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

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.

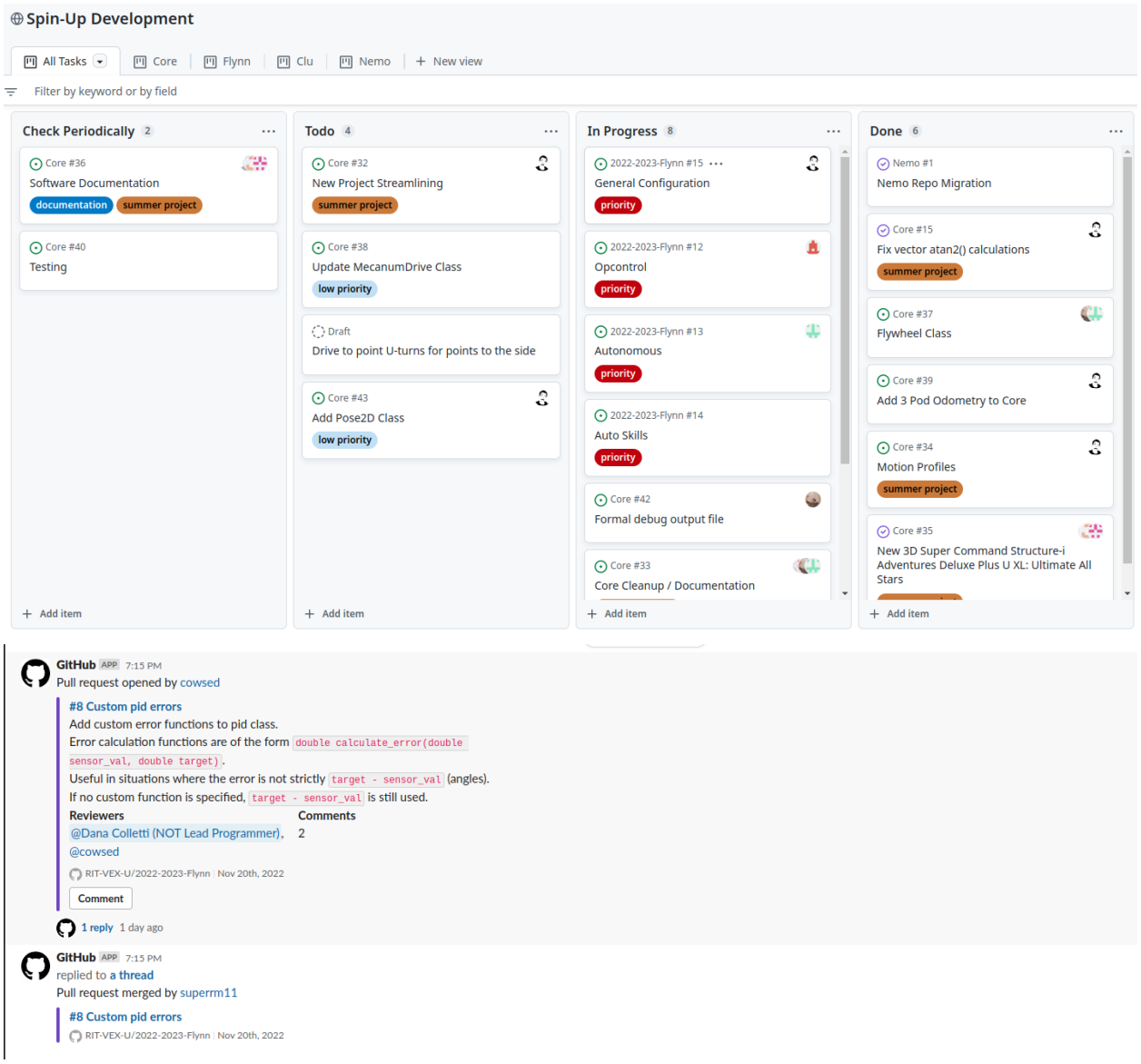
Git Subrepo

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.

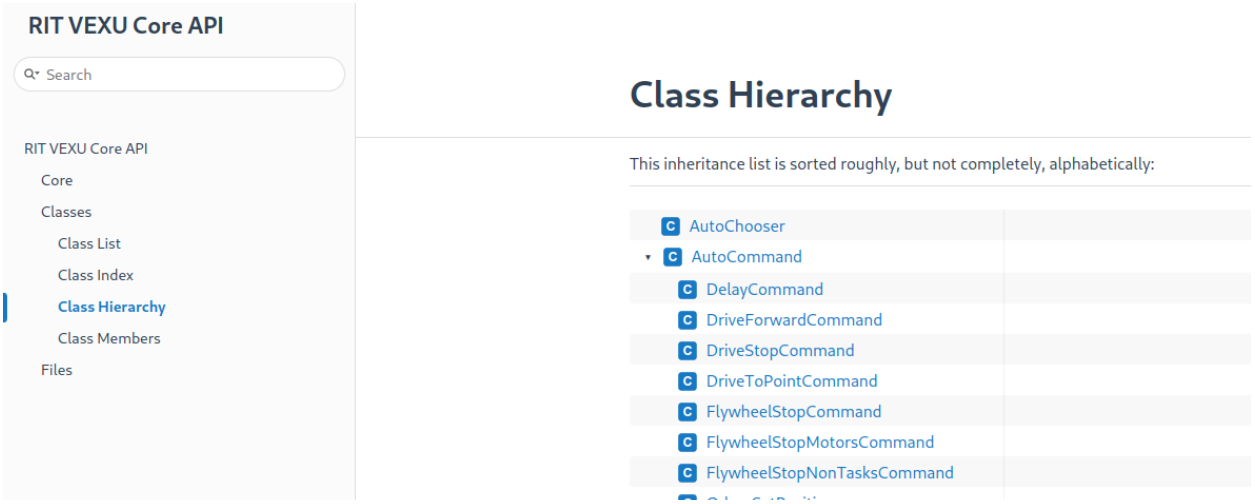


Documentation

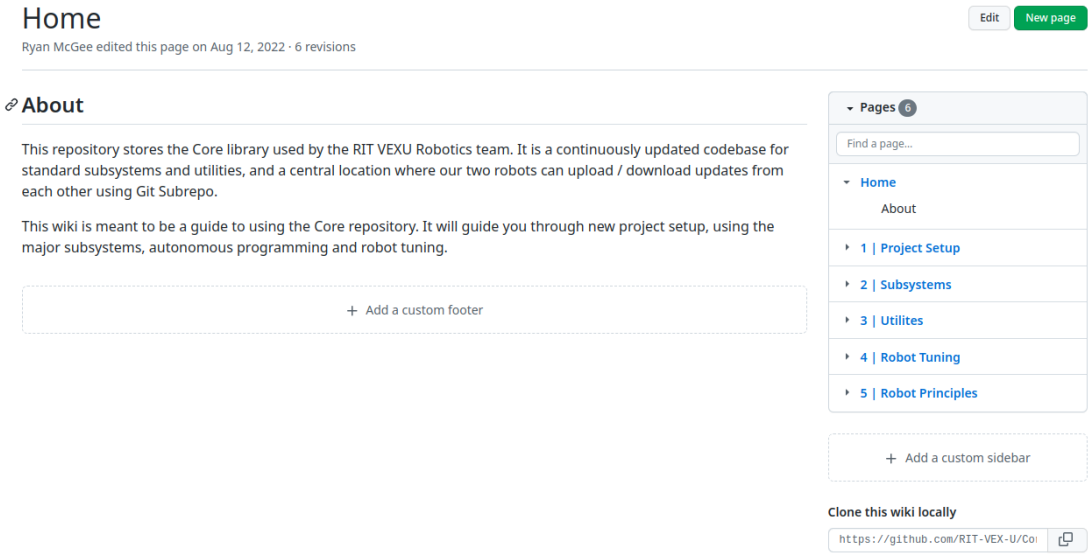
Our documentation pipeline has 3 steps: Inline code comments, Auto-generated Doxygen documentation, and the Core API Wiki. Our software team has an internal style guide that dictates formatting and what information must be included in comments at the beginning of classes and functions, to ensure our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



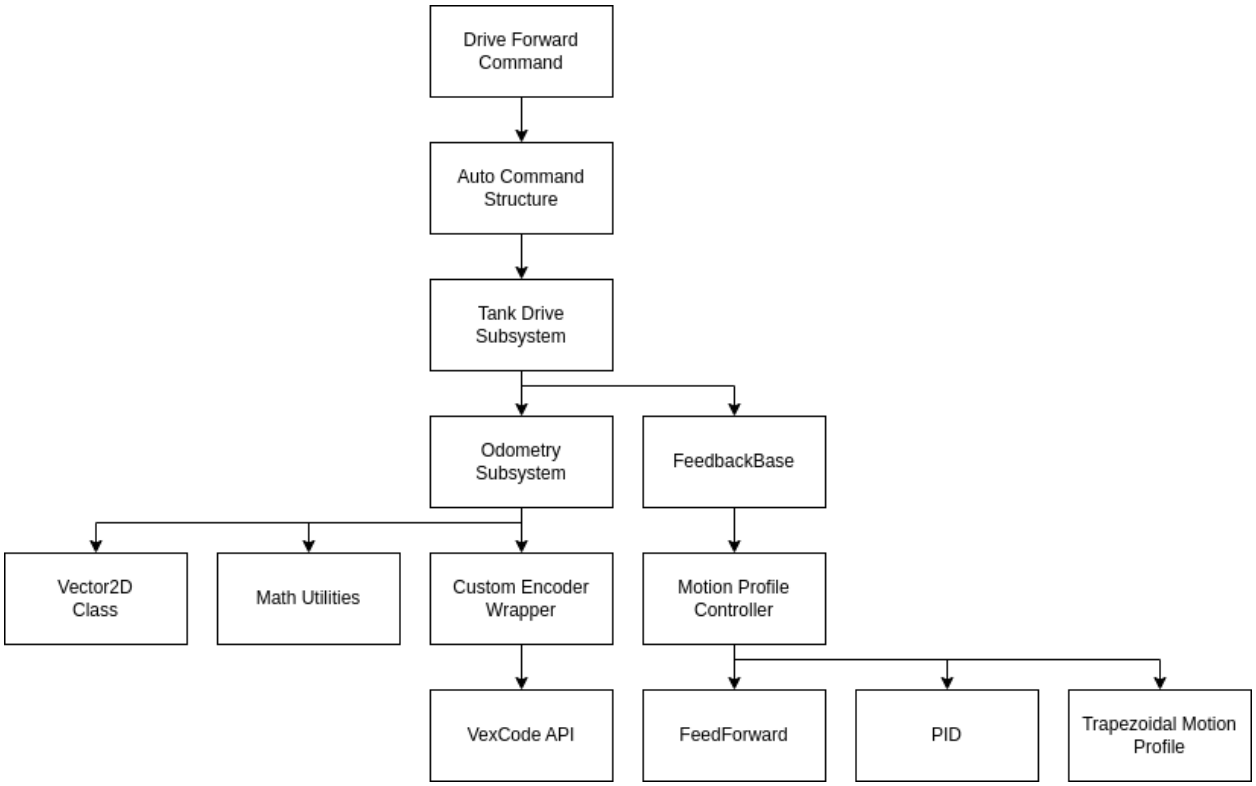
The Core Wiki contains information and tutorials to help people who are new to the ecosystem. This includes tutorials for project setup, subsystem overviews and explanations, and example usage code. For utilities, there are mathematical formulas and theory, as well as tuning tutorials for control loops. 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:

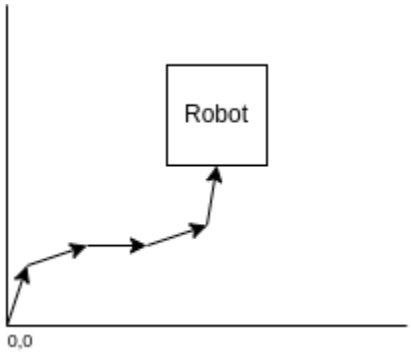


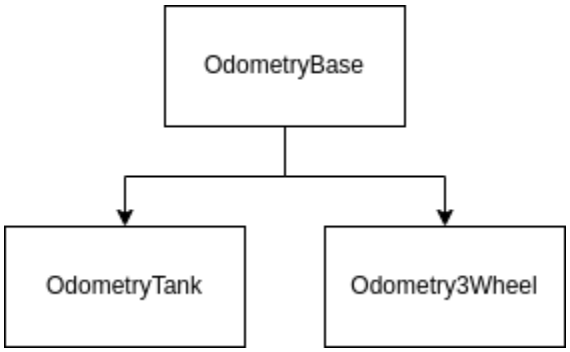
Each subsystem is built on top of another, which compartmentalizes tuning and simplifies development for higher-abstraction and more complex code.

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.

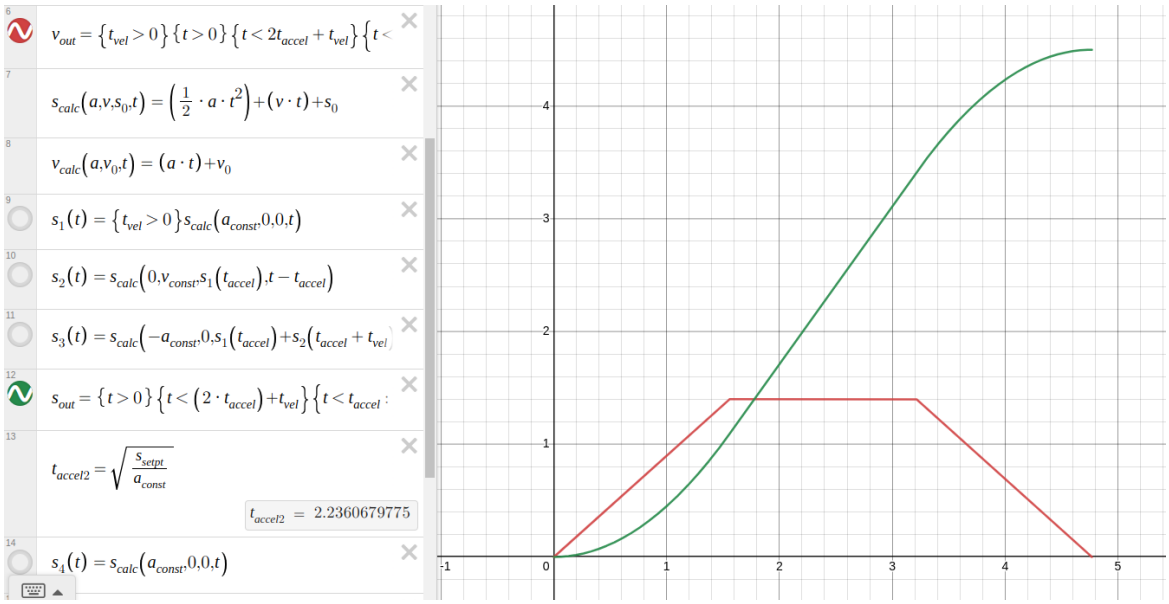
Generic Feedback

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

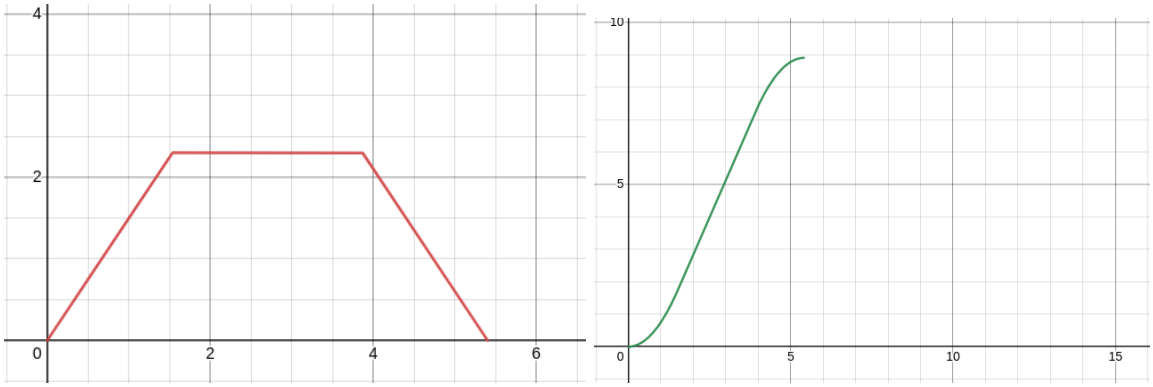
Every year brings new improvements to our API, as students work on summer projects and as the software team is waiting on our new robot design. At the end of each season, we set goals of new features that can give us an edge in the new season. This year, we resolved to explore motion profiles, improve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

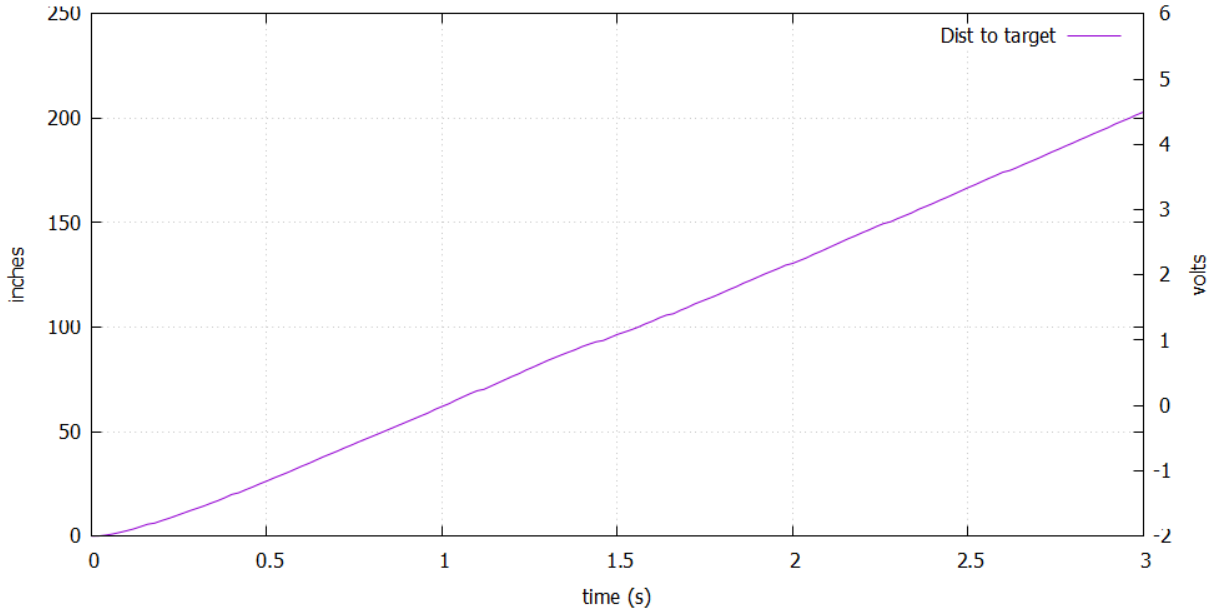
Tracking Tuning Process

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.

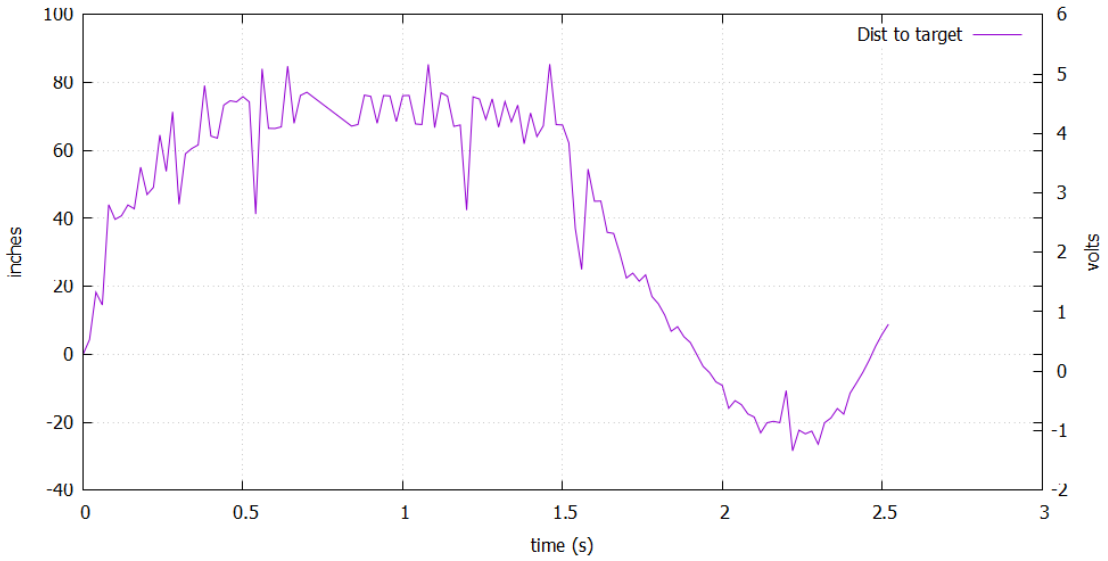
Drivetrain Tuning Manually

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.

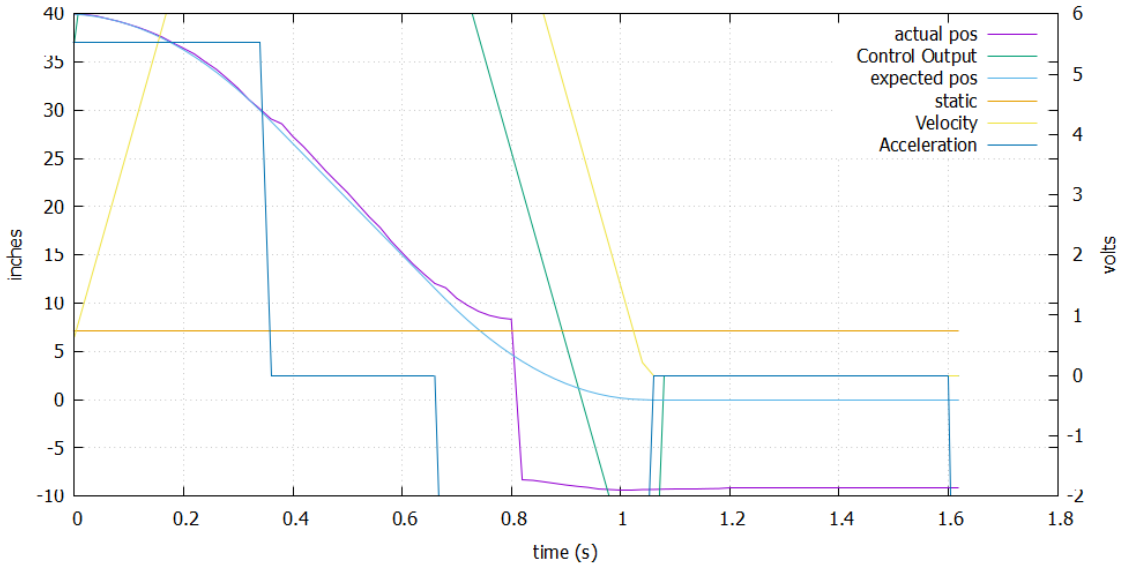


Distance covered at max motor output



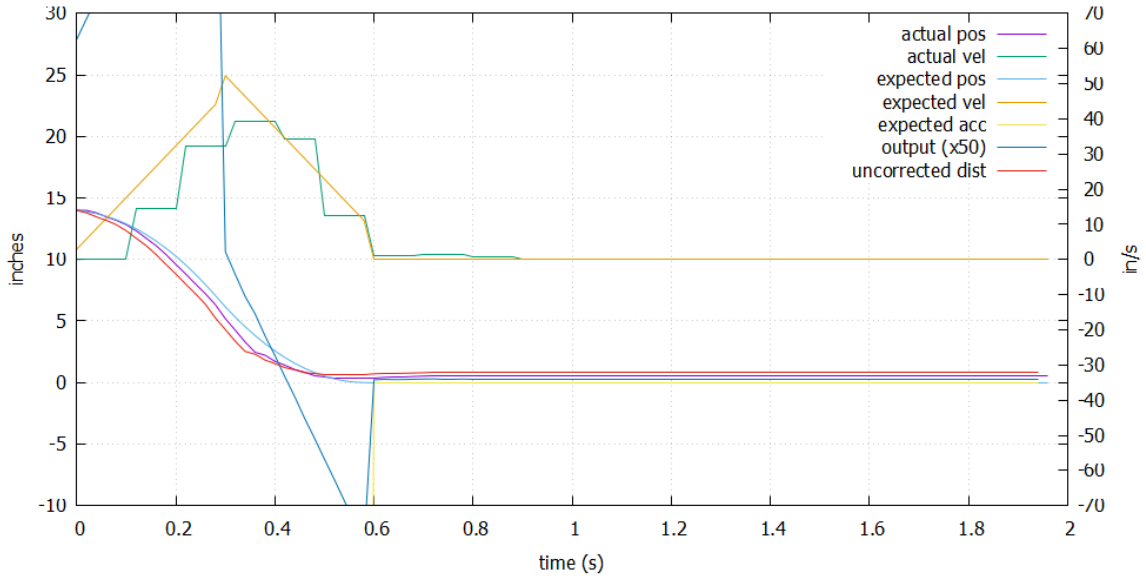
Velocity measured at max motor output

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.

