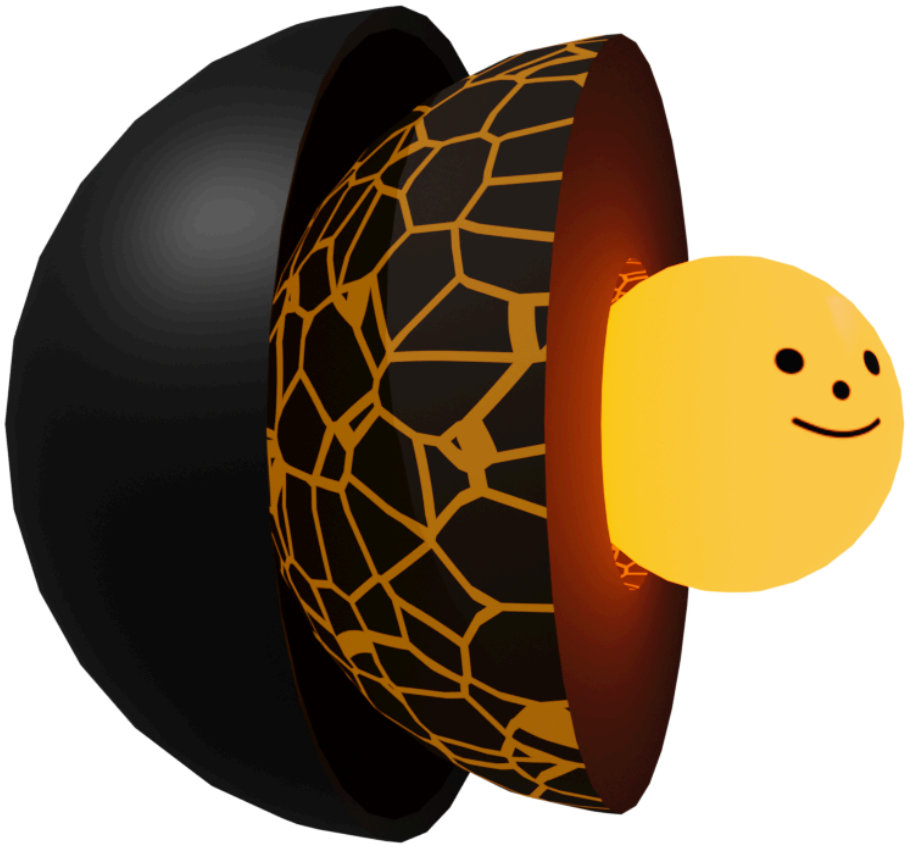
RIT VEXU Software Engineering Notebook

2023-2024

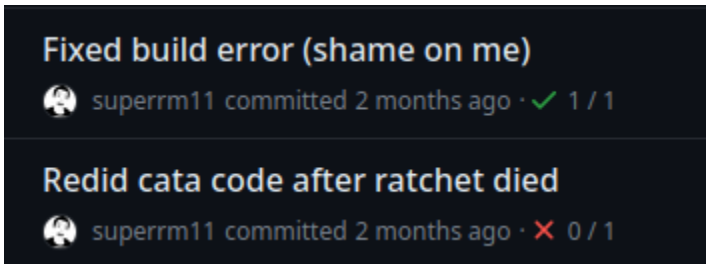


Software Development Process

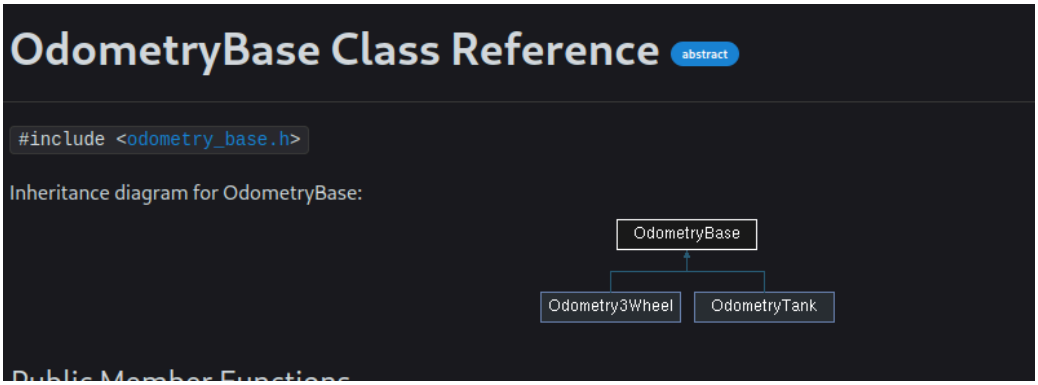
- Subrepo - why we use it, why tried to switch to submodules, and why we switched back
- Github project board
- Wiki
- ~~Clang-tidy~~
- ~~Github Actions~~

Github Actions

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



Continuous Integration directly improves the quality of our code.



