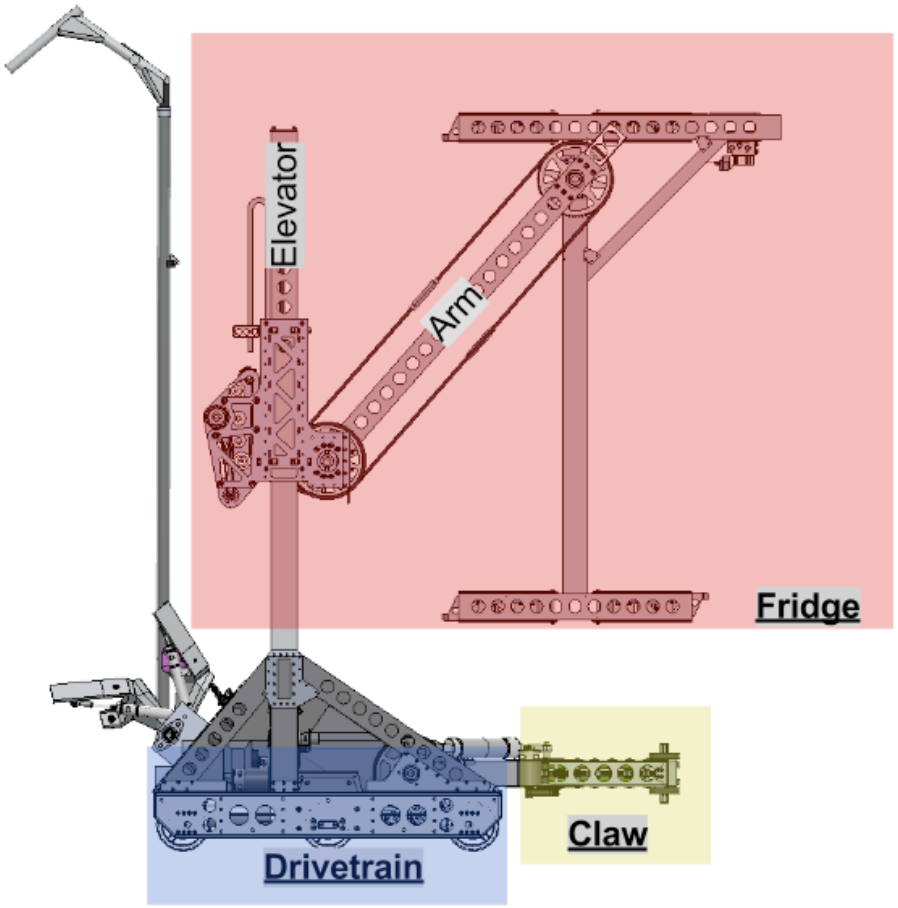
# SPARTAN ROBOTICS

# FRC 971




## Controls Documentation 2015

# Design Goals

Create a reliable and effective system for controlling and debugging robot code that provides greater flexibility and higher performance than what is available through traditional methods.

# Overall Control System

**Highlights of our code**

- *Logging:* We can log everything that happens on the RoboRIO and store it on a USB flash drive, allowing us to look after a match and see what happened in case we had any issues. Logs can also be streamed over SSH during debug sessions.
- *Control Loops:* Take advantage of our mechanical power and sensor quality using control loops designed to drive our robot quickly and predictably.
- *Unit-Testing:* Controls code on the robot utilizes Gtest unit-testing system to create a framework for testing every major system and control loop on the robot. This allows us to debug control loops before having a full robot to run it on, and alerts programmers if changes to the codebase could potentially damage the robot.

# Underlying code and processors

## RoboRIO

- Runs real-time version of Linux.
- Contains logic and instructions for autonomous and teleoperated modes.
- Logic for zeroing various superstructure systems quickly and with little physical movement.
- Control loops for the claw, arm, and elevator that iterate at 5ms.
- Logs all that happens on the robot.

