading and position setpoints at the same time, but **this will not work.**
  + This mainly arises from an issue regarding reference frames and coordinate systems. This is an implementation detail; it’s definitely possible to do 3-DOF PID (PID on x, y, heading) at the same time, but our current system doesn’t allow for it.
* Faster implementations exist
  + In some cases, this approach will be more than fast enough
  + However, even with the odometry based enhancements, the robot still has to speed up and slow down with each segment, so more advanced path following techniques with curved paths will usually outperform PID
* Requires one state for every PID operation in a state machine. This doesn’t really have a performance impact, but it increases the auto routine complexity. It’s a lot more work and makes more room for a stupid mistake.

## Talon Tank Motion Profiling

Please don’t use this. This was the first ever attempt at path following beyond PID waypoints. It uses the Talon SRX’s built-in motion profiling feature to follow a path. It’s not marked as such in the code, but it should be deprecated.

How to tune:

1. Tune position PID according the basic tuning section and transform these constants to talon native units. Generally, you can do this by doing constant\*inchesPerTick\*1023=scaledConstant.
2. Calculate kF as 1023/topSpeed, where topSpeed is the top speed, in native talon units. Refer to the CTRE Phoenix docs for more info.
3. The scale for PID may have changed since 2019, so verify that the number to use is 1023; it may be 1. Refer to the CTRE Phoenix docs.

The only pro:

* This is the only non-holonomic motion control in our library. However, if you have a non-holonomic robot, I highly suggest you just use the wpilib Ramsete controller implementation. It should perform a lot better. It’ll require a bit of coding and learning, since it uses the Command-based framework, but you can refer to FrcSwervePathFollower for an example on how to wrap a wpilib command as a TRC task.

There are several downsides:

* It requires the drive motors to be Talon SRXs. This hasn’t been the case since 2018.
* It requires a non-holonomic drivebase. Holonomic robots can be used, they’ll just be driven like tank drive. For holonomic drivebases, there are better alternatives.
* It requires pre-generated paths. It loads paths from CSV files that are stored on disk. Paths cannot be made or modified on the fly. It may be possible to create paths at runtime with wpilibs new path creation code, but this isn’t tested.
* It requires a lot of fudging. You have to measure an empirical drivebase width, which is very finicky, and has a large impact on path following performance.
* Inaccurate heading control. Since the control is running offboard, it cannot access the Nav-X gyro. Therefore, heading is calculated using the drivebase width and the encoder values, and this value will not be very accurate. However, it’s possible to use a CTRE Pigeon to give heading values.

## Holonomic Pure Pursuit

This was the first “real” holonomic path following we had, and it works pretty well. An upside of this method is that the path is only stored as a series of waypoints, and they can be created very cheaply at runtime. Look at the 2020 auto code for an example of paths being created and dynamically accounting for error. The setpoint is automatically interpolated between waypoints using a pure pursuit algorithm. The path is not guaranteed to be followed exactly. It will loosely follow the path, but only the ending point is guaranteed. Refer to the documentation of TrcHolonomicPurePursuitDrive for more information.

Additionally, keep in mind that **the supplied path must be in the reference frame of the robot at the start of the command**, and the first point should be the robot’s current pose. In special cases, you can probably get away with ignoring the second rule, but the first must always be followed.

How to tune:

1. Tune position and heading according to the steps outlined in Basic Tuning.
2. The velocity controller can easily be FF-only. This could be calculated by doing 1/topSpeed, which is generally good enough. The top speed can be empirical or calculated. All other PID constants can be set to 0. If you’d like, you can tune them as well, but they’re usually not required.

Pros:

* Works for any holonomic drivebase
* 3-DOF control (x, y, heading)
* Follow curved paths without stopping (much faster than PID waypoints)
* Follow entire path in one state
* In most cases, it’s probably good enough
* Time-invariant. This means that if for some reason it gets stuck somewhere, it won’t terminate early like a motion profile; it will keep going until is on target.

There are a few drawbacks to keep in mind:

* No guarantees are made on the time duration of the path. This is because it’s time invariant, so it will only finish when it’s on target, but therefore it can’t tell you exactly when it will finish. If you want to set a timeout, run it a few times to get a feel for how long it takes.
* The velocity controller is more of a suggestion if anything. This controller sucks at making the robot follow any sort of coherent velocity profile, mainly because the position and velocity controllers are contributing nonnegligible output, so the robot tends to go much faster than the velocity setpoint. This could *potentially* be tuned out, but it gets sketchy. Adherence to a top speed can be enforced with the move output limit, but this only sets the top speed for the entire path; it doesn’t offer any sort of accurate velocity control.
  + Usually this isn’t a huge problem, but in some cases, you need accurate velocities. In 2020, picking up balls on the move inside the trench couldn’t be done at high speeds.
* The path isn’t followed exactly. Only the ending pose is guaranteed.

