Tip-Up and Stabilization of an Autonomous Four-Wheeled Vehicle

by

Justin T. Lan

Submitted to the Department of Mechanical Engineering
in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Mechanical Engineering

at the

Massachusetts Institute of Technology

June 2011

Signature of Author: [illegible]
Department of Mechanical Engineering
May 5, 2011

Certified by: [illegible]
Karl Iagnemma
Thesis Supervisor

Accepted by: [illegible]
John H. Lienhard V
Professor of Mechanical Engineering
Chairman, Undergraduate Thesis Committee

Tip-Up and Stabilization of an Autonomous Four-Wheeled Vehicle

by

Justin T. Lan

Submitted to the Department of Mechanical Engineering
on May 5, 2011 in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Mechanical Engineering

ABSTRACT

An instrumented four-wheeled robot was constructed to investigate steering-induced tip-up and rollover of four-wheeled vehicles, and the possibility of stabilizing and driving while balanced on two wheels. Using an analogy to a cart-pole controller, an energy shaping controller was developed to drive the vehicle's roll energy to a fixed value and stabilize the vehicle in a tipped-up position. The controller was implemented on experimental hardware, which included an ARM7-based FEZ Domino microcontroller and a MicroStrain 3DM-GX2 inertial measurement unit. Data logged during experimental trials were analyzed in MATLAB to assess the stability and effectiveness of the controller. Experimental results indicate the controller is capable of driving the vehicle to the tipped-up balancing position. The addition of a steering angle sensor and tachometer are suggested to improve the effectiveness of the controller.

Thesis Supervisor: Karl Iagnemma
Title: Principal Research Scientist

Acknowledgements

The author would like to thank Karl Iagnemma for his guidance and assistance in obtaining materials, and Steven Peters for his support and suggestions. The author would also like to thank Christina Jaworsky for her assistance in obtaining materials.

Table of Contents

Introduction

When cornering at high speeds in a four-wheeled vehicle, vehicle rollover is a major concern, particularly in the case of vehicles with a high center of gravity such as SUVs and trucks. However, stunt car drivers have been able to balance vehicles while they are tipped-up and drive on two wheels on the same side. The ability to slow rollover and stabilize in the tilted position has numerous applications in vehicle safety and robot mobility.

The primary goal of this project was to modify a radio-controlled vehicle to autonomously initiate rollover maneuvers. Once rollover has been initiated the vehicle self-balances and continues driving on two wheels on the same side, a stunt known as "skiing". Although a self-balancing car-type robot that drives while "skiing" has been created [1], the complete problem of tip-up and balancing has not been carried out to the author's knowledge. The ability to autonomously react to impending rollover and stabilize has applications in both vehicle safety and augmented mobility for vehicles and robots.

Model

I. Cart-Pole Mechanics

A close approximation of the mechanics of a vehicle balancing while "skiing" is the cart-pole model, illustrated in Figure 1.

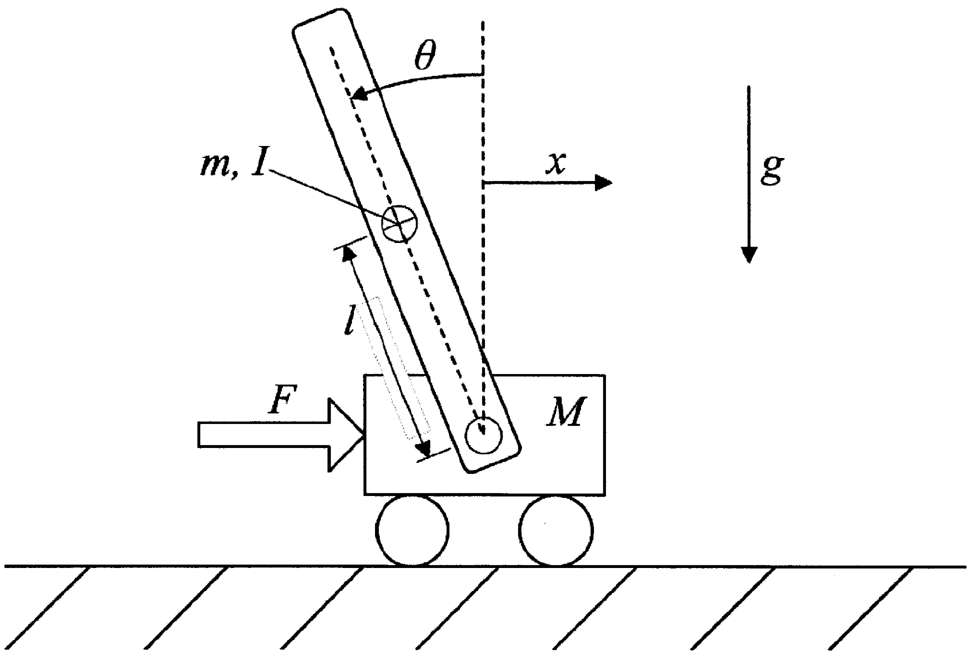

Figure 1: A diagram of a cart-pole system. The cart has mass *M*, and the pole has mass *m* and moment of inertia *I*. The pole is connected to a cart through a pivot located a distance *l* away from the center of mass.

