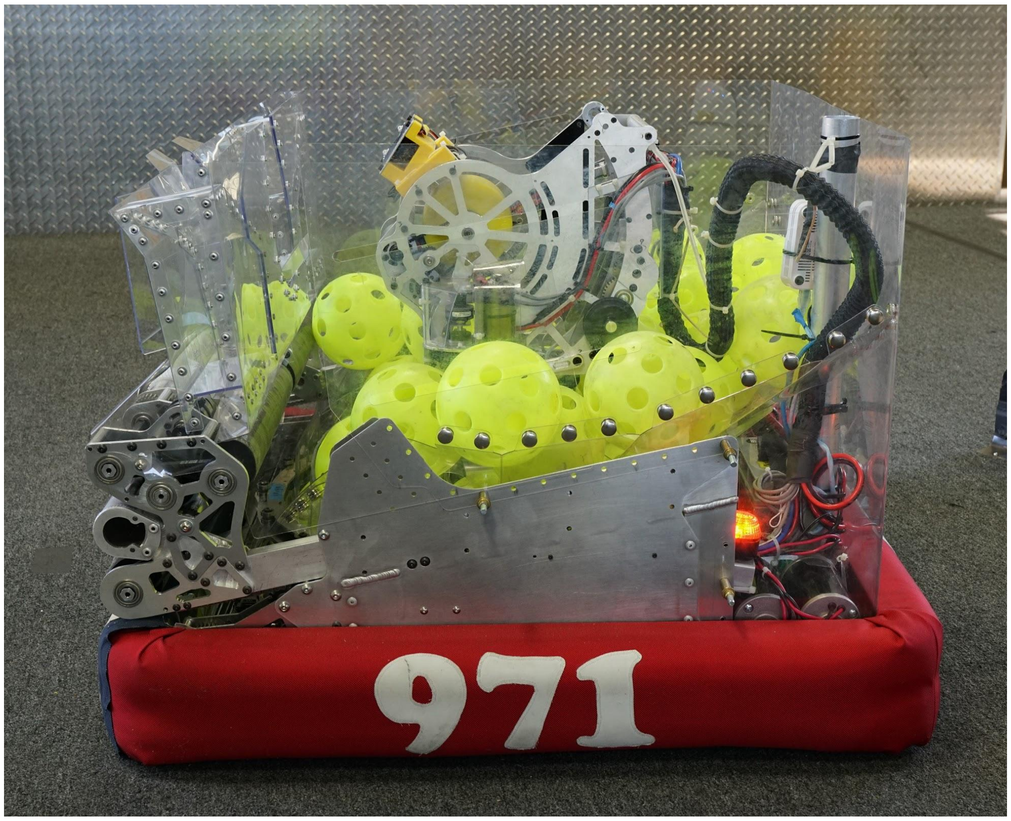
SPARTAN ROBOTICS

FRC 971



Controls Documentation 2017

Overall Control System 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.

Highlights

- *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 there were any issues. Logs can also be streamed over the network to a programmer's computer 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 the googletest unit-testing framework to test every major system and control loop on the robot. This allows us to debug control loops before having a robot to run it on, and alerts programmers if changes to the codebase could potentially damage the robot.

