

RIT VEXU Software Engineering Notebook

2023-2024

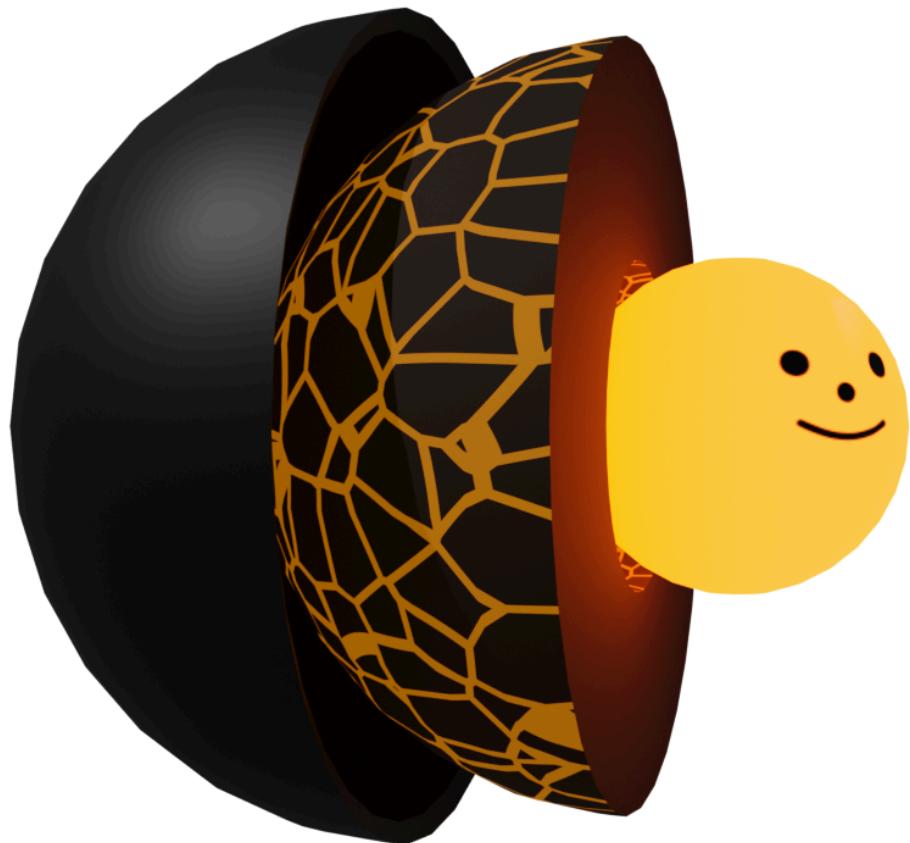


Table of Contents

RIT VEXU Software Engineering Notebook.....	1
Table of Contents.....	2
Overview.....	3
Core API.....	3
Open Source Software.....	3
Project Structure.....	4
Git Subrepo.....	4
Github Project Board.....	4
Github Actions.....	5
Auto-Notebook.....	5
Clang-Tidy.....	6
Wiki.....	6
Core: Fundamentals.....	7
Odometry.....	7
Drivetrain.....	9
Tank Drivetrain.....	9
Control Loops.....	14
Auto Command Structure (ACS).....	16
Serializer.....	18
Screen Subsystem.....	19
Catapult System.....	22
Vision.....	23
Core: Ongoing Projects.....	26
Rust.....	26
N-Pod Odometry.....	28
V5 Debug Board.....	34

Overview

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

The Core codebase is abstracted in a way that allows for simple use during a hectic build season, and creates a solid foundation for future expansion. Subsystems are divided into layers following an object-oriented approach to software development.



Figure 1 - Example of Object-Oriented Programming

Open Source Software

The 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.

Project Structure

During the season, there are three repositories (repos) that are actively developed. Two repositories for the two competition robots, and one for the Core API. Development and code building occurs in the robot repos, and any changes to shared code (drivetrain, math utilities, major subsystems) are merged with the Core repo. This method reduces redundant code and development time.



Figure 2 - Project Structure

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.

Before choosing Subrepo, the team experimented with using Git Submodules to incorporate the Core API into projects. This however made Core development cumbersome and difficult for anyone unfamiliar with Git submodules specifically. Subrepo made inter-project merges more streamlined, and simplified development.

Github Project Board

In order for our software team to collaborate together with these projects, we use the Github Projects kanban-style project board. This allows us to create and assign tasks, link it to a repository and additionally notify the assigned programmer through a slackbot.

Over Under Development	In Progress	Done
<p>● Todo 20 This item hasn't been started</p> <ul style="list-style-type: none">Core #32 New Project Streamlining WikiCore #33 Core Cleanup / DocumentationCore #36 Wiki Entries	<p>● In Progress 4 This is actively being worked on</p> <ul style="list-style-type: none">Core #5 Pure Pursuit functionalityCore #34 Motion ProfilesCore #73 Accelerometer for Lateral Odometry TrackingCore #39 Add 3 Pod Odometry to Core	<p>● Done 13 This has been completed</p> <ul style="list-style-type: none">Core #44 ACS Command TimeoutCore #39 Add 3 Pod Odometry to CoreCore #43 Add Pose2D Class

Figure 3 - Software Project Board

Github Actions

This year, our team enhanced our workflow by integrating GitHub Actions into our software development process. One notable addition was an action to build our C/C++ code in the appropriate Vex environment. This automated process involves a series of steps, including checking out the repository, downloading and unzipping the Vex Robotics SDK and toolchain, and compiling the code using a Makefile. A key feature of this GitHub Action is its ability to send a Slack notification to our team channel whenever a build fails, ensuring prompt awareness and response. Furthermore, it helps maintain code integrity by preventing the merging of pull requests with failing builds. This complements our other GitHub Action for building Doxygen documentation and deploying it to GitHub Pages, allowing for seamless documentation and code management. This systematic approach aligns with our commitment to maintaining a neat, organized, and efficient engineering process.



Figure 4: Continuous Integration directly improves the quality of our code.



Figure 5: Automatically generated documentation.

Auto-Notebook

Alongside the automatic documentation, whenever Core is updated or we manually trigger it, a Github Action copies the reference manual, exports the most up to date version of our written notebook document, stitches them together, and deploys to a webpage. This is publicly available for any person wishing to see our software development process. The most valuable effect, though, is automating most of the formatting work for our notebook, work that used to require a team member to use valuable pre-competition time to sit down, append, format and export the notebook.

Clang-Tidy

In an effort to improve the quality, reduce headaches, and make our code easier to read, write, and understand, we enabled many more warnings than what is supplied with the default Vex project Makefile. These warnings deal with uninitialized variables, missing returns, and other simple code errors that nonetheless have the tendency to introduce tiny, hard to track down bugs. However, sometimes these warnings do not explore deep enough and another tool must be used. We integrated clang-tidy, a c++ linter developed by the clang compiler project, to inspect our code. With a simple switch of a variable in the Makefile, we run clang-tidy during builds which gives many insights into the code that plain compiler warnings do not. Though it does increase compilation times, it tells us about code that is bug prone or poor for performance and tests many other checks developed and validated by the wider C++ community.

Wiki

Whenever a new feature is added to Core, we create a Wiki page on the Core Github repository that provides documentation on what the function does, how to use it, and some examples of how it can be used. This documentation is easily accessible as it can be found online within the Core repository itself. This allows for new members to get acquainted with Core faster and easier than before. This allows us to speed up our training process and allow new members to start developing sooner rather than later. In addition it provides us and anyone using Core great documentation that not only goes into method signatures and descriptions, but also detailed explanations of what different methods, classes or functions do.

Opcontrol

This class provides two ways of driving the robot with a controller: Tank drive and Arcade drive. Drivers can choose what they're most comfortable with.

Tank Drive - The left joystick controls the left-side motors, and the right joystick controls the right-side motors

Arcade Drive - Acts somewhat to how modern racing video games are controlled. The left joystick controls the forward / backward speed, and the right joystick controls turning left / right.

Both functions also have an optional parameter called `power`, and refers to how the joystick is scaled to the motors. The higher the power is, the more control you have over low-speed maneuvers. Because the scaling is non-linear, it may feel weird to those who aren't used to it.

Method Signatures

```
void drive_tank(double left, double right, int power=1);
void drive_arcade(double forward_back, double left_right, int power=1);
```

Usage Examples

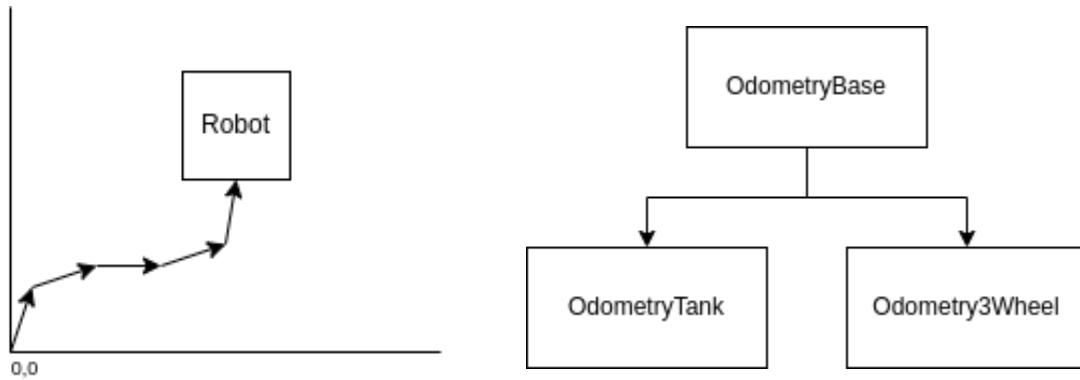
```
drive_system.drive_tank(controller.Axis3.position() / 100.0, controller.Axis2.position() / 100.0);
drive_system.drive_arcade(controller.Axis3.position() / 100.0, controller.Axis1.position() / 100.0);
```

Figure 6: A screenshot of the Core Wiki

Core: Fundamentals

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.

GPS + Odometry

In order to fit an 8-motor drivetrain into the 15" size requirement, the robots could not fit non-powered odometry wheels, leaving only the drive encoders to be used for position tracking. This isn't ideal, since sudden changes in acceleration and wheel slippage can easily cause the tracking to drift a substantial amount. To combat drifting, we looked to the GPS sensor for localization.

The GPS sensor uses a tag-based approach for localization, using a coded strip around the perimeter of the field to estimate position. Between pose estimates, the integrated IMU provides inertial information to estimate changes in position and heading for a constant flow of data, presumably using some sort of onboard Kalman filter. The pose (X, Y, Heading) data is sent back to the Brain over the smart port. In addition, the GPS provides a "quality" value, which is a percentage that increases when the camera can see a large amount of tape, and decreases when the camera is blocked and the IMU detects change in position over a period of time.

To properly characterize the GPS sensor, X/Y/Heading data points were gathered at different positions around the field, facing different headings. The following graphs show the data points on the 12' x 12' field grid. Distance error (in inches) to the actual measurements is shown by color.



Figure 7 - Raw Data Points



Figure 8 - Error Heat Map

There were some other errors with the GPS sensor, including issues localizing the robot when the sensor could not see enough of the coded tape. To take full advantage of the GPS sensor's localization capability, we'd need a way to perform sensor fusion alongside traditional ground-based odometry. To do this, a complementary filter was chosen - a filter that mixes two sets of data based on a proportional scalar *alpha* (α). The equation for a complementary filter is shown below:

$$out = \alpha * s_1 + (1 - \alpha) * s_2$$

where s_1 is *sensor 1*, s_2 is *sensor 2*, and *alpha* scales between the two.

To calculate *alpha*, first the X,Y position of the sensor and heading is taken into account. Since the robot will generally have a more accurate position when it's close to the wall and facing away from it, the following formula will report a score between 0 and 1 for the filter:

$$\alpha = \left(\frac{\vec{c} - \vec{p}}{\|\vec{c} - \vec{p}\|} \cdot \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix} \right) * \frac{\|\vec{c} - \vec{p}\|}{\|\vec{c}\|} * q$$

where \vec{c} is the constant vector of a point in the center of the field ($x=72$ in, $y=72$ in), \vec{p} is the robot's position as a vector (x, y), θ is the robot's heading, and x, y is again the robot's position (0 to 144 inches). The left side is the dot product of the normalized vector pointing from the robot to the center with the direction the robot is facing (gives 1 when the directions are aligned and -1 when opposite smoothly changing in between), and the right side is the sensor's distance to the center as a scalar percentage of the distance from corner to corner (between 0 and 1). Finally, q is the GPS's reported quality. We then remap this from the range [-1, 1] to [0, 1] when we do our mixing.

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.

Tank Drivetrain

Brake Mode

A VEX driver has many things to keep track of during a match. From game element position, match load status, and partner robot condition there is a great deal going on. Defense is another layer on top of the mental load of playing the game. To ease this burden, we implemented a brake mode on our drive train. It is a multi-modal system that can either bring the robot to a stop or hold the robot in a specific location on the field. We use the motion profiles we developed for auto programming to decelerate the robot when requested and use our auto driving functions to hold the robot's position. We implement a smarter form of position holding than just motor braking as we can return to the exact location on the field. Additionally, we combine our deceleration control with position holding such that we do not immediately "lock the brakes" and skid away thus losing the position we attempt to hold and making driving incredibly difficult.

Autonomous Driving

For autonomous driving, the TankDrive class has multiple functions:

- `drive_forward()`:
 - Drive X inches forward/back from the current position
 - Signature: `drive_forward(double inches, directionType dir, double max_speed=1)`
- `turn_degrees()`:
 - Drive X degrees CW/CCW from the current rotation
 - Signature: `turn_degrees(double degrees, double max_speed=1)`
- `drive_to_point()`:
 - Drive to an absolute point on the field, using odometry
 - Signature: `drive_to_point(double x, double y, vex::directionType dir, double max_speed=1);`
- `turn_to_heading()`:
 - Turn to an absolute heading relative to the field, using odometry
 - Signature: `turn_to_heading(double heading_deg, double max_speed=1)`

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.

Drive To Point

The defining feature of a drive to point function is the ability for a robot to calculate a relative direction and distance between its own position and the target position, and navigate to it using tuned control loops. The steps taken for our implementation are listed below.

1 - Gather information

To drive towards a specific point, the robot must know the change in angle between the robot's heading and the target, and the distance to the target. To get this, we first grab the robot's current position and heading and create a positional difference vector between this and the new point.

```
pose_t current_pos = odometry->get_position();
pose_t end_pos = { .x = x, .y = y };

point_t pos_diff_pt =
{
    .x = x - current_pos.x,
    .y = y - current_pos.y
};

Vector2D point_vec(pos_diff_pt);
```

Using this information, grab the distance to the target (using a function in the Odometry subsystem). An issue with the pure distance between points is that it does not represent how far the robot has to travel to be considered "on target" in the control loop. In order to properly reach its target, the robot should report its "aligned distance", and ignore the lateral error, as per Figure 9. This should only hold true when the robot is close to the target, or inside a given radius that is tuned by the user.



Figure 9 - Distance Modification



Figure 10 - Correction Cutoff Circle

```

double dist_left = OdometryBase::pos_diff(current_pos, end_pos);

if (fabs(dist_left) < config.drive_correction_cutoff)
{
    dist_left *= fabs(cos(angle * PI / 180.0));
}

```

The next data needed is the difference in angle between the robot's current heading and the vector between the robot's position and the target. This is calculated by using the arctangent of the difference vector, and subtracting it from the robot's current heading. The angle is then wrapped around 360 degrees.

```

double angle_to_point = atan2(y - current_pos.y, x - current_pos.x)
                           * 180.0 / PI;
double angle = fmod(current_pos.rot - angle_to_point, 360.0);
if (angle > 360)
    angle -= 360;
if (angle < 0)
    angle += 360;

double heading = rad2deg(point_vec.get_dir());
double delta_heading = 0;
if (dir == directionType::fwd)
    delta_heading = OdometryBase::smallest_angle(current_pos.rot, heading);
else
    delta_heading = OdometryBase::smallest_angle(current_pos.rot
                                                - 180, heading);

```

The last piece of information needed is whether the robot should be moving forwards or backwards. Since the distance is calculated as $\sqrt{x^2 + y^2}$, the sign is lost when squaring. Re-implement the sign based on the angle and initial driving direction.

```

int sign = 1;
if (dir == directionType::fwd && angle > 90 && angle < 270)
    sign = -1;
else if (dir == directionType::rev && (angle < 90 || angle > 270))
    sign = -1;

```

2 - Setting Control Loops

In this section, the robot takes the above information and sets its feedback loops. Since the function takes in a FeedbackBase abstract class, any feedback can be used to drive the robot's correction and linear movements. The most common situation is a trapezoidal motion profile for linear distance with PD for heading correction. Once the robot is close enough to the target point, the correction feedback is ignored to avoid issues with last-minute heading changes.

```
correction_pid.update(delta_heading);
feedback.update(sign * -1 * dist_left);

double correction = 0;
if (is_pure_pursuit || fabs(dist_left) > config.drive_correction_cutoff)
{
    correction = correction_pid.get();
}

double drive_pid_rval;
if (dir == directionType::rev) {
    drive_pid_rval = feedback.get() * -1;
} else {
    drive_pid_rval = feedback.get();
}

double lside = drive_pid_rval + correction;
double rside = drive_pid_rval - correction;

lside = clamp(lside, -max_speed, max_speed);
rside = clamp(rside, -max_speed, max_speed);

drive_tank(lside, rside);
```

Finally, when the linear feedback reports its on target, stop, return and report that the movement is over.

```
if (feedback.is_on_target())
{
    if (end_speed == 0) {
        stop();
    }
    func_initialized = false;
    return true;
}
```

Pure Pursuit

Pure Pursuit is a method of autonomous robot driving that allows the robot to autonomously drive through a set of waypoints without stopping and turning.

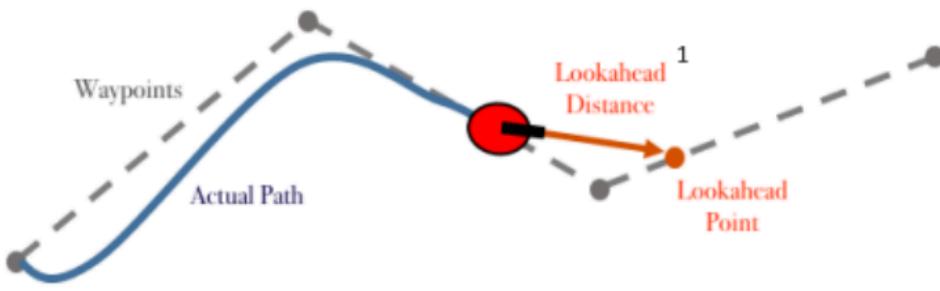


Figure 11 - Pure Pursuit Example

This is accomplished by taking a list of points (x,y), connecting them, and then choosing a "lookahead point" along the lines that the robot will attempt to follow. This will inherently cause the robot to smooth out the point map, and follow without sudden changes in direction.

The lookahead point is chosen by iterating along the path created by connecting the points and finding the furthest point that intersects a circle centered on the robot, with a set radius tuned by the programmer. Increasing this radius smooths the path, while decreasing it ensures the robot more closely follows the path.

The pure pursuit implementation in Core can either use the Autonomous Command System (ACS), or be called directly through the TankDrive class. See the sample code below for examples.

```
// Autonomous Command Controller
CommandController cmd{
    drive_sys.PurePursuitCmd(PurePursuit::Path({
        {.x=19, .y=133},
        {.x=40, .y=136},
        {.x=92, .y=136},
    }, 8), directionType::rev, .5)->withTimeout(4),
};

cmd.run();

// Standalone
while(!drive_sys.pure_pursuit(PurePursuit::Path({
    {.x=19, .y=133},
    {.x=40, .y=136},
    {.x=92, .y=136}}, 8), directionType::fwd, 0.5))
{
    vexDelay(20);
}
```

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. Additionally, there are simpler methods like Bang-Bang or Take Back Half. These adjust the outputs based on the current position relative to the target, where Take Back Half gradually refines the output until it settles at the desired position. These controller types 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.

Bang-Bang

Bang-Bang control is a straightforward control methodology where the output to the system is either fully on or fully off, with no intermediate states. It's used for systems where fine control isn't necessary or possible. In this method, when the process variable is below the setpoint, the controller output is set to maximum; when above, it's set to minimum. This approach is simple and often used for systems with high inertia or where the precise control of the variable isn't critical. However, it can lead to oscillations around the setpoint and isn't suited for systems requiring precise regulation.

Take Back Half (TBH)

The Take Back Half (TBH) method is an iterative approach used to refine control in systems where overshoot is a concern. This method adjusts the output by taking back half the value of the output each time the controlled variable overshoots the target. The adjustment continues until the system settles close to the desired setpoint. TBH is particularly useful in scenarios where a fine balance between responsiveness and stability is needed, as it reduces the oscillation or overshoot often seen in simpler control methods. It's a practical choice for systems where a PID controller might be too complex or unnecessary. TBH controllers only have one tuning parameter which allows for an incredibly easy tuning experience.

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 as well as cleaner, less tightly coupled code.

Motion Profile

As we learn from each event, our team has evolved our approach to robot control systems, transitioning from a simple PID controller to a more sophisticated Motion Profile controller. The PID system, while fundamental, had its drawbacks, such as limited speed specification, poor response to wheel slipping, and slower reaction times. These limitations highlighted the need for a more advanced control mechanism.

Our Motion Profile controller represents a significant upgrade. It integrates precise control over position, acceleration, and velocity, allowing for optimized performance of our robot's subsystems. Unlike the PID controller, which reacts only to discrepancies between actual and desired states, our Motion Profile controller proactively manages the robot's movements. It anticipates the required actions, thereby reducing response lags. Moreover, it avoids the rigidness of a pure feedforward controller by adapting dynamically to changing conditions in competition scenarios.



Figure 12: Trapezoidal motion profile

A key feature of the Motion Profile controller is its ability to handle varying accelerations. This functionality enables our robot to accelerate efficiently without wheel slipping, always maintaining optimal acceleration. This year, we've further refined our Motion Profile to accommodate non-zero starting and ending velocities. This enhancement allows for the seamless chaining of complex movements, ensuring smoother transitions and more fluid motion during competition tasks.

Auto Command Structure (ACS)

Principle

A recent addition to our core API 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;
}

```

Figure 13: ACS code from the 2023 competition season

Updates

This season, we found ourselves annoyed with having to repeat basic things such as `path.add(...)` and having to write `new ThingCommand(...)` over and over again. Our first solution to this was “shortcuts”. These were member functions of subsystems that would allocate, initialize and return an auto command for that subsystem. So, instead of `path.add(new DriveForwardCommand(drive_sys, drive_fast_mprofile, 20, fwd))` we could simply write `path.add(drive_sys.DriveForwardCommand(20, fwd))`. This reduced a great deal of typing but still left us with some issues.

The most hazardous, rather than the simply annoying downside of last year's system, was the memory unsafety of this system. Since our auto commands must use virtual functions, they must be on the other end of a pointer. So, they must be allocated using `new` or they must be initialized statically before we write the path which is a terrible user experience (Though, if constrained by an embedded system where allocating on the heap was deemed dangerous, the system could work with this). This became a real issue when we began to write more complicated constructs such as branching, asynchronous, and repeated commands as it became dangerously unclear who was responsible for deallocating these objects. As a solution for this, we developed an RAI wrapper for the Auto Command Interface. Inspired by C++'s `std::unique_ptr`, this wrapper provides a memory safe, value based way of using auto commands while still maintaining their adaptability. We used C++'s ideas of move semantics and ‘Resource Allocation Is Initialization’ to practically solve memory management so programmers (and even non programmers) can focus on writing paths.

```

CommandController cmd{
    odom.SetPositionCmd({.x = 16.0, .y = 16.0, .rot = 225}),
    // 1 - Turn and shoot preload
    {
        cata_sys.Fire(),
        drive_sys.DriveForwardCmd(dist, REV),
        DelayCommand(300),
        cata_sys.StopFiring(),
        cata_sys.IntakeFully(),
    },
    // 2 - Turn to matchload zone & begin matchloading
    drive_sys.DriveForwardCmd(dist + 2, FWD, 0.5)
        .with_timeout(1.5),

    // Matchloading phase
    Repeat{
        odom.SetPositionCmd({.x = 16.0, .y = 16.0, .rot = 225}),
        intakeToCata.with_timeout(1.75),
        cata_sys.Fire(),
        drive_sys.DriveForwardCmd(10, REV, 0.5),
        cata_sys.StopFiring(),
        cata_sys.IntakeFully(),
        drive_sys.TurnToHeadingCmd(load_angle, 0.5),
        drive_sys.DriveForwardCmd(12, FWD, 0.2).with_timeout(1.7),
    }.until(TimeSinceStartExceeds(30))
};

}

```

Figure 13: ACS code going into the 2024 competition season

Now that we were free to use auto commands without fear for leaking memory or messing with currently running commands, we began to create more powerful constructs such as branching on runtime information, timeouts so the robot can decide what to do based on how much time is left in the auto or skills period, fearless concurrency (driving and reloading at the same time), and a much much nicer user interface. This declarative, safe, and straightforward method of writing auto paths lets us spend less time writing and debugging custom code and more time exploring and optimizing auto paths.

Serializer

One pain point we found last year was configuring auto paths, color targets, path timeouts, and other parameters that changed often but for the most part should be persistent. Commonly, we found ourselves redeploying code at the last minute before a match. To solve this, we wrote a class that takes control of a file on the SD card to which users can read and write values at runtime using a simple key-value interface. This keeps us from having to change a value, redeploy, repeat which cost us valuable time in the past.

Screen Subsystem

Principle

One of the most powerful elements of the V5 Brain is the fairly substantial touch screen. However, its simple drawing API limits its utility as one person's part of the code will draw over another since there is no larger abstraction controlling who draws when. We have many different subsystems on our robot to observe and debug and many parameters that can be tuned at run time and the screen provides a way to do this. We provide an API that provides a 'page' interface that can be inserted into a slideshow-like interface. Each 'page' provides two functions, an update and a draw. The update runs more frequently allowing touch input and data collection at a reasonably fast rate while the draw function runs less frequently to not cause too much overhead on the system. At startup, users provide the screen subsystem a list of pages and the screen subsystem handles orchestration and input in a background thread while other robot code runs unaffected.