## Holonomic Pure Pursuit V2

This is very similar to the other form of pure pursuit, with some important modifications. There is no position PID controller here. All the work is done by the velocity PIDF and acceleration feedforward. The heading controller is untouched. This means that there are much stronger guarantees made about the velocity control. It should be a lot more accurate at following the supplied velocity profile. An important thing to note is that the I term of the velocity PIDF is analogous to a P term of position, so if necessary, that *could* be used to account for error. **This implementation is untested on a physical robot.** I’ve run it through simulations, but that is no replacement for an end-to-end test on a real robot.

**There is a very important complication.** A quirk of this method is that the velocity setpoint is the velocity at the point on the path closest to the robot. This means that if the robot is at the beginning, the commanded velocity is 0 (if the robot starts at rest), so the robot never moves. There are a few solutions to this. One is to set the start velocity to above 0. The value should be tuned. This will get the ball rolling and start the robot’s movement. The other is to rely on the acceleration FF to get the robot moving, and then the velocity PIDF will kick in. This is probably sufficient, but keep in mind that when tuning the acceleration FF, it starts at 0, and therefore cannot be relied upon, so the former solutions must be used.

How to tune:

1. Zero all constants.
2. Tune heading like normal.
3. Calculate the velocity feedforward using 1/topSpeed, measuring top speed according to the steps in Basic Tuning.
4. Run a simple trapezoidal velocity profile, and graph commanded velocity vs. actual velocity. Remember than since acceleration FF=0, the starting velocity should be nonzero. (refer to last paragraph above)
5. Increase acceleration FF to fix the phase shift between actual vel and commanded vel. If there is negligible phase shift, acceleration FF can be 0.
6. Repeat steps 4-5 until the commanded velocity matches the actual velocity graph.

Pros:

* More accurate velocity control
* All the other pros of Pure Pursuit V1

Cons:

* Harder to tune
* The path isn’t followed exactly
* Requires a rough hack to move (sometimes, refer to the last paragraph above)

## Swerve Motion Profiling

**This only works for swerve drivebases!**

**Important!** As of the summer of 2020, this controller will need reworking and improvements in order to work with the updated wpilib code. Refer to the bottom of the cons list for more information.

This is a wrapper around the swerve controller implemented in wpilib. This is a motion profile, so it is time-variant. Because it’s from wpilib, it offers powerful controls for path creation, in the form of path constraints. This can be used to limit the commanded velocity, the centripetal acceleration, etc. Additionally, paths can be created on the fly using the wpilib path generation, just don’t do it very frequently.

Velocity should also be accurately controlled with this implementation, however there’s a little less freedom in the velocity control. All paths must have a trapezoidal velocity profile. To fix this, however, it very naturally allows for multiple trapezoidal velocity profiles to be followed one after the other. Simply create multiple paths, starting one where the other one ends, and pass them all to the vararg parameter in the start method. This is really only necessary when the velocity profile cannot be trapezoidal, so it shouldn’t be required in most cases.

Additionally, another constraint is that heading is only guaranteed at the end of the path. All intermediate heading values will be ignored. If the robot is required to be facing certain directions at certain points in the path, split up the path into multiple segments and follow them all consecutively.

It’s important to note that a path segment does not have to end with 0 velocity. This allows path segments to be followed in rapid succession, since the robot doesn’t have to speed up and slow down. Therefore, if you split up the path into multiple segments for any of the reasons above, don’t set the intermediate segments to end at 0 velocity.

~~You may see some quirky behavior where the robot curves slightly in places that aren’t necessarily required. It’s not a huge problem anyway. This is intrinsic to the behavior of the path generation and cannot be fixed.~~ As of the summer of 2020, this may have changed. Refer to the bottom of the cons list for more information about wpilib path generation.

**This has not been tested on a physical robot!** Again, this has been tested in simulations, but this should be tested with a real robot before use.

How to tune:

1. Tune position and heading PID like normal, outlined in Basic Tuning.
2. Find max wheel speed. This is basically the max speed of the robot, so find like normal.
3. Find a suitable maxRotVel and maxRotAccel. These numbers don’t have to be super accurate, as long as they’re high enough to accurately follow the path, but not outside of the capabilities of the robot. Generally, maxRotVel can be calculated using maxWheelSpeed and some trig, and maxRotAccel should be empirically determined. **Your maxRotVel and maxRotAccel should be less than your physical maximums, to allow for headroom.**

Pros:

* Powerful path generation and constraints
* Guaranteed path duration (approximately)
* Accurate velocity control
* 3-DOF control
* Closely follows the entire path, unlike pure pursuit

Cons:

* A little cumbersome for complicated paths with non-trapezoidal velocity profiles or required intermediate heading values. In those cases, the path must be split up into multiple segments.
* There’s a minor hack in the path generation code, the heading setpoint is discontinuous at the exact last timestep. This is in order to decouple heading from the velocity vector direction, which isn’t done very well in the wpilib code. Make sure that this doesn’t cause any issues. ~~This is a known issue in wpilib, and I highly suggest you check issue #2438 on the allwpilib github repository before using this. If that issue has been fixed, then FrcTrajectory and FrcPath might need reworking. Good luck!~~ This has been fixed in the wpilib code. Refer back to the mentioned issue ([#2438](https://github.com/wpilibsuite/allwpilib/issues/2438)) to see the updated code. You’ll need to rework FrcTrajectory and FrcPath accordingly. Without those fixes, this controller may not be usable.
  + Having given the PR a brief look, it seems as if not much changing may be required. FrcSwervePathFollower will need to be updated to match the new constructor signature of SwerveControllerCommand. (and it will need to provide a desiredRotation supplier) After that, the FrcHolonomicTrajectory class can *probably* be scrapped, and the regular wpilib Trajectory class can be used instead.
    - **You should definitely double check me on this.**

# Shooters

One of the most common kind of shooter in FRC is the flywheel shooter. It’s very versatile, and if done properly, allows for very accurate shooting. Obviously, it’s application will depend on the challenge. Flywheel shooters generally work very well when you have to shoot multiple things, or have to shoot pretty accurately, or both.

## Flywheel Basics

When I refer to a flywheel’s MOI, I’m referring to the moment of inertia of the flywheel around the axis of rotation. For flywheel shooters, basic principles apply. High MOI is great because they lose less rotational momentum (and therefore velocity) with each projectile shot (this allows shooting in rapid succession with very little deviation), but it comes at the cost of taking longer to spin up. High torque from the motors is great since the flywheel spins up faster, but this should be balanced against the top speed, which obviously decides the range of the shooter. This all is mostly a mechanical problem, but it will affect the characteristics of the shooter.

Also, if your flywheel has multiple wheels, they **must** be mechanically bound. Software synchronization is insufficient to guarantee accuracy.

## Flywheel Control

Flywheels should be run on the tightest control loop possible, to maximize performance. To that end, I **highly** recommend moving the control loop to offboard the main controller. The Falcon is a great choice for flywheels. They’re compact, they offer great torque, they have a nice built-in encoder, and they run a high-resolution integrated control loop.

Yes, you could run fancy control like state space or something, but honestly velocity PIDF is sufficient. The FF constant is incredibly important, so this **needs** to be tuned accurately. The P term will be used to react quickly. Since this is a velocity controller, we can afford to be fairly aggressive, so the P term can speed up convergence to the target speed.

The I term (although usually ignored) can be pretty important in this controller, but not strictly required. The reason for including the I term arises from the fact that this is a 1st order controller. (we’re controlling velocity) Given that, an I term, if tuned properly, won’t destabilize the system as easily in a position controller. (0 order controller) The FF term does most of the heavy lifting in getting the velocity close to the target, but since FF is purely open-loop, the I term is the only term that can fix the steady-state error that arises from real life not matching the open-loop system. The P and D term contribution will be negligible when the velocity is only a little off from the target and relatively stable, so the I term will accumulate and push the velocity to the target velocity. Using an I zone will be helpful to prevent excessive accumulation.

## Aiming

Something to keep in mind is to **keep it simple.** I originally didn’t follow this advice in 2020, and it bit me in the ass. I tried to go the fancy route of calculating optimal trajectories using a linear drag model that were “theoretically correct”, but they really didn’t hold up in practice. See our accuracy in glacier peak if you want an example. (although technically we don’t know how much of the inaccuracy was the model, since that shooter was anything but consistent at that time) If you can get that to work, more power to you, but I found it to be more trouble than it was worth.

What ended up working really well was to empirically determine trajectories at a variety of distances, and then interpolate between those distances at runtime using camera data. For example, at every distance from 10ft to 35ft incrementing by 5ft, we measured the shooter angle and flywheel velocity that got us the perfect shot. We then stored this in a lookup table, and then we can interpolate between these values to get an approximation of the optimal angle and velocity. A significant downside of this approach is that it’s highly coupled to the physical characteristics of your robot. The minute anything changes, this entire process will have to be redone. For example, if the zero position of the shooter changes, or the gearing changes, or the flywheel radius changes, etc. It’s not a great idea to have a system so highly dependent on mechanical characteristics that’s unwieldy to retune, so if you find a better solution that works, use that.

It’s important that any sort of good aiming requires a mechanically sound shooter. Shooters are very sensitive, so they need to be very consistent and incredibly reliable.