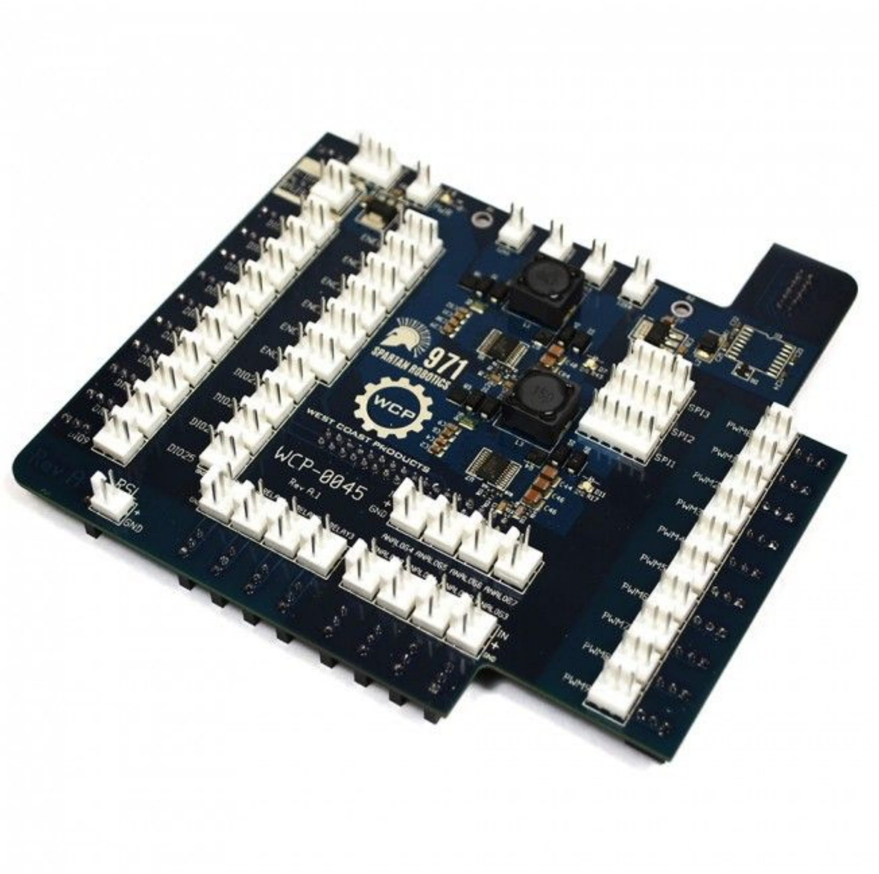
Underlying code and processors

RoboRIO

- Runs real-time version of Linux (CONFIG_PREEMPT_RT).
- 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 intake, turret, shooter, indexer, and drivetrain that iterate at 200Hz.
- Logs all that happens on the robot.

Spartan Board

- Receives electrical signals from various sensors (analog and digital) and sends them to the roboRIO.
- Developed by 971 students and mentors, and is being sold through West Coast Products.

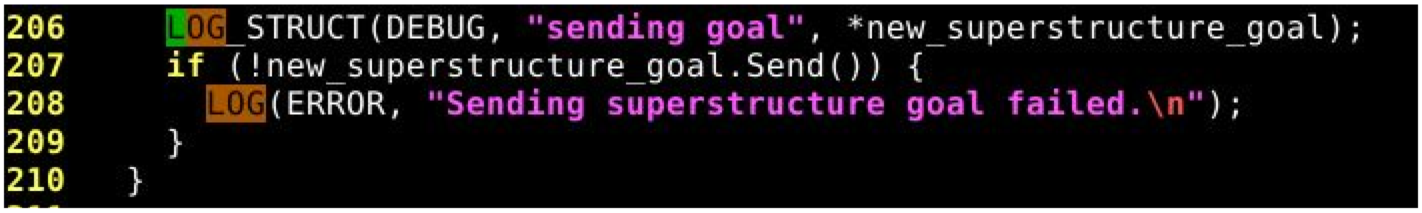


Underlying code and processors

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 200 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.

Logs messages are sent through the queues with a command similar to this:



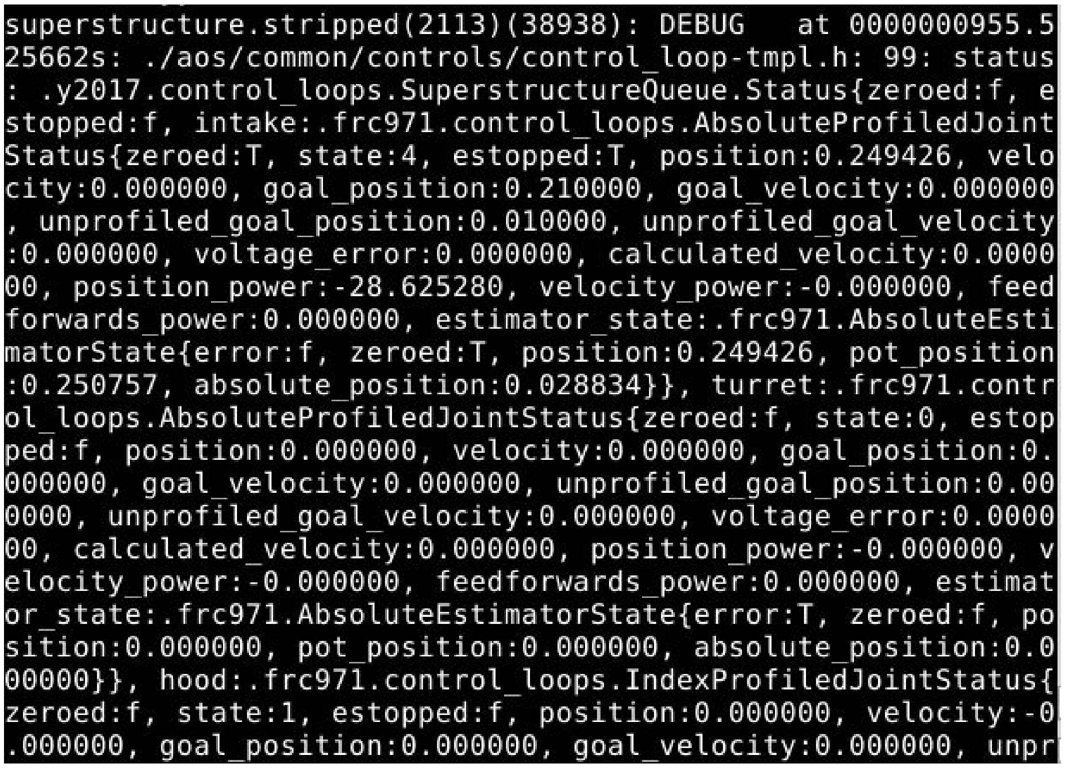
```
    LOG_STRUCT(DEBUG, "sending goal", *new_superstructure_goal);
    if (!new_superstructure_goal.Send()) {
      LOG(ERROR, "Sending superstructure goal failed.\n");
    }
  }
```