```
pages = {
    new AutoChooser({"Auto 1", "Auto 2", "Auto 3", "Auto 4"}),
    new screen::StatsPage(motor_names),
    new screen::OdometryPage(odom, 12, 12, true),
    cata_sys.Page(),
};

screen::start_screen(Brain.Screen, pages);
```

Figure 14: Configuration for the screen subsystem

Pages

Odometry Page

The odometry page has proved incredibly useful in writing and debugging auto and auto skills paths. It shows a picture of the robot on the field as well as a print out of the actual x,y coordinates and heading of the robot. Since we write our autos with respect to the coordinate system of the field, having a map to look at makes development much simpler.

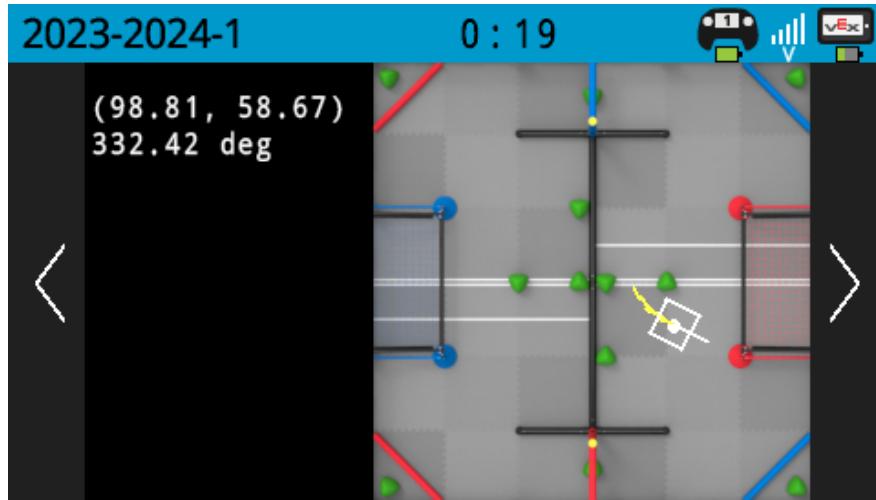


Figure 15: A field display for the Over Under season

PID Tuner

PID controllers are integral to many subsystems on our robots. Our drive code uses them for turning and forward motion, our catapult uses them for reloading, and subsystems across seasons require them for precise control. Tuning them, however, can be incredibly tedious. Changing one value, redeploying, and repeating over and over again is time consuming and unnecessary. Since we have a wonderful touchscreen, we simply added a series of sliders for PID parameters and we can now easily adjust a PID tuning in seconds rather than minutes saving a great deal of time on an already time consuming part of robot development.

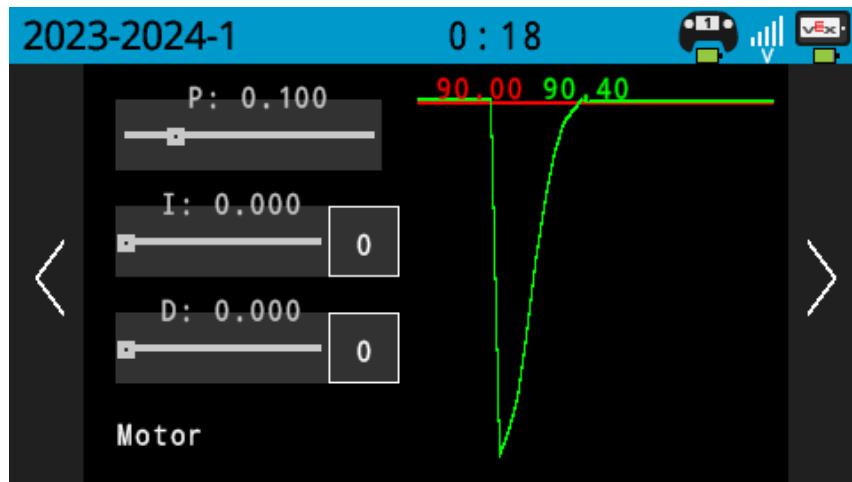


Figure 16: Tuning a motor for reaching an angle

Motor Stats

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 page was written 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.



Figure 17: Motor Stats screen from our 2023 robot

Cata System Page



Figure 18: The catapult status page. Includes a graph of Catapult PID values.

Catapult System

Motivation

Vex's Over Under game requires the effective utilization of the fascinatingly shaped triball. After much deliberation, our team decided on a catapult to launch the game element across the field and a reversible intake for picking up and scoring the triball. This system gives us a great deal of flexibility and power for strategy but does increase the system's complexity. This complexity mostly stems from the orchestration of intaking with catapult reloading such that we do not jam our catapult and never intake multiple game pieces leading to a disqualification.

Initial Design

We implemented a state machine that receives inputs from the controller, a distance sensor in the intake, a distance sensor in the catapult, and a potentiometer for watching the catapult's position. The system runs the appropriate motors to either intake to hold the triball, reset the catapult, intake into the catapult, or shoot depending on its state. Because we have so many sensors, we can determine when intaking would lead to disqualification and simply not honor the intaking command.

These messages are a simple enum that one passes to `CataSys::send_command()`. This was originally intended to make writing multi-threaded code less error-prone as there was one thread-safe and simple way to interact with the subsystem, rather than many disparate methods some of which are meant for internal usage of the class on the running thread and some accessors and setters meant to be used from the user thread. Although it started for implementation ease, it naturally brought about a very simple interface for auto. Instead of sending a command on a button press, we simply send a command at a certain point in our auto path and the system reacts accordingly.

Successor

Though the idea of the state machine modeled the intake and catapult system well, our haphazard implementation (very large and complicated switch statement on a worker thread) made changes exceedingly difficult. As we began competing, we identified changes we wished we could make to make driver and autonomous control easier and handle unforeseen hardware faults. However, our system was hard to read, modify, and all but impossible to prove correct.

Inspired by TinyFSM and other off the shelf C++ libraries for this problem, we created a generic state machine class that handles state transitions, background thread execution, and observability. While this tradeoff led to more code overall, its explicitness and separation of concerns allowed members to make changes in the behavior of the system without fear of deadlocking the threads or unknowingly modifying other states. The generic StateMachine class will remain in our Core library and can be reused year to year to achieve these benefits for any other stateful subsystem.

```

struct Reloading : public CataOnlySys::State {
    void entry(CataOnlySys &sys) override {
        sys.pid.update(sys.pot.angle(vex::deg));
        sys.pid.set_target(cata_target_charge);
    }

    CataOnlySys::MaybeMessage work(CataOnlySys &sys) override {
        // work on motor
        double cata_deg = sys.pot.angle(vex::deg);
        if (cata_deg == 0.0) {
            // adc hasn't warmed up yet, we're getting zero results
            return {};
        }
        sys.pid.update(cata_deg);
        sys.mot.spin(vex::fwd, sys.pid.get(), vex::volt);

        // are we there yettt
        if (sys.pid.is_on_target()) {
            return CataOnlyMessage::DoneReloading;
        }
        // otherwise keep chugging
        return {};
    }

    CataOnlyState id() const override { return CataOnlyState::Reloading; }
    State *respond(CataOnlySys &sys, CataOnlyMessage m) override;
};

```

Explicit separation of states allows simpler, more readable code

Due to the message passing interface to the catapult and intake system, this change was able to be made with very few modifications to the external interface of the system meaning driver code and autonomous paths did not have to be rewritten to use the advantages of the new system - a saving grace as we made this modification in the middle of competition season.

Vision

With the unpredictable way triballs roll across the field, our robots need a way to repeatedly track the game objects during the autonomous period. And so, a vision sensor is placed inside the intake subsystem on the front of the robot.

The Vex Vision sensor is notorious among teams for being unreliable, being highly dependent on field lighting conditions and often sensing random objects, sending the robot off course. Our team explored different methods of filtering and lighting to combat these issues, and are now successfully tracking triballs in our autonomous programs.

Filtering Vision Objects

The first issue to address was filtering - making sure the robot tracks the correct objects. Currently, we run a filtering algorithm that removes all vision objects that don't follow a strict criteria:

- Minimum area (object width * height)
 - Maximum area
 - Minimum aspect ratio (object width / height)
 - Maximum aspect ratio
 - Min / Max X value
 - Min / Max Y value

Finally, the filtered objects array is sorted by area, so that the largest objects are easily accessible at the start of the array. Here's an example of how it's used:

```
vision_filter_s filter{
    .min_area = 2000,
    .max_area = 100000,
    .aspect_low = 0.5,
    .aspect_high = 2,
    .min_x = 0,
    .max_x = 320,
    .min_y = 0,
    .max_y = 240,
};

vector<vision::object> obj list = vision run filter(filter);
```

Standardizing Lighting

In past competitions, we've found differing lighting conditions can spell an unfortunate end for autonomous programs using vision. Spotlights, windows and even the color temperature of the overhead lights caused slight differences which would cause the color profile to be off. We were able to completely eliminate this by adding a custom light to the robot - a board that uses two high-powered LEDs switched with a MOSFET over the three wire ports. Here's the schematic:

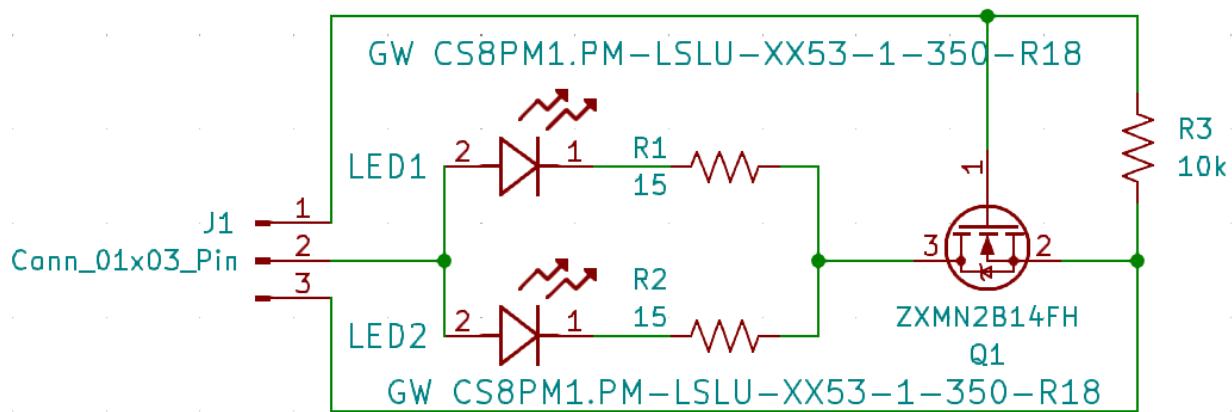


Figure 19 - LED Board schematic

The low-side FET switches power via the signal pin, allowing the programmer to use PWM to dim the lights as needed.

Here's the final PCB built for competition:



Figure 20 - LED Board PCB design

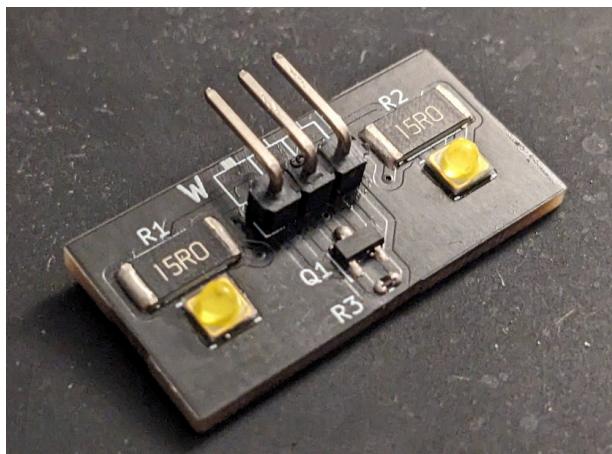


Figure 21 - LED Board, Front

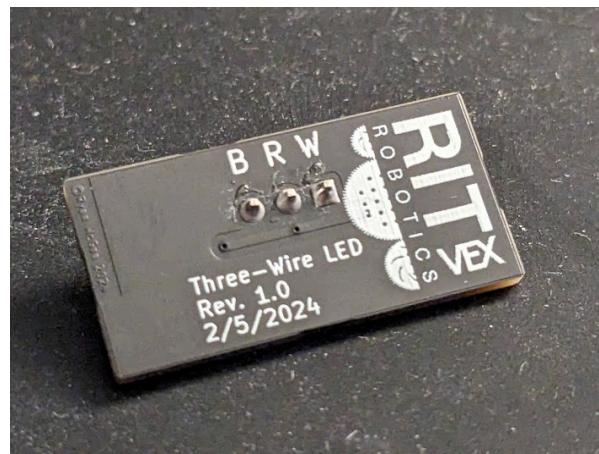


Figure 22 - LED Board, Back

The lighting system was tested and successfully used at the West Virginia tournament, where the robot was able to score five triballs in one autonomous program using this vision system.



Figure 23 - Vision Tracking in Auto



Figure 24 - LED board at full power (~2 Watts)

Core: Ongoing Projects

Rust

Over the course of the year, we have experimented with rewriting our Core API in Rust, a multi-paradigm programming language focused on performance and safety. Rust offers several potential advantages over C++:

Motivation

Memory Safety

One of the primary benefits of Rust is its emphasis on memory safety without sacrificing performance. Rust's ownership model ensures that memory is managed correctly at compile time, reducing the risk of memory leaks and buffer overflows which are common issues in C++. This is especially crucial in robotics, where memory management errors can lead to system crashes or unpredictable behavior in real-time operations.

Concurrency

Rust's approach to concurrency is another major advantage. Concurrency errors, like race conditions, are hard to debug and can be catastrophic in robotics, leading to inconsistent states and erratic behavior. Rust's type system and ownership model prevent data races at compile time, making concurrent programming more reliable and easier to reason about.

Performance

In terms of performance, Rust is comparable to C++, which is essential in robotics where processing speed and response time are critical. Rust's zero-cost abstractions mean that high-level constructs do not add overhead at runtime. This allows developers to write high-level code without compromising on performance, an important consideration in robotics where every millisecond can count.

Improved Code Maintenance and Readability

Rust also offers improved code maintainability and readability. Its modern syntax and language features make it easier to write clear and concise code. This reduces the cognitive load on developers, making it easier to develop and maintain complex robotic systems. The compiler's strictness also ensures that many potential bugs are caught early in the development cycle, reducing the time spent on debugging.

Growing Ecosystem and Community

The Rust ecosystem is rapidly growing, with a strong focus on safety and performance. There are increasing numbers of libraries and tools being developed for Rust, including those specifically for robotics. The Rust community is known for its dedication to improving code quality and security, which aligns well with the needs of robotics development.

Overall

While the transition from C++ to Rust in a robotics context requires investment in terms of learning and codebase modification, the benefits in memory safety, concurrency handling, performance, and maintainability make it a compelling choice. The modern features of Rust, combined with its growing ecosystem and community support, position it well for developing robust, efficient, and safe robotic systems.

Progress

Though progress slowed as competition season began, our rust build system is moving out of the proof of concept stage and into something useful. We setup a cargo (rust's build system manager) target and can cargo build a vex project into an architecture-correct .elf file linked according to vexcode's standard library version and linker configurations. We then created a simple python script to convert the .elf file into the stripped binary file that the vex brain expects and call the vexcom tool provided by the vscode extension to send binaries to the brain.

Findings

Though we did not have much time before our small software team's resources were needed elsewhere, our experiments with Rust programming for VEX found many interesting things.

A surprise we came across is that for proper and safe rust environments one must provide a panic handler. This will be called whenever an error occurs or the programmer signals that the specified behavior is invalid. Though rust does many things to insure 'if it compiles

correctly it runs correctly' there is still behavior that should be signaled to be an error at run time. With the custom panic handler we are able to provide detailed error messages including line numbers and function names - a feature that is sorely missed when programming with the C/C++ API.

Though Rust does come with many benefits, we did find a blocker that is limiting more widespread adoption on the team. The C/C++ API dynamically links the C and C++ standard library after deploying such that a much smaller binary must be transferred to the brain, a life saver when wirelessly uploading. Even with aggressive minimal size optimizations, the requirement to statically link rust core library functions means even simple rust binaries would match the size of our largest C/C++ projects. The PROS ecosystem ran into a similar problem and did work with hot/cold linking in order to not deploy non-changing code each time and we are looking to explore a similar solution. However, most of our research is into undocumented areas of the VEX ecosystem and this feature is still in the early phase of development.

The work on the rust port was split between two members: one of whom ported the API of core and modified it to fit into the rust programming style and safety model and one who set up the compiler toolchain and low level system for interfacing with the vex C/C++ library. Though this was originally an organizational decision, we realized that much of core could be completely abstracted away from dealing with VEX specific components and could operate on hardware that fulfills specific interface requirements. For example, as long as we can send a voltage to a motor and read a position our drivetrain and flywheel subsystems would work no matter the actual hardware. Thanks to rust's powerful generic programming features, this flexibility can be used without sacrificing helpful compiler errors (a common C++ issue) and without sacrificing performance using runtime polymorphism.

N-Pod Odometry

Motivation

Although we have been working on the GPS odometry system, wheel odometry is still vital. It provides great small-scale, quickly updating positions as well as having near-perfect, continuous, local velocity which a GPS system can not achieve. We use odometry in two ways; either tank or differential odometry where there is one wheel on either side of the drivetrain alongside the drive wheels and 3-wheel odometry where we have 3 wheels at ninety degrees to each other. Tank odometry is limited as it can not track horizontal movement and we simply hope that we never move sideways, though it is easiest to implement in the robot so it is our most commonly used system. 3-wheel odometry solves the side-to-side problem but is much harder to implement in hardware owing to the extra wheel where other subsystems would need space.

In a plea for mercy from the hardware team, we agreed that we would take tracking wheels wherever and we could make do. Though we once again got stuck with a tank system, if our dream of more tracking wheels ever comes true we would need code to handle such a system. Also, since tank and 3 wheel odometry are special cases of an n-pod system, we could reduce code duplication.



Figure 25: 2 pod, 3 pod, and arbitrary pods such a system could handle.

Syntax

After much brainstorming and many mad scientist whiteboard drawings, we believe that we have the fundamentals of a system figured out. Unfortunately, other responsibilities to the team came up so we do not yet have a functioning implementation of the system.

Imagine a robot with n number of tracking omni wheels. We could read encoder values E_1, E_2, \dots, E_n from the system in radians from the initial position. As well, each encoder has a configuration $(x_1, y_1, \theta_1, r_1), \dots, (x_n, y_n, \theta_n, r_n)$ describing its position (x, y) relative to the center of rotation, an angle describing its orientation relative to the robot frame (θ), and a radius of the wheel (r).



Figure 26: The configuration of a tracking wheel on the robot. (e_x, e_y) are the basis vectors of our coordinate system - the X and Y axes of the robot coordinate frame. d_i is the direction vector of the tracking wheel.

Now, if we pretend that these wheels are powered and we wish to translate and rotate the robot according to some controller input (x, y, θ) we can develop a formula for how much each wheel needs to rotate to move the robot in that direction with that rotation. Luckily, since the tracking wheels are omni wheels that roll freely in the axis against their “forward” direction, we do not need to worry about dragging a wheel so long as it is spinning the correct amount in its “forward” direction. For a desired (x, y, θ) (in the robots reference frame), for the i -th encoder, we say $E_i = xF_{xi}$. That is, for a movement in the x-axis the rotations of the i -th encoder, are the desired x movement times some scalar factor (F) for how far this specific wheel would rotate. Similarly, for a y only and θ only movement, $E_i = yF_{yi}$ and $E_i = \theta F_{\theta i}$ respectively.

Deriving Factors

F_x

F_x depends on the direction vector \vec{d} of the omni-wheel. If the omni-wheel is facing along the x-axis, F_x will be higher whereas if the omni-wheel is directly perpendicular to the x-axis, it will not spin when you move only in the x-direction. Since \vec{e}_x and \vec{d} are unit vectors, how closely they are related is given by $\vec{e}_x \cdot \vec{d} = \cos(\text{angle between } x \text{ axis and wheel})$

F_x also depends on the radius of the wheel r . One full rotation of the wheel moves a distance of $C = 2\pi r$. If we drive in the direction of the wheel i inches, the wheel will complete $\frac{i}{2\pi r}$ revolutions. If we measure the rotations in radians, the wheel will travel $\frac{i}{r}$ radians. That is, if the encoder wheel travels E radians, we will have traveled Er inches in that direction.

So, the distance traveled in the x direction of a wheel pointing in the direction \vec{d} , rotating E radians is $x = Er(\vec{e}_x \cdot \vec{d})$. This gives since F_x as how many inches per radian turned,

$$F_x = \frac{x}{E} = r(\vec{e}_x \cdot \vec{d})$$

F_y

F_y is derived almost identically as F_x just instead of testing against \vec{e}_x we test against \vec{e}_y . So,

$$F_y = \frac{y}{E} = r(\vec{e}_y \cdot \vec{d})$$

F_θ

F_θ is a little more complicated since it is determined by the position of the wheel \vec{v} as well as the orientation of the wheel \vec{v}

Imagine that the robot turns an angle of θ_r measured in radians. A wheel that is perfectly perpendicular to the rotation will travel an arc with distance $S = ||\vec{v}|| \theta_r$ by the arc length formula where the 'radius' of the arc is defined by the length of the vector \vec{v} .



Figure 27: A conceptual perfectly perpendicular wheel

So, if we have a wheel that is always tangent to the rotation, it will travel

$$E_t r = S = \vec{v} \cdot \vec{t}$$

Since our wheel isn't guaranteed to be perfectly tangent to the arc, we have to use our dot product trick to get the component of its motion that is tangent to the turning circle. That is, instead of comparing to \vec{e}_x or \vec{e}_y we compare to the normalized vector \vec{t} tangent to the turning circle.



$$E_t r = Er(\vec{t} \cdot \vec{d})$$

So

$$Er(\vec{t} \cdot \vec{d}) = S = ||\vec{v}||\theta_t$$

Since \vec{t} is just a unit vector 90 degrees counterclockwise of \vec{v} , We can find it by multiplying \vec{v} by the rotation matrix for 90 degrees and normalizing giving

$$\vec{t} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} norm(\vec{v}) = \begin{bmatrix} -norm(\vec{v}).y \\ norm(\vec{v}).x \end{bmatrix}$$

So

$$F_\theta = \frac{\theta_r}{E} = \frac{r(\vec{t} \cdot \vec{d})}{||\vec{v}||} = \frac{r(\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} norm(\vec{v}) \cdot \vec{d})}{||\vec{v}||} = \frac{r(\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \vec{v} \cdot \vec{d})}{||\vec{v}||^2}$$

Why factors

These factors make solving this problem much simpler. For the forward case, for some wheel i , its rotation is the sum of all the motions applied to it. So $E_i = F_{xi}x + F_{yi}y + F_{\theta i}\theta$. So, for all the wheels, we plug in the commanded (x, y, θ) to each wheel's factors to get its necessary rotation. Since the factors depend only on the wheel's pose in the frame, these can be calculated once at the start of the program and are constant (unless the frame breaks apart, in which case the robot has other problems).

Now Do It Backwards

We have now solved the forward system for when we have a delta of our pose and want our wheel deltas. Now we must take our wheel deltas and solve for our pose delta. We have our formulas for each wheel's encoder motion and can consider this as a system of linear equations. At runtime, we have our wheel encoder deltas we can plug in and then we can solve the system of linear equations. This requires that we have enough data to satisfy the equations. That is, we need at least 3 separate wheels with at least some angle between them, or else the system will be not fully constrained. In the case of tank odometry, we only have two wheels but as outside observers we know we can not measure change in one dimension. So, we know one variable is zero and then have two remaining free variables and two equations to satisfy the system. For robots with greater than three encoders, we have an over-constrained system of equations but this is not an issue. Since all the encoders are modeled on a physical system, they should agree on what the solution is. Using the technique of least squares regression, we can find our (x, y, θ) to solve the over-constrained system that minimizes the error between equations. This

also gives us a way to detect errors in our drive train. If a wheel gets jammed, its encoder reading will disagree with the rest of the system, and the error value will measurably increase. If we monitor this error value we can diagnose mechanical or electrical issues from the code.

$$T = \begin{bmatrix} F_{x1} & F_{y1} & F_{\theta 1} \\ \vdots & \vdots & \vdots \\ F_{xn} & F_{yn} & F_{\theta n} \end{bmatrix}$$

$$\vec{X} = \begin{bmatrix} \frac{dx_{robot}}{dt} \\ \frac{dy_{robot}}{dt} \\ \frac{d\theta_{robot}}{dt} \end{bmatrix}$$

$$\vec{E} = \begin{bmatrix} E_1 \\ \vdots \\ E_n \end{bmatrix}$$

$$\begin{bmatrix} \text{Length} & \text{Length} & \text{Angle} \\ \text{Angle} & \text{Angle} & \text{Angle} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

transfer matrix
from robot velocity
to encoder velocities
(f for factor)

$$\begin{bmatrix} \frac{\text{Distance}}{\text{Time}} \\ \frac{\text{Distance}}{\text{Time}} \\ \frac{\text{Angle}}{\text{Time}} \end{bmatrix}$$

pose velocity

$$\begin{bmatrix} \frac{\text{Angle}}{\text{Time}} \\ \vdots \\ \frac{\text{Angle}}{\text{Time}} \end{bmatrix}$$

encoder wheel
velocities

$$T\vec{X} = \vec{E}$$

$$\vec{X} = T^{-1}\vec{E}$$

or in the case where the matrix is not invertible, find the best solution

The linear algebra behind the solution

V5 Debug Board

The large number of features added to core, while extremely useful, are also very difficult to debug. Without a proper real-time c++ debugger and one stream serial data for print statements, data parsing can get very messy. The improvements to the Screen subsystem have helped, but a remote solution is needed to avoid chasing after the robot to get visual data.



Figure 28 - Debug Board (Back)



Figure 29 - Debug Board (Front)

The V5 Debug Board is a custom PCB designed by our team specifically to interface with the V5 Smart Ports, host a ROS2 node and a WiFi access point for programmers to connect to with laptops. This board is designed to ingest any kind of data and use it for graphs, real-time tuning and even displaying a 3D model of the robot on a virtual field using odometry data. This data can be viewed using either ROS' RViz or Foxglove visualization software.



Figure 30 - Debug Board PCB Layout

Hardware design is nearing completion, with 3 revisions built and tested. Revision 3.0 is powered by an ESP32-C3-WROOM2 microcontroller, and uses an RS-485 transceiver to communicate with the Brain over a smart-port. The new addition of a Micro-SD card allows users to upload their own 3D model of the robot, and provides data logging capabilities.

As of now, the hardware design is nearly complete. Software has achieved WiFi AP broadcasting, TCP communications and work is starting on the Micro-ROS implementation. The design is fully open source under the GPL-3 license, and is hosted at github.com/superrm11/VexDebugBoard and github.com/superrm11/VexDebugBoard_PCB.

RIT VEXU Core API

Generated by Doxygen 1.13.2

1 Core	1
1.1 Getting Started	2
1.2 Features	2
2 Hierarchical Index	3
2.1 Class Hierarchy	3
3 Class Index	5
3.1 Class List	5
4 Class Documentation	8
4.1 Async Class Reference	8
4.1.1 Detailed Description	9
4.2 AutoChooser Class Reference	9
4.2.1 Detailed Description	9
4.2.2 Constructor & Destructor Documentation	9
4.2.3 Member Function Documentation	10
4.2.4 Member Data Documentation	10
4.3 BasicSolenoidSet Class Reference	10
4.3.1 Detailed Description	11
4.3.2 Constructor & Destructor Documentation	11
4.3.3 Member Function Documentation	11
4.4 BasicSpinCommand Class Reference	11
4.4.1 Detailed Description	12
4.4.2 Constructor & Destructor Documentation	12
4.4.3 Member Function Documentation	12
4.5 BasicStopCommand Class Reference	13
4.5.1 Detailed Description	13
4.5.2 Constructor & Destructor Documentation	13
4.5.3 Member Function Documentation	13
4.6 Branch Class Reference	14
4.6.1 Detailed Description	14
4.7 screen::ButtonWidget Class Reference	14
4.7.1 Detailed Description	14
4.7.2 Constructor & Destructor Documentation	14
4.7.3 Member Function Documentation	15
4.8 CommandController Class Reference	16
4.8.1 Detailed Description	16
4.8.2 Constructor & Destructor Documentation	16
4.8.3 Member Function Documentation	16
4.9 Condition Class Reference	18
4.9.1 Detailed Description	19
4.10 CustomEncoder Class Reference	19

4.10.1 Detailed Description	19
4.10.2 Constructor & Destructor Documentation	19
4.10.3 Member Function Documentation	19
4.11 DelayCommand Class Reference	21
4.11.1 Detailed Description	21
4.11.2 Constructor & Destructor Documentation	21
4.11.3 Member Function Documentation	22
4.12 DriveForwardCommand Class Reference	22
4.12.1 Detailed Description	22
4.12.2 Constructor & Destructor Documentation	22
4.12.3 Member Function Documentation	23
4.13 DriveStopCommand Class Reference	23
4.13.1 Detailed Description	23
4.13.2 Constructor & Destructor Documentation	23
4.13.3 Member Function Documentation	24
4.14 DriveToPointCommand Class Reference	24
4.14.1 Detailed Description	24
4.14.2 Constructor & Destructor Documentation	24
4.14.3 Member Function Documentation	25
4.15 AutoChooser::entry_t Struct Reference	26
4.15.1 Detailed Description	26
4.15.2 Member Data Documentation	26
4.16 ExponentialMovingAverage Class Reference	26
4.16.1 Detailed Description	27
4.16.2 Constructor & Destructor Documentation	27
4.16.3 Member Function Documentation	27
4.17 Feedback Class Reference	28
4.17.1 Detailed Description	29
4.17.2 Member Function Documentation	29
4.18 FeedForward Class Reference	30
4.18.1 Detailed Description	31
4.18.2 Constructor & Destructor Documentation	31
4.18.3 Member Function Documentation	31
4.19 FeedForward::ff_config_t Struct Reference	32
4.19.1 Detailed Description	32
4.19.2 Member Data Documentation	32
4.20 Filter Class Reference	33
4.20.1 Detailed Description	33
4.21 Flywheel Class Reference	33
4.21.1 Detailed Description	34
4.21.2 Constructor & Destructor Documentation	34
4.21.3 Member Function Documentation	35

4.21.4 Friends And Related Symbol Documentation	37
4.22 FlywheelStopCommand Class Reference	37
4.22.1 Detailed Description	37
4.22.2 Constructor & Destructor Documentation	37
4.22.3 Member Function Documentation	38
4.23 FlywheelStopMotorsCommand Class Reference	38
4.23.1 Detailed Description	38
4.23.2 Constructor & Destructor Documentation	38
4.23.3 Member Function Documentation	39
4.24 FlywheelStopNonTasksCommand Class Reference	39
4.24.1 Detailed Description	39
4.25 FunctionCommand Class Reference	39
4.25.1 Detailed Description	39
4.26 FunctionCondition Class Reference	40
4.26.1 Detailed Description	40
4.27 screen::FunctionPage Class Reference	40
4.27.1 Detailed Description	41
4.27.2 Constructor & Destructor Documentation	41
4.27.3 Member Function Documentation	41
4.28 GenericAuto Class Reference	42
4.28.1 Detailed Description	42
4.28.2 Member Function Documentation	42
4.29 PurePursuit::hermite_point Struct Reference	43
4.29.1 Detailed Description	44
4.30 IfTimePassed Class Reference	44
4.30.1 Detailed Description	44
4.31 InOrder Class Reference	44
4.31.1 Detailed Description	44
4.32 Lift< T > Class Template Reference	45
4.32.1 Detailed Description	45
4.32.2 Constructor & Destructor Documentation	45
4.32.3 Member Function Documentation	46
4.33 Lift< T >::lift_cfg_t Struct Reference	49
4.33.1 Detailed Description	49
4.34 Logger Class Reference	49
4.34.1 Detailed Description	50
4.34.2 Constructor & Destructor Documentation	50
4.34.3 Member Function Documentation	50
4.35 MotionController::m_profile_cfg_t Struct Reference	52
4.35.1 Detailed Description	52
4.36 StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage Class Reference	53
4.36.1 Detailed Description	53

4.36.2 Constructor & Destructor Documentation	53
4.36.3 Member Function Documentation	54
4.37 MecanumDrive Class Reference	54
4.37.1 Detailed Description	55
4.37.2 Constructor & Destructor Documentation	55
4.37.3 Member Function Documentation	55
4.38 MecanumDrive::mecanumdrive_config_t Struct Reference	57
4.38.1 Detailed Description	57
4.39 motion_t Struct Reference	57
4.39.1 Detailed Description	58
4.40 MotionController Class Reference	58
4.40.1 Detailed Description	59
4.40.2 Constructor & Destructor Documentation	59
4.40.3 Member Function Documentation	59
4.41 MovingAverage Class Reference	62
4.41.1 Detailed Description	62
4.41.2 Constructor & Destructor Documentation	62
4.41.3 Member Function Documentation	63
4.42 Odometry3Wheel Class Reference	64
4.42.1 Detailed Description	65
4.42.2 Constructor & Destructor Documentation	65
4.42.3 Member Function Documentation	65
4.43 Odometry3Wheel::odometry3wheel_cfg_t Struct Reference	66
4.43.1 Detailed Description	66
4.43.2 Member Data Documentation	66
4.44 OdometryBase Class Reference	67
4.44.1 Detailed Description	68
4.44.2 Constructor & Destructor Documentation	68
4.44.3 Member Function Documentation	68
4.44.4 Member Data Documentation	71
4.45 screen::OdometryPage Class Reference	72
4.45.1 Detailed Description	73
4.45.2 Constructor & Destructor Documentation	73
4.45.3 Member Function Documentation	73
4.46 OdometryTank Class Reference	74
4.46.1 Detailed Description	75
4.46.2 Constructor & Destructor Documentation	75
4.46.3 Member Function Documentation	76
4.47 OdomSetPosition Class Reference	77
4.47.1 Detailed Description	77
4.47.2 Constructor & Destructor Documentation	77
4.47.3 Member Function Documentation	77

4.48 screen::Page Class Reference	78
4.48.1 Detailed Description	78
4.48.2 Member Function Documentation	78
4.49 Parallel Class Reference	79
4.49.1 Detailed Description	79
4.50 PurePursuit::Path Class Reference	79
4.50.1 Detailed Description	79
4.50.2 Constructor & Destructor Documentation	79
4.50.3 Member Function Documentation	80
4.51 PID Class Reference	80
4.51.1 Detailed Description	81
4.51.2 Member Enumeration Documentation	81
4.51.3 Constructor & Destructor Documentation	81
4.51.4 Member Function Documentation	82
4.51.5 Member Data Documentation	85
4.52 PID::pid_config_t Struct Reference	86
4.52.1 Detailed Description	86
4.52.2 Member Data Documentation	86
4.53 screen::PIDPage Class Reference	87
4.53.1 Detailed Description	87
4.53.2 Constructor & Destructor Documentation	87
4.53.3 Member Function Documentation	88
4.54 Pose2d Class Reference	88
4.54.1 Detailed Description	89
4.54.2 Constructor & Destructor Documentation	89
4.54.3 Member Function Documentation	91
4.54.4 Friends And Related Symbol Documentation	95
4.55 PurePursuitCommand Class Reference	95
4.55.1 Detailed Description	95
4.55.2 Constructor & Destructor Documentation	95
4.55.3 Member Function Documentation	96
4.56 Rect Struct Reference	96
4.56.1 Detailed Description	96
4.57 robot_specs_t Struct Reference	96
4.57.1 Detailed Description	97
4.57.2 Member Data Documentation	97
4.58 Rotation2d Class Reference	97
4.58.1 Detailed Description	98
4.58.2 Constructor & Destructor Documentation	98
4.58.3 Member Function Documentation	99
4.58.4 Friends And Related Symbol Documentation	104
4.59 ScaledSphericalSimplexSigmaPoints< STATES > Class Template Reference	105

4.59.1 Detailed Description	105
4.59.2 Constructor & Destructor Documentation	105
4.59.3 Member Function Documentation	106
4.60 screen::ScreenData Struct Reference	107
4.60.1 Detailed Description	107
4.61 Serializer Class Reference	107
4.61.1 Detailed Description	108
4.61.2 Constructor & Destructor Documentation	108
4.61.3 Member Function Documentation	108
4.62 screen::SliderWidget Class Reference	111
4.62.1 Detailed Description	111
4.62.2 Constructor & Destructor Documentation	112
4.62.3 Member Function Documentation	112
4.63 SpinRPMCommand Class Reference	112
4.63.1 Detailed Description	113
4.63.2 Constructor & Destructor Documentation	113
4.63.3 Member Function Documentation	113
4.64 PurePursuit::spline Struct Reference	113
4.64.1 Detailed Description	114
4.65 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS > Class Template Reference .	114
4.65.1 Detailed Description	115
4.65.2 Constructor & Destructor Documentation	115
4.65.3 Member Function Documentation	117
4.66 StateMachine< System, IDType, Message, delay_ms, do_log >::State Struct Reference .	121
4.66.1 Detailed Description	121
4.67 StateMachine< System, IDType, Message, delay_ms, do_log > Class Template Reference .	121
4.67.1 Detailed Description	122
4.67.2 Constructor & Destructor Documentation	122
4.67.3 Member Function Documentation	122
4.68 screen::StatsPage Class Reference	123
4.68.1 Detailed Description	123
4.68.2 Constructor & Destructor Documentation	123
4.68.3 Member Function Documentation	124
4.69 TakeBackHalf Class Reference	124
4.69.1 Detailed Description	125
4.69.2 Member Function Documentation	125
4.70 TankDrive Class Reference	127
4.70.1 Detailed Description	127
4.70.2 Member Enumeration Documentation	127
4.70.3 Constructor & Destructor Documentation	128
4.70.4 Member Function Documentation	128
4.71 tracking_wheel_cfg_t Struct Reference	136

4.71.1 Detailed Description	136
4.71.2 Member Data Documentation	137
4.72 Transform2d Class Reference	137
4.72.1 Detailed Description	138
4.72.2 Constructor & Destructor Documentation	138
4.72.3 Member Function Documentation	141
4.72.4 Friends And Related Symbol Documentation	143
4.73 Translation2d Class Reference	143
4.73.1 Detailed Description	144
4.73.2 Constructor & Destructor Documentation	144
4.73.3 Member Function Documentation	145
4.73.4 Friends And Related Symbol Documentation	150
4.74 TrapezoidProfile Class Reference	150
4.74.1 Detailed Description	151
4.74.2 Constructor & Destructor Documentation	151
4.74.3 Member Function Documentation	151
4.75 TurnDegreesCommand Class Reference	153
4.75.1 Detailed Description	153
4.75.2 Constructor & Destructor Documentation	153
4.75.3 Member Function Documentation	154
4.76 TurnToHeadingCommand Class Reference	154
4.76.1 Detailed Description	154
4.76.2 Constructor & Destructor Documentation	154
4.76.3 Member Function Documentation	155
4.77 Twist2d Class Reference	155
4.77.1 Detailed Description	156
4.77.2 Constructor & Destructor Documentation	156
4.77.3 Member Function Documentation	157
4.77.4 Friends And Related Symbol Documentation	158
4.78 WaitUntilCondition Class Reference	158
4.78.1 Detailed Description	159
4.79 WaitUntilUpToSpeedCommand Class Reference	159
4.79.1 Detailed Description	159
4.79.2 Constructor & Destructor Documentation	159
4.79.3 Member Function Documentation	159
Index	161

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

If you just want to start a project with Core, make a fork of the [Fork Template](#) and follow it's instructions.

To setup core for an existing project:

1. Create a new vex project (using the VSCode extension or other methods)
2. Initialize a git repository for the project
3. Execute `git subtree add --prefix=core https://github.com/RIT-VEX-U/Core.git main`
4. Update the vex Makefile (or any other build system) to know about the core files (`core/src` for source files, `core/include` for headers) (See [here](#) for an example)
5. Enable [Eigen](#) (Latest supported version is 3.4.0):
 - `mkdir vendor`
 - `git submodule add https://gitlab.com/libeigen/eigen.git vendor/eigen`
 - `cd vendor/eigen`
 - `git checkout 3.4.0`
 - Add the following to the `makefile` to give Core access to the library: `INC += -Ivendor/eigen` (See [here](#) for an example)

If you only wish to use a single version of Core, you can simply clone `core/` into your project and add the core source and header files to your makefile.

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
 - Tank (Differential)
 - [N-Pod](#)
- [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)

2 Hierarchical Index

2.1 Class Hierarchy

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

Async	8
BasicSolenoidSet	10
BasicSpinCommand	11
BasicStopCommand	13
Branch	14
screen::ButtonWidget	14
CommandController	16
Condition	18
FunctionCondition	40
IfTimePassed	44
CustomEncoder	19
DelayCommand	21
DriveForwardCommand	22
DriveStopCommand	23
DriveToPointCommand	24
AutoChooser::entry_t	26
Feedback	28
MotionController	58
PID	80
TakeBackHalf	124
FeedForward	30
FeedForward::ff_config_t	32
Filter	33
ExponentialMovingAverage	26
MovingAverage	62
Flywheel	33
FlywheelStopCommand	37
FlywheelStopMotorsCommand	38

FlywheelStopNonTasksCommand	39
FunctionCommand	39
GenericAuto	42
PurePursuit::hermite_point	43
InOrder	44
Lift< T >	45
Lift< T >::lift_cfg_t	49
Logger	49
MotionController::m_profile_cfg_t	52
StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage	53
MecanumDrive	54
MecanumDrive::mecanumdrive_config_t	57
motion_t	57
Odometry3Wheel::odometry3wheel_cfg_t	66
OdometryBase	67
Odometry3Wheel	64
OdometryTank	74
OdomSetPosition	77
screen::Page	78
AutoChooser	9
screen::FunctionPage	40
screen::OdometryPage	72
screen::PIDPage	87
screen::StatsPage	123
Parallel	79
PurePursuit::Path	79
PID::pid_config_t	86
Pose2d	88
PurePursuitCommand	95
Rect	96
robot_specs_t	96
Rotation2d	97

ScaledSphericalSimplexSigmaPoints< STATES >	105
screen::ScreenData	107
Serializer	107
screen::SliderWidget	111
SpinRPMCommand	112
PurePursuit::spline	113
SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >	114
StateMachine< System, IDType, Message, delay_ms, do_log >::State	121
StateMachine< System, IDType, Message, delay_ms, do_log >	121
TankDrive	127
tracking_wheel_cfg_t	136
Transform2d	137
Translation2d	143
TrapezoidProfile	150
TurnDegreesCommand	153
TurnToHeadingCommand	154
Twist2d	155
WaitUntilCondition	158
WaitUntilUpToSpeedCommand	159

3 Class Index

3.1 Class List

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

Async	
Async runs a command asynchronously will simply let it go and never look back THIS HAS A VERY NICHE USE CASE. THINK ABOUT IF YOU REALLY NEED IT	8
AutoChooser	9
BasicSolenoidSet	10
BasicSpinCommand	11
BasicStopCommand	13
Branch	
Branch chooses from multiple options at runtime. the function decider returns an index into the choices vector If you wish to make no choice and skip this section, return NO_CHOICE; any choice that is out of bounds set to NO_CHOICE	14

screen::ButtonWidget	
Widget that does something when you tap it. The function is only called once when you first tap it	14
CommandController	16
Condition	18
CustomEncoder	19
DelayCommand	21
DriveForwardCommand	22
DriveStopCommand	23
DriveToPointCommand	24
AutoChooser::entry_t	26
ExponentialMovingAverage	26
Feedback	28
FeedForward	30
FeedForward::ff_config_t	32
Filter	33
Flywheel	33
FlywheelStopCommand	37
FlywheelStopMotorsCommand	38
FlywheelStopNonTasksCommand	39
FunctionCommand	39
FunctionCondition	
FunctionCondition is a quick and dirty Condition to wrap some expression that should be evaluated at runtime	40
screen::FunctionPage	
Simple page that stores no internal data. the draw and update functions use only global data rather than storing anything	40
GenericAuto	42
PurePursuit::hermite_point	43
IfTimePassed	
IfTimePassed tests based on time since the command controller was constructed. Returns true if elapsed time > time_s	44
InOrder	
InOrder runs its commands sequentially then continues. How to handle timeout in this case. Automatically set it to sum of commands timeouts?	44
Lift< T >	45

Lift< T >::lift_cfg_t	49
Logger	
Class to simplify writing to files	49
MotionController::m_profile_cfg_t	52
StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage	
MaybeMessage a message of Message type or nothing	
MaybeMessage m = Message::EnumField1	53
MecanumDrive	54
MecanumDrive::mecanumdrive_config_t	57
motion_t	57
MotionController	58
MovingAverage	62
Odometry3Wheel	64
Odometry3Wheel::odometry3wheel_cfg_t	66
OdometryBase	67
screen::OdometryPage	
Page that shows odometry position and rotation and a map (if an sd card with the file is on)	72
OdometryTank	74
OdomSetPosition	77
screen::Page	
Page describes one part of the screen slideshow	78
Parallel	
Parallel runs multiple commands in parallel and waits for all to finish before continuing. if none	
finish before this command's timeout, it will call on_timeout on all children continue	79
PurePursuit::Path	79
PID	80
PID::pid_config_t	86
screen::PIDPage	
PIDPage provides a way to tune a pid controller on the screen	87
Pose2d	88
PurePursuitCommand	95
Rect	96
robot_specs_t	96
Rotation2d	97
ScaledSphericalSimplexSigmaPoints< STATES >	105

screen::ScreenData	Holds the data that will be passed to the screen thread you probably shouldnt have to use it	107
Serializer	Serializes Arbitrary data to a file on the SD Card	107
screen::SliderWidget	Widget that updates a double value. Updates by reference so watch out for race conditions cuz the screen stuff lives on another thread	111
SpinRPMCommand		112
PurePursuit::spline		113
SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >		114
StateMachine< System, IDType, Message, delay_ms, do_log >::State		121
StateMachine< System, IDType, Message, delay_ms, do_log >	State Machine))))) A fun fun way of controlling stateful subsystems - used in the 2023-2024 Over Under game for our overly complex intake-cata subsystem (see there for an example) The statemachine runs in a background thread and a user thread can interact with it through current_state and send_message	121
screen::StatsPage	Draws motor stats and battery stats to the screen	123
TakeBackHalf	A velocity controller	124
TankDrive		127
tracking_wheel_cfg_t		136
Transform2d		137
Translation2d		143
TrapezoidProfile		150
TurnDegreesCommand		153
TurnToHeadingCommand		154
Twist2d		155
WaitUntilCondition	Waits until the condition is true	158
WaitUntilUpToSpeedCommand		159

4 Class Documentation

4.1 Async Class Reference

[Async](#) runs a command asynchronously will simply let it go and never look back THIS HAS A VERY NICHE USE CASE. THINK ABOUT IF YOU REALLY NEED IT.

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

4.1.1 Detailed Description

[Async](#) runs a command asynchronously will simply let it go and never look back THIS HAS A VERY NICHE USE CASE. THINK ABOUT IF YOU REALLY NEED IT.

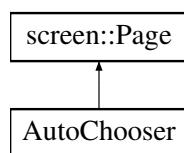
The documentation for this class was generated from the following files:

- auto_command.h
- auto_command.cpp

4.2 AutoChooser Class Reference

