RIT Software Team Notebook 2023

Vex U Spin Up

Software Team:

Ryan McGee
Richie Sommers
Victor Rabinovich
Brianna Vottis

Table of Contents

Section 1: Technology Overview **3**
- Custom Core API 3
- Open Source Software 3
- Git Subrepo 3
- Collaboration Tools 4
- Documentation 5

Section 2: Software Stack **7**
- Overview 7
- Odometry 8
- Drivetrain 8
- Control Loops 10
- Auto Command Structure (ACS) 11
- Flywheel 11
- Other Subsystems 12

Section 3: The Development Process **13**
- Overview 13
- Motion Profiles 13
- Odometry 14
- Tracking Tuning Process 15
  - Drivetrain Tuning Manually 15
  - Drivetrain Tuning Helpers 17
  - Generic PID Tuning Helper 18
  - Manual Flywheel Tuning 21
  - Flywheel Tuning Helpers 22
- Screen Helpers 22
  - Graphing API 23
  - Motor Statistics 23
  - Odometry Map 24
  - Vision Chooser 25
  - Screen Subsystem 25
- Roller Sensing With Vision 26
- Achieving Robustness 28

References and Links **29**

Section 1: Technology Overview

Custom Core API

All of the robot code we use is built on top of our own custom library, called the Core API, which itself is built on top of the official VEX v5 library. This API contains template code for common subsystems such as drivetrains, lifts, flywheels, and odometry, and common utilities such as vector math and command-based autonomous functions. This code remains persistent between years and is constantly updated and improved. The library can be found at ***github.com/RIT-VEX-U/Core***

Open Source Software

The RIT Core API is under the MIT open-source license, and is open for other teams to use and improve upon via pull requests. This system was modeled after the Okapi library from the Pros ecosystem, and offers similar functionality for the VexCode ecosystem. Teams that use this API are also encouraged to open source their software.

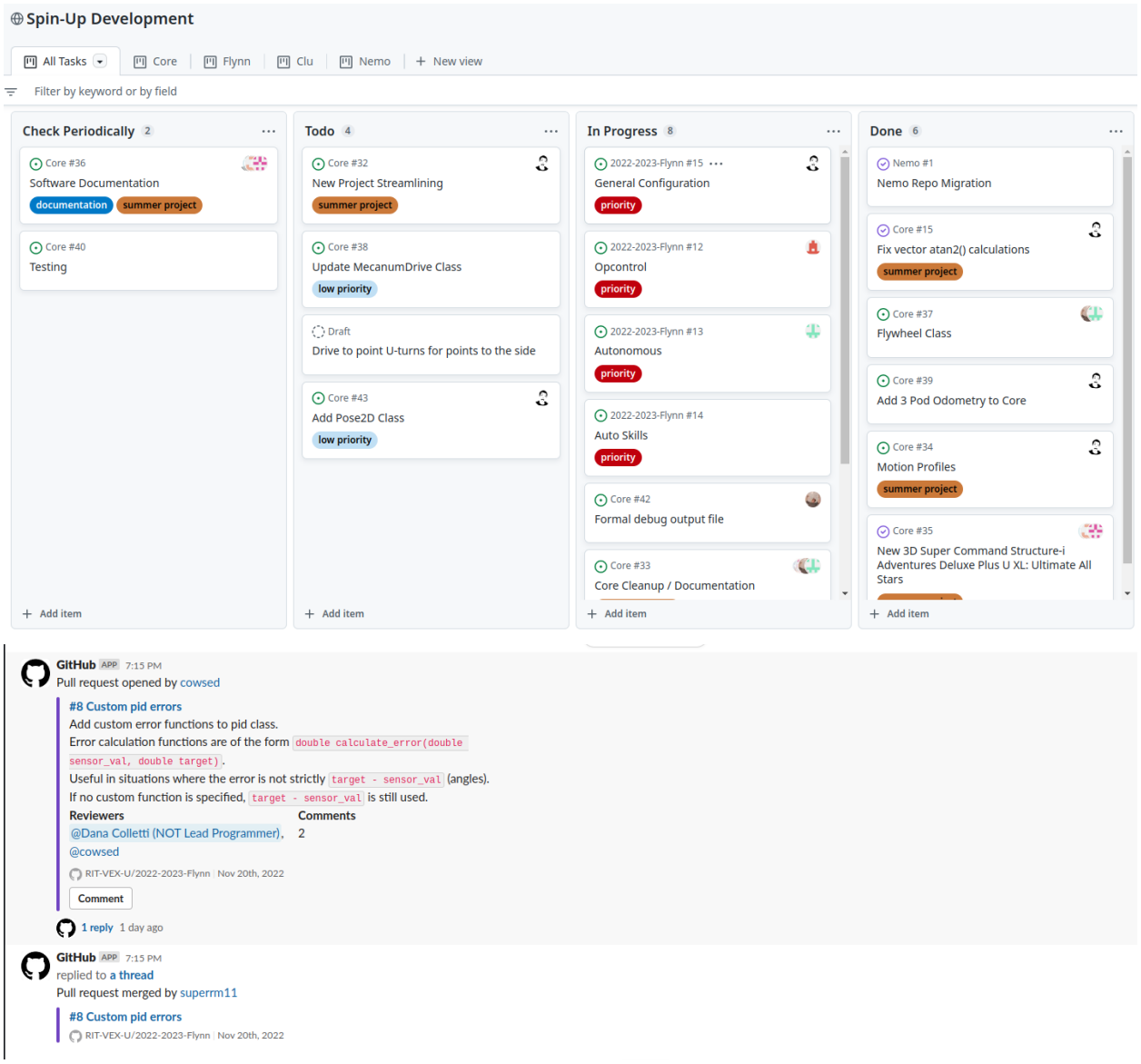
Git Subrepo

The Core API uses a unique type of version control called Git Subrepo (github.com/ingydotnet/git-subrepo). This allows users to simply clone the repository into an existing VexCode project to have instant access to all the tools. It also allows users to instantly receive updates by pulling from the main branch, and makes sharing code between two robot projects easier with git code merges.

Collaboration Tools

We used GitHub along with its Projects tool to collaborate on the code. Using Github, we hosted a git repository that can be accessed by the members of the team in order to collaboratively work on the code using branches, to minimize the amount of conflicts that would occur. The project board tool was used to assign and create tasks for members to complete. Each task would describe what needs to be done for it, the people assigned to it, and the state of each task. As such we were able simultaneously complete different tasks and parts of the code while also tracking the progress made to the code.

The Github Project is linked to respective repository issues, and new cards / card updates automatically send notifications to our Slack server, pinging our programmers.