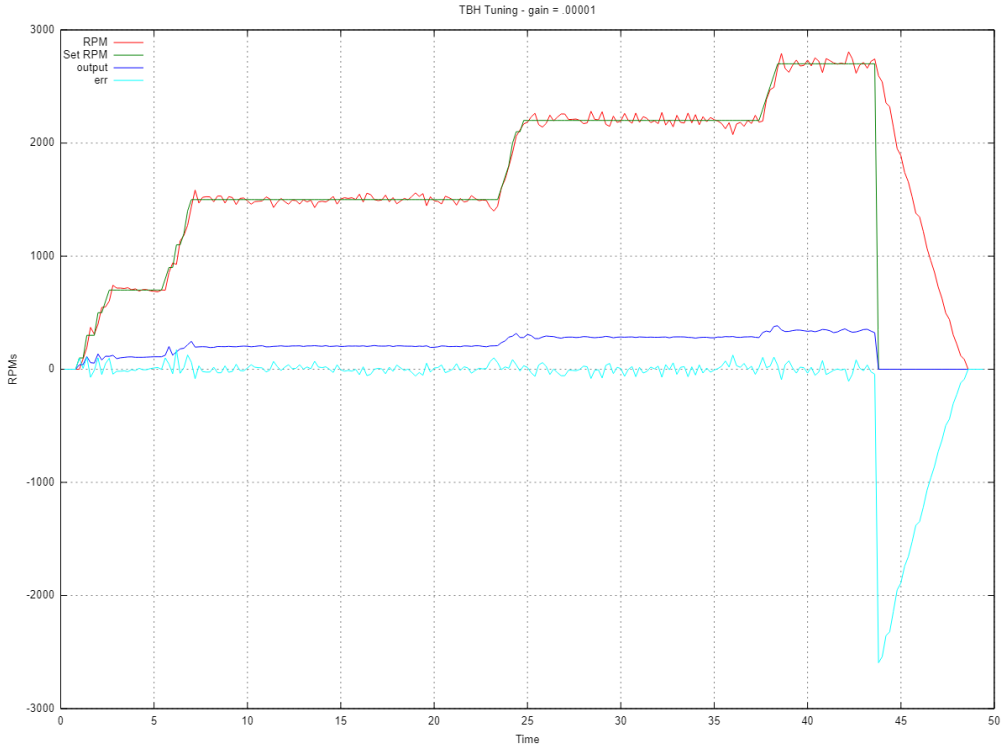
Manual Flywheel Tuning

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

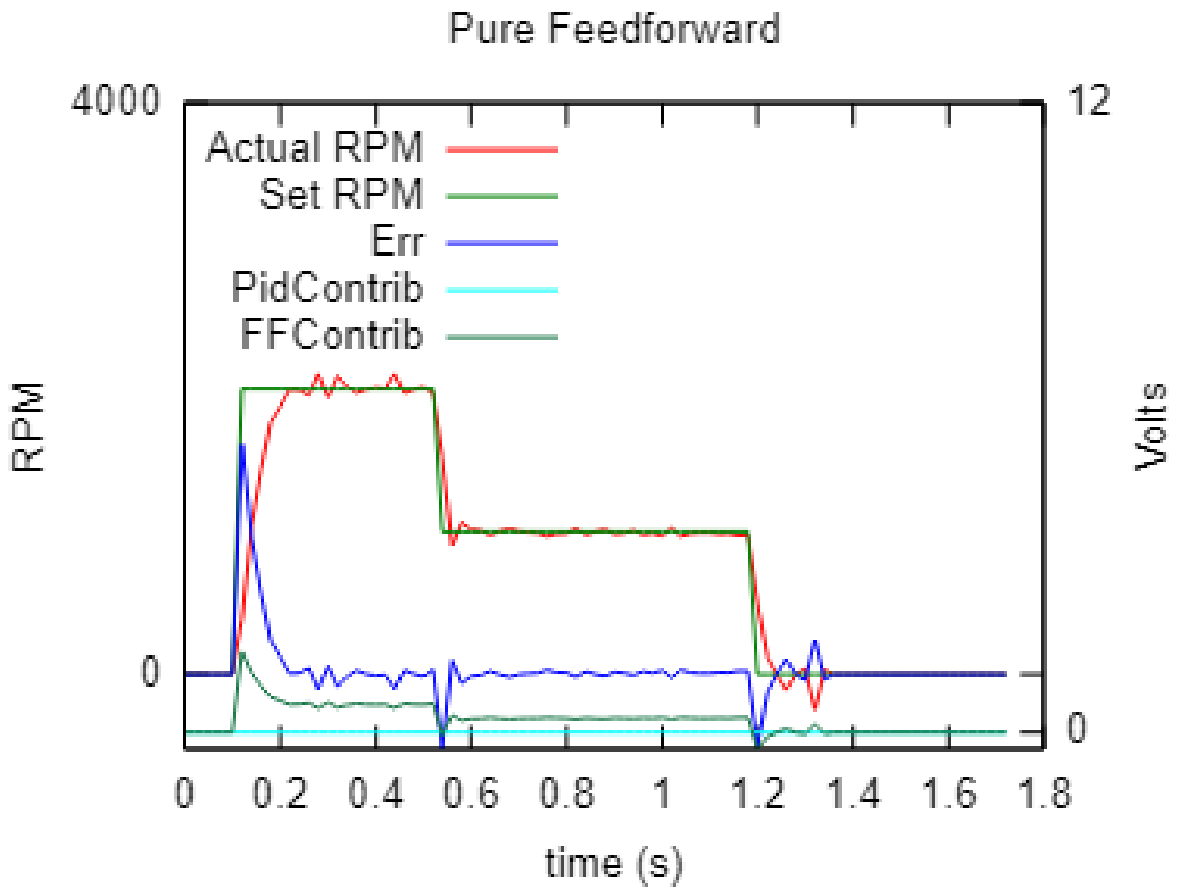
Bang-Bang control was a pleasant thought as we constructed our flywheel and wanted a quick way of reaching velocity. However, as we sought further accuracy, it became too unstable for our uses.

Take Back Half



The peak of our Take Back Half control tuning.

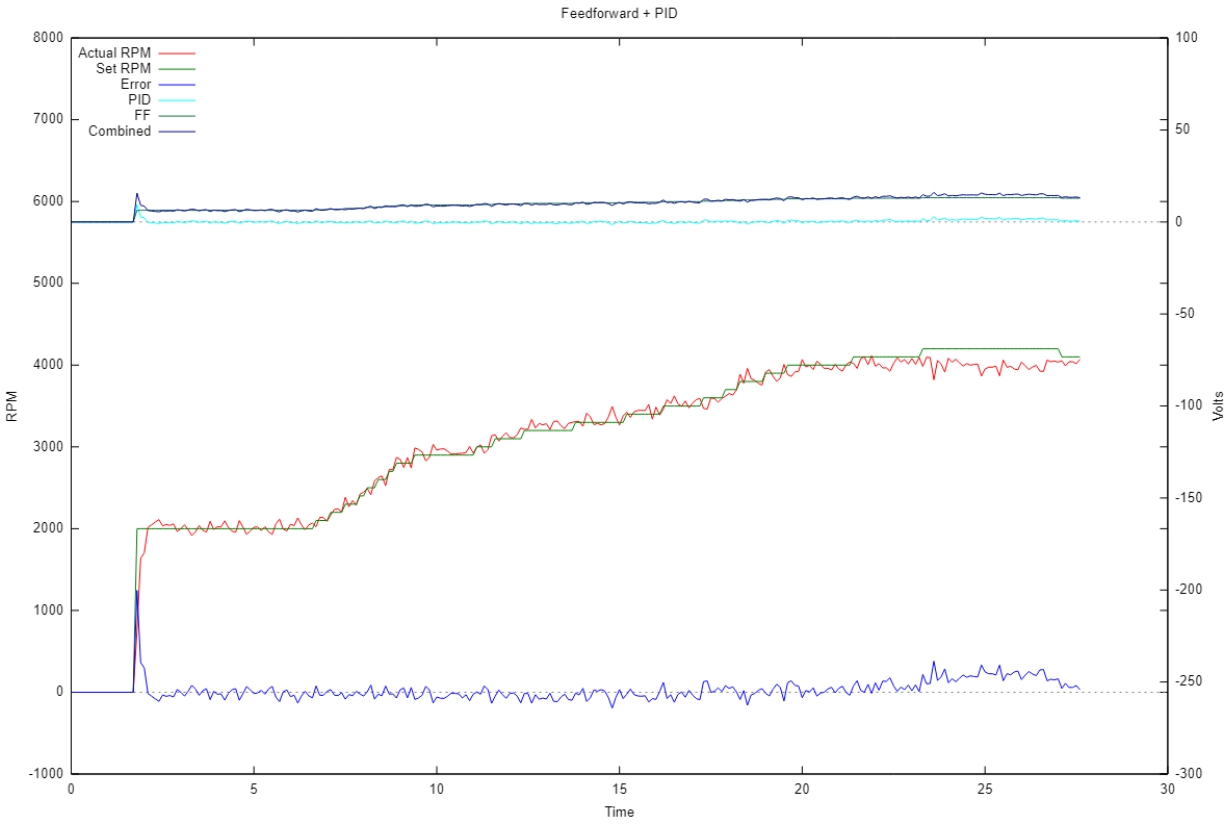
Take Back Half control was a new scheme we introduced into our codebase this year. It is a feedback control scheme applied to velocity that requires only one tuning parameter. For this simplistic tuning process it provided excellent results however we found a feedforward and feedback controller combination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

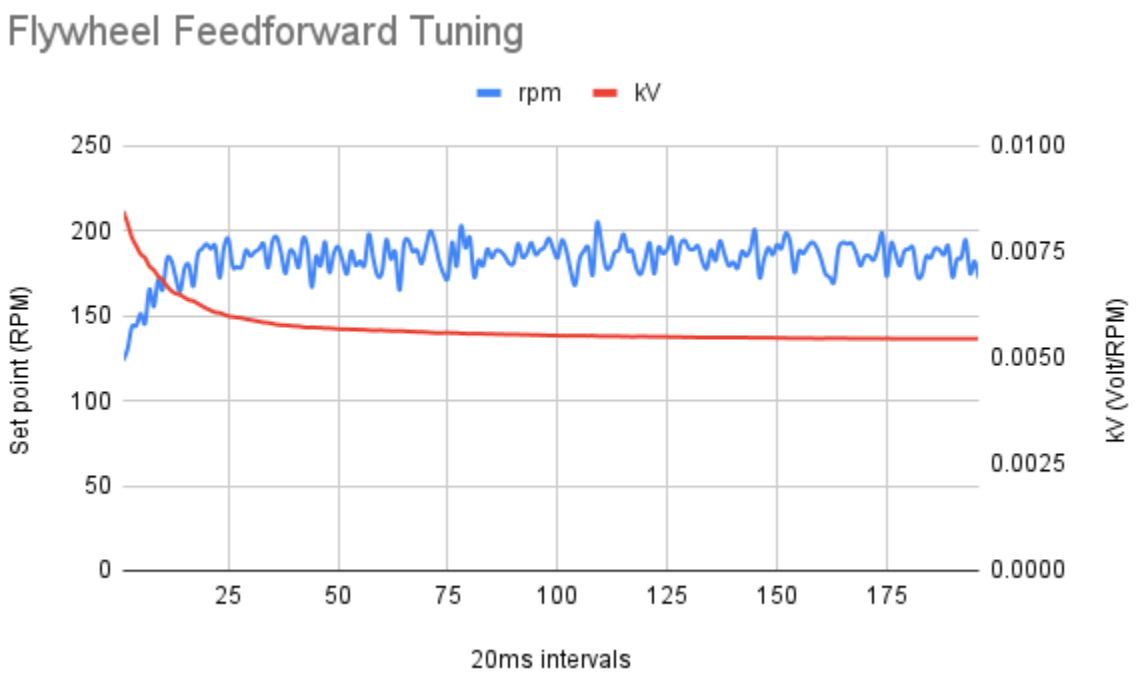
Feedforward + PID



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

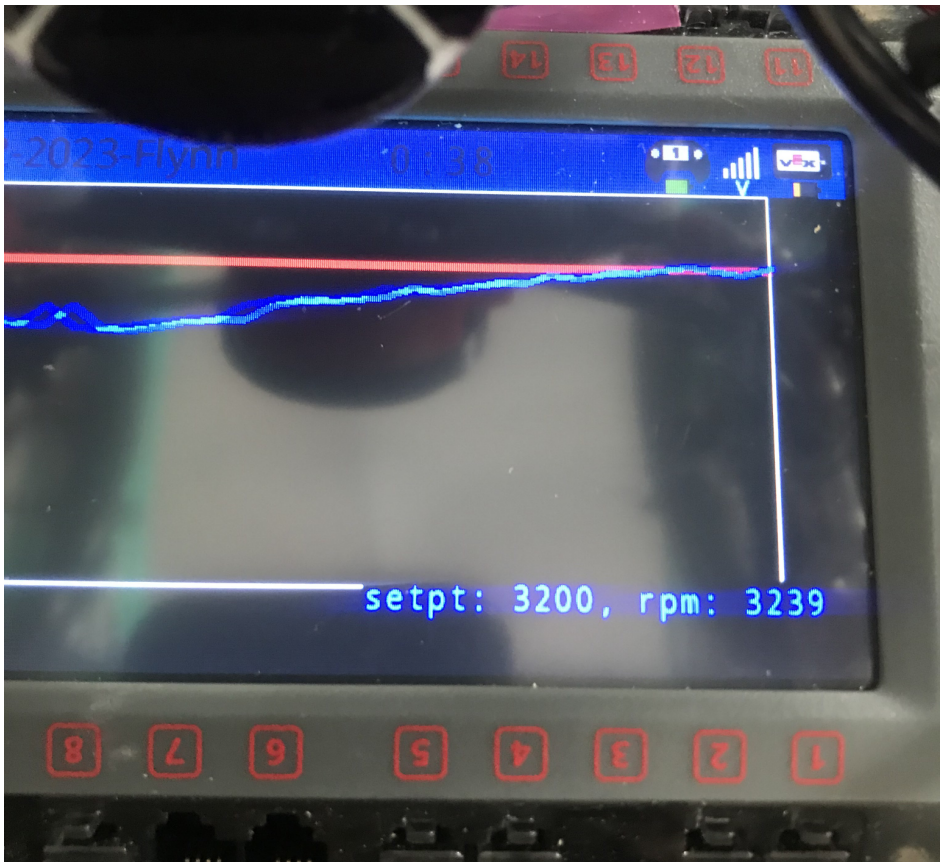
With the new addition of the Flywheel class, we needed tuning helpers to assist us. These work similarly to the drivetrain helpers above, but outputs more detailed information and allows direct control of the flywheel's RPM during tuning. These helpers are much more useful while working on Feedforward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



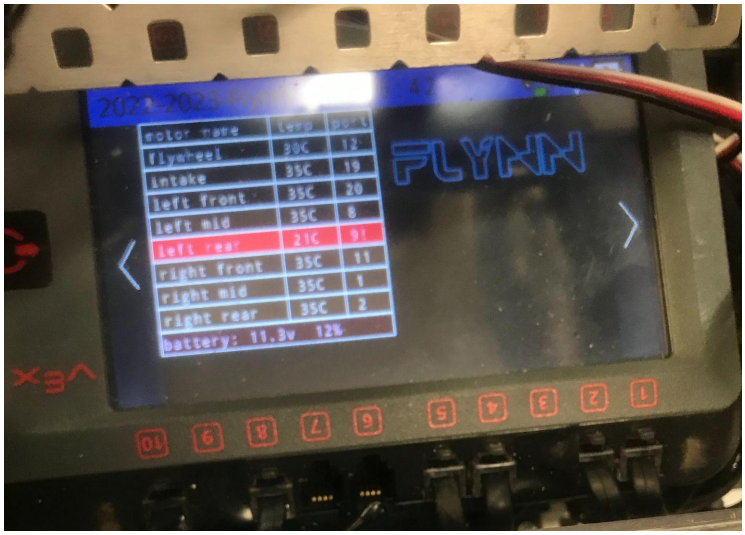
A graph of Flywheel Set Point and RPM used to improve recovery time

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

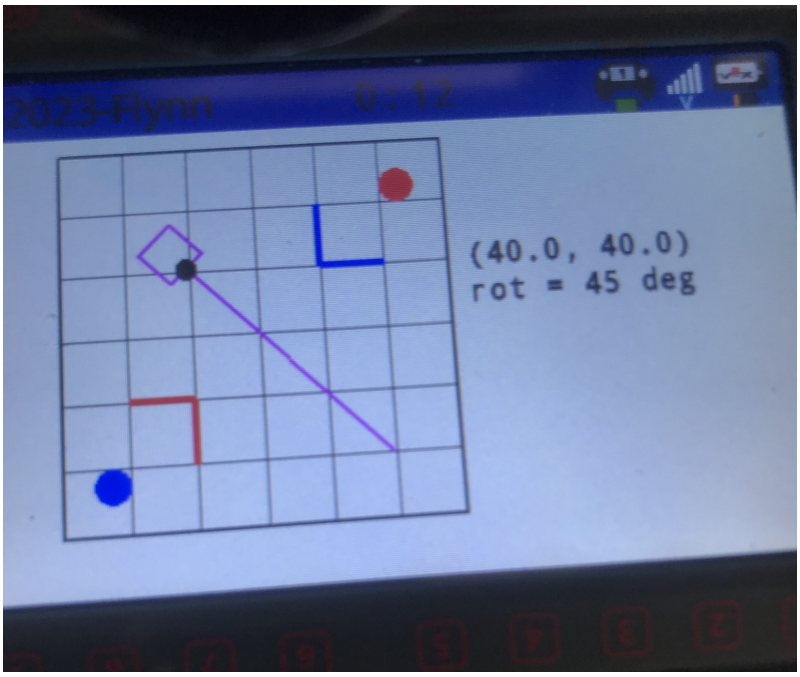
One would think it an easy step to remember to plug motors in, and yet multiple times this season we have been bewildered and hindered by an unplugged motor. This tool was built to continuously display that the motor had been unplugged and was not cancellable like the

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

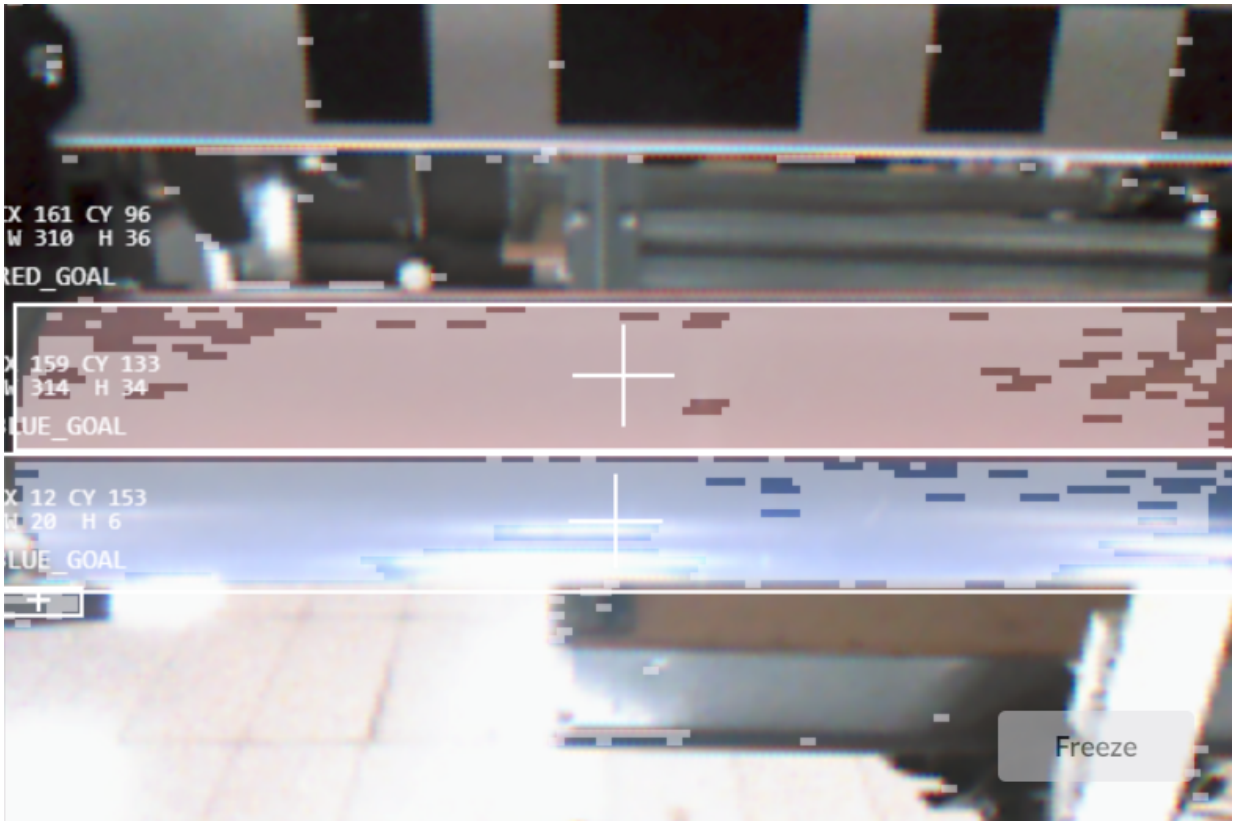
After our first competition, it became clear that for any cross field shot to have a chance of scoring a high goal, a more precise targeting system was needed. We implemented color goal detection but needed a way to specify if the robot should use vision or fall back to odometry targeting and which colored 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*

RIT VEXU Core API

Generated by Doxygen 1.9.8

Chapter 1

Core

This is the host repository for the custom VEX libraries used by the RIT VEXU team

Automatically updated documentation is available at here. There is also a downloadable reference manual.

1.1 Getting Started

In order to simply use this repo, you can either clone it into your VEXcode project folder, or download the .zip and place it into a core/ subfolder. Then follow the instructions for setting up compilation at Wiki/BuildSystem

If you wish to contribute, follow the instructions at Wiki/ProjectSetup

1.2 Features

Here is the current feature list this repo provides:

Subsystems (See Wiki/Subsystems):

- Tank drivetrain (user control / autonomous)
- Mecanum drivetrain (user control / autonomous)
- Odometry
- Flywheel
- Lift
- Custom encoders

Utilities (See Wiki/Utilites):

- PID controller
- FeedForward controller
- Trapezoidal motion profile controller
- Pure Pursuit
- Generic auto program builder
- Auto program UI selector
- Mathematical classes (Vector2D, Moving Average)

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 AutoChooser Class Reference

`#include <auto_chooser.h>`

Classes

- struct entry_t

Public Member Functions

- AutoChooser (vex::brain &brain)
- void add (std::string name)
- std::string get_choice ()

Protected Member Functions

- void render (entry_t *selected)

Protected Attributes

- std::string choice
- std::vector< entry_t > list
- vex::brain & brain

5.1.1 Detailed Description

Autochooser is a utility to make selecting robot autonomous programs easier source: RIT VexU Wiki During a season, we usually code between 4 and 6 autonomous programs. Most teams will change their entire robot program as a way of choosing autonomi but this may cause issues if you have an emergency patch to upload during a competition. This class was built as a way of using the robot screen to list autonomous programs, and the touchscreen to select them.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 AutoChooser()

```
AutoChooser::AutoChooser (
            vex::brain & brain )
```

Initialize the auto-chooser. This class places a choice menu on the brain screen, so the driver can choose which autonomous to run.

Parameters

brain	the brain on which to draw the selection boxes

5.1.3 Member Function Documentation

5.1.3.1 add()

```
void AutoChooser::add (
            std::string name )
```

Add an auto path to the chooser

Parameters

name	The name of the path. This should be used as an human readable identifier to the auto path

Add a new autonomous option. There are 3 options per row.

5.1.3.2 get_choice()

```
std::string AutoChooser::get_choice ( )
```

Get the currently selected auto choice

Returns

the identifier to the auto path

Return the selected autonomous

5.1.3.3 render()

```
void AutoChooser::render (
            entry_t * selected )  [protected]
```

Place all the autonomous choices on the screen. If one is selected, change it's color

Parameters

selected	the choice that is currently selected

5.1.4 Member Data Documentation

5.1.4.1 brain

```
vex::brain& AutoChooser::brain  [protected]
```

the brain to show the choices on

5.1.4.2 choice

```
std::string AutoChooser::choice  [protected]
```

the current choice of auto

5.1.4.3 list

```
std::vector<entry_t> AutoChooser::list  [protected]
```

< a list of all possible auto choices

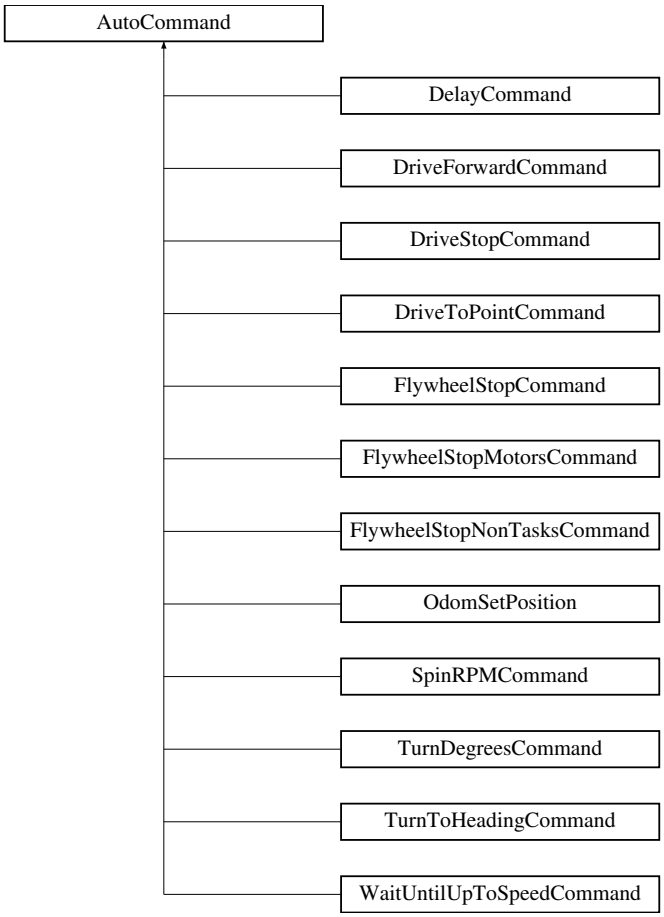
The documentation for this class was generated from the following files:

- include/utils/auto_chooser.h
- src/utils/auto_chooser.cpp

5.2 AutoCommand Class Reference

```
#include <auto_command.h>
```

Inheritance diagram for AutoCommand:



Public Member Functions

- virtual bool run ()
- virtual void on_timeout ()
- AutoCommand * **withTimeout** (double t_seconds)

Public Attributes

- double timeout_seconds = default_timeout

Static Public Attributes

- static constexpr double **default_timeout** = 10.0

5.2.1 Detailed Description

File: auto_command.h Desc: Interface for module-specifc commands

5.2.2 Member Function Documentation

5.2.2.1 on_timeout()

```
virtual void AutoCommand::on_timeout ( )  [inline], [virtual]
```

What to do if we timeout instead of finishing. timeout is specified by the timeout seconds in the constructor

Reimplemented in DriveForwardCommand, TurnDegreesCommand, TurnToHeadingCommand, and DriveStopCommand.