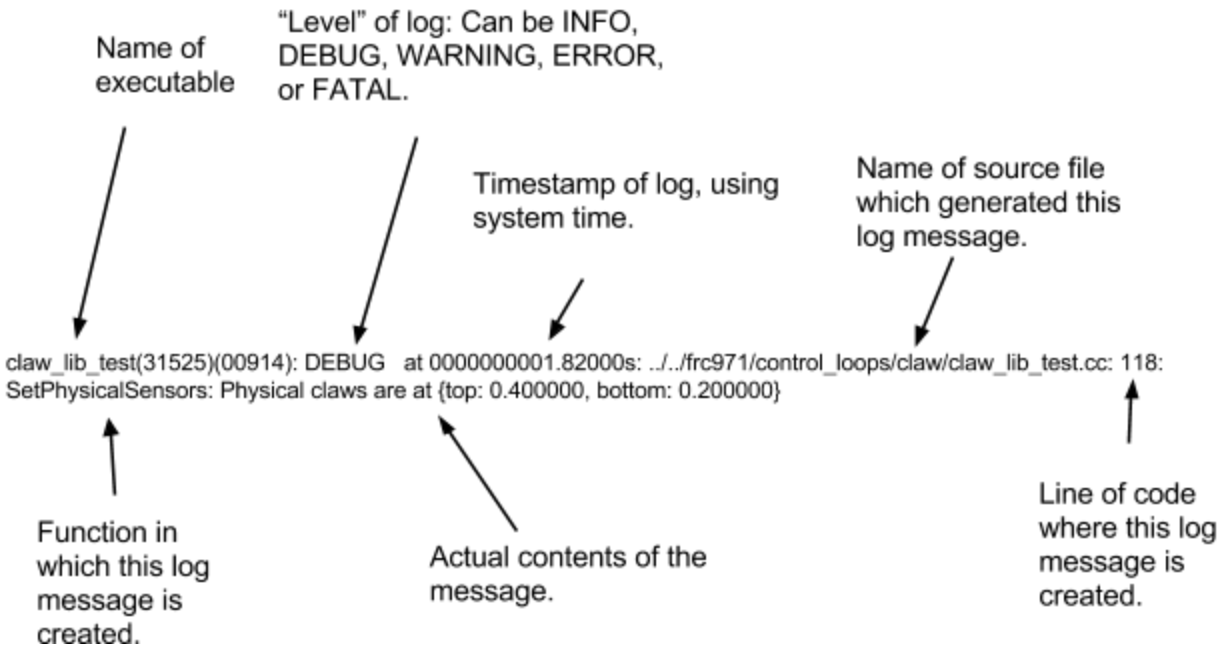
## Sensor Board

- Receives electrical signals from various sensors (analog and digital) and sends them to the RoboRIO.
    - Includes a high quality gyroscope that feeds rotational data to the control loops.
- Developed by 971 members and mentors, and has been shared with other FRC teams choosing to use the RoboRIO as well for this year's competition.

## Logging

- Logging makes it much easier to debug issues and determine what the robot was doing at any given time.
- Among other things, we log:
    - Any significant warnings or errors.
    - Sensor values at every run of the control loops, which are gathered at 100 Hz.
    - The raw motor output value sent to the motor controllers.
    - The internal state of the various control loops and zeroing logic.
    - The values of the various joysticks on the driver’s stations, which buttons were hit and released when, when the robot entered and left autonomous or teleop.

An individual log message will look something like this (this log message is taken from the output of a unit test):




A single iteration of each control loop will generally generate 4 - 5 of these messages each, providing information about position, status, outputs, goals, etc. In order to create the log message above, all that had to be done was to call the LOG macro:

```
LOG(DEBUG, "Physical claws are at {top: %f, bottom: %f}\n", pos[TOP_CLAW], pos[BOTTOM_CLAW]);
```

In order to view robot logs, team members simply SSH into the RoboRIO and run the "log_displayer" executable. An example usage might call:

```
log_displayer -f -n claw -l DEBUG
```

This would look in the claw logs (-n claw) down to the level DEBUG (-l DEBUG; in this case, it would also display WARNING, ERROR, and FATAL logs, allowing us to filter by severity) and continuously display logs as they were being created by the

robot (-f), allowing someone with access to the robot network to follow what the robot thinks it is doing in real time.

## Queues

In order communicate between the various processes running at any given time (for instance, the autonomous routine, the drivetrain control loop, and the process which sends motor output), we wrote code which creates and manages pieces of shared memory which we refer to as “queues”. To whoever uses the queues, it appears that there is what amounts to a struct in each queue; processes can write and read to the queues to communicate with other processes. For instance, there is an output queue associated with the drivetrain which contains doubles for left and right voltages and two booleans to represent shifter positions. The drivetrain control loop writes values to the output queue whenever it runs and the process which sends values to the cRIO then reads from the queue and sends all the information from the drivetrain, claw, and shooter output queues to the cRIO. In a similar manner, the autonomous routine may write position goals for the drivetrain to the drivetrain goal queue; the drivetrain control loop will then try to reach these goals.