The cart-pole system consists of a cart with mass *M* connected through a free pivot to a pendulum with mass *m*, length *l*, and rotational inertia *I*. A force *F* is applied to the cart, but the pivot is not actuated.

The dynamics of the cart-pole system are given by the system of equations

$$\begin{bmatrix} M+m & -ml^2\cos\theta \\ -ml^2\cos\theta & I+ml^2 \end{bmatrix}\begin{bmatrix}\ddot{x}\\ \ddot{\theta}\end{bmatrix}+\begin{bmatrix}0 & ml\dot{\theta}\sin\theta \\ 0 & 0\end{bmatrix}\begin{bmatrix}\dot{x}\\ \dot{\theta}\end{bmatrix}+\begin{bmatrix}0\\ -mgl\sin\theta\end{bmatrix}=\begin{bmatrix}F\\ 0\end{bmatrix}, \quad (1)$$

where *x* is the horizontal position of the cart and *θ* is the angle of the pendulum [2].

II. Vehicle in Rollover

In the event that a vehicle enters rollover its dynamics can be approximated using a cart-pole model so long as one set of wheels remains on the ground. The system is equivalent to a cart-pole system with a massless cart, with the horizontal driving force created from steering dynamics applied at the tire-ground interface, as seen in Figure 2.

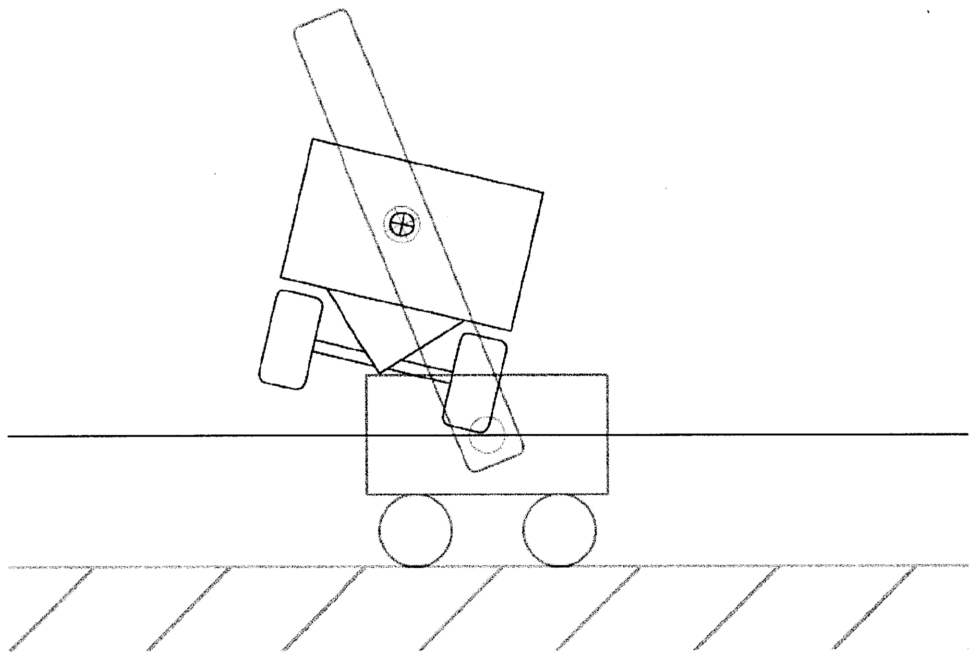

Figure 2: An illustration of the similarity between the cart-pole problem and vehicle tip-up.

As the tire forces on a vehicle are generated by friction, there is a practical limit on the available driving force *F* such that

$$-\mu N \leq F \leq \mu N, \tag{2}$$

where μ is the coefficient of static friction between the tires and the driving surface and *N* is the contact force between the tires and the ground.

The net force vector formed by the combination of the friction force and normal force makes an angle α with the vertical. At the limits of friction, this angle is equivalent to $\tan^{-1}(\mu)$, as illustrated in Figure 3.

