# RIT VEXU Core API

Generated by Doxygen 1.9.8

# Chapter 1

# Core

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

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

## 1.1 Getting Started

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

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

## 1.2 Features

Here is the current feature list this repo provides:

Subsystems (See `Wiki/Subsystems`):

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

Utilities (See `Wiki/Utilites`):

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

# Chapter 2

# Hierarchical Index

## 2.1 Class Hierarchy

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

# Chapter 3

# Class Index

## 3.1 Class List

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

# Chapter 4

# File Index

## 4.1 File List

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

# Chapter 5

# Class Documentation

## 5.1 AutoChooser Class Reference

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

**Classes**

- struct entry_t

**Public Member Functions**

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

**Protected Member Functions**

- void render (entry_t *selected)

**Protected Attributes**

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

### 5.1.1 Detailed Description

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

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 AutoChooser()

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

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

**Parameters**

| | |
|---|---|
| *brain* | the brain on which to draw the selection boxes |

### 5.1.3 Member Function Documentation

#### 5.1.3.1 add()

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

Add an auto path to the chooser

**Parameters**

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

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

#### 5.1.3.2 get_choice()

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

Get the currently selected auto choice

**Returns**

the identifier to the auto path

Return the selected autonomous

#### 5.1.3.3 render()

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

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

**Parameters**

| | |
|---|---|
| *selected* | the choice that is currently selected |

### 5.1.4 Member Data Documentation

#### 5.1.4.1 brain

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

the brain to show the choices on

#### 5.1.4.2 choice

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

the current choice of auto

#### 5.1.4.3 list

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

< a list of all possible auto choices

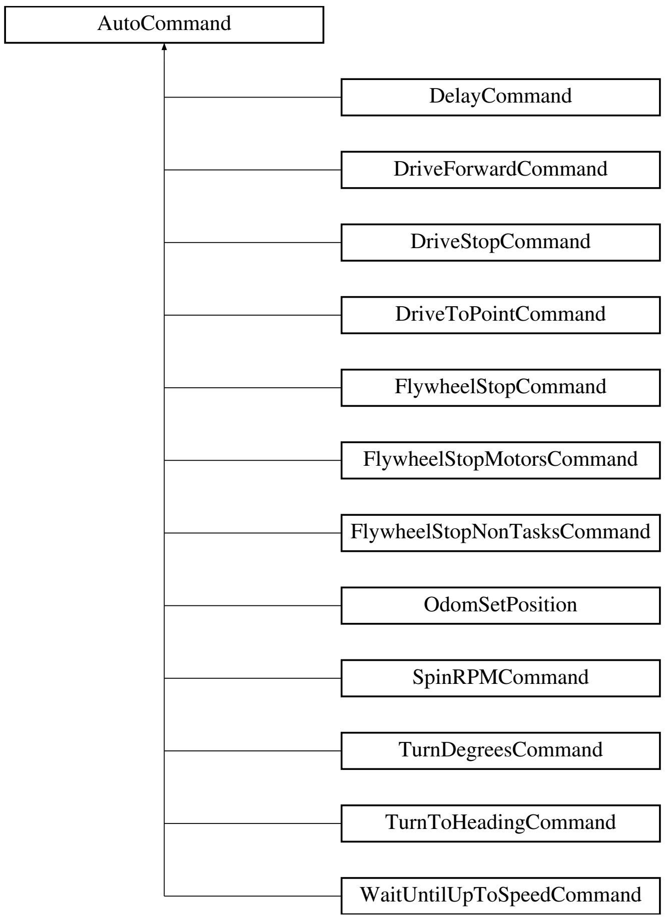
The documentation for this class was generated from the following files:

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

## 5.2 AutoCommand Class Reference

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

Inheritance diagram for AutoCommand:



**Public Member Functions**

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

**Public Attributes**

- double timeout_seconds = default_timeout

**Static Public Attributes**

- static constexpr double **default_timeout** = 10.0

### 5.2.1 Detailed Description

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

### 5.2.2 Member Function Documentation

#### 5.2.2.1 on_timeout()

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

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

Reimplemented in DriveForwardCommand, TurnDegreesCommand, TurnToHeadingCommand, and DriveStopCommand.

#### 5.2.2.2 run()

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

Executes the command Overridden by child classes

**Returns**

true when the command is finished, false otherwise

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

### 5.2.3 Member Data Documentation

#### 5.2.3.1 timeout_seconds

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

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

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

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

- include/utils/command_structure/auto_command.h

## 5.3 CommandController Class Reference

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

**Public Member Functions**

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

## 5.3.1 Detailed Description

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

## 5.3.2 Member Function Documentation

### 5.3.2.1 add() [1/3]

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

Adds a command to the queue

**Parameters**

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

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

**Parameters**

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

### 5.3.2.2 add() [2/3]

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

Add multiple commands to the queue. No timeout here.

**Parameters**

| | |
|---|---|
| *cmds* | the AutoCommands we want to add to our list |

#### 5.3.2.3 add() [3/3]

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

Add multiple commands to the queue. No timeout here.

**Parameters**

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

Add multiple commands to the queue. No timeout here.

**Parameters**

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

#### 5.3.2.4 add_delay()

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

Adds a command that will delay progression of the queue

**Parameters**

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

#### 5.3.2.5 last_command_timed_out()

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

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

**Returns**

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

#### 5.3.2.6 run()

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

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

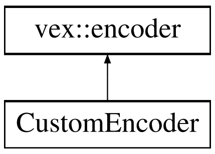
The documentation for this class was generated from the following files:

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

## 5.4 CustomEncoder Class Reference

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

Inheritance diagram for CustomEncoder:



**Public Member Functions**

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

### 5.4.1 Detailed Description

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

### 5.4.2 Constructor & Destructor Documentation

#### 5.4.2.1 CustomEncoder()

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

Construct an encoder with a custom number of ticks

**Parameters**

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

## 5.4.3 Member Function Documentation

### 5.4.3.1 position()

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

get the position that the encoder is at

**Parameters**

| | |
|---|---|
| *units* | the unit we want the return value to be in |

**Returns**

the position of the encoder in the units specified

### 5.4.3.2 rotation()

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

get the rotation that the encoder is at

**Parameters**

| | |
|---|---|
| *units* | the unit we want the return value to be in |

**Returns**

the rotation of the encoder in the units specified

### 5.4.3.3 setPosition()

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

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

**Parameters**

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

#### 5.4.3.4 setRotation()

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

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

**Parameters**

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

#### 5.4.3.5 velocity()

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

get the velocity that the encoder is moving at

**Parameters**

| | |
|---|---|
| *units* | the unit we want the return value to be in |

**Returns**

the velocity of the encoder in the units specified

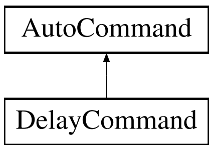
The documentation for this class was generated from the following files:

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

## 5.5 DelayCommand Class Reference

`#include <delay_command.h>`

Inheritance diagram for DelayCommand:



#### Public Member Functions

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

### Public Member Functions inherited from AutoCommand

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

#### Additional Inherited Members

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

## 5.5.1 Detailed Description

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

## 5.5.2 Constructor & Destructor Documentation

### 5.5.2.1 DelayCommand()

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

Construct a delay command

**Parameters**

| | |
|---|---|
| *ms* | the number of milliseconds to delay for |

## 5.5.3 Member Function Documentation

### 5.5.3.1 run()

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

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

**Returns**

true when complete

Reimplemented from AutoCommand.

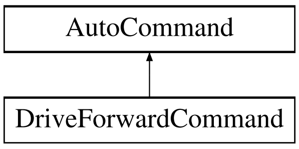
The documentation for this class was generated from the following file:

- include/utils/command_structure/delay_command.h

## 5.6 DriveForwardCommand Class Reference

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

Inheritance diagram for DriveForwardCommand:



**Public Member Functions**

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

### Public Member Functions inherited from AutoCommand

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

**Additional Inherited Members**

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.6.1 Detailed Description

AutoCommand wrapper class for the drive_forward function in the TankDrive class

## 5.6.2 Constructor & Destructor Documentation

### 5.6.2.1 DriveForwardCommand()

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

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

Currently includes:

- drive_forward
- turn_degrees
- drive_to_point
- turn_to_heading
- stop

Also holds AutoCommand subclasses that wrap OdometryBase functions

Currently includes:

- set_position Construct a DriveForward Command

**Parameters**

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

## 5.6.3 Member Function Documentation

### 5.6.3.1 on_timeout()

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

Cleans up drive system if we time out before finishing

reset the drive system if we timeout

Reimplemented from AutoCommand.

#### 5.6.3.2 run()

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

Run drive_forward Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

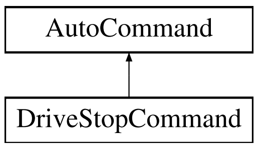
The documentation for this class was generated from the following files:

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

## 5.7 DriveStopCommand Class Reference

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

Inheritance diagram for DriveStopCommand:



**Public Member Functions**

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

### Public Member Functions inherited from AutoCommand

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

**Additional Inherited Members**

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.7.1 Detailed Description

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

### 5.7.2 Constructor & Destructor Documentation

#### 5.7.2.1 DriveStopCommand()

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

Construct a DriveStop Command

**Parameters**

| | |
|---|---|
| *drive_sys* | the drive system we are commanding |

### 5.7.3 Member Function Documentation

#### 5.7.3.1 on_timeout()

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

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

Reimplemented from AutoCommand.

#### 5.7.3.2 run()

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

Stop the drive system Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Stop the drive train Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

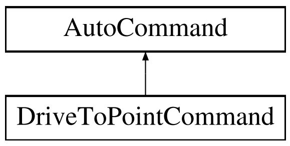
The documentation for this class was generated from the following files:

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

## 5.8 DriveToPointCommand Class Reference

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

Inheritance diagram for DriveToPointCommand:



#### Public Member Functions

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

### Public Member Functions inherited from AutoCommand

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

#### Additional Inherited Members

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.8.1 Detailed Description

AutoCommand wrapper class for the drive_to_point function in the TankDrive class

### 5.8.2 Constructor & Destructor Documentation

#### 5.8.2.1 DriveToPointCommand() [1/2]

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

Construct a DriveForward Command

**Parameters**

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

#### 5.8.2.2 DriveToPointCommand() [2/2]

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

Construct a DriveForward Command

**Parameters**

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

### 5.8.3 Member Function Documentation

#### 5.8.3.1 run()

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

Run drive_to_point Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

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

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

## 5.9 AutoChooser::entry_t Struct Reference

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

**Public Attributes**

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

### 5.9.1 Detailed Description

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

### 5.9.2 Member Data Documentation

#### 5.9.2.1 height

```
int AutoChooser::entry_t::height
```

height of the block

#### 5.9.2.2 name

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

name of the auto repretsented by the block

#### 5.9.2.3 width

```
int AutoChooser::entry_t::width
```

width of the block

#### 5.9.2.4 x

```
int AutoChooser::entry_t::x
```

screen x position of the block

#### 5.9.2.5 y

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

screen y position of the block

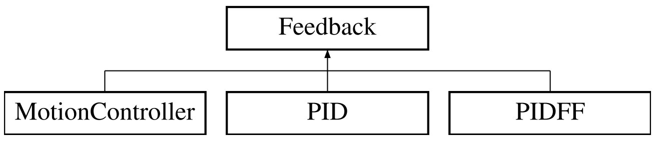
The documentation for this struct was generated from the following file:

- include/utils/auto_chooser.h

## 5.10 Feedback Class Reference

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

Inheritance diagram for Feedback:



#### Public Types

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

#### Public Member Functions

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

### 5.10.1 Detailed Description

Interface so that subsystems can easily switch between feedback loops

**Author**

Ryan McGee

**Date**

9/25/2022

## 5.10.2 Member Function Documentation

### 5.10.2.1 get()

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

**Returns**

the last saved result from the feedback controller

Implemented in MotionController, PID, and PIDFF.