Underlying code and processors

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

```
./log_displayer -f -n superstructure -l DEBUG | grep shooter
```

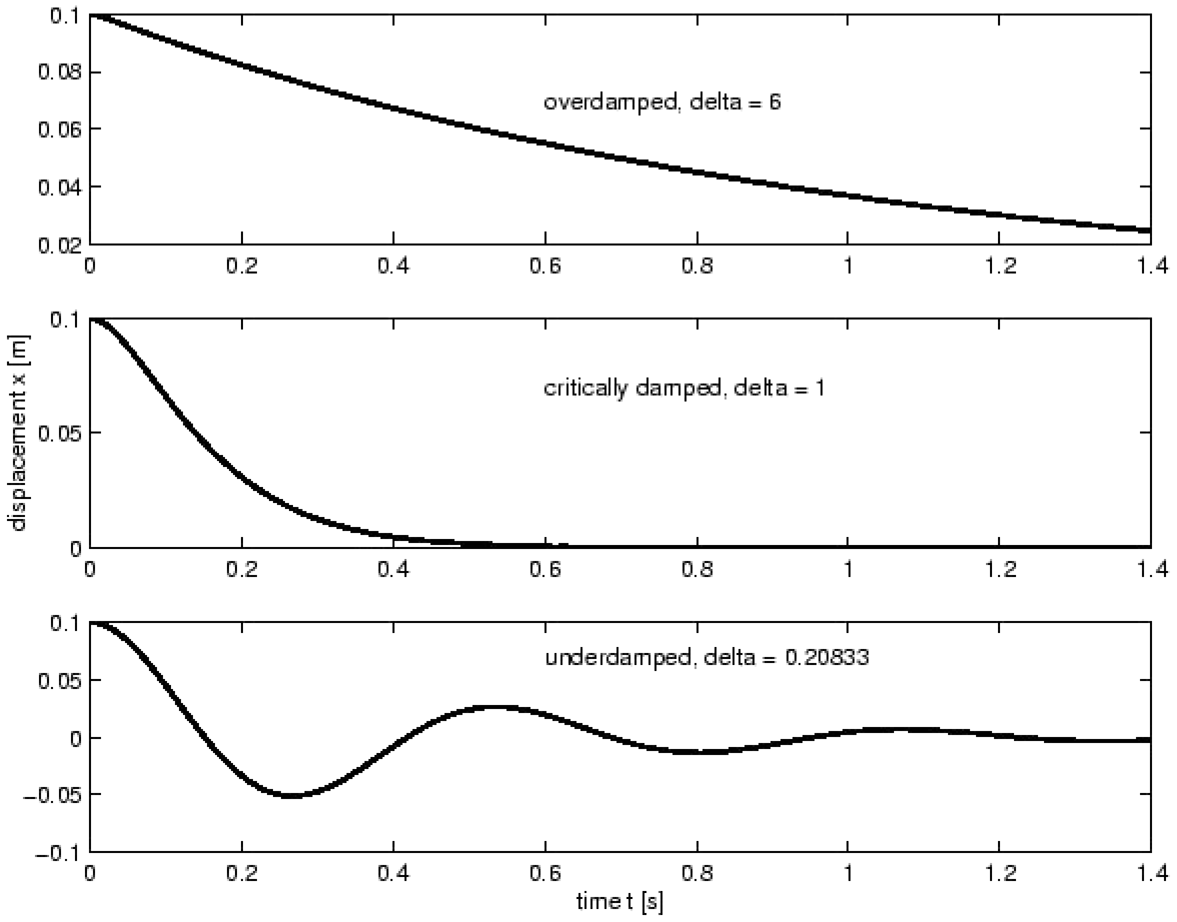
This would look in the superstructure logs (-n superstructure) down to the level DEBUG (-l DEBUG; in this case, it would also display WARNING, ERROR, and FATAL logs, allowing us to filter by severity) and continuously display logs as they were being created by the robot (-f), allowing someone with access to the robot network to follow what the robot thinks is happening in real time.



```
superstructure.stripped(2113)(38938): DEBUG   at 0000000955.5
25662s: ./aos/common/controls/control_loop-tmpl.h: 99: status
: .y2017.control_loops.SuperstructureQueue.Status{zeroed:f, e
stopped:f, intake:.frc971.control_loops.AbsoluteProfiledJoint
Status{zeroed:T, state:4, estopped:T, position:0.249426, velo
city:0.000000, goal_position:0.210000, goal_velocity:0.000000
, unprofiled_goal_position:0.010000, unprofiled_goal_velocity
:0.000000, voltage_error:0.000000, calculated_velocity:0.0000
00, position_power:-28.625280, velocity_power:-0.000000, feed
forwards_power:0.000000, estimator_state:.frc971.AbsoluteEsti
matorState{error:f, zeroed:T, position:0.249426, pot_position
:0.250757, absolute_position:0.028834}}, turret:.frc971.contr
ol_loops.AbsoluteProfiledJointStatus{zeroed:f, state:0, estop
ped:f, position:0.000000, velocity:0.000000, goal_position:0.
000000, goal_velocity:0.000000, unprofiled_goal_position:0.00
0000, unprofiled_goal_velocity:0.000000, voltage_error:0.0000
00, calculated_velocity:0.000000, position_power:-0.000000, v
elocity_power:-0.000000, feedforwards_power:0.000000, estimat
or_state:.frc971.AbsoluteEstimatorState{error:T, zeroed:f, po
sition:0.000000, pot_position:0.000000, absolute_position:0.0
00000}}, hood:.frc971.control_loops.IndexProfiledJointStatus{
zeroed:f, state:1, estopped:f, position:0.000000, velocity:-0
.000000, goal_position:0.000000, goal_velocity:0.000000, unpr
```