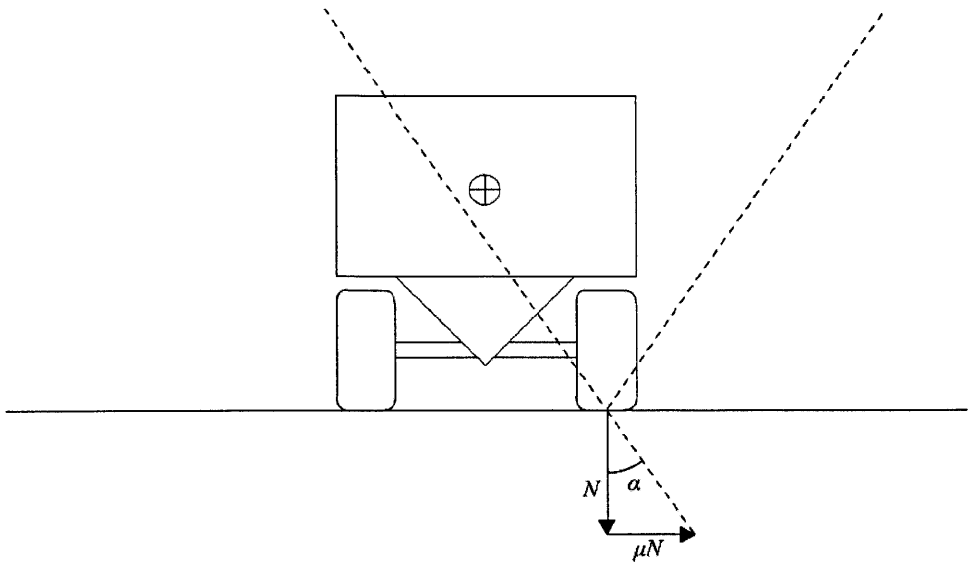

Figure 3: An illustration of the friction cone formed by the normal force and maximum friction force. If the c.g. of the vehicle is within this friction cone, tip-up is possible.

The area between the limiting angles is known as the friction cone. The line of action of all possible combinations of the friction and normal forces lies within the friction cone. If the c.g. of the vehicle lies outside the friction cone, frictional forces are unable to rotate the vehicle away from the ground, preventing the vehicle from tipping up and rolling over. However, if the vehicle's c.g. is moved within the friction cone, tip-up becomes possible again. This may be accomplished by driving the vehicle on an angled surface or a sudden vertical impulse to one side of the vehicle, as might happen when a vehicle travels over a bump. Additionally, if the vehicle has suspension, the center of mass of the vehicle may sway into the friction cone during travel over uneven ground or due to lateral forces during turns.

Hardware Design

I. Chassis

As this research intends to simulate the effects of tip-up and subsequent stabilization in a typical 4-wheeled vehicle, a suitable model had to be chosen. The robot is based on the chassis of an HPI Racing Wheely King 1/12 scale RC monster truck, seen in Figure 4.

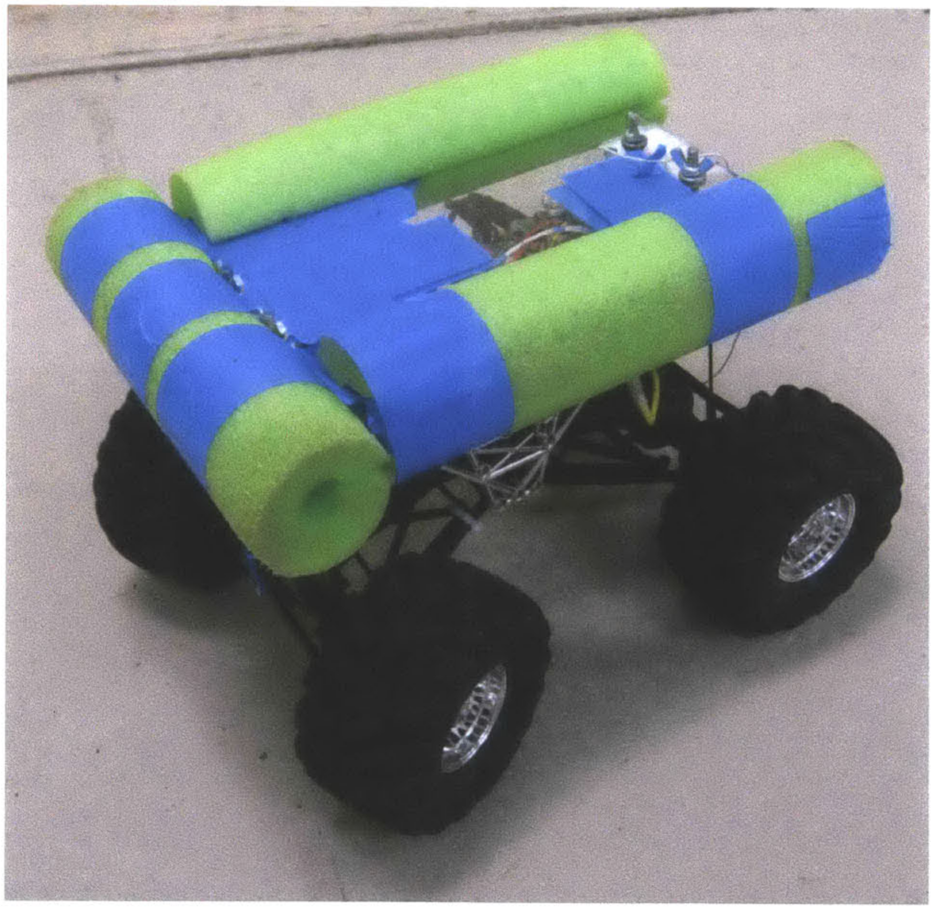

Figure 4: The robot used in the study. The sides have been padded with foam to cushion the vehicle during rollover.

