BARC Project - Traction Control Implementation and Analysis

Clayton Langbein, Johan Kok, and Maxwell Gerber

May 9, 2016

1 Abstract

This project is intended to be an in-depth analysis of the process behind designing an effective traction control system for a Berkeley Autonomous Race Car (BARC). The process is split into three distinct parts: initial evaluation of the properties of the vehicle using theory and models developed in UC Berkeley's ME131 course, design and implementation of a control system in Python and C++ using the ROS framework, and data analysis, filtering, and processing in Matlab.

2 Introduction & Theory

As the shift from manually-driven to autonomous cars continues, one of the main concerns among the public is safety. How we rely on this machine controlled car in extreme scenarios? For the world to make a switch to a fully autonomous driving population, these vehicles must be able to operate in all conditions:rain, snow, hear, cold, etc. This requires a finely tuned traction control system.

Traction control depends on two things, the slip ratio of the vehicle and the tire traction force. These both depend on a several characteristic values of the vehicle and some conditional state variables such as velocity and road conditions.

3 Drive-line Dynamics Model

The dynamics model for longitudinal motion is made up of two main elements:the vehicle dynamics model and the drive-line(powertrain) model. The powertrain model is comprised of the engine,the torque converter, the transmission, and the wheels. These two models are made up of eight equations [2]. They are

$$\dot{m_m} = \dot{m_{a_i}} - \dot{m_{a_o}} \tag{1}$$

$$I_e\dot{\omega}_e = T_{ind} - T_f - T_a - T_p \tag{2}$$

$$I_t\dot{\omega}_t = T_t - R_iR_dT_s \tag{3}$$

$$\dot{T}_s = k_s(R_d\omega_{cr} - \omega_{wd}) \tag{4}$$

$$I_{wd}\dot{\omega_{wd}} = T_s - h_dF_d - T_{brd} - T_{md} \tag{5}$$

$$I_{wv}\dot{\omega_{wv}} = -h_uF_u - T_{brv} - T_{rru} \tag{6}$$

$$m\dot{V_x} = F_{bd} + F_{tv} - F_a - F_rr \tag{7}$$

$$\tau_bT_{br} = \mu A_br_bP_c \tag{8}$$

However, when doing controller design, we have to simplify the model. We start in equation (4), with the assumption that K_s is very large. Thus, $\frac{\dot{T}_s}{K_s} \approx 0$ and $R_d\omega_{cr} - \omega_{wd} \approx 0$. From this we

can see easily that $\omega_{cr} = \frac{\omega_{wd}}{R_d}$. Since we know that $w_{cr} = R_x\omega_t$, we can solve for T_s in equation (3) and substitute it into equation (5) to get

$$(I_{wd} + \frac{I_t}{R_d^2 R_x^2})\dot{\omega_{wd}} = \frac{T_t}{R_d R_i} - h_d F_{td} - F_{brd} \tag{9}$$

By using the fact that $\dot{w_t} = \frac{\dot{w_{wd}}}{R_d R_i}$ we have combined equations (3), (4), and (5) into just one equation.

Now, looking at equation (1) we can assume that $\dot{m_a} = 0$. Also, in equation (2), T_a is zero and we assume $T_{ind} - T_f \equiv T_{eng}$ (Engine Torque). We do this because we can not directly obtain T_{ind}, but we can measure $T_{ind} - T_f$. T_eng is found using a lookup table based on the throttle angle, α and ω_e This makes equation (2) become

$$I_e\dot{\omega_e} = T_{eng}(\alpha, \omega_e) - T_p \tag{10}$$

Taking this, along with combining equations (6),(7), and (8), we get our final longitudinal dynamics model with only three equations.

$$m\dot{V_x} = 2F_{xd} \tag{11}$$

$$I_e\dot{\omega_e} = -b_e\omega_e + T_c - 2Rr_eF_{xd} + RT_{brd} \tag{12}$$

$$T_c = T_{eng}(t - \tau_f) \tag{13}$$

4 Slip Ratio

The longitudinal slip is defined as the difference between the longitudinal velocity of the vehicle, V_x , and the effective longitudinal velocity of the tires, $r_eff * \omega_w$. The slip ratio then follows as

$$\sigma_x = \frac{r_{eff} * \omega_w - Vx}{r_{eff} * \omega_w}. \quad \text{(During Acceleration)} \tag{14}$$

and

$$\sigma_x = \frac{(r_{eff} * \omega_w - V_x)}{V_x} \quad \text{(During Braking)} \tag{15}$$

The slip ratio is relatively small when driving on a dry surface, around 0.1. When this is the case, longitudinal tire force is directly proportional to the slip ratio. In this small slip region, the tire force can written as

$$F_{xf} = C_{\sigma f} * \sigma_{xf} \tag{16}$$

$$F_{xr} = C_{\sigma r} * \sigma_{xr} \tag{17}$$

where F_{xf} and F_{xr} are the longitudinal tire forces on the front and rear tire respectively. When tire traction force is plotted vs. slip ratio, it results in the following curve. Longitudinal tire force depends on slip ratio, in part, because of region of the tire called the "static region." The tread elements of a tire are modeled as a series of independent spring that compress and decompress with constant stiffness. As the tire is rolling, it makes contact with the ground over a non-zero span of space due to the normal load. In this contact area, there is a region where the tire threads do not slip; this zone is known as the "static" region. The tip of this region must have no velocity because of the no slip condition.

If we define the longitudinal velocity of the car as V_x and the rotational velocity of the wheel as ω_w, then the net velocity, or slip velocity, of the treads is $r_{eff} * \omega_w - V_x$ where r_{eff} is the effective radius of the tire. The bending deflection is proportional to slip Velocity and the time the the tire treads remain in the contact region. The contact time is inversely proportional to the rotational velocity. Thus, the longitudinal force from the ground to the tires is proportional to the slip ratio, $\frac{r_{eff} * \omega_w - Vx}{r_{eff} * \omega_w}$.

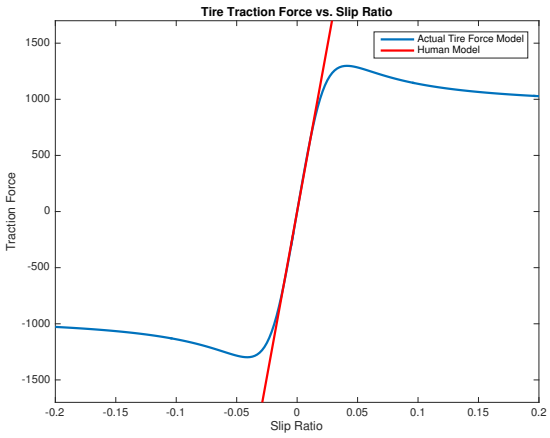


Figure 1: Tire Force Model

5 Traction Control

The human brain's model of the Tire Force Model is not the same as the one seen in the blue line of Figure 1. It is a linear model, similar to the red line. Push harder on the gas, go faster. Push harder on the brakes, stop faster. As we've seen from above, this is not the case. If you push too hard on the gas, your wheels will begin to slip, and you won't get to your desired speed.

Traction Control is used to minimize the slip ratio for a vehicle. A well designed Traction Control system takes the desired, optimal slip ratio, the actual measured slip ratio and uses PID control to reduce the error between them. It changes the actual slip by adjusting the engine torque so that it does not over shoot the tire traction force on the Tire Force Model as seen in Figure 1.

6 BARC - The Plant

Our traction controller is implemented on a 1/10 scale fully electronic RC car. The Berkeley Autonomous Race Car (BARC) is a development platform for autonomous driving and nonlinear model predictive control. Pioneered at UC Berkeley under Professor Borrelli and Professor Hedrick, the BARC project has been an area of interest of the Model Predictive Control lab since 2015. Built on top of the Basher RZ-4 car chassis, the car is controlled by an embedded Odroid XU4 running ROS. Custom machined standoffs, plates, and brackets make it possible to mount controllers and numerous sensors, including cameras, ultrasonic distance sensors, and inertial measurement units. Plastic 3-D printed parts allow for the installation of hall effect encoders on each of the RZ-4's wheels. All plans and installation instructions are available online through `http://www.barc-project.com/` as part of the project's commitment to be fully open-source.