Control Loops

To accomplish this, we chose to implement our control loops using a state-space representation of our various systems. Essentially, this means that we are taking into account the characteristics of our sensors and the known torques of the motors to arrive at optimal output values that are then sent to the motor controllers. We enter various constants into Python programs, developed by team members over several seasons, to run simulations of the mechanisms and arrive at a set of constant matrices. These are then exported as C++ code which actually runs on the robot. On the robot, at each 200 Hz control loop cycle, the current sensor values are combined with the previous output values and some internal state to generate a new output value and new state estimate using these matrices.



Control Loops

We control how fast each mechanism in the robot converges to its goal using "poles" - adjustable constants that determine the balance between stability and quick response times. In the Python code, we adjust the poles and look at the effect on a simulation of the robot. Too much oscillation (due to responding too quickly) wastes energy and can increase overall convergence times, while an overdamped system can slow down the robot's operation. Since this is all done in our Python simulators, resulting values can simply be plugged into 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. 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.

Hybrid Kalman Filter

Due to the heavy duty the shooter had to undergo this year, we had to extend the regular kalman filter that we have always used to get it to stabilize fast enough after a shot. This hybrid kalman filter takes into consideration the discrepancy between two individual time stamps in the sensors, allowing for a more accurate prediction step.

The equations change, but the biggest step is that now A and B are not constants, so they need to be calculated in real time while the robot is running.

Since the code to achieve this is now written, there is no reason why we shouldn't implement it for every subsystem in the robot, so that's what we are working on.

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

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

$$\Big\downarrow$$

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

Hybrid Kalman Filter

The reason why this was such a complicated task is because the equations to find the discrete version of A and B used to be calculated in compile-time in python. This is easily done with libraries provided in python, but in C++, it's not as simple. The equations are the following:

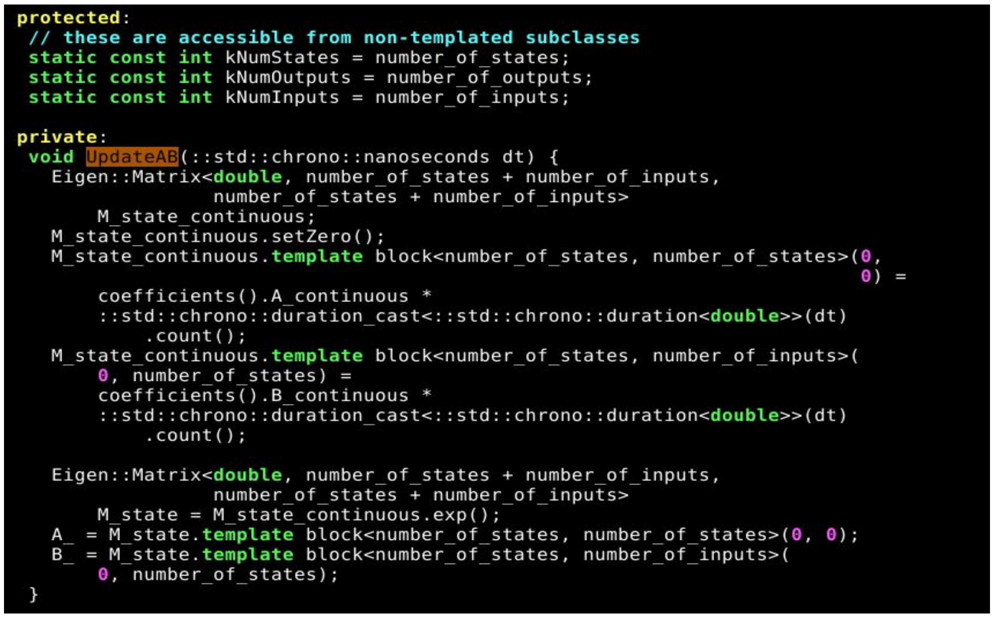
$$\mathbf{A}_d = e^{\mathbf{A}T} \qquad \mathbf{B}_d = \left(\int_{\tau=0}^{T} e^{\mathbf{A}\tau} d\tau\right)\mathbf{B} = \mathbf{A}^{-1}(\mathbf{A}_d - I)\mathbf{B}$$

But this is actually impossible to compute algebraically since A is a singular matrix. Therefore, we found a second way of calculating both constants. The equation is

$$e^{\begin{bmatrix}\mathbf{A} & \mathbf{B}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}T} = \begin{bmatrix}\mathbf{M}_{11} & \mathbf{M}_{12}\\ \mathbf{0} & \mathbf{I}\end{bmatrix} \text{ where } \begin{aligned}\mathbf{A}_d &= \mathbf{M}_{11}\\ \mathbf{B}_d &= \mathbf{M}_{12}\end{aligned}$$

and T is the **Δ**time.

Using this last equation, we are able to find the discrete version of A and B at runtime in C++.



```
protected:
 // these are accessible from non-templated subclasses
 static const int kNumStates = number_of_states;
 static const int kNumOutputs = number_of_outputs;
 static const int kNumInputs = number_of_inputs;

private:
 void UpdateAB(::std::chrono::nanoseconds dt) {
   Eigen::Matrix<double, number_of_states + number_of_inputs,
                 number_of_states + number_of_inputs>
       M_state_continuous;
   M_state_continuous.setZero();
   M_state_continuous.template block<number_of_states, number_of_states>(0,
                                                                           0) =
       coefficients().A_continuous *
       ::std::chrono::duration_cast<::std::chrono::duration<double>>(dt)
           .count();
   M_state_continuous.template block<number_of_states, number_of_inputs>(
       0, number_of_states) =
       coefficients().B_continuous *
       ::std::chrono::duration_cast<::std::chrono::duration<double>>(dt)
           .count();

   Eigen::Matrix<double, number_of_states + number_of_inputs,
                 number_of_states + number_of_inputs>
       M_state = M_state_continuous.exp();
   A_ = M_state.template block<number_of_states, number_of_states>(0, 0);
   B_ = M_state.template block<number_of_states, number_of_inputs>(
       0, number_of_states);
 }
```

Control Loops

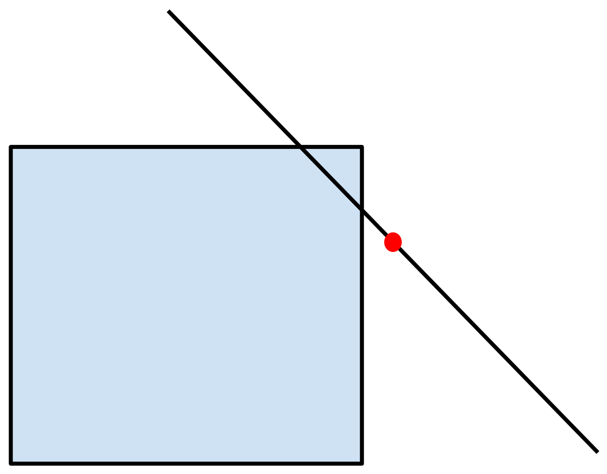
Summary of how we use state space control loops:

- Model the response of a DC motor to a given input voltage, given certain characteristics of the motor, such as its 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 elasticity of the belts and the expansion of the shooter as it rotates.).
- 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.