## Control Loops

On the RoboRIO, we are running two main control loops that ultimately control the entire robot: The drivetrain, and the fridge. Each control loop presents its own unique set of challenges, although the control loops also share a great many features, such as the underlying state space models, as well as a set of custom libraries which allow for easy code-reuse from year to year and thereby quicker development of stable, effective control loops for robot mechanisms.

## Common Features

### State Space Control and Modeling

Although our robot has many powerful motors and fancy sensors, software is ultimately in command over how these devices are put into use. With

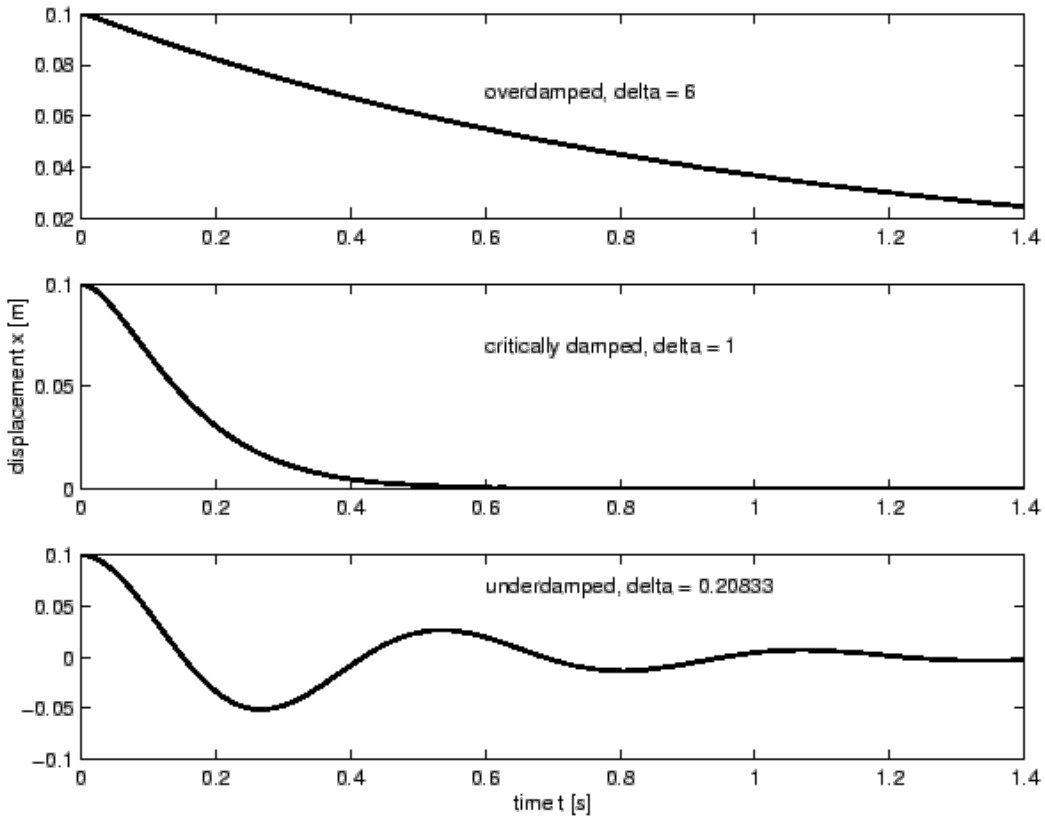
traditional control solutions, such as a simple PID loop, these systems on our robot may be able to run relatively efficiently. However, due to the competitive nature of FRC gameplay each year, our overall goal from a software perspective is to reach goal positions as fast as possible while maintaining stable control loops and a safe robot.

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t),$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t),$$

$$\mathbf{x}(0) = \mathbf{x}_0. \qquad (1.1)$$

To accomplish this, we chose to design our control loops using a state-space representation of our various systems. Essentially, this means that we are taking into account the precision of our sensors and the known torques of the motors to arrive at smart output values that get sent to the motor controllers. In our case, we are plugging these values into a Python-driven program, developed by both team mentors and students, to run simulations on these constants provided, and arrive at a series of raw matrixes that can simply be exported as a C++ struct file. Finally, while control loops are running on the robot at 200 Hz, matrixes representing the raw values from the sensors at the current time point and the output value at the last control loop tick are multiplied by the raw matrixes generated from simulations to produce output values that can be parsed and sent to the motor controllers.