### 5.10.2.2 init()

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

Initialize the feedback controller for a movement

**Parameters**

| | |
|---|---|
| *start↩_pt* | the current sensor value |
| *set_pt* | where the sensor value should be |

Implemented in MotionController, PID, and PIDFF.

### 5.10.2.3 is_on_target()

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

**Returns**

true if the feedback controller has reached it's setpoint

Implemented in MotionController, PID, and PIDFF.

### 5.10.2.4 set_limits()

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

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

**Parameters**

| | |
|---|---|
| *lower* | Upper limit |
| *upper* | Lower limit |

Implemented in MotionController, PID, and PIDFF.

#### 5.10.2.5 update()

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

Iterate the feedback loop once with an updated sensor value

**Parameters**

| | |
|---|---|
| *val* | value from the sensor |

**Returns**

feedback loop result

Implemented in MotionController, PID, and PIDFF.

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

- include/utils/feedback_base.h

## 5.11 FeedForward Class Reference

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

### Classes

- struct ff_config_t

### Public Member Functions

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

  *Perform the feedforward calculation.*

### 5.11.1 Detailed Description

FeedForward

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

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

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

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

**Author**

Ryan McGee

**Date**

6/13/2022

### 5.11.2 Constructor & Destructor Documentation

#### 5.11.2.1 FeedForward()

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

Creates a FeedForward object.

**Parameters**

| | |
|---|---|
| *cfg* | Configuration Struct for tuning |

### 5.11.3 Member Function Documentation

#### 5.11.3.1 calculate()

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

Perform the feedforward calculation.

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

**Parameters**

| | |
|---|---|
| *v* | Requested velocity of system |
| *a* | Requested acceleration of system |

**Returns**

A feedforward that should closely represent the system if tuned correctly

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

- include/utils/feedforward.h

## 5.12 FeedForward::ff_config_t Struct Reference

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

**Public Attributes**

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

### 5.12.1 Detailed Description

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

- kV ∗ desired velocity
- kA ∗ desired acceleration
- kG

### 5.12.2 Member Data Documentation

#### 5.12.2.1 kA

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

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

#### 5.12.2.2 kG

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

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

#### 5.12.2.3 kS

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

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

#### 5.12.2.4 kV

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

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

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

- include/utils/feedforward.h

## 5.13 Flywheel Class Reference

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

### Public Member Functions

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

### 5.13.1 Detailed Description

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

### 5.13.2 Constructor & Destructor Documentation

#### 5.13.2.1 Flywheel() [1/4]

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

Create the Flywheel object using PID + feedforward for control.

**Parameters**

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

Create the Flywheel object using PID + feedforward for control.

#### 5.13.2.2 Flywheel() [2/4]

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

Create the Flywheel object using only feedforward for control

**Parameters**

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

Create the Flywheel object using only feedforward for control

#### 5.13.2.3 Flywheel() [3/4]

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

Create the Flywheel object using Take Back Half for control

**Parameters**

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

Create the Flywheel object using Take Back Half for control

#### 5.13.2.4 Flywheel() [4/4]

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

Create the Flywheel object using Bang Bang for control

**Parameters**

| | |
|---|---|
| *motors* | the motors on the fly wheel |
| *ratio* | ratio of the whatever just multiplies the velocity |

Create the Flywheel object using Bang Bang for control

### 5.13.3 Member Function Documentation

#### 5.13.3.1 getDesiredRPM()

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

Return the RPM that the flywheel is currently trying to achieve

**Returns**

RPM the target rpm

Return the current value that the RPM should be set to

#### 5.13.3.2 getFeedforwardValue()

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

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

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

**Returns**

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

#### 5.13.3.3 getMotors()

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

Returns a POINTER to the motors

Returns a POINTER TO the motors; not currently used.

**Returns**

motorPointer -pointer to the motors

#### 5.13.3.4 getPID()

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

Returns a POINTER to the PID.

Returns a POINTER TO the PID; not currently used.

**Returns**

pidPointer -pointer to the PID

#### 5.13.3.5 getPIDValue()

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

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

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

**Returns**

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

#### 5.13.3.6 getRPM()

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

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

#### 5.13.3.7 getTBHGain()

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

get the gain used for TBH control

get the gain used for TBH control

**Returns**

the gain used in TBH control

#### 5.13.3.8 isTaskRunning()

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

Checks if the background RPM controlling task is running

**Returns**

true if the task is running

Checks if the background RPM controlling task is running

**Returns**

taskRunning - If the task is running

#### 5.13.3.9 measureRPM()

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

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

return the current velocity of the flywheel motors, in RPM

**Returns**

the measured velocity of the flywheel

#### 5.13.3.10 setPIDTarget()

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

Sets the value of the PID target

**Parameters**

| | |
|---|---|
| *value* | - desired value of the PID |

#### 5.13.3.11 spin_manual()

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

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

**Parameters**

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

#### 5.13.3.12 spin_raw()

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

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

**Parameters**

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

#### 5.13.3.13 spinRPM()

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

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

**Parameters**

| | |
|---|---|
| *rpm* | - the RPM we want to spin at |

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

**Parameters**

| | |
|---|---|
| *inputRPM* | - set the current RPM |

#### 5.13.3.14 stop()

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

stop the RPM thread and the wheel

#### 5.13.3.15 stopMotors()

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

stop only the motors; exclusively for BANG BANG use

#### 5.13.3.16 stopNonTasks()

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

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

#### 5.13.3.17 updatePID()

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

updates the value of the PID

**Parameters**

| | |
|---|---|
| *value* | - value to update the PID with |

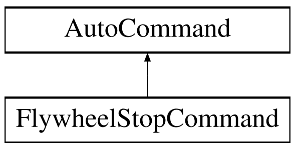
The documentation for this class was generated from the following files:

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

## 5.14 FlywheelStopCommand Class Reference

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

Inheritance diagram for FlywheelStopCommand:



### Public Member Functions

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

### Public Member Functions inherited from AutoCommand

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

**Additional Inherited Members**

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

## 5.14.1 Detailed Description

AutoCommand wrapper class for the stop function in the Flywheel class

## 5.14.2 Constructor & Destructor Documentation

### 5.14.2.1 FlywheelStopCommand()

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

Construct a FlywheelStopCommand

**Parameters**

| | |
|---|---|
| *flywheel* | the flywheel system we are commanding |

## 5.14.3 Member Function Documentation

### 5.14.3.1 run()

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

Run stop Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

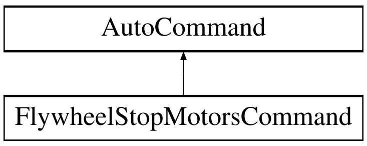
The documentation for this class was generated from the following files:

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

## 5.15 FlywheelStopMotorsCommand Class Reference

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

Inheritance diagram for FlywheelStopMotorsCommand:



#### Public Member Functions

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

### Public Member Functions inherited from AutoCommand

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

#### Additional Inherited Members

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.15.1 Detailed Description

AutoCommand wrapper class for the stopMotors function in the Flywheel class

### 5.15.2 Constructor & Destructor Documentation

#### 5.15.2.1 FlywheelStopMotorsCommand()

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

Construct a FlywheeStopMotors Command

**Parameters**

| | |
|---|---|
| *flywheel* | the flywheel system we are commanding |

### 5.15.3 Member Function Documentation

#### 5.15.3.1 run()

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

Run stop Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

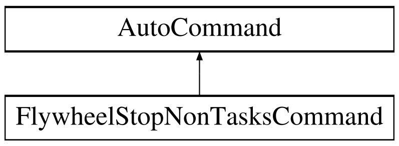
The documentation for this class was generated from the following files:

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

## 5.16 FlywheelStopNonTasksCommand Class Reference

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

Inheritance diagram for FlywheelStopNonTasksCommand:



**Additional Inherited Members**

### Public Member Functions inherited from AutoCommand

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

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.16.1 Detailed Description

AutoCommand wrapper class for the stopNonTasks function in the Flywheel class

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

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

## 5.17 GenericAuto Class Reference

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

#### Public Member Functions

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

### 5.17.1 Detailed Description

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

### 5.17.2 Member Function Documentation

#### 5.17.2.1 add()

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

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

**Parameters**

| | |
|---|---|
| *new_state* | the function to run |

#### 5.17.2.2 add_async()

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

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

**Parameters**

| | |
|---|---|
| *async_state* | the function to run |

#### 5.17.2.3 add_delay()

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

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

**Parameters**

| | |
|---|---|
| *ms* | how long to wait in milliseconds |

#### 5.17.2.4 run()

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

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

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

**Parameters**

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

**Returns**

true after all states have finished.

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

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

## 5.18 GraphDrawer Class Reference

### Public Member Functions

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

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

### 5.18.1 Constructor & Destructor Documentation

#### 5.18.1.1 GraphDrawer()

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

a helper class to graph values on the brain screen

Construct a GraphDrawer

**Parameters**

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

### 5.18.2 Member Function Documentation

#### 5.18.2.1 add_sample()

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

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

**Parameters**

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

#### 5.18.2.2 draw()

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

draws the graph to the screen in the constructor

**Parameters**

| | |
|---|---|
| *x* | x position of the top left of the graphed region |
| *y* | y position of the top left of the graphed region |
| *width* | the width of the graphed region |
| *height* | the height of the graphed region |

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

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

## 5.19 PurePursuit::hermite_point Struct Reference

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

### Public Member Functions

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

### Public Attributes

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

### 5.19.1 Detailed Description

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

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

- include/utils/pure_pursuit.h

## 5.20 Lift< T > Class Template Reference

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

#### Classes

- struct lift_cfg_t

#### Public Member Functions

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

### 5.20.1 Detailed Description

**template<typename T>**
**class Lift< T >**

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

**Author**

Ryan McGee

### 5.20.2 Constructor & Destructor Documentation

#### 5.20.2.1 Lift()

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

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

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

**Parameters**

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

### 5.20.3 Member Function Documentation

#### 5.20.3.1 control_continuous()

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

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

**Parameters**

| | |
|---|---|
| *up_ctrl* | Button controlling the "UP" motion |
| *down_ctrl* | Button controlling the "DOWN" motion |

#### 5.20.3.2 control_manual()

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

Control the lift with manual controls (no holding voltage)