5.2.2.2 run()

```
virtual bool AutoCommand::run ( )  [inline], [virtual]
```

Executes the command Overridden by child classes

Returns

true when the command is finished, false otherwise

Reimplemented in DelayCommand, DriveForwardCommand, TurnDegreesCommand, DriveToPointCommand, TurnToHeadingCommand, DriveStopCommand, OdomSetPosition, SpinRPMCommand, WaitUntilUpToSpeedCommand, FlywheelStopCommand, and FlywheelStopMotorsCommand.

5.2.3 Member Data Documentation

5.2.3.1 timeout_seconds

```
double AutoCommand::timeout_seconds = default_timeout
```

How long to run until we cancel this command. If the command is cancelled, on_timeout() is called to allow any cleanup from the function. If the timeout_seconds <= 0, no timeout will be applied and this command will run forever A timeout can come in handy for some commands that can not reach the end due to some physical limitation such as

- a drive command hitting a wall and not being able to reach its target
- a command that waits until something is up to speed that never gets up to speed because of battery voltage
- something else...

The documentation for this class was generated from the following file:

- include/utils/command_structure/auto_command.h

5.3 CommandController Class Reference

```
#include <command_controller.h>
```

Public Member Functions

- void add (AutoCommand *cmd, double timeout_seconds=10.0)
- void add (std::vector< AutoCommand * > cmds)
- void add (std::vector< AutoCommand * > cmds, double timeout_sec)
- void add_delay (int ms)
- void run ()
- bool last_command_timed_out ()

5.3.1 Detailed Description

File: command_controller.h Desc: A CommandController manages the AutoCommands that make up an autonomous route. The AutoCommands are kept in a queue and get executed and removed from the queue in FIFO order.

5.3.2 Member Function Documentation

5.3.2.1 add() [1/3]

```
void CommandController::add (
            AutoCommand * cmd,
            double timeout_seconds = 10.0 )
```

Adds a command to the queue

Parameters

cmd	the AutoCommand we want to add to our list
timeout_seconds	the number of seconds we will let the command run for. If it exceeds this, we cancel it and run on_timeout. if it is <= 0 no time out will be applied

File: command_controller.cpp Desc: A CommandController manages the AutoCommands that make up an autonomous route. The AutoCommands are kept in a queue and get executed and removed from the queue in FIFO order. Adds a command to the queue

Parameters

cmd	the AutoCommand we want to add to our list
timeout_seconds	the number of seconds we will let the command run for. If it exceeds this, we cancel it and run on_timeout

5.3.2.2 add() [2/3]

```
void CommandController::add (
            std::vector< AutoCommand * > cmds )
```

Add multiple commands to the queue. No timeout here.

Parameters

cmds	the AutoCommands we want to add to our list

5.3.2.3 add() [3/3]

```
void CommandController::add (
            std::vector< AutoCommand * > cmds,
            double timeout_sec )
```

Add multiple commands to the queue. No timeout here.

Parameters

cmds	the AutoCommands we want to add to our list
timeout_sec	timeout in seconds to apply to all commands if they are still the default

Add multiple commands to the queue. No timeout here.

Parameters

cmds	the AutoCommands we want to add to our list
timeout	timeout in seconds to apply to all commands if they are still the default

5.3.2.4 add_delay()

```
void CommandController::add_delay (
            int ms )
```

Adds a command that will delay progression of the queue

Parameters

ms	- number of milliseconds to wait before continuing execution of autonomous

5.3.2.5 last_command_timed_out()

```
bool CommandController::last_command_timed_out ( )
```

last_command_timed_out tells how the last command ended Use this if you want to make decisions based on the end of the last command

Returns

true if the last command timed out. false if it finished regularly

5.3.2.6 run()

```
void CommandController::run ( )
```

Begin execution of the queue Execute and remove commands in FIFO order

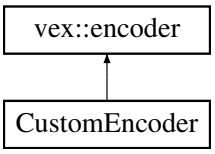
The documentation for this class was generated from the following files:

- include/utils/command_structure/command_controller.h
- src/utils/command_structure/command_controller.cpp

5.4 CustomEncoder Class Reference

```
#include <custom_encoder.h>
```

Inheritance diagram for CustomEncoder:



Public Member Functions

- CustomEncoder (vex::triport::port &port, double ticks_per_rev)
- void setRotation (double val, vex::rotationUnits units)
- void setPosition (double val, vex::rotationUnits units)
- double rotation (vex::rotationUnits units)
- double position (vex::rotationUnits units)
- double velocity (vex::velocityUnits units)

5.4.1 Detailed Description

A wrapper class for the vex encoder that allows the use of 3rd party encoders with different tick-per-revolution values.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 CustomEncoder()

```
CustomEncoder::CustomEncoder (
            vex::triport::port & port,
            double ticks_per_rev )
```

Construct an encoder with a custom number of ticks

Parameters

port	the triport port on the brain the encoder is plugged into
ticks_per_rev	the number of ticks the encoder will report for one revolution

5.4.3 Member Function Documentation

5.4.3.1 position()

```
double CustomEncoder::position (
            vex::rotationUnits units )
```

get the position that the encoder is at

Parameters

units	the unit we want the return value to be in

Returns

the position of the encoder in the units specified

5.4.3.2 rotation()

```
double CustomEncoder::rotation (
            vex::rotationUnits units )
```

get the rotation that the encoder is at

Parameters

units	the unit we want the return value to be in

Returns

the rotation of the encoder in the units specified

5.4.3.3 setPosition()

```
void CustomEncoder::setPosition (
            double val,
            vex::rotationUnits units )
```

sets the stored position of the encoder. Any further movements will be from this value

Parameters

val	the numerical value of the position we are setting to
units	the unit of val

5.4.3.4 setRotation()

```
void CustomEncoder::setRotation (
            double val,
            vex::rotationUnits units )
```

sets the stored rotation of the encoder. Any further movements will be from this value

Parameters

val	the numerical value of the angle we are setting to
units	the unit of val

5.4.3.5 velocity()

```
double CustomEncoder::velocity (
            vex::velocityUnits units )
```

get the velocity that the encoder is moving at

Parameters

units	the unit we want the return value to be in

Returns

the velocity of the encoder in the units specified

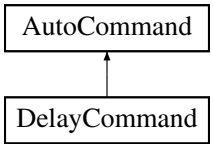
The documentation for this class was generated from the following files:

- include/subsystems/custom_encoder.h
- src/subsystems/custom_encoder.cpp

5.5 DelayCommand Class Reference

`#include <delay_command.h>`

Inheritance diagram for DelayCommand:



Public Member Functions

- DelayCommand (int ms)
- bool run () override

Public Member Functions inherited from AutoCommand

- virtual void on_timeout ()
- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.5.1 Detailed Description

File: delay_command.h Desc: A DelayCommand will make the robot wait the set amount of milliseconds before continuing execution of the autonomous route

5.5.2 Constructor & Destructor Documentation

5.5.2.1 DelayCommand()

```
DelayCommand::DelayCommand (
            int ms )  [inline]
```

Construct a delay command

Parameters

ms	the number of milliseconds to delay for

5.5.3 Member Function Documentation

5.5.3.1 run()

```
bool DelayCommand::run ( )  [inline], [override], [virtual]
```

Delays for the amount of milliseconds stored in the command Overrides run from AutoCommand

Returns

true when complete

Reimplemented from AutoCommand.

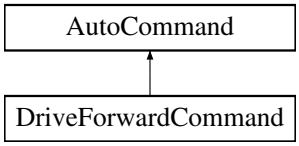
The documentation for this class was generated from the following file:

- include/utils/command_structure/delay_command.h

5.6 DriveForwardCommand Class Reference

`#include <drive_commands.h>`

Inheritance diagram for DriveForwardCommand:



Public Member Functions

- DriveForwardCommand (TankDrive &drive_sys, Feedback &feedback, double inches, directionType dir, double max_speed=1)
- bool run () override
- void on_timeout () override

Public Member Functions inherited from AutoCommand

- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.6.1 Detailed Description

AutoCommand wrapper class for the drive_forward function in the TankDrive class

5.6.2 Constructor & Destructor Documentation

5.6.2.1 DriveForwardCommand()

```
DriveForwardCommand::DriveForwardCommand (
            TankDrive & drive_sys,
            Feedback & feedback,
            double inches,
            directionType dir,
            double max_speed = 1 )
```

File: drive_commands.h Desc: Holds all the AutoCommand subclasses that wrap (currently) TankDrive functions

Currently includes:

- drive_forward
- turn_degrees
- drive_to_point
- turn_to_heading
- stop

Also holds AutoCommand subclasses that wrap OdometryBase functions

Currently includes:

- set_position Construct a DriveForward Command

Parameters

drive_sys	the drive system we are commanding
feedback	the feedback controller we are using to execute the drive
inches	how far forward to drive
dir	the direction to drive
max_speed	0 -> 1 percentage of the drive systems speed to drive at

5.6.3 Member Function Documentation

5.6.3.1 on_timeout()

```
void DriveForwardCommand::on_timeout ( )  [override], [virtual]
```

Cleans up drive system if we time out before finishing

reset the drive system if we timeout

Reimplemented from AutoCommand.

5.6.3.2 run()

```
bool DriveForwardCommand::run ( )  [override], [virtual]
```

Run drive_forward Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

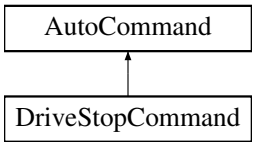
The documentation for this class was generated from the following files:

- include/utils/command_structure/drive_commands.h
- src/utils/command_structure/drive_commands.cpp

5.7 DriveStopCommand Class Reference

```
#include <drive_commands.h>
```

Inheritance diagram for DriveStopCommand:



Public Member Functions

- DriveStopCommand (TankDrive &drive_sys)
- bool run () override
- void on_timeout () override

Public Member Functions inherited from AutoCommand

- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.7.1 Detailed Description

AutoCommand wrapper class for the stop() function in the TankDrive class

5.7.2 Constructor & Destructor Documentation

5.7.2.1 DriveStopCommand()

```
DriveStopCommand::DriveStopCommand (
            TankDrive & drive_sys )
```

Construct a DriveStop Command

Parameters

drive_sys	the drive system we are commanding

5.7.3 Member Function Documentation

5.7.3.1 on_timeout()

```
void DriveStopCommand::on_timeout ( )  [override], [virtual]
```

What to do if we timeout instead of finishing. timeout is specified by the timeout seconds in the constructor

Reimplemented from AutoCommand.

5.7.3.2 run()

```
bool DriveStopCommand::run ( )  [override], [virtual]
```

Stop the drive system Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Stop the drive train Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

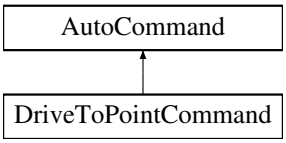
The documentation for this class was generated from the following files:

- include/utils/command_structure/drive_commands.h
- src/utils/command_structure/drive_commands.cpp

5.8 DriveToPointCommand Class Reference

`#include <drive_commands.h>`

Inheritance diagram for DriveToPointCommand:



Public Member Functions

- DriveToPointCommand (TankDrive &drive_sys, Feedback &feedback, double x, double y, directionType dir, double max_speed=1)
- DriveToPointCommand (TankDrive &drive_sys, Feedback &feedback, point_t point, directionType dir, double max_speed=1)
- bool run () override

Public Member Functions inherited from AutoCommand

- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.8.1 Detailed Description

AutoCommand wrapper class for the drive_to_point function in the TankDrive class

5.8.2 Constructor & Destructor Documentation

5.8.2.1 DriveToPointCommand() [1/2]

```
DriveToPointCommand::DriveToPointCommand (
            TankDrive & drive_sys,
            Feedback & feedback,
            double x,
            double y,
            directionType dir,
            double max_speed = 1 )
```

Construct a DriveForward Command

Parameters

drive_sys	the drive system we are commanding
feedback	the feedback controller we are using to execute the drive
x	where to drive in the x dimension
y	where to drive in the y dimension
dir	the direction to drive
max_speed	0 -> 1 percentage of the drive systems speed to drive at

5.8.2.2 DriveToPointCommand() [2/2]

```
DriveToPointCommand::DriveToPointCommand (
            TankDrive & drive_sys,
            Feedback & feedback,
            point_t point,
            directionType dir,
            double max_speed = 1 )
```

Construct a DriveForward Command

Parameters

drive_sys	the drive system we are commanding
feedback	the feedback controller we are using to execute the drive
point	the point to drive to
dir	the direction to drive
max_speed	0 -> 1 percentage of the drive systems speed to drive at

5.8.3 Member Function Documentation

5.8.3.1 run()

```
bool DriveToPointCommand::run ( )  [override], [virtual]
```

Run drive_to_point Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

The documentation for this class was generated from the following files:

- include/utils/command_structure/drive_commands.h
- src/utils/command_structure/drive_commands.cpp

5.9 AutoChooser::entry_t Struct Reference

`#include <auto_chooser.h>`

Public Attributes

- int x
- int y
- int width
- int height
- std::string name

5.9.1 Detailed Description

entry_t is a datatype used to store information that the chooser knows about an auto selection button

5.9.2 Member Data Documentation

5.9.2.1 height

`int AutoChooser::entry_t::height`

height of the block

5.9.2.2 name

`std::string AutoChooser::entry_t::name`

name of the auto repretsented by the block

5.9.2.3 width

`int AutoChooser::entry_t::width`

width of the block

5.9.2.4 x

`int AutoChooser::entry_t::x`

screen x position of the block

5.9.2.5 y

```
int AutoChooser::entry_t::y
```

screen y position of the block

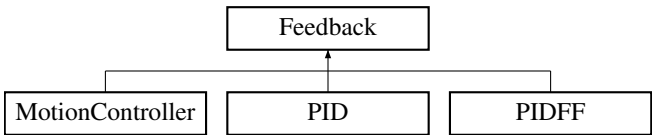
The documentation for this struct was generated from the following file:

- include/utils/auto_chooser.h

5.10 Feedback Class Reference

```
#include <feedback_base.h>
```

Inheritance diagram for Feedback:



Public Types

- enum **FeedbackType** { **PIDType** , **FeedforwardType** , **OtherType** }

Public Member Functions

- virtual void init (double start_pt, double set_pt)=0
- virtual double update (double val)=0
- virtual double get ()=0
- virtual void set_limits (double lower, double upper)=0
- virtual bool is_on_target ()=0
- virtual Feedback::FeedbackType **get_type** ()

5.10.1 Detailed Description

Interface so that subsystems can easily switch between feedback loops

Author

Ryan McGee

Date

9/25/2022

5.10.2 Member Function Documentation

5.10.2.1 get()

```
virtual double Feedback::get ( )  [pure virtual]
```

Returns

the last saved result from the feedback controller

Implemented in MotionController, PID, and PIDFF.

5.10.2.2 init()

```
virtual void Feedback::init (
            double start_pt,
            double set_pt )  [pure virtual]
```

Initialize the feedback controller for a movement

Parameters

start_pt	the current sensor value
set_pt	where the sensor value should be

Implemented in MotionController, PID, and PIDFF.

5.10.2.3 is_on_target()

```
virtual bool Feedback::is_on_target ( )  [pure virtual]
```

Returns

true if the feedback controller has reached it's setpoint

Implemented in MotionController, PID, and PIDFF.

5.10.2.4 set_limits()

```
virtual void Feedback::set_limits (
            double lower,
            double upper )  [pure virtual]
```

Clamp the upper and lower limits of the output. If both are 0, no limits should be applied.

Parameters

lower	Upper limit
upper	Lower limit

Implemented in MotionController, PID, and PIDFF.

5.10.2.5 update()

```
virtual double Feedback::update (
            double val )  [pure virtual]
```

Iterate the feedback loop once with an updated sensor value

Parameters

val	value from the sensor

Returns

feedback loop result

Implemented in MotionController, PID, and PIDFF.

The documentation for this class was generated from the following file:

- include/utils/feedback_base.h

5.11 FeedForward Class Reference

```
#include <feedforward.h>
```

Classes

- struct ff_config_t

Public Member Functions

- FeedForward (ff_config_t &cfg)
- double calculate (double v, double a, double pid_ref=0.0)

  Perform the feedforward calculation.

5.11.1 Detailed Description

FeedForward

Stores the feedfoward constants, and allows for quick computation. Feedfoward should be used in systems that require smooth precise movements and have high inertia, such as drivetrains and lifts.

This is best used alongside a PID loop, with the form: output = pid.get() + feedforward.calculate(v, a);

In this case, the feedforward does the majority of the heavy lifting, and the pid loop only corrects for inconsistencies

For information about tuning feedforward, I reccommend looking at this post: https://www.chiefdelphi.com/t/paper-frc-drivetrain-characterization/160915 (yes I know it's for FRC but trust me, it's useful)

Author

Ryan McGee

Date

6/13/2022

5.11.2 Constructor & Destructor Documentation

5.11.2.1 FeedForward()

```
FeedForward::FeedForward (
            ff_config_t & cfg )  [inline]
```

Creates a FeedForward object.

Parameters

cfg	Configuration Struct for tuning

5.11.3 Member Function Documentation

5.11.3.1 calculate()

```
double FeedForward::calculate (
            double v,
            double a,
            double pid_ref = 0.0 )  [inline]
```

Perform the feedforward calculation.

This calculation is the equation: F = kG + kS∗sgn(v) + kV∗v + kA∗a

Parameters

v	Requested velocity of system
a	Requested acceleration of system

Returns

A feedforward that should closely represent the system if tuned correctly

The documentation for this class was generated from the following file:

- include/utils/feedforward.h

5.12 FeedForward::ff_config_t Struct Reference

`#include <feedforward.h>`

Public Attributes

- double kS
- double kV
- double kA
- double kG

5.12.1 Detailed Description

ff_config_t holds the parameters to make the theoretical model of a real world system equation is of the form kS if the system is not stopped, 0 otherwise

- kV $*$ desired velocity
- kA $*$ desired acceleration
- kG

5.12.2 Member Data Documentation

5.12.2.1 kA

`double FeedForward::ff_config_t::kA`

kA - Acceleration coefficient: the power required to change the mechanism's speed. Multiplied by the requested acceleration.

5.12.2.2 kG

```
double FeedForward::ff_config_t::kG
```

kG - Gravity coefficient: only needed for lifts. The power required to overcome gravity and stay at steady state.

5.12.2.3 kS

```
double FeedForward::ff_config_t::kS
```

Coefficient to overcome static friction: the point at which the motor *starts* to move.

5.12.2.4 kV

```
double FeedForward::ff_config_t::kV
```

Veclocity coefficient: the power required to keep the mechanism in motion. Multiplied by the requested velocity.

The documentation for this struct was generated from the following file:

- include/utils/feedforward.h

5.13 Flywheel Class Reference

```
#include <flywheel.h>
```

Public Member Functions

- Flywheel (motor_group &motors, PID::pid_config_t &pid_config, FeedForward::ff_config_t &ff_config, const double ratio)
- Flywheel (motor_group &motors, FeedForward::ff_config_t &ff_config, const double ratio)
- Flywheel (motor_group &motors, double tbh_gain, const double ratio)
- Flywheel (motor_group &motors, const double ratio)
- double getDesiredRPM ()
- bool isTaskRunning ()
- motor_group * getMotors ()
- double measureRPM ()
- double getRPM ()
- PID * getPID ()
- double getPIDValue ()
- double getFeedforwardValue ()
- double getTBHGain ()
- void setPIDTarget (double value)
- void updatePID (double value)
- void spin_raw (double speed, directionType dir=fwd)
- void spin_manual (double speed, directionType dir=fwd)
- void spinRPM (int rpm)
- void stop ()
- void stopMotors ()
- void stopNonTasks ()

5.13.1 Detailed Description

a Flywheel class that handles all control of a high inertia spinning disk It gives multiple options for what control system to use in order to control wheel velocity and functions alerting the user when the flywheel is up to speed. Flywheel is a set and forget class. Once you create it you can call spinRPM or stop on it at any time and it will take all necessary steps to accomplish this

5.13.2 Constructor & Destructor Documentation

5.13.2.1 Flywheel() [1/4]

```
Flywheel::Flywheel (
            motor_group & motors,
            PID::pid_config_t & pid_config,
            FeedForward::ff_config_t & ff_config,
            const double ratio )
```

Create the Flywheel object using PID + feedforward for control.

Parameters

motors	pointer to the motors on the fly wheel
pid_config	pointer the pid config to use
ff_config	the feedforward config to use
ratio	ratio of the whatever just multiplies the velocity

Create the Flywheel object using PID + feedforward for control.

5.13.2.2 Flywheel() [2/4]

```
Flywheel::Flywheel (
            motor_group & motors,
            FeedForward::ff_config_t & ff_config,
            const double ratio )
```

Create the Flywheel object using only feedforward for control

Parameters

motors	the motors on the fly wheel
ff_config	the feedforward config to use
ratio	ratio of the whatever just multiplies the velocity

Create the Flywheel object using only feedforward for control

5.13.2.3 Flywheel() [3/4]

```
Flywheel::Flywheel (
            motor_group & motors,
```

```
            double tbh_gain,
            const double ratio )
```

Create the Flywheel object using Take Back Half for control

Parameters

motors	the motors on the fly wheel
tbh_gain	the TBH control paramater
ratio	ratio of the whatever just multiplies the velocity

Create the Flywheel object using Take Back Half for control

5.13.2.4 Flywheel() [4/4]

```
Flywheel::Flywheel (
            motor_group & motors,
            const double ratio )
```

Create the Flywheel object using Bang Bang for control

Parameters

motors	the motors on the fly wheel
ratio	ratio of the whatever just multiplies the velocity

Create the Flywheel object using Bang Bang for control

5.13.3 Member Function Documentation

5.13.3.1 getDesiredRPM()

```
double Flywheel::getDesiredRPM ( )
```

Return the RPM that the flywheel is currently trying to achieve

Returns

RPM the target rpm

Return the current value that the RPM should be set to

5.13.3.2 getFeedforwardValue()

```
double Flywheel::getFeedforwardValue ( )
```

returns the current OUT value of the PID - the value that the PID would set the motors to

returns the current OUT value of the Feedforward - the value that the Feedforward would set the motors to

Returns

the voltage that feedforward wants the motors at to achieve the target RPM

5.13.3.3 getMotors()

```
motor_group * Flywheel::getMotors ( )
```

Returns a POINTER to the motors

Returns a POINTER TO the motors; not currently used.

Returns

motorPointer -pointer to the motors

5.13.3.4 getPID()

```
PID * Flywheel::getPID ( )
```

Returns a POINTER to the PID.

Returns a POINTER TO the PID; not currently used.