In addition, our software team can control how fast the robot converges on a goal using "poles" - adjustable constants that ultimately determine the stability of a system. In the Python code, we adjust poles to adjust the damping of the system to ultimately converge on a critically damped graph, like the one above. Too much oscillation wastes battery and could lose us a game piece while in possession, while an overdamped system will leave the driveteam manipulator waiting on the software to finish an operation. Again, since this is all done in our Python simulators, resulting values can simply be plugged in to the C ++ code once we are satisfied with the simulated response.

Ultimately, our core control loops are abstracted algebraic equations. Once simulations are run based on the physical characteristics of the system and smart tuning of the poles, the resulting matrices are simply used to transform input and state values into motor outputs. However, using State Space, our systems arrive at our goals as fast and predictably as possible, and our top-notch hardware is able to work at the highest level of performance.

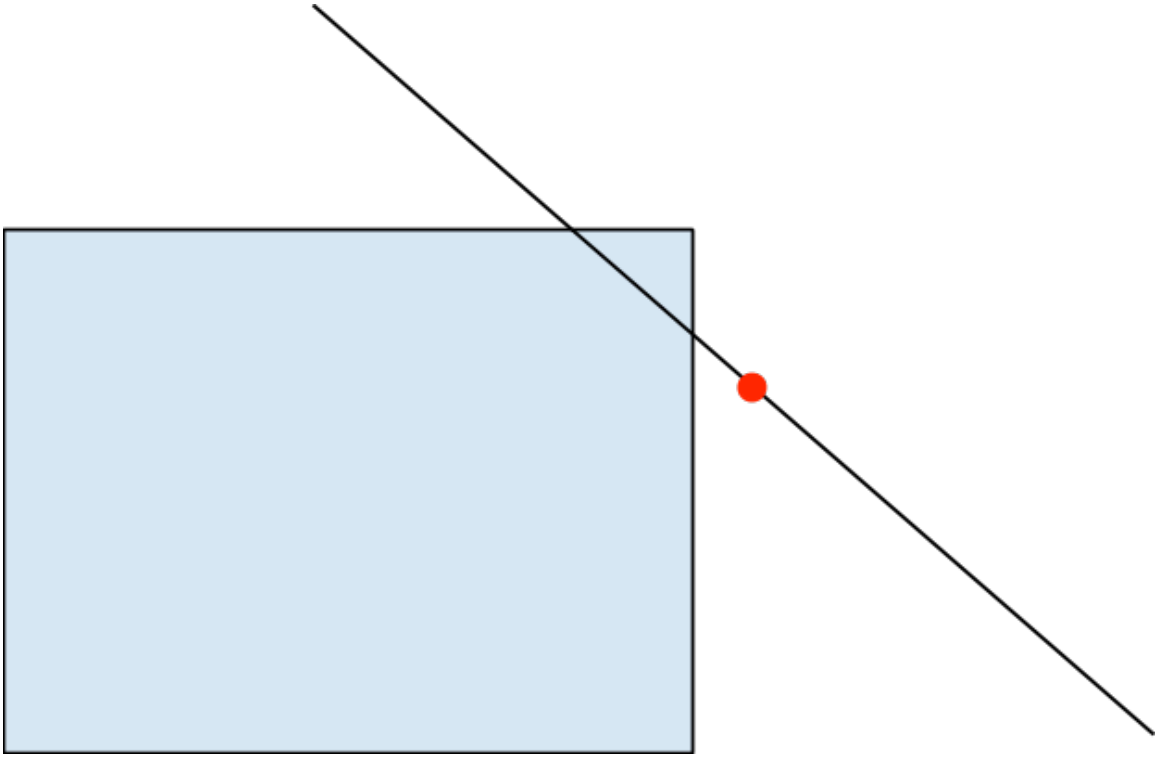
Summary of how we use State Space-driven control loops:

- Model the response of a DC motor to a given input voltage, given certain common statistics about a DC motor, such as free speed, free current, stall torque, and stall current.
- Model any other linear time-invariant aspects of the system which may be present (for example, the springs on the shooter).
- Determine the optimal controller using a Linear Quadratic Regulator and appropriate weightings of error and voltage in the cost function.
- Determine the stability and controllability of a system.
- Perform certain other operations on the system, as are described below with the polytopes.
- Use the model in our unit tests to test that the state machines are able to control the various robot systems to do what we desire them to do, and debug complicated problems before ever running code on the real hardware.