**Parameters**

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

#### 5.20.3.3 control_setpoints()

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

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

**Parameters**

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

#### 5.20.3.4 get_async()

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

**Returns**

whether or not the background thread is running the lift

#### 5.20.3.5 get_setpoint()

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

**Returns**

The current setpoint for the lift

#### 5.20.3.6 hold()

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

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

#### 5.20.3.7 home()

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

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

#### 5.20.3.8 set_async()

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

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

**Parameters**

| | |
|---|---|
| *val* | Whether or not the background thread should run the lift |

#### 5.20.3.9 set_position()

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

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

**Parameters**

| | |
|---|---|
| *pos* | A lift position enum type |

**Returns**

True if the pid has reached the setpoint

#### 5.20.3.10 set_sensor_function()

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

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

**Parameters**

| | |
|---|---|
| *fn_ptr* | Pointer to custom sensor function |

#### 5.20.3.11 set_sensor_reset()

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

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

#### 5.20.3.12 set_setpoint()

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

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

**Parameters**

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

**Returns**

True if the pid has reached the setpoint

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

- include/subsystems/lift.h

## 5.21 Lift< T >::lift_cfg_t Struct Reference

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

**Public Attributes**

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

### 5.21.1 Detailed Description

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

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

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

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

- include/subsystems/lift.h

## 5.22 Logger Class Reference

Class to simplify writing to files.

```
#include <logger.h>
```

### Public Member Functions

- Logger (const std::string &filename)

  *Create a logger that will save to a file.*
- **Logger** (const Logger &l)=delete

  *copying not allowed*
- Logger & **operator=** (const Logger &l)=delete

  *copying not allowed*
- void Log (const std::string &s)

  *Write a string to the log.*
- void Log (LogLevel level, const std::string &s)

  *Write a string to the log with a loglevel.*
- void 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.*

### Public Attributes

- const int **MAX_FORMAT_LEN** = 512

  *maximum size for a string to be before it's written*

### 5.22.1 Detailed Description

Class to simplify writing to files.

### 5.22.2 Constructor & Destructor Documentation

#### 5.22.2.1 Logger()

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

Create a logger that will save to a file.

**Parameters**

| | |
|---|---|
| *filename* | the file to save to |

## 5.22.3 Member Function Documentation

### 5.22.3.1 Log() [1/2]

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

Write a string to the log.

**Parameters**

| | |
|---|---|
| *s* | the string to write |

### 5.22.3.2 Log() [2/2]

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

Write a string to the log with a loglevel.

**Parameters**

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

### 5.22.3.3 Logf() [1/2]

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

Write a formatted string to the log.

**Parameters**

| | |
|---|---|
| *fmt* | the format string (like printf) |
| ... | the args |

#### 5.22.3.4 Logf() [2/2]

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

Write a formatted string to the log with a loglevel.

**Parameters**

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

#### 5.22.3.5 Logln() [1/2]

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

Write a string and newline to the log.

**Parameters**

| | |
|---|---|
| *s* | the string to write |

#### 5.22.3.6 Logln() [2/2]

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

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

**Parameters**

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

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

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

## 5.23 MotionController::m_profile_cfg_t Struct Reference

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

### Public Attributes

- double **max_v**

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

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

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

    *configuration parameters for the internal*

### 5.23.1 Detailed Description

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

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

- include/utils/motion_controller.h

## 5.24 MecanumDrive Class Reference

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

### Classes

- struct mecanumdrive_config_t

### Public Member Functions

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

### 5.24.1 Detailed Description

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

### 5.24.2 Constructor & Destructor Documentation

#### 5.24.2.1 MecanumDrive()

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

Create the Mecanum drivetrain object

### 5.24.3 Member Function Documentation

#### 5.24.3.1 auto_drive()

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

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

**Parameters**

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

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

**Parameters**

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

**Returns**

Whether or not the maneuver is complete.

#### 5.24.3.2 auto_turn()

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

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

**Parameters**

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

**Returns**

whether or not the robot has finished the maneuver

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

**Parameters**

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

**Returns**

whether or not the robot has finished the maneuver

#### 5.24.3.3 drive()

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

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

**Parameters**

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

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

**Parameters**

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

#### 5.24.3.4 drive_raw()

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

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

**Parameters**

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

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

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

## 5.25 MecanumDrive::mecanumdrive_config_t Struct Reference

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

**Public Attributes**

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

### 5.25.1 Detailed Description

Configure the Mecanum drive PID tunings and robot configurations

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

- include/subsystems/mecanum_drive.h

## 5.26 motion_t Struct Reference

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

**Public Attributes**

- double **pos**

  *1d position at this point in time*
- double **vel**

  *1d velocity at this point in time*
- double **accel**

  *1d acceleration at this point in time*

### 5.26.1 Detailed Description

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

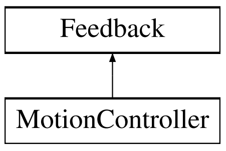
The documentation for this struct was generated from the following file:

- include/utils/trapezoid_profile.h

## 5.27 MotionController Class Reference

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

Inheritance diagram for MotionController:



#### Classes

- struct m_profile_cfg_t

#### Public Member Functions

- MotionController (m_profile_cfg_t &config)

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

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

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

### Public Member Functions inherited from Feedback

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

#### Static Public Member Functions

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

#### Additional Inherited Members

### Public Types inherited from Feedback

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

### 5.27.1 Detailed Description

Motion Controller class

This class defines a top-level motion profile, which can act as an intermediate between a subsystem class and the motors themselves

This takes the constants kS, kV, kA, kP, kI, kD, max_v and acceleration and wraps around a feedforward, PID and trapezoid profile. It does so with the following formula:

out = feedfoward.calculate(motion_profile.get(time_s)) + pid.get(motion_profile.get(time_s))

For PID and Feedforward specific formulae, see pid.h, feedforward.h, and trapezoid_profile.h

**Author**

Ryan McGee

**Date**

7/13/2022

## 5.27.2 Constructor & Destructor Documentation

### 5.27.2.1 MotionController()

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

Construct a new Motion Controller object.

**Parameters**

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

### 5.27.3 Member Function Documentation

#### 5.27.3.1 get()

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

**Returns**

the last saved result from the feedback controller

Implements Feedback.

#### 5.27.3.2 get_motion()

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

**Returns**

The current postion, velocity and acceleration setpoints

#### 5.27.3.3 init()

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

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

**Parameters**

| | |
|---|---|
| *start←_pt* | Movement starting position |
| *end_pt* | Movement ending posiiton |

Implements Feedback.

#### 5.27.3.4 is_on_target()

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

**Returns**

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

Implements Feedback.

### 5.27.3.5 set_limits()

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

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

**Parameters**

| | |
|---|---|
| *lower* | upper limit |
| *upper* | lower limiet |

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

**Parameters**

| | |
|---|---|
| *lower* | Upper limit |
| *upper* | Lower limit |

Implements Feedback.

### 5.27.3.6 tune_feedforward()

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

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

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

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

**Parameters**

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

**Returns**

A tuned feedforward object

#### 5.27.3.7 update()

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

Update the motion profile with a new sensor value.

**Parameters**

| | |
|---|---|
| *sensor_val* | Value from the sensor |

**Returns**

the motor input generated from the motion profile

Implements Feedback.

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

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

## 5.28 MovingAverage Class Reference

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

**Public Member Functions**

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

### 5.28.1 Detailed Description

MovingAverage

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

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

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

### 5.28.2 Constructor & Destructor Documentation

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

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

Create a moving average calculator with 0 as the default value

**Parameters**

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

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

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

Create a moving average calculator with a specified default value

**Parameters**

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

### 5.28.3 Member Function Documentation

#### 5.28.3.1 add_entry()

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

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

**Parameters**

| | |
|---|---|
| *n* | the sample that will be added to the moving average. |

#### 5.28.3.2 get_average()

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

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

**Returns**

the calculated average. sum(samples)/numsamples

How many samples the average is made from

**Returns**

the number of samples used to calculate this average

#### 5.28.3.3 get_size()

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

How many samples the average is made from

**Returns**

the number of samples used to calculate this average

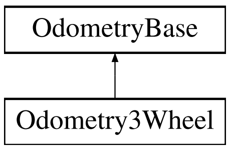
The documentation for this class was generated from the following files:

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

## 5.29 Odometry3Wheel Class Reference

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

Inheritance diagram for Odometry3Wheel:



### Classes

- struct odometry3wheel_cfg_t

### Public Member Functions

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

## Public Member Functions inherited from OdometryBase

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

**Additional Inherited Members**

## Static Public Member Functions inherited from OdometryBase

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

## Public Attributes inherited from OdometryBase

- bool **end_task** = false

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

## Static Public Attributes inherited from OdometryBase

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

## Protected Attributes inherited from OdometryBase

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

### 5.29.1 Detailed Description

Odometry3Wheel

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

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

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

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

**Author**

Ryan McGee

**Date**

Oct 31 2022

## 5.29.2 Constructor & Destructor Documentation

### 5.29.2.1 Odometry3Wheel()

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

Construct a new Odometry 3 Wheel object

**Parameters**

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

## 5.29.3 Member Function Documentation

### 5.29.3.1 tune()

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

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

**Parameters**

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

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

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

### 5.29.3.2 update()

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

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

**Returns**

the robot's updated position

Implements OdometryBase.

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

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

## 5.30 Odometry3Wheel::odometry3wheel_cfg_t Struct Reference

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

#### Public Attributes

- double wheelbase_dist
- double off_axis_center_dist
- double wheel_diam

### 5.30.1 Detailed Description

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

### 5.30.2 Member Data Documentation

#### 5.30.2.1 off_axis_center_dist

```
double Odometry3Wheel::odometry3wheel_cfg_t::off_axis_center_dist
```

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

#### 5.30.2.2 wheel_diam

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

the diameter of the tracking wheel

#### 5.30.2.3 wheelbase_dist

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

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

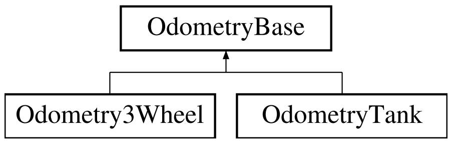
The documentation for this struct was generated from the following file:

- include/subsystems/odometry/odometry_3wheel.h

## 5.31 OdometryBase Class Reference

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

Inheritance diagram for OdometryBase:



### Public Member Functions

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

### Static Public Member Functions

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

### Public Attributes

- bool **end_task** = false

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

### Static Public Attributes

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

### Protected Attributes

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

### 5.31.1 Detailed Description

OdometryBase

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

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

**Author**

Ryan McGee

**Date**

Aug 11 2021

### 5.31.2 Constructor & Destructor Documentation

#### 5.31.2.1 OdometryBase()

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

Construct a new Odometry Base object

**Parameters**

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

### 5.31.3 Member Function Documentation

#### 5.31.3.1 background_task()

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

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

**Parameters**

| | |
|---|---|
| *ptr* | Pointer to OdometryBase object |

**Returns**

Required integer return code. Unused.