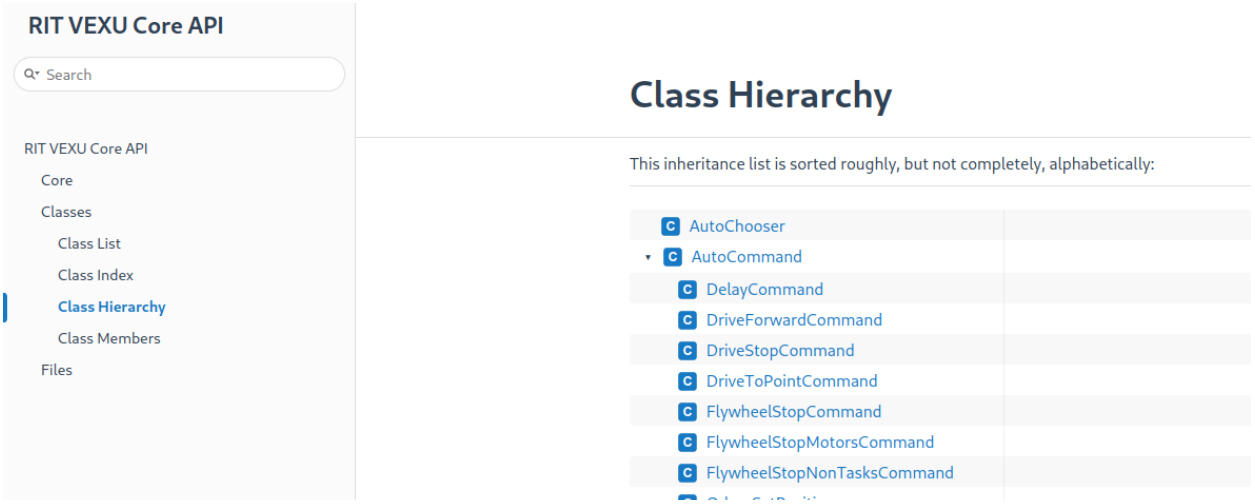


Documentation

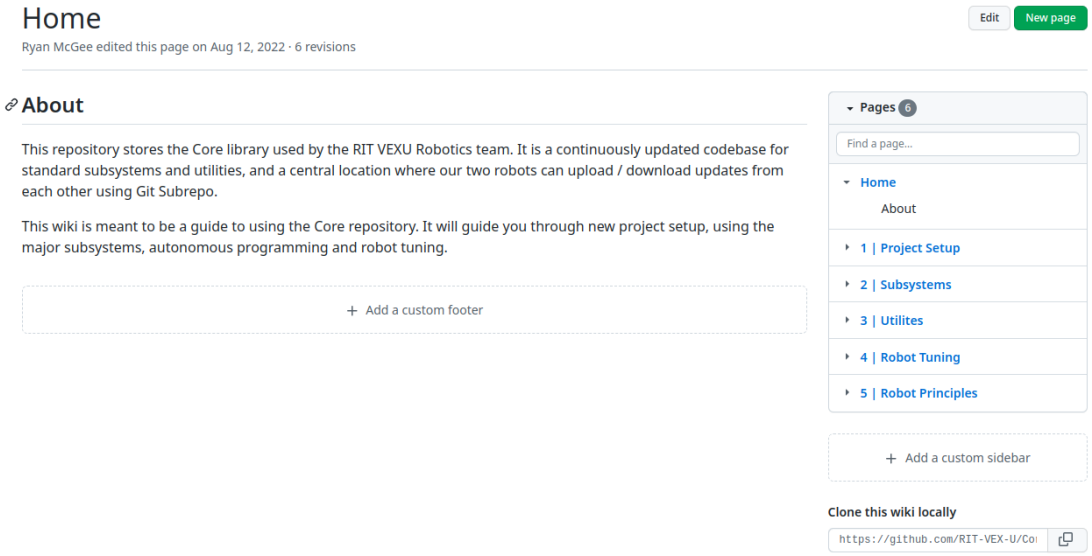
Our documentation pipeline has 3 steps: Inline code comments, Auto-generated Doxygen documentation, and the Core API Wiki. Our software team has an internal style guide that dictates formatting and what information must be included in comments at the beginning of classes and functions, to ensure our code is consistent and that our auto-generated documentation has the correct tags to work.

```
/**
 * Motion Controller class
 *
 * This class defines a top-level motion profile, which can act as an intermediate between
 * a subsystem class and the motors themselves
 *
 * This takes the constants kS, kV, kA, kP, kI, kD, max_v and acceleration and wraps around
 * a feedforward, PID and trapezoid profile. It does so with the following formula:
 *
 * out = feedfoward.calculate(motion_profile.get(time_s)) + pid.get(motion_profile.get(time_s))
 *
 * For PID and Feedforward specific formulae, see pid.h, feedforward.h, and trapezoid_profile.h
 *
 * @author Ryan McGee
 * @date 7/13/2022
 */
class MotionController : public Feedback
{
    public:
```

After every commit to the Core repository, we use Github Actions to automatically regenerate HTML documentation and deploy it to the Core website at **rit-vex-u.github.io/Core/**. This ensures that any documentation programmers wish to reference is always available and never out of date. Doxygen provides industry standard documentation styles and enables a much quicker learning experience than slowly poking through files.



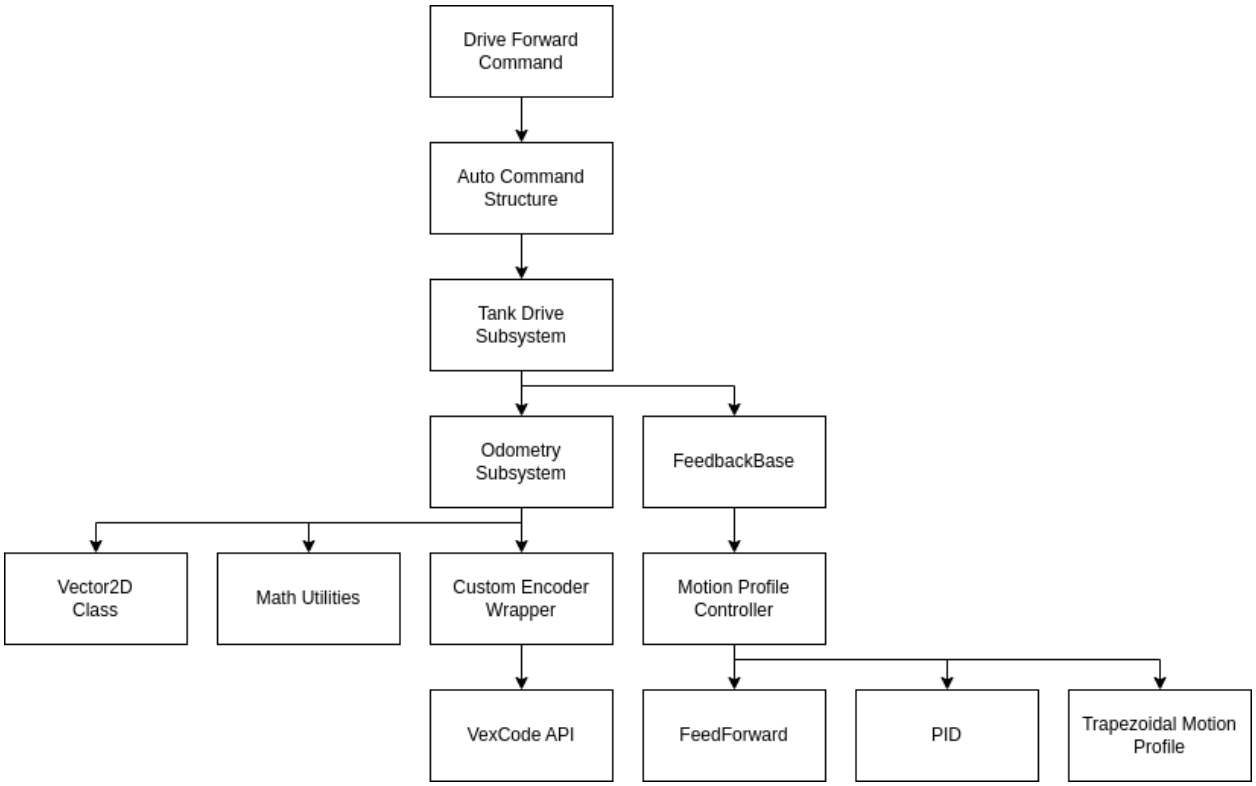
The Core Wiki contains information and tutorials to help people who are new to the ecosystem. This includes tutorials for project setup, subsystem overviews and explanations, and example usage code. For utilities, there are mathematical formulas and theory, as well as tuning tutorials for control loops. The Wiki can be found at ***github.com/RIT-VEX-U/Core/wiki***



Section 2: Software Stack

Overview

The Core API uses object-oriented programming and multiple layers of abstraction between hardware and high-level software, creating a software "stack". For example, a simple autonomous "Drive Forward" command would look like this:

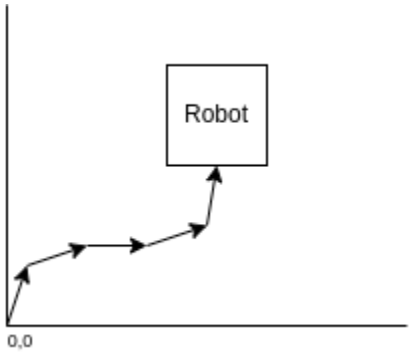


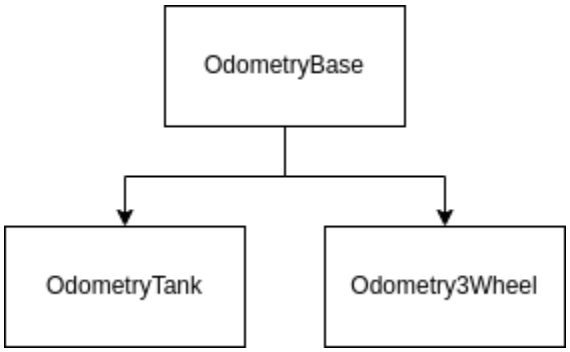
Each subsystem is built on top of another, which compartmentalizes tuning and simplifies development for higher-abstraction and more complex code.

Our subsystems are also designed to be "Set and Forget", using asynchronous programming to have all the subsystems run on separate threads. This not only allows us to run multiple subsystems at the same time during the autonomous period, but also simplifies general usage (not having to run an update() function), and speeds up execution on the main thread.

Odometry

In order for the robot to drive autonomously, it needs to know where it is, and constantly monitor changes to sensors. The Odometry subsystem takes inputs from encoders, and using vector math and previous position data, calculates the position and rotation of the robot on the field as a point in space (X, Y), and heading (deg).





The Odometry subsystem is broken down into an OdometryBase class, which controls the asynchronous behavior and getters/setters, and OdometryTank and Odometry3Wheel classes, which both extend OdometryBase and implement a two-encoder algorithm and a three-encoder algorithm, respectively.

Drivetrain

A drivetrain class has two functions: To control the robot remotely, and autonomously. In the Core API, the TankDrive class allows the operator to control the robot using Tank controls (Left stick controls the left drive wheels, right controls the right), and Arcade controls (Left stick is forward / backwards, right stick is turning). This means drivers can tailor their controls to whichever feels more natural.

For autonomous driving, the TankDrive class has multiple functions:

- drive_forward():
    - Drive X inches forward/back from the current position
- turn_degrees():
    - Drive X degrees CW/CCW from the current rotation
- drive_to_point()
    - Drive to an absolute point on the field, using odometry
- turn_to_heading()
    - Turn to an absolute heading relative to the field, using odometry

Generally, it is better to use drive_to_point and turn_to_heading to avoid compounding errors in position over relative movements. These functions implement the FeedbackBase class, so any control loop can be used to control it.