Control Loops

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, and a point which represents some sort of desired or optimal solution. We must then 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:



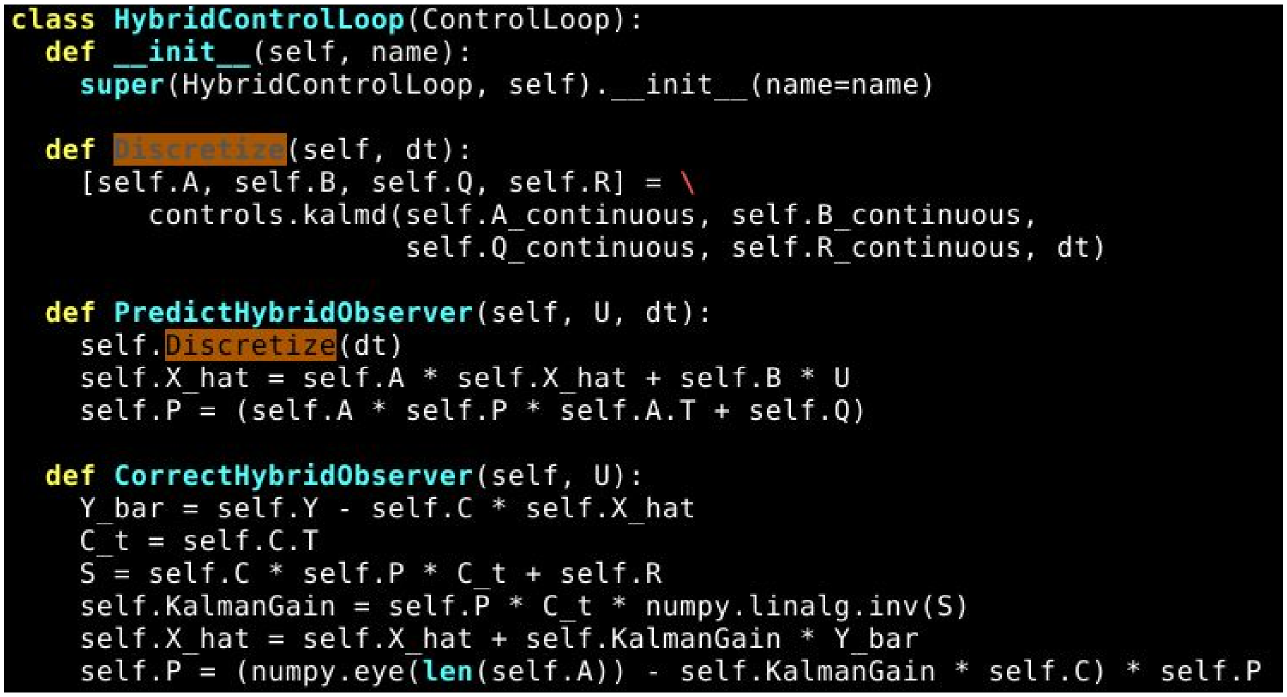
The “solution” point must lie inside the shaded region; if the line crosses the shaded region, the solution must fall on the line, otherwise the point in the shaded region and closest to the line is selected. Once these constraints have been satisfied, the goal is to find a solution as close to the dot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

Control Loops

Individual Control Loops

- Takes in a vector as a goal – often position and velocity – and the state-space controller tries to minimize the error between the goal and the actual position by using a Kalman Filter.
- The Kalman Filter uses models of the physical world along with sensor readings to predict the position and velocity of each subsystem. The controller will use the sensor readings to reduce the covariance, and thus, minimize the difference between the model and the real world. This process is repeated every 5ms since the control loops run at a 200 Hz frequency.