#### 5.31.3.2 end_async()

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

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

#### 5.31.3.3 get_accel()

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

Get the current acceleration

**Returns**

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

#### 5.31.3.4 get_angular_accel_deg()

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

Get the current angular acceleration in degrees

**Returns**

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

#### 5.31.3.5 get_angular_speed_deg()

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

Get the current angular speed in degrees

**Returns**

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

#### 5.31.3.6 get_position()

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

Gets the current position and rotation

**Returns**

the position that the odometry believes the robot is at

Gets the current position and rotation

#### 5.31.3.7 get_speed()

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

Get the current speed

**Returns**

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

#### 5.31.3.8 pos_diff()

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

Get the distance between two points

**Parameters**

| | |
|---|---|
| *start_pos* | distance from this point |
| *end_pos* | to this point |

**Returns**

the euclidean distance between start_pos and end_pos

#### 5.31.3.9 rot_diff()

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

Get the change in rotation between two points

**Parameters**

| | |
|---|---|
| *pos1* | position with initial rotation |
| *pos2* | position with final rotation |

**Returns**

change in rotation between pos1 and pos2

Get the change in rotation between two points

#### 5.31.3.10 set_position()

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

Sets the current position of the robot

**Parameters**

| | |
|---|---|
| *newpos* | the new position that the odometry will believe it is at |

Sets the current position of the robot

Reimplemented in OdometryTank.

#### 5.31.3.11 smallest_angle()

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

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

**Parameters**

| | |
|---|---|
| *start_deg* | intitial angle (degrees) |
| *end_deg* | final angle (degrees) |

**Returns**

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

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

#### 5.31.3.12 update()

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

Update the current position on the field based on the sensors

**Returns**

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

Implemented in Odometry3Wheel, and OdometryTank.

### 5.31.4 Member Data Documentation

#### 5.31.4.1 accel

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

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

#### 5.31.4.2 ang_accel_deg

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

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

#### 5.31.4.3 ang_speed_deg

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

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

#### 5.31.4.4 current_pos

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

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

#### 5.31.4.5 handle

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

handle to the vex task that is running the odometry code

#### 5.31.4.6 mut

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

Mutex to control multithreading

#### 5.31.4.7 speed

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

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

#### 5.31.4.8 zero_pos

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

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

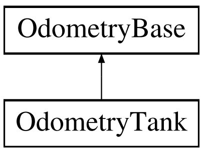
The documentation for this class was generated from the following files:

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

## 5.32 OdometryTank Class Reference

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

Inheritance diagram for OdometryTank:



#### Public Member Functions

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

### Public Member Functions inherited from OdometryBase

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

#### Additional Inherited Members

### Static Public Member Functions inherited from OdometryBase

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

## Public Attributes inherited from OdometryBase

- bool **end_task** = false

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

## Static Public Attributes inherited from OdometryBase

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

## Protected Attributes inherited from OdometryBase

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

## 5.32.1 Detailed Description

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

## 5.32.2 Constructor & Destructor Documentation

### 5.32.2.1 OdometryTank() `[1/2]`

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

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

**Parameters**

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

#### 5.32.2.2 OdometryTank() [2/2]

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

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

**Parameters**

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

### 5.32.3 Member Function Documentation

#### 5.32.3.1 set_position()

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

set_position tells the odometry to place itself at a position

**Parameters**

| | |
|---|---|
| *newpos* | the position the odometry will take |

Resets the position and rotational data to the input.

Reimplemented from OdometryBase.

#### 5.32.3.2 update()

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

Update the current position on the field based on the sensors

**Returns**

the position that odometry has calculated itself to be at

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

Implements OdometryBase.

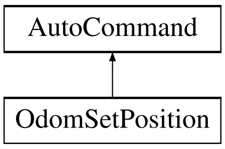
The documentation for this class was generated from the following files:

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

## 5.33 OdomSetPosition Class Reference

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

Inheritance diagram for OdomSetPosition:



#### Public Member Functions

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

### Public Member Functions inherited from AutoCommand

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

#### Additional Inherited Members

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.33.1 Detailed Description

AutoCommand wrapper class for the set_position function in the Odometry class

### 5.33.2 Constructor & Destructor Documentation

#### 5.33.2.1 OdomSetPosition()

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

constructs a new OdomSetPosition command

**Parameters**

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

Construct an Odometry set pos

**Parameters**

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

### 5.33.3 Member Function Documentation

#### 5.33.3.1 run()

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

Run set_position Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

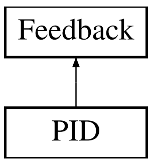
The documentation for this class was generated from the following files:

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

## 5.34 PID Class Reference

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

Inheritance diagram for PID:



### Classes

- struct pid_config_t

**Public Types**

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

**Public Types inherited from Feedback**

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

**Public Member Functions**

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

**Public Attributes**

- pid_config_t & **config**

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

## 5.34.1 Detailed Description

PID Class

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

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

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

**Author**

Ryan McGee

**Date**

4/3/2020

## 5.34.2 Member Enumeration Documentation

### 5.34.2.1 ERROR_TYPE

```
enum PID::ERROR_TYPE
```

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

## 5.34.3 Constructor & Destructor Documentation

### 5.34.3.1 PID()

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

Create the PID object

**Parameters**

| | |
|---|---|
| *config* | the configuration data for this controller |

Create the PID object

## 5.34.4 Member Function Documentation

### 5.34.4.1 get()

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

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

**Returns**

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

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

Implements Feedback.

### 5.34.4.2 get_error()

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

Get the delta between the current sensor data and the target

**Returns**

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

Get the delta between the current sensor data and the target

### 5.34.4.3 get_target()

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

Get the PID's target

**Returns**

the target the PID controller is trying to achieve

### 5.34.4.4 get_type()

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

Reimplemented from Feedback.

### 5.34.4.5 init()

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

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

start_pt can be safely ignored in this feedback controller

**Parameters**

| | |
|---|---|
| *start←_pt* | commpletely ignored for PID. necessary to satisfy Feedback base |
| *set_pt* | sets the target of the PID controller |

Implements Feedback.

#### 5.34.4.6 is_on_target()

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

Checks if the PID controller is on target.

**Returns**

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

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

Implements Feedback.

#### 5.34.4.7 reset()

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

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

#### 5.34.4.8 set_limits()

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

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

**Parameters**

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

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

Implements Feedback.

#### 5.34.4.9 set_target()

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

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

**Parameters**

| | |
|---|---|
| *target* | the sensor reading we would like to achieve |

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

#### 5.34.4.10 update()

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

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

**Parameters**

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

**Returns**

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

Implements Feedback.

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

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

## 5.35 PID::pid_config_t Struct Reference

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

**Public Attributes**

- double **p**

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

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

*derivitave coeffecient d ∗ derivative(error)*
- double **deadband**

  *at what threshold are we close enough to be finished*
- double **on_target_time**

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

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

### 5.35.1 Detailed Description

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

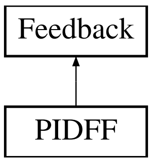
- deadband -
- on_target_time - for how long do we have to be at the target to stop As well, pid_config_t holds an error type which determines whether errors should be calculated as if the sensor position is a measure of distance or an angle

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

- include/utils/pid.h

## 5.36 PIDFF Class Reference

Inheritance diagram for PIDFF:



#### Public Member Functions

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

### Public Member Functions inherited from Feedback

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

### Public Attributes

- PID **pid**

### Additional Inherited Members

## Public Types inherited from Feedback

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

## 5.36.1 Member Function Documentation

### 5.36.1.1 get()

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

**Returns**

the last saved result from the feedback controller

Implements Feedback.

### 5.36.1.2 init()

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

Initialize the feedback controller for a movement

**Parameters**

| | |
|---|---|
| *start←↩ _pt* | the current sensor value |
| *set_pt* | where the sensor value should be |

Implements Feedback.

### 5.36.1.3 is_on_target()

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

**Returns**

true if the feedback controller has reached it's setpoint

Implements Feedback.

#### 5.36.1.4 set_limits()

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

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

**Parameters**

| | |
|---|---|
| *lower* | Upper limit |
| *upper* | Lower limit |

Implements Feedback.

#### 5.36.1.5 set_target()

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

Set the target of the PID loop

**Parameters**

| | |
|---|---|
| *set←_pt* | Setpoint / target value |

#### 5.36.1.6 update() `[1/2]`

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

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

**Parameters**

| | |
|---|---|
| *val* | value from the sensor |

**Returns**

feedback loop result

Implements Feedback.

#### 5.36.1.7 update() `[2/2]`

```
double PIDFF::update (
             double val,
             double vel_setpt,
             double a_setpt = 0 )
```

Iterate the feedback loop once with an updated sensor value

**Parameters**

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

**Returns**

feedback loop result

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

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

## 5.37 point_t Struct Reference

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

### Public Member Functions

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

### Public Attributes

- double **x**

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

  *the y position in space*

### 5.37.1 Detailed Description

Data structure representing an X,Y coordinate

### 5.37.2 Member Function Documentation

#### 5.37.2.1 dist()

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

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

**Parameters**

| | |
|---|---|
| *other* | the point to measure the distance from |

**Returns**

the euclidian distance between this and other

### 5.37.2.2 operator+()

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

Vector2D addition operation on points

**Parameters**

| | |
|---|---|
| *other* | the point to add on to this |

**Returns**

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

### 5.37.2.3 operator-()

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

Vector2D subtraction operation on points

**Parameters**

| | |
|---|---|
| *other* | the point_t to subtract from this |

**Returns**

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

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

- include/utils/geometry.h

## 5.38 pose_t Struct Reference

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

**Public Attributes**

- double **x**

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

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

  *rotation in the world*

### 5.38.1 Detailed Description

Describes a single position and rotation

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

- include/utils/geometry.h

## 5.39 robot_specs_t Struct Reference

```
#include <robot_specs.h>
```

**Public Attributes**

- double **robot_radius**

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

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

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

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

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

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

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

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

### 5.39.1 Detailed Description

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

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

- include/robot_specs.h

## 5.40 Serializer Class Reference

Serializes Arbitrary data to a file on the SD Card.

```
#include <serializer.h>
```

#### Public Member Functions

- ∼**Serializer** ()

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

  *create a Serializer*
- void save_to_disk () const

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

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

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

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

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

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

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

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

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

### 5.40.1 Detailed Description

Serializes Arbitrary data to a file on the SD Card.

### 5.40.2 Constructor & Destructor Documentation

#### 5.40.2.1 Serializer()

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

create a Serializer

**Parameters**

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

### 5.40.3 Member Function Documentation

#### 5.40.3.1 bool_or()

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

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

**Parameters**

| | |
|---|---|
| *name* | name of value |
| *otherwise* | value if the name is not specified |

**Returns**

the value if found or otherwise

#### 5.40.3.2 double_or()

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

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

**Parameters**

| | |
|---|---|
| *name* | name of value |
| *otherwise* | value if the name is not specified |

**Returns**

the value if found or otherwise

#### 5.40.3.3 int_or()

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

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

Getters Return value if it exists in the serializer

**Parameters**

| | |
|---|---|
| *name* | name of value |
| *otherwise* | value if the name is not specified |

**Returns**

the value if found or otherwise

### 5.40.3.4 save_to_disk()

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

saves current Serializer state to disk

forms data bytes then saves to filename this was openned with

### 5.40.3.5 set_bool()

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

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

**Parameters**

| | |
|---|---|
| *name* | name of bool |
| *b* | value of bool |

### 5.40.3.6 set_double()

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

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

**Parameters**

| | |
|---|---|
| *name* | name of double |
| *d* | value of double |

### 5.40.3.7 set_int()

```
void Serializer::set_int (
            const std::string & name,
            int i )
```

Setters - not saved until save_to_disk is called.