The vehicle's is an ideal candidate for investigating tip-up, as its relatively high center of gravity makes tip-up maneuvers easier to perform. Additionally, the vehicle is four-wheel-drive with front wheel steering, giving it similar control characteristics to full-sized vehicles. This particular vehicle was inherited from a previous study on balancing vehicles in the tipped-up position [1].

II. Chassis Modifications

For the purposes of this project, several modifications were made to the vehicle. First, the decorative "shell" of the vehicle was removed; this was replaced with an acrylic cover supported on 4 threaded rods. The cover protects the vehicle's electronics from impacts, dust, and weather conditions. The edges of the cover were padded to cushion the vehicle against rollover impacts. The vehicle's radio antenna protrudes through a small hole in the protective cover. The cover is held in place with wing nuts; these can be easily removed in the event the cover needs to be removed to service the vehicle electronics.

Additionally, the compliant shock absorbers of the vehicle were replaced with solid rods, "locking" the suspension, as seen in Figure 5. This modification removed the additional degrees of freedom created by the suspension system, simplifying dynamic analysis of the vehicle and the balancing task.

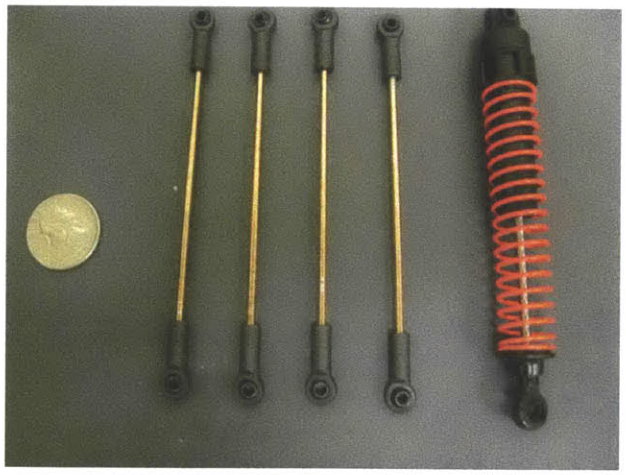

Figure 5: Solid rods (center) used to replace the vehicle's shock absorbers (right). A U.S. quarter is included for scale.

Finally, the steering servo has been moved, as seen in Figure 6. This modification had been previously carried out by the authors of a previous study from which this vehicle was inherited [1]. The servo was initially mounted high on the vehicle chassis and controlled the steering via a series of linkages. To avoid excessive compliance in the vehicle's steering system the servo was attached instead to the front differential and connected to the steering via a much shorter single steering link. This change has the additional benefit of decoupling the actions of the vehicle's steering and suspension systems, in the event that a compliant suspension is reintroduced on the vehicle.