Returns

pidPointer -pointer to the PID

5.13.3.5 getPIDValue()

```
double Flywheel::getPIDValue ( )
```

returns the current OUT value of the PID - the value that the PID would set the motors to

returns the current OUT value of the PID - the value that the PID would set the motors to

Returns

the voltage that PID wants the motors at to achieve the target RPM

5.13.3.6 getRPM()

```
double Flywheel::getRPM ( )
```

return the current smoothed velocity of the flywheel motors, in RPM

5.13.3.7 getTBHGain()

```
double Flywheel::getTBHGain ( )
```

get the gain used for TBH control

get the gain used for TBH control

Returns

the gain used in TBH control

5.13.3.8 isTaskRunning()

```
bool Flywheel::isTaskRunning ( )
```

Checks if the background RPM controlling task is running

Returns

true if the task is running

Checks if the background RPM controlling task is running

Returns

taskRunning - If the task is running

5.13.3.9 measureRPM()

```
double Flywheel::measureRPM ( )
```

make a measurement of the current RPM of the flywheel motor and return a smoothed version

return the current velocity of the flywheel motors, in RPM

Returns

the measured velocity of the flywheel

5.13.3.10 setPIDTarget()

```
void Flywheel::setPIDTarget (
            double value )
```

Sets the value of the PID target

Parameters

value	- desired value of the PID

5.13.3.11 spin_manual()

```
void Flywheel::spin_manual (
            double speed,
            directionType dir = fwd )
```

Spin motors using voltage; defaults forward at 12 volts FOR USE BY OPCONTROL AND AUTONOMOUS - this only applies if the RPM thread is not running

Parameters

speed	- speed (between -1 and 1) to set the motor
dir	- direction that the motor moves in; defaults to forward

5.13.3.12 spin_raw()

```
void Flywheel::spin_raw (
            double speed,
            directionType dir = fwd )
```

Spin motors using voltage; defaults forward at 12 volts FOR USE BY TASKS ONLY

Parameters

speed	- speed (between -1 and 1) to set the motor
dir	- direction that the motor moves in; defaults to forward

5.13.3.13 spinRPM()

```
void Flywheel::spinRPM (
            int inputRPM )
```

starts or sets the RPM thread at new value what control scheme is dependent on control_style

Parameters

rpm	- the RPM we want to spin at

starts or sets the RPM thread at new value what control scheme is dependent on control_style

Parameters

inputRPM	- set the current RPM

5.13.3.14 stop()

```
void Flywheel::stop ( )
```

stop the RPM thread and the wheel

5.13.3.15 stopMotors()

```
void Flywheel::stopMotors ( )
```

stop only the motors; exclusively for BANG BANG use

5.13.3.16 stopNonTasks()

```
void Flywheel::stopNonTasks ( )
```

Stop the motors if the task isn't running - stop manual control

5.13.3.17 updatePID()

```
void Flywheel::updatePID (
            double value )
```

updates the value of the PID

Parameters

value	- value to update the PID with

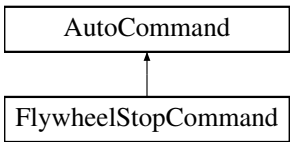
The documentation for this class was generated from the following files:

- include/subsystems/flywheel.h
- src/subsystems/flywheel.cpp

5.14 FlywheelStopCommand Class Reference

```
#include <flywheel_commands.h>
```

Inheritance diagram for FlywheelStopCommand:



Public Member Functions

- FlywheelStopCommand (Flywheel &flywheel)
- bool run () override

Public Member Functions inherited from AutoCommand

- virtual void on_timeout ()
- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.14.1 Detailed Description

AutoCommand wrapper class for the stop function in the Flywheel class

5.14.2 Constructor & Destructor Documentation

5.14.2.1 FlywheelStopCommand()

```
FlywheelStopCommand::FlywheelStopCommand (
            Flywheel & flywheel )
```

Construct a FlywheelStopCommand

Parameters

flywheel	the flywheel system we are commanding

5.14.3 Member Function Documentation

5.14.3.1 run()

```
bool FlywheelStopCommand::run ( )  [override], [virtual]
```

Run stop Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

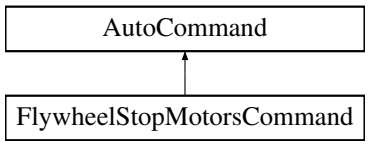
The documentation for this class was generated from the following files:

- include/utils/command_structure/flywheel_commands.h
- src/utils/command_structure/flywheel_commands.cpp

5.15 FlywheelStopMotorsCommand Class Reference

`#include <flywheel_commands.h>`

Inheritance diagram for FlywheelStopMotorsCommand:



Public Member Functions

- FlywheelStopMotorsCommand (Flywheel &flywheel)
- bool run () override

Public Member Functions inherited from AutoCommand

- virtual void on_timeout ()
- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.15.1 Detailed Description

AutoCommand wrapper class for the stopMotors function in the Flywheel class

5.15.2 Constructor & Destructor Documentation

5.15.2.1 FlywheelStopMotorsCommand()

```
FlywheelStopMotorsCommand::FlywheelStopMotorsCommand (
            Flywheel & flywheel )
```

Construct a FlywheeStopMotors Command

Parameters

flywheel	the flywheel system we are commanding

5.15.3 Member Function Documentation

5.15.3.1 run()

```
bool FlywheelStopMotorsCommand::run ( )  [override], [virtual]
```

Run stop Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

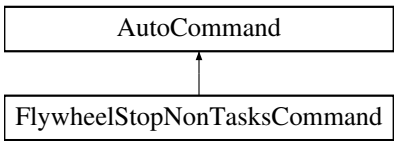
The documentation for this class was generated from the following files:

- include/utils/command_structure/flywheel_commands.h
- src/utils/command_structure/flywheel_commands.cpp

5.16 FlywheelStopNonTasksCommand Class Reference

```
#include <flywheel_commands.h>
```

Inheritance diagram for FlywheelStopNonTasksCommand:



Additional Inherited Members

Public Member Functions inherited from AutoCommand

- virtual void on_timeout ()
- AutoCommand * **withTimeout** (double t_seconds)

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.16.1 Detailed Description

AutoCommand wrapper class for the stopNonTasks function in the Flywheel class

The documentation for this class was generated from the following files:

- include/utils/command_structure/flywheel_commands.h
- src/utils/command_structure/flywheel_commands.cpp

5.17 GenericAuto Class Reference

```
#include <generic_auto.h>
```

Public Member Functions

- bool run (bool blocking)
- void add (state_ptr new_state)
- void add_async (state_ptr async_state)
- void add_delay (int ms)

5.17.1 Detailed Description

GenericAuto provides a pleasant interface for organizing an auto path steps of the path can be added with add() and when ready, calling run() will begin executing the path

5.17.2 Member Function Documentation

5.17.2.1 add()

```
void GenericAuto::add (
            state_ptr new_state )
```

Add a new state to the autonomous via function point of type "bool (ptr*)()"

Parameters

new_state	the function to run

5.17.2.2 add_async()

```
void GenericAuto::add_async (
            state_ptr async_state )
```

Add a new state to the autonomous via function point of type "bool (ptr*)()" that will run asynchronously

Parameters

async_state	the function to run

5.17.2.3 add_delay()

```
void GenericAuto::add_delay (
            int ms )
```

add_delay adds a period where the auto system will simply wait for the specified time

Parameters

ms	how long to wait in milliseconds

5.17.2.4 run()

```
bool GenericAuto::run (
            bool blocking )
```

The method that runs the autonomous. If 'blocking' is true, then this method will run through every state until it finished.

If blocking is false, then assuming every state is also non-blocking, the method will run through the current state in the list and return immediately.

Parameters

blocking	Whether or not to block the thread until all states have run

Returns

true after all states have finished.

The documentation for this class was generated from the following files:

- include/utils/generic_auto.h
- src/utils/generic_auto.cpp

5.18 GraphDrawer Class Reference

Public Member Functions

- GraphDrawer (vex::brain::lcd &screen, int num_samples, std::string x_label, std::string y_label, vex::color col, bool draw_border, double lower_bound, double upper_bound)

  a helper class to graph values on the brain screen
- void add_sample (point_t sample)
- void draw (int x, int y, int width, int height)

5.18.1 Constructor & Destructor Documentation

5.18.1.1 GraphDrawer()

```
GraphDrawer::GraphDrawer (
            vex::brain::lcd & screen,
            int num_samples,
            std::string x_label,
            std::string y_label,
            vex::color col,
            bool draw_border,
            double lower_bound,
            double upper_bound )
```

a helper class to graph values on the brain screen

Construct a GraphDrawer

Parameters

screen	a reference to Brain.screen we can save for later
num_samples	the graph works on a fixed window and will plot the last `num_samples` before the history is forgotten. Larger values give more context but may slow down if you have many graphs or an exceptionally high
x_label	the name of the x axis (currently unused)
y_label	the name of the y axis (currently unused)
draw_border	whether to draw the border around the graph. can be turned off if there are multiple graphs in the same space ie. a graph of error and output
lower_bound	the bottom of the window to graph. if lower_bound == upperbound, the graph will scale to it's datapoints
upper_bound	the top of the window to graph. if lower_bound == upperbound, the graph will scale to it's datapoints

5.18.2 Member Function Documentation

5.18.2.1 add_sample()

```
void GraphDrawer::add_sample (
            point_t sample )
```

add_sample adds a point to the graph, removing one from the back

Parameters

sample	an x, y coordinate of the next point to graph

5.18.2.2 draw()

```
void GraphDrawer::draw (
            int x,
            int y,
            int width,
            int height )
```

draws the graph to the screen in the constructor

Parameters

x	x position of the top left of the graphed region
y	y position of the top left of the graphed region
width	the width of the graphed region
height	the height of the graphed region

The documentation for this class was generated from the following files:

- include/utils/graph_drawer.h
- src/utils/graph_drawer.cpp

5.19 PurePursuit::hermite_point Struct Reference

`#include <pure_pursuit.h>`

Public Member Functions

- point_t **getPoint** ()
- Vector2D **getTangent** ()

Public Attributes

- double **x**
- double **y**
- double **dir**
- double **mag**

5.19.1 Detailed Description

a position along the hermite path contains a position and orientation information that the robot would be at at this point

The documentation for this struct was generated from the following file:

- include/utils/pure_pursuit.h

5.20 Lift< T > Class Template Reference

`#include <lift.h>`

Classes

- struct lift_cfg_t

Public Member Functions

- Lift (motor_group &lift_motors, lift_cfg_t &lift_cfg, map< T, double > &setpoint_map, limit *homing_switch=NULL)
- void control_continuous (bool up_ctrl, bool down_ctrl)
- void control_manual (bool up_btn, bool down_btn, int volt_up, int volt_down)
- void control_setpoints (bool up_step, bool down_step, vector< T > pos_list)
- bool set_position (T pos)
- bool set_setpoint (double val)
- double get_setpoint ()
- void hold ()
- void home ()
- bool get_async ()
- void set_async (bool val)
- void set_sensor_function (double(*fn_ptr)(void))
- void set_sensor_reset (void(*fn_ptr)(void))

5.20.1 Detailed Description

template<typename T>
class Lift< T >

LIFT A general class for lifts (e.g. 4bar, dr4bar, linear, etc) Uses a PID to hold the lift at a certain height under load, and to move the lift to different heights

Author

Ryan McGee

5.20.2 Constructor & Destructor Documentation

5.20.2.1 Lift()

```
template<typename T >
Lift< T >::Lift (
            motor_group & lift_motors,
            lift_cfg_t & lift_cfg,
            map< T, double > & setpoint_map,
            limit * homing_switch = NULL )  [inline]
```

Construct the Lift object and begin the background task that controls the lift.

Usage example: /code{.cpp} enum Positions {UP, MID, DOWN}; map<Positions, double> setpt_map { {DOWN, 0.0}, {MID, 0.5}, {UP, 1.0} }; Lift<Positions> my_lift(motors, lift_cfg, setpt_map); /endcode

Parameters

lift_motors	A set of motors, all set that positive rotation correlates with the lift going up
lift_cfg	Lift characterization information; PID tunings and movement speeds
setpoint_map	A map of enum type T, in which each enum entry corresponds to a different lift height

5.20.3 Member Function Documentation

5.20.3.1 control_continuous()

```
template<typename T >
void Lift< T >::control_continuous (
            bool up_ctrl,
            bool down_ctrl )  [inline]
```

Control the lift with an "up" button and a "down" button. Use PID to hold the lift when letting go.

Parameters

up_ctrl	Button controlling the "UP" motion
down_ctrl	Button controlling the "DOWN" motion

5.20.3.2 control_manual()

```
template<typename T >
void Lift< T >::control_manual (
            bool up_btn,
            bool down_btn,
            int volt_up,
            int volt_down )  [inline]
```

Control the lift with manual controls (no holding voltage)

Parameters

up_btn	Raise the lift when true
down_btn	Lower the lift when true
volt_up	Motor voltage when raising the lift
volt_down	Motor voltage when lowering the lift

5.20.3.3 control_setpoints()

```
template<typename T >
void Lift< T >::control_setpoints (
            bool up_step,
            bool down_step,
            vector< T > pos_list )  [inline]
```

Control the lift in "steps". When the "up" button is pressed, the lift will go to the next position as defined by pos_list. Order matters!

Parameters

up_step	A button that increments the position of the lift.
down_step	A button that decrements the position of the lift.
pos_list	A list of positions for the lift to go through. The higher the index, the higher the lift should be (generally).

5.20.3.4 get_async()

```
template<typename T >
bool Lift< T >::get_async ( )  [inline]
```

Returns

whether or not the background thread is running the lift

5.20.3.5 get_setpoint()

```
template<typename T >
double Lift< T >::get_setpoint ( )  [inline]
```

Returns

The current setpoint for the lift

5.20.3.6 hold()

```
template<typename T >
void Lift< T >::hold ( )  [inline]
```

Target the class's setpoint. Calculate the PID output and set the lift motors accordingly.

5.20.3.7 home()

```
template<typename T >
void Lift< T >::home ( )  [inline]
```

A blocking function that automatically homes the lift based on a sensor or hard stop, and sets the position to 0. A watchdog times out after 3 seconds, to avoid damage.

5.20.3.8 set_async()

```
template<typename T >
void Lift< T >::set_async (
            bool val )  [inline]
```

Enables or disables the background task. Note that running the control functions, or set_position functions will immediately re-enable the task for autonomous use.

Parameters

val	Whether or not the background thread should run the lift

5.20.3.9 set_position()

```
template<typename T >
bool Lift< T >::set_position (
            T pos )  [inline]
```

Enable the background task, and send the lift to a position, specified by the setpoint map from the constructor.

Parameters

pos	A lift position enum type

Returns

True if the pid has reached the setpoint

5.20.3.10 set_sensor_function()

```
template<typename T >
void Lift< T >::set_sensor_function (
            double(*)(void) fn_ptr )  [inline]
```

Creates a custom hook for any other type of sensor to be used on the lift. Example: /code{.cpp} my_lift.set_↩ sensor_function([](){return my_sensor.position();}); /endcode

Parameters

fn_ptr	Pointer to custom sensor function

5.20.3.11 set_sensor_reset()

```
template<typename T >
void Lift< T >::set_sensor_reset (
            void(*)(void) fn_ptr )  [inline]
```

Creates a custom hook to reset the sensor used in set_sensor_function(). Example: /code{.cpp} my_lift.set_↩ sensor_reset(my_sensor.resetPosition); /endcode

5.20.3.12 set_setpoint()

```
template<typename T >
bool Lift< T >::set_setpoint (
            double val )  [inline]
```

Manually set a setpoint value for the lift PID to go to.

Parameters

val	Lift setpoint, in motor revolutions or sensor units defined by get_sensor. Cannot be outside the softstops.

Returns

True if the pid has reached the setpoint

The documentation for this class was generated from the following file:

- include/subsystems/lift.h

5.21 Lift< T >::lift_cfg_t Struct Reference

`#include <lift.h>`

Public Attributes

- double **up_speed**
- double **down_speed**
- double **softstop_up**
- double **softstop_down**
- PID::pid_config_t **lift_pid_cfg**

5.21.1 Detailed Description

template<typename T>
struct Lift< T >::lift_cfg_t

lift_cfg_t holds the physical parameter specifications of a lify system. includes:

- maximum speeds for the system
- softstops to stop the lift from hitting the hard stops too hard

The documentation for this struct was generated from the following file:

- include/subsystems/lift.h

5.22 Logger Class Reference

Class to simplify writing to files.

`#include <logger.h>`

Public Member Functions

- Logger (const std::string &filename)
  Create a logger that will save to a file.
- **Logger** (const Logger &l)=delete
  copying not allowed
- Logger & **operator=** (const Logger &l)=delete
  copying not allowed
- void Log (const std::string &s)
  Write a string to the log.
- void Log (LogLevel level, const std::string &s)
  Write a string to the log with a loglevel.
- void LogIn (const std::string &s)
  Write a string and newline to the log.
- void LogIn (LogLevel level, const std::string &s)
  Write a string and a newline to the log with a loglevel.
- void Logf (const char *fmt,...)
  Write a formatted string to the log.
- void Logf (LogLevel level, const char *fmt,...)
  Write a formatted string to the log with a loglevel.

Public Attributes

- const int **MAX_FORMAT_LEN** = 512
  maximum size for a string to be before it's written

5.22.1 Detailed Description

Class to simplify writing to files.

5.22.2 Constructor & Destructor Documentation

5.22.2.1 Logger()

```
Logger::Logger (
            const std::string & filename )  [explicit]
```

Create a logger that will save to a file.

Parameters

filename	the file to save to

5.22.3 Member Function Documentation

5.22.3.1 Log() [1/2]

```
void Logger::Log (
            const std::string & s )
```

Write a string to the log.

Parameters

s	the string to write

5.22.3.2 Log() [2/2]

```
void Logger::Log (
            LogLevel level,
            const std::string & s )
```

Write a string to the log with a loglevel.

Parameters

level	the level to write. DEBUG, NOTICE, WARNING, ERROR, CRITICAL, TIME
s	the string to write

5.22.3.3 Logf() [1/2]

```
void Logger::Logf (
            const char * fmt,
             ...  )
```

Write a formatted string to the log.

Parameters

fmt	the format string (like printf)
...	the args

5.22.3.4 Logf() [2/2]

```
void Logger::Logf (
             LogLevel level,
             const char * fmt,
              ...  )
```

Write a formatted string to the log with a loglevel.

Parameters

level	the level to write. DEBUG, NOTICE, WARNING, ERROR, CRITICAL, TIME
fmt	the format string (like printf)
...	the args

5.22.3.5 Logln() [1/2]

```
void Logger::Logln (
             const std::string & s )
```

Write a string and newline to the log.

Parameters

s	the string to write

5.22.3.6 Logln() [2/2]

```
void Logger::Logln (
             LogLevel level,
             const std::string & s )
```

Write a string and a newline to the log with a loglevel.

Parameters

level	the level to write. DEBUG, NOTICE, WARNING, ERROR, CRITICAL, TIME
s	the string to write

The documentation for this class was generated from the following files:

- include/utils/logger.h
- src/utils/logger.cpp

5.23 MotionController::m_profile_cfg_t Struct Reference

```
#include <motion_controller.h>
```

Public Attributes

- double **max_v**

  the maximum velocity the robot can drive
- double **accel**

  the most acceleration the robot can do
- PID::pid_config_t **pid_cfg**

  configuration parameters for the internal PID controller
- FeedForward::ff_config_t **ff_cfg**

  configuration parameters for the internal

5.23.1 Detailed Description

m_profile_config holds all data the motion controller uses to plan paths When motion pofile is given a target to drive to, max_v and accel are used to make the trapezoid profile instructing the controller how to drive pid_cfg, ff_cfg are used to find the motor outputs necessary to execute this path

The documentation for this struct was generated from the following file:

- include/utils/motion_controller.h

5.24 MecanumDrive Class Reference

`#include <mecanum_drive.h>`

Classes

- struct mecanumdrive_config_t

Public Member Functions

- MecanumDrive (vex::motor &left_front, vex::motor &right_front, vex::motor &left_rear, vex::motor &right_rear, vex::rotation *lateral_wheel=NULL, vex::inertial *imu=NULL, mecanumdrive_config_t *config=NULL)
- void drive_raw (double direction_deg, double magnitude, double rotation)
- void drive (double left_y, double left_x, double right_x, int power=2)
- bool auto_drive (double inches, double direction, double speed, bool gyro_correction=true)
- bool auto_turn (double degrees, double speed, bool ignore_imu=false)

5.24.1 Detailed Description

A class representing the Mecanum drivetrain. Contains 4 motors, a possible IMU (intertial), and a possible undriven perpendicular wheel.

5.24.2 Constructor & Destructor Documentation

5.24.2.1 MecanumDrive()

```
MecanumDrive::MecanumDrive (
            vex::motor & left_front,
            vex::motor & right_front,
            vex::motor & left_rear,
            vex::motor & right_rear,
            vex::rotation * lateral_wheel = NULL,
            vex::inertial * imu = NULL,
            mecanumdrive_config_t * config = NULL )
```

Create the Mecanum drivetrain object

5.24.3 Member Function Documentation

5.24.3.1 auto_drive()

```
bool MecanumDrive::auto_drive (
            double inches,
            double direction,
            double speed,
            bool gyro_correction = true )
```

Drive the robot in a straight line automatically. If the inertial was declared in the constructor, use it to correct while driving. If the lateral wheel was declared in the constructor, use it for more accurate positioning while strafing.

Parameters

inches	How far the robot should drive, in inches
direction	What direction the robot should travel in, in degrees. 0 is forward, +/-180 is reverse, clockwise is positive.
speed	The maximum speed the robot should travel, in percent: -1.0->+1.0
gyro_correction	=true Whether or not to use the gyro to help correct while driving. Will always be false if no gyro was declared in the constructor.

Drive the robot in a straight line automatically. If the inertial was declared in the constructor, use it to correct while driving. If the lateral wheel was declared in the constructor, use it for more accurate positioning while strafing.

Parameters

inches	How far the robot should drive, in inches
direction	What direction the robot should travel in, in degrees. 0 is forward, +/-180 is reverse, clockwise is positive.
speed	The maximum speed the robot should travel, in percent: -1.0->+1.0
gyro_correction	= true Whether or not to use the gyro to help correct while driving. Will always be false if no gyro was declared in the constructor.

Returns

Whether or not the maneuver is complete.

5.24.3.2 auto_turn()

```
bool MecanumDrive::auto_turn (
            double degrees,
            double speed,
            bool ignore_imu = false )
```

Autonomously turn the robot X degrees over it's center point. Uses a closed loop for control.

Parameters

degrees	How many degrees to rotate the robot. Clockwise postive.
speed	What percentage to run the motors at: 0.0 -> 1.0
ignore_imu	=false Whether or not to use the Inertial for determining angle. Will instead use circumference formula + robot's wheelbase + encoders to determine.

Returns

whether or not the robot has finished the maneuver

Autonomously turn the robot X degrees over it's center point. Uses a closed loop for control.

Parameters

degrees	How many degrees to rotate the robot. Clockwise postive.
speed	What percentage to run the motors at: 0.0 -> 1.0
ignore_imu	= false Whether or not to use the Inertial for determining angle. Will instead use circumference formula + robot's wheelbase + encoders to determine.

Returns

whether or not the robot has finished the maneuver

5.24.3.3 drive()

```
void MecanumDrive::drive (
            double left_y,
            double left_x,
            double right_x,
            int power = 2 )
```

Drive the robot with a mecanum-style / arcade drive. Inputs are in percent (-100.0 -> 100.0) straight from the controller. Controls are mixed, so the robot can drive forward / strafe / rotate all at the same time.

Parameters

left_y	left joystick, Y axis (forward / backwards)
left_x	left joystick, X axis (strafe left / right)
right_x	right joystick, X axis (rotation left / right)
power	=2 how much of a "curve" there should be on drive controls; better for low speed maneuvers. Leave blank for a default curve of 2 (higher means more fidelity)

Drive the robot with a mecanum-style / arcade drive. Inputs are in percent (-100.0 -> 100.0) straight from the controller. Controls are mixed, so the robot can drive forward / strafe / rotate all at the same time.

Parameters

left_y	left joystick, Y axis (forward / backwards)
left_x	left joystick, X axis (strafe left / right)
right_x	right joystick, X axis (rotation left / right)
power	= 2 how much of a "curve" there should be on drive controls; better for low speed maneuvers. Leave blank for a default curve of 2 (higher means more fidelity)

5.24.3.4 drive_raw()

```
void MecanumDrive::drive_raw (
            double direction_deg,
            double magnitude,
            double rotation )
```

Drive the robot using vectors. This handles all the math required for mecanum control.

Parameters

direction_deg	the direction to drive the robot, in degrees. 0 is forward, 180 is back, clockwise is positive, counterclockwise is negative.
magnitude	How fast the robot should drive, in percent: 0.0->1.0
rotation	How fast the robot should rotate, in percent: -1.0->+1.0

The documentation for this class was generated from the following files:

- include/subsystems/mecanum_drive.h
- src/subsystems/mecanum_drive.cpp

5.25 MecanumDrive::mecanumdrive_config_t Struct Reference

```
#include <mecanum_drive.h>
```

Public Attributes

- PID::pid_config_t **drive_pid_conf**
- PID::pid_config_t **drive_gyro_pid_conf**
- PID::pid_config_t **turn_pid_conf**
- double **drive_wheel_diam**
- double **lateral_wheel_diam**
- double **wheelbase_width**

5.25.1 Detailed Description

Configure the Mecanum drive PID tunings and robot configurations

The documentation for this struct was generated from the following file:

- include/subsystems/mecanum_drive.h

5.26 motion_t Struct Reference

`#include <trapezoid_profile.h>`

Public Attributes

- double **pos**
  1d position at this point in time
- double **vel**
  1d velocity at this point in time
- double **accel**
  1d acceleration at this point in time

5.26.1 Detailed Description

motion_t is a description of 1 dimensional motion at a point in time.

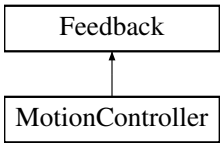
The documentation for this struct was generated from the following file:

- include/utils/trapezoid_profile.h

5.27 MotionController Class Reference

`#include <motion_controller.h>`

Inheritance diagram for MotionController:



Classes

- struct m_profile_cfg_t

Public Member Functions

- MotionController (m_profile_cfg_t &config)

  Construct a new Motion Controller object.
- void init (double start_pt, double end_pt) override

  Initialize the motion profile for a new movement This will also reset the PID and profile timers.
- double update (double sensor_val) override

  Update the motion profile with a new sensor value.
- double get () override
- void set_limits (double lower, double upper) override
- bool is_on_target () override
- motion_t get_motion ()

Public Member Functions inherited from Feedback

- virtual Feedback::FeedbackType **get_type** ()

Static Public Member Functions

- static FeedForward::ff_config_t tune_feedforward (TankDrive &drive, OdometryTank &odometry, double pct=0.6, double duration=2)

Additional Inherited Members

Public Types inherited from Feedback

- enum **FeedbackType** { **PIDType** , **FeedforwardType** , **OtherType** }

5.27.1 Detailed Description

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

5.27.2 Constructor & Destructor Documentation

5.27.2.1 MotionController()

```
MotionController::MotionController (
            m_profile_cfg_t & config )
```

Construct a new Motion Controller object.

Parameters

config	The definition of how the robot is able to move max_v Maximum velocity the movement is capable of accel Acceleration / deceleration of the movement pid_cfg Definitions of kP, kI, and kD ff_cfg Definitions of kS, kV, and kA

5.27.3 Member Function Documentation

5.27.3.1 get()

```
double MotionController::get ( )  [override], [virtual]
```

Returns

the last saved result from the feedback controller

Implements Feedback.

5.27.3.2 get_motion()

```
motion_t MotionController::get_motion ( )
```

Returns

The current postion, velocity and acceleration setpoints

5.27.3.3 init()

```
void MotionController::init (
            double start_pt,
            double end_pt )  [override], [virtual]
```

Initialize the motion profile for a new movement This will also reset the PID and profile timers.

Parameters

start↩ _pt	Movement starting position
end_pt	Movement ending posiiton

Implements Feedback.