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

**Parameters**

| | |
|---|---|
| *name* | name of integer |
| *i* | value of integer |

### 5.40.3.8 set_string()

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

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

**Parameters**

| | |
|---|---|
| *name* | name of string |
| *i* | value of string |

### 5.40.3.9 string_or()

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

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

**Parameters**

| | |
|---|---|
| *name* | name of value |
| *otherwise* | value if the name is not specified |

**Returns**

the value if found or otherwise

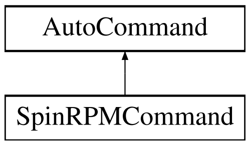
The documentation for this class was generated from the following files:

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

## 5.41 SpinRPMCommand Class Reference

`#include <flywheel_commands.h>`

Inheritance diagram for SpinRPMCommand:



**Public Member Functions**

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

### Public Member Functions inherited from AutoCommand

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

**Additional Inherited Members**

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.41.1 Detailed Description

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

### 5.41.2 Constructor & Destructor Documentation

#### 5.41.2.1 SpinRPMCommand()

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

Construct a SpinRPM Command

**Parameters**

| | |
|---|---|
| *flywheel* | the flywheel sys to command |
| *rpm* | the rpm that we should spin at |

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

### 5.41.3 Member Function Documentation

#### 5.41.3.1 run()

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

Run spin_manual Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

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

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

## 5.42 PurePursuit::spline Struct Reference

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

### Public Member Functions

- double **getY** (double x)

### Public Attributes

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

### 5.42.1 Detailed Description

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

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

- include/utils/pure_pursuit.h

## 5.43 TankDrive Class Reference

```
#include <tank_drive.h>
```

#### Public Member Functions

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

#### Static Public Member Functions

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

### 5.43.1 Detailed Description

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

### 5.43.2 Constructor & Destructor Documentation

#### 5.43.2.1 TankDrive()

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

Create the TankDrive object

**Parameters**

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

## 5.43.3 Member Function Documentation

### 5.43.3.1 drive_arcade()

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

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

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

**Parameters**

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

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

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

### 5.43.3.2 drive_forward() [1/2]

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

Autonomously drive the robot forward a certain distance

**Parameters**

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

Autonomously drive the robot forward a certain distance

**Parameters**

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

**Returns**

true if we have finished driving to our point

#### 5.43.3.3 drive_forward() [2/2]

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

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

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

**Parameters**

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

**Returns**

true when we have reached our target distance

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

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

**Parameters**

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

#### 5.43.3.4 drive_tank()

```
void TankDrive::drive_tank (
             double left,
             double right,
             int power = 1,
             bool isdriver = false )
```

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

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

**Parameters**

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

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

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

### 5.43.3.5 drive_to_point() [1/2]

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

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

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

**Parameters**

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

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

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

**Parameters**

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

**Returns**

true if we have reached our target point

#### 5.43.3.6 drive_to_point() [2/2]

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

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

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

**Parameters**

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

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

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

**Parameters**

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

**Returns**

true if we have reached our target point

#### 5.43.3.7 modify_inputs()

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

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

**Parameters**

| | |
|---|---|
| *input* | the input before modification |
| *power* | the power to raise input to |

**Returns**

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

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

**Parameters**

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

**Returns**

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

#### 5.43.3.8 pure_pursuit()

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

Follow a hermite curve using the pure pursuit algorithm.

**Parameters**

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

**Returns**

true when we reach the end of the path

#### 5.43.3.9 reset_auto()

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

Reset the initialization for autonomous drive functions

#### 5.43.3.10 stop()

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

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

#### 5.43.3.11 turn_degrees() [1/2]

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

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

Uses the defualt turning feedback of the drive system.

**Parameters**

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

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

Uses the defualt turning feedback of the drive system.

**Parameters**

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

**Returns**

true if we turned te target number of degrees

#### 5.43.3.12 turn_degrees() [2/2]

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

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

Uses PID + Feedforward for it's control.

**Parameters**

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

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

Uses the specified feedback for it's control.

**Parameters**

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

**Returns**

true if we have turned our target number of degrees

#### 5.43.3.13 turn_to_heading() `[1/2]`

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

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

**Parameters**

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

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

**Parameters**

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

**Returns**

true if we have reached our target heading

### 5.43.3.14 turn_to_heading() [2/2]

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

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

**Parameters**

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

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

**Parameters**

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

**Returns**

true if we have reached our target heading

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

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

## 5.44 TrapezoidProfile Class Reference

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

### Public Member Functions

- TrapezoidProfile (double max_v, double accel)

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

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

### 5.44.1 Detailed Description

Trapezoid Profile

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

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

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

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

**Author**

Ryan McGee

**Date**

7/12/2022

### 5.44.2 Constructor & Destructor Documentation

#### 5.44.2.1 TrapezoidProfile()

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

Construct a new Trapezoid Profile object.

**Parameters**

| | |
|---|---|
| *max←_v* | Maximum velocity the robot can run at |
| *accel* | Maximum acceleration of the robot |

### 5.44.3 Member Function Documentation

#### 5.44.3.1 calculate()

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

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

**Parameters**

| | |
|---|---|
| *time↩ _s* | Time since start of movement |

**Returns**

motion_t Position, velocity and acceleration

#### 5.44.3.2 get_movement_time()

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

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

**Returns**

the time the path will take to travel

#### 5.44.3.3 set_accel()

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

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

**Parameters**

| | |
|---|---|
| *accel* | the acceleration amount to use |

#### 5.44.3.4 set_endpts()

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

set_endpts defines a start and end position

**Parameters**

| | |
|---|---|
| *start* | the starting position of the path |
| *end* | the ending position of the path |

#### 5.44.3.5 set_max_v()

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

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

**Parameters**

| | |
|---|---|
| *max←_v* | the maximum velocity the robot can travel at |

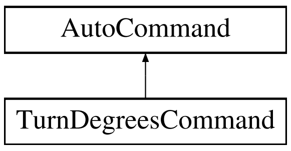
The documentation for this class was generated from the following files:

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

## 5.45 TurnDegreesCommand Class Reference

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

Inheritance diagram for TurnDegreesCommand:



### Public Member Functions

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

### Public Member Functions inherited from AutoCommand

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

### Additional Inherited Members

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

## Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.45.1 Detailed Description

AutoCommand wrapper class for the turn_degrees function in the TankDrive class

### 5.45.2 Constructor & Destructor Documentation

#### 5.45.2.1 TurnDegreesCommand()

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

Construct a TurnDegreesCommand Command

**Parameters**

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

### 5.45.3 Member Function Documentation

#### 5.45.3.1 on_timeout()

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

Cleans up drive system if we time out before finishing

reset the drive system if we timeout

Reimplemented from AutoCommand.

#### 5.45.3.2 run()

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

Run turn_degrees Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

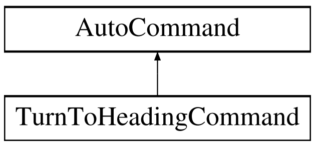
The documentation for this class was generated from the following files:

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

## 5.46 TurnToHeadingCommand Class Reference

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

Inheritance diagram for TurnToHeadingCommand:



#### Public Member Functions

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

### Public Member Functions inherited from AutoCommand

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

#### Additional Inherited Members

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

### 5.46.1 Detailed Description

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

### 5.46.2 Constructor & Destructor Documentation

#### 5.46.2.1 TurnToHeadingCommand()

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

Construct a TurnToHeadingCommand Command

**Parameters**

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

## 5.46.3 Member Function Documentation

### 5.46.3.1 on_timeout()

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

Cleans up drive system if we time out before finishing

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

Reimplemented from AutoCommand.

### 5.46.3.2 run()

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

Run turn_to_heading Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

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

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

## 5.47 Vector2D Class Reference

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

**Public Member Functions**

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

## 5.47.1 Detailed Description

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

## 5.47.2 Constructor & Destructor Documentation

### 5.47.2.1 Vector2D() `[1/2]`

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

Construct a vector object.

**Parameters**

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

### 5.47.2.2 Vector2D() `[2/2]`

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

Construct a vector object from a cartesian point.

**Parameters**

| | |
|---|---|
| *p* | point_t.x , point_t.y |

### 5.47.3 Member Function Documentation

#### 5.47.3.1 get_dir()

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

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

Use r2d() to convert.

**Returns**

the direction of the vetctor in radians

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

Use r2d() to convert.

#### 5.47.3.2 get_mag()

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

**Returns**

the magnitude of the vector

Get the magnitude of the vector

#### 5.47.3.3 get_x()

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

**Returns**

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

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

#### 5.47.3.4 get_y()

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

**Returns**

the Y component of the vector, positive forward.

Get the Y component of the vector, positive forward.

### 5.47.3.5 normalize()

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

Changes the magnitude of the vector to 1

**Returns**

the normalized vector

Changes the magnetude of the vector to 1

### 5.47.3.6 operator*()

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

Scales a Vector2D by a scalar with the * operator

**Parameters**

| | |
|---|---|
| *x* | the value to scale the vector by |

**Returns**

the this Vector2D scaled by x

### 5.47.3.7 operator+()

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

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

**Parameters**

| | |
|---|---|
| *other* | the vector to add to this |

**Returns**

the sum of the vectors

### 5.47.3.8 operator-()

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

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

**Parameters**

| | |
|---|---|
| *other* | the vector to subtract from this |

**Returns**

the difference of the vectors

### 5.47.3.9 point()

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

Returns a point from the vector

**Returns**

the point represented by the vector

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

**Returns**

the x, y representation of the vector

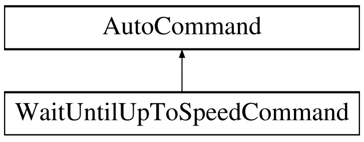
The documentation for this class was generated from the following files:

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

## 5.48 WaitUntilUpToSpeedCommand Class Reference

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

Inheritance diagram for WaitUntilUpToSpeedCommand:



**Public Member Functions**

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

### Public Member Functions inherited from AutoCommand

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

#### Additional Inherited Members

### Public Attributes inherited from AutoCommand

- double timeout_seconds = default_timeout

### Static Public Attributes inherited from AutoCommand

- static constexpr double **default_timeout** = 10.0

## 5.48.1 Detailed Description

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

## 5.48.2 Constructor & Destructor Documentation

### 5.48.2.1 WaitUntilUpToSpeedCommand()

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

Creat a WaitUntilUpToSpeedCommand

**Parameters**

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

## 5.48.3 Member Function Documentation

### 5.48.3.1 run()

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

Run spin_manual Overrides run from AutoCommand

**Returns**

true when execution is complete, false otherwise

Reimplemented from AutoCommand.