```
#include <auto_chooser.h>
```

Inheritance diagram for AutoChooser:



Classes

- struct [entry_t](#)

Public Member Functions

- [AutoChooser \(std::vector< std::string > paths, size_t def=0\)](#)
- [size_t get_choice \(\)](#)

Protected Attributes

- [size_t choice](#)
- [std::vector< entry_t > list](#)

4.2.1 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 autonomy 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.

4.2.2 Constructor & Destructor Documentation

AutoChooser()

```
AutoChooser::AutoChooser (
    std::vector< std::string > paths,
    size_t def = 0)
```

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

Parameters

<i>brain</i>	the brain on which to draw the selection boxes
--------------	--

4.2.3 Member Function Documentation**get_choice()**

```
size_t AutoChooser::get_choice ()
```

Get the currently selected auto choice

Returns

the identifier to the auto path

Return the selected autonomous

4.2.4 Member Data Documentation**choice**

```
size_t AutoChooser::choice [protected]
```

the current choice of auto

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:

- [auto_chooser.h](#)
- [auto_chooser.cpp](#)

4.3 BasicSolenoidSet Class Reference

```
#include <basic_command.h>
```

Public Member Functions

- [BasicSolenoidSet](#) (vex::pneumatics &solenoid, bool setting)
Construct a new BasicSolenoidSet Command.
- bool [run \(\)](#) override
Runs the BasicSolenoidSet Overrides run command from AutoCommand.

4.3.1 Detailed Description

AutoCommand wrapper class for [BasicSolenoidSet](#) Using the Vex hardware functions

4.3.2 Constructor & Destructor Documentation

BasicSolenoidSet()

```
BasicSolenoidSet::BasicSolenoidSet (
    vex::pneumatics & solenoid,
    bool setting)
```

Construct a new [BasicSolenoidSet](#) Command.

Parameters

<i>solenoid</i>	Solenoid being set
<i>setting</i>	Setting of the solenoid in boolean (true,false)

4.3.3 Member Function Documentation

run()

```
bool BasicSolenoidSet::run () [override]
```

Runs the [BasicSolenoidSet](#) Overrides run command from AutoCommand.

Returns

True Command runs once

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

- basic_command.h
- basic_command.cpp

4.4 BasicSpinCommand Class Reference

```
#include <basic_command.h>
```

Public Member Functions

- [BasicSpinCommand](#) (vex::motor &motor, vex::directionType dir, BasicSpinCommand::type setting, double power)

Construct a new BasicSpinCommand.

- bool [run \(\)](#) override

Runs the BasicSpinCommand Overrides run from Auto Command.

4.4.1 Detailed Description

AutoCommand wrapper class for [BasicSpinCommand](#) using the vex hardware functions

4.4.2 Constructor & Destructor Documentation

BasicSpinCommand()

```
BasicSpinCommand::BasicSpinCommand (
    vex::motor & motor,
    vex::directionType dir,
    BasicSpinCommand::type setting,
    double power)
```

Construct a new [BasicSpinCommand](#).

a BasicMotorSpin Command

Parameters

<i>motor</i>	Motor to spin
<i>direc</i>	Direction of motor spin
<i>setting</i>	Power setting in volts,percentage,velocity
<i>power</i>	Value of desired power
<i>motor</i>	Motor port to spin
<i>dir</i>	Direction for spinning
<i>setting</i>	Power setting in volts,percentage,velocity
<i>power</i>	Value of desired power

4.4.3 Member Function Documentation

run()

```
bool BasicSpinCommand::run () [override]
```

Runs the [BasicSpinCommand](#) Overrides run from Auto Command.

Run the [BasicSpinCommand](#) Overrides run from Auto Command.

Returns

True [Async](#) running command

True Command runs once

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

- basic_command.h
- basic_command.cpp

4.5 BasicStopCommand Class Reference

```
#include <basic_command.h>
```

Public Member Functions

- [BasicStopCommand](#) (vex::motor &motor, vex::brakeType setting)
Construct a new BasicMotorStop Command.
- bool [run \(\)](#) override
Runs the BasicMotorStop Command Overrides run command from AutoCommand.

4.5.1 Detailed Description

AutoCommand wrapper class for [BasicStopCommand](#) Using the Vex hardware functions

4.5.2 Constructor & Destructor Documentation

[BasicStopCommand\(\)](#)

```
BasicStopCommand::BasicStopCommand (
    vex::motor & motor,
    vex::brakeType setting)
```

Construct a new BasicMotorStop Command.

Construct a BasicMotorStop Command.

Parameters

<i>motor</i>	The motor to stop
<i>setting</i>	The brake setting for the motor
<i>motor</i>	Motor to stop
<i>setting</i>	Braketype setting brake,coast,hold

4.5.3 Member Function Documentation

[run\(\)](#)

```
bool BasicStopCommand::run () [override]
```

Runs the BasicMotorStop Command Overrides run command from AutoCommand.

Runs the BasicMotorStop command Overrides run command from AutoCommand.

Returns

True Command runs once

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

- basic_command.h
- basic_command.cpp

4.6 Branch Class Reference

Branch chooses from multiple options at runtime. the function decider returns an index into the choices vector If you wish to make no choice and skip this section, return NO_CHOICE; any choice that is out of bounds set to NO_CHOICE.

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

4.6.1 Detailed Description

Branch chooses from multiple options at runtime. the function decider returns an index into the choices vector If you wish to make no choice and skip this section, return NO_CHOICE; any choice that is out of bounds set to NO_CHOICE.

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

- auto_command.h
- auto_command.cpp

4.7 screen::ButtonWidget Class Reference

Widget that does something when you tap it. The function is only called once when you first tap it.

```
#include <screen.h>
```

Public Member Functions

- **ButtonWidget** (std::function< void(void)> onpress, **Rect** rect, std::string name)
Create a Button widget.
- **ButtonWidget** (void(*onpress)(), **Rect** rect, std::string name)
Create a Button widget.
- bool **update** (bool was_pressed, int x, int y)
responds to user input
- void **draw** (vex::brain::lcd &, bool first_draw, unsigned int frame_number)
draws the button to the screen

4.7.1 Detailed Description

Widget that does something when you tap it. The function is only called once when you first tap it.

4.7.2 Constructor & Destructor Documentation

ButtonWidget() [1/2]

```
screen::ButtonWidget::ButtonWidget (
    std::function< void(void)> onpress,
    Rect rect,
    std::string name)  [inline]
```

Create a Button widget.

Parameters

<i>onpress</i>	the function to be called when the button is tapped
<i>rect</i>	the area the button should take up on the screen
<i>name</i>	the label put on the button

ButtonWidget() [2/2]

```
screen::ButtonWidget::ButtonWidget (
    void(* onpress)(),
    Rect rect,
    std::string name) [inline]
```

Create a Button widget.

Parameters

<i>onpress</i>	the function to be called when the button is tapped
<i>rect</i>	the area the button should take up on the screen
<i>name</i>	the label put on the button

4.7.3 Member Function Documentation**update()**

```
bool screen::ButtonWidget::update (
    bool was_pressed,
    int x,
    int y)
```

responds to user input

Parameters

<i>was_pressed</i>	if the screen is pressed
<i>x</i>	x position if the screen was pressed
<i>y</i>	y position if the screen was pressed

Returns

true if the button was pressed

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

- screen.h
- screen.cpp

4.8 CommandController Class Reference

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

Public Member Functions

- **CommandController ()**
Create an empty [CommandController](#). Add Command with [CommandController::add\(\)](#)
- **CommandController (std::initializer_list< AutoCommand * > cmd)**
Create a [CommandController](#) with commands pre added. More can be added with [CommandController::add\(\)](#)
- void **add** (std::vector< AutoCommand * > cmd)
- void **add** (AutoCommand *cmd, double timeout_seconds=10.0)
- void **add** (std::vector< AutoCommand * > cmd, double timeout_sec)
- void **add_delay** (int ms)
- void **add_cancel_func** (std::function< bool(void)> true_if_cancel)
add_cancel_func specifies that when this func evaluates to true, to cancel the command controller
- void **run** ()
- bool **last_command_timed_out** ()

4.8.1 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.

4.8.2 Constructor & Destructor Documentation

CommandController()

```
CommandController::CommandController (
    std::initializer_list< AutoCommand * > cmd) [inline]
```

Create a [CommandController](#) with commands pre added. More can be added with [CommandController::add\(\)](#)

Parameters

<i>cmds</i>	
-------------	--

4.8.3 Member Function Documentation

add() [1/3]

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

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

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

add() [2/3]

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

Adds a command to the queue

Parameters

<i>cmd</i>	the AutoCommand we want to add to our list
<i>timeout_seconds</i>	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

Add multiple commands to the queue. No timeout here.

Parameters

<i>cmds</i>	the AutoCommands we want to add to our list
-------------	---

add() [3/3]

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

Add multiple commands to the queue. No timeout here.

Parameters

<i>cmds</i>	the AutoCommands we want to add to our list Add multiple commands to the queue. No timeout here.
<i>cmds</i>	the AutoCommands we want to add to our list
<i>timeout_sec</i>	timeout in seconds to apply to all commands if they are still the default

Add multiple commands to the queue. No timeout here.

Parameters

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

add_cancel_func()

```
void CommandController::add_cancel_func (
    std::function< bool(void) > true_if_cancel)
```

add_cancel_func specifies that when this func evaluates to true, to cancel the command controller

Parameters

<code>true_if_cancel</code>	a function that returns true when we want to cancel the command controller
-----------------------------	--

add_delay()

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

Adds a command that will delay progression of the queue

Parameters

<code>ms</code>	- number of milliseconds to wait before continuing execution of autonomous
-----------------	--

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

run()

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

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

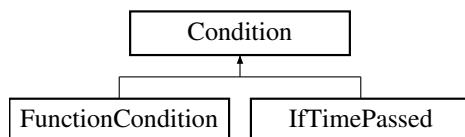
The documentation for this class was generated from the following files:

- `command_controller.h`
- `command_controller.cpp`

4.9 Condition Class Reference

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

Inheritance diagram for Condition:



4.9.1 Detailed Description

File: `auto_command.h` Desc: Interface for module-specific commands A `Condition` is a function that returns true or false `is_even` is a predicate that would return true if a number is even For our purposes, a `Condition` is a choice to be made at runtime `drive_sys.reached_point(10, 30)` is a predicate `time.has_elapsed(10, vex::seconds)` is a predicate extend this class for different choices you wish to make

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

- `auto_command.h`
- `auto_command.cpp`

4.10 CustomEncoder Class Reference

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

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)`

4.10.1 Detailed Description

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

4.10.2 Constructor & Destructor Documentation

`CustomEncoder()`

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

Construct an encoder with a custom number of ticks

Parameters

<code>port</code>	the tripot port on the brain the encoder is plugged into
<code>ticks_per_rev</code>	the number of ticks the encoder will report for one revolution

4.10.3 Member Function Documentation

`position()`

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

get the position that the encoder is at

Parameters

<i>units</i>	the unit we want the return value to be in
--------------	--

Returns

the position of the encoder in the units specified

rotation()

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

get the rotation that the encoder is at

Parameters

<i>units</i>	the unit we want the return value to be in
--------------	--

Returns

the rotation of the encoder in the units specified

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

<i>val</i>	the numerical value of the position we are setting to
<i>units</i>	the unit of val

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

<i>val</i>	the numerical value of the angle we are setting to
<i>units</i>	the unit of val

velocity()

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

get the velocity that the encoder is moving at

Parameters

<i>units</i>	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:

- custom_encoder.h
- custom_encoder.cpp

4.11 DelayCommand Class Reference