Figure 2: Basher RZ-4[1] and assembled BARC[3]

The BARC uses a brushless motor and Electronic Speed Control to convert a PWM signal provided by the Ardiono into a rotor speed. It also features an oil filled rear gear differential and front spool differential to enable better turning and drift. A high-torque servo (also controlled using PWM) sets the steering angle using Ackermann steering. [3]

7 A Brief Overview of ROS - The Controller

In order to streamline the prototyping process, we made use of the Robotic Operating System (ROS) platform. Contrary to its name, ROS is actually not an operating system. Rather, it is a complex and complete of middleware that runs on top of the Linux OS to provide numerous useful features. ROS at its core is a set of libraries and tools that help software developers focus less on the bits-and-bytes implementation and more on the overall project. Some of its most notable features

include powerful hardware abstraction, real-time data visualization, and automated scripting.

The core of the ROS environment is the concept of a Node. A Node is a standalone process that isolates one specific part of the robot from the other parts. For example, a line-following robot might have one node dedicated to sensing the location of the line, one node to handle the control process, and one node to drive the motors. Each node is only concerned with its own specific task and is entirely oblivious to how the other nodes complete their tasks. The beauty of this relates back to a software development concept known as encapsulation. If a developer decides to change the implementation of the line-sensing node later down the line, they can be certain that their changes will not affect any other node in the system. Furthermore, this enables ROS to be language agnostic. One node can be written in Python, another in Java, a third in Julia or C++. All of them can run at the same time on the same system. [4]

Nodes communicate necessary information with each other by sending specialized signals (messages) over specific channels (called Topics in ROS). Nodes have the ability decide which topics they wish to send (publish) and receive (subscribe to). This contributed to our goal of encapsulation because it means each node is only aware of what it needs to be aware of. In our previous example, the node dedicated to the sensing of the line might publish to a Position topic. The controller node can subscribe to the Position topic and publish to a Motor_Signal topic, which the motor driver will subscribe to in turn. However, the motor driver node has absolutely no idea that the line sensing node or the Position topic even exist. It only concerns itself with the driving of the motors.

The messages themselves are specialized data formats designed by the user. Very similar to Structs in C or Objects in MATLAB, each message is made up of certain fields. Theses files dictate the exact types of information conveyed in each message. For example, the messages published to the Motor_Signal topic might be composed as follows:

```
float32 right_mot_val
float32 left_mot_val
String timestamp
```

In order to facilitate rapid deployment custom deployment of large amounts of nodes, ROS makes use of specialized .launch files. These files, written in XML syntax, tell ROS which nodes to run and what parameters to pass in to the nodes. An example .launch file is as follows:

This .launch file starts three nodes: an IMU node which records acceleration data from the Inertial Measurement Unit, an Arduino node which is used to write data to the motor controller and servo, and a circular controller node used to determine steering angle over the course of the run. In addition, the .launch file specifies numerous values which are read into the program at runtime. This makes it possible to change numerous system parameters without recompiling code,

```
<launch>
    <!-- IMU NODE -->
    <node pkg="barc" type="imu_data_acquisition.py" name="imu_node" >
        <param name="port" value="/dev/ttyACM0" />
    </node>

    <!-- ARDUINO NODE -->
    <!--  * encoders and ultrasound sensors -->
    <node pkg="rosserial_python" type="serial_node.py" name="arduino_node" >
        <param name="port" value="/dev/ttyUSB0" />
    </node>

        <!-- OPEN LOOP MANUEVERS -->
    <node pkg="barc" type="controller_circular.py" name="controller" >
        <param name="v_x_pwm" type="int" value="97" />
        <param name="steering_angle" type="int" value="25" />
                <param name="t_exp" type="int" value="5" />
    </node>
</launch>
```

saving additional developer time.

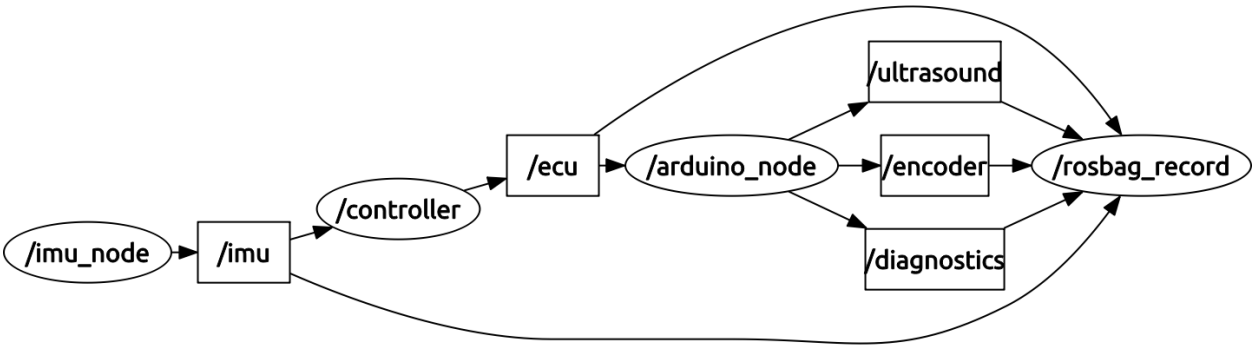


Figure 3: Map of ROS nodes showing recording process

A final incredibly useful feature of ROS is its powerful data logging abilities. All messages broadcast by nodes at any time can be saved for future playback or analysis. This makes it possible to completely understand the state of the BARC system at any point during an experiment. In Figure 3, nodes are represented by circles and messages are represented by squares. The *rosbag_record* node is subscribed to every topic and records every message sent. All of this data is automatically uploaded into the cloud, using a service called Dator.

Dator is an open-source data aggregation program for groups of robots. It allows for data to be pushed from robots on private networks to a centralized platform, avoiding the need to bridge into the local robot's network to control and analyze data. The main role of the platform is to provide a standardized way to record data and actuation events from one or more local computers (robots) for later analysis of robot performance and simulation of new control regimens.[5]

This means that every experiment we conducted over the course of writing this paper is available for analysis on `http://www.dator.forge9.com` for viewing by the public.

8 ROS Naive Controller Implementation

The purpose of this section is to analyze and explain the control structure as set up in the openLoop_straight.launch file included in the appendix. The flowchart in Figure 4 shows the active nodes and the messages they use.

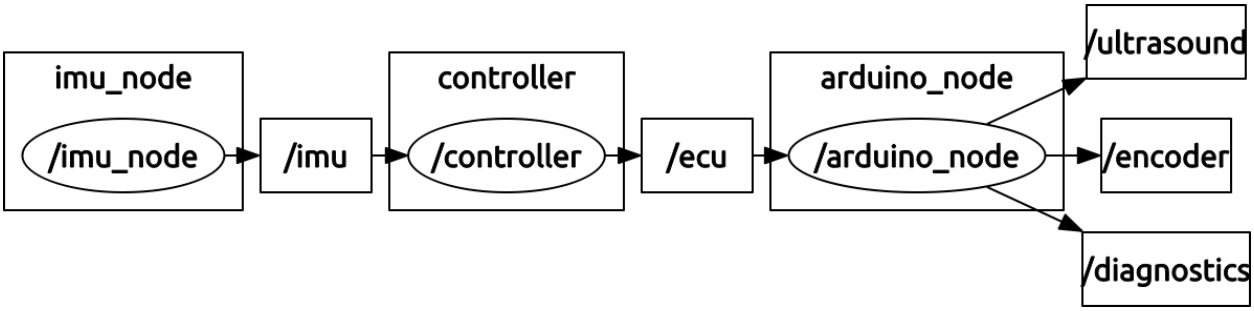


Figure 4: Map of ROS nodes in Naive Controller

Node	Purpouse
imu_node	Records and broadcasts IMU data
controller	Processes yaw error and generates control signal
arduino_node	Directs control signal to plant (BARC)

Table 1: Overview of Nodes

Message	Contents
imu	yaw, pitch, roll, acc_x, acc_y, acc_z, timestamp, etc.
ecu	Motor PWM, Servo PWM
ultrasound	Ultrasound Readings (Not Used)
encoder	Front Encoder Count, Rear Encouder Count
diagnostics	General System Data (Not Used

Table 2: Overview of Messages

Although this controller provides rudimentary lateral control by means of a PID loop used to moderate yaw, it allows for lateral slip and leaves much room for improvement.

9 ROS Traction Controller Implementation

This section will analyze and explain the control structure set up in the traction.launch file included in the appendix. The flowchart in Figure 5 shows the active nodes and the messages they use. Three significant improvements over the naive implementation are present in this model.