5.27.3.4 is_on_target()

```
bool MotionController::is_on_target ( )  [override], [virtual]
```

Returns

Whether or not the movement has finished, and the PID confirms it is on target

Implements Feedback.

5.27.3.5 set_limits()

```
void MotionController::set_limits (
            double lower,
            double upper )  [override], [virtual]
```

Clamp the upper and lower limits of the output. If both are 0, no limits should be applied. if limits are applied, the controller will not target any value below lower or above upper

Parameters

lower	upper limit
upper	lower limiet

Clamp the upper and lower limits of the output. If both are 0, no limits should be applied.

Parameters

lower	Upper limit
upper	Lower limit

Implements Feedback.

5.27.3.6 tune_feedforward()

```
FeedForward::ff_config_t MotionController::tune_feedforward (
            TankDrive & drive,
            OdometryTank & odometry,
            double pct = 0.6,
            double duration = 2 )  [static]
```

This method attempts to characterize the robot's drivetrain and automatically tune the feedforward. It does this by first calculating the kS (voltage to overcome static friction) by slowly increasing the voltage until it moves.

Next is kV (voltage to sustain a certain velocity), where the robot will record it's steady-state velocity at 'pct' speed.

Finally, kA (voltage needed to accelerate by a certain rate), where the robot will record the entire movement's velocity and acceleration, record a plot of [X=(pct-kV∗V-kS), Y=(Acceleration)] along the movement, and since kA∗Accel = pct-kV∗V-kS, the reciprocal of the linear regression is the kA value.

Parameters

drive	The tankdrive to operate on
odometry	The robot's odometry subsystem
pct	Maximum velocity in percent (0->1.0)
duration	Amount of time the robot should be moving for the test

Returns

A tuned feedforward object

5.27.3.7 update()

```
double MotionController::update (
            double sensor_val )  [override], [virtual]
```

Update the motion profile with a new sensor value.

Parameters

sensor_val	Value from the sensor

Returns

the motor input generated from the motion profile

Implements Feedback.

The documentation for this class was generated from the following files:

- include/utils/motion_controller.h
- src/utils/motion_controller.cpp

5.28 MovingAverage Class Reference

```
#include <moving_average.h>
```

Public Member Functions

- MovingAverage (int buffer_size)
- MovingAverage (int buffer_size, double starting_value)
- void add_entry (double n)
- double get_average ()
- int get_size ()

5.28.1 Detailed Description

MovingAverage

A moving average is a way of smoothing out noisy data. For many sensor readings, the noise is roughly symmetric around the actual value. This means that if you collect enough samples those that are too high are cancelled out by the samples that are too low leaving the real value.

The MovingAverage class provides a simple interface to do this smoothing from our noisy sensor values.

WARNING: because we need a lot of samples to get the actual value, the value given by the MovingAverage will 'lag' behind the actual value that the sensor is reading. Using a MovingAverage is thus a tradeoff between accuracy and lag time (more samples) vs. less accuracy and faster updating (less samples).

5.28.2 Constructor & Destructor Documentation

5.28.2.1 MovingAverage() [1/2]

```
MovingAverage::MovingAverage (
            int buffer_size )
```

Create a moving average calculator with 0 as the default value

Parameters

buffer_size	The size of the buffer. The number of samples that constitute a valid reading

5.28.2.2 MovingAverage() [2/2]

```
MovingAverage::MovingAverage (
            int buffer_size,
            double starting_value )
```

Create a moving average calculator with a specified default value

Parameters

buffer_size	The size of the buffer. The number of samples that constitute a valid reading
starting_value	The value that the average will be before any data is added

5.28.3 Member Function Documentation

5.28.3.1 add_entry()

```
void MovingAverage::add_entry (
            double n )
```

Add a reading to the buffer Before: [1 1 2 2 3 3] => 2 ^ After: [2 1 2 2 3 3] => 2.16 ^

Parameters

n	the sample that will be added to the moving average.

5.28.3.2 get_average()

```
double MovingAverage::get_average ( )
```

Returns the average based off of all the samples collected so far

Returns

the calculated average. sum(samples)/numsamples

How many samples the average is made from

Returns

the number of samples used to calculate this average

5.28.3.3 get_size()

```
int MovingAverage::get_size ( )
```

How many samples the average is made from

Returns

the number of samples used to calculate this average

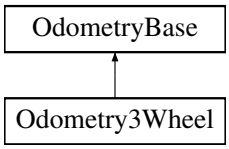
The documentation for this class was generated from the following files:

- include/utils/moving_average.h
- src/utils/moving_average.cpp

5.29 Odometry3Wheel Class Reference

```
#include <odometry_3wheel.h>
```

Inheritance diagram for Odometry3Wheel:



Classes

- struct odometry3wheel_cfg_t

Public Member Functions

- Odometry3Wheel (CustomEncoder &lside_fwd, CustomEncoder &rside_fwd, CustomEncoder &off_axis, odometry3wheel_cfg_t &cfg, bool is_async=true)
- pose_t update () override
- void tune (vex::controller &con, TankDrive &drive)

Public Member Functions inherited from OdometryBase

- OdometryBase (bool is_async)
- pose_t get_position (void)
- virtual void set_position (const pose_t &newpos=zero_pos)
- void end_async ()
- double get_speed ()
- double get_accel ()
- double get_angular_speed_deg ()
- double get_angular_accel_deg ()

Additional Inherited Members

Static Public Member Functions inherited from OdometryBase

- static int background_task (void *ptr)
- static double pos_diff (pose_t start_pos, pose_t end_pos)
- static double rot_diff (pose_t pos1, pose_t pos2)
- static double smallest_angle (double start_deg, double end_deg)

Public Attributes inherited from OdometryBase

- bool **end_task** = false

  end_task is true if we instruct the odometry thread to shut down

Static Public Attributes inherited from OdometryBase

- static constexpr pose_t zero_pos = {.x=0.0L, .y=0.0L, .rot=90.0L}

Protected Attributes inherited from OdometryBase

- vex::task * handle
- vex::mutex mut
- pose_t current_pos
- double speed
- double accel
- double ang_speed_deg
- double ang_accel_deg

5.29.1 Detailed Description

Odometry3Wheel

This class handles the code for a standard 3-pod odometry setup, where there are 3 "pods" made up of undriven (dead) wheels connected to encoders in the following configuration:

+Y ----------— ^ | | | | | | | | || O || | | | | | | | === | | ----------— | +----------—> + X

Where O is the center of rotation. The robot will monitor the changes in rotation of these wheels and calculate the robot's X, Y and rotation on the field.

This is a "set and forget" class, meaning once the object is created, the robot will immediately begin tracking it's movement in the background.

Author

Ryan McGee

Date

Oct 31 2022

5.29.2 Constructor & Destructor Documentation

5.29.2.1 Odometry3Wheel()

```
Odometry3Wheel::Odometry3Wheel (
            CustomEncoder & lside_fwd,
            CustomEncoder & rside_fwd,
            CustomEncoder & off_axis,
            odometry3wheel_cfg_t & cfg,
            bool is_async = true )
```

Construct a new Odometry 3 Wheel object

Parameters

lside_fwd	left-side encoder reference
rside_fwd	right-side encoder reference
off_axis	off-axis (perpendicular) encoder reference
cfg	robot odometry configuration
is_async	true to constantly run in the background

5.29.3 Member Function Documentation

5.29.3.1 tune()

```
void Odometry3Wheel::tune (
            vex::controller & con,
            TankDrive & drive )
```

A guided tuning process to automatically find tuning parameters. This method is blocking, and returns when tuning has finished. Follow the instructions on the controller to complete the tuning process

Parameters

con	Controller reference, for screen and button control
drive	Drivetrain reference for robot control

A guided tuning process to automatically find tuning parameters. This method is blocking, and returns when tuning has finished. Follow the instructions on the controller to complete the tuning process

It is assumed the gear ratio and encoder PPR have been set correctly

5.29.3.2 update()

```
pose_t Odometry3Wheel::update ( )  [override], [virtual]
```

Update the current position of the robot once, using the current state of the encoders and the previous known location

Returns

the robot's updated position

Implements OdometryBase.

The documentation for this class was generated from the following files:

- include/subsystems/odometry/odometry_3wheel.h
- src/subsystems/odometry/odometry_3wheel.cpp

5.30 Odometry3Wheel::odometry3wheel_cfg_t Struct Reference

`#include <odometry_3wheel.h>`

Public Attributes

- double wheelbase_dist
- double off_axis_center_dist
- double wheel_diam

5.30.1 Detailed Description

odometry3wheel_cfg_t holds all the specifications for how to calculate position with 3 encoders See the core wiki for what exactly each of these parameters measures

5.30.2 Member Data Documentation

5.30.2.1 off_axis_center_dist

`double Odometry3Wheel::odometry3wheel_cfg_t::off_axis_center_dist`

distance from the center of the robot to the center off axis wheel

5.30.2.2 wheel_diam

`double Odometry3Wheel::odometry3wheel_cfg_t::wheel_diam`

the diameter of the tracking wheel

5.30.2.3 wheelbase_dist

`double Odometry3Wheel::odometry3wheel_cfg_t::wheelbase_dist`

distance from the center of the left wheel to the center of the right wheel

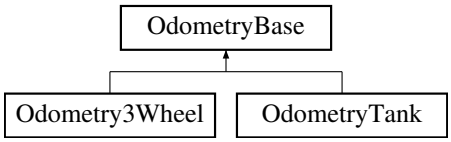
The documentation for this struct was generated from the following file:

- include/subsystems/odometry/odometry_3wheel.h

5.31 OdometryBase Class Reference

`#include <odometry_base.h>`

Inheritance diagram for OdometryBase:



Public Member Functions

- OdometryBase (bool is_async)
- pose_t get_position (void)
- virtual void set_position (const pose_t &newpos=zero_pos)
- virtual pose_t update ()=0
- void end_async ()
- double get_speed ()
- double get_accel ()
- double get_angular_speed_deg ()
- double get_angular_accel_deg ()

Static Public Member Functions

- static int background_task (void *ptr)
- static double pos_diff (pose_t start_pos, pose_t end_pos)
- static double rot_diff (pose_t pos1, pose_t pos2)
- static double smallest_angle (double start_deg, double end_deg)

Public Attributes

- bool **end_task** = false

  end_task is true if we instruct the odometry thread to shut down

Static Public Attributes

- static constexpr pose_t zero_pos = {.x=0.0L, .y=0.0L, .rot=90.0L}

Protected Attributes

- vex::task * handle
- vex::mutex mut
- pose_t current_pos
- double speed
- double accel
- double ang_speed_deg
- double ang_accel_deg

5.31.1 Detailed Description

OdometryBase

This base class contains all the shared code between different implementations of odometry. It handles the asynchronous management, position input/output and basic math functions, and holds positional types specific to field orientation.

All future odometry implementations should extend this file and redefine update() function.

Author

Ryan McGee

Date

Aug 11 2021

5.31.2 Constructor & Destructor Documentation

5.31.2.1 OdometryBase()

```
OdometryBase::OdometryBase (
            bool is_async )
```

Construct a new Odometry Base object

Parameters

is_async	True to run constantly in the background, false to call update() manually

5.31.3 Member Function Documentation

5.31.3.1 background_task()

```
int OdometryBase::background_task (
            void * ptr )  [static]
```

Function that runs in the background task. This function pointer is passed to the vex::task constructor.

Parameters

ptr	Pointer to OdometryBase object

Returns

Required integer return code. Unused.

5.31.3.2 end_async()

```
void OdometryBase::end_async ( )
```

End the background task. Cannot be restarted. If the user wants to end the thread but keep the data up to date, they must run the update() function manually from then on.

5.31.3.3 get_accel()

```
double OdometryBase::get_accel ( )
```

Get the current acceleration

Returns

the acceleration rate of the robot (inch/s^2)

5.31.3.4 get_angular_accel_deg()

```
double OdometryBase::get_angular_accel_deg ( )
```

Get the current angular acceleration in degrees

Returns

the angular acceleration at which we are turning (deg/s^2)

5.31.3.5 get_angular_speed_deg()

```
double OdometryBase::get_angular_speed_deg ( )
```

Get the current angular speed in degrees

Returns

the angular velocity at which we are turning (deg/s)

5.31.3.6 get_position()

```
pose_t OdometryBase::get_position (
            void  )
```

Gets the current position and rotation

Returns

the position that the odometry believes the robot is at

Gets the current position and rotation

5.31.3.7 get_speed()

```
double OdometryBase::get_speed ( )
```

Get the current speed

Returns

the speed at which the robot is moving and grooving (inch/s)

5.31.3.8 pos_diff()

```
double OdometryBase::pos_diff (
            pose_t start_pos,
            pose_t end_pos )  [static]
```

Get the distance between two points

Parameters

start_pos	distance from this point
end_pos	to this point

Returns

the euclidean distance between start_pos and end_pos

5.31.3.9 rot_diff()

```
double OdometryBase::rot_diff (
            pose_t pos1,
            pose_t pos2 )  [static]
```

Get the change in rotation between two points

Parameters

pos1	position with initial rotation
pos2	position with final rotation

Returns

change in rotation between pos1 and pos2

Get the change in rotation between two points

5.31.3.10 set_position()

```
void OdometryBase::set_position (
            const pose_t & newpos = zero_pos )  [virtual]
```

Sets the current position of the robot

Parameters

newpos	the new position that the odometry will believe it is at

Sets the current position of the robot

Reimplemented in OdometryTank.

5.31.3.11 smallest_angle()

```
double OdometryBase::smallest_angle (
            double start_deg,
            double end_deg )  [static]
```

Get the smallest difference in angle between a start heading and end heading. Returns the difference between -180 degrees and +180 degrees, representing the robot turning left or right, respectively.

Parameters

start_deg	intitial angle (degrees)
end_deg	final angle (degrees)

Returns

the smallest angle from the initial to the final angle. This takes into account the wrapping of rotations around 360 degrees

Get the smallest difference in angle between a start heading and end heading. Returns the difference between -180 degrees and +180 degrees, representing the robot turning left or right, respectively.

5.31.3.12 update()

```
virtual pose_t OdometryBase::update ( )  [pure virtual]
```

Update the current position on the field based on the sensors

Returns

the location that the robot is at after the odometry does its calculations

Implemented in Odometry3Wheel, and OdometryTank.

5.31.4 Member Data Documentation

5.31.4.1 accel

```
double OdometryBase::accel  [protected]
```

the rate at which we are accelerating (inch/s^2)

5.31.4.2 ang_accel_deg

```
double OdometryBase::ang_accel_deg  [protected]
```

the rate at which we are accelerating our turn (deg/s^2)

5.31.4.3 ang_speed_deg

```
double OdometryBase::ang_speed_deg  [protected]
```

the speed at which we are turning (deg/s)

5.31.4.4 current_pos

```
pose_t OdometryBase::current_pos  [protected]
```

Current position of the robot in terms of x,y,rotation

5.31.4.5 handle

```
vex::task* OdometryBase::handle  [protected]
```

handle to the vex task that is running the odometry code

5.31.4.6 mut

```
vex::mutex OdometryBase::mut  [protected]
```

Mutex to control multithreading

5.31.4.7 speed

```
double OdometryBase::speed  [protected]
```

the speed at which we are travelling (inch/s)

5.31.4.8 zero_pos

```
constexpr pose_t OdometryBase::zero_pos = {.x=0.0L, .y=0.0L, .rot=90.0L}  [inline], [static],
[constexpr]
```

Zeroed position. X=0, Y=0, Rotation= 90 degrees

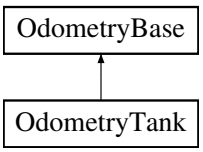
The documentation for this class was generated from the following files:

- include/subsystems/odometry/odometry_base.h
- src/subsystems/odometry/odometry_base.cpp

5.32 OdometryTank Class Reference

```
#include <odometry_tank.h>
```

Inheritance diagram for OdometryTank:



Public Member Functions

- OdometryTank (vex::motor_group &left_side, vex::motor_group &right_side, robot_specs_t &config, vex↩ ::inertial ∗imu=NULL, bool is_async=true)
- OdometryTank (CustomEncoder &left_enc, CustomEncoder &right_enc, robot_specs_t &config, vex::inertial ∗imu=NULL, bool is_async=true)
- pose_t update () override
- void set_position (const pose_t &newpos=zero_pos) override

Public Member Functions inherited from OdometryBase

- OdometryBase (bool is_async)
- pose_t get_position (void)
- void end_async ()
- double get_speed ()
- double get_accel ()
- double get_angular_speed_deg ()
- double get_angular_accel_deg ()

Additional Inherited Members

Static Public Member Functions inherited from OdometryBase

- static int background_task (void ∗ptr)
- static double pos_diff (pose_t start_pos, pose_t end_pos)
- static double rot_diff (pose_t pos1, pose_t pos2)
- static double smallest_angle (double start_deg, double end_deg)

Public Attributes inherited from OdometryBase

- bool **end_task** = false

  end_task is true if we instruct the odometry thread to shut down

Static Public Attributes inherited from OdometryBase

- static constexpr pose_t zero_pos = {.x=0.0L, .y=0.0L, .rot=90.0L}

Protected Attributes inherited from OdometryBase

- vex::task * handle
- vex::mutex mut
- pose_t current_pos
- double speed
- double accel
- double ang_speed_deg
- double ang_accel_deg

5.32.1 Detailed Description

OdometryTank defines an odometry system for a tank drivetrain This requires encoders in the same orientation as the drive wheels Odometry is a "start and forget" subsystem, which means once it's created and configured, it will constantly run in the background and track the robot's X, Y and rotation coordinates.

5.32.2 Constructor & Destructor Documentation

5.32.2.1 OdometryTank() [1/2]

```
OdometryTank::OdometryTank (
            vex::motor_group & left_side,
            vex::motor_group & right_side,
            robot_specs_t & config,
            vex::inertial * imu = NULL,
            bool is_async = true )
```

Initialize the Odometry module, calculating position from the drive motors.

Parameters

left_side	The left motors
right_side	The right motors
config	the specifications that supply the odometry with descriptions of the robot. See robot_specs_t for what is contained
imu	The robot's inertial sensor. If not included, rotation is calculated from the encoders.
is_async	If true, position will be updated in the background continuously. If false, the programmer will have to manually call update().

5.32.2.2 OdometryTank() [2/2]

```
OdometryTank::OdometryTank (
            CustomEncoder & left_enc,
            CustomEncoder & right_enc,
            robot_specs_t & config,
            vex::inertial * imu = NULL,
            bool is_async = true )
```

Initialize the Odometry module, calculating position from the drive motors.

Parameters

left_enc	The left motors
right_enc	The right motors
config	the specifications that supply the odometry with descriptions of the robot. See robot_specs_t for what is contained
imu	The robot's inertial sensor. If not included, rotation is calculated from the encoders.
is_async	If true, position will be updated in the background continuously. If false, the programmer will have to manually call update().

5.32.3 Member Function Documentation

5.32.3.1 set_position()

```
void OdometryTank::set_position (
            const pose_t & newpos = zero_pos )  [override], [virtual]
```

set_position tells the odometry to place itself at a position

Parameters

newpos	the position the odometry will take

Resets the position and rotational data to the input.

Reimplemented from OdometryBase.

5.32.3.2 update()

```
pose_t OdometryTank::update ( )  [override], [virtual]
```

Update the current position on the field based on the sensors

Returns

the position that odometry has calculated itself to be at

Update, store and return the current position of the robot. Only use if not initializing with a separate thread.

Implements OdometryBase.

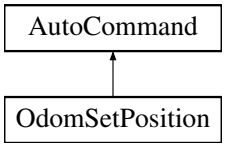
The documentation for this class was generated from the following files:

- include/subsystems/odometry/odometry_tank.h
- src/subsystems/odometry/odometry_tank.cpp

5.33 OdomSetPosition Class Reference

`#include <drive_commands.h>`

Inheritance diagram for OdomSetPosition:



Public Member Functions

- OdomSetPosition (OdometryBase &odom, const pose_t &newpos=OdometryBase::zero_pos)
- bool run () override

Public Member Functions inherited from AutoCommand

- virtual void on_timeout ()
- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.33.1 Detailed Description

AutoCommand wrapper class for the set_position function in the Odometry class

5.33.2 Constructor & Destructor Documentation

5.33.2.1 OdomSetPosition()

```
OdomSetPosition::OdomSetPosition (
            OdometryBase & odom,
            const pose_t & newpos = OdometryBase::zero_pos )
```

constructs a new OdomSetPosition command

Parameters

odom	the odometry system we are setting
newpos	the position we are telling the odometry to take. defaults to (0, 0), angle = 90

Construct an Odometry set pos

Parameters

odom	the odometry system we are setting
newpos	the now position to set the odometry to

5.33.3 Member Function Documentation

5.33.3.1 run()

```
bool OdomSetPosition::run ( )  [override], [virtual]
```

Run set_position Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

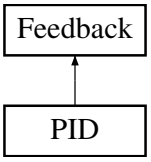
The documentation for this class was generated from the following files:

- include/utils/command_structure/drive_commands.h
- src/utils/command_structure/drive_commands.cpp

5.34 PID Class Reference

```
#include <pid.h>
```

Inheritance diagram for PID:



Classes

- struct pid_config_t

Public Types

- enum ERROR_TYPE { **LINEAR** , **ANGULAR** }

Public Types inherited from Feedback

- enum **FeedbackType** { **PIDType** , **FeedforwardType** , **OtherType** }

Public Member Functions

- PID (pid_config_t &config)
- void init (double start_pt, double set_pt) override
- double update (double sensor_val) override
- double get () override
- void set_limits (double lower, double upper) override
- bool is_on_target () override
- void reset ()
- double get_error ()
- double get_target ()
- void set_target (double target)
- Feedback::FeedbackType get_type () override

Public Attributes

- pid_config_t & **config**

  configuration struct for this controller. see pid_config_t for information about what this contains

5.34.1 Detailed Description

PID Class

Defines a standard feedback loop using the constants kP, kI, kD, deadband, and on_target_time. The formula is:

out = kP∗error + kI∗integral(d Error) + kD∗(dError/dt)

The PID object will determine it is "on target" when the error is within the deadband, for a duration of on_target_time

Author

Ryan McGee

Date

4/3/2020

5.34.2 Member Enumeration Documentation

5.34.2.1 ERROR_TYPE

```
enum PID::ERROR_TYPE
```

An enum to distinguish between a linear and angular caluclation of PID error.

5.34.3 Constructor & Destructor Documentation

5.34.3.1 PID()

```
PID::PID (
            pid_config_t & config )
```

Create the PID object

Parameters

config	the configuration data for this controller

Create the PID object

5.34.4 Member Function Documentation

5.34.4.1 get()

```
double PID::get ( )  [override], [virtual]
```

Gets the current PID out value, from when update() was last run

Returns

the Out value of the controller (voltage, RPM, whatever the PID controller is controlling)

Gets the current PID out value, from when update() was last run

Implements Feedback.

5.34.4.2 get_error()

```
double PID::get_error ( )
```

Get the delta between the current sensor data and the target

Returns

the error calculated. how it is calculated depends on error_method specified in pid_config_t

Get the delta between the current sensor data and the target

5.34.4.3 get_target()

```
double PID::get_target ( )
```

Get the PID's target

Returns

the target the PID controller is trying to achieve

5.34.4.4 get_type()

```
Feedback::FeedbackType PID::get_type ( )  [override], [virtual]
```

Reimplemented from Feedback.

5.34.4.5 init()

```
void PID::init (
            double start_pt,
            double set_pt )  [override], [virtual]
```

Inherited from Feedback for interoperability. Update the setpoint and reset integral accumulation

start_pt can be safely ignored in this feedback controller

Parameters

start_pt	commpletely ignored for PID. necessary to satisfy Feedback base
set_pt	sets the target of the PID controller

Implements Feedback.

5.34.4.6 is_on_target()

```
bool PID::is_on_target ( )  [override], [virtual]
```

Checks if the PID controller is on target.

Returns

true if the loop is within [deadband] for [on_target_time] seconds

Returns true if the loop is within [deadband] for [on_target_time] seconds

Implements Feedback.

5.34.4.7 reset()

```
void PID::reset ( )
```

Reset the PID loop by resetting time since 0 and accumulated error.

5.34.4.8 set_limits()

```
void PID::set_limits (
            double lower,
            double upper )  [override], [virtual]
```

Set the limits on the PID out. The PID out will "clip" itself to be between the limits.

Parameters

lower	the lower limit. the PID controller will never command the output go below `lower`
upper	the upper limit. the PID controller will never command the output go higher than `upper`

Set the limits on the PID out. The PID out will "clip" itself to be between the limits.

Implements Feedback.

5.34.4.9 set_target()

```
void PID::set_target (
            double target )
```

Set the target for the PID loop, where the robot is trying to end up

Parameters

target	the sensor reading we would like to achieve

Set the target for the PID loop, where the robot is trying to end up

5.34.4.10 update()

```
double PID::update (
            double sensor_val )  [override], [virtual]
```

Update the PID loop by taking the time difference from last update, and running the PID formula with the new sensor data

Parameters

sensor_val	the distance, angle, encoder position or whatever it is we are measuring

Returns

the new output. What would be returned by PID::get()

Implements Feedback.

The documentation for this class was generated from the following files:

- include/utils/pid.h
- src/utils/pid.cpp

5.35 PID::pid_config_t Struct Reference

```
#include <pid.h>
```

Public Attributes

- double **p**

  proportional coeffecient p ∗ error()
- double **i**

  integral coeffecient i ∗ integral(error)
- double **d**

derivitave coeffecient d ∗ derivative(error)

- double **deadband**

  at what threshold are we close enough to be finished

- double **on_target_time**

  the time in seconds that we have to be on target for to say we are officially at the target

- ERROR_TYPE **error_method**

  Linear or angular. wheter to do error as a simple subtraction or to wrap.