Automatically generated documentation.

Auto-Notebook

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

Clang-Tidy

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

Core: Fundamentals

- Odometry
    - Tank (same old)
    - GPS + Odometry
- Drivetrain
    - Tank Drive
        - What it do
            - DriveForward, TurnToHeading
            - DriveToPoint
        - Whats new
            - Pure pursuit
            - Brake mode
- ~~Control systems~~
    - ~~What it do~~
        - ~~FeedbackBase (Swapping control loops)~~
        - ~~PID~~
        - ~~PIDFF~~
        - ~~Bang Bang?~~
        - ~~Take Back Half?~~
        - ~~Motion Profile~~
    - ~~Whats new~~
        - ~~Trapezoid with velocity and such~~
- ~~ACS~~
    - ~~What it do~~
        - ~~Organizing commands in orders, not having 1000 if statements~~
    - ~~Whats new~~
        - ~~Shortcuts~~
        - ~~Memory Management that isnt bad~~
        - Cyborg control
            - Using auto paths in teleop after we convince the drivers not to be luddites
- ~~Logger/Serializer~~
- Screen Stuff
    - Graphing
    - Auto Choosing
    - Odometrying
    - Motor stats
    - Pid Tuner
    - Widgets
    - Not redeploying code every 30 seconds

- Catapult Shenanigans
    - Sequencing, deploying
- Vision (to be written later)
    - Localizing nuts
    - Squirrel behavior
-

Control Loops

In order for the Autonomous Command Structure to function, we need a way to tell the robot how we want it to move. There are two broad categories of telling a robot to achieve a requested position - Feedback and Feedforward. Feedback relies on sensors and adjusts the output of the robot according to the error between where it is and where it wants to be. On the other hand, a feedforward controller takes a mathematical model of the system and creates outputs based on what it calculates to be the necessary output to achieve the goal. Additionally, there are simpler methods like Bang-Bang or Take Back Half. These adjust the outputs based on the current position relative to the target, where Take Back Half gradually refines the output until it settled at the desired position. These controller types work for many applications, but a combination of them can achieve an even better control over robot actuators.

PID

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

Feedforward

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

Bang-Bang

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

Take Back Half (TBH)