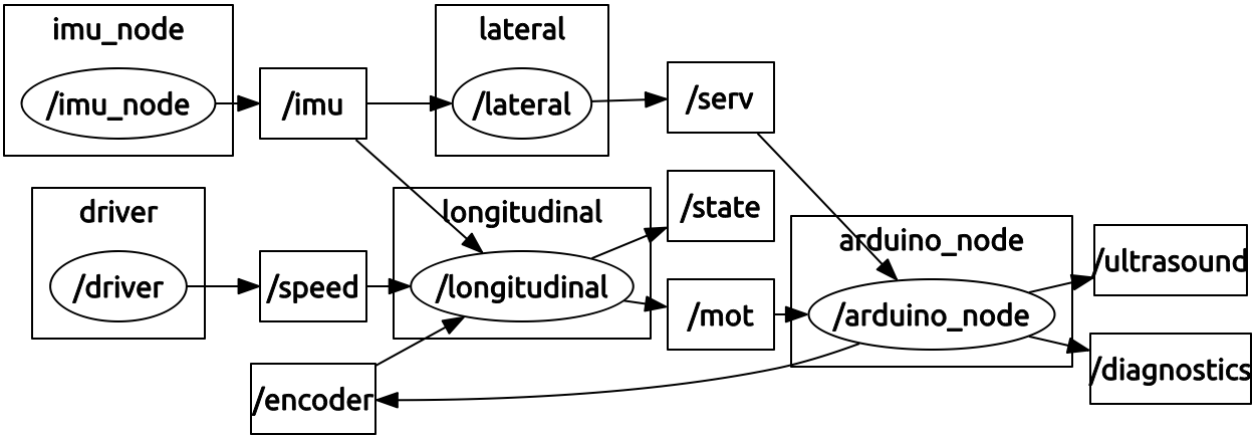


Figure 5: Map of ROS nodes in Traction Controller

Node	Purpouse
imu_node	Records and broadcasts IMU data
driver	Broadcasts Driver Input Speed
lateral	Yaw control
longitudinal	Slip Control
arduino_node	Directs control signal to plant (BARC)

Table 3: Overview of Nodes

Message	Contents
imu	yaw, pitch, roll, acc_x, acc_y, acc_z, timestamp, etc.
speed	Driver Desired Speed
serv	Servo PWM
mot	Motor PWM
state	Vx, WwEff, and error
ultrasound	Ultrasound Readings (Not Used)
encoder	Front Encoder Count, Rear Encoder Count
diagnostics	General System Data (Not Used

Table 4: Overview of Messages

First, instead of having v_x as a static parameter, we have created a driver node which broadcasts a speed message. Abstraction allows for us to change this later without impacting the rest of the system. This allows us to test dynamically created driver inputs, such as step inputs, sine waves, saved data from files, etc. Such potential was not utilized in this project, but remains open for future experimentation.

Second, lateral and longitudinal control are now handled by two separate nodes. This allows us to implement them independently of each other, and also allows the controllers to run as two separate processes on the computer. If we had both controllers inside a single node, then one controller would stall while waiting for the other one to update. Because the Odroid is a quad-core computer, both processes can run at the same time, allowing for faster and more accurate responses.

The third significant improvement relates to the arduino node. The separation of lateral and longitudinal control made it easy to split the ECU message into the MOT and SERV messages. This is important because the arduino node utilizes callback functions to respond to messages. Essentially, every time the node receives a message, it updates the BARC plant. Initially, we were sending ECU messages at a rate of 50Hz, containing both motor and servo values. It would not be possible to affect the rate of broadcast of either value without affecting the other one. Now, each node can have an independent rate, allowing these parameters to be tuned over time.

10 Traction Control Implementation

Our ideal traction controller should sustain the vehicle at a slip ratio that gives us the most tire traction force for a given surface. The optimal slip ratio should be predetermined via running multiple experiments on the same terrain to obtain the graph shown in Fig 1. Assuming that our ideal slip ratio is denoted via σ_{ideal}, we can implement a closed loop PID control that feeds back the current slip ratio of the vehicle. If the vehicle overshoots σ_{ideal}, our controller will automatically reduce the PWM signal to the motor, slowing ω_w.

Our traction control is best summarized via the following logic:

- Driver inputs his desired signal to simulate his pressing of the accelerator
- Slip ratio, σ is calculated
- If $\sigma \leq \sigma_{ideal}$: Maintain engine torque at driven input
- Else if $\sigma > \sigma_{ideal}$: Check if $\sigma - \sigma_{ideal} >$ tolerance. If true, implement traction control by slowing the speed. If false, maintain motor torque at driver input

This logic breaks down control of the vehicle both to the driver as well as to the controller. The traction controller only kicks in when the driver slams the accelerometer beyond the ideal

slip ratio to a certain degree of tolerance. Once control has been passed over to system, a PID loop is implemented to correct the system to the desired slip angle. The traction controller will always have control over the vehicle until the driver changes his input signal. Doing so will reset the computation from the very start.

To ensure a smooth handling of control between the driver and the traction controller, the vehicle was made to travel at the initial driver input. Any subsequent correction by the traction controller will cause the signal to corrected starting from driver input. This allows us to obtain a continuous transition of the vehicle's acceleration during control changeover.

11 Slip Ratio Measurement

Determining the slip ratio with the BARC was not as straightforward as we thought. Ideally, we would have had a speedometer to measure our velocity overtime. Even without an speedometer, we could have measured the velocity using the angular velocity and radius of a non-driven wheel. However, we only had an accelerometer and both the front and back tires were driven. This created an issue because we needed the car's longitudinal velocity to calculate the slip.

To solve this problem, we had to take the data from our accelerometer and integrate it to get the velocity. But, the acceleration data was very noisy, this made a direct integration nearly impossible. So, we ran it through a low pass filter to clean up the signal. Figure 6 shows the raw and filtered signal.

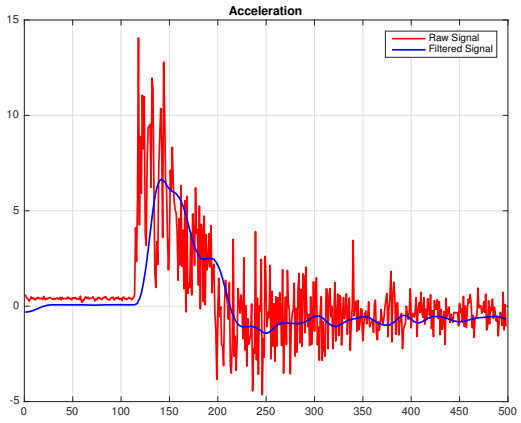


Figure 6: Tire Force Model

As we can see, the filter lead to a much cleaner signal. Using the filtered signal, we integrated to get the velocity. This velocity is the longitudinal velocity of the entire vehicle. This is one of the three components that we need to calculate the slip ratio. The other two are the effective radius, r_eff, and the angular velocity of the driven wheel, ω_w.

The effective radius is dependent on the size of the contact region, the static radius of the tire and the undeformed radius of the tire. These are defined in Figure 7.

As seen from Figure 7, $r_{stat} = r_w cos(\phi)$ and $a = r_w sin(\phi)$. Hence r_eff can be written as

$$r_{eff} = \frac{sin(\arccos(\frac{r_{stat}}{r_w}))}{\arccos(\frac{r_{stat}}{r_w})} r_w \tag{18}$$

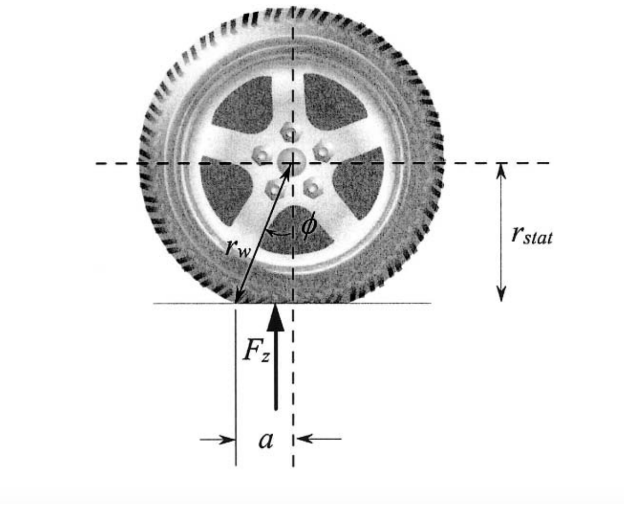


Figure 7: Tire Force Model

Simplifying this gives us

$$r_{eff} = \frac{sin(\phi)}{\phi} \tag{19}$$

The nominal radius of our wheel was 1.4595 inches The contact region was measured at 0.8105 inches , giving us $a = 0.4052$. Using these two values and the equations above gave us $r_{eff} = 1.4413$ inches. Next, we had to find the rotational velocity.