5.35.1 Detailed Description

pid_config_t holds the configuration parameters for a pid controller In addtion to the constant of proportional, integral and derivative, these parameters include:

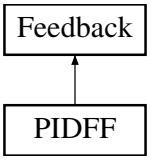
- deadband -
- on_target_time - for how long do we have to be at the target to stop As well, pid_config_t holds an error type which determines whether errors should be calculated as if the sensor position is a measure of distance or an angle

The documentation for this struct was generated from the following file:

- include/utils/pid.h

5.36 PIDFF Class Reference

Inheritance diagram for PIDFF:



Public Member Functions

- **PIDFF** (PID::pid_config_t &pid_cfg, FeedForward::ff_config_t &ff_cfg)
- void init (double start_pt, double set_pt) override
- void set_target (double set_pt)
- double update (double val) override
- double update (double val, double vel_setpt, double a_setpt=0)
- double get () override
- void set_limits (double lower, double upper) override
- bool is_on_target () override

Public Member Functions inherited from Feedback

- virtual Feedback::FeedbackType **get_type** ()

Public Attributes

- PID **pid**

Additional Inherited Members

Public Types inherited from Feedback

- enum **FeedbackType** { **PIDType** , **FeedforwardType** , **OtherType** }

5.36.1 Member Function Documentation

5.36.1.1 get()

```
double PIDFF::get ( )  [override], [virtual]
```

Returns

the last saved result from the feedback controller

Implements Feedback.

5.36.1.2 init()

```
void PIDFF::init (
            double start_pt,
            double set_pt )  [override], [virtual]
```

Initialize the feedback controller for a movement

Parameters

start_pt	the current sensor value
set_pt	where the sensor value should be

Implements Feedback.

5.36.1.3 is_on_target()

```
bool PIDFF::is_on_target ( )  [override], [virtual]
```

Returns

true if the feedback controller has reached it's setpoint

Implements Feedback.

5.36.1.4 set_limits()

```
void PIDFF::set_limits (
            double lower,
            double upper )  [override], [virtual]
```

Clamp the upper and lower limits of the output. If both are 0, no limits should be applied.

Parameters

lower	Upper limit
upper	Lower limit

Implements Feedback.

5.36.1.5 set_target()

```
void PIDFF::set_target (
            double set_pt )
```

Set the target of the PID loop

Parameters

set_pt	Setpoint / target value

5.36.1.6 update() [1/2]

```
double PIDFF::update (
            double val )  [override], [virtual]
```

Iterate the feedback loop once with an updated sensor value. Only kS for feedfoward will be applied.

Parameters

val	value from the sensor

Returns

feedback loop result

Implements Feedback.

5.36.1.7 update() [2/2]

```
double PIDFF::update (
            double val,
```

```
            double vel_setpt,
            double a_setpt = 0 )
```

Iterate the feedback loop once with an updated sensor value

Parameters

val	value from the sensor
vel_setpt	Velocity for feedforward
a_setpt	Acceleration for feedfoward

Returns

feedback loop result

The documentation for this class was generated from the following files:

- include/utils/pidff.h
- src/utils/pidff.cpp

5.37 point_t Struct Reference

`#include <geometry.h>`

Public Member Functions

- double dist (const point_t other)
- point_t operator+ (const point_t &other)
- point_t operator- (const point_t &other)

Public Attributes

- double **x**

  the x position in space
- double **y**

  the y position in space

5.37.1 Detailed Description

Data structure representing an X,Y coordinate

5.37.2 Member Function Documentation

5.37.2.1 dist()

```
double point_t::dist (
            const point_t other )  [inline]
```

dist calculates the euclidian distance between this point and another point using the pythagorean theorem

Parameters

other	the point to measure the distance from

Returns

the euclidian distance between this and other

5.37.2.2 operator+()

```
point_t point_t::operator+ (
            const point_t & other )  [inline]
```

Vector2D addition operation on points

Parameters

other	the point to add on to this

Returns

this + other (this.x + other.x, this.y + other.y)

5.37.2.3 operator-()

```
point_t point_t::operator- (
            const point_t & other )  [inline]
```

Vector2D subtraction operation on points

Parameters

other	the point_t to subtract from this

Returns

this - other (this.x - other.x, this.y - other.y)

The documentation for this struct was generated from the following file:

- include/utils/geometry.h

5.38 pose_t Struct Reference

```
#include <geometry.h>
```

Public Attributes

- double **x**

  x position in the world
- double **y**

  y position in the world
- double **rot**

  rotation in the world

5.38.1 Detailed Description

Describes a single position and rotation

The documentation for this struct was generated from the following file:

- include/utils/geometry.h

5.39 robot_specs_t Struct Reference

`#include <robot_specs.h>`

Public Attributes

- double **robot_radius**

  if you were to draw a circle with this radius, the robot would be entirely contained within it
- double **odom_wheel_diam**

  the diameter of the wheels used for
- double **odom_gear_ratio**

  the ratio of the odometry wheel to the encoder reading odometry data
- double **dist_between_wheels**

  the distance between centers of the central drive wheels
- double **drive_correction_cutoff**

  the distance at which to stop trying to turn towards the target. If we are less than this value, we can continue driving forward to minimize our distance but will not try to spin around to point directly at the target
- Feedback * **drive_feedback**

  the default feedback for autonomous driving
- Feedback * **turn_feedback**

  the defualt feedback for autonomous turning
- PID::pid_config_t **correction_pid**

  the pid controller to keep the robot driving in as straight a line as possible

5.39.1 Detailed Description

Main robot characterization struct. This will be passed to all the major subsystems that require info about the robot. All distance measurements are in inches.

The documentation for this struct was generated from the following file:

- include/robot_specs.h

5.40 Serializer Class Reference

Serializes Arbitrary data to a file on the SD Card.

`#include <serializer.h>`

Public Member Functions

- ~**Serializer** ()

  Save and close upon destruction (bc of vex, this doesnt always get called when the program ends. To be sure, call save_to_disk)
- Serializer (const std::string &filename, bool flush_always=true)

  create a Serializer
- void save_to_disk () const

  saves current Serializer state to disk
- void set_int (const std::string &name, int i)

  Setters - not saved until save_to_disk is called.
- void set_bool (const std::string &name, bool b)

  sets a bool by the name of name to b. If flush_always == true, this will save to the sd card
- void set_double (const std::string &name, double d)

  sets a double by the name of name to d. If flush_always == true, this will save to the sd card
- void set_string (const std::string &name, std::string str)

  sets a string by the name of name to s. If flush_always == true, this will save to the sd card
- int int_or (const std::string &name, int otherwise)

  gets a value stored in the serializer. If not found, sets the value to otherwise
- bool bool_or (const std::string &name, bool otherwise)

  gets a value stored in the serializer. If not, sets the value to otherwise
- double double_or (const std::string &name, double otherwise)

  gets a value stored in the serializer. If not, sets the value to otherwise
- std::string string_or (const std::string &name, std::string otherwise)

  gets a value stored in the serializer. If not, sets the value to otherwise

5.40.1 Detailed Description

Serializes Arbitrary data to a file on the SD Card.

5.40.2 Constructor & Destructor Documentation

5.40.2.1 Serializer()

```
Serializer::Serializer (
            const std::string & filename,
            bool flush_always = true )  [inline], [explicit]
```

create a Serializer

Parameters

filename	the file to read from. If filename does not exist we will create that file
flush_always	If true, after every write flush to a file. If false, you are responsible for calling save_to_disk

5.40.3 Member Function Documentation

5.40.3.1 bool_or()

```
bool Serializer::bool_or (
            const std::string & name,
            bool otherwise )
```

gets a value stored in the serializer. If not, sets the value to otherwise

Parameters

name	name of value
otherwise	value if the name is not specified

Returns

the value if found or otherwise

5.40.3.2 double_or()

```
double Serializer::double_or (
            const std::string & name,
            double otherwise )
```

gets a value stored in the serializer. If not, sets the value to otherwise

Parameters

name	name of value
otherwise	value if the name is not specified

Returns

the value if found or otherwise

5.40.3.3 int_or()

```
int Serializer::int_or (
            const std::string & name,
            int otherwise )
```

gets a value stored in the serializer. If not found, sets the value to otherwise

Getters Return value if it exists in the serializer

Parameters

name	name of value
otherwise	value if the name is not specified

Returns

the value if found or otherwise

5.40.3.4 save_to_disk()

```
void Serializer::save_to_disk ( ) const
```

saves current Serializer state to disk

forms data bytes then saves to filename this was openned with

5.40.3.5 set_bool()

```
void Serializer::set_bool (
            const std::string & name,
            bool b )
```

sets a bool by the name of name to b. If flush_always == true, this will save to the sd card

Parameters

name	name of bool
b	value of bool

5.40.3.6 set_double()

```
void Serializer::set_double (
            const std::string & name,
            double d )
```

sets a double by the name of name to d. If flush_always == true, this will save to the sd card

Parameters

name	name of double
d	value of double

5.40.3.7 set_int()

```
void Serializer::set_int (
```

```
            const std::string & name,
            int i )
```

Setters - not saved until save_to_disk is called.

sets an integer by the name of name to i. If flush_always == true, this will save to the sd card

Parameters

name	name of integer
i	value of integer

5.40.3.8 set_string()

```
void Serializer::set_string (
            const std::string & name,
            std::string str )
```

sets a string by the name of name to s. If flush_always == true, this will save to the sd card

Parameters

name	name of string
i	value of string

5.40.3.9 string_or()

```
std::string Serializer::string_or (
            const std::string & name,
            std::string otherwise )
```

gets a value stored in the serializer. If not, sets the value to otherwise

Parameters

name	name of value
otherwise	value if the name is not specified

Returns

the value if found or otherwise

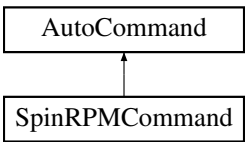
The documentation for this class was generated from the following files:

- include/utils/serializer.h
- src/utils/serializer.cpp

5.41 SpinRPMCommand Class Reference

`#include <flywheel_commands.h>`

Inheritance diagram for SpinRPMCommand:



Public Member Functions

- SpinRPMCommand (Flywheel &flywheel, int rpm)
- bool run () override

Public Member Functions inherited from AutoCommand

- virtual void on_timeout ()
- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.41.1 Detailed Description

File: flywheel_commands.h Desc: [insert meaningful desc] AutoCommand wrapper class for the spinRPM function in the Flywheel class

5.41.2 Constructor & Destructor Documentation

5.41.2.1 SpinRPMCommand()

```
SpinRPMCommand::SpinRPMCommand (
            Flywheel & flywheel,
            int rpm )
```

Construct a SpinRPM Command

Parameters

flywheel	the flywheel sys to command
rpm	the rpm that we should spin at

File: flywheel_commands.cpp Desc: [insert meaningful desc]

5.41.3 Member Function Documentation

5.41.3.1 run()

```
bool SpinRPMCommand::run ( )  [override], [virtual]
```

Run spin_manual Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

The documentation for this class was generated from the following files:

- include/utils/command_structure/flywheel_commands.h
- src/utils/command_structure/flywheel_commands.cpp

5.42 PurePursuit::spline Struct Reference

```
#include <pure_pursuit.h>
```

Public Member Functions

- double **getY** (double x)

Public Attributes

- double **a**
- double **b**
- double **c**
- double **d**
- double **x_start**
- double **x_end**

5.42.1 Detailed Description

Represents a piece of a cubic spline with s(x) = a(x-xi)^3 + b(x-xi)^2 + c(x-xi) + d The x_start and x_end shows where the equation is valid.

The documentation for this struct was generated from the following file:

- include/utils/pure_pursuit.h

5.43 TankDrive Class Reference

`#include <tank_drive.h>`

Public Member Functions

- TankDrive (motor_group &left_motors, motor_group &right_motors, robot_specs_t &config, OdometryBase *odom=NULL)
- void stop ()
- void drive_tank (double left, double right, int power=1, bool isdriver=false)
- void drive_arcade (double forward_back, double left_right, int power=1)
- bool drive_forward (double inches, directionType dir, Feedback &feedback, double max_speed=1)
- bool drive_forward (double inches, directionType dir, double max_speed=1)
- bool turn_degrees (double degrees, Feedback &feedback, double max_speed=1)
- bool turn_degrees (double degrees, double max_speed=1)
- bool drive_to_point (double x, double y, vex::directionType dir, Feedback &feedback, double max_speed=1)
- bool drive_to_point (double x, double y, vex::directionType dir, double max_speed=1)
- bool turn_to_heading (double heading_deg, Feedback &feedback, double max_speed=1)
- bool turn_to_heading (double heading_deg, double max_speed=1)
- void reset_auto ()
- bool pure_pursuit (std::vector< PurePursuit::hermite_point > path, directionType dir, double radius, double res, Feedback &feedback, double max_speed=1)

Static Public Member Functions

- static double modify_inputs (double input, int power=2)

5.43.1 Detailed Description

TankDrive is a class to run a tank drive system. A tank drive system, sometimes called differential drive, has a motor (or group of synchronized motors) on the left and right side

5.43.2 Constructor & Destructor Documentation

5.43.2.1 TankDrive()

```
TankDrive::TankDrive (
            motor_group & left_motors,
            motor_group & right_motors,
            robot_specs_t & config,
            OdometryBase * odom = NULL )
```

Create the TankDrive object

Parameters

left_motors	left side drive motors
right_motors	right side drive motors
config	the configuration specification defining physical dimensions about the robot. See robot_specs_t for more info
odom	an odometry system to track position and rotation. this is necessary to execute autonomous paths

5.43.3 Member Function Documentation

5.43.3.1 drive_arcade()

```
void TankDrive::drive_arcade (
            double forward_back,
            double left_right,
            int power = 1 )
```

Drive the robot using arcade style controls. forward_back controls the linear motion, left_right controls the turning.

forward_back and left_right are in "percent": -1.0 -> 1.0

Parameters

forward_back	the percent to move forward or backward
left_right	the percent to turn left or right
power	modifies the input velocities left^power, right^power

Drive the robot using arcade style controls. forward_back controls the linear motion, left_right controls the turning.

left_motors and right_motors are in "percent": -1.0 -> 1.0

5.43.3.2 drive_forward() [1/2]

```
bool TankDrive::drive_forward (
            double inches,
            directionType dir,
            double max_speed = 1 )
```

Autonomously drive the robot forward a certain distance

Parameters

inches	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
dir	the direction we want to travel forward and backward
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Autonomously drive the robot forward a certain distance

Parameters

inches	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
dir	the direction we want to travel forward and backward
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Returns

true if we have finished driving to our point

5.43.3.3 drive_forward() [2/2]

```
bool TankDrive::drive_forward (
            double inches,
            directionType dir,
            Feedback & feedback,
            double max_speed = 1 )
```

Use odometry to drive forward a certain distance using a custom feedback controller

Returns whether or not the robot has reached it's destination.

Parameters

inches	the distance to drive forward
dir	the direction we want to travel forward and backward
feedback	the custom feedback controller we will use to travel. controls the rate at which we accelerate and drive.
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Returns

true when we have reached our target distance

Use odometry to drive forward a certain distance using a custom feedback controller

Returns whether or not the robot has reached it's destination.

Parameters

inches	the distance to drive forward
dir	the direction we want to travel forward and backward
feedback	the custom feedback controller we will use to travel. controls the rate at which we accelerate and drive.
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

5.43.3.4 drive_tank()

```
void TankDrive::drive_tank (
```

```
            double left,
            double right,
            int power = 1,
            bool isdriver = false )
```

Drive the robot using differential style controls. left_motors controls the left motors, right_motors controls the right motors.

left_motors and right_motors are in "percent": -1.0 -> 1.0

Parameters

left	the percent to run the left motors
right	the percent to run the right motors
power	modifies the input velocities left^power, right^power
isdriver	default false. if true uses motor percentage. if false uses plain percentage of maximum voltage

Drive the robot using differential style controls. left_motors controls the left motors, right_motors controls the right motors.

left_motors and right_motors are in "percent": -1.0 -> 1.0

5.43.3.5 drive_to_point() [1/2]

```
bool TankDrive::drive_to_point (
            double x,
            double y,
            vex::directionType dir,
            double max_speed = 1 )
```

Use odometry to automatically drive the robot to a point on the field. X and Y is the final point we want the robot. Here we use the default feedback controller from the drive_sys

Returns whether or not the robot has reached it's destination.

Parameters

x	the x position of the target
y	the y position of the target
dir	the direction we want to travel forward and backward
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Use odometry to automatically drive the robot to a point on the field. X and Y is the final point we want the robot. Here we use the default feedback controller from the drive_sys

Returns whether or not the robot has reached it's destination.

Parameters

x	the x position of the target
y	the y position of the target
dir	the direction we want to travel forward and backward
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Returns

true if we have reached our target point

5.43.3.6 drive_to_point() [2/2]

```
bool TankDrive::drive_to_point (
            double x,
            double y,
            vex::directionType dir,
            Feedback & feedback,
            double max_speed = 1 )
```

Use odometry to automatically drive the robot to a point on the field. X and Y is the final point we want the robot.

Returns whether or not the robot has reached it's destination.

Parameters

x	the x position of the target
y	the y position of the target
dir	the direction we want to travel forward and backward
feedback	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Use odometry to automatically drive the robot to a point on the field. X and Y is the final point we want the robot.

Returns whether or not the robot has reached it's destination.

Parameters

x	the x position of the target
y	the y position of the target
dir	the direction we want to travel forward and backward
feedback	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Returns

true if we have reached our target point

5.43.3.7 modify_inputs()

```
double TankDrive::modify_inputs (
            double input,
            int power = 2 )  [static]
```

Create a curve for the inputs, so that drivers have more control at lower speeds. Curves are exponential, with the default being squaring the inputs.

Parameters

input	the input before modification
power	the power to raise input to

Returns

input ^ power (accounts for negative inputs and odd numbered powers)

Modify the inputs from the controller by squaring / cubing, etc Allows for better control of the robot at slower speeds

Parameters

input	the input signal -1 -> 1
power	the power to raise the signal to

Returns

input^power accounting for any sign issues that would arise with this naive solution

5.43.3.8 pure_pursuit()

```
bool TankDrive::pure_pursuit (
            std::vector< PurePursuit::hermite_point > path,
            directionType dir,
            double radius,
            double res,
            Feedback & feedback,
            double max_speed = 1 )
```

Follow a hermite curve using the pure pursuit algorithm.

Parameters

path	The hermite curve for the robot to take. Must have 2 or more points.
dir	Whether the robot should move forward or backwards
radius	How the pure pursuit radius, in inches, for finding the lookahead point
res	The number of points to use along the path; the hermite curve is split up into "res" individual points.
feedback	The feedback controller to use
max_speed	Robot's maximum speed throughout the path, between 0 and 1.0

Returns

true when we reach the end of the path

5.43.3.9 reset_auto()

```
void TankDrive::reset_auto ( )
```

Reset the initialization for autonomous drive functions

5.43.3.10 stop()

```
void TankDrive::stop ( )
```

Stops rotation of all the motors using their "brake mode"

5.43.3.11 turn_degrees() [1/2]

```
bool TankDrive::turn_degrees (
            double degrees,
            double max_speed = 1 )
```

Autonomously turn the robot X degrees to counterclockwise (negative for clockwise), with a maximum motor speed of percent_speed (-1.0 -> 1.0)

Uses the defualt turning feedback of the drive system.

Parameters

degrees	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Autonomously turn the robot X degrees to counterclockwise (negative for clockwise), with a maximum motor speed of percent_speed (-1.0 -> 1.0)

Uses the defualt turning feedback of the drive system.

Parameters

degrees	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Returns

true if we turned te target number of degrees

5.43.3.12 turn_degrees() [2/2]

```
bool TankDrive::turn_degrees (
            double degrees,
            Feedback & feedback,
            double max_speed = 1 )
```

Autonomously turn the robot X degrees counterclockwise (negative for clockwise), with a maximum motor speed of percent_speed (-1.0 -> 1.0)

Uses PID + Feedforward for it's control.

Parameters

degrees	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
feedback	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Autonomously turn the robot X degrees to counterclockwise (negative for clockwise), with a maximum motor speed of percent_speed (-1.0 -> 1.0)

Uses the specified feedback for it's control.

Parameters

degrees	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
feedback	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Returns

true if we have turned our target number of degrees

5.43.3.13 turn_to_heading() [1/2]

```
bool TankDrive::turn_to_heading (
            double heading_deg,
            double max_speed = 1 )
```

Turn the robot in place to an exact heading relative to the field. 0 is forward. Uses the defualt turn feedback of the drive system

Parameters

heading_deg	the heading to which we will turn
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Turn the robot in place to an exact heading relative to the field. 0 is forward. Uses the defualt turn feedback of the drive system

Parameters

heading_deg	the heading to which we will turn
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Returns

true if we have reached our target heading

5.43.3.14 turn_to_heading() [2/2]

```
bool TankDrive::turn_to_heading (
            double heading_deg,
            Feedback & feedback,
            double max_speed = 1 )
```

Turn the robot in place to an exact heading relative to the field. 0 is forward.

Parameters

heading_deg	the heading to which we will turn
feedback	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Turn the robot in place to an exact heading relative to the field. 0 is forward.

Parameters

heading_deg	the heading to which we will turn
feedback	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
max_speed	the maximum percentage of robot speed at which the robot will travel. 1 = full power

Returns

true if we have reached our target heading

The documentation for this class was generated from the following files:

- include/subsystems/tank_drive.h
- src/subsystems/tank_drive.cpp

5.44 TrapezoidProfile Class Reference

```
#include <trapezoid_profile.h>
```

Public Member Functions

- TrapezoidProfile (double max_v, double accel)

  Construct a new Trapezoid Profile object.
- motion_t calculate (double time_s)

  Run the trapezoidal profile based on the time that's ellapsed.
- void set_endpts (double start, double end)
- void set_accel (double accel)
- void set_max_v (double max_v)
- double get_movement_time ()

5.44.1 Detailed Description

Trapezoid Profile

This is a motion profile defined by an acceleration, maximum velocity, start point and end point. Using this information, a parametric function is generated, with a period of acceleration, constant velocity, and deceleration. The velocity graph looks like a trapezoid, giving it it's name.

If the maximum velocity is set high enough, this will become a S-curve profile, with only acceleration and deceleration.

This class is designed for use in properly modelling the motion of the robots to create a feedfoward and target for PID. Acceleration and Maximum velocity should be measured on the robot and tuned down slightly to account for battery drop.

Here are the equations graphed for ease of understanding: https://www.desmos.com/calculator/rkm3ivu1yk

Author

Ryan McGee

Date

7/12/2022

5.44.2 Constructor & Destructor Documentation

5.44.2.1 TrapezoidProfile()

```
TrapezoidProfile::TrapezoidProfile (
            double max_v,
            double accel )
```

Construct a new Trapezoid Profile object.

Parameters

max_v	Maximum velocity the robot can run at
accel	Maximum acceleration of the robot

5.44.3 Member Function Documentation

5.44.3.1 calculate()

```
motion_t TrapezoidProfile::calculate (
            double time_s )
```

Run the trapezoidal profile based on the time that's ellapsed.

Parameters

time_s	Time since start of movement

Returns

motion_t Position, velocity and acceleration

5.44.3.2 get_movement_time()

```
double TrapezoidProfile::get_movement_time ( )
```

uses the kinematic equations to and specified accel and max_v to figure out how long moving along the profile would take

Returns

the time the path will take to travel

5.44.3.3 set_accel()

```
void TrapezoidProfile::set_accel (
            double accel )
```

set_accel sets the acceleration this profile will use (the left and right legs of the trapezoid)

Parameters

accel	the acceleration amount to use

5.44.3.4 set_endpts()

```
void TrapezoidProfile::set_endpts (
            double start,
            double end )
```

set_endpts defines a start and end position

Parameters

start	the starting position of the path
end	the ending position of the path

5.44.3.5 set_max_v()

```
void TrapezoidProfile::set_max_v (
            double max_v )
```

sets the maximum velocity for the profile (the height of the top of the trapezoid)

Parameters

max_v	the maximum velocity the robot can travel at

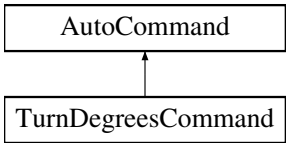
The documentation for this class was generated from the following files:

- include/utils/trapezoid_profile.h
- src/utils/trapezoid_profile.cpp

5.45 TurnDegreesCommand Class Reference

```
#include <drive_commands.h>
```

Inheritance diagram for TurnDegreesCommand:



Public Member Functions

- TurnDegreesCommand (TankDrive &drive_sys, Feedback &feedback, double degrees, double max_speed=1)
- bool run () override
- void on_timeout () override

Public Member Functions inherited from AutoCommand

- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.45.1 Detailed Description

AutoCommand wrapper class for the turn_degrees function in the TankDrive class

5.45.2 Constructor & Destructor Documentation

5.45.2.1 TurnDegreesCommand()

```
TurnDegreesCommand::TurnDegreesCommand (
            TankDrive & drive_sys,
            Feedback & feedback,
            double degrees,
            double max_speed = 1 )
```

Construct a TurnDegreesCommand Command

Parameters

drive_sys	the drive system we are commanding
feedback	the feedback controller we are using to execute the turn
degrees	how many degrees to rotate
max_speed	0 -> 1 percentage of the drive systems speed to drive at

5.45.3 Member Function Documentation

5.45.3.1 on_timeout()

```
void TurnDegreesCommand::on_timeout ( )  [override], [virtual]
```

Cleans up drive system if we time out before finishing

reset the drive system if we timeout

Reimplemented from AutoCommand.