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

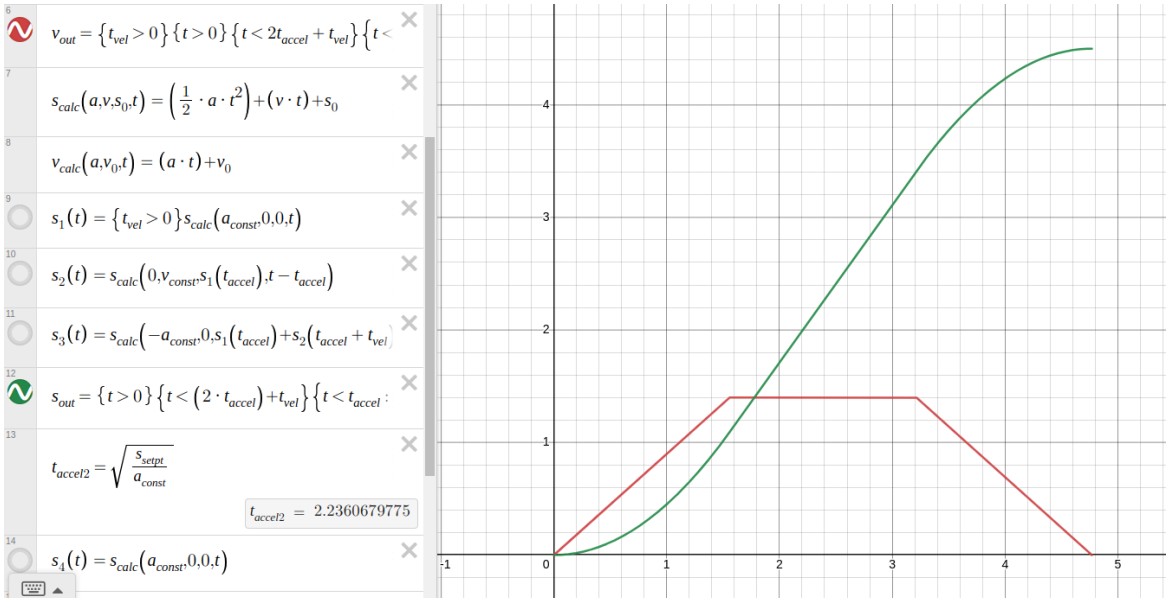
Generic Feedback

Different control systems work best in different environments. Because of this, we found ourselves switching control schemes often enough that rewriting the code each time was time consuming and often led to rushed, worse quality code. To solve this problem we implemented a generic feedback interface so that none of our subsystem code needs to change when we use a different control scheme. Instead, the subsystem reports to the controller where it wants to be, measurements from its environment and some information about the system’s capabilities and the controller will report back the actions needed to achieve that target. This allows for much faster prototyping and cleaner, less tightly coupled code.

Motion Profile

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

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



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

Auto Command Structure (ACS)