```
#include <delay_command.h>
```

Public Member Functions

- [DelayCommand](#) (int ms)
- bool [run \(\)](#) override

4.11.1 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

4.11.2 Constructor & Destructor Documentation

DelayCommand()

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

Construct a delay command

Parameters

<i>ms</i>	the number of milliseconds to delay for
-----------	---

4.11.3 Member Function Documentation

run()

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

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

Returns

true when complete

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

- delay_command.h

4.12 DriveForwardCommand Class Reference

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

Public Member Functions

- [DriveForwardCommand](#) ([TankDrive](#) &drive_sys, [Feedback](#) &feedback, double inches, directionType dir, double max_speed=1, double end_speed=0)
- bool [run](#) () override
- void [on_timeout](#) () override

4.12.1 Detailed Description

AutoCommand wrapper class for the drive_forward function in the [TankDrive](#) class

4.12.2 Constructor & Destructor Documentation

DriveForwardCommand()

```
DriveForwardCommand::DriveForwardCommand (
    TankDrive & drive_sys,
    Feedback & feedback,
    double inches,
    directionType dir,
    double max_speed = 1,
    double end_speed = 0)
```

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

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

4.12.3 Member Function Documentation**on_timeout()**

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

Cleans up drive system if we time out before finishing

reset the drive system if we timeout

run()

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

Run drive_forward Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

- drive_commands.h
- drive_commands.cpp

4.13 DriveStopCommand Class Reference

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

Public Member Functions

- [DriveStopCommand \(TankDrive &drive_sys\)](#)
- [bool run \(\) override](#)

4.13.1 Detailed Description

AutoCommand wrapper class for the stop() function in the [TankDrive](#) class

4.13.2 Constructor & Destructor Documentation**DriveStopCommand()**

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

Construct a DriveStop Command

Parameters

<i>drive_sys</i>	the drive system we are commanding
------------------	------------------------------------

4.13.3 Member Function Documentation**run()**

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

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

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

- `drive_commands.h`
- `drive_commands.cpp`

4.14 DriveToPointCommand Class Reference

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

Public Member Functions

- `DriveToPointCommand (TankDrive &drive_sys, Feedback &feedback, double x, double y, directionType dir, double max_speed=1, double end_speed=0)`
- `DriveToPointCommand (TankDrive &drive_sys, Feedback &feedback, Translation2d translation, directionType dir, double max_speed=1, double end_speed=0)`
- `bool run () override`

4.14.1 Detailed Description

AutoCommand wrapper class for the `drive_to_point` function in the `TankDrive` class

4.14.2 Constructor & Destructor Documentation**DriveToPointCommand() [1/2]**

```
DriveToPointCommand::DriveToPointCommand (
    TankDrive & drive_sys,
    Feedback & feedback,
    double x,
    double y,
    directionType dir,
    double max_speed = 1,
    double end_speed = 0)
```

Construct a DriveForward Command

Parameters

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

DriveToPointCommand() [2/2]

```
DriveToPointCommand::DriveToPointCommand (
    TankDrive & drive_sys,
    Feedback & feedback,
    Translation2d translation,
    directionType dir,
    double max_speed = 1,
    double end_speed = 0)
```

Construct a DriveForward Command

Parameters

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

4.14.3 Member Function Documentation**run()**

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

Run drive_to_point Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

- drive_commands.h
- drive_commands.cpp

4.15 AutoChooser::entry_t Struct Reference

```
#include <auto_chooser.h>
```

Public Attributes

- std::string [name](#)

4.15.1 Detailed Description

[entry_t](#) is a datatype used to store information that the chooser knows about an auto selection button

4.15.2 Member Data Documentation

[name](#)

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

name of the auto represented by the block

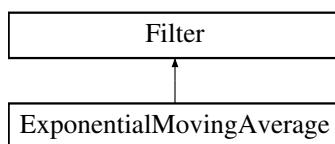
The documentation for this struct was generated from the following file:

- [auto_chooser.h](#)

4.16 ExponentialMovingAverage Class Reference

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

Inheritance diagram for ExponentialMovingAverage:



Public Member Functions

- [ExponentialMovingAverage](#) (int buffer_size)
- [ExponentialMovingAverage](#) (int buffer_size, double starting_value)
- void [add_entry](#) (double n) override
- double [get_value](#) () const override
- int [get_size](#) ()

4.16.1 Detailed Description

ExponentialMovingAverage

An exponential 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.

A simple moving average lags significantly with time as it has to counteract old samples. An exponential moving average keeps more up to date by weighting newer readings higher than older readings so it is more up to date while also still smoothed.

The [ExponentialMovingAverage](#) class provides a simple interface to do this smoothing from our noisy sensor values.

4.16.2 Constructor & Destructor Documentation

ExponentialMovingAverage() [1/2]

```
ExponentialMovingAverage::ExponentialMovingAverage (
    int buffer_size)
```

Create a moving average calculator with 0 as the default value

Parameters

<i>buffer_size</i>	The size of the buffer. The number of samples that constitute a valid reading
--------------------	---

ExponentialMovingAverage() [2/2]

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

Create a moving average calculator with a specified default value

Parameters

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

4.16.3 Member Function Documentation

add_entry()

```
void ExponentialMovingAverage::add_entry (
    double n) [override], [virtual]
```

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

Parameters

<code>n</code>	the sample that will be added to the moving average.
----------------	--

Implements [Filter](#).

get_size()

```
int ExponentialMovingAverage::get_size ()
```

How many samples the average is made from

Returns

the number of samples used to calculate this average

get_value()

```
double ExponentialMovingAverage::get_value () const [override], [virtual]
```

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

Implements [Filter](#).

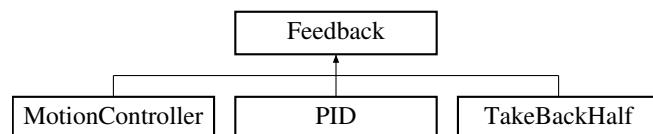
The documentation for this class was generated from the following files:

- `moving_average.h`
- `moving_average.cpp`

4.17 Feedback Class Reference

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

Inheritance diagram for Feedback:



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

4.17.1 Detailed Description

Interface so that subsystems can easily switch between feedback loops

Author

Ryan McGee

Date

9/25/2022

4.17.2 Member Function Documentation

`get()`

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

Returns

the last saved result from the feedback controller

Implemented in [MotionController](#), [PID](#), and [TakeBackHalf](#).

`init()`

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

Initialize the feedback controller for a movement

Parameters

<code>start_pt</code>	the current sensor value
<code>set_pt</code>	where the sensor value should be
<code>start_vel</code>	Movement starting velocity
<code>end_vel</code>	Movement ending velocity

Implemented in [MotionController](#), [PID](#), and [TakeBackHalf](#).

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 [TakeBackHalf](#).

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

<i>lower</i>	Upper limit
<i>upper</i>	Lower limit

Implemented in [MotionController](#), [PID](#), and [TakeBackHalf](#).

update()

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

Iterate the feedback loop once with an updated sensor value

Parameters

<i>val</i>	value from the sensor
------------	-----------------------

Returns

feedback loop result

Implemented in [MotionController](#), [PID](#), and [TakeBackHalf](#).

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

- `feedback_base.h`

4.18 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.

4.18.1 Detailed Description

FeedForward

Stores the feedforward constants, and allows for quick computation. Feedforward 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 recommend 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

4.18.2 Constructor & Destructor Documentation

FeedForward()

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

Creates a `FeedForward` object.

Parameters

<code>cfg</code>	Configuration Struct for tuning
------------------	---------------------------------

4.18.3 Member Function Documentation

calculate()

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

Perform the feedforward calculation.

This calculation is the equation: $F = kG + kS \cdot \text{sgn}(v) + kV \cdot v + kA \cdot a$

Parameters

<i>v</i>	Requested velocity of system
<i>a</i>	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:

- feedforward.h

4.19 FeedForward::ff_config_t Struct Reference

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

Public Attributes

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

4.19.1 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**

4.19.2 Member Data Documentation

kA

```
double FeedForward::ff_config_t::kA
```

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

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.

kS

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

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

kV

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

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

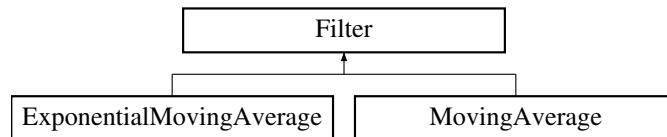
The documentation for this struct was generated from the following file:

- feedforward.h

4.20 Filter Class Reference

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

Inheritance diagram for Filter:



4.20.1 Detailed Description

Interface for filters Use add_entry to supply data and get_value to retrieve the filtered value

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

- moving_average.h

4.21 Flywheel Class Reference

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

Public Member Functions

- `Flywheel` (vex::motor_group &motors, `Feedback` &feedback, `FeedForward` &helper, const double ratio, `Filter` &filt)
 - double `get_target` () const
 - double `getRPM` () const
 - vex::motor_group & `get_motors` () const
 - void `spin_manual` (double speed, directionType dir=fwd)
 - void `spin_rpm` (double rpm)
 - void `stop` ()
 - bool `is_on_target` ()
 - check if the feedback controller thinks the flywheel is on target*
- `screen::Page * Page` () const
 - Creates a page displaying info about the flywheel.*
- AutoCommand * `SpinRpmCmd` (int rpm)
 - Creates a new auto command to spin the flywheel at the desired velocity.*
- AutoCommand * `WaitUntilUpToSpeedCmd` ()
 - Creates a new auto command that will hold until the flywheel has its target as defined by its feedback controller.*

Friends

- int `spinRPMTask` (void *wheelPointer)

4.21.1 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 `spin_rpm` or `stop` on it at any time and it will take all necessary steps to accomplish this

4.21.2 Constructor & Destructor Documentation

`Flywheel()`

```
Flywheel::Flywheel (
    vex::motor_group & motors,
    Feedback & feedback,
    FeedForward & helper,
    const double ratio,
    Filter & filt)
```

Create the `Flywheel` object using `PID` + feedforward for control.

Parameters

<code>motors</code>	pointer to the motors on the fly wheel
<code>feedback</code>	a feedback controller
<code>helper</code>	a feedforward config (only kV is used) to help the feedback controller along
<code>ratio</code>	ratio of the gears from the motor to the flywheel just multiplies the velocity
<code>filter</code>	the filter to use to smooth noisy motor readings

4.21.3 Member Function Documentation

get_motors()

```
motor_group & Flywheel::get_motors () const
```

Returns the motors

Returns

the motors used to run the flywheel

get_target()

```
double Flywheel::get_target () const
```

Return the target_rpm that the flywheel is currently trying to achieve

Returns

target_rpm the target rpm

Return the current value that the target_rpm should be set to

getRPM()

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

return the velocity of the flywheel

is_on_target()

```
bool Flywheel::is_on_target () [inline]
```

check if the feedback controller thinks the flywheel is on target

Returns

true if on target

Page()

```
screen::Page * Flywheel::Page () const
```

Creates a page displaying info about the flywheel.

Returns

the page should be used for `screen::start_screen(screen, {fw.Page()});`

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 target_rpm thread is not running

Parameters

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

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

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

spin_rpm()

```
void Flywheel::spin_rpm (
    double input_rpm)
```

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

Parameters

<i>rpm</i>	- the target_rpm we want to spin at
------------	-------------------------------------

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

Parameters

<i>input_rpm</i>	- set the current RPM
------------------	-----------------------

SpinRpmCmd()

```
AutoCommand * Flywheel::SpinRpmCmd (
    int rpm) [inline]
```

Creates a new auto command to spin the flywheel at the desired velocity.

Parameters

<i>rpm</i>	the rpm to spin at
------------	--------------------

Returns

an auto command to add to a command controller

stop()

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

Stops the motors. If manually spinning, this will do nothing just call spin_mainual(0.0) to send 0 volts stop the RPM thread and the wheel

WaitUntilUpToSpeedCmd()

```
AutoCommand * Flywheel::WaitUntilUpToSpeedCmd () [inline]
```

Creates a new auto command that will hold until the flywheel has its target as defined by its feedback controller.

Returns

an auto command to add to a command controller

4.21.4 Friends And Related Symbol Documentation**spinRPMTask**

```
int spinRPMTask (
    void * wheelPointer) [friend]
```

Runs a thread that keeps track of updating flywheel RPM and controlling it accordingly

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

- [flywheel.h](#)
- [flywheel.cpp](#)

4.22 FlywheelStopCommand Class Reference

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

Public Member Functions

- [FlywheelStopCommand \(Flywheel &flywheel\)](#)
- [bool run \(\) override](#)

4.22.1 Detailed Description

AutoCommand wrapper class for the stop function in the [Flywheel](#) class

4.22.2 Constructor & Destructor Documentation**FlywheelStopCommand()**

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

Construct a [FlywheelStopCommand](#)

Parameters

<i>flywheel</i>	the flywheel system we are commanding
-----------------	---------------------------------------

4.22.3 Member Function Documentation

run()

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

Run stop Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

- `flywheel_commands.h`
- `flywheel_commands.cpp`

4.23 FlywheelStopMotorsCommand Class Reference

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

Public Member Functions

- [FlywheelStopMotorsCommand \(`Flywheel` &*flywheel*\)](#)
- `bool run () override`

4.23.1 Detailed Description

AutoCommand wrapper class for the stopMotors function in the [Flywheel](#) class

4.23.2 Constructor & Destructor Documentation

FlywheelStopMotorsCommand()

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

Construct a FlywheelStopMotors Command

Parameters

<i>flywheel</i>	the flywheel system we are commanding
-----------------	---------------------------------------

4.23.3 Member Function Documentation**run()**

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

Run stop Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

- flywheel_commands.h
- flywheel_commands.cpp

4.24 FlywheelStopNonTasksCommand Class Reference

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

4.24.1 Detailed Description

AutoCommand wrapper class for the stopNonTasks function in the [Flywheel](#) class

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

- flywheel_commands.h
- flywheel_commands.cpp

4.25 FunctionCommand Class Reference

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

4.25.1 Detailed Description

[FunctionCommand](#) is fun and good way to do simple things Printing, launching nukes, and other quick and dirty one time things

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

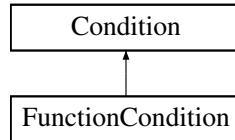
- auto_command.h

4.26 FunctionCondition Class Reference

[FunctionCondition](#) is a quick and dirty [Condition](#) to wrap some expression that should be evaluated at runtime.

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

Inheritance diagram for FunctionCondition:



4.26.1 Detailed Description

[FunctionCondition](#) is a quick and dirty [Condition](#) to wrap some expression that should be evaluated at runtime.

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

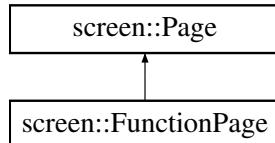
- `auto_command.h`
- `auto_command.cpp`

4.27 screen::FunctionPage Class Reference

Simple page that stores no internal data. the draw and update functions use only global data rather than storing anything.

```
#include <screen.h>
```

Inheritance diagram for screen::FunctionPage:



Public Member Functions

- [FunctionPage](#) (`update_func_t update_f, draw_func_t draw_t`)
Creates a function page.
- void [update](#) (`bool was_pressed, int x, int y`) override
update uses the supplied update function to update this page
- void [draw](#) (`vex::brain::lcd &, bool first_draw, unsigned int frame_number`) override
draw uses the supplied draw function to draw to the screen

4.27.1 Detailed Description

Simple page that stores no internal data. the draw and update functions use only global data rather than storing anything.

4.27.2 Constructor & Destructor Documentation

FunctionPage()

```
screen::FunctionPage::FunctionPage (
    update_func_t update_f,
    draw_func_t draw_f)
```

Creates a function page.

[FunctionPage](#).

Parameters

<i>update_f</i>	the function called every tick to respond to user input or do data collection
<i>draw_t</i>	the function called to draw to the screen
<i>update_f</i>	drawing function
<i>draw_f</i>	drawing function

4.27.3 Member Function Documentation

draw()

```
void screen::FunctionPage::draw (
    vex::brain::lcd & screen,
    bool first_draw,
    unsigned int frame_number) [override], [virtual]
```

draw uses the supplied draw function to draw to the screen

See also

[Page::draw](#)

Reimplemented from [screen::Page](#).

update()

```
void screen::FunctionPage::update (
    bool was_pressed,
    int x,
    int y) [override], [virtual]
```

update uses the supplied update function to update this page

See also

[Page::update](#)

Reimplemented from [screen::Page](#).

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

- screen.h
- screen.cpp

4.28 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)

4.28.1 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

4.28.2 Member Function Documentation

add()

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

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

Parameters

<i>new_state</i>	the function to run
------------------	---------------------

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

<code>async_state</code>	the function to run
--------------------------	---------------------

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

<code>ms</code>	how long to wait in milliseconds
-----------------	----------------------------------

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

<code>blocking</code>	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:

- `generic_auto.h`
- `generic_auto.cpp`

4.29 PurePursuit::hermite_point Struct Reference

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

4.29.1 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:

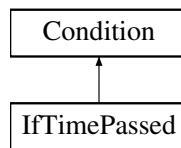
- pure_pursuit.h

4.30 IfTimePassed Class Reference

[IfTimePassed](#) tests based on time since the command controller was constructed. Returns true if elapsed time > time_s.

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

Inheritance diagram for IfTimePassed:



4.30.1 Detailed Description

[IfTimePassed](#) tests based on time since the command controller was constructed. Returns true if elapsed time > time_s.

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

- auto_command.h
- auto_command.cpp

4.31 InOrder Class Reference

[InOrder](#) runs its commands sequentially then continues. How to handle timeout in this case. Automatically set it to sum of commands timeouts?

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

4.31.1 Detailed Description

[InOrder](#) runs its commands sequentially then continues. How to handle timeout in this case. Automatically set it to sum of commands timeouts?

[InOrder](#) runs its commands sequentially then continues. How to handle timeout in this case. Automatically set it to sum of commands timeouts?

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

- auto_command.h
- auto_command.cpp

4.32 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 *homming_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))`

4.32.1 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

4.32.2 Constructor & Destructor Documentation

`Lift()`

```
template<typename T>
Lift< T >::Lift (
    motor_group & lift_motors,
    lift_cfg_t & lift_cfg,
    map< T, double > & setpoint_map,
    limit * homming_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

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

4.32.3 Member Function Documentation**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

<i>up_ctrl</i>	Button controlling the "UP" motion
<i>down_ctrl</i>	Button controlling the "DOWN" motion

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

<i>up_btn</i>	Raise the lift when true
<i>down_btn</i>	Lower the lift when true
<i>volt_up</i>	Motor voltage when raising the lift
<i>volt_down</i>	Motor voltage when lowering the lift

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

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

get_async()

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

Returns

whether or not the background thread is running the lift

get_setpoint()

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

Returns

The current setpoint for the lift

hold()

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

Target the class's setpoint. Calculate the [PID](#) output and set the lift motors accordingly.

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.

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

<i>val</i>	Whether or not the background thread should run the lift
------------	--

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

<i>pos</i>	A lift position enum type
------------	---------------------------

Returns

True if the pid has reached the setpoint

set_sensor_function()

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

<i>fn_ptr</i>	Pointer to custom sensor function
---------------	-----------------------------------

set_sensor_reset()

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

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

<i>val</i>	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:

- lift.h

4.33 Lift< T >::lift_cfg_t Struct Reference

```
#include <lift.h>
```

4.33.1 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:

- lift.h

4.34 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 **Logln** (const std::string &s)
Write a string and newline to the log.
- void **Logln** (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.

Static Public Attributes

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

4.34.1 Detailed Description

Class to simplify writing to files.

4.34.2 Constructor & Destructor Documentation

Logger()

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

Create a logger that will save to a file.

Parameters

<i>filename</i>	the file to save to
-----------------	---------------------

4.34.3 Member Function Documentation

Log() [1/2]

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

Write a string to the log.

Parameters

<i>s</i>	the string to write
----------	---------------------

Log() [2/2]

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

Write a string to the log with a loglevel.

Parameters

<i>level</i>	the level to write. DEBUG, NOTICE, WARNING, ERROR, CRITICAL, TIME
<i>s</i>	the string to write

Logf() [1/2]

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

Write a formatted string to the log.

Parameters

<i>fmt</i>	the format string (like printf)
...	the args

Logf() [2/2]

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

Write a formatted string to the log with a loglevel.

Parameters

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

Logln() [1/2]

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

Write a string and newline to the log.

Parameters

s	the string to write
---	---------------------

Login() [2/2]

```
void Logger::Login (
    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:

- logger.h
- logger.cpp

4.35 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

4.35.1 Detailed Description

m_profile_config holds all data the motion controller uses to plan paths. When motion profile 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:

- motion_controller.h

4.36 StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage Class Reference

`MaybeMessage` a message of `Message` type or nothing `MaybeMessage m = {};` // empty `MaybeMessage m = Message::EnumField1.`

```
#include <state_machine.h>
```

Public Member Functions

- **MaybeMessage ()**
Empty message - when theres no message.
- **MaybeMessage (Message msg)**
Create a maybemessage with a message.
- **bool has_message ()**
check if the message is here
- **Message message ()**
Get the message stored. The return value is invalid unless has_message returned true.

4.36.1 Detailed Description

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log = false>
class StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage
```

`MaybeMessage` a message of `Message` type or nothing `MaybeMessage m = {};` // empty `MaybeMessage m = Message::EnumField1.`

4.36.2 Constructor & Destructor Documentation

MaybeMessage()

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log =
false>
StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage::MaybeMessage (
    Message msg) [inline]
```

Create a maybemessage with a message.

Parameters

<code>msg</code>	the message to hold on to
------------------	---------------------------

4.36.3 Member Function Documentation

has_message()

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log =  
false>  
bool StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage::has_message ()  
[inline]
```

check if the message is here

Returns

true if there is a message

message()

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log =  
false>  
Message StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage::message ()  
[inline]
```

Get the message stored. The return value is invalid unless has_message returned true.

Returns

The message if it exists. Undefined otherwise

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

- state_machine.h

4.37 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)

4.37.1 Detailed Description

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

4.37.2 Constructor & Destructor Documentation

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

4.37.3 Member Function Documentation

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

<i>inches</i>	How far the robot should drive, in inches
<i>direction</i>	What direction the robot should travel in, in degrees. 0 is forward, +/-180 is reverse, clockwise is positive.
<i>speed</i>	The maximum speed the robot should travel, in percent: -1.0->+1.0
<i>gyro_correction</i>	=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

<i>inches</i>	How far the robot should drive, in inches
<i>direction</i>	What direction the robot should travel in, in degrees. 0 is forward, +/-180 is reverse, clockwise is positive.
<i>speed</i>	The maximum speed the robot should travel, in percent: -1.0->+1.0
<i>gyro_correction</i>	= 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.

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

<i>degrees</i>	How many degrees to rotate the robot. Clockwise positive.
<i>speed</i>	What percentage to run the motors at: 0.0 -> 1.0
<i>ignore_imu</i>	=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

<i>degrees</i>	How many degrees to rotate the robot. Clockwise positive.
<i>speed</i>	What percentage to run the motors at: 0.0 -> 1.0
<i>ignore_imu</i>	= 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

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

<i>left_y</i>	left joystick, Y axis (forward / backwards)
<i>left_x</i>	left joystick, X axis (strafe left / right)
<i>right_x</i>	right joystick, X axis (rotation left / right)
<i>power</i>	=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

<i>left_y</i>	left joystick, Y axis (forward / backwards)
<i>left_x</i>	left joystick, X axis (strafe left / right)
<i>right_x</i>	right joystick, X axis (rotation left / right)
<i>power</i>	= 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_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

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

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

- `mecanum_drive.h`
- `mecanum_drive.cpp`

4.38 MecanumDrive::mecanumdrive_config_t Struct Reference

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

4.38.1 Detailed Description

Configure the Mecanum drive [PID](#) tunings and robot configurations

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

- `mecanum_drive.h`

4.39 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

4.39.1 Detailed Description

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

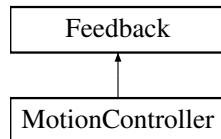
The documentation for this struct was generated from the following file:

- trapezoid_profile.h

4.40 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 () const`

Static Public Member Functions

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

4.40.1 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 = feedforward.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

4.40.2 Constructor & Destructor Documentation

`MotionController()`

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

Construct a new Motion Controller object.

Parameters

<code>config</code>	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
---------------------	--

4.40.3 Member Function Documentation

`get()`

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

Returns

the last saved result from the feedback controller

Implements `Feedback`.

get_motion()

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

Returns

The current position, velocity and acceleration setpoints

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

<i>start_pt</i>	Movement starting position
<i>end_pt</i>	Movement ending position

Implements [Feedback](#).

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](#).

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

<i>lower</i>	upper limit
<i>upper</i>	lower limit

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

Parameters

<i>lower</i>	Upper limit
<i>upper</i>	Lower limit

Implements [Feedback](#).

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

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

Returns

A tuned feedforward object

update()

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

Update the motion profile with a new sensor value.

Parameters

<i>sensor_val</i>	Value from the sensor
-------------------	-----------------------

Returns

the motor input generated from the motion profile

Implements [Feedback](#).

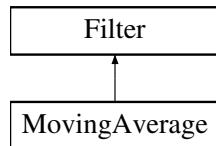
The documentation for this class was generated from the following files:

- motion_controller.h
- motion_controller.cpp

4.41 MovingAverage Class Reference

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

Inheritance diagram for MovingAverage:



Public Member Functions

- [MovingAverage \(int buffer_size\)](#)
- [MovingAverage \(int buffer_size, double starting_value\)](#)
- void [add_entry \(double n\)](#) override
- double [get_value \(\) const](#) override
- int [get_size \(\) const](#)

4.41.1 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).

4.41.2 Constructor & Destructor Documentation

[MovingAverage\(\) \[1/2\]](#)

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

Create a moving average calculator with 0 as the default value

Parameters

<code>buffer_size</code>	The size of the buffer. The number of samples that constitute a valid reading
--------------------------	---

[MovingAverage\(\) \[2/2\]](#)

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

Create a moving average calculator with a specified default value

Parameters

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

4.41.3 Member Function Documentation**add_entry()**

```
void MovingAverage::add_entry (
    double n) [override], [virtual]
```

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

Parameters

<i>n</i>	the sample that will be added to the moving average.
----------	--

Implements [Filter](#).

get_size()

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

How many samples the average is made from

Returns

the number of samples used to calculate this average

get_value()

```
double MovingAverage::get_value () const [override], [virtual]
```

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

Implements [Filter](#).

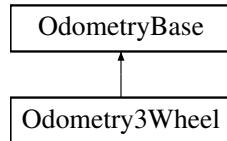
The documentation for this class was generated from the following files:

- moving_average.h
- moving_average.cpp

4.42 Odometry3Wheel Class Reference

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

Inheritance diagram for Odometry3Wheel:



Classes

- struct [odometry3wheel_cfg_t](#)

Public Member Functions

- [Odometry3Wheel \(CustomEncoder &lside_fwd, CustomEncoder &rside_fwd, CustomEncoder &off_axis, odometry3wheel_cfg_t &cfg, bool is_async=true\)](#)
- [Pose2d update \(\) override](#)
- [void tune \(vex::controller &con, TankDrive &drive\)](#)

Public Member Functions inherited from [OdometryBase](#)

- [OdometryBase \(bool is_async\)](#)
- virtual [Pose2d get_position \(\)](#)
- virtual void [set_position \(const Pose2d &newpos=zero_pos\)](#)
- void [end_async \(\)](#)
- virtual double [get_speed \(\)](#)
- virtual 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 [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
- vex::task * [handle](#)
- vex::mutex [mut](#)
- [Pose2d current_pos](#)
- double [speed](#)
- double [accel](#)
- double [ang_speed_deg](#)
- double [ang_accel_deg](#)

4.42.1 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

4.42.2 Constructor & Destructor Documentation

[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

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

4.42.3 Member Function Documentation

[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

<i>con</i>	Controller reference, for screen and button control
<i>drive</i>	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

update()

```
Pose2d 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:

- [odometry_3wheel.h](#)
- [odometry_3wheel.cpp](#)

4.43 Odometry3Wheel::odometry3wheel_cfg_t Struct Reference

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

Public Attributes

- double [wheelbase_dist](#)
- double [off_axis_center_dist](#)
- double [wheel_diam](#)

4.43.1 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

4.43.2 Member Data Documentation

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

wheel_diam

```
double Odometry3Wheel::odometry3wheel_cfg_t::wheel_diam
```

the diameter of the tracking wheel

wheelbase_dist

```
double Odometry3Wheel::odometry3wheel_cfg_t::wheelbase_dist
```

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

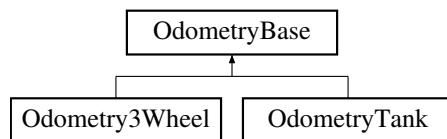
The documentation for this struct was generated from the following file:

- odometry_3wheel.h

4.44 OdometryBase Class Reference

```
#include <odometry_base.h>
```

Inheritance diagram for OdometryBase:

**Public Member Functions**

- [OdometryBase \(bool is_async\)](#)
- virtual [Pose2d get_position \(\)](#)
- virtual void [set_position \(const Pose2d &newpos=zero_pos\)](#)
- virtual [Pose2d update \(\)=0](#)
- void [end_async \(\)](#)
- virtual double [get_speed \(\)](#)
- virtual 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 [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
- vex::task * **handle**
- vex::mutex **mut**
- Pose2d **current_pos**
- double **speed**
- double **accel**
- double **ang_speed_deg**
- double **ang_accel_deg**

4.44.1 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

4.44.2 Constructor & Destructor Documentation

OdometryBase()

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

Construct a new Odometry Base object

Parameters

<code>is_async</code>	True to run constantly in the background, false to call update() manually
-----------------------	---

4.44.3 Member Function Documentation

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

<i>ptr</i>	Pointer to OdometryBase object
------------	--

Returns

Required integer return code. Unused.