Control Loops

In order for the Autonomous Command Structure to function, we need a way to tell the robot how we want it to move. There are two broad categories of telling a robot to achieve a requested position - Feedback and Feedforward. Feedback relies on sensors and adjusts the output of the robot according to the error between where it is and where it wants to be. On the other hand, a feedforward controller takes a mathematical model of the system and creates outputs based on what it calculates to be the necessary output to achieve the goal. These controller types, Feedback and Feedforward, work for many applications but a combination of them can achieve an even better control over robot actuators.

PID

A PID controller is perhaps the most common type of Feedback control. It uses measurements of the error at its current state (proportional), measurements of how the error was in the past (integral) and measurements of how the error changes over time(derivative). The controller acts accordingly to bring the errors towards 0. We implemented a standard PID controller but made some alterations to fit our needs. The most important of these are custom error calculations. The standard error calculation function *(target - measured)* works for many of our uses but causes problems when we use a PID controller to control angles. Since angles wrap around at 360 degrees or 2π radians we wrote our own error calculation function that gives the error that accounts for this wrapping.

Feedforward

A feedforward controller differs from a feedback controller in that it does not rely on any measurement of error to command a system. Instead, built into a feedforward controller is a mathematical model of the domain. When a target is requested by the controller, the model is queried to figure out what the robot actuators must output to achieve that target. A key advantage of this form of control is that instead of waiting for an error to build up in the system, the controller acts directly to achieve the target and can reach the target much faster.

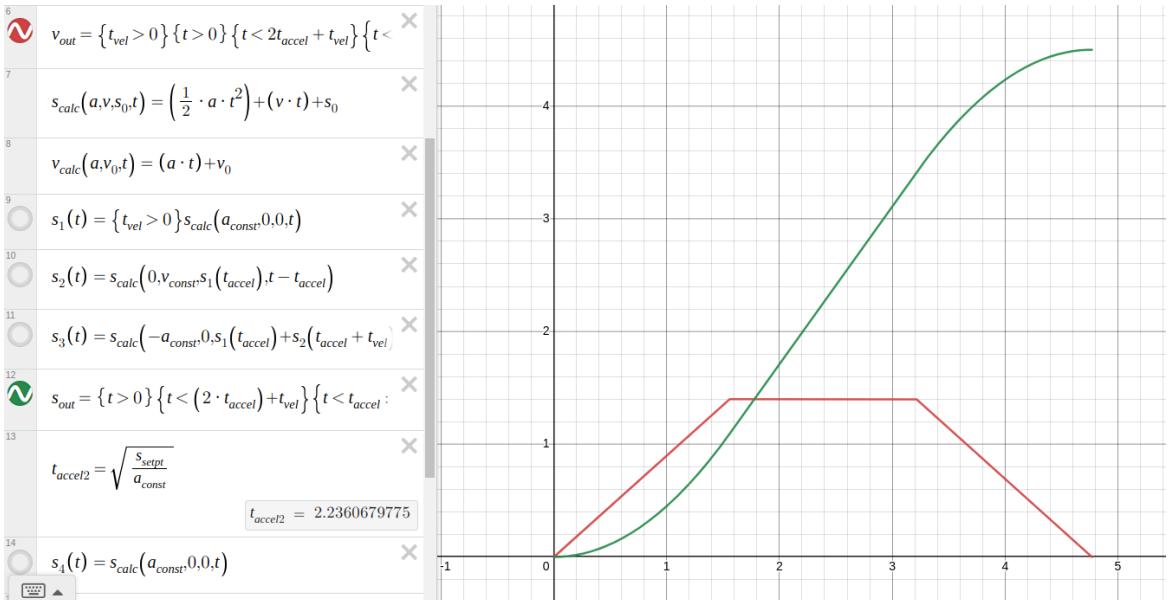
Generic Feedback

Different control systems work best in different environments. Because of this, we found ourselves switching control schemes often enough that rewriting the code each time was time consuming and often led to rushed, worse quality code. To solve this problem we implemented a generic feedback interface so that none of our subsystem code needs to change when we use a different control scheme. Instead, the subsystem reports to the controller where it wants to be,

measurements from its environment and some information about the system's capabilities and the controller will report back the actions needed to achieve that target. This allows for much faster prototyping and cleaner, less tightly coupled code.

Motion Profile

In past years we have used a simple PID controller to control robot position but this gave only a very simple method of control. We found limitations in its ability to specify speed, its inability to respond to wheel slipping, and its slow response time. With all our subsystems requiring only a generic feedback mechanism, we could develop a better controller without interfering with other projects on our team. Our research led us to create a motion profile controller - an exact system of position, acceleration, and velocity controls that enables our controller to extract the maximum performance of our robot systems. Now that we can specify an acceleration, we can measure how quickly our robot can accelerate without slipping and always accelerate the optimal way.



With a pure PID controller, the robot only acts once there is a difference between the measurement and the target while with our new motion controller we can act ahead on time with our commands instead of lagging behind. With a pure feedforward controller, the robot could not adapt to new situations it finds during competition. With our fusion of these control schemes using our Motion Profile controller, we can take the advantages of both of these control schemes.

Auto Command Structure (ACS)

A new addition to our core API this year was that of the Autonomous Command Structure. No more will our eyes glaze over staring at brackets as we trawl through an ocean of anonymous functions nor lose our way in a labyrinthine state machine constructed not of brick and stone but blocks of ifs and whiles. Instead, we provide named Commands for all the actions that our robot can execute and infrastructure to run them sequentially or concurrently. The API is written in a declarative way allowing even programmers unfamiliar with the code to see a step by step, annotated guide to our autonomous path while keeping the procedures of how to execute the actions from hurting the readability of the path.

```
CommandController auto_non_loader_side(){
    int non_loader_side_full_court_shot_rpm = 3000;
    CommandController non_loader_side_auto;

    non_loader_side_auto.add(new SpinRPMCommand(flywheel_sys, non_loader_side_full_court_shot_rpm));
    non_loader_side_auto.add(new WaitUntilUpToSpeedCommand(flywheel_sys, 10));
    non_loader_side_auto.add(new ShootCommand(intake, 2));
    non_loader_side_auto.add(new FlywheelStopCommand(flywheel_sys));
    non_loader_side_auto.add(new TurnDegreesCommand(drive_sys, turn_fast_mprofile, -60, 1));
    non_loader_side_auto.add(new DriveForwardCommand(drive_sys, drive_fast_mprofile, 20, fwd, 1));
    non_loader_side_auto.add(new TurnDegreesCommand(drive_sys, turn_fast_mprofile, -90, 1));
    non_loader_side_auto.add(new DriveForwardCommand(drive_sys, drive_fast_mprofile, 2, fwd, 1));
    non_loader_side_auto.add(new SpinRollerCommand(roller));

    return non_loader_side_auto;
}
```

Flywheel

Another new addition to the Core API with this year's game is the Flywheel subsystem. This controller allows for many different methods of control loops:

- Bang-Bang
- Feedforward
- PID
- Feedforward+PID
- Take-Back-Half

This allows us to test many different types of control and use one that works best in our situation.

The Flywheel subsystem is another "Set and Forget" class, automatically updating feedback loops with new sensor values, and accepting new RPM targets asynchronously.

Other Subsystems

Roller

The roller mechanism has gone through a few different iterations, with the most recent being a passive mechanism, where the robot drives forward and backward into the roller to change its scored color. To control this in code, we use our DriveForwardCommand twice to drive forwards (scoring the roller), and drive backwards to disengage. We then use sensors to detect whether the roller has been scored, or if we need to loop and do it again (See "Roller Sensing with Vision", pg 26).

String Launcher

During the endgame, a pneumatic cylinder will actuate a string launcher, expanding the robot to cover as many tiles as possible. To avoid early actuation (and an illegal expansion), a safeguard was put in place via controls. The operator must press two buttons on opposite sides of the robot at the same time to trigger the solenoid.

Section 3: The Development Process

Overview

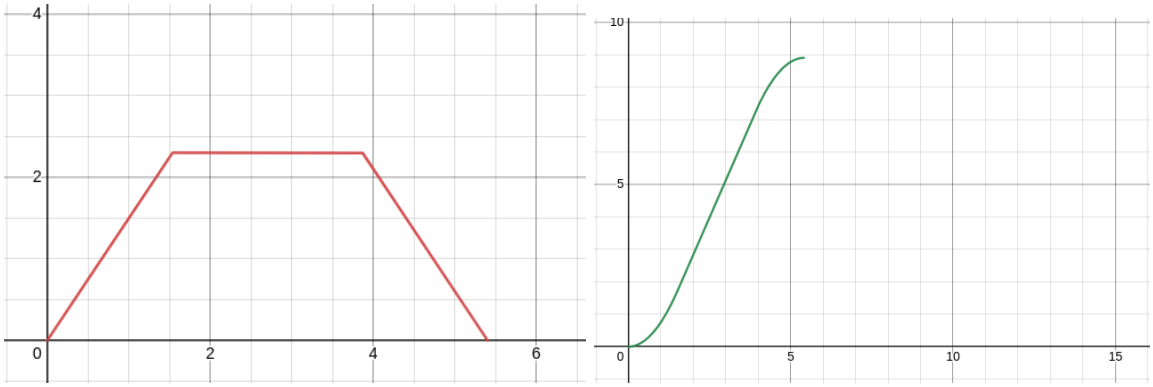
Every year brings new improvements to our API, as students work on summer projects and as the software team is waiting on our new robot design. At the end of each season, we set goals of new features that can give us an edge in the new season. This year, we resolved to explore motion profiles, improve our autonomous coding methods, add support for more odometry options and improve our methods of tuning subsystems.