### Polytopes for setting goals to match certain conditions

In order to achieve certain desired characteristics in the behavior of a given system, such as constant radius turning in the drivetrain, we can use the state-space model of the system and certain constraints (such as a maximum of 12 volts to apply to the motors). We then end up with a situation where we have a region of possible goal states (for our purposes, these regions are two-dimensional, although the problem does generalize to n dimensions), a line which represents certain constraints which we are imposing on the system, a point which represents some sort of desired or optimal solution, and we must determine what action will fall within the provided constraints while being as close to the desired point as possible.

To provide a general example, consider the following box, line, and dot:



The “solution” point must lie inside the shaded region; if the line crosses the shaded region, the solution must fall on the line, otherwise the point in the shaded region and closest to the line is selected; once these constraints have been satisfied, the goal is to find a solution as close to the dot as possible.

How we use this math on the robot:

- The drivetrain, where we want generally to achieve a constant radius turn (which creates the line) while going as close to the desired linear and rotational velocity (the dot) as possible, all while sending out voltages which are physically possible (the region).
- The fridge, where we prioritize maintaining alignment between the left and right side when dips in voltage would potentially affect this difference and break the fridge.

# Individual Control Loops

## The Drivetrain

- Takes a throttle and steering for an input; then attempts to achieve a certain speed (the throttle) while turning at a constant radius which

corresponds to the steering input. Uses the aforementioned polytope code to achieve this.
    - On the driver's station, we have a steering wheel and a joystick; the steering wheel provides a turning radius (just as it would in a real car) and the joystick provides a desired velocity.
- State-Space model accounts for physics involved in the interactions between the left and right sides of the drivetrain—the control loop must account for the fact that, when one side of the drivetrain moves, the other side does not necessary stay in place.
- Uses quadrature encoders, one on each side of the drivetrain, to provide feedback; 4 analog hall effect sensors on the shifters provide feedback as to whether the dog shifter is fully engaged in either high or low gear on either side of the drivetrain.
- Gyroscope allows further feedback to ensure that drivetrain is driving straight and turning correctly during autonomous.

## The Fridge

- Controls both the arm and the elevator.
- Utilizes tunable models that can be simulated and used to create motion profiles that allow the fridge to move smoothly and predictably.
- Movements of the fridge are commanded by various actions that are run on command from the manipulator's button presses.
    - For example: A score action may command the robot to:
        - move the elevator up halfway
        - rotate out the arm 45º to hover a stack over the scoring platform
        - lower the elevator a half foot
        - release the pneumatic pistons to let go of the stack.
- Both sides of the robot are matched in either height (for the elevator) or angle (for the arm) to match the other side. By stabilizing the two sides independently using software, we removed a significant amount of weight to keep our robot mass under the limit of 120 lbs.

- Kinematics software in development that ensures that collisions between the fridge and the claw are averted while allowing robot actions to proceed without halting.

### The Claw

- Traditional 971-style intake that has been developed over past competition seasons.
- Voltage of the intake rollers can be adjusted to control how aggressively totes get sucked in.
- Kinematics avoids collisions with the fridge.

# Other Notes

## Unit Tests

By running unit tests on all of the individual systems in the robot, we are able to check whether changing the code in a certain way will cause any obvious issues in the functionality and safety of a given mechanism. We also adhere to test driven development practices where possible. Whenever the robot does something new which exposes a bug in the software, we read the log files to understand the bug, write a unit test to expose the bug in simulation, and then fix the bug in simulation. This means that once bugs are caught, they don't come back.

## Improvements over previous years

In the 2012 and 2013 seasons, we used a Fit-PC as a secondary processor, rather than the BBB, but we ran into several issues which led us to switch (both run linux, so the code itself did not have to change much). This year, we have transferred to the National Instruments RoboRIO as our one and only processor for all robot code. This allows us to consolidate code all onto one platform while maintaining the benefits we saw by using a BBB.