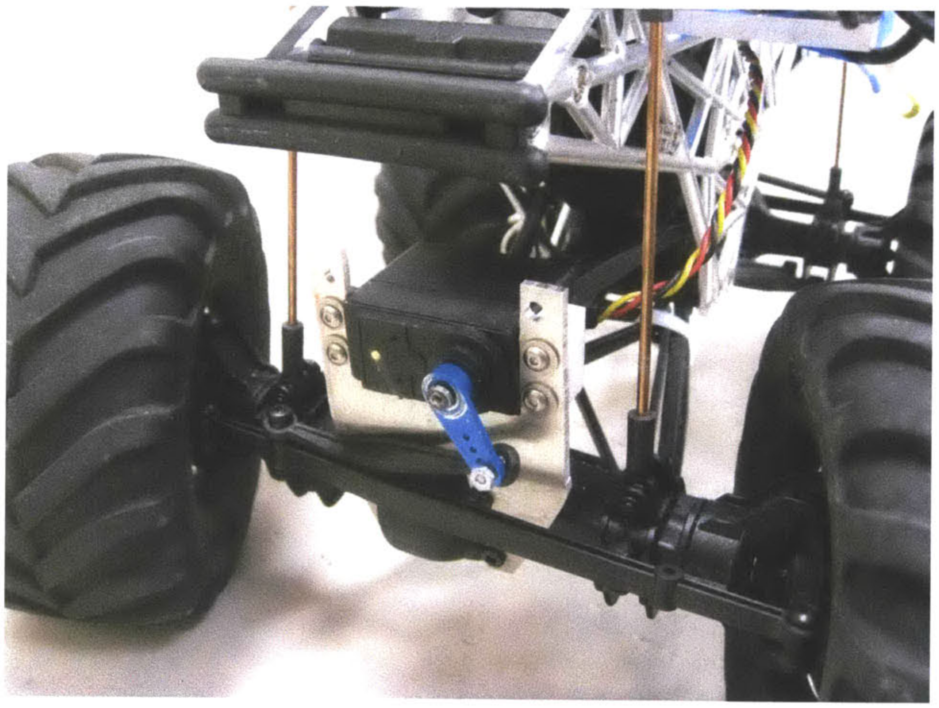

Figure 6: Close-up of the modified steering servo assembly. This image also shows the solid rods of Figure 5 mounted to the vehicle.

III. Electronics Mounting

Due to the irregular shape of the vehicle's chassis, which was designed to emulate the appearance of a frame made of welded steel tubes, flat areas for mounting sensors and electronics were limited. Therefore, ¼-inch acrylic sheets were mounted to various parts of the chassis to provide flat surfaces; the electronics were in turn mounted to these acrylic sheets.

Electronics

I. Inertial Measurement Unit

The vehicle is equipped with an Inertia-Link® 3DM-GX2® inertial measurement unit (IMU) manufactured by MicroStrain, Inc [3]. The 3DM-GX2 features a three-axis accelerometer, gyroscope, and magnetometer, as well as a temperature sensor and an onboard processor that merges data from these sensors. The sensor is powered by and transmits data via a USB 2.0 connection. The sensor was chosen due to its small size, flexibility, and ability to communicate over USB.

II. Microcontroller

The robot is controlled by a GHI Electronics FEZ Domino microcontroller [4]. The FEZ Domino is based around 72 MHz ARM7 processor and incorporates USB host functionality, enabling it to connect to the 3DM-GX2. Additionally, the FEZ Domino features a Micro SD card connector, allowing for easy data logging. The data can then be transferred to a computer for later analysis without the need to connect the robot to a computer.

III. Radio and Motor Controller

The robot features a commercial hobby radio control system. This system is used to send throttle and steering commands to the FEZ Domino. The pulse-width modulation (PWM) signals output by the radio receiver are read using interrupt pins on the FEZ Domino. The throttle signal is echoed to a M.sonik-4 Electronic Speed Controller (ESC), allowing remote control of the vehicle's forward speed.

IV. Wiring

The entire vehicle is powered by a single onboard 7.2 battery. The battery is a commercial six-cell rechargeable NiMH battery intended for use with radio-controlled vehicles. Power is supplied directly to the vehicle's electronic speed controller and to the FEZ Domino, which in turn supplies filtered power to the IMU, radio, and steering servomotor. A simplified wiring and information-flow diagram can be seen in Figure 7.

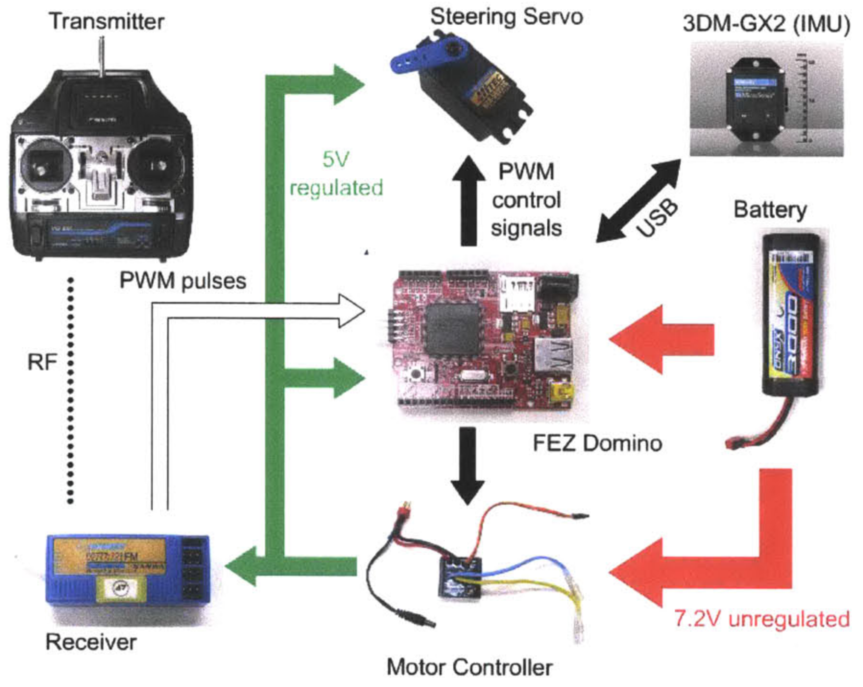

Figure 7: Diagram showing power and information flow on the robot. Unregulated battery power is supplied to both the speed controller and the FEZ Domino, which then produce regulated 5V power for the other onboard electronics.