Motion Profiles

The first step to developing this system was to define what we wanted the robot's autonomous driving to look like. The ideal profile is a smooth acceleration, followed by a constant velocity and a final deceleration back to a stop. In order to model this, we need the following variables:

- Robot's Acceleration (units/s^2)
- Robot's Maximum Velocity (units/s)
- Robot's Position Setpoint (units)

From there, you can use kinematic equations to determine the amount of time each section of the profile will take. The final velocity/time (left) and position/time (right) graphs look like this:



Because of the velocity graph, the name of the mathematical model is called a "Trapezoidal Motion Profile." These initial tests were modeled in Desmos given the variables listed above as sliders. The model can be seen at ***desmos.com/calculator/lcxoilnvfv***

The next step was to translate this into code, so that at any given time *t*, a position, velocity and acceleration could be output. This was done by parameterizing the kinematic equations as macros, and calculating the variables for each time period (Accel, MaxV, Decel).

```
// Kinematic equations as macros
#define CALC_POS(time_s,a,v,s) ((0.5*(a)*(time_s)*(time_s))+((v)*(time_s))+(s))
#define CALC_VEL(time_s,a,v) (((a)*(time_s))+(v))
```

The final step for implementing motion profiles into our codebase was to create a "Controller" that couples the kinematic equations to a control loop. In this case, a combination Feedforward and PID was chosen, since it can be configured to accurately model the properties of a motor, through the equations:

$$feedforward = (kS * signum(V)) + (kV * V) + (kA * A)$$

$$PID = (kP * error(t)) + (kI * \int error(t)dt) + (kD * \frac{d\,error}{d(t)})$$

$$voltage = feedforward + PID$$

Finally, this was packaged in a class, extending the Feedback interface, allowing any existing code to use this implementation.

Odometry

Last season was the first time our software team used an odometry subsystem, and the only supported hardware was either 2 drive motors (left and right), or 2 encoders, with an optional IMU. While this worked for our hardware last year, it has limitations in that any lateral motion could not be tracked. Our goal this past year was to start supporting a third encoder, to start tracking this third degree of freedom. Now, if the robot slides from quick turns or is pushed by an opponent, tracking remains true.

To implement this system, the existing odometry system had to be split into a super class, OdometryBase, and its subclass OdometryTank. OdometryBase was created to handle shared functionality, such as asynchronous updates and getters/setters, simplifying the creation of new forms of odometry for the future. From there, Odometry3Wheel inherited this OdometryBase class with a different algorithm to translate 3 encoder inputs into a (X,Y,Rot) pose relative to the field.

During development, some drawbacks were found. While this setup is in theory more accurate than the older OdometryTank, it is significantly harder to tune. Any inaccuracy in tuning the third tracking wheel will throw off the result, and tuning this value is difficult. To mitigate this, many tuning "helper" functions were created to simplify the tuning process.

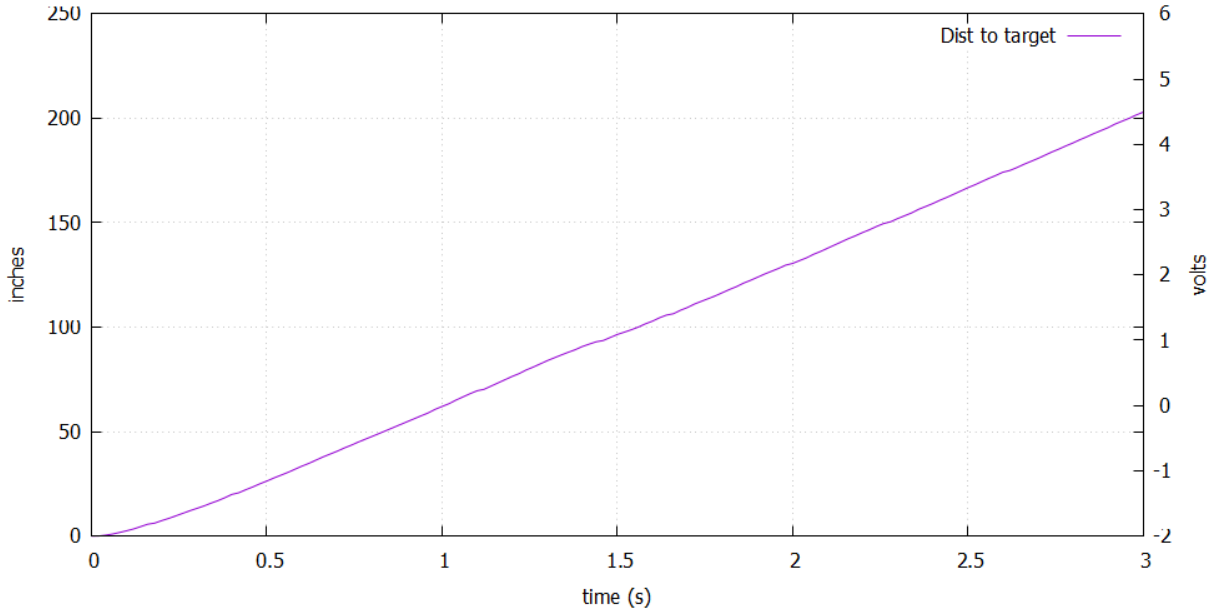
Tracking Tuning Process

No matter what controller or odometry setup a robot uses, it is useless without accurate tuning based on real world measurements. At the beginning of the season we worked on tuning our new systems by hand, carefully measuring real world variables and repeatedly testing the new systems. While the knowledge gained from this tuning was imperative to debugging and gaining knowledge about our new systems, it was a very fragile and frustrating experience. As the season progressed, with the knowledge we gained from earlier tests, we crafted guided tuning that used the VEX controller to instruct an operator on what to move and measure and output the optimal parameters for our subsystems.

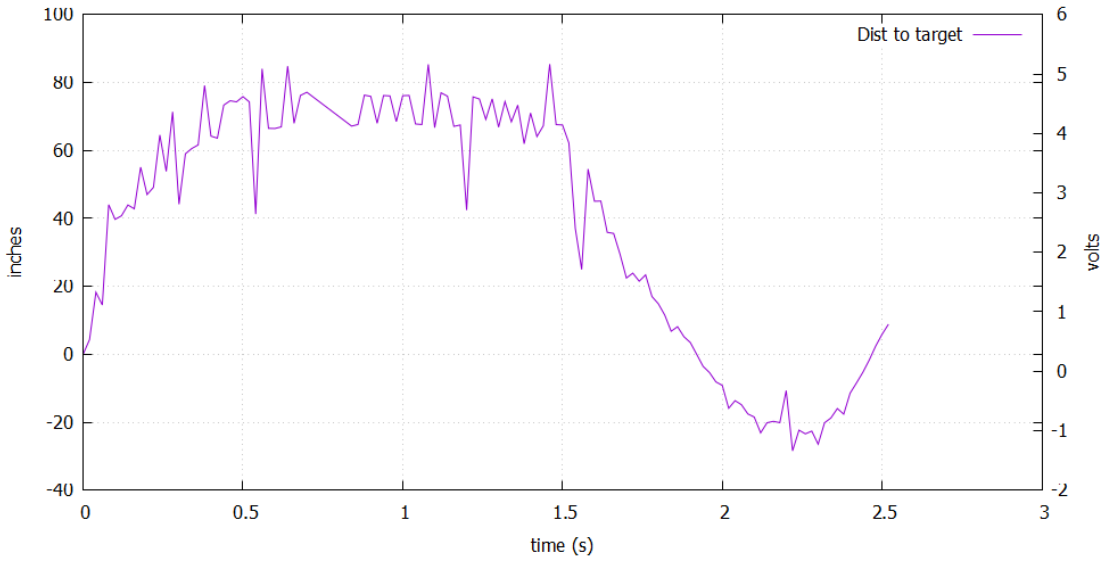
Drivetrain Tuning Manually

One of the first software projects we tested was our new Motion Profile system as applied to the drivetrain. Since what type of drivetrain we planned on using was determined early on, the software team was quickly able to get a functioning drive system on which to test our technical schemes.

The first step was to estimate the limits of our robot. Through in-code calculations as well as standalone analysis tools we determined a rough guess as to the physical limits of that drive train.