Finding the rotational velocity wasn't a straightforward process either. Because the wheel encoders record data in the form of clicks per second, we had to take the rate of these clicks by taking the numerical derivative. The encoder rate has units of $clicks/sec$. The encoder registers four clicks every revolution, $\frac{\pi}{2}$ rads/ click. When multiplying this by $\frac{\pi}{2}$ rads/ click we get units of rads/sec. Putting this all together, along with the effective radius, we get

$$r_{eff}\omega_w = rate * \frac{\pi}{2} * r_{eff} \tag{20}$$

We then took this value of $r_{eff}\omega_w$ and the calculated value of V_x to calculate our slip ratio.

12 Simulation Result

We ran the first simulation of the BARC at 115 for 2 seconds. The input range is 90-180, 90 being no velocity and 180 being max velocity. We ran this experiment in the hallways of Etcheverry Hall. Even with this relatively tame velocity input we found there to be a quite a bit of slip. This may be because of the ceramic surface the car was running on.

As we can see from Figure 8, the slip decreased at a much faster rate with the traction controller. We can also see in Figure 9 $r_{eff} * \omega_w$ is much smoother when the traction controller is implemented.

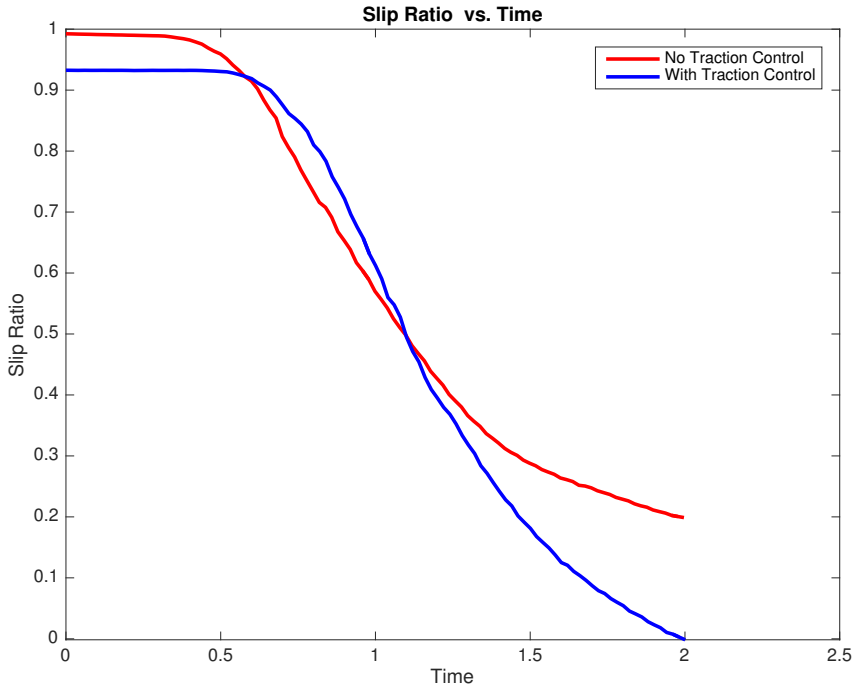


Figure 8: Slip Vs. Time

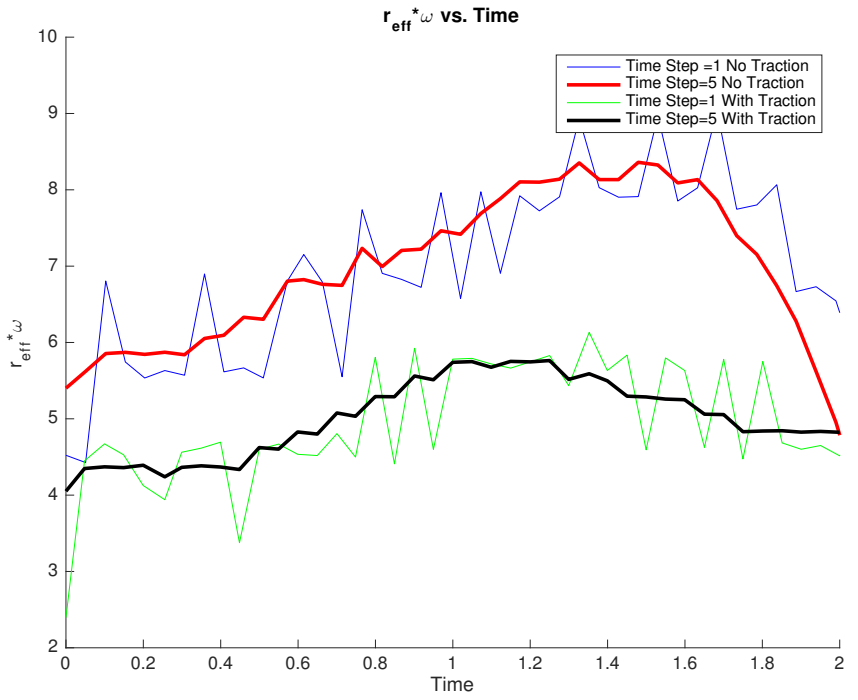


Figure 9: $r_{eff} * \omega$ vs. Time

For our second experiment we changed the environment and cranked up the velocity. These tests were run outside one concrete, a more realistic surface for a car to be running on. The input velocity was 140 for 2 seconds.

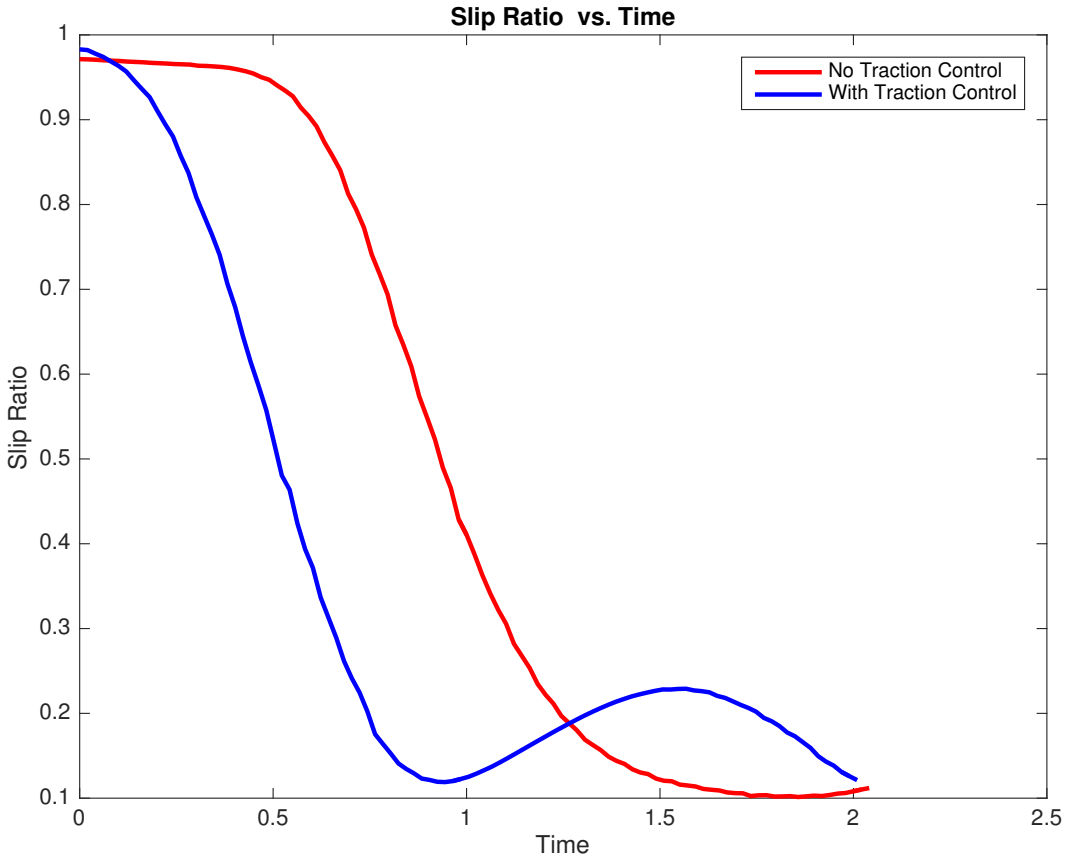
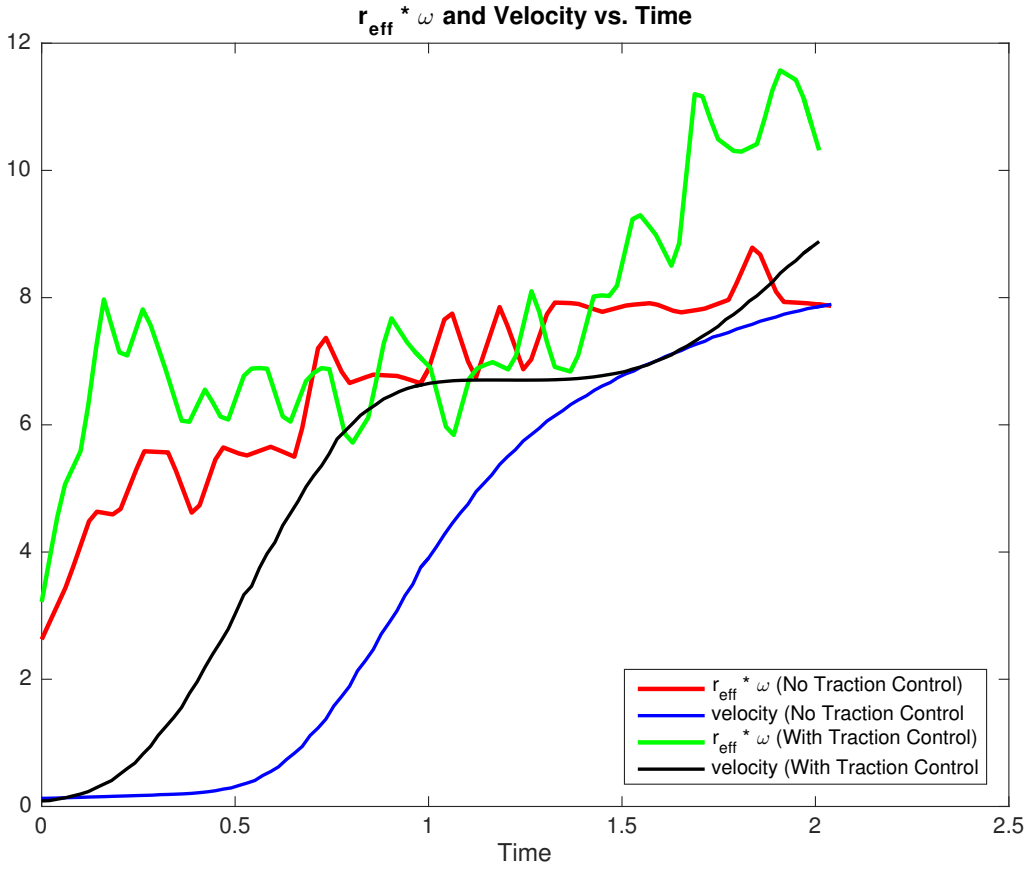