Software

I. Software Environment

The robot is programmed in C# using Microsoft's Visual Studio development environment. The FEZ Domino runs Microsoft's .NET Micro Framework with a library extension created by GHI Electronics for the FEZ Domino.

II. Program Architecture

The robot's program is based around a master thread and a control thread. The master thread is responsible for overall program flow control, including the starting and stopping of data logging. The control thread reads sensor and command data and outputs control signals to the steering servo in order to tip-up and control the vehicle. In addition, interrupt signals capture throttle and steering commands from the radio and handle connection to the IMU over USB.

Algorithm

I. Lateral Force Controller

The lateral forces on a vehicle are the result of friction dynamics while steering. In particular, steering forces are strongly affected by the difference between the steering angle and the direction of travel of a moving vehicle, an quantity known as the slip angle [5]. As the actual lateral force as a function of steering angle and vehicle speed was not known, the steering rate was controlled using a proportional controller of the form

$$\dot{\delta} = K_\delta(a_d - a_{lat}) \,, \tag{3}$$

where $\dot{\delta}$ is the steering *rate*, K_δ is a gain, a_d is the desired lateral force, and a_{lat} is the measured lateral force. Using this controller, the steering angle δ will stabilize when $a_{lat} = a_d$. The maximum steering rate is clamped to values actually realizable with the steering actuator mounted on the vehicle.

However, the controller is required to track a constantly changing a_{lat}. Therefore, the phase lag that results from using a simple proportional controller was reduced with the addition of a feedforward term predicted using a no-slip model to approximate the vehicle mechanics. In this study the vehicle was driving on asphalt, a surface with a relatively high coefficient of friction. Thus, the radius of the vehicle's turn r is approximated using the formula

$$r = \frac{L}{\sin\delta}, \tag{4}$$

where L is the length of the vehicle's wheelbase and δ is the steering angle, as seen in the "bicycle model" diagram in Figure 8.

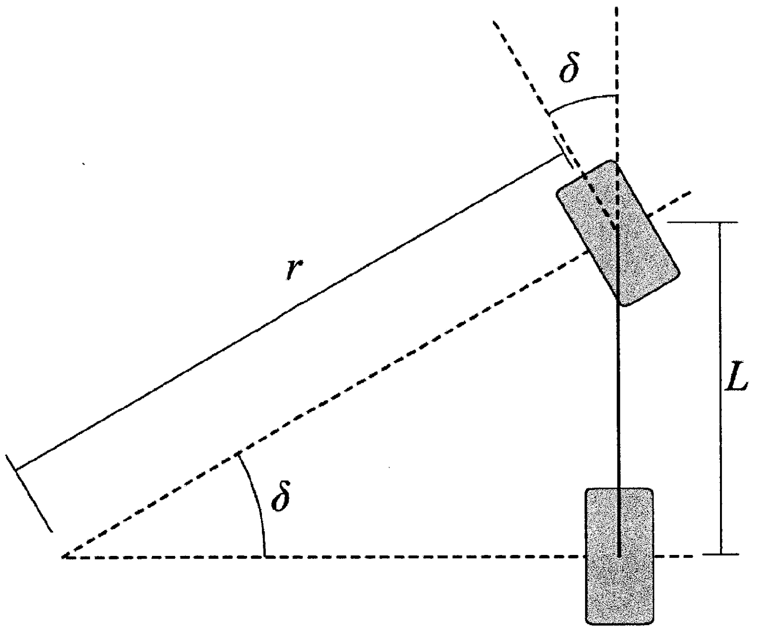

Figure 8: Diagram of a two-wheeled vehicle in a turn. Using this bicycle model as an approximation for a four-wheeled vehicle, the turn radius r can be derived from the steering angle δ and the wheelbase length L.

In this approximation the center of the vehicle is assumed to follow the path of the turn. In this case, the centripetal, and hence lateral, acceleration of the vehicle is given by

$$a_c = \frac{v^2}{r} = \frac{v^2 \sin\delta}{L}, \tag{5}$$

where a_c is the centripetal acceleration and v is the forward velocity of the vehicle. This forward velocity was estimated by first mapping the commanded throttle pulse to a velocity, then gradually increasing an internal simulated velocity towards this commanded velocity. The mapping between commanded throttle pulses and velocity was determined experimentally.

Using this model, a "naïve" desired steering angle was predicted using the equation

$$\delta_n \cong \frac{L}{v^2} a_d, \tag{6}$$

with saturation limits to keep δ_n within physical limits. This angle was then combined with the desired steering angle from the proportional controller,

$$\delta_p = \dot{\delta} \cdot \Delta t, \tag{7}$$

where $\dot{\delta}$ is the steering rate defined in (3) and Δt is the time step of the controller, equal to the inverse of the controller frequency. This combination yields a controller with the diagram seen in Figure 9.