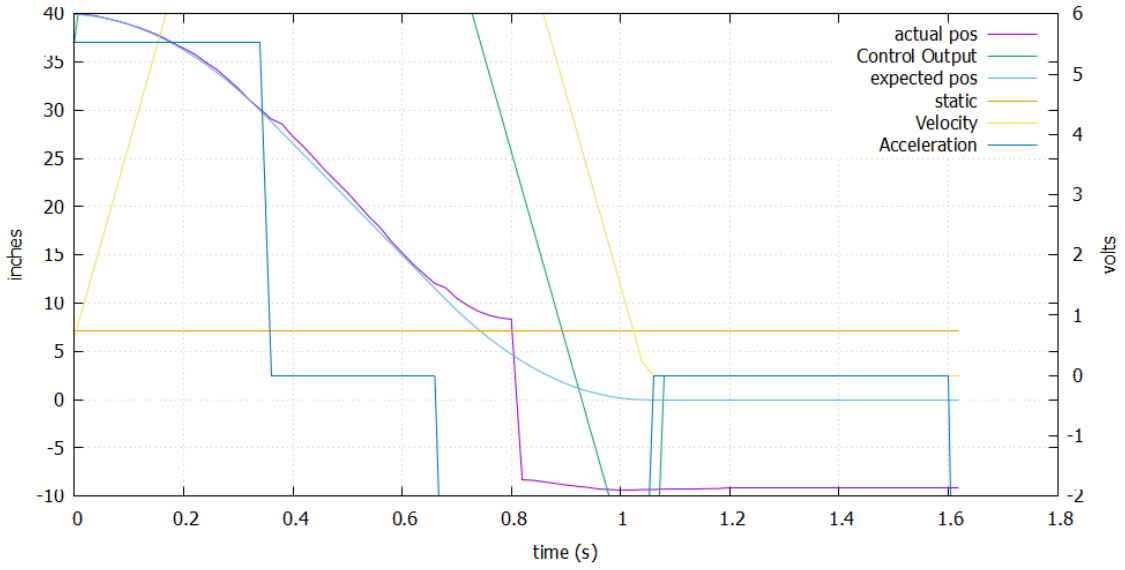


Distance covered at max motor output



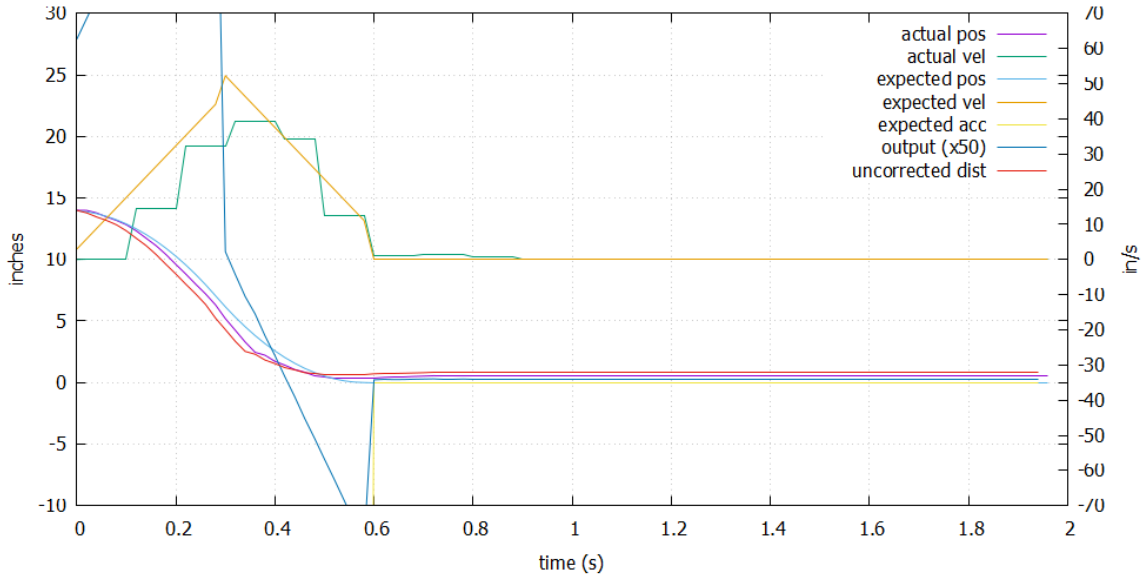
Velocity measured at max motor output

The second step was to use these parameters to drive the feed forward control of the Motion Profile and tune these parameters to get the best possible results based on only feed forward.



A first attempt at using pure feedforward to control robot position

During this tuning step we discovered an error in our process that led to a poor estimation of position when near the target. This was due to an error in the way we transformed 2 dimensional position measurements into a one dimensional input to a Motion controller. (This can be seen by the steep drop off just after 0.8 seconds in Fig 1.3)



Correct input to the Motion controller and better feedforward parameters

Finally, we used this setup to apply standard PID tuning to deal with deviations from the ideal that a feedforward controller cannot anticipate.

Drivetrain Tuning Helpers

While one can do the classic "slowly raise the constant" method of tuning control loops, it quickly becomes infeasible when a motion profile needs tuning for the constants kS, kV, kA, maxV, accel, kP, kI, kD, deadzone, and onTargetTime. Thus, we created helper functions designed to guide a programmer through the process of tuning, outputting the calculated tunings.

The tuning functions start by printing out to the controller screen instructions, such as "Move the robot forward 100 inches" or "Turn the robot in place 5 times." While doing so, the robot monitors sensors and solves equations to output accurate tunings, which can be directly plugged into the code. Some functions are completely autonomous, such as finding the robot's static friction constant (kS), or maximum velocity (maxV).

For tunings that require more active support, the robot will complete simple autonomous tasks like "Drive Forward 24 inches and stop" or "Turn 90 Degrees then Stop." The robot will print out sensor debug information that can be graphed and analyzed by the programmer. Autonomous PID tuning was attempted using the Ziegler–Nichols method, however the tunings were poor and the idea was scrapped for now.

Generic PID Tuning Helper

As we tuned assorted subsystems, we found that a limiting factor was how often we had to rebuild our code to change a PID constant. This took ~5-10 seconds and on meetings where we changed constants probably hundreds of times this delay added up. To improve this system, we wrote a generic PID tuner. This tuner accepts input from the VEX controller and displays graphs of sensor readings and outputs on the brain screen which allows a programmer to iterate through possible tunings in fractions of the time it took before and is as simple as calling one method with a pointer to a PID controller alongside the tuning helper function mentioned above. The graphing API was written from scratch this season and we hope to refine and use it more generally in the future to aid in situational awareness for both drivers and programmers.

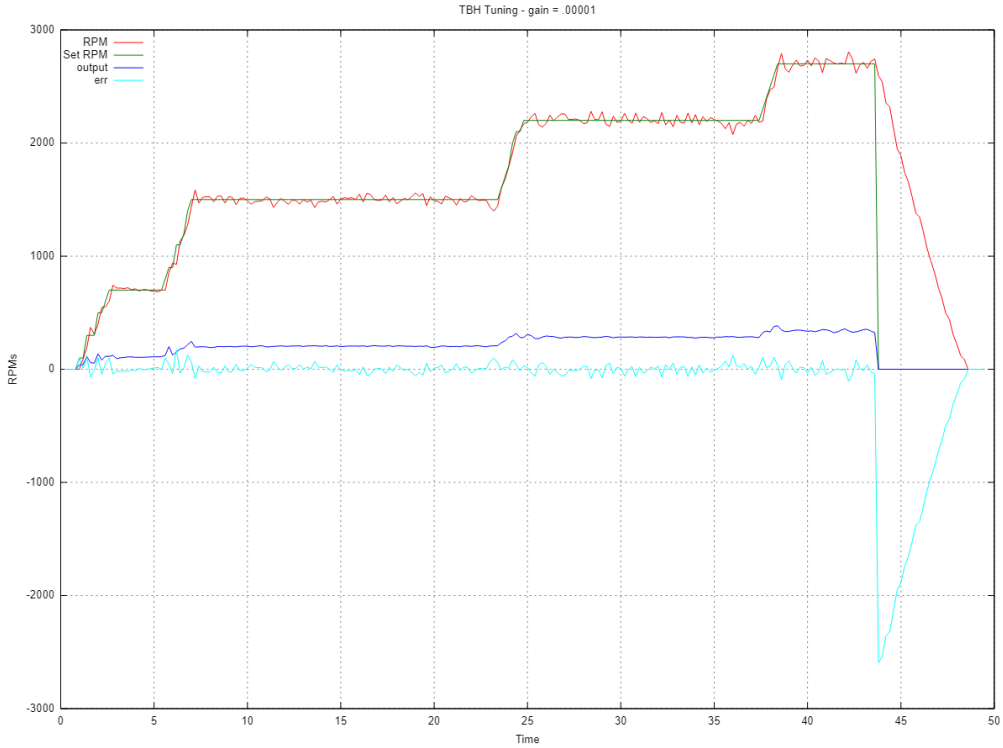
Manual Flywheel Tuning

Another part of the robot we were able to test without the entire system was the flywheel. This tuning process was slightly different from drivetrain tuning due to the fact that we are only tuning for velocity and there are no transformations from measurements to input variables to the controller.