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.

get_accel()

```
double OdometryBase::get_accel () [virtual]
```

Get the current acceleration

Returns

the acceleration rate of the robot (inch/s²)

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/s²)

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)

get_position()

```
Pose2d OdometryBase::get_position (
    void ) [virtual]
```

Gets the current position and rotation

Returns

the position that the odometry believes the robot is at

Gets the current position and rotation

get_speed()

```
double OdometryBase::get_speed () [virtual]
```

Get the current speed

Returns

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

set_position()

```
void OdometryBase::set_position (
    const Pose2d & newpos = zero_pos) [virtual]
```

Sets the current position of the robot

Parameters

<i>newpos</i>	the new position that the odometry will believe it is at
---------------	--

Sets the current position of the robot

Reimplemented in [OdometryTank](#).

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

<code>start_deg</code>	initial angle (degrees)
<code>end_deg</code>	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.

update()

```
virtual Pose2d 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](#).

4.44.4 Member Data Documentation

accel

```
double OdometryBase::accel
```

the rate at which we are accelerating (inch/s²)

ang_accel_deg

```
double OdometryBase::ang_accel_deg
```

the rate at which we are accelerating our turn (deg/s²)

ang_speed_deg

```
double OdometryBase::ang_speed_deg
```

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

current_pos

```
Pose2d OdometryBase::current_pos
```

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

handle

```
vex::task* OdometryBase::handle
```

handle to the vex task that is running the odometry code

mut

```
vex::mutex OdometryBase::mut
```

Mutex to control multithreading

speed

```
double OdometryBase::speed
```

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

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

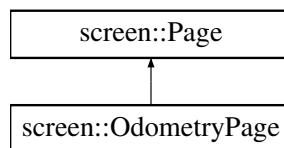
- [odometry_base.h](#)
- [odometry_base.cpp](#)

4.45 screen::OdometryPage Class Reference

a page that shows odometry position and rotation and a map (if an sd card with the file is on)

```
#include <screen.h>
```

Inheritance diagram for screen::OdometryPage:



Public Member Functions

- [OdometryPage \(OdometryBase &odom, double robot_width, double robot_height, bool do_trail\)](#)
Create an odometry trail. Make sure odometry is initialized before now.
- void [update \(bool was_pressed, int x, int y\) override](#)
- void [draw \(vex::brain::lcd &, bool first_draw, unsigned int frame_number\) override](#)

4.45.1 Detailed Description

a page that shows odometry position and rotation and a map (if an sd card with the file is on)

4.45.2 Constructor & Destructor Documentation

OdometryPage()

```
screen::OdometryPage::OdometryPage (
    OdometryBase & odom,
    double robot_width,
    double robot_height,
    bool do_trail)
```

Create an odometry trail. Make sure odometry is initialized before now.

Parameters

<i>odom</i>	the odometry system to monitor
<i>robot_width</i>	the width (side to side) of the robot in inches. Used for visualization
<i>robot_height</i>	the robot_height (front to back) of the robot in inches. Used for visualization
<i>do_trail</i>	whether or not to calculate and draw the trail. Drawing and storing takes a very <i>slight</i> extra amount of processing power

4.45.3 Member Function Documentation

draw()

```
void screen::OdometryPage::draw (
    vex::brain::lcd & scr,
    bool first_draw,
    unsigned int frame_number) [override], [virtual]
```

See also

[Page::draw](#)

Reimplemented from [screen::Page](#).

update()

```
void screen::OdometryPage::update (
    bool was_pressed,
    int x,
    int y) [override], [virtual]
```

See also

[Page::update](#)

Reimplemented from [screen::Page](#).

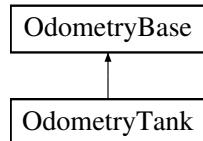
The documentation for this class was generated from the following files:

- [screen.h](#)
- [screen.cpp](#)

4.46 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_custom_enc, [CustomEncoder](#) &right_custom_enc, [robot_specs_t](#) &config, vex::inertial *imu=NULL, bool is_async=true)
- [OdometryTank](#) (vex::encoder &left_vex_enc, vex::encoder &right_vex_enc, [robot_specs_t](#) &config, vex::inertial *imu=NULL, bool is_async=true)
- [Pose2d update](#) () override
- void [set_position](#) (const [Pose2d](#) &newpos=zero_pos) override

Public Member Functions inherited from [OdometryBase](#)

- [OdometryBase](#) (bool is_async)
- virtual [Pose2d get_position](#) (void)
- void [end_async](#) ()
- virtual double [get_speed](#) ()
- virtual 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 [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
- vex::task * [handle](#)
- vex::mutex [mut](#)
- [Pose2d](#) [current_pos](#)
- double [speed](#)
- double [accel](#)
- double [ang_speed_deg](#)
- double [ang_accel_deg](#)

4.46.1 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.

4.46.2 Constructor & Destructor Documentation

OdometryTank() [1/3]

```
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

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

OdometryTank() [2/3]

```
OdometryTank::OdometryTank (
    CustomEncoder & left_custom_enc,
    CustomEncoder & right_custom_enc,
    robot\_specs\_t & config,
    vex::inertial * imu = NULL,
    bool is_async = true)
```

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

Parameters

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

OdometryTank() [3/3]

```
OdometryTank::OdometryTank (
    vex::encoder & left_vex_enc,
    vex::encoder & right_vex_enc,
    robot_specs_t & config,
    vex::inertial * imu = NULL,
    bool is_async = true)
```

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

Parameters

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

4.46.3 Member Function Documentation**set_position()**

```
void OdometryTank::set_position (
    const Pose2d & newpos = zero_pos) [override], [virtual]
```

set_position tells the odometry to place itself at a position

Parameters

<i>newpos</i>	the position the odometry will take
---------------	-------------------------------------

Resets the position and rotational data to the input.

Reimplemented from [OdometryBase](#).

update()

```
Pose2d 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:

- [odometry_tank.h](#)
- [odometry_tank.cpp](#)

4.47 OdomSetPosition Class Reference

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

Public Member Functions

- `OdomSetPosition (OdometryBase &odom, const Pose2d &newpos=OdometryBase::zero_pos)`
- `bool run () override`

4.47.1 Detailed Description

AutoCommand wrapper class for the set_position function in the Odometry class

4.47.2 Constructor & Destructor Documentation

`OdomSetPosition()`

```
OdomSetPosition::OdomSetPosition (
    OdometryBase & odom,
    const Pose2d & newpos = OdometryBase::zero_pos)
```

constructs a new `OdomSetPosition` command

Parameters

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

Construct an Odometry set pos

Parameters

<code>odom</code>	the odometry system we are setting
<code>newpos</code>	the now position to set the odometry to

4.47.3 Member Function Documentation

`run()`

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

Run set_position Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

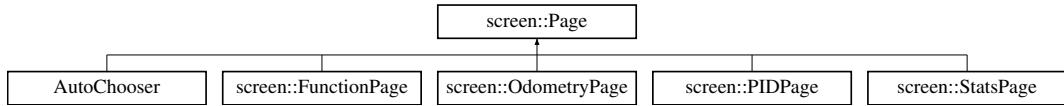
- `drive_commands.h`
- `drive_commands.cpp`

4.48 screen::Page Class Reference

[Page](#) describes one part of the screen slideshow.

```
#include <screen.h>
```

Inheritance diagram for screen::Page:



Public Member Functions

- virtual void [update](#) (bool was_pressed, int x, int y)
collect data, respond to screen input, do fast things (runs at 50hz even if you're not focused on this [Page](#) (only drawn page gets touch updates))
- virtual void [draw](#) (vex::brain::lcd &screen, bool first_draw, unsigned int frame_number)
draw stored data to the screen (runs at 10 hz and only runs if this page is in front)

4.48.1 Detailed Description

[Page](#) describes one part of the screen slideshow.

4.48.2 Member Function Documentation

[draw\(\)](#)

```
virtual void screen::Page::draw (
    vex::brain::lcd & screen,
    bool first_draw,
    unsigned int frame_number) [virtual]
```

draw stored data to the screen (runs at 10 hz and only runs if this page is in front)

Parameters

<i>first_draw</i>	true if we just switched to this page
<i>frame_number</i>	frame of drawing we are on (basically an animation tick)

Reimplemented in [screen::FunctionPage](#), [screen::OdometryPage](#), [screen::PIDPage](#), and [screen::StatsPage](#).

[update\(\)](#)

```
virtual void screen::Page::update (
    bool was_pressed,
    int x,
    int y) [virtual]
```

collect data, respond to screen input, do fast things (runs at 50hz even if you're not focused on this [Page](#) (only drawn page gets touch updates))

Parameters

<code>was_pressed</code>	true if the screen has been pressed
<code>x</code>	x position of screen press (if the screen was pressed)
<code>y</code>	y position of screen press (if the screen was pressed)

Reimplemented in [screen::FunctionPage](#), [screen::OdometryPage](#), [screen::PIDPage](#), and [screen::StatsPage](#).

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

- `screen.h`

4.49 Parallel Class Reference

[Parallel](#) runs multiple commands in parallel and waits for all to finish before continuing. if none finish before this command's timeout, it will call `on_timeout` on all children continue.

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

4.49.1 Detailed Description

[Parallel](#) runs multiple commands in parallel and waits for all to finish before continuing. if none finish before this command's timeout, it will call `on_timeout` on all children continue.

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

- `auto_command.h`
- `auto_command.cpp`

4.50 PurePursuit::Path Class Reference

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

Public Member Functions

- `Path (std::vector< Translation2d > points, double radius)`
- `const std::vector< Translation2d > get_points ()`
- `double get_radius ()`
- `bool is_valid ()`

4.50.1 Detailed Description

Wrapper for a vector of points, checking if any of the points are too close for pure pursuit

4.50.2 Constructor & Destructor Documentation

`Path()`

```
PurePursuit::Path::Path (
    std::vector< Translation2d > points,
    double radius)
```

Create a [Path](#)

Parameters

<i>points</i>	the points that make up the path
<i>radius</i>	the lookahead radius for pure pursuit

4.50.3 Member Function Documentation**get_points()**

```
const std::vector< Translation2d > PurePursuit::Path::get_points ()
```

Get the points associated with this Path

get_radius()

```
double PurePursuit::Path::get_radius ()
```

Get the radius associated with this Path

is_valid()

```
bool PurePursuit::Path::is_valid ()
```

Get whether this path will behave as expected

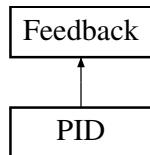
The documentation for this class was generated from the following files:

- pure_pursuit.h
- pure_pursuit.cpp

4.51 PID Class Reference

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

Inheritance diagram for PID:

**Classes**

- struct [pid_config_t](#)

Public Types

- enum [ERROR_TYPE](#)

Public Member Functions

- [PID \(pid_config_t &config\)](#)
- void [init \(double start_pt, double set_pt\)](#) override
- double [update \(double sensor_val\)](#) override
- double [update \(double sensor_val, double v_setpt\)](#)
- double [get_sensor_val \(\) const](#)
gets the sensor value that we were last updated with
- double [get \(\) override](#)
- void [set_limits \(double lower, double upper\)](#) override
- bool [is_on_target \(\) override](#)
- void [reset \(\)](#)
- double [get_error \(\)](#)
- double [get_target \(\) const](#)
- void [set_target \(double target\)](#)

Public Attributes

- [pid_config_t & config](#)

4.51.1 Detailed Description

PID Class

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

$\text{out} = \text{kP} * \text{error} + \text{kI} * \text{integral}(\text{d Error}) + \text{kD} * (\text{dError}/\text{dt})$

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

4.51.2 Member Enumeration Documentation

[ERROR_TYPE](#)

```
enum PID::ERROR\_TYPE
```

An enum to distinguish between a linear and angular calculation of [PID](#) error.

4.51.3 Constructor & Destructor Documentation

[PID\(\)](#)

```
PID::PID (  
    pid\_config\_t & config)
```

Create the [PID](#) object

Parameters

<i>config</i>	the configuration data for this controller
---------------	--

Create the [PID](#) object

4.51.4 Member Function Documentation

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](#).

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

get_sensor_val()

```
double PID::get_sensor_val () const
```

gets the sensor value that we were last updated with

Returns

`sensor_val`

get_target()

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

Get the [PID](#)'s target

Returns

the target the [PID](#) controller is trying to achieve

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

<i>start_pt</i>	completely ignored for PID . necessary to satisfy Feedback base
<i>set_pt</i>	sets the target of the PID controller
<i>start_vel</i>	completely ignored for PID . necessary to satisfy Feedback base
<i>end_vel</i>	sets the target end velocity of the PID controller

Implements [Feedback](#).

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](#).

reset()

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

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

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

<i>lower</i>	the lower limit. the PID controller will never command the output go below <i>lower</i>
<i>upper</i>	the upper limit. the PID controller will never command the output go higher than <i>upper</i>

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

Implements [Feedback](#).

set_target()

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

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

Parameters

<i>target</i>	the sensor reading we would like to achieve
---------------	---

Set the target for the [PID](#) loop, where the robot is trying to end up

update() [1/2]

```
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

<i>sensor_val</i>	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](#).

update() [2/2]

```
double PID::update (
    double sensor_val,
    double v_setpt)
```

Update the [PID](#) loop by taking the time difference from last update, and running the [PID](#) formula with the new sensor data

Parameters

<i>sensor_val</i>	the distance, angle, encoder position or whatever it is we are measuring
<i>v_setpt</i>	Expected velocity setpoint, to subtract from the D term (for velocity control)

Returns

the new output. What would be returned by [PID::get\(\)](#)

4.51.5 Member Data Documentation**config**

`pid_config_t& PID::config`

configuration struct for this controller. see [pid_config_t](#) for information about what this contains

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

- pid.h
- pid.cpp

4.52 PID::pid_config_t Struct Reference

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

Public Attributes

- double **p**
*proportional coefficient p * error()*
- double **i**
*integral coefficient i * integral(error)*
- double **d**
*derivative coefficient d * derivative(error)*
- double **deadband**
at what threshold are we close enough to be finished
- double **on_target_time**
- [ERROR_TYPE](#) **error_method**

4.52.1 Detailed Description

`pid_config_t` holds the configuration parameters for a pid controller In addition 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

4.52.2 Member Data Documentation

error_method

[ERROR_TYPE](#) `PID::pid_config_t::error_method`

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

on_target_time

`double PID::pid_config_t::on_target_time`

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

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

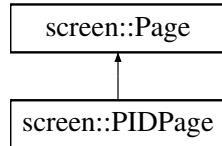
- `pid.h`

4.53 screen::PIDPage Class Reference

[PIDPage](#) provides a way to tune a pid controller on the screen.

```
#include <screen.h>
```

Inheritance diagram for screen::PIDPage:



Public Member Functions

- [PIDPage](#) (`PID &pid, std::string name, std::function< void(void)> onchange=[]() {}`)
Create a [PIDPage](#).
- void [update](#) (bool was_pressed, int x, int y) override
- void [draw](#) (vex::brain::lcd &, bool first_draw, unsigned int frame_number) override

4.53.1 Detailed Description

[PIDPage](#) provides a way to tune a pid controller on the screen.

4.53.2 Constructor & Destructor Documentation

[PIDPage\(\)](#)

```
screen::PIDPage::PIDPage (
    PID & pid,
    std::string name,
    std::function< void(void)> onchange = [ ]() {})
```

Create a [PIDPage](#).

Parameters

<code>pid</code>	the pid controller we're changing
<code>name</code>	a name to recognize this pid controller if we've got multiple pid screens
<code>onchange</code>	a function that is called when a tuning parameter is changed. If you need to update stuff on that change register a handler here

4.53.3 Member Function Documentation

draw()

```
void screen::PIDPage::draw (
    vex::brain::lcd & scr,
    bool first_draw,
    unsigned int frame_number) [override], [virtual]
```

See also

[Page::draw](#)

Reimplemented from [screen::Page](#).

update()

```
void screen::PIDPage::update (
    bool was_pressed,
    int x,
    int y) [override], [virtual]
```

See also

[Page::update](#)

Reimplemented from [screen::Page](#).

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

- [screen.h](#)
- [screen.cpp](#)

4.54 Pose2d Class Reference

```
#include <pose2d.h>
```

Public Member Functions

- `constexpr Pose2d ()`
- `Pose2d (const Translation2d &translation, const Rotation2d &rotation)`
- `Pose2d (const double &x, const double &y, const Rotation2d &rotation)`
- `Pose2d (const double &x, const double &y, const double &radians)`
- `Pose2d (const Translation2d &translation, const double &radians)`
- `Pose2d (const Eigen::Vector3d &pose_vector)`
- `Translation2d translation () const`
- `double x () const`
- `void setX (double x)`
- `double y () const`
- `void setY (double y)`
- `Rotation2d rotation () const`
- `void setRotationRad (double rotRad)`
- `void setRotationDeg (double rotDeg)`
- `bool operator== (const Pose2d other) const`
- `Pose2d operator* (const double &scalar) const`
- `Pose2d operator/ (const double &scalar) const`
- `Pose2d operator+ (const Transform2d &transform) const`
- `Transform2d operator- (const Pose2d &other) const`
- `Pose2d relative_to (const Pose2d &other) const`
- `Pose2d transform_by (const Transform2d &transform) const`
- `Pose2d exp (const Twist2d &twist) const`
- `Twist2d log (const Pose2d &end_pose) const`

Friends

- `std::ostream & operator<< (std::ostream &os, const Pose2d &pose)`

4.54.1 Detailed Description

Class representing a pose in 2d space with x, y, and rotational components

Assumes conventional cartesian coordinate system: Looking down at the coordinate plane, +X is right +Y is up
+Theta is counterclockwise

4.54.2 Constructor & Destructor Documentation

Pose2d() [1/6]

```
Pose2d::Pose2d () [inline], [constexpr]
```

Default Constructor for `Pose2d`

Pose2d() [2/6]

```
Pose2d::Pose2d (
    const Translation2d & translation,
    const Rotation2d & rotation)
```

Constructs a pose with given translation and rotation components.

Parameters

<i>translation</i>	translational component.
<i>rotation</i>	rotational component.

Pose2d() [3/6]

```
Pose2d::Pose2d (
    const double & x,
    const double & y,
    const Rotation2d & rotation)
```

Constructs a pose with given translation and rotation components.

Parameters

<i>x</i>	x component.
<i>y</i>	y component.
<i>rotation</i>	rotational component.

Pose2d() [4/6]

```
Pose2d::Pose2d (
    const double & x,
    const double & y,
    const double & radians)
```

Constructs a pose with given translation and rotation components.

Parameters

<i>x</i>	x component.
<i>y</i>	y component.
<i>radians</i>	rotational component in radians.

Pose2d() [5/6]

```
Pose2d::Pose2d (
    const Translation2d & translation,
    const double & radians)
```

Constructs a pose with given translation and rotation components.

Parameters

<i>translation</i>	translational component.
<i>radians</i>	rotational component in radians.

Pose2d() [6/6]

```
Pose2d::Pose2d (
    const Eigen::Vector3d & pose_vector)
```

Constructs a pose with given translation and rotation components.

Parameters

<i>pose_vector</i>	vector of the form [x, y, theta].
--------------------	-----------------------------------

4.54.3 Member Function Documentation**exp()**

```
Pose2d Pose2d::exp (
    const Twist2d & twist) const
```

Applies a twist (pose delta) to a pose by including first order dynamics of heading.

When applying a twist, imagine a constant angular velocity, the translational components must be rotated into the global frame at every point along the twist, simply adding the deltas does not do this, and using euler integration results in some error. This is the analytic solution to that problem.

Can also be thought of more simply as applying a twist as following an arc rather than a straight line.

See this document for more information on the pose exponential and its derivation. <https://file.c-tavsys.net/control/controls-engineering-in-frc.pdf#section.10.2>

Parameters

<i>old_pose</i>	The pose to which the twist will be applied.
<i>twist</i>	The twist, represents a pose delta.

Returns

new pose that has been moved forward according to the twist.

log()

```
Twist2d Pose2d::log (
    const Pose2d & end_pose) const
```

The inverse of the pose exponential.

Determines the twist required to go from this pose to the given end pose. suppose you have `Pose2d a, Twist2d twist` if `a.exp(twist) = b` then `a.log(b) = twist`

Parameters

<i>end_pose</i>	the end pose to find the mapping to.
-----------------	--------------------------------------

Returns

the twist required to go from this pose to the given end

operator*()

```
Pose2d Pose2d::operator* (
    const double & scalar) const
```

Multiplies this pose by a scalar. Simply multiplies each component.

Parameters

<i>scalar</i>	the scalar value to multiply by.
---------------	----------------------------------

operator+()

```
Pose2d Pose2d::operator+ (
    const Transform2d & transform) const
```

Adds a transform to this pose. Transforms the pose in the pose's frame.

Parameters

<i>transform</i>	the change in pose.
------------------	---------------------

operator-()

```
Transform2d Pose2d::operator- (
    const Pose2d & other) const
```

Subtracts one pose from another to find the transform between them.

Parameters

<i>other</i>	the pose to subtract.
--------------	-----------------------

operator/()

```
Pose2d Pose2d::operator/ (
    const double & scalar) const
```

Divides this pose by a scalar. Simply divides each component.

Parameters

<i>scalar</i>	the scalar value to divide by.
---------------	--------------------------------

operator==()

```
bool Pose2d::operator== (
    const Pose2d other) const
```

Compares this to another pose.

Parameters

<i>other</i>	the other pose to compare to.
--------------	-------------------------------

Returns

true if each of the components are within 1e-9 of each other.

relative_to()

```
Pose2d Pose2d::relative_to (
    const Pose2d & other) const
```

Finds the pose equivalent to this pose relative to another arbitrary pose rather than the origin.

Parameters

<i>other</i>	the pose representing the new origin.
--------------	---------------------------------------

Returns

this pose relative to another pose.

rotation()

```
Rotation2d Pose2d::rotation () const
```

Returns the rotational component.

Returns

the rotational component.

setRotationDeg()

```
void Pose2d::setRotationDeg (
    double rotDeg)
```

sets the ration value of the rotational component in Degrees

setRotationRad()

```
void Pose2d::setRotationRad (
    double rotRad)
```

sets the ration value of the rotational component in Radians

setX()

```
void Pose2d::setX (
    double x)
```

sets the x value of the translational component.

setY()

```
void Pose2d::setY (
    double y)
```

sets the y value of the translational component.

transform_by()

```
Pose2d Pose2d::transform_by (
    const Transform2d & transform) const
```

Adds a transform to this pose. Simply adds each component.

Parameters

<i>transform</i>	the change in pose.
------------------	---------------------

Returns

the pose after being transformed.

translation()

```
Translation2d Pose2d::translation () const
```

Returns the translational component.

Returns

the translational component.

x()

```
double Pose2d::x () const
```

Returns the x value of the translational component.

Returns

the x value of the translational component.

y()

```
double Pose2d::y () const
```

Returns the y value of the translational component.

Returns

the y value of the translational component.

4.54.4 Friends And Related Symbol Documentation**operator<<**

```
std::ostream & operator<< (
    std::ostream & os,
    const Pose2d & pose) [friend]
```

Sends a pose to an output stream. Ex. std::cout << pose;

prints "Pose2d[x: (value), y: (value), rad: (radians), deg: (degrees)]"

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

- pose2d.h
- pose2d.cpp

4.55 PurePursuitCommand Class Reference

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

Public Member Functions

- [PurePursuitCommand \(TankDrive &drive_sys, Feedback &feedback, PurePursuit::Path path, directionType dir, double max_speed=1, double end_speed=0\)](#)
- [bool run \(\) override](#)
- [void on_timeout \(\) override](#)

4.55.1 Detailed Description

Autocommand wrapper class for pure pursuit function in the [TankDrive](#) class

4.55.2 Constructor & Destructor Documentation**PurePursuitCommand()**

```
PurePursuitCommand::PurePursuitCommand (
    TankDrive & drive_sys,
    Feedback & feedback,
    PurePursuit::Path path,
    directionType dir,
    double max_speed = 1,
    double end_speed = 0)
```

Construct a Pure Pursuit AutoCommand

Parameters

<i>path</i>	The list of coordinates to follow, in order
<i>dir</i>	Run the bot forwards or backwards
<i>feedback</i>	The feedback controller determining speed
<i>max_speed</i>	Limit the speed of the robot (for pid / pidff feedbacks)

4.55.3 Member Function Documentation**on_timeout()**

```
void PurePursuitCommand::on_timeout () [override]
```

Reset the drive system when it times out

run()

```
bool PurePursuitCommand::run () [override]
```

Direct call to [TankDrive::pure_pursuit](#)

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

- `drive_commands.h`
- `drive_commands.cpp`

4.56 Rect Struct Reference

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

4.56.1 Detailed Description

Describes a Rectangle with a minimum and maximum point

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

- `geometry.h`

4.57 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**
- 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

4.57.1 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.