Figure 10

As we can see from Figure 10, the traction controller reduced the slip ratio at a much faster rate than with no traction controller. Also, Figure 13 shows us that despite having the same input, the traction controller run had a faster overall longitudinal velocity, a product of the reduced slip.

These simulation results show that our traction controller was successful in its goal to reduce slip ratio and provide a smoother ride experience for the driver.



13 Conclusion

A fully functional traction controller has been developed that improves acceleration and allows for better handling. The controller relies on a hall-effect encoder and an IMU in conjunction with a PID loop. Use of this controller has been shown to contribute to a faster acceleration from a dead stop.

This project was completed by Clayton Langbein, Johan Kok, and Maxwell Gerber. Clayton developed the theory and equations behind the slip ratio and the drive-line dynamic model as well as writing the Matlab code that processed and plotted the data. Maxwell worked on the BARC and wrote much of the python and c code used. Johan preformed data analysis and developed filters used to effectively determine Vx.

14 Appendix

References

[1] Basher rz-4 1/10 rally racer (pre-assembled kit), May 2016. [Online; accessed 3-May-2016].

[2] Thomas D. Gillespie. *Fundamentals of Vehicle Dynamics*. Society of Automotive Engineers, Inc., 400 Commonwealth Drive, Warrendale, PA, feb 1992.

[3] Jon Gonzales. Robotic operating system berkeley autonomous race car (barc) platform, Mar 2016.

[4] Jason M. OKane. *A Gentle Introduction to ROS*. University of South Carolina, 315 Main Street, Columbia, SC 29208, 2014.

[5] Bruce Wootton. Dator - cloud data for robots, Feb 2016. [Git repo readme; accessed 3-May-2016].

14.1 Parameters and Symbols

14.2 ROS Node Source Code

```
// include libraries
#include <ros.h>
#include <barc/Ultrasound.h>
#include <barc/Encoder.h>
#include <barc/MOT.h>
#include <barc/SERV.h>
#include <Servo.h>
#include "Maxbotix.h"

// Number of encoder counts on tires
// count tick on {FL, FR, BL, BR}
// F = front, B = back, L = left, R = right
volatile int FL_count = 0;
volatile int FR_count = 0;

//encoder pins: pins 2,3 are hardware interrupts
const int encPinA = 2;
const int encPinB = 3;

// Actuator pins: 5,6
// <Servo> data type performs PWM
// Declare variables to hold actuator commands
Servo motor;
Servo steering;
const int motorPin = 10;
const int servoPin = 11;
int motorCMD;
int servoCMD;
const int noAction = 0;

// Actuator constraints (servo)
// Not sure if constraints should be active on motor as well
int d_theta_max = 50;
int theta_center = 90;
int motor_neutral = 90;
int theta_max = theta_center + d_theta_max;
```

Parameter	Description
b_e	dampening coefficient of engine
h_d	height of drag force
r_{eff}	effective radius of rotating tire
A_b	Frontal Area
$C_{\sigma f}$	cornering stiffness of front tires
$C_{\sigma r}$	cornering stiffness of rear tires
F_{xf}	longitudinal tire force at the front tires
F_{xr}	longitudinal tire force at the rear tires
I_e	engine inertia
I_t	transmission shaft moment of inertia
T_a	accessory torque
T_f	torque frictional losses
T_p	pump torque
T_t	turbine torque
T_{eng}	engine torque
V_x	longitudinal vehicle velocity
ω_e	rotational engine speed
ω_t	angular speed of turbine on torque converter
ω_w	angular velocity of wheel
σ_x	slip ratio
σ_{xf}	front slip ratio
σ_{xr}	rear slip ratio

Table 5: Parameters and Symbols

```
int theta_min = theta_center - d_theta_max;
int motor_max = 180;
int motor_min = 0;

// variable for time
volatile unsigned long dt;
volatile unsigned long t0;

ros::NodeHandle nh;

// define global message variables
// Encoder, Electronic Control Unit, Ultrasound
barc::Ultrasound ultrasound;
barc::MOT mot;
barc::SERV serv;
barc::Encoder encoder;

ros::Publisher pub_encoder("encoder", &encoder);
ros::Publisher pub_ultrasound("ultrasound", &ultrasound);
```

```
/*****************************************************************************
MOT COMMAND CALLBACK
*****************************************************************************/
void messageCbMOT(const barc::MOT& mot){
  // deconstruct message
  motorCMD = saturateMotor( int(mot.motor_pwm) );
  // apply commands to motor and servo
  motor.write( motorCMD );
}
// ECU := Engine Control Unit
ros::Subscriber<barc::MOT> s("mot", messageCbMOT);

/*****************************************************************************
SERV COMMAND CALLBACK
*****************************************************************************/
void messageCbSERV(const barc::SERV& serv){
  // deconstruct message
  servoCMD = saturateServo( int(serv.servo_pwm) );
  // apply commands to motor and servo
  steering.write(  servoCMD );
}
// ECU := Engine Control Unit
ros::Subscriber<barc::SERV> t("serv", messageCbSERV);

// Set up ultrasound sensors
/*
Maxbotix us_fr(14, Maxbotix::PW, Maxbotix::LV); // front
Maxbotix us_bk(15, Maxbotix::PW, Maxbotix::LV); // back
Maxbotix us_rt(16, Maxbotix::PW, Maxbotix::LV); // right
Maxbotix us_lt(17, Maxbotix::PW, Maxbotix::LV); // left
*/

/*****************************************************************************
ARDUINO INITIALIZATION
*****************************************************************************/
void setup()
{
  // Set up encoder sensors
  pinMode(encPinA, INPUT_PULLUP);
  pinMode(encPinB, INPUT_PULLUP);
  attachInterrupt(0, FL_inc, CHANGE); // args = (digitalPintoInterrupt, ISR, mode), mode set = {LOW, CHANGE, RISI
  attachInterrupt(1, FR_inc, CHANGE); //pin 1 = INT1, which is pin D3

  // Set up actuators
  motor.attach(motorPin);
  steering.attach(servoPin);

    // Start ROS node
  nh.initNode();

  // Publish / Subscribe to topics
  nh.advertise(pub_ultrasound);
  nh.advertise(pub_encoder);
  nh.subscribe(s);
  nh.subscribe(t);

  // Arming ESC, 1 sec delay for arming and ROS
  motor.write(theta_center);
  steering.write(theta_center);
  delay(1000);
  t0 = millis();

}


/*****************************************************************************
ARDUINO MAIN lOOP
*****************************************************************************/
void loop()
{
```

```
    // compute time elapsed (in ms)
  dt = millis() - t0;

  // publish measurements
  if (dt >= 50) {
    // publish encodeer measurement

    encoder.FL = FL_count;
    encoder.FR = FR_count;
    encoder.BL = 0;
    encoder.BR = 0;
    pub_encoder.publish(&encoder);

    // publish ultra-sound measurement
    /*
    ultrasound.front = us_fr.getRange();
    ultrasound.back = us_bk.getRange();
    ultrasound.right = us_rt.getRange();
    ultrasound.left = us_lt.getRange();
    */
    pub_ultrasound.publish(&ultrasound);
    t0 = millis();
  }

  nh.spinOnce();
}

/*****************************************************************************
ENCODER COUNTERS
*****************************************************************************/
// increment the counters
void FL_inc() { FL_count++; }
void FR_inc() { FR_count++; }

/*****************************************************************************
SATURATE MOTOR AND SERVO COMMANDS
*****************************************************************************/
int saturateMotor(int x)
{
  if (x  == noAction ){ return motor_neutral; }

  if (x  >  motor_max) {
    x = motor_max;
  }
  else if (x < motor_min) {
    x = motor_min;
  }
  return x;
}

int saturateServo(int x)
{

  if (x  == noAction ){
    return theta_center;
  }

  if (x  >  theta_max) {
    x = theta_max;
  }
  else if (x < theta_min) {
    x = theta_min;
  }
  return x;
}
```