Bang-Bang

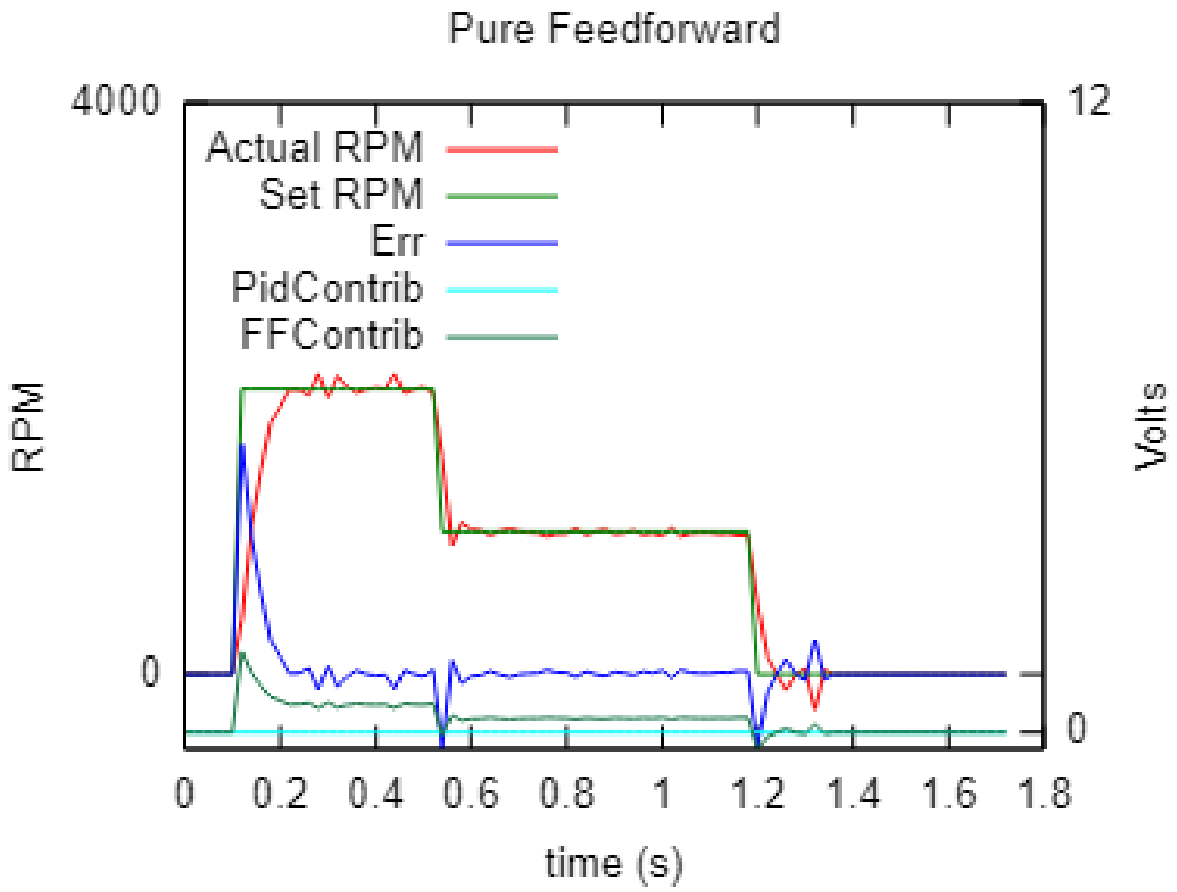
Bang-Bang control was a pleasant thought as we constructed our flywheel and wanted a quick way of reaching velocity. However, as we sought further accuracy, it became too unstable for our uses.

Take Back Half



The peak of our Take Back Half control tuning.

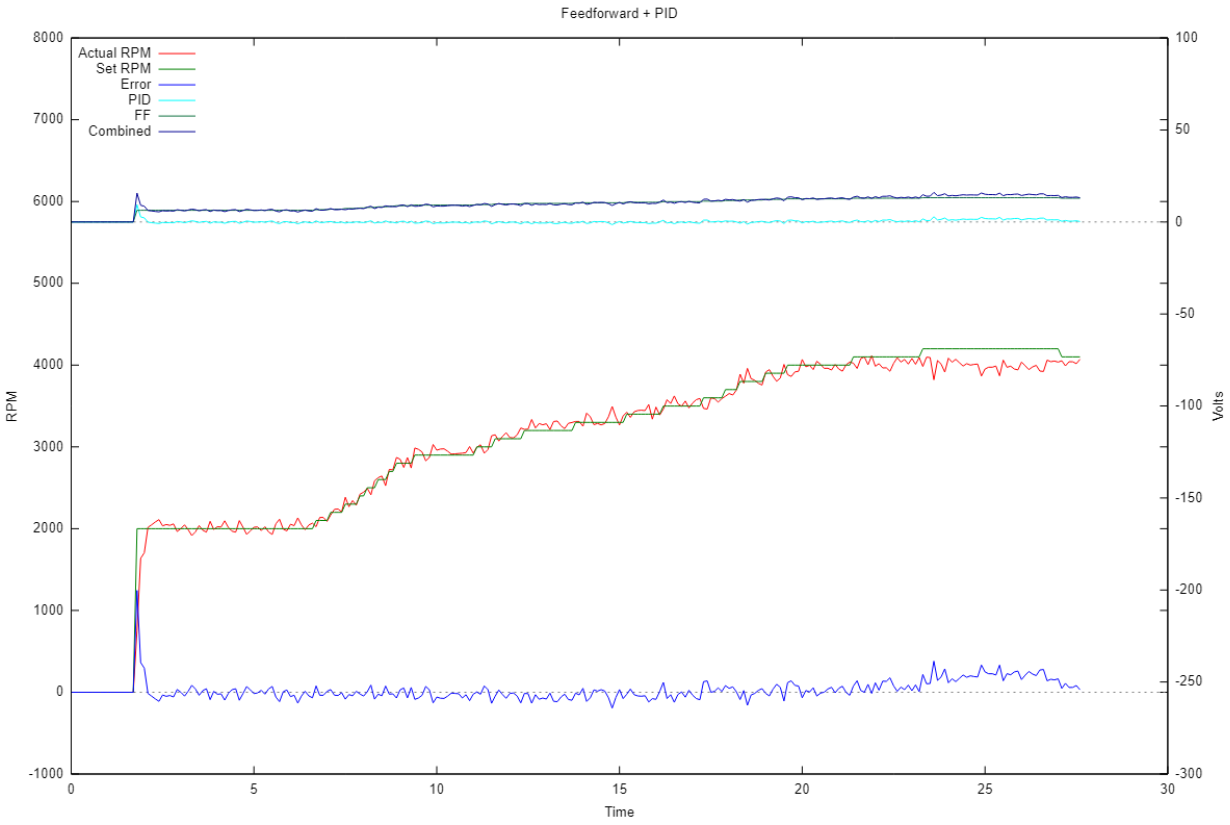
Take Back Half control was a new scheme we introduced into our codebase this year. It is a feedback control scheme applied to velocity that requires only one tuning parameter. For this simplistic tuning process it provided excellent results however we found a feedforward and feedback controller combination to work slightly better - especially when rapidly changing the target velocity.



Pure feedforward control

On our journey to our final control scheme, we tuned a pure feedforward controller for our flywheel. This worked surprisingly well even without any feedback control as our theoretical model was very close to the actual system. However, we knew we could get more precision with some amount of error correction.

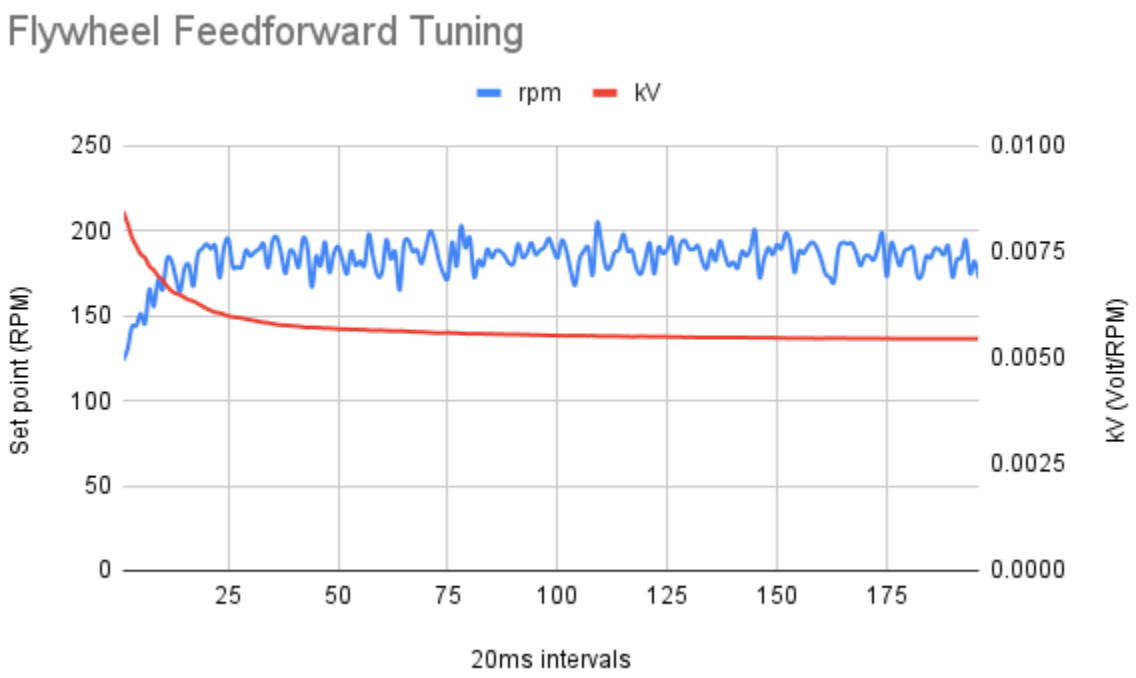
Feedforward + PID



A well tuned Feedforward and PID fusion

At last we reach the final form of our flywheel tuning. Combining our Feedforward for general command of the flywheel as well as a PID controller to resolve any remaining error. This tuning was more complicated than other forms but the results were very impressive. The complexity of tuning also pushed us to create "helper functions" to speed up the tuning process.