4.57.2 Member Data Documentation

drive_correction_cutoff

```
double robot_specs_t::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

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

- robot_specs.h

4.58 Rotation2d Class Reference

```
#include <rotation2d.h>
```

Public Member Functions

- `constexpr Rotation2d ()`
- `Rotation2d (const double &radians)`
- `Rotation2d (const double &x, const double &y)`
- `Rotation2d (const Translation2d &translation)`
- `double radians () const`
- `void setRad (double radRot)`
- `double degrees () const`
- `void setDeg (double degRot)`
- `double revolutions () const`
- `double f_cos () const`
- `double f_sin () const`
- `double f_tan () const`
- `Eigen::Matrix2d rotation_matrix () const`
- `double wrapped_radians_180 () const`
- `double wrapped_degrees_180 () const`
- `double wrapped_revolutions_180 () const`
- `double wrapped_radians_360 () const`
- `double wrapped_degrees_360 () const`
- `double wrapped_revolutions_360 () const`
- `Rotation2d operator+ (const Rotation2d &other) const`
- `Rotation2d operator- (const Rotation2d &other) const`
- `Rotation2d operator* (const double &scalar) const`
- `Rotation2d operator/ (const double &scalar) const`
- `bool operator== (const Rotation2d &other) const`

Friends

- `std::ostream & operator<< (std::ostream &os, const Rotation2d &rotation)`

4.58.1 Detailed Description

Class representing a rotation in 2d space. Stores theta in radians, as well as cos and sin.

Internally this angle is stored continuously, however there are functions that return wrapped angles: "180" is from [-pi, pi], [-180, 180), [-0.5, 0.5] "360" is from [0, 2pi], [0, 360), [0, 1)

4.58.2 Constructor & Destructor Documentation

Rotation2d() [1/4]

```
Rotation2d::Rotation2d () [inline], [constexpr]
```

Default Constructor for `Rotation2d`

Rotation2d() [2/4]

```
Rotation2d::Rotation2d (
    const double & radians)
```

Constructs a rotation with the given value in radians.

Parameters

<i>radians</i>	the value of the rotation in radians.
----------------	---------------------------------------

Rotation2d() [3/4]

```
Rotation2d::Rotation2d (
    const double & x,
    const double & y)
```

Constructs a rotation given x and y values. Does not have to be normalized. The angle from the x axis to the point.

$$[\theta] = [\text{atan2}(y, x)]$$

Parameters

<i>x</i>	the x value of the point
<i>y</i>	the y value of the point

Rotation2d() [4/4]

```
Rotation2d::Rotation2d (
    const Translation2d & translation)
```

Constructs a rotation given x and y values in the form of a [Translation2d](#). Does not have to be normalized. The angle from the x axis to the point.

$$[\theta] = [\text{atan2}(y, x)]$$

Parameters

<i>translation</i>	
--------------------	--

4.58.3 Member Function Documentation**degrees()**

```
double Rotation2d::degrees () const
```

Returns the degree angle value.

Returns

the degree angle value.

f_cos()

```
double Rotation2d::f_cos () const
```

Returns the cosine of the angle value.

Returns

the cosine of the angle value

f_sin()

```
double Rotation2d::f_sin () const
```

Returns the sine of the angle value.

Returns

the sine of the angle value.

f_tan()

```
double Rotation2d::f_tan () const
```

Returns the tangent of the angle value.

Returns

the tangent of the angle value.

operator*()

```
Rotation2d Rotation2d::operator* (
    const double & scalar) const
```

Multiplies this rotation by a scalar.

Parameters

<code>scalar</code>	the scalar value to multiply the rotation by.
---------------------	---

Returns

the rotation multiplied by the scalar.

operator+()

```
Rotation2d Rotation2d::operator+ (
    const Rotation2d & other) const
```

Adds the values of two rotations using a rotation matrix

[new_cos] = [other.cos, -other.sin][cos] [new_sin] = [other.sin, other.cos][sin] new_value = atan2(new_sin, new_cos)

Parameters

<i>other</i>	the other rotation to add to this rotation.
--------------	---

Returns

the sum of the two rotations.

Adds the values of two rotations using a rotation matrix.

[new_cos] = [other.cos, -other.sin][cos] [new_sin] = [other.sin, other.cos][sin] new_value = atan2(new_sin, new_cos)

Parameters

<i>other</i>	the other rotation to add to this rotation.
--------------	---

Returns

the sum of the two rotations.

operator-() [1/2]

`Rotation2d` `Rotation2d::operator- () const`

Takes the inverse of this rotation by flipping it. Equivalent to adding 180 degrees.

Returns

this inverse of the rotation.

Takes the inverse of this rotation by flipping it.

Returns

this inverse of the rotation.

operator-() [2/2]

`Rotation2d` `Rotation2d::operator- (`
`const Rotation2d & other) const`

Subtracts the values of two rotations.

Parameters

<i>other</i>	the other rotation to subtract from this rotation.
--------------	--

Returns

the difference between the two rotations.

operator/()

`Rotation2d` `Rotation2d::operator/ (`
`const double & scalar) const`

Divides this rotation by a scalar.

Parameters

<i>scalar</i>	the scalar value to divide the rotation by.
---------------	---

Returns

the rotation divided by the scalar.

operator==()

```
bool Rotation2d::operator== (
    const Rotation2d & other) const
```

Compares two rotations. Returns true if their values are within 1e-9 radians of each other, to account for floating point error.

Parameters

<i>other</i>	the other rotation to compare to
--------------	----------------------------------

Returns

whether the values of the rotations are within 1e-9 radians of each other

radians()

```
double Rotation2d::radians () const
```

Returns the radian angle value.

Returns

the radian angle value.

revolutions()

```
double Rotation2d::revolutions () const
```

Returns the revolution angle value.

Returns

the revolution angle value.

rotation_matrix()

```
Eigen::Matrix2d Rotation2d::rotation_matrix () const
```

Returns the rotation matrix equivalent to this rotation [cos, -sin] R = [sin, cos]

Returns

the rotation matrix equivalent to this rotation

setDeg()

```
void Rotation2d::setDeg (
    double degRot)
```

sets the angle value in degrees

setRad()

```
void Rotation2d::setRad (
    double radRot)
```

sets the angle value in radians

wrapped_degrees_180()

```
double Rotation2d::wrapped_degrees_180 () const
```

Returns the degree angle value, wrapped from [-180, 180].

Returns

the degree angle value, wrapped from [-180, 180]

wrapped_degrees_360()

```
double Rotation2d::wrapped_degrees_360 () const
```

Returns the degree angle value, wrapped from [0, 360].

Returns

the degree angle value, wrapped from [0, 360]

wrapped_radians_180()

```
double Rotation2d::wrapped_radians_180 () const
```

Returns the radian angle value, wrapped from [-pi, pi).

Returns

the radian angle value, wrapped from [-pi, pi)

wrapped_radians_360()

```
double Rotation2d::wrapped_radians_360 () const
```

Returns the radian angle value, wrapped from [0, 2pi).

Returns

the radian angle value, wrapped from [0, 2pi)

wrapped_revolutions_180()

```
double Rotation2d::wrapped_revolutions_180 () const
```

Returns the revolution angle value, wrapped from [-0.5, 0.5).

Returns

the revolution angle value, wrapped from [-0.5, 0.5)

wrapped_revolutions_360()

```
double Rotation2d::wrapped_revolutions_360 () const
```

Returns the revolution angle value, wrapped from [0, 1).

Returns

the revolution angle value, wrapped from [0, 1)

4.58.4 Friends And Related Symbol Documentation

operator<<

```
std::ostream & operator<< (
    std::ostream & os,
    const Rotation2d & rotation) [friend]
```

Sends a rotation to an output stream. Ex. std::cout << rotation;

prints "Rotation2d[rad: (radians), deg: (degrees)]"

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

- rotation2d.h
- rotation2d.cpp

4.59 ScaledSphericalSimplexSigmaPoints< STATES > Class Template Reference

```
#include <srukf.h>
```

Public Member Functions

- `ScaledSphericalSimplexSigmaPoints` (double alpha=0.001, double beta=2)
- int `num_sigmas` ()
- EMat< STATES, NUM_SIGMAS > `square_root_sigma_points` (const EVec< STATES > &x, const EMat< STATES, STATES > &S)
- const EVec< NUM_SIGMAS > & `Wm` () const
- const EVec< NUM_SIGMAS > & `Wc` () const
- double `Wm` (int i) const
- double `Wc` (int i) const

4.59.1 Detailed Description

```
template<int STATES>
class ScaledSphericalSimplexSigmaPoints< STATES >
```

Generates sigma points and weights according to the paper [1] This is very different from Wan and Merwe's formulation.

This only requires $N + 2$ sigma points instead of $2N + 1$ sigma points. Rather than generating sigma points symmetrically around the mean, it generates them as vertices of an N -simplex.

The performance of the filter using this reduced set of sigma points is identical to the standard method, so there is no downside to using it here.

[1] A Scaled Spherical Simplex [Filter](#) (S3F) with a decreased $n + 2$ sigma points set size and equivalent $2n + 1$ Unscented Kalman [Filter](#) (UKF) accuracy

Template Parameters

<code>STATES</code>	the dimension of the state. NUM_SIGMAS sigma points and weights will be generated.
---------------------	--

4.59.2 Constructor & Destructor Documentation

`ScaledSphericalSimplexSigmaPoints()`

```
template<int STATES>
ScaledSphericalSimplexSigmaPoints< STATES >::ScaledSphericalSimplexSigmaPoints (
    double alpha = 0.001,
    double beta = 2) [inline]
```

Constructs a sigma point generator for Spherical Simplex sigma points

Parameters

<i>alpha</i>	Determines the spread of the sigma points around the mean. Smaller values are closer to the mean, this is usually a small value.
<i>beta</i>	Incorporates prior knowledge of the distribution of the state. For Gaussian distributions, beta = 2 is optimal.

4.59.3 Member Function Documentation**num_sigmas()**

```
template<int STATES>
int ScaledSphericalSimplexSigmaPoints< STATES >::num_sigmas () [inline]
```

Returns the number of sigma points, for simplex sigma points this is N+2.

square_root_sigma_points()

```
template<int STATES>
EMat< STATES, NUM_SIGMAS > ScaledSphericalSimplexSigmaPoints< STATES >::square_root_sigma_points (
    const EVec< STATES > & x,
    const EMat< STATES, STATES > & S) [inline]
```

Computes the sigma points given a mean (x) and square-root covariance (S).

Parameters

<i>x</i>	Vector of the means.
<i>S</i>	Square-root covariance.

Returns

Matrix containing the sigma points. Each column contains one sigma point in the same space as x. The first column is the same as the mean, with the others arranged around the mean.

Wc() [1/2]

```
template<int STATES>
const EVec< NUM_SIGMAS > & ScaledSphericalSimplexSigmaPoints< STATES >::Wc () const [inline]
```

Returns a vector containing the weights of each sigma point for the covariance.

Wc() [2/2]

```
template<int STATES>
double ScaledSphericalSimplexSigmaPoints< STATES >::Wc (
    int i) const [inline]
```

Returns the weight for the i-th sigma point for the covariance.

Parameters

<i>i</i>	Element of the weights vector to return.
----------	--

Wm() [1/2]

```
template<int STATES>
const EVec< NUM_SIGMAS > & ScaledSphericalSimplexSigmaPoints< STATES >::Wm () const [inline]
```

Returns a vector containing the weights of each sigma point for the mean.

Wm() [2/2]

```
template<int STATES>
double ScaledSphericalSimplexSigmaPoints< STATES >::Wm (
    int i) const [inline]
```

Returns the weight for the *i*-th sigma point for the mean.

Parameters

<i>i</i>	Element of the weights vector to return.
----------	--

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

- srukf.h

4.60 screen::ScreenData Struct Reference

The [ScreenData](#) class holds the data that will be passed to the screen thread you probably shouldnt have to use it.

4.60.1 Detailed Description

The [ScreenData](#) class holds the data that will be passed to the screen thread you probably shouldnt have to use it.

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

- screen.cpp

4.61 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 doesn't 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

4.61.1 Detailed Description

Serializes Arbitrary data to a file on the SD Card.

4.61.2 Constructor & Destructor Documentation

Serializer()

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

create a `Serializer`

Parameters

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

4.61.3 Member Function Documentation

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

<i>name</i>	name of value
<i>otherwise</i>	value if the name is not specified

Returns

the value if found or otherwise

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

<i>name</i>	name of value
<i>otherwise</i>	value if the name is not specified

Returns

the value if found or otherwise

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

<i>name</i>	name of value
<i>otherwise</i>	value if the name is not specified

Returns

the value if found or otherwise

save_to_disk()

```
void Serializer::save_to_disk () const  
saves current Serializer state to disk  
forms data bytes then saves to filename this was opened with
```

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

<i>name</i>	name of bool
<i>b</i>	value of bool

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

<i>name</i>	name of double
<i>d</i>	value of double

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

<i>name</i>	name of integer
<i>i</i>	value of integer

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

<i>name</i>	name of string
<i>i</i>	value of string

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

<i>name</i>	name of value
<i>otherwise</i>	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:

- `serializer.h`
- `serializer.cpp`

4.62 screen::SliderWidget Class Reference

Widget that updates a double value. Updates by reference so watch out for race conditions cuz the screen stuff lives on another thread.

```
#include <screen.h>
```

Public Member Functions

- **SliderWidget** (double &val, double low, double high, `Rect` rect, std::string name)
Creates a slider widget.
- bool **update** (bool was_pressed, int x, int y)
responds to user input
- void **draw** (vex::brain::lcd &, bool first_draw, unsigned int frame_number)
Page::draws the slide to the screen

4.62.1 Detailed Description

Widget that updates a double value. Updates by reference so watch out for race conditions cuz the screen stuff lives on another thread.

4.62.2 Constructor & Destructor Documentation

SliderWidget()

```
screen::SliderWidget::SliderWidget (
    double & val,
    double low,
    double high,
    Rect rect,
    std::string name) [inline]
```

Creates a slider widget.

Parameters

<i>val</i>	reference to the value to modify
<i>low</i>	minimum value to go to
<i>high</i>	maximum value to go to
<i>rect</i>	rect to draw it
<i>name</i>	name of the value

4.62.3 Member Function Documentation

update()

```
bool screen::SliderWidget::update (
    bool was_pressed,
    int x,
    int y)
```

responds to user input

Parameters

<i>was_pressed</i>	if the screen is pressed
<i>x</i>	x position if the screen was pressed
<i>y</i>	y position if the screen was pressed

Returns

true if the value updated

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

- screen.h
- screen.cpp

4.63 SpinRPMCommand Class Reference

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

Public Member Functions

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

4.63.1 Detailed Description

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

4.63.2 Constructor & Destructor Documentation

`SpinRPMCommand()`

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

Construct a SpinRPM Command

Parameters

<code>flywheel</code>	the flywheel sys to command
<code>rpm</code>	the rpm that we should spin at

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

4.63.3 Member Function Documentation

`run()`

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

Run spin_manual Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

- `flywheel_commands.h`
- `flywheel_commands.cpp`

4.64 PurePursuit::spline Struct Reference

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

4.64.1 Detailed Description

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

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

- `pure_pursuit.h`

4.65 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS > Class Template Reference

```
#include <srukf.h>
```

Public Member Functions

- `SquareRootUnscentedKalmanFilter (const std::function< StateVector(const StateVector &, const InputVector &) > &f, const std::function< OutputVector(const StateVector &, const InputVector &) > &h, const WithInput<- Integrator &integrator, const StateVector &state_stddevs, const OutputVector &measurement_stddevs)`
- `SquareRootUnscentedKalmanFilter (const std::function< StateVector(const StateVector &, const InputVector &) > &f, const std::function< OutputVector(const StateVector &, const InputVector &) > &h, const With<- InputIntegrator &integrator, const StateVector &state_stddevs, const OutputVector &measurement_stddevs, const std::function< StateVector(const EMat< STATES, NUM_SIGMAS > &, const EVec< NUM_SIGMAS > &) > &mean_func_X, const std::function< OutputVector(const EMat< OUTPUTS, NUM_SIGMAS > &, const EVec< NUM_SIGMAS > &) > &mean_func_Y, const std::function< StateVector(const StateVector &, const StateVector &) > &residual_func_X, const std::function< OutputVector(const OutputVector &, const Output<- Vector &) > &residual_func_Y, const std::function< StateVector(const StateVector &, const StateVector &) > &add_func_X)`
- `StateMatrix S () const`
- `double S (int i, int j) const`
- `void set_S (const StateMatrix &S)`
- `StateMatrix P () const`
- `void set_P (const StateMatrix &P)`
- `StateVector xhat () const`
- `double xhat (int i) const`
- `void set_xhat (const StateVector &xhat)`
- `void set_xhat (int i, double value)`
- `void reset ()`
- `void predict (const InputVector &u, double dt)`
- `void correct (const InputVector &u, const OutputVector &y)`
- `void correct (const InputVector &u, const OutputVector &y, const EVec< OUTPUTS > &measurement<- stddevs)`
- `template<int ROWS> void correct (const InputVector &u, const EVec< ROWS > &y, const std::function< EVec< ROWS >(const StateVector &, const InputVector &) > &h, const EVec< ROWS > &measurement_stddevs)`
- `template<int ROWS> void correct (const InputVector &u, const EVec< ROWS > &y, const std::function< EVec< ROWS >(const StateVector &, const InputVector &) > &h, const EVec< ROWS > measurement_stddevs, const std::function< EVec< ROWS >(const EMat< ROWS, NUM_SIGMAS > &, const EVec< NUM_SIGMAS > &) > &mean_func_Y, const std::function< EVec< ROWS >(const EVec< ROWS > &, const EVec< ROWS > &) > &residual_func_Y, const std::function< StateVector(const StateVector &, const StateVector &) > &residual_func_X, const std::function< StateVector(const StateVector &, const StateVector &) > &add<- func_X)`

4.65.1 Detailed Description

```
template<int STATES, int INPUTS, int OUTPUTS>
class SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >
```

Kalman filters combine predictions from a model and measurements to estimate a system's true state.

The Unscented Kalman Filter is a nonlinear estimator, meaning that the model used to predict how the state changes over time can be nonlinear. The model that determines the expected measurement given the current state can also be nonlinear.

At each timestep, sigma points are generated close to the mean, they are all propagated forward in time according to the nonlinear model. The Unscented Transform uses the propagated sigma points to compute the prior state and covariance.

When correcting the state and covariance with a measurement, sigma points are again generated, but are transformed into the measurement space using the measurement function. A Kalman gain matrix K is then computed, and used to update the state and covariance.

To read more about Kalman filters and the standard UKF read: <https://github.com/rablabbe/Kalman-and-Bayesian-Filters-in-Python>

This implementation is somewhat non-standard. The square-root form of the UKF (SR-UKF) is used, and the way the sigma points are generated is different than most implementations. The square-root form is used to ensure that the covariance matrix remains positive definite.

To learn more about the SR-UKF, and see the exact formulation that most of this implementation follows, read: <https://www.researchgate.net/publication/3908304>

The sigma points are not generated symmetrically around the mean, instead they are generated as vertices of a simplex. Using N = # of states, this method uses N + 2 sigma points instead of the standard 2N + 1 sigma points. This reduces computation up to 50%. To learn more about this method, read: <https://www.sciencedirect.com/science/article/pii/S0888327020308190>

This filter uses a method of "recalibrating" by essentially applying a measurement twice instead of once, and only using it if it is more accurate than before the measurement was applied. To learn more about this framework for nonlinear filters, read: <https://arxiv.org/pdf/2407.05717.pdf>

Template Parameters

<i>STATES</i>	Dimension of the state vector.
<i>INPUTS</i>	Dimension of the control input vector.
<i>OUTPUTS</i>	Dimension of the measurement vector.

4.65.2 Constructor & Destructor Documentation

SquareRootUnscentedKalmanFilter() [1/2]

```
template<int STATES, int INPUTS, int OUTPUTS>
SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::SquareRootUnscentedKalmanFilter (
    const std::function< StateVector(const StateVector &, const InputVector &) > & f,
    const std::function< OutputVector(const StateVector &, const InputVector &) > & h,
    const WithInputIntegrator & integrator,
    const StateVector & state_stddevs,
    const OutputVector & measurement_stddevs) [inline]
```

Constructs an Unscented Kalman Filter.

Parameters

<i>f</i>	A vector valued function of x and u that returns the derivative of the state vector with respect to time.
<i>h</i>	A vector valued function of x and u that returns the expected measurement at the given state.
<i>integrator</i>	A function from "numerical_integration.h" that integrates a differential equation of the form $f(x, u)$.
<i>state_stddevs</i>	Standard deviations of the states in the model.
<i>measurement_stddevs</i>	Standard deviations of the measurements.

SquareRootUnscentedKalmanFilter() [2/2]

```
template<int STATES, int INPUTS, int OUTPUTS>
SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::SquareRootUnscentedKalmanFilter (
    const std::function< StateVector(const StateVector &, const InputVector &) > & f,
    const std::function< OutputVector(const StateVector &, const InputVector &) > & h,
    const WithInputIntegrator & integrator,
    const StateVector & state_stddevs,
    const OutputVector & measurement_stddevs,
    const std::function< StateVector(const EMat< STATES, NUM_SIGMAS > &, const EVec< NUM_SIGMAS > &) > & mean_func_X,
    const std::function< OutputVector(const EMat< OUTPUTS, NUM_SIGMAS > &, const EVec< NUM_SIGMAS > &) > & mean_func_Y,
    const std::function< StateVector(const StateVector &, const StateVector &) > & residual_func_X,
    const std::function< OutputVector(const OutputVector &, const OutputVector &) > & residual_func_Y,
    const std::function< StateVector(const StateVector &, const StateVector &) > & add_func_X) [inline]
```

Constructs an Unscented Kalman [Filter](#) with custom mean, residual, and addition functions. The most common use for these functions is when you are estimating angles whose arithmetic operations need to be wrapped.

Parameters

<i>f</i>	A vector valued function of x and u that returns the derivative of the state vector with respect to time.
<i>h</i>	A vector valued function of x and u that returns the expected measurement at the given state.
<i>integrator</i>	A function from "numerical_integration.h" that integrates a differential equation of the form $f(x, u)$.
<i>state_stddevs</i>	Standard deviations of the states in the model.
<i>measurement_stddevs</i>	Standard deviations of the measurements.
<i>mean_func_X</i>	A function that computes the mean of a matrix containing <code>NUM_SIGMAS</code> state sigma points with a set of weights for each.
<i>mean_func_Y</i>	A function that computes the mean of a matrix containing <code>NUM_SIGMAS</code> measurement sigma points with a set of weights for each.
<i>residual_func_X</i>	A function that computes the residual of two state vectors, usually by simple subtraction.
<i>residual_func_Y</i>	A function that computes the residual of two measurement vectors, usually by simple subtraction.
<i>add_func_X</i>	A function that adds two state vectors.

4.65.3 Member Function Documentation

correct() [1/4]

```
template<int STATES, int INPUTS, int OUTPUTS>
template<int ROWS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::correct (
    const InputVector & u,
    const EVec< ROWS > & y,
    const std::function< EVec< ROWS >(const StateVector &, const InputVector &) > &
h,
    const EVec< ROWS > & measurement_stddevs) [inline]
```

Correct the state estimate using the measurements in y , a custom measurement function, and custom standard deviations. This is useful for when a different set of measurements are being applied.

Parameters

u	The control input used in the last predict step.
y	The vector of measurements.
h	A vector valued function of x and u that returns the expected measurement at the given state.
$measurement_stddevs$	The vector of standard deviations for each measurement to be used for this correct step.

correct() [2/4]

```
template<int STATES, int INPUTS, int OUTPUTS>
template<int ROWS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::correct (
    const InputVector & u,
    const EVec< ROWS > & y,
    const std::function< EVec< ROWS >(const StateVector &, const InputVector &) > &
h,
    const EVec< ROWS > measurement_stddevs,
    const std::function< EVec< ROWS >(const EMat< ROWS, NUM_SIGMAS > &, const EVec< NUM_SIGMAS > &) > & mean_func_Y,
    const std::function< EVec< ROWS >(const EVec< ROWS > &, const EVec< ROWS > &) > & residual_func_Y,
    const std::function< StateVector(const StateVector &, const StateVector &) > & residual_func_X,
    const std::function< StateVector(const StateVector &, const StateVector &) > & add_func_X) [inline]
```

Correct the state estimate using the measurements in y , a custom measurement function, custom standard deviations, and custom mean, residual, and addition functions. This is useful for when a different set of measurements are being applied, and they require custom arithmetic functions.

Parameters

u	The control input used in the last predict step.
y	The vector of measurements.
h	A vector valued function of x and u that returns the expected measurement at the given state.

Parameters

<i>measurement_stddevs</i>	The vector of standard deviations for each measurement to be used for this correct step.
<i>mean_func_Y</i>	A function that computes the mean of a matrix containing NUM_SIGMAS measurement sigma points with a set of weights for each.
<i>residual_func_X</i>	A function that computes the residual of two state vectors, usually by simple subtraction.
<i>residual_func_Y</i>	A function that computes the residual of two measurement vectors, usually by simple subtraction.
<i>add_func_X</i>	A function that adds two state vectors.

correct() [3/4]

```
template<int STATES, int INPUTS, int OUTPUTS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::correct (
    const InputVector & u,
    const OutputVector & y) [inline]
```

Correct the state estimate using the measurements in y.

Parameters

<i>u</i>	The control input used in the last predict step.
<i>y</i>	The vector of measurements.

correct() [4/4]

```
template<int STATES, int INPUTS, int OUTPUTS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::correct (
    const InputVector & u,
    const OutputVector & y,
    const EVec< OUTPUTS > & measurement_stddevs) [inline]
```

Correct the state estimate using the measurements in y, and custom standard deviations. This is useful for when the noise in the measurements vary.

Parameters

<i>u</i>	The control input used in the last predict step.
<i>y</i>	The vector of measurements.
<i>measurement_stddevs</i>	The vector of standard deviations for each measurement to be used for this correct step.

P()

```
template<int STATES, int INPUTS, int OUTPUTS>
StateMatrix SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::P () const [inline]
```

Returns the reconstructed covariance matrix P.

predict()

```
template<int STATES, int INPUTS, int OUTPUTS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::predict (
    const InputVector & u,
    double dt) [inline]
```

Projects the state into the future by dt seconds with control input u.

Parameters

<i>u</i>	The control input.
<i>dt</i>	The timestep in seconds.

reset()

```
template<int STATES, int INPUTS, int OUTPUTS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::reset () [inline]
```

Resets the filter. Make sure to explicitly set S after calling this.

S() [1/2]

```
template<int STATES, int INPUTS, int OUTPUTS>
StateMatrix SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::S () const [inline]
```

Returns the square-root covariance matrix S.

S() [2/2]

```
template<int STATES, int INPUTS, int OUTPUTS>
double SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::S (
    int i,
    int j) const [inline]
```

Returns one element of the square-root covariance matrix S.

Parameters

<i>i</i>	Row of S.
<i>j</i>	Column of S.

set_P()

```
template<int STATES, int INPUTS, int OUTPUTS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::set_P (
    const StateMatrix & P) [inline]
```

Set the current square-root covariance matrix S to the square-root of P.

Parameters

<i>P</i>	The covariance matrix P.
----------	--------------------------

set_S()

```
template<int STATES, int INPUTS, int OUTPUTS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::set_S (
    const StateMatrix & S) [inline]
```

Set the current square-root covariance matrix S.

Parameters

<i>S</i>	The new square-root covariance matrix S.
----------	--

set_xhat() [1/2]

```
template<int STATES, int INPUTS, int OUTPUTS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::set_xhat (
    const StateVector & xhat) [inline]
```

Set the current state estimate x-hat.

set_xhat() [2/2]

```
template<int STATES, int INPUTS, int OUTPUTS>
void SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::set_xhat (
    int i,
    double value) [inline]
```

Set one element of the current state estimate x-hat.

Parameters

<i>i</i>	Row of x-hat.
----------	---------------

xhat() [1/2]

```
template<int STATES, int INPUTS, int OUTPUTS>
StateVector SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::xhat () const [inline]
```

Returns the current state estimate x-hat.

xhat() [2/2]

```
template<int STATES, int INPUTS, int OUTPUTS>
double SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >::xhat (
    int i) const [inline]
```

Returns one element of the current state estimate x-hat.

Parameters

<i>i</i>	Row of $x\hat{}$.
----------	--------------------

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

- srukf.h

4.66 StateMachine< System, IDType, Message, delay_ms, do_log >::State Struct Reference

```
#include <state_machine.h>
```

4.66.1 Detailed Description

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log = false>
struct StateMachine< System, IDType, Message, delay_ms, do_log >::State
```

Abstract class that all states for this machine must inherit from States MUST override respond() and id() in order to function correctly (the compiler won't have it any other way)

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

- state_machine.h

4.67 StateMachine< System, IDType, Message, delay_ms, do_log > Class Template Reference

State Machine :)))))) A fun fun way of controlling stateful subsystems - used in the 2023-2024 Over Under game for our overly complex intake-cata subsystem (see there for an example) The statemachine runs in a background thread and a user thread can interact with it through current_state and send_message.

```
#include <state_machine.h>
```

Classes

- class [MaybeMessage](#)
MaybeMessage a message of Message type or nothing `MaybeMessage m = {};` // empty `MaybeMessage m = Message::EnumField1.`
- struct [State](#)

Public Member Functions

- [StateMachine \(State *initial\)](#)
Construct a state machine and immediately start running it.
- IDType [current_state \(\) const](#)
retrieve the current state of the state machine. This is safe to call from external threads
- void [send_message \(Message msg\)](#)
send a message to the state machine from outside

4.67.1 Detailed Description

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log = false>
class StateMachine< System, IDType, Message, delay_ms, do_log >
```

State Machine :)))))) A fun fun way of controlling stateful subsystems - used in the 2023-2024 Over Under game for our overly complex intake-cata subsystem (see there for an example) The statemachine runs in a background thread and a user thread can interact with it through current_state and send_message.

Designwise: the System class should hold onto any motors, feedback controllers, etc that are persistent in the system States themselves should hold any data that *only* that state needs. For example if a state should be exitted after a certain amount of time, it should hold a timer rather than the System holding that timer. (see Junder from 2024 for an example of this design)

Template Parameters

<i>System</i>	The system that this is the base class of <code>class Thing : public StateMachine<Thing></code> @tparam IDType The ID enum that recognizes states. Hint hint, use anenum class`
<i>Message</i>	the message enum that a state or an outside can send and that states respond to
<i>delay_ms</i>	the delay to wait between each state processing to allow other threads to work
<i>do_log</i>	true if you want print statements describing incoming messages and current states. If true, it is expected that IDType and Message have a function called <code>to_string</code> that takes them as its only parameter and returns a std::string

4.67.2 Constructor & Destructor Documentation

StateMachine()

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log = false>
StateMachine< System, IDType, Message, delay_ms, do_log >::StateMachine (
    State * initial) [inline]
```

Construct a state machine and immediatly start running it.