traction_driver.py

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from data_service.msg import TimeData
from barc.msg import SPEED
from math import pi,sin
import time
import serial
from numpy import zeros, hstack, cos, array, dot, arctan
from manuevers import TestSettings, StraightBrake

def driver():
    pub = rospy.Publisher('speed', SPEED, queue_size=10)
    rospy.init_node('driver', anonymous=True)

    rateHz  = 50
    dt      = 1.0 / rateHz
    rate    = rospy.Rate(rateHz)
    t_i     = 0

    speed = rospy.get_param("driver/speed")
    t_0 = rospy.get_param("driver/t_0")
    t_exp = rospy.get_param("driver/t_exp")

    opt         = TestSettings(SPD = speed, turn = 0, dt=t_exp)
    opt.t_0     = t_0

    while not rospy.is_shutdown():
        (motorCMD, _) = StraightBrake(opt, rateHz, t_i)
        pub.publish(SPEED(motorCMD))
        rate.sleep()
        t_i +=1

if __name__ == '__main__':
    try:
        driver()
    except rospy.ROSInterruptException:
        pass
```

traction_longitudinal.py

```
#!/usr/bin/env python

# ---------------------------------------------------------------------------
# Licensing Information: You are free to use or extend these projects for
# education or reserach purposes provided that (1) you retain this notice
# and (2) you provide clear attribution to UC Berkeley, including a link
# to http://barc-project.com
#
# Attibution Information: The barc project ROS code-base was developed
# at UC Berkeley in the Model Predictive Control (MPC) lab by Jon Gonzales
# (jon.gonzales@berkeley.edu). The cloud services integation with ROS was developed
# by Kiet Lam  (kiet.lam@berkeley.edu). The web-server app Dator was
# based on an open source project by Bruce Wootton
# ---------------------------------------------------------------------------

import rospy
import time
from data_service.msg import TimeData
from barc.msg import Encoder, MOT, STATE, SPEED
from math import pi,sin
import time
import serial
from numpy import zeros, hstack, cos, array, dot, arctan
from input_map import angle_2_servo, servo_2_angle
from pid import PID
from filtering import filteredSignal
from scipy.signal import filtfilt, butter, buttord

############################################################
# Set up measure callbacks
# imu measurement update
# TODO

prev_time       = 0
delta_time      = 0
v_x             = 0
speed           = 0
a_x             = 0
sleep_time      = 0.01

N                   = rospy.get_param("longitudinal/N")
Reff                = rospy.get_param("longitudinal/Reff")
threshold           = rospy.get_param("longitudinal/threshold")
counts              = zeros(N)
times               = zeros(N)
Ww_Reff             = 0
desired_slip_ratio  = 0.04
B                   = rospy.get_param("longitudinal/B")
buff = zeros(B)
(ordin, wn) = buttord(.1, .9, 3, 40)
(b,a) = butter(ordin, wn)
def imu_callback(data):
    global a_x, v_x, delta_time, prev_time, buff, b, a
    (_, _, _, a_x, _, _, _, _, _) = data.value
    buff = hstack(([ax], buff[:-1]))
    filtered_buff = filtfilt(b, a, buff)
    acc_filtered = filtered_buff[B/3]
    current_time    = data.timestamp
    delta_time      = current_time - prev_time      # Get the time difference inbetween call backs
    if(prev_time <= 0): delta_time = 0
    v_x += acc_filtered * delta_time                # Integrates accel to get velocity
    prev_time = current_time

# encoder measurement update
def encoder_callback(data):
    global counts, times, Ww_Reff
    counts = hstack(([data.FR],counts[:-1]))
    times = hstack(([time.time()], times[:-1]))
    Ww = (counts[0] - counts[-1])/(times[0] - times[-1])
```

```
    Ww_Reff = Ww * pi/2 * Reff

def speed_callback(data):
    global speed
    speed = data.speed

def filter_acc_x(acc_raw, filter_obj):
    filter_obj.update(a_x)
    return filter_obj.getFilteredSignal()
##############################################################
# main code
def main_auto():
    # initialize ROS node
    rospy.init_node('auto_mode', anonymous=True)
    rospy.Subscriber('imu', TimeData, imu_callback)
    rospy.Subscriber('encoder', Encoder, encoder_callback)
    rospy.Subscriber('speed', SPEED, speed_callback)
    nh = rospy.Publisher('mot', MOT, queue_size = 10)
    log = rospy.Publisher('state', STATE, queue_size = 10)

        # set node rate
    rateHz  = 50
    rate        = rospy.Rate(rateHz)
    dt      = 1.0 / rateHz
    p       = rospy.get_param("longitudinal/p")
    i       = rospy.get_param("longitudinal/i")
    d       = rospy.get_param("longitudinal/d")
    pid     = PID(P=p, I=i, D=d)

    # main loop
    while not rospy.is_shutdown():
        global v_x, Ww_Reff, desired_slip_ratio, delta_time, speed
        slip_ratio  = (Ww_Reff -v_x) /Ww_Reff
        err         = slip_ratio - desired_slip_ratio
        u = 90 + pid.update(err, dt)

        if err > threshold:
            motor_PWM = max(min(speed, u), 90)
        else:
            pid.reset()
            motor_PWM = speed

        # publish data
        nh.publish(MOT(motor_PWM))
        log.publish(STATE(v_x, Ww_Reff, err))

        # wait
        rate.sleep()

##############################################################
if __name__ == '__main__':
        try:
                main_auto()
        except rospy.ROSInterruptException:
        pass
```

traction_lateral.py

```
#!/usr/bin/env python

# ---------------------------------------------------------------------------
# Licensing Information: You are free to use or extend these projects for
# education or reserach purposes provided that (1) you retain this notice
# and (2) you provide clear attribution to UC Berkeley, including a link
# to http://barc-project.com
#
# Attibution Information: The barc project ROS code-base was developed
# at UC Berkeley in the Model Predictive Control (MPC) lab by Jon Gonzales
# (jon.gonzales@berkeley.edu). The cloud services integation with ROS was developed
# by Kiet Lam  (kiet.lam@berkeley.edu). The web-server app Dator was
# based on an open source project by Bruce Wootton
# ---------------------------------------------------------------------------

import rospy
from barc.msg import SERV
from data_service.msg import TimeData
from math import pi,sin
import time
import serial
from numpy import zeros, hstack, cos, array, dot, arctan, sign
from numpy import unwrap
from input_map import angle_2_servo, servo_2_angle
from manuevers import TestSettings, CircularTest, Straight, SineSweep, DoubleLaneChange, CoastDown
from pid import PID