Flywheel Tuning Helpers



Finding kV automatically

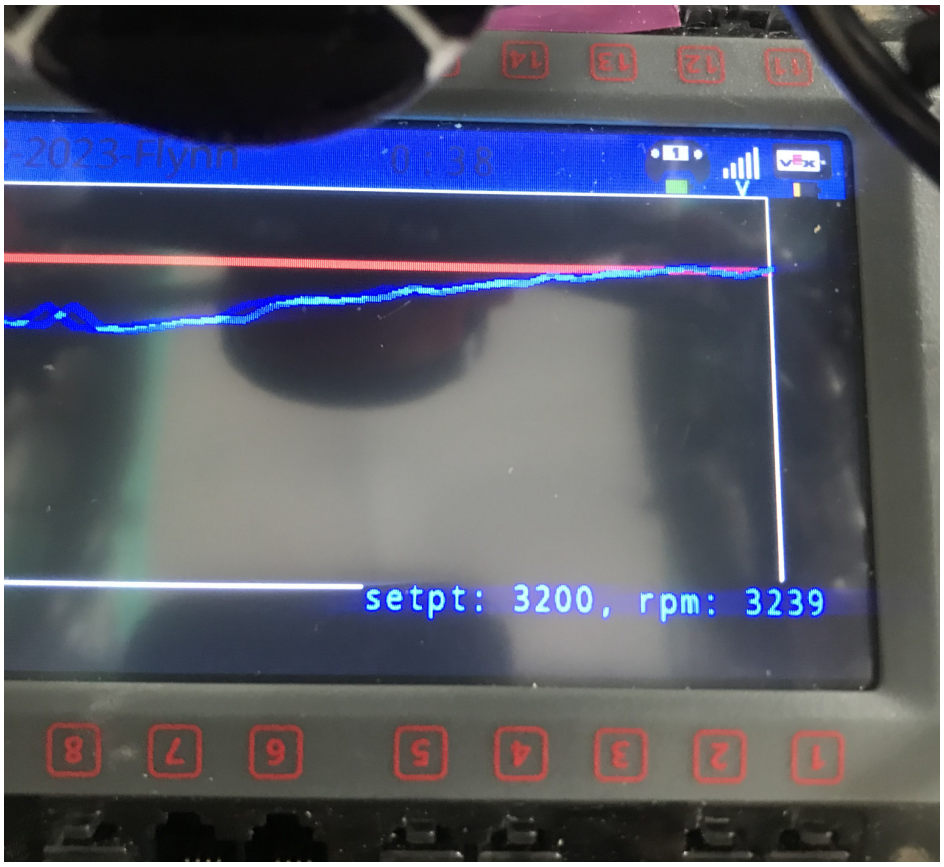
With the new addition of the Flywheel class, we needed tuning helpers to assist us. These work similarly to the drivetrain helpers above, but outputs more detailed information and allows direct control of the flywheel's RPM during tuning. These helpers are much more useful while working on Feedforward applications, as Feedback still requires manual tuning.

Screen Helpers

When programming, subsystems, building paths, or debugging robot code, the single greatest help is data. This data can be sensor measurements over time, odometry position information, or arbitrary program variables that need inspection to debug a problem. The VEX controller's serial output to VEX Code is helpful but we quickly ran into issues of bandwidth. The controller to robot serial output runs at 115200 baud, meaning we could observe about 12800 bytes per second. With the robot main loop running at 50 hertz, this means the serial terminal could receive about 256 characters of data each iteration before characters are dropped and the output text becomes a garbled mess. When tuning or debugging a multivariate program, this is

simply not enough. Additionally, while text output is better than viewing raw bytes, it is often not the best way to conceive of robot state. So, we wrote a few tools to utilize the VEX Brain screen as our output medium.

Graphing API



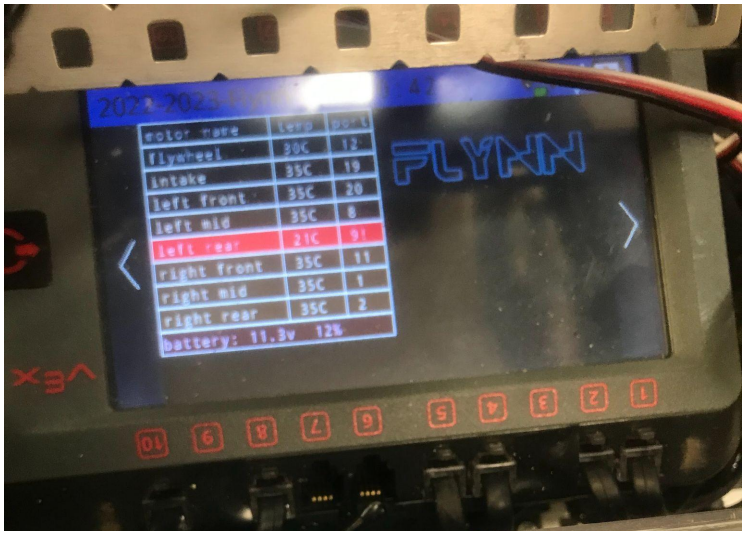
A graph of Flywheel Set Point and RPM used to improve recovery time

Many of the values we wish to examine while debugging or planning are simple value vs time measurements. While a numerical output means something, seeing the past gives a lot more intuition. Especially when measuring time to achieve a target, seeing the journey the value takes gives programmers the ability to reason about the system. This graphing API allows us to examine these values in time.

Motor Statistics

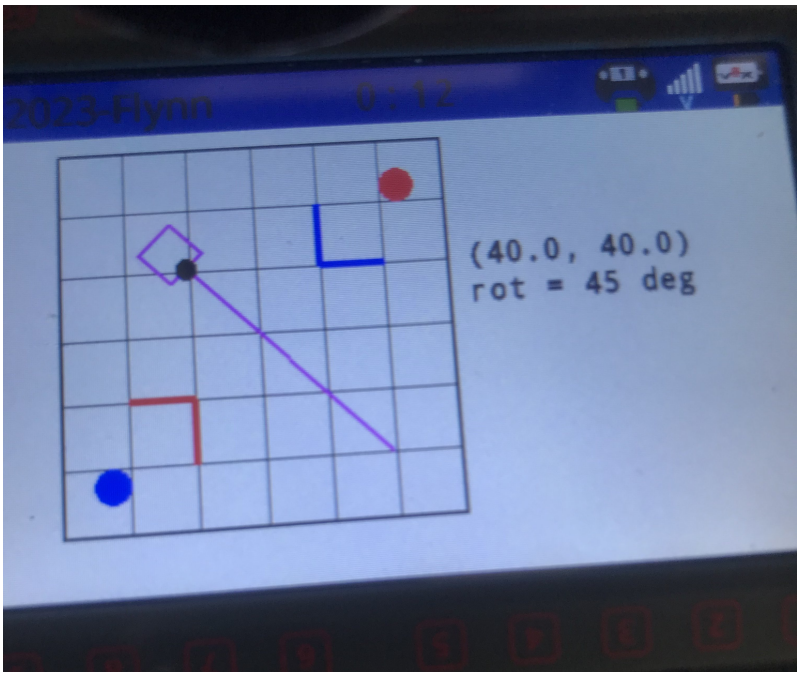
One would think it an easy step to remember to plug motors in, and yet multiple times this season we have been bewildered and hindered by an unplugged motor. This tool was built to continuously display that the motor had been unplugged and was not cancellable like the

built-in VEX alert. This screen also displays what port to plug it into as well as a color coded temperature displaying when the robot needs to cool down. This tool proved extremely useful as we discovered an alarmingly high number of dead or nonfunctioning ports on the brain.



The motor statistics screen warning about an unplugged motor

Odometry Map



The robots perceived position on the field

The skills and autonomous paths we created rely on position feedback to move and remain resilient in the face of slightly differing field conditions. However, from a human point of view, outputs such as `(25.6, 12.8) angle = 115 degrees` do not give the best intuition for where a robot thinks it is. Instead, a simple map that draws the robot in its perceived position and orientation gives operators a much better idea of where the robot believes it is and where it should go when planning autonomous and skills paths.