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

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

# Chapter 6

# File Documentation

## 6.1 robot_specs.h

```
#pragma once
#include "../core/include/utils/pid.h"
#include "../core/include/utils/feedback_base.h"

typedef struct
{
  double robot_radius;

  double odom_wheel_diam;
  double odom_gear_ratio;
  double dist_between_wheels;

  double drive_correction_cutoff;

  Feedback *drive_feedback;
  Feedback *turn_feedback;
  PID::pid_config_t correction_pid;

} robot_specs_t;
```

## 6.2 custom_encoder.h

```
#pragma once
#include "vex.h"

class CustomEncoder : public vex::encoder
{
  typedef vex::encoder super;

  public:
  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);


  private:
  double tick_scalar;
};
```

## 6.3 flywheel.h

```
#pragma once
/*********************************************************
*
*     File:     Flywheel.h
*     Purpose:  Generalized flywheel class for Core.
*     Author:   Chris Nokes
*
**********************************************************
* EDIT HISTORY
**********************************************************
* 09/23/2022  <CRN> Reorganized, added documentation.
* 09/23/2022  <CRN> Added functions elaborated on in .cpp.
*********************************************************/
#include "../core/include/utils/feedforward.h"
#include "vex.h"
#include "../core/include/robot_specs.h"
#include "../core/include/utils/pid.h"
#include <atomic>

using namespace vex;

class Flywheel{
  enum FlywheelControlStyle{
    PID_Feedforward,
    Feedforward,
    Take_Back_Half,
    Bang_Bang,
  };
  public:

  // CONSTRUCTORS, GETTERS, AND SETTERS
  Flywheel(motor_group &motors, PID::pid_config_t &pid_config, FeedForward::ff_config_t &ff_config,
const double ratio);

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

  Flywheel(motor_group &motors, double tbh_gain, const double ratio);

  Flywheel(motor_group &motors, const double ratio);

  double getDesiredRPM();

  bool isTaskRunning();

  motor_group* getMotors();

  double measureRPM();

  double getRPM();
  PID* getPID();

  double getPIDValue();

  double getFeedforwardValue();

  double getTBHGain();

  void setPIDTarget(double value);

  void updatePID(double value);

  // SPINNERS AND STOPPERS

  void spin_raw(double speed, directionType dir=fwd);

  void spin_manual(double speed, directionType dir=fwd);

  void spinRPM(int rpm);

  void stop();


  void stopMotors();

  void stopNonTasks();

  private:

  motor_group &motors;                  // motors that make up the flywheel
  bool taskRunning = false;             // is the task (thread but not) currently running?
  PID pid;                              // PID on the flywheel
  FeedForward ff;                       // FF constants for the flywheel
  double TBH_gain;                      // TBH gain parameter for the flywheel
  double ratio;                         // multiplies the velocity by this value
  std::atomic<double> RPM;              // Desired RPM of the flywheel.
  task rpmTask;                        // task (thread but not) that handles spinning the wheel at a given RPM
  FlywheelControlStyle control_style; // how the flywheel should be controlled
  double smoothedRPM;
  MovingAverage RPM_avger;
  };
```

## 6.4 lift.h

```
#pragma once

#include "vex.h"
#include "../core/include/utils/pid.h"
#include <iostream>
#include <map>
#include <atomic>
#include <vector>

using namespace vex;
using namespace std;

template <typename T>
class Lift
{
  public:

  struct lift_cfg_t
  {
    double up_speed, down_speed;
    double softstop_up, softstop_down;

    PID::pid_config_t lift_pid_cfg;
  };

  Lift(motor_group &lift_motors, lift_cfg_t &lift_cfg, map<T, double> &setpoint_map, limit *homing_switch=NULL)
  : lift_motors(lift_motors), cfg(lift_cfg), lift_pid(cfg.lift_pid_cfg), setpoint_map(setpoint_map), homing_switch(homing_switch)
  {

    is_async = true;
    setpoint = 0;

    // Create a background task that is constantly updating the lift PID, if requested.
    // Set once, and forget.
    task t([](void* ptr){
      Lift &lift = *((Lift*) ptr);

      while(true)
      {
        if(lift.get_async())
          lift.hold();

        vexDelay(50);
      }

      return 0;
    }, this);

  }

  void control_continuous(bool up_ctrl, bool down_ctrl)
  {
    static timer tmr;

    double cur_pos = 0;

    // Check if there’s a hook for a custom sensor. If not, use the motors.
    if(get_sensor == NULL)
      cur_pos = lift_motors.position(rev);
    else
      cur_pos = get_sensor();

    if(up_ctrl && cur_pos < cfg.softstop_up)
    {
      lift_motors.spin(directionType::fwd, cfg.up_speed, volt);
      setpoint = cur_pos + .3;

      // std::cout « "DEBUG OUT: UP " « setpoint « ", " « tmr.time(sec) « ", " « cfg.down_speed « "\n";

      // Disable the PID while going UP.
      is_async = false;
    } else if(down_ctrl && cur_pos > cfg.softstop_down)
    {
      // Lower the lift slowly, at a rate defined by down_speed
      if(setpoint > cfg.softstop_down)
        setpoint = setpoint - (tmr.time(sec) * cfg.down_speed);
      // std::cout « "DEBUG OUT: DOWN " « setpoint « ", " « tmr.time(sec) « ", " « cfg.down_speed «
"\n";
      is_async = true;
    } else
    {
      // Hold the lift at the last setpoint
      is_async = true;
    }

    tmr.reset();
  }

  void control_manual(bool up_btn, bool down_btn, int volt_up, int volt_down)
  {
    static bool down_hold = false;
    static bool init = true;

    // Allow for setting position while still calling this function
    if(init || up_btn || down_btn)
    {
      init = false;
      is_async = false;
    }

    double rev = lift_motors.position(rotationUnits::rev);

    if(rev < cfg.softstop_down && down_btn)
      down_hold = true;
    else if( !down_btn )
      down_hold = false;

    if(up_btn && rev < cfg.softstop_up)
      lift_motors.spin(directionType::fwd, volt_up, voltageUnits::volt);
    else if(down_btn && rev > cfg.softstop_down && !down_hold)
      lift_motors.spin(directionType::rev, volt_down, voltageUnits::volt);
    else
      lift_motors.spin(directionType::fwd, 0, voltageUnits::volt);

  }

  void control_setpoints(bool up_step, bool down_step, vector<T> pos_list)
  {
    // Make sure inputs are only processed on the rising edge of the button
    static bool up_last = up_step, down_last = down_step;

    bool up_rising = up_step && !up_last;
    bool down_rising = down_step && !down_last;

    up_last = up_step;
    down_last = down_step;

    static int cur_index = 0;

    // Avoid an index overflow. Shouldn't happen unless the user changes pos_list between calls.
    if(cur_index >= pos_list.size())
      cur_index = pos_list.size() - 1;

    // Increment or decrement the index of the list, bringing it up or down.
    if(up_rising && cur_index < (pos_list.size() - 1))
      cur_index++;
    else if(down_rising && cur_index > 0)
      cur_index--;

    // Set the lift to hold the position in the background with the PID loop
    set_position(pos_list[cur_index]);
    is_async = true;

  }

  bool set_position(T pos)
  {
    this->setpoint = setpoint_map[pos];
    is_async = true;

    return (lift_pid.get_target() == this->setpoint) && lift_pid.is_on_target();
  }

  bool set_setpoint(double val)
  {
    this->setpoint = val;
    return (lift_pid.get_target() == this->setpoint) && lift_pid.is_on_target();
  }

  double get_setpoint()
  {
    return this->setpoint;
  }

  void hold()
  {
    lift_pid.set_target(setpoint);
    // std::cout « "DEBUG OUT: SETPOINT " « setpoint « "\n";

    if(get_sensor != NULL)
      lift_pid.update(get_sensor());
    else
      lift_pid.update(lift_motors.position(rev));

    // std::cout « "DEBUG OUT: ROTATION " « lift_motors.rotation(rev) « "\n\n";

    lift_motors.spin(fwd, lift_pid.get(), volt);
  }

  void home()
  {
    static timer tmr;
    tmr.reset();

    while(tmr.time(sec) < 3)
    {
      lift_motors.spin(directionType::rev, 6, volt);

      if (homing_switch == NULL && lift_motors.current(currentUnits::amp) > 1.5)
        break;
      else if (homing_switch != NULL && homing_switch->pressing())
        break;
    }

    if(reset_sensor != NULL)
      reset_sensor();

    lift_motors.resetPosition();
    lift_motors.stop();

  }

  bool get_async()
  {
    return is_async;
  }

  void set_async(bool val)
  {
    this->is_async = val;
  }

  void set_sensor_function(double (*fn_ptr) (void))
  {
    this->get_sensor = fn_ptr;
  }

  void set_sensor_reset(void (*fn_ptr) (void))
  {
    this->reset_sensor = fn_ptr;
  }

  private:

  motor_group &lift_motors;
  lift_cfg_t &cfg;
  PID lift_pid;
  map<T, double> &setpoint_map;
  limit *homing_switch;

  atomic<double> setpoint;
  atomic<bool> is_async;

  double (*get_sensor)(void) = NULL;
  void (*reset_sensor)(void) = NULL;


};
```

## 6.5 mecanum_drive.h

```
#pragma once
```

```

#include "vex.h"
#include "../core/include/utils/pid.h"

#ifndef PI
#define PI 3.141592654
#endif

class MecanumDrive
{

  public:

  struct mecanumdrive_config_t
  {
    // PID configurations for autonomous driving
    PID::pid_config_t drive_pid_conf;
    PID::pid_config_t drive_gyro_pid_conf;
    PID::pid_config_t turn_pid_conf;

    // Diameter of the mecanum wheels
    double drive_wheel_diam;

    // Diameter of the perpendicular undriven encoder wheel
    double lateral_wheel_diam;

    // Width between the center of the left and right wheels
    double wheelbase_width;

  };

  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);

  private:

  vex::motor &left_front, &right_front, &left_rear, &right_rear;

  mecanumdrive_config_t *config;
  vex::rotation *lateral_wheel;
  vex::inertial *imu;

  PID *drive_pid = NULL;
  PID *drive_gyro_pid = NULL;
  PID *turn_pid = NULL;

  bool init = true;

};
```

## 6.6 odometry_3wheel.h

```
#pragma once
#include "../core/include/subsystems/odometry/odometry_base.h"
#include "../core/include/subsystems/tank_drive.h"
#include "../core/include/subsystems/custom_encoder.h"

class Odometry3Wheel : public OdometryBase
{
    public:

    typedef struct
    {
        double wheelbase_dist;
        double off_axis_center_dist;
        double wheel_diam;
    } odometry3wheel_cfg_t;

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

    pose_t update() override;

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

    private:

    static pose_t calculate_new_pos(double lside_delta_deg, double rside_delta_deg, double
offax_delta_deg, pose_t old_pos, odometry3wheel_cfg_t cfg);

    CustomEncoder &lside_fwd, &rside_fwd, &off_axis;
    odometry3wheel_cfg_t &cfg;


};
```

## 6.7 odometry_base.h

```
#pragma once

#include "vex.h"
#include "../core/include/utils/geometry.h"
#include "../core/include/robot_specs.h"

#ifndef PI
#define PI 3.141592654
#endif



class OdometryBase
{
public:

    OdometryBase(bool is_async);

    pose_t get_position(void);

    virtual void set_position(const pose_t& newpos=zero_pos);

    virtual pose_t update() = 0;

    static int background_task(void* ptr);

    void end_async();

    static double pos_diff(pose_t start_pos, pose_t end_pos);

    static double rot_diff(pose_t pos1, pose_t pos2);

    static double smallest_angle(double start_deg, double end_deg);

    bool end_task = false;

    double get_speed();

    double get_accel();

    double get_angular_speed_deg();

    double get_angular_accel_deg();

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

protected:
    vex::task *handle;

    vex::mutex mut;

    pose_t current_pos;

    double speed;
    double accel;
    double ang_speed_deg;
    double ang_accel_deg;
};
```