# pid control for constrant yaw angle
yaw0        = 0
read_yaw0   = False
yaw_prev    = 0
yaw         = 0
err         = 0
def imu_callback(data):
    global yaw0, read_yaw0, yaw_prev, yaw, err

    # extract yaw angle
    (_,_,yaw, _,_,_, _,_,_) = data.value

    # save initial measurements
    if not read_yaw0:
        read_yaw0 = True
        yaw0    = yaw
    else:
        temp        = unwrap(array([yaw_prev, yaw]))
        yaw         = temp[1]
        yaw_prev    = yaw

    err = yaw - yaw0

############################################################
def main_auto():
    # initialize ROS node
    rospy.init_node('auto_mode', anonymous=True)
    nh = rospy.Publisher('serv', SERV, queue_size = 10)
    rospy.Subscriber('imu', TimeData, imu_callback)

        # set node rate
    rateHz  = 50
    dt      = 1.0 / rateHz
    rate        = rospy.Rate(rateHz)

    # use simple pid control to keep steering straight
    p           = rospy.get_param("lateral/p")
    i           = rospy.get_param("lateral/i")
    d           = rospy.get_param("lateral/d")
    pid     = PID(P=p, I=i, D=d)

    # main loop
```

```
    while not rospy.is_shutdown():
        # get steering wheel command
        u          = pid.update(err, dt)
        servoCMD   = angle_2_servo(u)

        # send command signal
        serv_cmd = SERV(servoCMD)
        nh.publish(serv_cmd)

        # wait
        rate.sleep()

#################################################################
if __name__ == '__main__':
        try:
                main_auto()
        except rospy.ROSInterruptException:
                pass
```

14.3 ROS Messages

ECU.msg

```
float32 motor_pwm
float32 servo_pwm
```

Ultrasound.msg

```
float32 front
float32 back
float32 right
float32 left
```

MOT.msg

```
float32 motor_pwm
```

SERV.msg

```
float32 servo_pwm
```

SPEED.msg

```
float32 speed
```

STATE.msg

```
float32 v_x
float32 Ww_Reff
float32 err
```

14.4 ROS Launch Files

openLoop_straight.launch

```
<launch>
        <!-- IMU NODE -->
        <node pkg="barc" type="imu_data_acquisition.py" name="imu_node" >
                <param name="port" value="/dev/ttyACM0" />
        </node>

        <!-- ARDUINO NODE -->
    <!--  * encoders and ultrasound sensors -->
        <node pkg="rosserial_python" type="serial_node.py" name="arduino_node" >
                <param name="port" value="/dev/ttyUSB0" />
        </node>

        <!-- OPEN LOOP MANUEVERS -->
        <node pkg="barc" type="controller_straight.py" name="controller" output="screen">
         <!--  SELECTION -->
                <param name="user" value="jgon13" />
                <param name="experiment_sel" type="int" value="4" />

         <!-- PROPERTIES -->
        <param name="v_x_pwm" type="int" value="96" />
                <param name="steering_angle" type="int" value="5" />
                <param name="t_exp" type="int" value="6" />
        <param name="t_0" type="int" value="2" />

        <!-- PID for straight test using imu gyro -->
        <param name="p" type="double" value="40" />
                <param name="i" type="double" value="5" />
                <param name="d" type="double" value="0" />

   </node>

    <!-- Record the experiment data -->
    <node pkg="rosbag" type="record" name="rosbag_record"
         args="_--all" />

</launch>
```

traction.launch

```
<launch>
        <!-- IMU NODE -->
        <node pkg="barc" type="imu_data_acquisition.py" name="imu_node" >
                <param name="port" value="/dev/ttyACM0" />
        </node>

        <!-- ARDUINO NODE -->
    <!--  * encoders and ultrasound sensors -->
        <node pkg="rosserial_python" type="serial_node.py" name="arduino_node" >
                <param name="port" value="/dev/ttyUSB0" />
        </node>

        <!-- DRIVER INPUT -->
        <node pkg="barc" type="traction_driver.py" name="driver" output="screen">
         <!-- PROPERTIES -->
        <param name="speed" type="int" value="150" />
                <param name="t_exp" type="int" value="2" />
        <param name="t_0" type="int" value="1" />
    </node>

        <!-- LATERAL CONTROL -->
        <node pkg="barc" type="traction_lateral.py" name="lateral" output="screen">

        <!-- PID for straight test using imu gyro -->
        <param name="p" type="double" value="30" />
                <param name="i" type="double" value="2" />
                <param name="d" type="double" value="0" />

    </node>

        <!-- LONGITUDINAL CONTROL  -->
        <node pkg="barc" type="traction_longitudinal.py" name="longitudinal" output="screen">

         <!-- PROPERTIES -->
        <param name="N" type="int" value="5" />
                <param name="Reff" type="double" value="1.3928" />
                <param name="threshold" type="double" value="0.1" />

        <!-- PID for straight test using imu gyro -->
        <param name="p" type="double" value="1" />
                <param name="i" type="double" value="0.01" />
                <param name="d" type="double" value="0.0001" />

    </node>

    <!-- Record the experiment data -->
    <node pkg="rosbag" type="record" name="rosbag_record"
         args="␣--all" />

</launch>
```

slip_ratio_calc.m

```
%% Setup
clear all
close all

%% User defined params

%No Traction Control
acc_filename = 'acc_x_nt';
enc_FR_filename = 'encoder_FR_nt';
enc_interv = 1;
acc_bias_len = 20;
low = 0.1;
high = 3;
r1 = 100;
r2 = 200;

% With Traction Control
acc_filename_trac = 'acc_x_t';
enc_FR_filename_trac = 'encoder_FR_t';
enc_interv_trac = 1;
acc_bias_len_tra = 20;
low = 0.1;
high = 3;
r1 = 100;
r2 = 200;


%% Load values

%No Traction Control
acc_raw = csvread(acc_filename);
acc_timestamp = acc_raw(:,1);
acc_data = acc_raw(:,2);

encoder_FR_data_raw = csvread('encoder_FR_nt');
encoder_FR_timestamp = encoder_FR_data_raw(:,1); %Take first column only
encoder_FR_data = encoder_FR_data_raw(:,2) - encoder_FR_data_raw(1,2); %Start at zero


% With Traction Control
acc_raw_trac = csvread(acc_filename_trac);
acc_timestamp_trac = acc_raw_trac(:,1);
acc_data_trac = acc_raw_trac(:,2);

encoder_FR_data_raw_trac = csvread(enc_FR_filename_trac);
encoder_FR_timestamp_trac = encoder_FR_data_raw_trac(:,1); %Take first column only
encoder_FR_data_trac = encoder_FR_data_raw_trac(:,2) - encoder_FR_data_raw_trac(1,2); %Start at zero
%% Calculate r_eff
in_to_m = 0.0254;
r_w=2.919/2; %in
r_deformed=2.76/2; %in
a=0.8105/2; %in
phi=sin(a/r_w);
r_eff=(sin(phi)*r_w)/phi;
r_eff_meters=in_to_m*r_eff
%% Calculate r_eff*omega

%No Traction Control
for k=1:length(encoder_FR_data) -enc_interv
    dt=encoder_FR_timestamp(k+ enc_interv)-encoder_FR_timestamp(k);
    rate=(encoder_FR_data(k+ enc_interv)-encoder_FR_data(k))/dt;
    r_eff_omega(k)=in_to_m *rate*(2*pi*r_eff)/4;
end
for k=1:length(encoder_FR_data) -5
    dt=encoder_FR_timestamp(k+ 5)-encoder_FR_timestamp(k);
    rate=(encoder_FR_data(k+ 5)-encoder_FR_data(k))/dt;
    r_eff_omega5(k)=in_to_m *rate*(2*pi*r_eff)/4;
end
```

```
% With Traction Control
for k=1:length(encoder_FR_data_trac) -enc_interv_trac
    dt_trac=encoder_FR_timestamp_trac(k+ enc_interv_trac)-encoder_FR_timestamp_trac(k);
    rate_trac=(encoder_FR_data_trac(k+ enc_interv_trac)-encoder_FR_data_trac(k))/dt_trac;
    r_eff_omega_trac(k)=in_to_m *rate_trac*(2*pi*r_eff)/4;
end
for k=1:length(encoder_FR_data_trac) -5
    dt_trac=encoder_FR_timestamp_trac(k+ 5)-encoder_FR_timestamp_trac(k);
    rate_trac=(encoder_FR_data_trac(k+ 5)-encoder_FR_data_trac(k))/dt_trac;
    r_eff_omega5_trac(k)=in_to_m *rate_trac*(2*pi*r_eff)/4;
end
%% Calculate Encoder Timescale
%No Traction Control
encoder_time=encoder_FR_timestamp(1:end -enc_interv)-encoder_FR_timestamp(1);
encoder_time2=encoder_FR_timestamp(1:end -5)-encoder_FR_timestamp(1);