Parameters

<i>initial</i>	the state that the machine will begin in
----------------	--

4.67.3 Member Function Documentation

current_state()

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log = false>
IDType StateMachine< System, IDType, Message, delay_ms, do_log >::current_state () const
[inline]
```

retrieve the current state of the state machine. This is safe to call from external threads

Returns

the current state

send_message()

```
template<typename System, typename IDType, typename Message, int32_t delay_ms, bool do_log = false>
void StateMachine< System, IDType, Message, delay_ms, do_log >::send_message (
    Message msg) [inline]
```

send a message to the state machine from outside

Parameters

<i>msg</i>	the message to send This is safe to call from external threads
------------	--

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

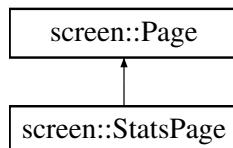
- state_machine.h

4.68 screen::StatsPage Class Reference

Draws motor stats and battery stats to the screen.

```
#include <screen.h>
```

Inheritance diagram for screen::StatsPage:

**Public Member Functions**

- **StatsPage** (std::map< std::string, vex::motor & > motors)
Creates a stats page.
- void **update** (bool was_pressed, int x, int y) override
- void **draw** (vex::brain::lcd &, bool first_draw, unsigned int frame_number) override

4.68.1 Detailed Description

Draws motor stats and battery stats to the screen.

4.68.2 Constructor & Destructor Documentation**StatsPage()**

```
screen::StatsPage::StatsPage (
    std::map< std::string, vex::motor & > motors)
```

Creates a stats page.

Parameters

<i>motors</i>	a map of string to motor that we want to draw on this page
---------------	--

4.68.3 Member Function Documentation**draw()**

```
void screen::StatsPage::draw (
    vex::brain::lcd & scr,
    bool first_draw,
    unsigned int frame_number) [override], [virtual]
```

See also

[Page::draw](#)

Reimplemented from [screen::Page](#).

update()

```
void screen::StatsPage::update (
    bool was_pressed,
    int x,
    int y) [override], [virtual]
```

See also

[Page::update](#)

Reimplemented from [screen::Page](#).

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

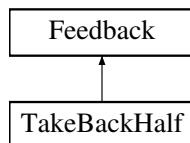
- [screen.h](#)
- [screen.cpp](#)

4.69 TakeBackHalf Class Reference

A velocity controller.

```
#include <take_back_half.h>
```

Inheritance diagram for TakeBackHalf:



Public Member Functions

- void [init](#) (double start_pt, double set_pt)
- double [update](#) (double val) override
- double [get](#) () override
- void [set_limits](#) (double lower, double upper) override
- bool [is_on_target](#) () override

Public Attributes

- double **TBH_gain**
tuned parameter

4.69.1 Detailed Description

A velocity controller.

Warning

If you try to use this as a position controller, it will fail.

4.69.2 Member Function Documentation**get()**

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

Returns

the last saved result from the feedback controller

Implements [Feedback](#).

init()

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

Initialize the feedback controller for a movement

Parameters

<i>start_pt</i>	the current sensor value
<i>set_pt</i>	where the sensor value should be
<i>start_vel</i>	Movement starting velocity (IGNORED)
<i>end_vel</i>	Movement ending velocity (IGNORED)

Implements [Feedback](#).

is_on_target()

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

Returns

true if the feedback controller has reached it's setpoint

Implements [Feedback](#).

set_limits()

```
void TakeBackHalf::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

<i>lower</i>	Upper limit
<i>upper</i>	Lower limit

Implements [Feedback](#).

update()

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

Iterate the feedback loop once with an updated sensor value

Parameters

<i>val</i>	value from the sensor
------------	-----------------------

Returns

feedback loop result

Implements [Feedback](#).

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

- take_back_half.h
- take_back_half.cpp

4.70 TankDrive Class Reference

```
#include <tank_drive.h>
```

Public Types

- enum class `BrakeType` { `None` , `ZeroVelocity` , `Smart` , `TurnOnly` }

Public Member Functions

- `TankDrive` (`motor_group &left_motors`, `motor_group &right_motors`, `robot_specs_t &config`, `OdometryBase *odom=NULL`)
- void `stop` ()
- `Pose2d get_position` ()
- void `drive_tank` (`double left`, `double right`, `int power=1`, `BrakeType bt=BrakeType::None`)
- void `drive_tank_raw` (`double left`, `double right`)
- void `drive_arena` (`double forward_back`, `double left_right`, `int power=1`, `BrakeType bt=BrakeType::None`)
- bool `drive_forward` (`double inches`, `directionType dir`, `Feedback &feedback`, `double max_speed=1`, `double end_speed=0`)
- bool `drive_forward` (`double inches`, `directionType dir`, `double max_speed=1`, `double end_speed=0`)
- bool `turn_degrees` (`double degrees`, `Feedback &feedback`, `double max_speed=1`, `double end_speed=0`)
- bool `turn_degrees` (`double degrees`, `double max_speed=1`, `double end_speed=0`)
- bool `drive_to_point` (`double x`, `double y`, `vex::directionType dir`, `Feedback &feedback`, `double max_speed=1`, `double end_speed=0`)
- bool `drive_to_point` (`double x`, `double y`, `vex::directionType dir`, `double max_speed=1`, `double end_speed=0`)
- bool `turn_to_heading` (`double heading_deg`, `Feedback &feedback`, `double max_speed=1`, `double end_speed=0`)
- bool `turn_to_heading` (`double heading_deg`, `double max_speed=1`, `double end_speed=0`)
- void `reset_auto` ()
- bool `pure_pursuit` (`PurePursuit::Path path`, `directionType dir`, `Feedback &feedback`, `double max_speed=1`, `double end_speed=0`)

Static Public Member Functions

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

4.70.1 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

4.70.2 Member Enumeration Documentation

BrakeType

```
enum class TankDrive::BrakeType [strong]
```

Enumerator

None	just send 0 volts to the motors
ZeroVelocity	try to bring the robot to rest. But don't try to hold position
Smart	bring the robot to rest and once it's stopped, try to hold that position

4.70.3 Constructor & Destructor Documentation**TankDrive()**

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

Create the [TankDrive](#) object

Parameters

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

4.70.4 Member Function Documentation**drive_arcade()**

```
void TankDrive::drive_arcade (
    double forward_back,
    double left_right,
    int power = 1,
    BrakeType bt = BrakeType::None)
```

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

<i>forward_back</i>	the percent to move forward or backward
<i>left_right</i>	the percent to turn left or right
<i>power</i>	modifies the input velocities left^power, right^power
<i>bt</i>	breaktype. What to do if the driver lets go of the sticks

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

drive_forward() [1/2]

```
bool TankDrive::drive_forward (
    double inches,
    directionType dir,
    double max_speed = 1,
    double end_speed = 0)
```

Autonomously drive the robot forward a certain distance

Parameters

<i>inches</i>	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
<i>dir</i>	the direction we want to travel forward and backward
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Autonomously drive the robot forward a certain distance

Parameters

<i>inches</i>	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
<i>dir</i>	the direction we want to travel forward and backward
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Returns

true if we have finished driving to our point

drive_forward() [2/2]

```
bool TankDrive::drive_forward (
    double inches,
    directionType dir,
    Feedback & feedback,
    double max_speed = 1,
    double end_speed = 0)
```

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

<i>inches</i>	the distance to drive forward
<i>dir</i>	the direction we want to travel forward and backward
<i>feedback</i>	the custom feedback controller we will use to travel. controls the rate at which we accelerate and drive.
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

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

<i>inches</i>	the distance to drive forward
<i>dir</i>	the direction we want to travel forward and backward
<i>feedback</i>	the custom feedback controller we will use to travel. controls the rate at which we accelerate and drive.
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

drive_tank()

```
void TankDrive::drive_tank (
    double left,
    double right,
    int power = 1,
    BrakeType bt = BrakeType::None)
```

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

<i>left</i>	the percent to run the left motors
<i>right</i>	the percent to run the right motors
<i>power</i>	modifies the input velocities <i>left</i> ^{<i>power</i>} , <i>right</i> ^{<i>power</i>}
<i>bt</i>	breaktype. What to do if the driver lets go of the sticks

drive_tank_raw()

```
void TankDrive::drive_tank_raw (
    double left,
    double right)
```

Drive the robot raw-ly

Parameters

<i>left</i>	the percent to run the left motors (-1, 1)
<i>right</i>	the percent to run the right motors (-1, 1)

drive_to_point() [1/2]

```
bool TankDrive::drive_to_point (
    double x,
    double y,
    vex::directionType dir,
    double max_speed = 1,
    double end_speed = 0)
```

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

<i>x</i>	the x position of the target
<i>y</i>	the y position of the target
<i>dir</i>	the direction we want to travel forward and backward
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

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

<i>x</i>	the x position of the target
<i>y</i>	the y position of the target
<i>dir</i>	the direction we want to travel forward and backward
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Returns

true if we have reached our target point

drive_to_point() [2/2]

```
bool TankDrive::drive_to_point (
    double x,
    double y,
    vex::directionType dir,
    Feedback & feedback,
    double max_speed = 1,
    double end_speed = 0)
```

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

<i>x</i>	the x position of the target
<i>y</i>	the y position of the target
<i>dir</i>	the direction we want to travel forward and backward
<i>feedback</i>	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

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

<i>x</i>	the x position of the target
<i>y</i>	the y position of the target
<i>dir</i>	the direction we want to travel forward and backward
<i>feedback</i>	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Returns

true if we have reached our target point

get_position()

```
Pose2d TankDrive::get_position ()
```

Returns the Robot position as a [Pose2d](#)

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

<i>input</i>	the input before modification
<i>power</i>	the power to raise input to

Returns

$\text{input}^{\wedge} \text{power}$ (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

<i>input</i>	the input signal -1 -> 1
<i>power</i>	the power to raise the signal to

Returns

$\text{input}^{\wedge} \text{power}$ accounting for any sign issues that would arise with this naive solution

pure_pursuit()

```
bool TankDrive::pure_pursuit (
    PurePursuit::Path path,
    directionType dir,
    Feedback & feedback,
    double max_speed = 1,
    double end_speed = 0)
```

Drive the robot autonomously using a pure-pursuit algorithm - Input path with a set of waypoints - the robot will attempt to follow the points while cutting corners (radius) to save time (compared to stop / turn / start)

Parameters

<i>path</i>	The list of coordinates to follow, in order
<i>dir</i>	Run the bot forwards or backwards
<i>feedback</i>	The feedback controller determining speed
<i>max_speed</i>	Limit the speed of the robot (for pid / pidff feedbacks)
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Returns

True when the path is complete

Drive the robot autonomously using a pure-pursuit algorithm - Input path with a set of waypoints - the robot will attempt to follow the points while cutting corners (radius) to save time (compared to stop / turn / start)

Parameters

<i>path</i>	The list of coordinates to follow, in order
<i>dir</i>	Run the bot forwards or backwards
<i>feedback</i>	The feedback controller determining speed
<i>max_speed</i>	Limit the speed of the robot (for pid / pidff feedbacks)

Returns

True when the path is complete

reset_auto()

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

Reset the initialization for autonomous drive functions

stop()

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

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

turn_degrees() [1/2]

```
bool TankDrive::turn_degrees (
    double degrees,
    double max_speed = 1,
    double end_speed = 0)
```

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 default turning feedback of the drive system.

Parameters

<i>degrees</i>	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

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 default turning feedback of the drive system.

Parameters

<i>degrees</i>	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Returns

true if we turned to target number of degrees

turn_degrees() [2/2]

```
bool TankDrive::turn_degrees (
    double degrees,
    Feedback & feedback,
    double max_speed = 1,
    double end_speed = 0)
```

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 its control.

Parameters

<i>degrees</i>	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
<i>feedback</i>	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
<i>max_speed</i>	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 its control.

Parameters

<i>degrees</i>	degrees by which we will turn relative to the robot (+) turns ccw, (-) turns cw
<i>feedback</i>	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Returns

true if we have turned our target number of degrees

turn_to_heading() [1/2]

```
bool TankDrive::turn_to_heading (
    double heading_deg,
    double max_speed = 1,
    double end_speed = 0)
```

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

<i>heading_deg</i>	the heading to which we will turn
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

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

<i>heading_deg</i>	the heading to which we will turn
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Returns

true if we have reached our target heading

turn_to_heading() [2/2]

```
bool TankDrive::turn_to_heading (
    double heading_deg,
    Feedback & feedback,
    double max_speed = 1,
    double end_speed = 0)
```

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

Parameters

<i>heading_deg</i>	the heading to which we will turn
<i>feedback</i>	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

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

Parameters

<i>heading_deg</i>	the heading to which we will turn
<i>feedback</i>	the feedback controller we will use to travel. controls the rate at which we accelerate and drive.
<i>max_speed</i>	the maximum percentage of robot speed at which the robot will travel. 1 = full power
<i>end_speed</i>	the movement profile will attempt to reach this velocity by its completion

Returns

true if we have reached our target heading

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

- tank_drive.h
- tank_drive.cpp

4.71 tracking_wheel_cfg_t Struct Reference

```
#include <odometry_nwheel.h>
```

Public Attributes

- double *x*
- double *y*
- double *theta_rad*
- double *radius*

4.71.1 Detailed Description

OdometryNWheel

This class handles the code for an N-pod odometry setup, where there are N <WHEELS> free spinning omni wheels (dead wheels) placed in any known configuration on the robot.

Example of a possible wheel configuration:

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

Where O is the center of rotation. The robot will monitor the changes in rotation of these wheels, use this to calculate a pose delta, then integrate the deltas over time to determine the robot's position.

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.

<https://rit.enterprise.slack.com/files/U04112Y5RB6/F080M01KPA5/predictperpendiculars2.pdf> 2024-2025 Notebook: Entries/Software Entries/Localization/N-Pod Odometry

Author

Jack Cammarata, Richie Sommers

Date

Nov 14 2024 [tracking_wheel_cfg_t](#) holds all the specifications for a single tracking wheel. The units for x, y, and radius will determine the units of the position estimate.

4.71.2 Member Data Documentation

radius

double tracking_wheel_cfg_t::radius

radius of the wheel

theta_rad

double tracking_wheel_cfg_t::theta_rad

angle between wheel direction and x axis in the robot frame

x

double tracking_wheel_cfg_t::x

x position of the center of the wheel

y

double tracking_wheel_cfg_t::y

y position of the center of the wheel

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

- [odometry_nwheel.h](#)

4.72 Transform2d Class Reference

```
#include <transform2d.h>
```

Public Member Functions

- `constexpr Transform2d ()`
- `Transform2d (const Translation2d &translation, const Rotation2d &rotation)`
- `Transform2d (const double &x, const double &y, const Rotation2d &rotation)`
- `Transform2d (const double &x, const double &y, const double &radians)`
- `Transform2d (const Translation2d &translation, const double &radians)`
- `Transform2d (const Eigen::Vector3d &transform_vector)`
- `Transform2d (const Pose2d &start, const Pose2d &end)`
- `Translation2d translation () const`
- `double x () const`
- `double y () const`
- `Rotation2d rotation () const`
- `Transform2d inverse () const`
- `Transform2d operator* (const double &scalar) const`
- `Transform2d operator/ (const double &scalar) const`
- `Transform2d operator- () const`
- `bool operator== (const Transform2d &other) const`

Friends

- `std::ostream & operator<< (std::ostream &os, const Transform2d &transform)`

4.72.1 Detailed Description

Class representing a transformation of a pose2d, or a linear difference between the components of poses.

Assumes conventional cartesian coordinate system: Looking down at the coordinate plane, +X is right +Y is up
+Theta is counterclockwise

4.72.2 Constructor & Destructor Documentation

Transform2d() [1/7]

```
Transform2d::Transform2d () [constexpr]
```

Default Constructor for `Transform2d`

Transform2d() [2/7]

```
Transform2d::Transform2d (
    const Translation2d & translation,
    const Rotation2d & rotation)
```

Constructs a transform given translation and rotation components.

Parameters

<code>translation</code>	the translational component of the transform.
<code>rotation</code>	the rotational component of the transform.

Transform2d() [3/7]

```
Transform2d::Transform2d (
    const double & x,
    const double & y,
    const Rotation2d & rotation)
```

Constructs a transform given translation and rotation components.

Parameters

<i>x</i>	the x component of the transform.
<i>y</i>	the y component of the transform.
<i>rotation</i>	the rotational component of the transform.

Transform2d() [4/7]

```
Transform2d::Transform2d (
    const double & x,
    const double & y,
    const double & radians)
```

Constructs a transform given translation and rotation components.

Parameters

<i>x</i>	the x component of the transform.
<i>y</i>	the y component of the transform.
<i>radians</i>	the rotational component of the transform in radians.

Transform2d() [5/7]

```
Transform2d::Transform2d (
    const Translation2d & translation,
    const double & radians)
```

Constructs a transform given translation and rotation components.

Parameters

<i>translation</i>	the translational component of the transform.
<i>radians</i>	the rotational component of the transform in radians.

Transform2d() [6/7]

```
Transform2d::Transform2d (
    const Eigen::Vector3d & transform_vector)
```

Constructs a transform given translation and rotation components given as a vector.

Parameters

<i>transform_vector</i>	vector of the form [x, y, theta]
-------------------------	----------------------------------

Transform2d() [7/7]

```
Transform2d::Transform2d (
    const Pose2d & start,
    const Pose2d & end)
```

Constructs a transform given translation and rotation components.

Parameters

<i>translation</i>	the translational component of the transform.
<i>rotation</i>	the rotational component of the transform.

4.72.3 Member Function Documentation**inverse()**

```
Transform2d Transform2d::inverse () const
```

Inverts the transform.

Returns

the inverted transform.

operator*()

```
Transform2d Transform2d::operator* (
    const double & scalar) const
```

Multiplies this transform by a scalar.

Parameters

<i>scalar</i>	the scalar to multiply this transform by.
---------------	---

operator-()

```
Transform2d Transform2d::operator- () const
```

Inverts the transform.

Returns

the inverted transform.

operator/()

```
Transform2d Transform2d::operator/ (
    const double & scalar) const
```

Divides this transform by a scalar.

Parameters

<code>scalar</code>	the scalar to divide this transform by.
---------------------	---

operator==()

```
bool Transform2d::operator== (
    const Transform2d & other) const
```

Compares this to another transform.

Parameters

<code>other</code>	the other transform to compare to.
--------------------	------------------------------------

Returns

true if the components are within 1e-9 of each other.

rotation()

```
Rotation2d Transform2d::rotation () const
```

Returns the rotational component of the transform.

Returns

the rotational component of the transform.

translation()

```
Translation2d Transform2d::translation () const
```

Returns the translational component of the transform.

Returns

the translational component of the transform.

x()

```
double Transform2d::x () const
```

Returns the x component of the transform.

Returns

the x component of the transform.

y()

```
double Transform2d::y () const
```

Returns the y component of the transform.

Returns

the y component of the transform.

4.72.4 Friends And Related Symbol Documentation**operator<<**

```
std::ostream & operator<< (
    std::ostream & os,
    const Transform2d & transform) [friend]
```

Sends a transform to an output stream. Ex. std::cout << transform;

prints "Transform2d[dx: (value), dy: (value), drad: (radians), ddeg: (degrees)]"

Sends a transform to an output stream. Ex. std::cout << transform;

prints "Transform2d[x: (value), y: (value), rad: (radians), deg: (degrees)]"

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

- transform2d.h
- transform2d.cpp

4.73 Translation2d Class Reference

```
#include <translation2d.h>
```

Public Member Functions

- `constexpr Translation2d ()`
- `Translation2d (const double &x, const double &y)`
- `Translation2d (const Eigen::Vector2d &vector)`
- `Translation2d (const double &r, const Rotation2d &theta)`
- `double x () const`
- `void setX (double x)`
- `double y () const`
- `void setY (double y)`
- `Rotation2d theta () const`
- `Eigen::Vector2d as_vector () const`
- `double norm () const`
- `double distance (const Translation2d &other) const`
- `Translation2d rotate_by (const Rotation2d &rotation) const`
- `Translation2d rotate_around (const Translation2d &other, const Rotation2d &rotation) const`
- `Translation2d operator+ (const Translation2d &other) const`
- `Translation2d operator- (const Translation2d &other) const`
- `Translation2d operator- () const`
- `Translation2d operator* (const double &scalar) const`
- `Translation2d operator/ (const double &scalar) const`
- `double operator* (const Translation2d &other) const`
- `bool operator== (const Translation2d &other) const`

Friends

- std::ostream & [operator<<](#) (std::ostream &os, const [Translation2d](#) &translation)

4.73.1 Detailed Description

Class representing a point in 2d space with x and y coordinates.

Assumes conventional cartesian coordinate system: Looking down at the coordinate plane, +X is right +Y is up
+Theta is counterclockwise

4.73.2 Constructor & Destructor Documentation

[Translation2d\(\)](#) [1/4]

```
Translation2d::Translation2d () [inline], [constexpr]
```

Default Constructor for [Translation2d](#)

[Translation2d\(\)](#) [2/4]

```
Translation2d::Translation2d (
    const double & x,
    const double & y)
```

Constructs a [Translation2d](#) with the given x and y values.

Parameters

x	The x component of the translation.
y	The y component of the translation.

[Translation2d\(\)](#) [3/4]

```
Translation2d::Translation2d (
    const Eigen::Vector2d & vector)
```

Constructs a [Translation2d](#) with the values from the given vector.

Parameters

vector	The vector whose values will be used.
--------	---------------------------------------

[Translation2d\(\)](#) [4/4]

```
Translation2d::Translation2d (
    const double & r,
    const Rotation2d & theta)
```

Constructs a [Translation2d](#) given polar coordinates of the form (r, theta).

Parameters

<i>r</i>	The radius (magnitude) of the vector.
<i>theta</i>	The angle (direction) of the vector.

4.73.3 Member Function Documentation**as_vector()**

```
Eigen::Vector2d Translation2d::as_vector () const
```

Returns the vector as an Eigen::Vector2d.

Returns

Eigen::Vector2d with the same values as the translation.

distance()

```
double Translation2d::distance (
    const Translation2d & other) const
```

Returns the distance between two translations.

Returns

the distance between two translations.

norm()

```
double Translation2d::norm () const
```

Returns the norm/radius/magnitude/distance from origin.

Returns

the norm of the translation.

operator*() [1/2]

```
Translation2d Translation2d::operator* (
    const double & scalar) const
```

Returns this translation multiplied by a scalar.

[x] = [x] * [scalar] [y] = [y] * [scalar]

Parameters

<i>scalar</i>	the scalar to multiply by.
---------------	----------------------------

Returns

this translation multiplied by a scalar.

operator*() [2/2]

```
double Translation2d::operator* (
    const Translation2d & other) const
```

Returns the dot product of two translations.

[scalar] = [x][otherx] + [y][othery]

Parameters

<i>other</i>	the other translation to find the dot product with.
--------------	---

Returns

the scalar valued dot product.

operator+()

```
Translation2d Translation2d::operator+ (
    const Translation2d & other) const
```

Returns the sum of two translations.

[x] = [x] + [otherx]; [y] = [y] + [othery];

Parameters

<i>other</i>	the other translation to add to this translation.
--------------	---

Returns

the sum of the two translations.

operator-() [1/2]

```
Translation2d Translation2d::operator- () const
```

Returns the inverse of this translation. Equivalent to flipping the vector across the origin.

[x] = [-x] [y] = [-y]

Returns

the inverse of this translation.

operator-() [2/2]

```
Translation2d Translation2d::operator- (
    const Translation2d & other) const
```

Returns the difference of two translations.

$[x] = [x] - [\text{other}x]$ $[y] = [y] - [\text{other}y]$

Parameters

<i>other</i>	the translation to subtract from this translation.
--------------	--

Returns

the difference of the two translations.

operator/()

```
Translation2d Translation2d::operator/ (
    const double & scalar) const
```

Returns this translation divided by a scalar.

$[x] = [x] / [\text{scalar}]$ $[y] = [y] / [\text{scalar}]$

Parameters

<i>scalar</i>	the scalar to divide by.
---------------	--------------------------

Returns

this translation divided by a scalar.

operator==()

```
bool Translation2d::operator== (
    const Translation2d & other) const
```

Compares two translations. Returns true if their components are each within 1e-9, to account for floating point error.

Parameters

<i>other</i>	the translation to compare to.
--------------	--------------------------------

Returns

whether the two translations are equal.

rotate_around()

```
Translation2d Translation2d::rotate_around (
    const Translation2d & other,
    const Rotation2d & rotation) const
```

Applies a rotation to this translation around another given point.

$[x] = [\cos, -\sin][x - \text{other}x] + [\text{other}x]$ $[y] = [\sin, \cos][y - \text{other}y] + [\text{other}y]$

Parameters

<i>other</i>	the center of rotation.
<i>rotation</i>	the angle amount the translation will be rotated.

Returns

the translation that has been rotated.

rotate_by()

```
Translation2d Translation2d::rotate_by (
    const Rotation2d & rotation) const
```

Applies a rotation to this translation around the origin.

Equivalent to multiplying a vector by a rotation matrix: $x = [\cos, -\sin][x]$ $y = [\sin, \cos][y]$

Parameters

<i>rotation</i>	the angle amount the translation will be rotated.
-----------------	---

Returns

the new translation that has been rotated around the origin.

setX()

```
void Translation2d::setX (
    double x)
```

Sets the x value of the translation.

setY()

```
void Translation2d::setY (
    double y)
```

Sets the y value of the translation.

theta()

```
Rotation2d Translation2d::theta () const
```

Returns the angle of the translation.

Returns

the angle of the translation.

x()

```
double Translation2d::x () const
```

Returns the x value of the translation.

Returns

the x value of the translation.

y()

```
double Translation2d::y () const
```

Returns the y value of the translation.

Returns

the y value of the translation.

4.73.4 Friends And Related Symbol Documentation

operator<<

```
std::ostream & operator<< (
    std::ostream & os,
    const Translation2d & translation) [friend]
```

Sends a translation to an output stream. Ex. std::cout << translation;

prints "Translation2d[x: (value), y: (value)]"

Sends a translation to an output stream. Ex. std::cout << translation;

prints "Translation2d[x: (value), y: (value), rad: (radians), deg: (degrees)]"

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

- translation2d.h
- translation2d.cpp

4.74 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 elapsed.`
- `void set_endpts (double start, double end)`
- `void set_accel (double accel)`
- `void set_max_v (double max_v)`
- `double get_movement_time ()`

4.74.1 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 its name.