## 6.8 odometry_tank.h

```
#pragma once

#include "../core/include/subsystems/odometry/odometry_base.h"
```

```
#include "../core/include/subsystems/custom_encoder.h"
#include "../core/include/utils/geometry.h"
#include "../core/include/utils/vector2d.h"
#include "../core/include/robot_specs.h"

static int background_task(void* odom_obj);


class OdometryTank : public OdometryBase
{
public:
    OdometryTank(vex::motor_group &left_side, vex::motor_group &right_side, robot_specs_t &config, vex::inertial *imu=NULL, bool is_async=true);

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

    pose_t update() override;

    void set_position(const pose_t &newpos=zero_pos) override;



private:
    static pose_t calculate_new_pos(robot_specs_t &config, pose_t &stored_info, double lside_diff, double rside_diff, double angle_deg);

    vex::motor_group *left_side, *right_side;
    CustomEncoder *left_enc, *right_enc;
    vex::inertial *imu;
    robot_specs_t &config;

    double rotation_offset = 0;

};
```

## 6.9 screen.h

```
#pragma once
#include "vex.h"
#include <vector>


typedef void (*screenFunc)(vex::brain::lcd &screen, int x, int y, int width, int height, bool first_run);

void draw_mot_header(vex::brain::lcd &screen, int x, int y, int width);
// name should be no longer than 15 characters
void draw_mot_stats(vex::brain::lcd &screen, int x, int y, int width, const char *name, vex::motor &motor, int animation_tick);
void draw_dev_stats(vex::brain::lcd &screen, int x, int y, int width, const char *name, vex::device &dev, int animation_tick);

void draw_battery_stats(vex::brain::lcd &screen, int x, int y, double voltage, double percentage);



void draw_lr_arrows(vex::brain::lcd &screen, int bar_width, int width, int height);

int handle_screen_thread(vex::brain::lcd &screen, std::vector<screenFunc> pages, int first_page);
void StartScreen(vex::brain::lcd &screen, std::vector<screenFunc> pages, int first_page = 0);
```

## 6.10 tank_drive.h

```
#pragma once

#ifndef PI
#define PI 3.141592654
#endif

#include "vex.h"
#include "../core/include/subsystems/odometry/odometry_tank.h"
#include "../core/include/utils/pid.h"
#include "../core/include/utils/feedback_base.h"
#include "../core/include/robot_specs.h"
#include "../core/src/utils/pure_pursuit.cpp"
#include <vector>


```

```
using namespace vex;

class TankDrive
{
public:

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

  void stop();

  void drive_tank(double left, double right, int power=1, bool isdriver=false);

  void drive_arcade(double forward_back, double left_right, int power=1);

  bool drive_forward(double inches, directionType dir, Feedback &feedback, double max_speed=1);

  bool drive_forward(double inches, directionType dir, double max_speed=1);

  bool turn_degrees(double degrees, Feedback &feedback, double max_speed=1);

  bool turn_degrees(double degrees, double max_speed=1);

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

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

  bool turn_to_heading(double heading_deg, Feedback &feedback, double max_speed=1);
  bool turn_to_heading(double heading_deg, double max_speed=1);

  void reset_auto();

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

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

private:
  motor_group &left_motors;
  motor_group &right_motors;

  PID correction_pid;
  Feedback *drive_default_feedback = NULL;
  Feedback *turn_default_feedback = NULL;

  OdometryBase *odometry;

  robot_specs_t &config;

  bool func_initialized = false;
  bool is_pure_pursuit = false;
};
```

## 6.11 auto_chooser.h

```
#pragma once
#include "vex.h"
#include <string>
#include <vector>


class AutoChooser
{
  public:
  AutoChooser(vex::brain &brain);

  void add(std::string name);

  std::string get_choice();

  protected:

  struct entry_t
  {
    int x;
    int y;
    int width;
    int height;
    std::string name;
  };

  void render(entry_t *selected);

  std::string choice;
  std::vector<entry_t> list ;
  vex::brain &brain;
};
```

## 6.12 auto_command.h

```

#pragma once

#include "vex.h"

class AutoCommand {
  public:
    static constexpr double default_timeout = 10.0;
    virtual bool run() { return true; }
    virtual void on_timeout(){}
    AutoCommand* withTimeout(double t_seconds){
      this->timeout_seconds = t_seconds;
      return this;
    }
    double timeout_seconds = default_timeout;

};
```

## 6.13 command_controller.h

```

#pragma once
#include <vector>
#include <queue>
#include "../core/include/utils/command_structure/auto_command.h"

class CommandController
{
public:
  void add(AutoCommand *cmd, double timeout_seconds = 10.0);

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

  void add(std::vector<AutoCommand *> cmds, double timeout_sec);
  void add_delay(int ms);

  void run();
  bool last_command_timed_out();

private:
  std::queue<AutoCommand *> command_queue;
  bool command_timed_out = false;
};
```

## 6.14 delay_command.h

```

#pragma once

#include "../core/include/utils/command_structure/auto_command.h"

class DelayCommand: public AutoCommand {
  public:
    DelayCommand(int ms): ms(ms) {}

    bool run() override {
      vexDelay(ms);
      return true;
    }

  private:
    // amount of milliseconds to wait
    int ms;
};
```

## 6.15 drive_commands.h

```
#pragma once

#include "vex.h"
#include "../core/include/utils/geometry.h"
#include "../core/include/utils/command_structure/auto_command.h"
#include "../core/include/subsystems/tank_drive.h"

using namespace vex;


// ==== DRIVING ====

class DriveForwardCommand: public AutoCommand {
  public:
    DriveForwardCommand(TankDrive &drive_sys, Feedback &feedback, double inches, directionType dir,
double max_speed=1);

    bool run() override;
    void on_timeout() override;

  private:
    // drive system to run the function on
    TankDrive &drive_sys;

    // feedback controller to use
    Feedback &feedback;

    // parameters for drive_forward
    double inches;
    directionType dir;
    double max_speed;
};

class TurnDegreesCommand: public AutoCommand {
  public:
    TurnDegreesCommand(TankDrive &drive_sys, Feedback &feedback, double degrees, double max_speed =
1);

    bool run() override;
    void on_timeout() override;


  private:
    // drive system to run the function on
    TankDrive &drive_sys;

    // feedback controller to use
    Feedback &feedback;

    // parameters for turn_degrees
    double degrees;
    double max_speed;
};

class DriveToPointCommand: public AutoCommand {
  public:
    DriveToPointCommand(TankDrive &drive_sys, Feedback &feedback, double x, double y, directionType
dir, double max_speed = 1);
    DriveToPointCommand(TankDrive &drive_sys, Feedback &feedback, point_t point, directionType dir,
double max_speed=1);

    bool run() override;

  private:
    // drive system to run the function on
    TankDrive &drive_sys;

    void on_timeout() override;


    // feedback controller to use
    Feedback &feedback;

    // parameters for drive_to_point
    double x;
    double y;
    directionType dir;
    double max_speed;

};

class TurnToHeadingCommand: public AutoCommand {
  public:
    TurnToHeadingCommand(TankDrive &drive_sys, Feedback &feedback, double heading_deg, double speed =
1);

    bool run() override;
    void on_timeout() override;


  private:
    // drive system to run the function on
    TankDrive &drive_sys;

    // feedback controller to use
    Feedback &feedback;

    // parameters for turn_to_heading
    double heading_deg;
    double max_speed;
};

class DriveStopCommand: public AutoCommand {
  public:
    DriveStopCommand(TankDrive &drive_sys);

    bool run() override;
    void on_timeout() override;

  private:
    // drive system to run the function on
    TankDrive &drive_sys;
};


// ==== ODOMETRY ====

class OdomSetPosition: public AutoCommand {
  public:
    OdomSetPosition(OdometryBase &odom, const pose_t &newpos=OdometryBase::zero_pos);

    bool run() override;

  private:
    // drive system with an odometry config
    OdometryBase &odom;
    pose_t newpos;
};
```

## 6.16 flywheel_commands.h

```

#pragma once

#include "../core/include/subsystems/flywheel.h"
#include "../core/include/utils/command_structure/auto_command.h"

class SpinRPMCommand: public AutoCommand {
  public:
  SpinRPMCommand(Flywheel &flywheel, int rpm);

    bool run() override;

  private:
    // Flywheel instance to run the function on
    Flywheel &flywheel;

    // parameters for spinRPM
    int rpm;
};

class WaitUntilUpToSpeedCommand: public AutoCommand {
  public:
    WaitUntilUpToSpeedCommand(Flywheel &flywheel, int threshold_rpm);

    bool run() override;

  private:
    // Flywheel instance to run the function on
    Flywheel &flywheel;

    // if the actual speed is equal to the desired speed +/- this value, we are ready to fire
    int threshold_rpm;
};

class FlywheelStopCommand: public AutoCommand {
  public:
```

```
  FlywheelStopCommand(Flywheel &flywheel);

    bool run() override;

  private:
    // Flywheel instance to run the function on
    Flywheel &flywheel;
};

class FlywheelStopMotorsCommand: public AutoCommand {
  public:
  FlywheelStopMotorsCommand(Flywheel &flywheel);

    bool run() override;

  private:
    // Flywheel instance to run the function on
    Flywheel &flywheel;
};

class FlywheelStopNonTasksCommand: public AutoCommand {
  FlywheelStopNonTasksCommand(Flywheel &flywheel);

    bool run() override;

  private:
    // Flywheel instance to run the function on
    Flywheel &flywheel;
};
```

## 6.17 feedback_base.h

```
#pragma once

class Feedback
{
public:
    enum FeedbackType
    {
        PIDType,
        FeedforwardType,
        OtherType,
    };

    virtual void init(double start_pt, double set_pt) = 0;

    virtual double update(double val) = 0;

    virtual double get() = 0;

    virtual void set_limits(double lower, double upper) = 0;

    virtual bool is_on_target() = 0;

    virtual Feedback::FeedbackType get_type()
    {
        return FeedbackType::OtherType;
    }
};
```

## 6.18 feedforward.h

```
#pragma once

#include <math.h>
#include <vector>
#include "../core/include/utils/math_util.h"
#include "../core/include/utils/moving_average.h"
#include "vex.h"

class FeedForward
{
    public:

    typedef struct
    {
        double kS;
        double kV;
        double kA;
        double kG;
    } ff_config_t;


    FeedForward(ff_config_t &cfg) : cfg(cfg) {}

    double calculate(double v, double a, double pid_ref=0.0)
    {
        double ks_sign = 0;
        if(v != 0)
            ks_sign = sign(v);
        else if(pid_ref != 0)
            ks_sign = sign(pid_ref);

        return (cfg.kS * ks_sign) + (cfg.kV * v) + (cfg.kA * a) + cfg.kG;
    }

    private:

    ff_config_t &cfg;

};


FeedForward::ff_config_t tune_feedforward(vex::motor_group &motor, double pct, double duration);
```