Principle

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

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

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

    return non_loader_side_auto;
}
```

ACS code from the 2023 competition season

Updates

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

The most hazardous, rather than simply annoying downside of last year's system, was the memory unsafety of this system. Since our auto commands must use virtual functions, they must be on the other end of a pointer. So, they must be allocated using new or they must be initialized statically before we write the path which is a terrible user experience (Though, if constrained by an embedded system where allocating on the heap was deemed dangerous, the system could work with this). This became a real issue when we began to write more complicated constructs such as branching, asynchronous, and repeated commands as it became dangerously unclear who was responsible for deallocating these objects. As a solution for this, we developed an RAII wrapper for the Auto Command Interface. Inspired by C++’s std::unique_ptr, this wrapper provides a memory safe, value based way of using auto

commands while still maintaining their adaptability. We used C++'s ideas of move semantics and 'Resource Allocation Is Initialization' to practically solve memory management so programmers (and even non programmers) can focus on writing paths.

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

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

ACS code going into the 2024 competition season

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

Serializer

One pain point we found last year was configuring auto paths, color targets, path timeouts, and other parameters that changed match to match but that also should be persistent. Commonly, we found ourselves redeploying code at the last minute before a match. To solve this, we wrote a class that takes control of a file on the SD card to which users can read and

write values at runtime. This keeps us from having to change a value, redeploy, repeat which cost us valuable time in the past.

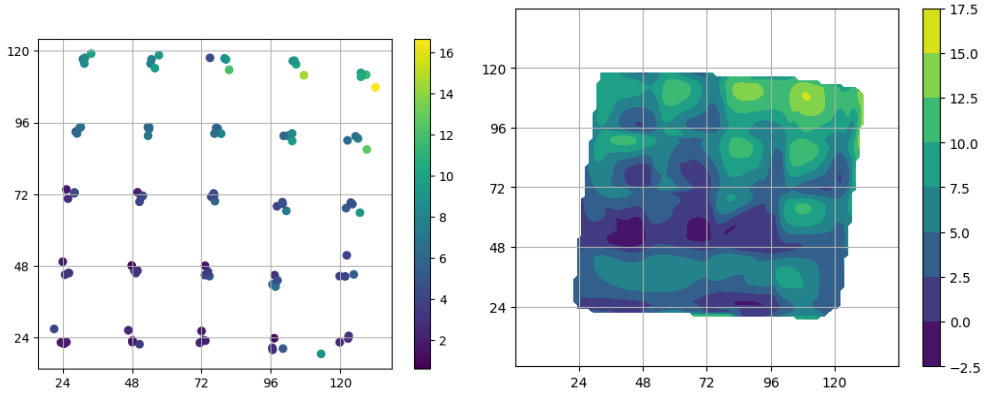
Draft work for previous section

- Cata system

- Biggerer and betterer screen system
    - Buttons and sliders enabling hardware to test speeds without a coder redeploying every 5 seconds
    - PID, PIDFF tuners
    - Odometry Map
    - 

    - Motor stats
    - Auto chooser
    - Cata State Representation for debugging, tuning
    -

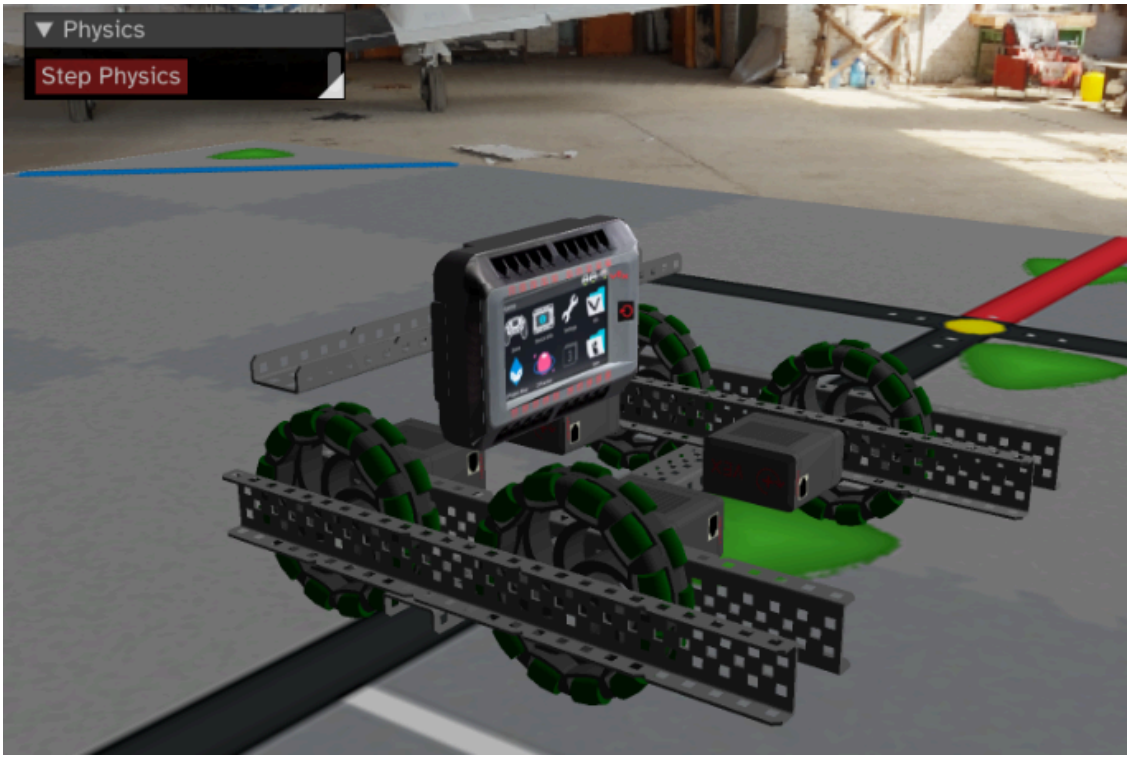
- GPS testing (with all the pretty plots (hell yeah))
- Using GPS to tune odometry



-

        - ***These pics were from testing it but i think they show a lot about the drift of our wheeled odom***
        - ***Also reasonable to point out that the middle was more accurate than the edges. Especially looking at the 24 row (near the wall but not too far off from drift) to 48 row (farther from the wall but still not too far from drift)***
- GPS odom
    - With our fun filter
    - How it fits with our other odometry components
- Cata system state machine
    - Message passing parallelism 😎
- Brake mode stuff
    - Velocity brake
    - Smart brake
    - Motivation. Why pure position brake is bad
- Pure pursuit
    - Idrk ask mcgee
- Failed experiments in IMU fusion odom?
    - We cant buy submarine IMUs :(
- Motion controller coolerness + how it interacts with pure pursuit
- Layout system

Core: Ongoing Projects

- N pod odom
- Core-rs - big things cooking
    - Bridge layer if theres anything interesting to say about it
- Vex debug board if we get it working
- Vex sim?
- 

Machine Learning

-

Core: The Funny

Only maybe, this is not professional or something

- 3d renderer
- Video player
- NES (Non-nintendo emulated system) Emulator (did outreach with this cuz a middle schooler was a big fan)