Vision Chooser

A vision configuration selected at the start of a match

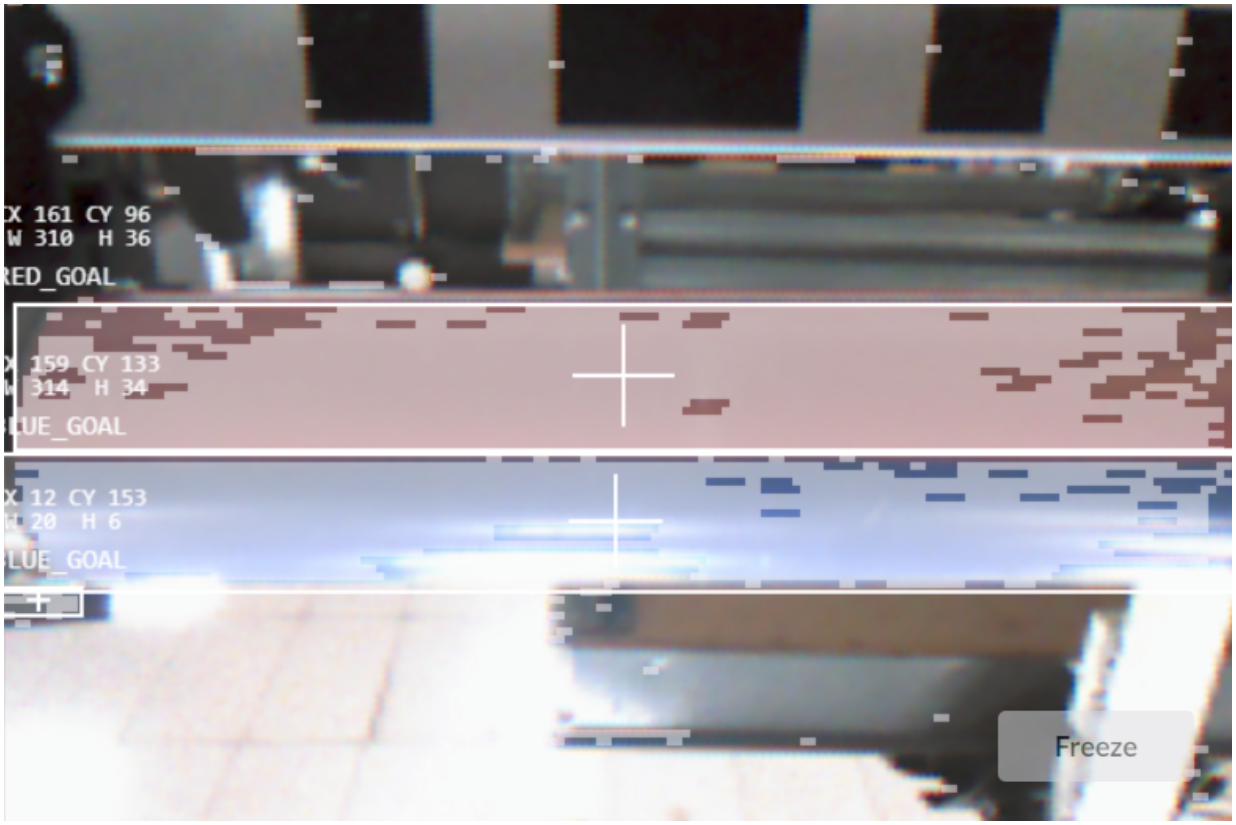
After our first competition, it became clear that for any cross field shot to have a chance of scoring a high goal, a more precise targeting system was needed. We implemented color goal detection but needed a way to specify if the robot should use vision or fall back to odometry targeting and which colored goal should the robot aim for. This screen provided a simple interface for us to choose these parameters before a match or skills run rather than hardcoding it into the robot code making it difficult to change at a moment's notice.

Screen Subsystem

As we wrote more and more of these tools, we realized that we needed a way to handle which tool was reporting at any given moment. This was solved by a simple manager class that accepts an input of drawing functions and creates a slideshow of all of them that the user can scroll through to discover many different insights about robot performance. The class also makes sure that no screen is drawn when it is not visible which keeps performance of the system high and limits degradation of robot performance.

Roller Sensing With Vision

An issue we found while at competitions is that the rollers are very different on every field, so we needed a way to sense whether or not the rollers were scored. To do this, we used the vision sensor to detect the red and blue pattern. For example, if we see no color, only red or only blue, we know the roller is neutral. If we see both colors and red is above blue, we know red is scored on the roller (and vice versa for blue).



Roller color sensing - camera inverted

Achieving Robustness

It is every programmer's dream that everything works first try and works in every environment on every machine. Unfortunately, reality has a bothersome way of stepping on that dream. In the context of VEX, differing battery voltages, motor temperature, metal or portable field, or any different field even if the same type and many other factors determine the performance of an autonomous program. As well, on our robot specifically, different mechanisms worked differently throughout the season up to days before the event in some cases. This called for a robust and generalized method of control for the autonomous match

section and skills section. Many of these are detailed above including odometry for corrective actions on driving motions, a myriad of control systems for flywheel control that allow us to make shots amidst constantly changing gears, weights and geometries with less input from the programmer. In addition to these control schemes, we developed ways for the autonomous programs to be reactive to their environment.

Timeouts

Depending on the field and battery condition, some driving commands have the chance of not completing. An example of this is when the tape marking the center line of the field came slightly unstuck from the field tiles and caught the corner of our robot. Originally, this completely halted our autonomous path usually leading to catastrophically fewer scored points. After realizing this flaw, we implemented timeouts, that specify a number of seconds for which a command can run before it is canceled. Cancellation required us to make an addition to our auto command that allows a command to specify what needs to be done when it is stopped. Although we had to specify some extra behavior, the automatic cancellation of commands allowed the autonomous robot to move around the obstacle and continue so even if it missed some points, it could work around it and score others.

Cancel Functions

Sometimes, a simple timeout is not enough and something more sophisticated is needed. For example, a primary way of scoring points in Spin Up is the endgame. However, the endgame can only be triggered in the last ten seconds unless one wishes to forfeit time and the ability to score points. Since certain actions such as spinning rollers or waiting for a flywheel to spin up to speed can take variable amounts of time, planning autos was often a tradeoff of doing more in a skills program but missing the ability to score endgame points and scoring less disk points but being sure that the endgame points would happen be scored. What we really wanted was a way to leave out sections of the path if we were running late. However, this conflicted with our quality of life improvements that created a path from a list of commands rather than running a mess of custom code meant to handle these situations. The compromise we came to was a `with_early_end` modifier. This modifier takes a command in our list and a function that will be evaluated at runtime to determine if we should stop and try something else. This gave us a solution to the previously presented problem as when we were running commands that might cause the endgame to not fire, we can guard them with an early end that will cancel the function if there are fewer than five seconds left in the match and skip to the commands that operate and maximize points for the endgame.

Wiring Fault Tolerance

Although it is rare, a wire coming loose is a situation not entirely out of the realm of possibility. This season, three separate times a faulty cable has cost us at least half of our auto or skills paths. For a wire lost in the drivetrain, our drive system code will do its best to reach the

target with the remaining motors. For a wire in our flywheel or intake, there is unfortunately not much we can do. The place where this faulty cable affected us the most is the connection to the inertial measurement unit. After a forensic investigation of a few matches in which our drive system executed seemingly nonsensical maneuvers, we realized that when we began a turn, the centrifugal force had a tendency to pull the cable out of the IMU socket. While there are mechanical ways of fixing this problem, for safety, we decided to implement a fallback such that if the IMU that our odometry uses for orientation becomes unplugged, we drop back to wheel based angle calculation. This is a slightly less accurate system of pose estimation but is predictable and will not create uncontrollable or undefined behavior.

Vision Aim Fallback

While attempting to tune our odometry and pose estimation to perfection, we realized that there will always be situations where the ‘dead reckoning' style of odometry becomes off for some reason outside of the control of the robot. For this reason we utilized the VEX Vision sensor to target the blue and red color signature of the high goal. However, as we discovered the difficult way at a competition, there are very few rules governing the actions of people outside of the drive boxes or for the environment around the field. This led to a situation where our robot got a lock on a passerby and launched a disk away from the goal. In order to prevent this rather unfortunate situation, we created an angle limiter such that if the vision aim leads the robot off of the point where we reckon the goal is by too far we assume we have a lock on a non goal object and return to our best odometry guess.

Roller Vision Fallback

For the same reasons as vision aim, using vision to check if rollers are scored can be faulty in certain environments. As a workaround, we implemented a simple number selector widget. Using that, we can set the default behavior of our vision path by turning off roller vision and specifying a number of times to blindly hit the roller based on how the rollers at that field feel. This came in very handy especially at multi field events when we could lean over, poke a button on the screen and have a semi-reliable roller even with no sensors.

References and Links

Core API

- https://github.com/RIT-VEX-U/Core

Core API Wiki

- https://github.com/RIT-VEX-U/Core/wiki

Core API Documentation

- https://rit-vex-u.github.io/Core/

Git Subrepo

- https://github.com/ingydotnet/git-subrepo

Trapezoid Profile Calculations

- *desmos.com/calculator/lcxoilnvfv*