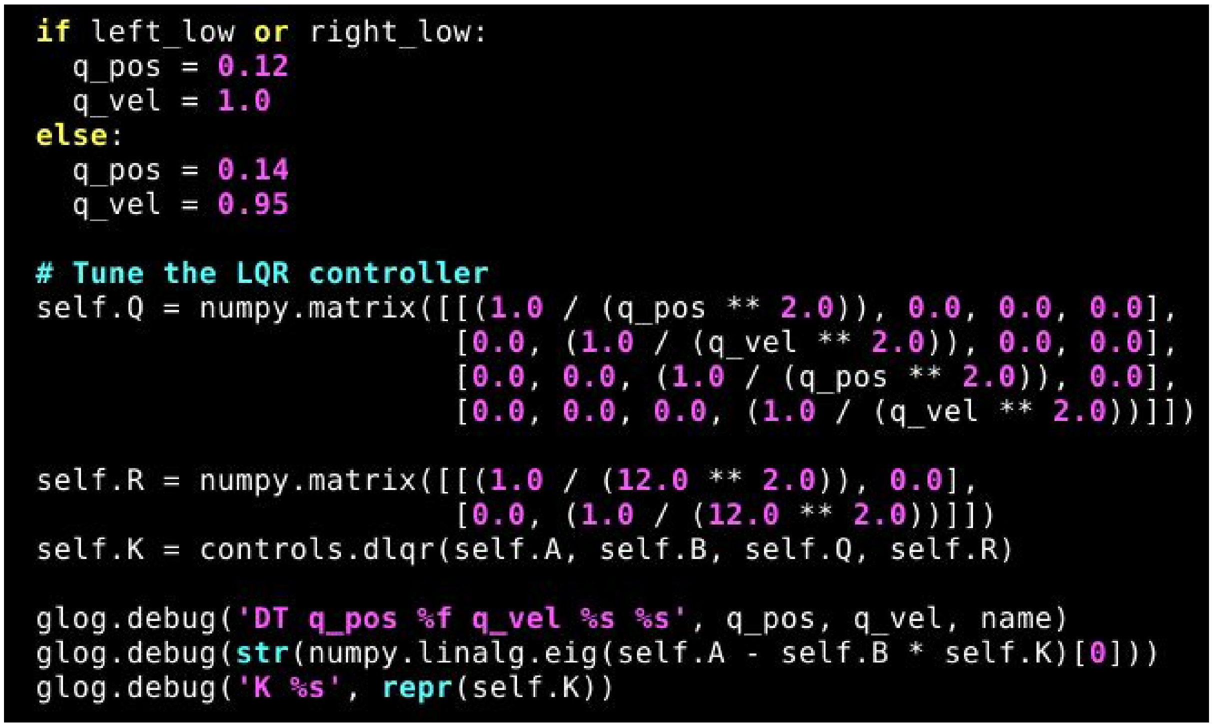
```
class HybridControlLoop(ControlLoop):
  def __init__(self, name):
    super(HybridControlLoop, self).__init__(name=name)

  def Discretize(self, dt):
    [self.A, self.B, self.Q, self.R] = \
        controls.kalmd(self.A_continuous, self.B_continuous,
                       self.Q_continuous, self.R_continuous, dt)

  def PredictHybridObserver(self, U, dt):
    self.Discretize(dt)
    self.X_hat = self.A * self.X_hat + self.B * U
    self.P = (self.A * self.P * self.A.T + self.Q)

  def CorrectHybridObserver(self, U):
    Y_bar = self.Y - self.C * self.X_hat
    C_t = self.C.T
    S = self.C * self.P * C_t + self.R
    self.KalmanGain = self.P * C_t * numpy.linalg.inv(S)
    self.X_hat = self.X_hat + self.KalmanGain * Y_bar
    self.P = (numpy.eye(len(self.A)) - self.KalmanGain * self.C) * self.P
```

Control Loops

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.
- The 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 necessarily stay in place.
- Gyroscope allows further feedback to ensure that drivetrain is driving straight and turning correctly during autonomous.