5.45.3.2 run()

```
bool TurnDegreesCommand::run ( )  [override], [virtual]
```

Run turn_degrees Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

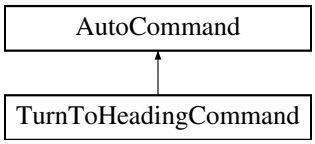
The documentation for this class was generated from the following files:

- include/utils/command_structure/drive_commands.h
- src/utils/command_structure/drive_commands.cpp

5.46 TurnToHeadingCommand Class Reference

`#include <drive_commands.h>`

Inheritance diagram for TurnToHeadingCommand:



Public Member Functions

- TurnToHeadingCommand (TankDrive &drive_sys, Feedback &feedback, double heading_deg, double speed=1)
- bool run () override
- void on_timeout () override

Public Member Functions inherited from AutoCommand

- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.46.1 Detailed Description

AutoCommand wrapper class for the turn_to_heading() function in the TankDrive class

5.46.2 Constructor & Destructor Documentation

5.46.2.1 TurnToHeadingCommand()

```
TurnToHeadingCommand::TurnToHeadingCommand (
            TankDrive & drive_sys,
            Feedback & feedback,
            double heading_deg,
            double max_speed = 1 )
```

Construct a TurnToHeadingCommand Command

Parameters

drive_sys	the drive system we are commanding
feedback	the feedback controller we are using to execute the drive
heading_deg	the heading to turn to in degrees
max_speed	0 -> 1 percentage of the drive systems speed to drive at

5.46.3 Member Function Documentation

5.46.3.1 on_timeout()

```
void TurnToHeadingCommand::on_timeout ( )  [override], [virtual]
```

Cleans up drive system if we time out before finishing

reset the drive system if we don't hit our target

Reimplemented from AutoCommand.

5.46.3.2 run()

```
bool TurnToHeadingCommand::run ( )  [override], [virtual]
```

Run turn_to_heading Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

The documentation for this class was generated from the following files:

- include/utils/command_structure/drive_commands.h
- src/utils/command_structure/drive_commands.cpp

5.47 Vector2D Class Reference

```
#include <vector2d.h>
```

Public Member Functions

- Vector2D (double dir, double mag)
- Vector2D (point_t p)
- double get_dir () const
- double get_mag () const
- double get_x () const
- double get_y () const
- Vector2D normalize ()
- point_t point ()
- Vector2D operator∗ (const double &x)
- Vector2D operator+ (const Vector2D &other)
- Vector2D operator- (const Vector2D &other)

5.47.1 Detailed Description

Vector2D is an x,y pair Used to represent 2D locations on the field. It can also be treated as a direction and magnitude

5.47.2 Constructor & Destructor Documentation

5.47.2.1 Vector2D() [1/2]

```
Vector2D::Vector2D (
            double dir,
            double mag )
```

Construct a vector object.

Parameters

dir	Direction, in radians. 'foward' is 0, clockwise positive when viewed from the top.
mag	Magnitude.

5.47.2.2 Vector2D() [2/2]

```
Vector2D::Vector2D (
            point_t p )
```

Construct a vector object from a cartesian point.

Parameters

p	point_t.x , point_t.y

5.47.3 Member Function Documentation

5.47.3.1 get_dir()

```
double Vector2D::get_dir ( ) const
```

Get the direction of the vector, in radians. '0' is forward, clockwise positive when viewed from the top.

Use r2d() to convert.

Returns

the direction of the vetctor in radians

Get the direction of the vector, in radians. '0' is forward, clockwise positive when viewed from the top.

Use r2d() to convert.

5.47.3.2 get_mag()

```
double Vector2D::get_mag ( ) const
```

Returns

the magnitude of the vector

Get the magnitude of the vector

5.47.3.3 get_x()

```
double Vector2D::get_x ( ) const
```

Returns

the X component of the vector; positive to the right.

Get the X component of the vector; positive to the right.

5.47.3.4 get_y()

```
double Vector2D::get_y ( ) const
```

Returns

the Y component of the vector, positive forward.

Get the Y component of the vector, positive forward.

5.47.3.5 normalize()

```
Vector2D Vector2D::normalize ( )
```

Changes the magnitude of the vector to 1

Returns

the normalized vector

Changes the magnetude of the vector to 1

5.47.3.6 operator∗()

```
Vector2D Vector2D::operator* (
            const double & x )
```

Scales a Vector2D by a scalar with the ∗ operator

Parameters

x	the value to scale the vector by

Returns

the this Vector2D scaled by x

5.47.3.7 operator+()

```
Vector2D Vector2D::operator+ (
            const Vector2D & other )
```

Add the components of two vectors together Vector2D + Vector2D = (this.x + other.x, this.y + other.y)

Parameters

other	the vector to add to this

Returns

the sum of the vectors

5.47.3.8 operator-()

```
Vector2D Vector2D::operator- (
            const Vector2D & other )
```

Subtract the components of two vectors together Vector2D - Vector2D = (this.x - other.x, this.y - other.y)

Parameters

other	the vector to subtract from this

Returns

the difference of the vectors

5.47.3.9 point()

```
point_t Vector2D::point ( )
```

Returns a point from the vector

Returns

the point represented by the vector

Convert a direction and magnitude representation to an x, y representation

Returns

the x, y representation of the vector

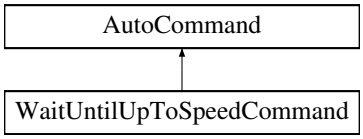
The documentation for this class was generated from the following files:

- include/utils/vector2d.h
- src/utils/vector2d.cpp

5.48 WaitUntilUpToSpeedCommand Class Reference

```
#include <flywheel_commands.h>
```

Inheritance diagram for WaitUntilUpToSpeedCommand:



Public Member Functions

- WaitUntilUpToSpeedCommand (Flywheel &flywheel, int threshold_rpm)
- bool run () override

Public Member Functions inherited from AutoCommand

- virtual void on_timeout ()
- AutoCommand * **withTimeout** (double t_seconds)

Additional Inherited Members

Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

5.48.1 Detailed Description

AutoCommand that listens to the Flywheel and waits until it is at its target speed +/- the specified threshold

5.48.2 Constructor & Destructor Documentation

5.48.2.1 WaitUntilUpToSpeedCommand()

```
WaitUntilUpToSpeedCommand::WaitUntilUpToSpeedCommand (
            Flywheel & flywheel,
            int threshold_rpm )
```

Creat a WaitUntilUpToSpeedCommand

Parameters

flywheel	the flywheel system we are commanding
threshold_rpm	the threshold over and under the flywheel target RPM that we define to be acceptable

5.48.3 Member Function Documentation

5.48.3.1 run()

```
bool WaitUntilUpToSpeedCommand::run ( )  [override], [virtual]
```

Run spin_manual Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

The documentation for this class was generated from the following files:

- include/utils/command_structure/flywheel_commands.h
- src/utils/command_structure/flywheel_commands.cpp

Chapter 6

File Documentation

6.1 robot_specs.h

```
00001 #pragma once
00002 #include "../core/include/utils/pid.h"
00003 #include "../core/include/utils/feedback_base.h"
00004
00011 typedef struct
00012 {
00013   double robot_radius;
00014
00015   double odom_wheel_diam;
00016   double odom_gear_ratio;
00017   double dist_between_wheels;
00018
00019   double drive_correction_cutoff;
00020
00021   Feedback *drive_feedback;
00022   Feedback *turn_feedback;
00023   PID::pid_config_t correction_pid;
00024
00025 } robot_specs_t;
```

6.2 custom_encoder.h

```
00001 #pragma once
00002 #include "vex.h"
00003
00008 class CustomEncoder : public vex::encoder
00009 {
00010   typedef vex::encoder super;
00011
00012   public:
00018   CustomEncoder(vex::triport::port &port, double ticks_per_rev);
00019
00025   void setRotation(double val, vex::rotationUnits units);
00026
00032   void setPosition(double val, vex::rotationUnits units);
00033
00039   double rotation(vex::rotationUnits units);
00040
00046   double position(vex::rotationUnits units);
00047
00053   double velocity(vex::velocityUnits units);
00054
00055
00056   private:
00057   double tick_scalar;
00058 };
```

6.3 flywheel.h

```
00001 #pragma once
00002 /*********************************************************
00003 *
00004 *     File:     Flywheel.h
00005 *     Purpose:  Generalized flywheel class for Core.
00006 *     Author:   Chris Nokes
00007 *
00008 *********************************************************
00009 * EDIT HISTORY
00010 *********************************************************
00011 * 09/23/2022  <CRN> Reorganized, added documentation.
00012 * 09/23/2022  <CRN> Added functions elaborated on in .cpp.
00013 *********************************************************/
00014 #include "../core/include/utils/feedforward.h"
00015 #include "vex.h"
00016 #include "../core/include/robot_specs.h"
00017 #include "../core/include/utils/pid.h"
00018 #include <atomic>
00019
00020 using namespace vex;
00021
00029 class Flywheel{
00030   enum FlywheelControlStyle{
00031     PID_Feedforward,
00032     Feedforward,
00033     Take_Back_Half,
00034     Bang_Bang,
00035   };
00036   public:
00037
00038   // CONSTRUCTORS, GETTERS, AND SETTERS
00046   Flywheel(motor_group &motors, PID::pid_config_t &pid_config, FeedForward::ff_config_t &ff_config,
      const double ratio);
00047
00054   Flywheel(motor_group &motors, FeedForward::ff_config_t &ff_config, const double ratio);
00055
00062   Flywheel(motor_group &motors, double tbh_gain, const double ratio);
00063
00069   Flywheel(motor_group &motors, const double ratio);
00070
00075   double getDesiredRPM();
00076
00081   bool isTaskRunning();
00082
00086   motor_group* getMotors();
00087
00091   double measureRPM();
00092
00096   double getRPM();
00100   PID* getPID();
00101
00105   double getPIDValue();
00106
00110   double getFeedforwardValue();
00111
00115   double getTBHGain();
00116
00121   void setPIDTarget(double value);
00122
00127   void updatePID(double value);
00128
00129   // SPINNERS AND STOPPERS
00130
00137   void spin_raw(double speed, directionType dir=fwd);
00138
00145   void spin_manual(double speed, directionType dir=fwd);
00146
00152   void spinRPM(int rpm);
00153
00157   void stop();
00158
00159
00163   void stopMotors();
00164
00168   void stopNonTasks();
00169
00170   private:
00171
00172   motor_group &motors;                // motors that make up the flywheel
00173   bool taskRunning = false;           // is the task (thread but not) currently running?
00174   PID pid;                            // PID on the flywheel
00175   FeedForward ff;                     // FF constants for the flywheel
00176   double TBH_gain;                    // TBH gain parameter for the flywheel
00177   double ratio;                       // multiplies the velocity by this value
00178   std::atomic<double> RPM;            // Desired RPM of the flywheel.
```

```
00179   task rpmTask;                       // task (thread but not) that handles spinning the wheel at a
      given RPM
00180   FlywheelControlStyle control_style; // how the flywheel should be controlled
00181   double smoothedRPM;
00182   MovingAverage RPM_avger;
00183   };
```

6.4 lift.h

```
00001 #pragma once
00002
00003 #include "vex.h"
00004 #include "../core/include/utils/pid.h"
00005 #include <iostream>
00006 #include <map>
00007 #include <atomic>
00008 #include <vector>
00009
00010 using namespace vex;
00011 using namespace std;
00012
00020 template <typename T>
00021 class Lift
00022 {
00023   public:
00024
00031   struct lift_cfg_t
00032   {
00033     double up_speed, down_speed;
00034     double softstop_up, softstop_down;
00035
00036     PID::pid_config_t lift_pid_cfg;
00037   };
00038
00060   Lift(motor_group &lift_motors, lift_cfg_t &lift_cfg, map<T, double> &setpoint_map, limit
      *homing_switch=NULL)
00061   : lift_motors(lift_motors), cfg(lift_cfg), lift_pid(cfg.lift_pid_cfg), setpoint_map(setpoint_map),
      homing_switch(homing_switch)
00062   {
00063
00064     is_async = true;
00065     setpoint = 0;
00066
00067     // Create a background task that is constantly updating the lift PID, if requested.
00068     // Set once, and forget.
00069     task t([](void* ptr){
00070       Lift &lift = *((Lift*) ptr);
00071
00072       while(true)
00073       {
00074         if(lift.get_async())
00075           lift.hold();
00076
00077         vexDelay(50);
00078       }
00079
00080       return 0;
00081     }, this);
00082
00083   }
00084
00093   void control_continuous(bool up_ctrl, bool down_ctrl)
00094   {
00095     static timer tmr;
00096
00097     double cur_pos = 0;
00098
00099     // Check if there's a hook for a custom sensor. If not, use the motors.
00100     if(get_sensor == NULL)
00101       cur_pos = lift_motors.position(rev);
00102     else
00103       cur_pos = get_sensor();
00104
00105     if(up_ctrl && cur_pos < cfg.softstop_up)
00106     {
00107       lift_motors.spin(directionType::fwd, cfg.up_speed, volt);
00108       setpoint = cur_pos + .3;
00109
00110       // std::cout « "DEBUG OUT: UP " « setpoint « ", " « tmr.time(sec) « ", " « cfg.down_speed «
      "\n";
00111
00112       // Disable the PID while going UP.
00113       is_async = false;
```

```
00114     } else if(down_ctrl && cur_pos > cfg.softstop_down)
00115     {
00116       // Lower the lift slowly, at a rate defined by down_speed
00117       if(setpoint > cfg.softstop_down)
00118         setpoint = setpoint - (tmr.time(sec) * cfg.down_speed);
00119       // std::cout « "DEBUG OUT: DOWN " « setpoint « ", " « tmr.time(sec) « ", " « cfg.down_speed «
    "\n";
00120       is_async = true;
00121     } else
00122     {
00123       // Hold the lift at the last setpoint
00124       is_async = true;
00125     }
00126
00127     tmr.reset();
00128   }
00129
00138   void control_manual(bool up_btn, bool down_btn, int volt_up, int volt_down)
00139   {
00140     static bool down_hold = false;
00141     static bool init = true;
00142
00143     // Allow for setting position while still calling this function
00144     if(init || up_btn || down_btn)
00145     {
00146       init = false;
00147       is_async = false;
00148     }
00149
00150     double rev = lift_motors.position(rotationUnits::rev);
00151
00152     if(rev < cfg.softstop_down && down_btn)
00153       down_hold = true;
00154     else if( !down_btn )
00155       down_hold = false;
00156
00157     if(up_btn && rev < cfg.softstop_up)
00158       lift_motors.spin(directionType::fwd, volt_up, voltageUnits::volt);
00159     else if(down_btn && rev > cfg.softstop_down && !down_hold)
00160       lift_motors.spin(directionType::rev, volt_down, voltageUnits::volt);
00161     else
00162       lift_motors.spin(directionType::fwd, 0, voltageUnits::volt);
00163
00164   }
00165
00177   void control_setpoints(bool up_step, bool down_step, vector<T> pos_list)
00178   {
00179     // Make sure inputs are only processed on the rising edge of the button
00180     static bool up_last = up_step, down_last = down_step;
00181
00182     bool up_rising = up_step && !up_last;
00183     bool down_rising = down_step && !down_last;
00184
00185     up_last = up_step;
00186     down_last = down_step;
00187
00188     static int cur_index = 0;
00189
00190     // Avoid an index overflow. Shouldn't happen unless the user changes pos_list between calls.
00191     if(cur_index >= pos_list.size())
00192       cur_index = pos_list.size() - 1;
00193
00194     // Increment or decrement the index of the list, bringing it up or down.
00195     if(up_rising && cur_index < (pos_list.size() - 1))
00196       cur_index++;
00197     else if(down_rising && cur_index > 0)
00198       cur_index--;
00199
00200     // Set the lift to hold the position in the background with the PID loop
00201     set_position(pos_list[cur_index]);
00202     is_async = true;
00203
00204   }
00205
00214   bool set_position(T pos)
00215   {
00216     this->setpoint = setpoint_map[pos];
00217     is_async = true;
00218
00219     return (lift_pid.get_target() == this->setpoint) && lift_pid.is_on_target();
00220   }
00221
00228   bool set_setpoint(double val)
00229   {
00230     this->setpoint = val;
00231     return (lift_pid.get_target() == this->setpoint) && lift_pid.is_on_target();
00232   }
```

```
00233
00237   double get_setpoint()
00238   {
00239     return this->setpoint;
00240   }
00241
00246   void hold()
00247   {
00248     lift_pid.set_target(setpoint);
00249     // std::cout « "DEBUG OUT: SETPOINT " « setpoint « "\n";
00250
00251     if(get_sensor != NULL)
00252       lift_pid.update(get_sensor());
00253     else
00254       lift_pid.update(lift_motors.position(rev));
00255
00256     // std::cout « "DEBUG OUT: ROTATION " « lift_motors.rotation(rev) « "\n\n";
00257
00258     lift_motors.spin(fwd, lift_pid.get(), volt);
00259   }
00260
00265   void home()
00266   {
00267     static timer tmr;
00268     tmr.reset();
00269
00270     while(tmr.time(sec) < 3)
00271     {
00272       lift_motors.spin(directionType::rev, 6, volt);
00273
00274       if (homing_switch == NULL && lift_motors.current(currentUnits::amp) > 1.5)
00275         break;
00276       else if (homing_switch != NULL && homing_switch->pressing())
00277         break;
00278     }
00279
00280     if(reset_sensor != NULL)
00281       reset_sensor();
00282
00283     lift_motors.resetPosition();
00284     lift_motors.stop();
00285
00286   }
00287
00291   bool get_async()
00292   {
00293     return is_async;
00294   }
00295
00301   void set_async(bool val)
00302   {
00303     this->is_async = val;
00304   }
00305
00315   void set_sensor_function(double (*fn_ptr) (void))
00316   {
00317     this->get_sensor = fn_ptr;
00318   }
00319
00326   void set_sensor_reset(void (*fn_ptr) (void))
00327   {
00328     this->reset_sensor = fn_ptr;
00329   }
00330
00331   private:
00332
00333   motor_group &lift_motors;
00334   lift_cfg_t &cfg;
00335   PID lift_pid;
00336   map<T, double> &setpoint_map;
00337   limit *homing_switch;
00338
00339   atomic<double> setpoint;
00340   atomic<bool> is_async;
00341
00342   double (*get_sensor)(void) = NULL;
00343   void (*reset_sensor)(void) = NULL;
00344
00345
00346 };
```

6.5 mecanum_drive.h

```
00001 #pragma once
```

```
00002
00003 #include "vex.h"
00004 #include "../core/include/utils/pid.h"
00005
00006 #ifndef PI
00007 #define PI 3.141592654
00008 #endif
00009
00014 class MecanumDrive
00015 {
00016
00017   public:
00018
00022   struct mecanumdrive_config_t
00023   {
00024     // PID configurations for autonomous driving
00025     PID::pid_config_t drive_pid_conf;
00026     PID::pid_config_t drive_gyro_pid_conf;
00027     PID::pid_config_t turn_pid_conf;
00028
00029     // Diameter of the mecanum wheels
00030     double drive_wheel_diam;
00031
00032     // Diameter of the perpendicular undriven encoder wheel
00033     double lateral_wheel_diam;
00034
00035     // Width between the center of the left and right wheels
00036     double wheelbase_width;
00037
00038   };
00039
00043   MecanumDrive(vex::motor &left_front, vex::motor &right_front, vex::motor &left_rear, vex::motor
      &right_rear,
00044                vex::rotation *lateral_wheel=NULL, vex::inertial *imu=NULL, mecanumdrive_config_t
      *config=NULL);
00045
00054   void drive_raw(double direction_deg, double magnitude, double rotation);
00055
00066   void drive(double left_y, double left_x, double right_x, int power=2);
00067
00080   bool auto_drive(double inches, double direction, double speed, bool gyro_correction=true);
00081
00092   bool auto_turn(double degrees, double speed, bool ignore_imu=false);
00093
00094   private:
00095
00096   vex::motor &left_front, &right_front, &left_rear, &right_rear;
00097
00098   mecanumdrive_config_t *config;
00099   vex::rotation *lateral_wheel;
00100   vex::inertial *imu;
00101
00102   PID *drive_pid = NULL;
00103   PID *drive_gyro_pid = NULL;
00104   PID *turn_pid = NULL;
00105
00106   bool init = true;
00107
00108 };
```

6.6 odometry_3wheel.h

```
00001 #pragma once
00002 #include "../core/include/subsystems/odometry/odometry_base.h"
00003 #include "../core/include/subsystems/tank_drive.h"
00004 #include "../core/include/subsystems/custom_encoder.h"
00005
00032 class Odometry3Wheel : public OdometryBase
00033 {
00034     public:
00035
00040     typedef struct
00041     {
00042         double wheelbase_dist;
00043         double off_axis_center_dist;
00044         double wheel_diam;
00046     } odometry3wheel_cfg_t;
00047
00057     Odometry3Wheel(CustomEncoder &lside_fwd, CustomEncoder &rside_fwd, CustomEncoder &off_axis,
      odometry3wheel_cfg_t &cfg, bool is_async=true);
00058
00065     pose_t update() override;
00066
```

```
00075     void tune(vex::controller &con, TankDrive &drive);
00076
00077     private:
00078
00091     static pose_t calculate_new_pos(double lside_delta_deg, double rside_delta_deg, double
      offax_delta_deg, pose_t old_pos, odometry3wheel_cfg_t cfg);
00092
00093     CustomEncoder &lside_fwd, &rside_fwd, &off_axis;
00094     odometry3wheel_cfg_t &cfg;
00095
00096
00097 };
```

6.7 odometry_base.h

```
00001 #pragma once
00002
00003 #include "vex.h"
00004 #include "../core/include/utils/geometry.h"
00005 #include "../core/include/robot_specs.h"
00006
00007 #ifndef PI
00008 #define PI 3.141592654
00009 #endif
00010
00011
00012
00025 class OdometryBase
00026 {
00027 public:
00028
00034     OdometryBase(bool is_async);
00035
00040     pose_t get_position(void);
00041
00046     virtual void set_position(const pose_t& newpos=zero_pos);
00047
00052     virtual pose_t update() = 0;
00053
00061     static int background_task(void* ptr);
00062
00068     void end_async();
00069
00076     static double pos_diff(pose_t start_pos, pose_t end_pos);
00077
00084     static double rot_diff(pose_t pos1, pose_t pos2);
00085
00094     static double smallest_angle(double start_deg, double end_deg);
00095
00097     bool end_task = false;
00098
00103     double get_speed();
00104
00109     double get_accel();
00110
00115     double get_angular_speed_deg();
00116
00121     double get_angular_accel_deg();
00122
00126     inline static constexpr pose_t zero_pos = {.x=0.0L, .y=0.0L, .rot=90.0L};
00127
00128 protected:
00132     vex::task *handle;
00133
00137     vex::mutex mut;
00138
00142     pose_t current_pos;
00143
00144     double speed;
00145     double accel;
00146     double ang_speed_deg;
00147     double ang_accel_deg;
00148 };
```

6.8 odometry_tank.h

```
00001 #pragma once
00002
00003 #include "../core/include/subsystems/odometry/odometry_base.h"
```

```
00004 #include "../core/include/subsystems/custom_encoder.h"
00005 #include "../core/include/utils/geometry.h"
00006 #include "../core/include/utils/vector2d.h"
00007 #include "../core/include/robot_specs.h"
00008
00009 static int background_task(void* odom_obj);
00010
00011
00018 class OdometryTank : public OdometryBase
00019 {
00020 public:
00029     OdometryTank(vex::motor_group &left_side, vex::motor_group &right_side, robot_specs_t &config,
      vex::inertial *imu=NULL, bool is_async=true);
00030
00040     OdometryTank(CustomEncoder &left_enc, CustomEncoder &right_enc, robot_specs_t &config,
      vex::inertial *imu=NULL, bool is_async=true);
00041
00046     pose_t update() override;
00047
00052     void set_position(const pose_t &newpos=zero_pos) override;
00053
00054
00055
00056 private:
00060     static pose_t calculate_new_pos(robot_specs_t &config, pose_t &stored_info, double lside_diff,
      double rside_diff, double angle_deg);
00061
00062     vex::motor_group *left_side, *right_side;
00063     CustomEncoder *left_enc, *right_enc;
00064     vex::inertial *imu;
00065     robot_specs_t &config;
00066
00067     double rotation_offset = 0;
00068
00069 };
```

6.9 screen.h

```
00001 #pragma once
00002 #include "vex.h"
00003 #include <vector>
00004
00009
00010 typedef void (*screenFunc)(vex::brain::lcd &screen, int x, int y, int width, int height, bool
      first_run);
00011
00012 void draw_mot_header(vex::brain::lcd &screen, int x, int y, int width);
00013 // name should be no longer than 15 characters
00014 void draw_mot_stats(vex::brain::lcd &screen, int x, int y, int width, const char *name, vex::motor
      &motor, int animation_tick);
00015 void draw_dev_stats(vex::brain::lcd &screen, int x, int y, int width, const char *name, vex::device
      &dev, int animation_tick);
00016
00017 void draw_battery_stats(vex::brain::lcd &screen, int x, int y, double voltage, double percentage);
00018
00019
00020
00021 void draw_lr_arrows(vex::brain::lcd &screen, int bar_width, int width, int height);
00022
00023 int handle_screen_thread(vex::brain::lcd &screen, std::vector<screenFunc> pages, int first_page);
00024 void StartScreen(vex::brain::lcd &screen, std::vector<screenFunc> pages, int first_page = 0);
```