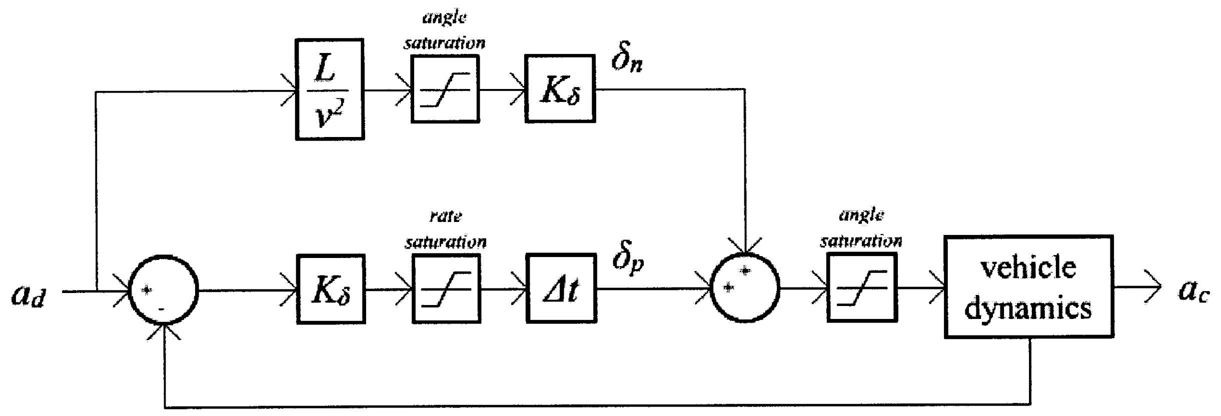

Figure 9: Diagram of the lateral force controller.

II. Analogy Between Vehicle in Rollover and Cart-Pole

In order to treat vehicle rollover as a cart-pole problem a conversion must be made between the two. The cart is reduced to a massless contact point between the wheels and the ground, and the driving force is provided by ground forces generated by steering. The body of the vehicle is analogous to the pendulum; however, there is a discontinuity in the angle of the pendulum, owing to a change in mode between balancing on the left wheels and the right wheels, as illustrated in Figure 10.

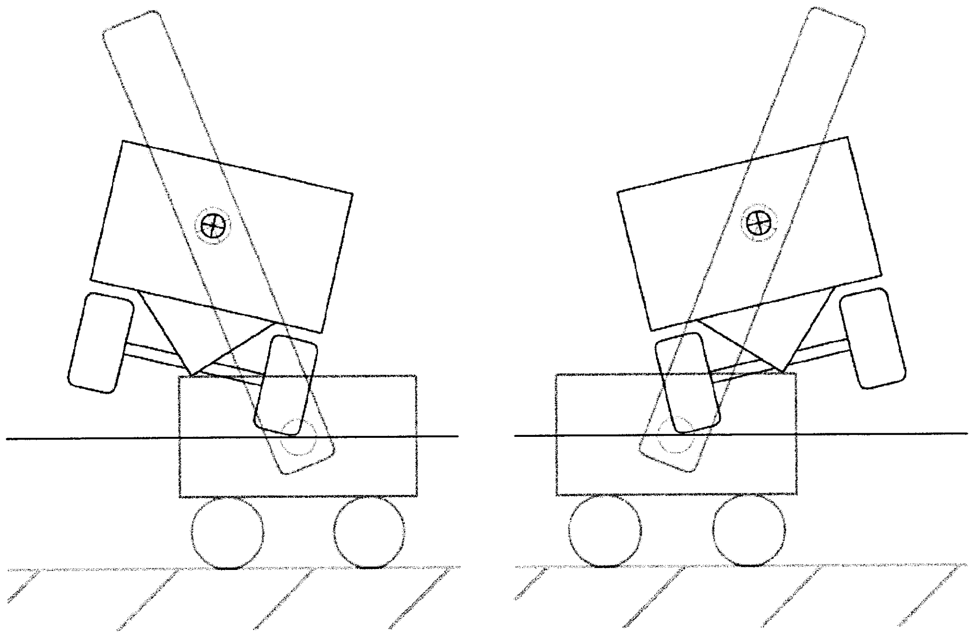

Figure 10: A discontinuity in the equivalent pendulum angle for the vehicle occurs as the vehicle roll angle passes through zero.

The mapping between vehicle roll angle and effective pendulum is resolved with the equations

$$\theta = \begin{cases} \psi + \varphi_{internal} - 90°, & \psi \geq 0 \\ 90° + \psi - \varphi_{internal}, & \psi < 0 \end{cases}, \tag{8}$$

where θ is the effective pendulum angle, ψ is the roll angle, and $\varphi_{internal}$ is the "internal angle" of the vehicle's center of mass, such that

$$\varphi_{internal} = \tan^{-1}\left(\frac{z_{cg}}{y_{cg}}\right), \tag{9}$$

where y_{cg} and z_{cg} are measured in the vehicle frame as shown in Figure 11.