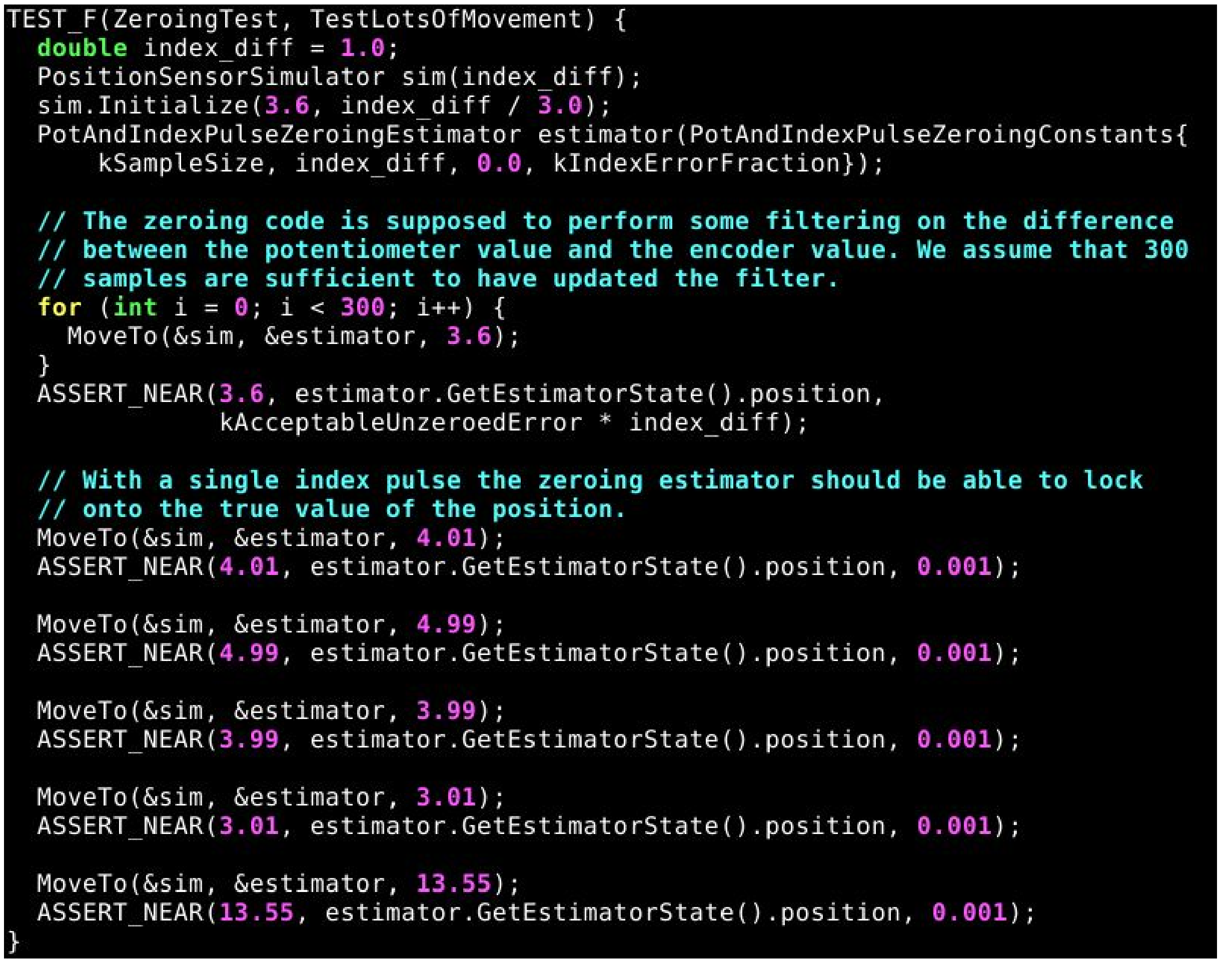


```
if left_low or right_low:
  q_pos = 0.12
  q_vel = 1.0
else:
  q_pos = 0.14
  q_vel = 0.95

# Tune the LQR controller
self.Q = numpy.matrix([[(1.0 / (q_pos ** 2.0)), 0.0, 0.0, 0.0],
                       [0.0, (1.0 / (q_vel ** 2.0)), 0.0, 0.0],
                       [0.0, 0.0, (1.0 / (q_pos ** 2.0)), 0.0],
                       [0.0, 0.0, 0.0, (1.0 / (q_vel ** 2.0))]])

self.R = numpy.matrix([[(1.0 / (12.0 ** 2.0)), 0.0],
                       [0.0, (1.0 / (12.0 ** 2.0))]])
self.K = controls.dlqr(self.A, self.B, self.Q, self.R)

glog.debug('DT q_pos %f q_vel %s %s', q_pos, q_vel, name)
glog.debug(str(numpy.linalg.eig(self.A - self.B * self.K)[0]))
glog.debug('K %s', repr(self.K))
```

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.



```
TEST_F(ZeroingTest, TestLotsOfMovement) {
  double index_diff = 1.0;
  PositionSensorSimulator sim(index_diff);
  sim.Initialize(3.6, index_diff / 3.0);
  PotAndIndexPulseZeroingEstimator estimator(PotAndIndexPulseZeroingConstants{
      kSampleSize, index_diff, 0.0, kIndexErrorFraction});

  // The zeroing code is supposed to perform some filtering on the difference
  // between the potentiometer value and the encoder value. We assume that 300
  // samples are sufficient to have updated the filter.
  for (int i = 0; i < 300; i++) {
    MoveTo(&sim, &estimator, 3.6);
  }
  ASSERT_NEAR(3.6, estimator.GetEstimatorState().position,
              kAcceptableUnzeroedError * index_diff);

  // With a single index pulse the zeroing estimator should be able to lock
  // onto the true value of the position.
  MoveTo(&sim, &estimator, 4.01);
  ASSERT_NEAR(4.01, estimator.GetEstimatorState().position, 0.001);

  MoveTo(&sim, &estimator, 4.99);
  ASSERT_NEAR(4.99, estimator.GetEstimatorState().position, 0.001);

  MoveTo(&sim, &estimator, 3.99);
  ASSERT_NEAR(3.99, estimator.GetEstimatorState().position, 0.001);

  MoveTo(&sim, &estimator, 3.01);
  ASSERT_NEAR(3.01, estimator.GetEstimatorState().position, 0.001);

  MoveTo(&sim, &estimator, 13.55);
  ASSERT_NEAR(13.55, estimator.GetEstimatorState().position, 0.001);
}
```