%With Traction Control
encoder_time_trac=encoder_FR_timestamp_trac(1:end -enc_interv_trac)-encoder_FR_timestamp_trac(1);
encoder_time2_trac=encoder_FR_timestamp_trac(1:end -5)-encoder_FR_timestamp_trac(1);
%% Plot r_eff*omega

figure
hold on
plot(encoder_time,r_eff_omega, 'b')
plot(encoder_time2,r_eff_omega5, 'r','Linewidth',2)
plot(encoder_time_trac,r_eff_omega_trac, 'g')
plot(encoder_time2_trac,r_eff_omega5_trac, 'k','Linewidth',2)
xlim([0 2])
xlabel('Time','Fontsize',14)
ylabel('r_e_f_f*\omega')
title('r_e_f_f*\omega_vs._Time')
legend('Time_Step_=1_No_Traction','Time_Step=5_No_Traction','Time_Step=1_With_Traction','Time_Step=5_With_Tractio

%% Cut acc_x to match with encoder times

%No Traction
first = knnsearch(acc_data(:,1), encoder_FR_timestamp(1));
last = knnsearch(acc_data(:,1), encoder_FR_timestamp(end));

% With Traction Control
first_trac = knnsearch(acc_data_trac(:,1), encoder_FR_timestamp_trac(1));
last_trac = knnsearch(acc_data_trac(:,1), encoder_FR_timestamp_trac(end));

%% Plot Acceleration Vs. Time
figure(2)
plot(acc_timestamp, acc_data,'b','Linewidth',2);
hold all
plot(acc_timestamp_trac, acc_data_trac,'r','Linewidth',2);
ylabel('Acceleration')
xlabel('time')
title('Acceleration_wrt._time')
legend('No_Traction_Control','With_Traction_Control')

%% Apply filtering on acceleration data

%No Traction Control
F = acc_raw;
timestamp = F(:,1); acc = F(:,2);
acc_x_dt = timestamp(2:end) - timestamp(1:end-1);

d = fdesign.lowpass('Fp,Fst,Ap,Ast',low,high,1,40,50);
Hd = design(d,'equiripple');
acc_out = filter(Hd,acc);
acc_out = acc_out - max(acc_out(1:acc_bias_len));

smpl_size = length(acc);
dt = (max(timestamp) - min(timestamp)) /smpl_size;
smpl_rate = 1 /dt; % Hz;
df = smpl_rate/ smpl_size;

%Traction Control
```

```
F_trac = acc_raw_trac;
timestamp_trac = F_trac(:,1); acc_trac = F_trac(:,2);
acc_x_dt_trac = timestamp_trac(2:end) - timestamp_trac(1:end-1);

d_trac = fdesign.lowpass('Fp,Fst,Ap,Ast',low,high,1,40,50);
Hd_trac = design(d_trac,'equiripple');
acc_out_trac = filter(Hd_trac,acc_trac);
acc_out_trac = acc_out_trac - max(acc_out_trac(1:acc_bias_len_tra));

smpl_size_trac = length(acc_trac);
dt_trac = (max(timestamp_trac) - min(timestamp_trac)) /smpl_size_trac;
smpl_rate_trac = 1 /dt_trac; % Hz;
df_trac = smpl_rate_trac/ smpl_size_trac;
%% Plot Acceleration, Frequency Domain, and Velocity

figure(3)
sub = subplot(3,1,1);
%No Traction Control
plot(acc, 'r', 'linewidth' , 1.5);
hold all
plot(acc_out,'b', 'linewidth' , 1.5);
%With Traction Control
plot(acc_trac, 'g', 'linewidth' , 1.5);
plot(acc_out_trac,'y', 'linewidth' , 1.5);

hold off;
grid on;
title('Acceleration')
legend('Raw Signal (No Traction)','Filtered Signal (No Traction)','Raw Signal (With Traction Control)','Filtered
% Plot FFT processed data
subplot(3,1,2)
%No Traction Control
valid_reg = ceil(smpl_size /2);
fd = fft(acc);
freq_rng = 0 : df : df*(valid_reg -1);
freq_dom = fd(1:valid_reg);
plot(freq_rng', abs(freq_dom));
hold all
%With Traction Control
valid_reg_trac = ceil(smpl_size_trac /2);
fd_trac = fft(acc_trac);
freq_rng_trac = 0: df_trac : df_trac*(valid_reg_trac -1);
freq_dom_trac = fd_trac(1:valid_reg_trac);
plot(freq_rng_trac', abs(freq_dom_trac));
title('Frequency Domain')
legend('No Traction Control','With Traction Control')
grid on;

subplot(3,1,3)
%No Traction Control
v_x = cumsum(acc_out(2:end) .* acc_x_dt);
velocity_time=timestamp(1:end-1)-timestamp(1);
plot(velocity_time, v_x, 'linewidth', 1.5);
hold all
%With Traction Control
v_x_trac = cumsum(acc_out_trac(2:end) .* acc_x_dt_trac);

velocity_time_trac=timestamp_trac(1:end-1)-timestamp_trac(1);

plot(velocity_time_trac, v_x_trac, 'linewidth', 1.5);
title('Velocity')
legend('No Traction Control','With Traction Control')
hold off
grid on
%% Creating Time Array

%No Traction Control
ts = timestamp(r1:r2);
v_x_enc = v_x(r1:r2);
indx_start = knnsearch(encoder_FR_timestamp, min(ts));      % Searches for the closest matching timestamp in v_x
indx_end = knnsearch(encoder_FR_timestamp, max(ts));
```

```
indx_diff = indx_end -indx_start +1;
t_prior = linspace(min(ts), max(ts), indx_diff);    % Assume that the region of interest has the same timing for
t_target = linspace(min(ts), max(ts), r2 -r1 +1);

omega_new = interp1q(t_prior',r_eff_omega(indx_start:indx_end)', t_target');

P = polyfit(t_target', omega_new, 1);
omega_fit = P(1)*t_target'+P(2);
slip_ratio = (omega_fit - v_x_enc)./omega_fit;
t_target=t_target-t_target(1);

% (With Traction Control)
ts_trac = timestamp_trac(r1:r2);
v_x_enc_trac = v_x_trac(r1:r2);
indx_start_trac = knnsearch(encoder_FR_timestamp_trac, min(ts_trac));      % Searches for the closest matching ti
indx_end_trac = knnsearch(encoder_FR_timestamp_trac, max(ts_trac));
indx_diff_trac = indx_end_trac - indx_start_trac +1;
t_prior_trac = linspace(min(ts_trac), max(ts_trac), indx_diff_trac);    % Assume that the region of interest has
t_target_trac = linspace(min(ts_trac), max(ts_trac), r2 -r1 +1);

omega_new_trac = interp1q(t_prior_trac', r_eff_omega_trac(indx_start_trac:indx_end_trac)', t_target_trac');

P = polyfit(t_target_trac', omega_new_trac, 1);
omega_fit_trac = P(1)*t_target_trac'+P(2);
slip_ratio_trac = (omega_fit_trac - v_x_enc_trac)./omega_fit_trac;
t_target_trac=t_target_trac-t_target_trac(1);
%% Plot r_eff*omega and Velocity

figure;
%No Traction Control
plot(t_target',omega_new,'r','Linewidth',2)
hold all
plot(t_target', v_x_enc,'b', 'linewidth', 1.5);
%With Traction Control
plot(t_target_trac',omega_new_trac,'g','Linewidth',2)
plot(t_target_trac', v_x_enc_trac,'k', 'linewidth', 1.5);
xlabel('Time','Fontsize',14)
title('r_e_f_f_*\omega and Velocity vs. Time')
legend('r_e_f_f_*\omega (No Traction Control)','velocity (No Traction Control','r_e_f_f_*\omega (With Traction

%% Plot Slip Ratio vs. Time

figure;
%No Traction Control
plot(t_target',slip_ratio,'r','Linewidth',2);
%With Traction Control
hold all
plot(t_target_trac',slip_ratio_trac,'b','Linewidth',2);
xlabel('Time','Fontsize',14)
ylabel('Slip Ratio','Fontsize',14)
title('Slip Ratio  vs. Time','Fontsize',18)
legend('No Traction Control','With Traction Control')
```