6.10 tank_drive.h

```
00001 #pragma once
00002
00003 #ifndef PI
00004 #define PI 3.141592654
00005 #endif
00006
00007 #include "vex.h"
00008 #include "../core/include/subsystems/odometry/odometry_tank.h"
00009 #include "../core/include/utils/pid.h"
00010 #include "../core/include/utils/feedback_base.h"
00011 #include "../core/include/robot_specs.h"
00012 #include "../core/src/utils/pure_pursuit.cpp"
00013 #include <vector>
00014
00015
```

```
00016 using namespace vex;
00017
00022 class TankDrive
00023 {
00024 public:
00025
00033   TankDrive(motor_group &left_motors, motor_group &right_motors, robot_specs_t &config, OdometryBase
      *odom=NULL);
00034
00038   void stop();
00039
00050   void drive_tank(double left, double right, int power=1, bool isdriver=false);
00051
00062   void drive_arcade(double forward_back, double left_right, int power=1);
00063
00074   bool drive_forward(double inches, directionType dir, Feedback &feedback, double max_speed=1);
00075
00084   bool drive_forward(double inches, directionType dir, double max_speed=1);
00085
00096   bool turn_degrees(double degrees, Feedback &feedback, double max_speed=1);
00097
00107   bool turn_degrees(double degrees, double max_speed=1);
00108
00120   bool drive_to_point(double x, double y, vex::directionType dir, Feedback &feedback, double
      max_speed=1);
00121
00133   bool drive_to_point(double x, double y, vex::directionType dir, double max_speed=1);
00134
00143   bool turn_to_heading(double heading_deg, Feedback &feedback, double max_speed=1);
00151   bool turn_to_heading(double heading_deg, double max_speed=1);
00152
00156   void reset_auto();
00157
00166   static double modify_inputs(double input, int power=2);
00167
00179   bool pure_pursuit(std::vector<PurePursuit::hermite_point> path, directionType dir, double radius,
      double res, Feedback &feedback, double max_speed=1);
00180
00181 private:
00182   motor_group &left_motors;
00183   motor_group &right_motors;
00184
00185   PID correction_pid;
00186   Feedback *drive_default_feedback = NULL;
00187   Feedback *turn_default_feedback = NULL;
00188
00189   OdometryBase *odometry;
00190
00191   robot_specs_t &config;
00192
00193   bool func_initialized = false;
00194   bool is_pure_pursuit = false;
00195 };
```

6.11 auto_chooser.h

```
00001 #pragma once
00002 #include "vex.h"
00003 #include <string>
00004 #include <vector>
00005
00006
00015 class AutoChooser
00016 {
00017   public:
00023   AutoChooser(vex::brain &brain);
00024
00029   void add(std::string name);
00030
00035   std::string get_choice();
00036
00037   protected:
00038
00042   struct entry_t
00043   {
00044     int x;
00045     int y;
00046     int width;
00047     int height;
00048     std::string name;
00049   };
00050
00051   void render(entry_t *selected);
```

```
00052
00053   std::string choice;
00054   std::vector<entry_t> list ;
00055   vex::brain &brain;
00058 };
```

6.12 auto_command.h

```
00001
00007 #pragma once
00008
00009 #include "vex.h"
00010
00011 class AutoCommand {
00012   public:
00013     static constexpr double default_timeout = 10.0;
00019     virtual bool run() { return true; }
00023     virtual void on_timeout(){}
00024     AutoCommand* withTimeout(double t_seconds){
00025       this->timeout_seconds = t_seconds;
00026       return this;
00027     }
00037     double timeout_seconds = default_timeout;
00038
00039 };
```

6.13 command_controller.h

```
00001
00010 #pragma once
00011 #include <vector>
00012 #include <queue>
00013 #include "../core/include/utils/command_structure/auto_command.h"
00014
00015 class CommandController
00016 {
00017 public:
00023   void add(AutoCommand *cmd, double timeout_seconds = 10.0);
00024
00029   void add(std::vector<AutoCommand *> cmds);
00030
00036   void add(std::vector<AutoCommand *> cmds, double timeout_sec);
00043   void add_delay(int ms);
00044
00049   void run();
00055   bool last_command_timed_out();
00056
00057 private:
00058   std::queue<AutoCommand *> command_queue;
00059   bool command_timed_out = false;
00060 };
```

6.14 delay_command.h

```
00001
00008 #pragma once
00009
00010 #include "../core/include/utils/command_structure/auto_command.h"
00011
00012 class DelayCommand: public AutoCommand {
00013   public:
00018     DelayCommand(int ms): ms(ms) {}
00019
00025     bool run() override {
00026       vexDelay(ms);
00027       return true;
00028     }
00029
00030   private:
00031     // amount of milliseconds to wait
00032     int ms;
00033 };
```

6.15 drive_commands.h

```
00001
00019 #pragma once
00020
00021 #include "vex.h"
00022 #include "../core/include/utils/geometry.h"
00023 #include "../core/include/utils/command_structure/auto_command.h"
00024 #include "../core/include/subsystems/tank_drive.h"
00025
00026 using namespace vex;
00027
00028
00029 // ==== DRIVING ====
00030
00036 class DriveForwardCommand: public AutoCommand {
00037   public:
00038     DriveForwardCommand(TankDrive &drive_sys, Feedback &feedback, double inches, directionType dir,
      double max_speed=1);
00039
00045     bool run() override;
00049     void on_timeout() override;
00050
00051   private:
00052     // drive system to run the function on
00053     TankDrive &drive_sys;
00054
00055     // feedback controller to use
00056     Feedback &feedback;
00057
00058     // parameters for drive_forward
00059     double inches;
00060     directionType dir;
00061     double max_speed;
00062 };
00063
00068 class TurnDegreesCommand: public AutoCommand {
00069   public:
00070     TurnDegreesCommand(TankDrive &drive_sys, Feedback &feedback, double degrees, double max_speed =
      1);
00071
00077     bool run() override;
00081     void on_timeout() override;
00082
00083
00084   private:
00085     // drive system to run the function on
00086     TankDrive &drive_sys;
00087
00088     // feedback controller to use
00089     Feedback &feedback;
00090
00091     // parameters for turn_degrees
00092     double degrees;
00093     double max_speed;
00094 };
00095
00100 class DriveToPointCommand: public AutoCommand {
00101   public:
00102     DriveToPointCommand(TankDrive &drive_sys, Feedback &feedback, double x, double y, directionType
      dir, double max_speed = 1);
00103     DriveToPointCommand(TankDrive &drive_sys, Feedback &feedback, point_t point, directionType dir,
      double max_speed=1);
00104
00110     bool run() override;
00111
00112   private:
00113     // drive system to run the function on
00114     TankDrive &drive_sys;
00115
00119     void on_timeout() override;
00120
00121
00122     // feedback controller to use
00123     Feedback &feedback;
00124
00125     // parameters for drive_to_point
00126     double x;
00127     double y;
00128     directionType dir;
00129     double max_speed;
00130
00131 };
00132
00138 class TurnToHeadingCommand: public AutoCommand {
00139   public:
00140     TurnToHeadingCommand(TankDrive &drive_sys, Feedback &feedback, double heading_deg, double speed =
```

```
    1);
00141
00147     bool run() override;
00151     void on_timeout() override;
00152
00153
00154   private:
00155     // drive system to run the function on
00156     TankDrive &drive_sys;
00157
00158     // feedback controller to use
00159     Feedback &feedback;
00160
00161     // parameters for turn_to_heading
00162     double heading_deg;
00163     double max_speed;
00164 };
00165
00170 class DriveStopCommand: public AutoCommand {
00171   public:
00172     DriveStopCommand(TankDrive &drive_sys);
00173
00179     bool run() override;
00180     void on_timeout() override;
00181
00182   private:
00183     // drive system to run the function on
00184     TankDrive &drive_sys;
00185 };
00186
00187
00188 // ==== ODOMETRY ====
00189
00194 class OdomSetPosition: public AutoCommand {
00195   public:
00201     OdomSetPosition(OdometryBase &odom, const pose_t &newpos=OdometryBase::zero_pos);
00202
00208     bool run() override;
00209
00210   private:
00211     // drive system with an odometry config
00212     OdometryBase &odom;
00213     pose_t newpos;
00214 };
```

6.16 flywheel_commands.h

```
00001
00007 #pragma once
00008
00009 #include "../core/include/subsystems/flywheel.h"
00010 #include "../core/include/utils/command_structure/auto_command.h"
00011
00017 class SpinRPMCommand: public AutoCommand {
00018   public:
00024   SpinRPMCommand(Flywheel &flywheel, int rpm);
00025
00031     bool run() override;
00032
00033   private:
00034     // Flywheel instance to run the function on
00035     Flywheel &flywheel;
00036
00037     // parameters for spinRPM
00038     int rpm;
00039 };
00040
00045 class WaitUntilUpToSpeedCommand: public AutoCommand {
00046   public:
00052     WaitUntilUpToSpeedCommand(Flywheel &flywheel, int threshold_rpm);
00053
00059     bool run() override;
00060
00061   private:
00062     // Flywheel instance to run the function on
00063     Flywheel &flywheel;
00064
00065     // if the actual speed is equal to the desired speed +/- this value, we are ready to fire
00066     int threshold_rpm;
00067 };
00068
00074 class FlywheelStopCommand: public AutoCommand {
00075   public:
```

```
00080   FlywheelStopCommand(Flywheel &flywheel);
00081
00087     bool run() override;
00088
00089   private:
00090     // Flywheel instance to run the function on
00091     Flywheel &flywheel;
00092 };
00093
00099 class FlywheelStopMotorsCommand: public AutoCommand {
00100   public:
00105   FlywheelStopMotorsCommand(Flywheel &flywheel);
00106
00112     bool run() override;
00113
00114   private:
00115     // Flywheel instance to run the function on
00116     Flywheel &flywheel;
00117 };
00118
00124 class FlywheelStopNonTasksCommand: public AutoCommand {
00125   FlywheelStopNonTasksCommand(Flywheel &flywheel);
00126
00132     bool run() override;
00133
00134   private:
00135     // Flywheel instance to run the function on
00136     Flywheel &flywheel;
00137 };
```

6.17 feedback_base.h

```
00001 #pragma once
00002
00010 class Feedback
00011 {
00012 public:
00013     enum FeedbackType
00014     {
00015         PIDType,
00016         FeedforwardType,
00017         OtherType,
00018     };
00019
00026     virtual void init(double start_pt, double set_pt) = 0;
00027
00034     virtual double update(double val) = 0;
00035
00039     virtual double get() = 0;
00040
00047     virtual void set_limits(double lower, double upper) = 0;
00048
00052     virtual bool is_on_target() = 0;
00053
00054     virtual Feedback::FeedbackType get_type()
00055     {
00056         return FeedbackType::OtherType;
00057     }
00058 };
```

6.18 feedforward.h

```
00001 #pragma once
00002
00003 #include <math.h>
00004 #include <vector>
00005 #include "../core/include/utils/math_util.h"
00006 #include "../core/include/utils/moving_average.h"
00007 #include "vex.h"
00008
00029 class FeedForward
00030 {
00031     public:
00032
00041     typedef struct
00042     {
00043         double kS;
00044         double kV;
00045         double kA;
```

```
00046         double kG;
00047     } ff_config_t;
00048
00049
00054     FeedForward(ff_config_t &cfg) : cfg(cfg) {}
00055
00066     double calculate(double v, double a, double pid_ref=0.0)
00067     {
00068         double ks_sign = 0;
00069         if(v != 0)
00070             ks_sign = sign(v);
00071         else if(pid_ref != 0)
00072             ks_sign = sign(pid_ref);
00073
00074         return (cfg.kS * ks_sign) + (cfg.kV * v) + (cfg.kA * a) + cfg.kG;
00075     }
00076
00077     private:
00078
00079     ff_config_t &cfg;
00080
00081 };
00082
00083
00091 FeedForward::ff_config_t tune_feedforward(vex::motor_group &motor, double pct, double duration);
```

6.19 generic_auto.h

```
00001 #pragma once
00002
00003 #include <queue>
00004 #include <map>
00005 #include "vex.h"
00006 #include <functional>
00007
00008 typedef std::function<bool(void)> state_ptr;
00009
00014 class GenericAuto
00015 {
00016   public:
00017
00031   bool run(bool blocking);
00032
00037   void add(state_ptr new_state);
00038
00043   void add_async(state_ptr async_state);
00044
00049   void add_delay(int ms);
00050
00051   private:
00052
00053   std::queue<state_ptr> state_list;
00054
00055 };
```

6.20 geometry.h

```
00001 #pragma once
00002 #include <cmath>
00003
00007 struct point_t
00008 {
00009     double x;
00010     double y;
00011
00017     double dist(const point_t other)
00018     {
00019         return std::sqrt(std::pow(this->x - other.x, 2) + pow(this->y - other.y, 2));
00020     }
00021
00027     point_t operator+(const point_t &other)
00028     {
00029         point_t p{
00030             .x = this->x + other.x,
00031             .y = this->y + other.y};
00032         return p;
00033     }
00034
00040     point_t operator-(const point_t &other)
```

```
00041     {
00042         point_t p{
00043             .x = this->x - other.x,
00044             .y = this->y - other.y};
00045         return p;
00046     }
00047 };
00048
00049
00053 typedef struct
00054 {
00055     double x;
00056     double y;
00057     double rot;
00058 } pose_t;
```

6.21 graph_drawer.h

```
00001 #pragma once
00002
00003 #include <string>
00004 #include <stdio.h>
00005 #include <vector>
00006 #include <cmath>
00007 #include "vex.h"
00008 #include "../core/include/utils/geometry.h"
00009 #include "../core/include/utils/vector2d.h"
00010
00011 class GraphDrawer
00012 {
00013 public:
00025   GraphDrawer(vex::brain::lcd &screen, int num_samples, std::string x_label, std::string y_label,
      vex::color col, bool draw_border, double lower_bound, double upper_bound);
00030   void add_sample(point_t sample);
00038   void draw(int x, int y, int width, int height);
00039
00040 private:
00041   vex::brain::lcd &Screen;
00042   std::vector<point_t> samples;
00043   int sample_index = 0;
00044   std::string xlabel;
00045   std::string ylabel;
00046   vex::color col = vex::red;
00047   vex::color bgcol = vex::transparent;
00048   bool border;
00049   double upper;
00050   double lower;
00051 };
```

6.22 logger.h

```
00001 #pragma once
00002
00003 #include <cstdarg>
00004 #include <cstdio>
00005 #include <string>
00006 #include "vex.h"
00007
00009 enum LogLevel
00010 {
00011     DEBUG,
00012     NOTICE,
00013     WARNING,
00014     ERROR,
00015     CRITICAL,
00016     TIME
00017 };
00018
00020 class Logger
00021 {
00022 private:
00023     const std::string filename;
00024     vex::brain::sdcard sd;
00025     void write_level(LogLevel l);
00026
00027 public:
00029     const int MAX_FORMAT_LEN = 512;
00032     explicit Logger(const std::string &filename);
00033
```

```
00035     Logger(const Logger &l) = delete;
00037     Logger &operator=(const Logger &l) = delete;
00038
00039
00042     void Log(const std::string &s);
00043
00047     void Log(LogLevel level, const std::string &s);
00048
00051     void Logln(const std::string &s);
00052
00056     void Logln(LogLevel level, const std::string &s);
00057
00061     void Logf(const char *fmt, ...);
00062
00067     void Logf(LogLevel level, const char *fmt, ...);
00068 };
```

6.23 math_util.h

```
00001 #pragma once
00002 #include "math.h"
00003 #include "vex.h"
00004 #include <vector>
00005
00013 double clamp(double value, double low, double high);
00014
00021 double sign(double x);
00022
00023 double wrap_angle_deg(double input);
00024 double wrap_angle_rad(double input);
00025
00026 /*
00027 Calculates the variance of  a set of numbers (needed for linear regression)
00028 https://en.wikipedia.org/wiki/Variance
00029 @param values   the values for which the variance is taken
00030 @param mean     the average of values
00031 */
00032 double variance(std::vector<double> const &values, double mean);
00033
00034
00035 /*
00036 Calculates the average of a vector of doubles
00037 @param values   the list of values for which the average is taken
00038 */
00039 double mean(std::vector<double> const &values);
00040
00041 /*
00042 Calculates the covariance of a set of points (needed for linear regression)
00043 https://en.wikipedia.org/wiki/Covariance
00044
00045 @param points   the points for which the covariance is taken
00046 @param meanx    the mean value of all x coordinates in points
00047 @param meany    the mean value of all y coordinates in points
00048 */
00049 double covariance(std::vector<std::pair<double, double» const &points, double meanx, double meany);
00050
00051 /*
00052 Calculates the slope and y intercept of the line of best fit for the data
00053 @param points the points for the data
00054 */
00055 std::pair<double, double> calculate_linear_regression(std::vector<std::pair<double, double» const
      &points);
00056
```

6.24 motion_controller.h

```
00001 #pragma once
00002 #include "../core/include/utils/pid.h"
00003 #include "../core/include/utils/feedforward.h"
00004 #include "../core/include/utils/trapezoid_profile.h"
00005 #include "../core/include/utils/feedback_base.h"
00006 #include "../core/include/subsystems/tank_drive.h"
00007 #include "vex.h"
00008
00025 class MotionController : public Feedback
00026 {
00027     public:
00028
00034     typedef struct
```

```
00035     {
00036         double max_v;
00037         double accel;
00038         PID::pid_config_t pid_cfg;
00039         FeedForward::ff_config_t ff_cfg;
00040     } m_profile_cfg_t;
00041
00051     MotionController(m_profile_cfg_t &config);
00052
00057     void init(double start_pt, double end_pt) override;
00058
00065     double update(double sensor_val) override;
00066
00070     double get() override;
00071
00079     void set_limits(double lower, double upper) override;
00080
00085     bool is_on_target() override;
00086
00090     motion_t get_motion();
00091
00110     static FeedForward::ff_config_t tune_feedforward(TankDrive &drive, OdometryTank &odometry, double
      pct=0.6, double duration=2);
00111
00112     private:
00113
00114     m_profile_cfg_t config;
00115
00116     PID pid;
00117     FeedForward ff;
00118     TrapezoidProfile profile;
00119
00120     double lower_limit = 0, upper_limit = 0;
00121     double out = 0;
00122     motion_t cur_motion;
00123
00124     vex::timer tmr;
00125
00126 };
```

6.25 moving_average.h

```
00001 #include <vector>
00002
00015 class MovingAverage {
00016   public:
00017   /*
00018    * Create a moving average calculator with 0 as the default value
00019    *
00020    * @param buffer_size    The size of the buffer. The number of samples that constitute a valid
      reading
00021    */
00022   MovingAverage(int buffer_size);
00023   /*
00024    * Create a moving average calculator with a specified default value
00025    * @param buffer_size    The size of the buffer. The number of samples that constitute a valid
      reading
00026    * @param starting_value The value that the average will be before any data is added
00027    */
00028   MovingAverage(int buffer_size, double starting_value);
00029
00030   /*
00031   * Add a reading to the buffer
00032   * Before:
00033   * [ 1 1 2 2 3 3] => 2
00034   *   ^
00035   * After:
00036   * [ 2 1 2 2 3 3] => 2.16
00037   *     ^
00038   * @param n  the sample that will be added to the moving average.
00039   */
00040   void add_entry(double n);
00041
00046   double get_average();
00047
00052   int get_size();
00053
00054
00055   private:
00056     int buffer_index;               //index of the next value to be overridden
00057     std::vector<double> buffer;     //all current data readings we've taken
00058     double current_avg;             //the current value of the data
00059
00060 };
```

6.26 pid.h

```
00001 #pragma once
00002
00003 #include <cmath>
00004 #include "vex.h"
00005 #include "../core/include/utils/feedback_base.h"
00006
00007 using namespace vex;
00008
00023 class PID : public Feedback
00024 {
00025 public:
00029   enum ERROR_TYPE{
00030     LINEAR,
00031     ANGULAR // assumes degrees
00032   };
00040   struct pid_config_t
00041   {
00042     double p;
00043     double i;
00044     double d;
00045     double deadband;
00046     double on_target_time;
00047     ERROR_TYPE error_method;
00048   };
00049
00050
00051
00056   PID(pid_config_t &config);
00057
00058
00067   void init(double start_pt, double set_pt) override;
00068
00075   double update(double sensor_val) override;
00076
00081   double get() override;
00082
00089   void set_limits(double lower, double upper) override;
00090
00095   bool is_on_target() override;
00096
00100   void reset();
00101
00106   double get_error();
00107
00112   double get_target();
00113
00118   void set_target(double target);
00119
00120   Feedback::FeedbackType get_type() override;
00121
00122   pid_config_t &config;
00123
00124 private:
00125
00126
00127   double last_error = 0;
00128   double accum_error = 0;
00129
00130   double last_time = 0;
00131   double on_target_last_time = 0;
00132
00133   double lower_limit = 0;
00134   double upper_limit = 0;
00135
00136   double target = 0;
00137   double sensor_val = 0;
00138   double out = 0;
00139
00140   bool is_checking_on_target = false;
00141
00142   timer pid_timer;
00143 };
```

6.27 pidff.h

```
00001 #pragma once
00002 #include "../core/include/utils/feedback_base.h"
00003 #include "../core/include/utils/pid.h"
00004 #include "../core/include/utils/feedforward.h"
00005
00006 class PIDFF : public Feedback
```

```
00007 {
00008     public:
00009
00010     PIDFF(PID::pid_config_t &pid_cfg, FeedForward::ff_config_t &ff_cfg);
00011
00018     void init(double start_pt, double set_pt) override;
00019
00024     void set_target(double set_pt);
00025
00033     double update(double val) override;
00034
00043     double update(double val, double vel_setpt, double a_setpt=0);
00044
00048     double get() override;
00049
00056     void set_limits(double lower, double upper) override;
00057
00061     bool is_on_target() override;
00062
00063     PID pid;
00064
00065
00066     private:
00067
00068     FeedForward::ff_config_t &ff_cfg;
00069
00070     FeedForward ff;
00071
00072     double out;
00073     double lower_lim, upper_lim;
00074
00075 };
```

6.28 pure_pursuit.h

```
00001 #pragma once
00002
00003 #include <vector>
00004 #include "../core/include/utils/geometry.h"
00005 #include "../core/include/utils/vector2d.h"
00006 #include "vex.h"
00007
00008 using namespace vex;
00009
00010 namespace PurePursuit {
00015   struct spline
00016   {
00017     double a, b, c, d, x_start, x_end;
00018
00019     double getY(double x) {
00020       return a * pow((x - x_start), 3) + b * pow((x - x_start), 2) + c * (x - x_start) + d;
00021     }
00022   };
00027   struct hermite_point
00028   {
00029     double x;
00030     double y;
00031     double dir;
00032     double mag;
00033
00034     point_t getPoint() {
00035       return {x, y};
00036     }
00037
00038     Vector2D getTangent() {
00039       return Vector2D(dir, mag);
00040     }
00041   };
00042
00047   static std::vector<point_t> line_circle_intersections(point_t center, double r, point_t point1,
      point_t point2);
00051   static point_t get_lookahead(std::vector<point_t> path, point_t robot_loc, double radius);
00052
00056   static std::vector<point_t> inject_path(std::vector<point_t> path, double spacing);
00057
00069   static std::vector<point_t> smooth_path(std::vector<point_t> path, double weight_data, double
      weight_smooth, double tolerance);
00070
00071   static std::vector<point_t> smooth_path_cubic(std::vector<point_t> path, double res);
00072
00081   static std::vector<point_t> smooth_path_hermite(std::vector<hermite_point> path, double step);
00082 }
```

6.29 serializer.h

```
00001 #pragma once
00002 #include <algorithm>
00003 #include <map>
00004 #include <string>
00005 #include <vector>
00006 #include <stdio.h>
00007
00009 const char serialization_separator = '$';
00011 const std::size_t MAX_FILE_SIZE = 4096;
00012
00014 class Serializer
00015 {
00016 private:
00017     bool flush_always;
00018     std::string filename;
00019     std::map<std::string, int> ints;
00020     std::map<std::string, bool> bools;
00021     std::map<std::string, double> doubles;
00022     std::map<std::string, std::string> strings;
00023
00025     bool read_from_disk();
00026
00027 public:
00029     ~Serializer()
00030     {
00031         save_to_disk();
00032         printf("Saving %s\n", filename.c_str());
00033         fflush(stdout);
00034     }
00035
00039     explicit Serializer(const std::string &filename, bool flush_always = true) :
      flush_always(flush_always), filename(filename), ints({}), bools({}), doubles({}), strings({}) {
      read_from_disk(); }
00040
00042     void save_to_disk() const;
00043
00045
00049     void set_int(const std::string &name, int i);
00050
00054     void set_bool(const std::string &name, bool b);
00055
00059     void set_double(const std::string &name, double d);
00060
00064     void set_string(const std::string &name, std::string str);
00065
00068
00073     int int_or(const std::string &name, int otherwise);
00074
00079     bool bool_or(const std::string &name, bool otherwise);
00080
00085     double double_or(const std::string &name, double otherwise);
00086
00091     std::string string_or(const std::string &name, std::string otherwise);
00092 };
```

6.30 trapezoid_profile.h

```
00001 #pragma once
00002
00006 typedef struct
00007 {
00008     double pos;
00009     double vel;
00010     double accel;
00011
00012 } motion_t;
00013
00034 class TrapezoidProfile
00035 {
00036     public:
00037
00044     TrapezoidProfile(double max_v, double accel);
00045
00052     motion_t calculate(double time_s);
00053
00059     void set_endpts(double start, double end);
00060
00065     void set_accel(double accel);
00066
00072     void set_max_v(double max_v);
00073
```

```
00078     double get_movement_time();
00079
00080     private:
00081     double start, end;
00082     double max_v;
00083     double accel;
00084     double time;
00085
00086
00087 };
```

6.31 vector2d.h

```
00001 #pragma once
00002
00003
00004 #include <cmath>
00005 #include "../core/include/utils/geometry.h"
00006
00007 #ifndef PI
00008 #define PI 3.141592654
00009 #endif
00015 class Vector2D
00016 {
00017 public:
00024     Vector2D(double dir, double mag);
00025
00031     Vector2D(point_t p);
00032
00040     double get_dir() const;
00041
00045     double get_mag() const;
00046
00050     double get_x() const;
00051
00055     double get_y() const;
00056
00061     Vector2D normalize();
00062
00067     point_t point();
00068
00074     Vector2D operator*(const double &x);
00081     Vector2D operator+(const Vector2D &other);
00088     Vector2D operator-(const Vector2D &other);
00089
00090 private:
00091
00092     double dir, mag;
00093
00094 };
00095
00101 double deg2rad(double deg);
00102
00109 double rad2deg(double r);
```

Index