If the maximum velocity is set high enough, this will become an 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 feedforward 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/rkm3ivulyk>

Author

Ryan McGee

Date

7/12/2022

4.74.2 Constructor & Destructor Documentation

`TrapezoidProfile()`

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

Construct a new Trapezoid Profile object.

Parameters

<code>max_v</code>	Maximum velocity the robot can run at
<code>accel</code>	Maximum acceleration of the robot

4.74.3 Member Function Documentation

`calculate()`

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

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

Parameters

<i>time</i> _s	Time since start of movement
-------------------	------------------------------

Returns

[motion_t](#) Position, velocity and acceleration

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

set_accel()

```
void TrapezoidProfile::set_accel (
```

<i>accel</i>	the acceleration amount to use
--------------	--------------------------------

```
    double accel)
```

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

Parameters

<i>accel</i>	the acceleration amount to use
--------------	--------------------------------

set_endpts()

```
void TrapezoidProfile::set_endpts (
```

<i>start</i>	the starting position of the path
<i>end</i>	the ending position of the path

```
    double start,
```

```
    double end)
```

set_endpts defines a start and end position

Parameters

<i>start</i>	the starting position of the path
<i>end</i>	the ending position of the path

set_max_v()

```
void TrapezoidProfile::set_max_v (
```

<i>max_v</i>

```
    double max_v)
```

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

Parameters

<i>max_v</i>	the maximum velocity the robot can travel at
--------------	--

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

- trapezoid_profile.h
- trapezoid_profile.cpp

4.75 TurnDegreesCommand Class Reference

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

Public Member Functions

- [TurnDegreesCommand \(TankDrive &drive_sys, Feedback &feedback, double degrees, double max_speed=1, double end_speed=0\)](#)
- bool [run \(\) override](#)
- void [on_timeout \(\) override](#)

4.75.1 Detailed Description

AutoCommand wrapper class for the turn_degrees function in the [TankDrive](#) class

4.75.2 Constructor & Destructor Documentation

[TurnDegreesCommand\(\)](#)

```
TurnDegreesCommand::TurnDegreesCommand (
    TankDrive & drive_sys,
    Feedback & feedback,
    double degrees,
    double max_speed = 1,
    double end_speed = 0)
```

Construct a [TurnDegreesCommand](#) Command

Parameters

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

4.75.3 Member Function Documentation

on_timeout()

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

Cleans up drive system if we time out before finishing

reset the drive system if we timeout

run()

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

Run turn_degrees Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

- `drive_commands.h`
- `drive_commands.cpp`

4.76 TurnToHeadingCommand Class Reference

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

Public Member Functions

- `TurnToHeadingCommand (TankDrive &drive_sys, Feedback &feedback, double heading_deg, double speed=1, double end_speed=0)`
- `bool run () override`
- `void on_timeout () override`

4.76.1 Detailed Description

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

4.76.2 Constructor & Destructor Documentation

TurnToHeadingCommand()

```
TurnToHeadingCommand::TurnToHeadingCommand (
    TankDrive & drive_sys,
    Feedback & feedback,
    double heading_deg,
    double max_speed = 1,
    double end_speed = 0)
```

Construct a `TurnToHeadingCommand` Command

Parameters

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

4.76.3 Member Function Documentation**on_timeout()**

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

Cleans up drive system if we time out before finishing

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

run()

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

Run turn_to_heading Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

- drive_commands.h
- drive_commands.cpp

4.77 Twist2d Class Reference

```
#include <twist2d.h>
```

Public Member Functions

- `constexpr Twist2d ()`
- `Twist2d (const double &dx, const double &dy, const double &dtheta)`
- `Twist2d (const Eigen::Vector3d &twist_vector)`
- `double dx () const`
- `double dy () const`
- `double dtheta () const`
- `bool operator== (const Twist2d &other) const`
- `Twist2d operator* (const double &scalar) const`
- `Twist2d operator/ (const double &scalar) const`

Friends

- std::ostream & [operator<<](#) (std::ostream &os, const [Twist2d](#) &twist)

4.77.1 Detailed Description

Class representing a difference between two poses, more specifically a distance along an arc from a pose.

Assumes conventional cartesian coordinate system: Looking down at the coordinate plane, +X is right +Y is up
+Theta is counterclockwise

4.77.2 Constructor & Destructor Documentation**Twist2d() [1/3]**

```
Twist2d::Twist2d () [constexpr]
```

Default Constructor for [Twist2d](#)

Twist2d() [2/3]

```
Twist2d::Twist2d (
    const double & dx,
    const double & dy,
    const double & dtheta)
```

Constructs a twist with given translation and angle deltas.

Parameters

<i>dx</i>	the linear dx component.
<i>dy</i>	the linear dy component.
<i>dtheta</i>	the angular dtheta component.

Twist2d() [3/3]

```
Twist2d::Twist2d (
    const Eigen::Vector3d & twist_vector)
```

Constructs a twist with given translation and angle deltas.

Parameters

<i>twist_vector</i>	vector of the form [dx, dy, dtheta]
---------------------	-------------------------------------

4.77.3 Member Function Documentation

dtheta()

```
double Twist2d::dtheta () const
```

Returns the angular dtheta component.

Returns

the angular dtheta component.

dx()

```
double Twist2d::dx () const
```

Returns the linear dx component.

Returns

the linear dx component.

dy()

```
double Twist2d::dy () const
```

Returns the linear dy component.

Returns

the linear dy component.

operator*()

```
Twist2d Twist2d::operator* (
    const double & scalar) const
```

Multiplies this twist by a scalar.

Parameters

<i>scalar</i>	the scalar value to multiply by.
---------------	----------------------------------

operator/()

```
Twist2d Twist2d::operator/ (
    const double & scalar) const
```

Divides this twist by a scalar.

Parameters

<code>scalar</code>	the scalar value to divide by.
---------------------	--------------------------------

operator==()

```
bool Twist2d::operator== (
    const Twist2d & other) const
```

Compares this to another twist.

Parameters

<code>other</code>	the other twist to compare to.
--------------------	--------------------------------

Returns

true if each of the components are within 1e-9 of each other.

4.77.4 Friends And Related Symbol Documentation

operator<<

```
std::ostream & operator<< (
    std::ostream & os,
    const Twist2d & twist) [friend]
```

Sends a twist to an output stream. Ex. `std::cout << twist;`

prints "Twist2d[dx: (value), dy: (value), drad: (radians)]"

Sends a twist to an output stream. Ex. `std::cout << twist;`

prints "Twist2d[x: (value), y: (value), rad: (radians), deg: (degrees)]"

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

- `twist2d.h`
- `twist2d.cpp`

4.78 WaitUntilCondition Class Reference

Waits until the condition is true.

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

4.78.1 Detailed Description

Waits until the condition is true.

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

- auto_command.h

4.79 WaitUntilUpToSpeedCommand Class Reference

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

Public Member Functions

- [WaitUntilUpToSpeedCommand \(Flywheel &flywheel, int threshold_rpm\)](#)
- bool [run \(\) override](#)

4.79.1 Detailed Description

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

4.79.2 Constructor & Destructor Documentation

[WaitUntilUpToSpeedCommand\(\)](#)

```
WaitUntilUpToSpeedCommand::WaitUntilUpToSpeedCommand (
```

<pre> Flywheel & flywheel,</pre>	
	<pre> int threshold_rpm)</pre>

Create a [WaitUntilUpToSpeedCommand](#)

Parameters

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

4.79.3 Member Function Documentation

[run\(\)](#)

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

Run spin_manual Overrides run from AutoCommand

Returns

true when execution is complete, false otherwise

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

- [flywheel_commands.h](#)
- [flywheel_commands.cpp](#)

Index

accel
 OdometryBase, 71

add
 CommandController, 16, 17
 GenericAuto, 42

add_async
 GenericAuto, 42

add_cancel_func
 CommandController, 17

add_delay
 CommandController, 18
 GenericAuto, 43

add_entry
 ExponentialMovingAverage, 27
 MovingAverage, 63

ang_accel_deg
 OdometryBase, 71

ang_speed_deg
 OdometryBase, 71

as_vector
 Translation2d, 145

Async, 8

auto_drive
 MecanumDrive, 55

auto_turn
 MecanumDrive, 55

AutoChooser, 9
 AutoChooser, 9
 choice, 10
 get_choice, 10
 list, 10

AutoChooser::entry_t, 26
 name, 26

background_task
 OdometryBase, 68

BasicSolenoidSet, 10
 BasicSolenoidSet, 11
 run, 11

BasicSpinCommand, 11
 BasicSpinCommand, 12
 run, 12

BasicStopCommand, 13
 BasicStopCommand, 13
 run, 13

bool_or
 Serializer, 108

BrakeType
 TankDrive, 127

Branch, 14

ButtonWidget
 screen::ButtonWidget, 14, 15

calculate
 FeedForward, 31
 TrapezoidProfile, 151

choice
 AutoChooser, 10

CommandController, 16
 add, 16, 17
 add_cancel_func, 17
 add_delay, 18
 CommandController, 16
 last_command_timed_out, 18
 run, 18

Condition, 18

config
 PID, 85

control_continuous
 Lift< T >, 46

control_manual
 Lift< T >, 46

control_setpoints
 Lift< T >, 46

Core, 1

correct
 SquareRootUnscentedKalmanFilter< STATES, IN-
 PUTS, OUTPUTS >, 117, 118

current_pos
 OdometryBase, 71

current_state
 StateMachine< System, IDType, Message, de-
 lay_ms, do_log >, 122

CustomEncoder, 19
 CustomEncoder, 19
 position, 19
 rotation, 20
 setPosition, 20
 setRotation, 20
 velocity, 20

degrees
 Rotation2d, 99

DelayCommand, 21
 DelayCommand, 21
 run, 22

distance
 Translation2d, 145

double_or
 Serializer, 109

draw
 screen::FunctionPage, 41
 screen::OdometryPage, 73
 screen::Page, 78
 screen::PIDPage, 88
 screen::StatsPage, 124

drive
 MecanumDrive, 56

drive_arcade
 TankDrive, 128

drive_correction_cutoff

robot_specs_t, 97
 drive_forward
 TankDrive, 128, 129
 drive_raw
 MecanumDrive, 57
 drive_tank
 TankDrive, 130
 drive_tank_raw
 TankDrive, 130
 drive_to_point
 TankDrive, 130, 131
 DriveForwardCommand, 22
 DriveForwardCommand, 22
 on_timeout, 23
 run, 23
 DriveStopCommand, 23
 DriveStopCommand, 23
 run, 24
 DriveToPointCommand, 24
 DriveToPointCommand, 24, 25
 run, 25
 dtheta
 Twist2d, 157
 dx
 Twist2d, 157
 dy
 Twist2d, 157

 end_async
 OdometryBase, 69
 error_method
 PID::pid_config_t, 86
 ERROR_TYPE
 PID, 81
 exp
 Pose2d, 91
 ExponentialMovingAverage, 26
 add_entry, 27
 ExponentialMovingAverage, 27
 get_size, 28
 get_value, 28

 f_cos
 Rotation2d, 99
 f_sin
 Rotation2d, 100
 f_tan
 Rotation2d, 100
 Feedback, 28
 get, 29
 init, 29
 is_on_target, 29
 set_limits, 30
 update, 30
 FeedForward, 30
 calculate, 31
 FeedForward, 31
 FeedForward::ff_config_t, 32
 kA, 32

 kG, 32
 kS, 32
 kV, 33
 Filter, 33
 Flywheel, 33
 Flywheel, 34
 get_motors, 35
 get_target, 35
 getRPM, 35
 is_on_target, 35
 Page, 35
 spin_manual, 35
 spin_rpm, 36
 SpinRpmCmd, 36
 spinRPMTask, 37
 stop, 36
 WaitUntilUpToSpeedCmd, 37
 FlywheelStopCommand, 37
 FlywheelStopCommand, 37
 run, 38
 FlywheelStopMotorsCommand, 38
 FlywheelStopMotorsCommand, 38
 run, 39
 FlywheelStopNonTasksCommand, 39
 FunctionCommand, 39
 FunctionCondition, 40
 FunctionPage
 screen::FunctionPage, 41

 GenericAuto, 42
 add, 42
 add_async, 42
 add_delay, 43
 run, 43
 get
 Feedback, 29
 MotionController, 59
 PID, 82
 TakeBackHalf, 125
 get_accel
 OdometryBase, 69
 get_angular_accel_deg
 OdometryBase, 69
 get_angular_speed_deg
 OdometryBase, 69
 get_async
 Lift< T >, 47
 get_choice
 AutoChooser, 10
 get_error
 PID, 82
 get_motion
 MotionController, 59
 get_motors
 Flywheel, 35
 get_movement_time
 TrapezoidProfile, 152
 get_points
 PurePursuit::Path, 80

get_position
 OdometryBase, 69
 TankDrive, 132

get_radius
 PurePursuit::Path, 80

get_sensor_val
 PID, 82

get_setpoint
 Lift< T >, 47

get_size
 ExponentialMovingAverage, 28
 MovingAverage, 63

get_speed
 OdometryBase, 70

get_target
 Flywheel, 35
 PID, 82

get_value
 ExponentialMovingAverage, 28
 MovingAverage, 63

getRPM
 Flywheel, 35

handle
 OdometryBase, 72

has_message
 StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage, 54

hold
 Lift< T >, 47

home
 Lift< T >, 47

IfTimePassed, 44

init
 Feedback, 29
 MotionController, 60
 PID, 83
 TakeBackHalf, 125

InOrder, 44

int_or
 Serializer, 109

inverse
 Transform2d, 141

is_on_target
 Feedback, 29
 Flywheel, 35
 MotionController, 60
 PID, 84
 TakeBackHalf, 125

is_valid
 PurePursuit::Path, 80

kA
 FeedForward::ff_config_t, 32

kG
 FeedForward::ff_config_t, 32

kS
 FeedForward::ff_config_t, 32

kV
 FeedForward::ff_config_t, 33

last_command_timed_out
 CommandController, 18

Lift
 Lift< T >, 45

Lift< T >, 45
 control_continuous, 46
 control_manual, 46
 control_setpoints, 46
 get_async, 47
 get_setpoint, 47
 hold, 47
 home, 47
 Lift, 45
 set_async, 47
 set_position, 48
 set_sensor_function, 48
 set_sensor_reset, 48
 set_setpoint, 48

Lift< T >::lift_cfg_t, 49

list
 AutoChooser, 10

Log
 Logger, 50, 51

log
 Pose2d, 91

Logf
 Logger, 51

Logger, 49
 Log, 50, 51
 Logf, 51
 Logger, 50
 LogIn, 51, 52

LogIn
 Logger, 51, 52

MaybeMessage
 StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage, 53

MecanumDrive, 54
 auto_drive, 55
 auto_turn, 55
 drive, 56
 drive_raw, 57
 MecanumDrive, 55

MecanumDrive::mecanumdrive_config_t, 57

message
 StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage, 54

modify_inputs
 TankDrive, 132

motion_t, 57

MotionController, 58
 get, 59
 get_motion, 59
 init, 60
 is_on_target, 60

MotionController, 59
 set_limits, 60
 tune_feedforward, 61
 update, 61
MotionController::m_profile_cfg_t, 52
MovingAverage, 62
 add_entry, 63
 get_size, 63
 get_value, 63
 MovingAverage, 62
mut
 OdometryBase, 72
name
 AutoChooser::entry_t, 26
None
 TankDrive, 128
norm
 Translation2d, 145
num_sigmas
 ScaledSphericalSimplexSigmaPoints< STATES >, 106
Odometry3Wheel, 64
 Odometry3Wheel, 65
 tune, 65
 update, 66
Odometry3Wheel::odometry3wheel_cfg_t, 66
 off_axis_center_dist, 66
 wheel_diam, 66
 wheelbase_dist, 67
OdometryBase, 67
 accel, 71
 ang_accel_deg, 71
 ang_speed_deg, 71
 background_task, 68
 current_pos, 71
 end_async, 69
 get_accel, 69
 get_angular_accel_deg, 69
 get_angular_speed_deg, 69
 get_position, 69
 get_speed, 70
 handle, 72
 mut, 72
OdometryBase, 68
 set_position, 70
 smallest_angle, 70
 speed, 72
 update, 71
OdometryPage
 screen::OdometryPage, 73
OdometryTank, 74
 OdometryTank, 75
 set_position, 76
 update, 76
OdomSetPosition, 77
 OdomSetPosition, 77
 run, 77
 off_axis_center_dist
 Odometry3Wheel::odometry3wheel_cfg_t, 66
on_target_time
 PID::pid_config_t, 86
on_timeout
 DriveForwardCommand, 23
 PurePursuitCommand, 96
 TurnDegreesCommand, 154
 TurnToHeadingCommand, 155
operator<<
 Pose2d, 95
 Rotation2d, 104
 Transform2d, 143
 Translation2d, 150
 Twist2d, 158
operator+
 Pose2d, 92
 Rotation2d, 100
 Translation2d, 146
operator-
 Pose2d, 92
 Rotation2d, 101
 Transform2d, 141
 Translation2d, 146
operator/
 Pose2d, 92
 Rotation2d, 101
 Transform2d, 141
 Translation2d, 148
 Twist2d, 157
operator==
 Pose2d, 92
 Rotation2d, 102
 Transform2d, 142
 Translation2d, 148
 Twist2d, 158
operator*
 Pose2d, 91
 Rotation2d, 100
 Transform2d, 141
 Translation2d, 145, 146
 Twist2d, 157
P
 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 118
Page
 Flywheel, 35
Parallel, 79
Path
 PurePursuit::Path, 79
PID, 80
 config, 85
 ERROR_TYPE, 81
 get, 82
 get_error, 82
 get_sensor_val, 82
 get_target, 82
 init, 83

is_on_target, 84
PID, 81
reset, 84
set_limits, 84
set_target, 84
update, 85
PID::pid_config_t, 86
error_method, 86
on_target_time, 86
PIDPage
screen::PIDPage, 87
Pose2d, 88
exp, 91
log, 91
operator<<, 95
operator+, 92
operator-, 92
operator/, 92
operator==, 92
operator*, 91
Pose2d, 89, 90
relative_to, 93
rotation, 93
setRotationDeg, 93
setRotationRad, 93
setX, 93
setY, 94
transform_by, 94
translation, 94
x, 94
y, 94
position
CustomEncoder, 19
predict
SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 118
pure_pursuit
TankDrive, 132
PurePursuit::hermite_point, 43
PurePursuit::Path, 79
get_points, 80
get_radius, 80
is_valid, 80
Path, 79
PurePursuit::spline, 113
PurePursuitCommand, 95
on_timeout, 96
PurePursuitCommand, 95
run, 96
radians
Rotation2d, 102
radius
tracking_wheel_cfg_t, 137
Rect, 96
relative_to
Pose2d, 93
reset
PID, 84
SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 119
reset_auto
TankDrive, 133
revolutions
Rotation2d, 102
robot_specs_t, 96
drive_correction_cutoff, 97
rotate_around
Translation2d, 148
rotate_by
Translation2d, 149
rotation
CustomEncoder, 20
Pose2d, 93
Transform2d, 142
Rotation2d, 97
degrees, 99
f_cos, 99
f_sin, 100
f_tan, 100
operator<<, 104
operator+, 100
operator-, 101
operator/, 101
operator==, 102
operator*, 100
radians, 102
revolutions, 102
Rotation2d, 98, 99
rotation_matrix, 102
setDeg, 103
setRad, 103
wrapped_degrees_180, 103
wrapped_degrees_360, 103
wrapped_radians_180, 103
wrapped_radians_360, 104
wrapped_revolutions_180, 104
wrapped_revolutions_360, 104
rotation_matrix
Rotation2d, 102
run
BasicSolenoidSet, 11
BasicSpinCommand, 12
BasicStopCommand, 13
CommandController, 18
DelayCommand, 22
DriveForwardCommand, 23
DriveStopCommand, 24
DriveToPointCommand, 25
FlywheelStopCommand, 38
FlywheelStopMotorsCommand, 39
GenericAuto, 43
OdomSetPosition, 77
PurePursuitCommand, 96
SpinRPMCommand, 113
TurnDegreesCommand, 154
TurnToHeadingCommand, 155

WaitUntilUpToSpeedCommand, 159
S
 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 119
 save_to_disk
 Serializer, 109
 ScaledSphericalSimplexSigmaPoints
 ScaledSphericalSimplexSigmaPoints< STATES >, 105
 ScaledSphericalSimplexSigmaPoints< STATES >, 105
 num_sigmas, 106
 ScaledSphericalSimplexSigmaPoints, 105
 square_root_sigma_points, 106
 Wc, 106
 Wm, 107
 screen::ButtonWidget, 14
 ButtonWidget, 14, 15
 update, 15
 screen::FunctionPage, 40
 draw, 41
 FunctionPage, 41
 update, 41
 screen::OdometryPage, 72
 draw, 73
 OdometryPage, 73
 update, 73
 screen::Page, 78
 draw, 78
 update, 78
 screen::PIDPage, 87
 draw, 88
 PIDPage, 87
 update, 88
 screen::ScreenData, 107
 screen::SliderWidget, 111
 SliderWidget, 112
 update, 112
 screen::StatsPage, 123
 draw, 124
 StatsPage, 123
 update, 124
 send_message
 StateMachine< System, IDType, Message, delay_ms, do_log >, 122
 Serializer, 107
 bool_or, 108
 double_or, 109
 int_or, 109
 save_to_disk, 109
 Serializer, 108
 set_bool, 110
 set_double, 110
 set_int, 110
 set_string, 110
 string_or, 111
 set_accel
 TrapezoidProfile, 152
 set_async

 Lift< T >, 47
 set_bool
 Serializer, 110
 set_double
 Serializer, 110
 set_endpts
 TrapezoidProfile, 152
 set_int
 Serializer, 110
 set_limits
 Feedback, 30
 MotionController, 60
 PID, 84
 TakeBackHalf, 126
 set_max_v
 TrapezoidProfile, 152
 set_P
 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 119
 set_position
 Lift< T >, 48
 OdometryBase, 70
 OdometryTank, 76
 set_S
 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 120
 set_sensor_function
 Lift< T >, 48
 set_sensor_reset
 Lift< T >, 48
 set_setpoint
 Lift< T >, 48
 set_string
 Serializer, 110
 set_target
 PID, 84
 set_xhat
 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 120
 setDeg
 Rotation2d, 103
 setPosition
 CustomEncoder, 20
 setRad
 Rotation2d, 103
 setRotation
 CustomEncoder, 20
 setRotationDeg
 Pose2d, 93
 setRotationRad
 Pose2d, 93
 setX
 Pose2d, 93
 Translation2d, 149
 setY
 Pose2d, 94
 Translation2d, 149
 SliderWidget

screen::SliderWidget, 112
smallest_angle
 OdometryBase, 70
Smart
 TankDrive, 128
speed
 OdometryBase, 72
spin_manual
 Flywheel, 35
spin_rpm
 Flywheel, 36
SpinRpmCmd
 Flywheel, 36
SpinRPMCommand, 112
 run, 113
 SpinRPMCommand, 113
spinRPMTask
 Flywheel, 37
square_root_sigma_points
 ScaledSphericalSimplexSigmaPoints< STATES >, 106
SquareRootUnscentedKalmanFilter
 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 115, 116
SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 114
 correct, 117, 118
 P, 118
 predict, 118
 reset, 119
 S, 119
 set_P, 119
 set_S, 120
 set_xhat, 120
 SquareRootUnscentedKalmanFilter, 115, 116
 xhat, 120
StateMachine
 StateMachine< System, IDType, Message, delay_ms, do_log >, 122
StateMachine< System, IDType, Message, delay_ms, do_log >, 121
 current_state, 122
 send_message, 122
 StateMachine, 122
StateMachine< System, IDType, Message, delay_ms, do_log >::MaybeMessage, 53
 has_message, 54
 MaybeMessage, 53
 message, 54
StateMachine< System, IDType, Message, delay_ms, do_log >::State, 121
StatsPage
 screen::StatsPage, 123
stop
 Flywheel, 36
 TankDrive, 133
string_or
 Serializer, 111
TakeBackHalf, 124
get, 125
init, 125
is_on_target, 125
set_limits, 126
update, 126
TankDrive, 127
 BrakeType, 127
 drive_arcade, 128
 drive_forward, 128, 129
 drive_tank, 130
 drive_tank_raw, 130
 drive_to_point, 130, 131
 get_position, 132
 modify_inputs, 132
 None, 128
 pure_pursuit, 132
 reset_auto, 133
 Smart, 128
 stop, 133
 TankDrive, 128
 turn_degrees, 133, 134
 turn_to_heading, 135
 ZeroVelocity, 128
theta
 Translation2d, 149
theta_rad
 tracking_wheel_cfg_t, 137
tracking_wheel_cfg_t, 136
 radius, 137
 theta_rad, 137
 x, 137
 y, 137
Transform2d, 137
 inverse, 141
 operator<<, 143
 operator-, 141
 operator/, 141
 operator==, 142
 operator*, 141
 rotation, 142
 Transform2d, 138, 140
 translation, 142
 x, 142
 y, 142
transform_by
 Pose2d, 94
translation
 Pose2d, 94
 Transform2d, 142
Translation2d, 143
 as_vector, 145
 distance, 145
 norm, 145
 operator<<, 150
 operator+, 146
 operator-, 146
 operator/, 148

operator==, 148
 operator*, 145, 146
 rotate_around, 148
 rotate_by, 149
 setX, 149
 setY, 149
 theta, 149
 Translation2d, 144
 x, 149
 y, 150
 TrapezoidProfile, 150
 calculate, 151
 get_movement_time, 152
 set_accel, 152
 set_endpts, 152
 set_max_v, 152
 TrapezoidProfile, 151
 tune
 Odometry3Wheel, 65
 tune_feedforward
 MotionController, 61
 turn_degrees
 TankDrive, 133, 134
 turn_to_heading
 TankDrive, 135
 TurnDegreesCommand, 153
 on_timeout, 154
 run, 154
 TurnDegreesCommand, 153
 TurnToHeadingCommand, 154
 on_timeout, 155
 run, 155
 TurnToHeadingCommand, 154
 Twist2d, 155
 dtheta, 157
 dx, 157
 dy, 157
 operator<<, 158
 operator/, 157
 operator==, 158
 operator*, 157
 Twist2d, 156

 update
 Feedback, 30
 MotionController, 61
 Odometry3Wheel, 66
 OdometryBase, 71
 OdometryTank, 76
 PID, 85
 screen::ButtonWidget, 15
 screen::FunctionPage, 41
 screen::OdometryPage, 73
 screen::Page, 78
 screen::PIDPage, 88
 screen::SliderWidget, 112
 screen::StatsPage, 124
 TakeBackHalf, 126

velocity
 CustomEncoder, 20

WaitUntilCondition, 158
 WaitUntilUpToSpeedCmd
 Flywheel, 37
 WaitUntilUpToSpeedCommand, 159
 run, 159
 WaitUntilUpToSpeedCommand, 159

Wc
 ScaledSphericalSimplexSigmaPoints< STATES >, 106

wheel_diam
 Odometry3Wheel::odometry3wheel_cfg_t, 66

wheelbase_dist
 Odometry3Wheel::odometry3wheel_cfg_t, 67

Wm
 ScaledSphericalSimplexSigmaPoints< STATES >, 107

wrapped_degrees_180
 Rotation2d, 103

wrapped_degrees_360
 Rotation2d, 103

wrapped_radians_180
 Rotation2d, 103

wrapped_radians_360
 Rotation2d, 104

wrapped_revolutions_180
 Rotation2d, 104

wrapped_revolutions_360
 Rotation2d, 104

x
 Pose2d, 94
 tracking_wheel_cfg_t, 137
 Transform2d, 142
 Translation2d, 149

xhat
 SquareRootUnscentedKalmanFilter< STATES, INPUTS, OUTPUTS >, 120

y
 Pose2d, 94
 tracking_wheel_cfg_t, 137
 Transform2d, 142
 Translation2d, 150

ZeroVelocity
 TankDrive, 128