## 6.19 generic_auto.h

```
#pragma once

#include <queue>
#include <map>
#include "vex.h"
#include <functional>

typedef std::function<bool(void)> state_ptr;

class GenericAuto
{
  public:

  bool run(bool blocking);

  void add(state_ptr new_state);

  void add_async(state_ptr async_state);

  void add_delay(int ms);

  private:

  std::queue<state_ptr> state_list;

};
```

## 6.20 geometry.h

```
#pragma once
#include <cmath>

struct point_t
{
    double x;
    double y;

    double dist(const point_t other)
    {
        return std::sqrt(std::pow(this->x - other.x, 2) + pow(this->y - other.y, 2));
    }

    point_t operator+(const point_t &other)
    {
        point_t p{
            .x = this->x + other.x,
            .y = this->y + other.y};
        return p;
    }

    point_t operator-(const point_t &other)
    {
        point_t p{
            .x = this->x - other.x,
            .y = this->y - other.y};
        return p;
    }
};


typedef struct
{
    double x;
    double y;
    double rot;
} pose_t;
```

## 6.21 graph_drawer.h

```
#pragma once

#include <string>
#include <stdio.h>
#include <vector>
#include <cmath>
#include "vex.h"
#include "../core/include/utils/geometry.h"
#include "../core/include/utils/vector2d.h"

class GraphDrawer
{
public:
  GraphDrawer(vex::brain::lcd &screen, int num_samples, std::string x_label, std::string y_label,
vex::color col, bool draw_border, double lower_bound, double upper_bound);
  void add_sample(point_t sample);
  void draw(int x, int y, int width, int height);

private:
  vex::brain::lcd &Screen;
  std::vector<point_t> samples;
  int sample_index = 0;
  std::string xlabel;
  std::string ylabel;
  vex::color col = vex::red;
  vex::color bgcol = vex::transparent;
  bool border;
  double upper;
  double lower;
};
```

## 6.22 logger.h

```
#pragma once

#include <cstdarg>
#include <cstdio>
#include <string>
#include "vex.h"

enum LogLevel
{
    DEBUG,
    NOTICE,
    WARNING,
    ERROR,
    CRITICAL,
    TIME
};

class Logger
{
private:
    const std::string filename;
    vex::brain::sdcard sd;
    void write_level(LogLevel l);

public:
    const int MAX_FORMAT_LEN = 512;
    explicit Logger(const std::string &filename);

    Logger(const Logger &l) = delete;
    Logger &operator=(const Logger &l) = delete;


    void Log(const std::string &s);

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

    void Logln(const std::string &s);

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

    void Logf(const char *fmt, ...);

    void Logf(LogLevel level, const char *fmt, ...);
};
```

## 6.23 math_util.h

```
#pragma once
#include "math.h"
#include "vex.h"
#include <vector>

double clamp(double value, double low, double high);

double sign(double x);

double wrap_angle_deg(double input);
double wrap_angle_rad(double input);

/*
Calculates the variance of  a set of numbers (needed for linear regression)
https://en.wikipedia.org/wiki/Variance
@param values   the values for which the variance is taken
@param mean     the average of values
*/
double variance(std::vector<double> const &values, double mean);


/*
Calculates the average of a vector of doubles
@param values   the list of values for which the average is taken
*/
double mean(std::vector<double> const &values);

/*
Calculates the covariance of a set of points (needed for linear regression)
https://en.wikipedia.org/wiki/Covariance

@param points   the points for which the covariance is taken
@param meanx    the mean value of all x coordinates in points
@param meany    the mean value of all y coordinates in points
*/
double covariance(std::vector<std::pair<double, double» const &points, double meanx, double meany);

/*
Calculates the slope and y intercept of the line of best fit for the data
@param points the points for the data
*/
std::pair<double, double> calculate_linear_regression(std::vector<std::pair<double, double» const
&points);

```

## 6.24 motion_controller.h

```
#pragma once
#include "../core/include/utils/pid.h"
#include "../core/include/utils/feedforward.h"
#include "../core/include/utils/trapezoid_profile.h"
#include "../core/include/utils/feedback_base.h"
#include "../core/include/subsystems/tank_drive.h"
#include "vex.h"

class MotionController : public Feedback
{
    public:

    typedef struct
    {
        double max_v;
        double accel;
        PID::pid_config_t pid_cfg;
        FeedForward::ff_config_t ff_cfg;
    } m_profile_cfg_t;

    MotionController(m_profile_cfg_t &config);

    void init(double start_pt, double end_pt) override;

    double update(double sensor_val) override;

    double get() override;

    void set_limits(double lower, double upper) override;

    bool is_on_target() override;

    motion_t get_motion();

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

    private:

    m_profile_cfg_t config;

    PID pid;
    FeedForward ff;
    TrapezoidProfile profile;

    double lower_limit = 0, upper_limit = 0;
    double out = 0;
    motion_t cur_motion;

    vex::timer tmr;

};
```

## 6.25 moving_average.h

```
#include <vector>

class MovingAverage {
  public:
  /*
   * Create a moving average calculator with 0 as the default value
   *
   * @param buffer_size       The size of the buffer. The number of samples that constitute a valid reading
   */
  MovingAverage(int buffer_size);
  /*
   * Create a moving average calculator with a specified default value
   * @param buffer_size       The size of the buffer. The number of samples that constitute a valid reading
   * @param starting_value The value that the average will be before any data is added
   */
  MovingAverage(int buffer_size, double starting_value);

  /*
  * Add a reading to the buffer
  * Before:
  * [ 1 1 2 2 3 3] => 2
  *   ^
  * After:
  * [ 2 1 2 2 3 3] => 2.16
  *     ^
  * @param n  the sample that will be added to the moving average.
  */
  void add_entry(double n);

  double get_average();

  int get_size();


  private:
    int buffer_index;              //index of the next value to be overridden
    std::vector<double> buffer;    //all current data readings we've taken
    double current_avg;            //the current value of the data

};
```

## 6.26 pid.h

```
#pragma once

#include <cmath>
#include "vex.h"
#include "../core/include/utils/feedback_base.h"

using namespace vex;

class PID : public Feedback
{
public:
  enum ERROR_TYPE{
    LINEAR,
    ANGULAR // assumes degrees
  };
  struct pid_config_t
  {
    double p;
    double i;
    double d;
    double deadband;
    double on_target_time;
    ERROR_TYPE error_method;
  };



  PID(pid_config_t &config);


  void init(double start_pt, double set_pt) override;

  double update(double sensor_val) override;

  double get() override;

  void set_limits(double lower, double upper) override;

  bool is_on_target() override;

  void reset();

  double get_error();

  double get_target();

  void set_target(double target);

  Feedback::FeedbackType get_type() override;

  pid_config_t &config;

private:


  double last_error = 0;
  double accum_error = 0;

  double last_time = 0;
  double on_target_last_time = 0;

  double lower_limit = 0;
  double upper_limit = 0;

  double target = 0;
  double sensor_val = 0;
  double out = 0;

  bool is_checking_on_target = false;

  timer pid_timer;
};
```

## 6.27 pidff.h

```
#pragma once
#include "../core/include/utils/feedback_base.h"
#include "../core/include/utils/pid.h"
#include "../core/include/utils/feedforward.h"

class PIDFF : public Feedback
```

```
{
  public:

  PIDFF(PID::pid_config_t &pid_cfg, FeedForward::ff_config_t &ff_cfg);

  void init(double start_pt, double set_pt) override;

  void set_target(double set_pt);

  double update(double val) override;

  double update(double val, double vel_setpt, double a_setpt=0);

  double get() override;

  void set_limits(double lower, double upper) override;

  bool is_on_target() override;

  PID pid;


  private:

  FeedForward::ff_config_t &ff_cfg;

  FeedForward ff;

  double out;
  double lower_lim, upper_lim;

};
```

## 6.28 pure_pursuit.h

```
#pragma once

#include <vector>
#include "../core/include/utils/geometry.h"
#include "../core/include/utils/vector2d.h"
#include "vex.h"

using namespace vex;

namespace PurePursuit {
 struct spline
 {
  double a, b, c, d, x_start, x_end;

  double getY(double x) {
   return a * pow((x - x_start), 3) + b * pow((x - x_start), 2) + c * (x - x_start) + d;
  }
 };
 struct hermite_point
 {
  double x;
  double y;
  double dir;
  double mag;

  point_t getPoint() {
   return {x, y};
  }

  Vector2D getTangent() {
   return Vector2D(dir, mag);
  }
 };

 static std::vector<point_t> line_circle_intersections(point_t center, double r, point_t point1,
point_t point2);
 static point_t get_lookahead(std::vector<point_t> path, point_t robot_loc, double radius);

 static std::vector<point_t> inject_path(std::vector<point_t> path, double spacing);

 static std::vector<point_t> smooth_path(std::vector<point_t> path, double weight_data, double
weight_smooth, double tolerance);

 static std::vector<point_t> smooth_path_cubic(std::vector<point_t> path, double res);

 static std::vector<point_t> smooth_path_hermite(std::vector<hermite_point> path, double step);
}
```

## 6.29 serializer.h

```
#pragma once
#include <algorithm>
#include <map>
#include <string>
#include <vector>
#include <stdio.h>

const char serialization_separator = '$';
const std::size_t MAX_FILE_SIZE = 4096;

class Serializer
{
private:
    bool flush_always;
    std::string filename;
    std::map<std::string, int> ints;
    std::map<std::string, bool> bools;
    std::map<std::string, double> doubles;
    std::map<std::string, std::string> strings;

    bool read_from_disk();

public:
    ~Serializer()
    {
        save_to_disk();
        printf("Saving %s\n", filename.c_str());
        fflush(stdout);
    }

    explicit Serializer(const std::string &filename, bool flush_always = true) :
flush_always(flush_always), filename(filename), ints({}), bools({}), doubles({}), strings({}) {
read_from_disk(); }

    void save_to_disk() const;


    void set_int(const std::string &name, int i);

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

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

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


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

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

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

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

## 6.30 trapezoid_profile.h

```
#pragma once

typedef struct
{
    double pos;
    double vel;
    double accel;

} motion_t;

class TrapezoidProfile
{
    public:

    TrapezoidProfile(double max_v, double accel);

    motion_t calculate(double time_s);

    void set_endpts(double start, double end);

    void set_accel(double accel);

    void set_max_v(double max_v);

    double get_movement_time();

    private:
    double start, end;
    double max_v;
    double accel;
    double time;


};
```

## 6.31 vector2d.h

```
#pragma once


#include <cmath>
#include "../core/include/utils/geometry.h"

#ifndef PI
#define PI 3.141592654
#endif
class Vector2D
{
public:
    Vector2D(double dir, double mag);

    Vector2D(point_t p);

    double get_dir() const;

    double get_mag() const;

    double get_x() const;

    double get_y() const;

    Vector2D normalize();

    point_t point();

    Vector2D operator*(const double &x);
    Vector2D operator+(const Vector2D &other);
    Vector2D operator-(const Vector2D &other);

private:

    double dir, mag;

};

double deg2rad(double deg);

double rad2deg(double r);
```

# Index