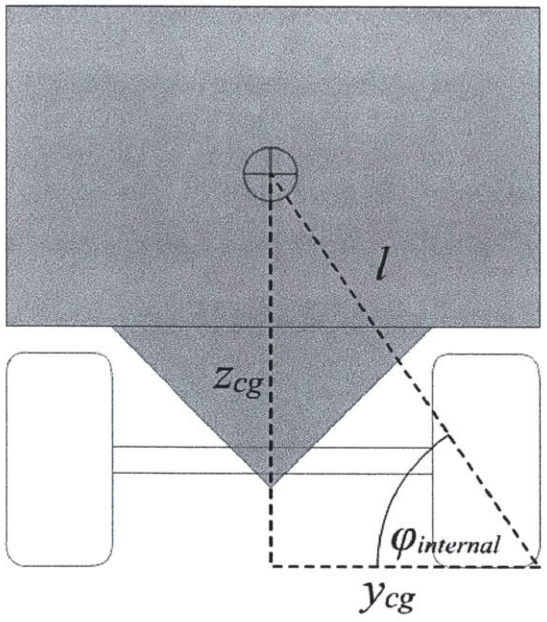

Figure 11: Diagram showing location of vehicle c.g. and related measurements $\varphi_{internal}$ and l.

These equations create the mapping shown in Figure 12.

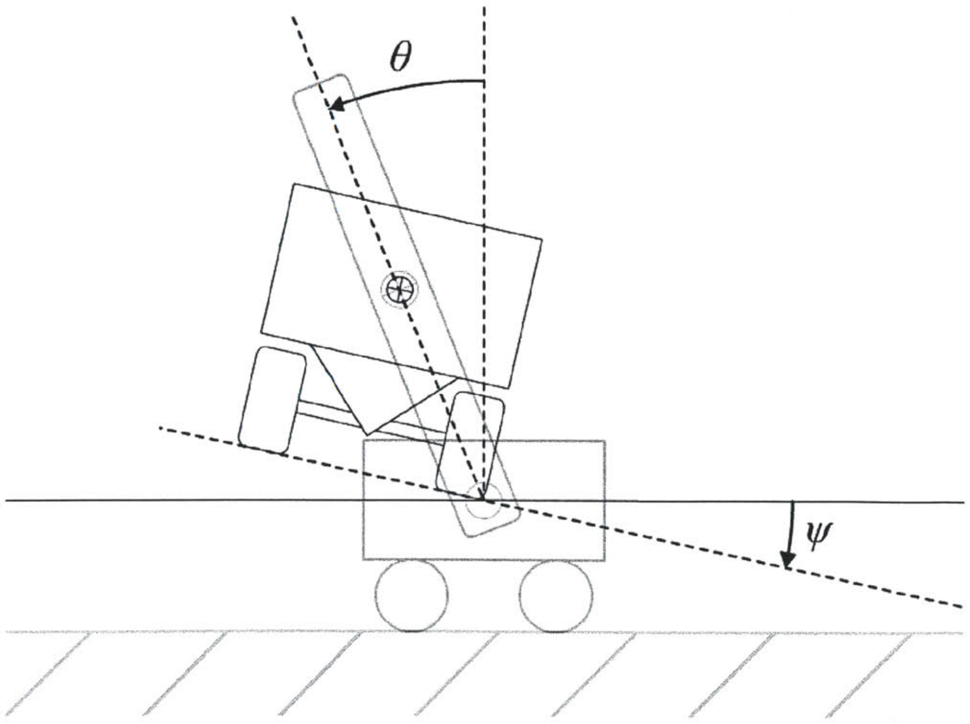

Figure 12: Mapping between roll angle and equivalent pendulum angle. ψ is defined as positive counterclockwise from the ground plane and θ is defined as positive counterclockwise from vertical.

III. Energy Shaping Controller

For certain mechanical systems, such as the cart-pole, a constant energy curve produces a homoclinic orbit that passes through points of unstable equilibrium. Thus, by driving the system's energy to that constant value, the system will be driven to the unstable equilibrium point. In the case of a vehicle in rollover, this constant energy curve produces a homoclinic path rather than an orbit, as the equivalent pendulum angle is discontinuous. Regardless, the energy shaping method may still be used to drive the vehicle towards a tipped-up position.

Given the mapping between roll angle and effective pendulum angle, the roll energy is found to be

$$E = mgl\cos\theta + \frac{1}{2}(I + ml^2)\dot{\theta}^2, \tag{10}$$

where m is the mass of the vehicle, g is the gravitational acceleration, I is the rotational inertia of the vehicle about its center of mass, and l is the effective length of the "pendulum", found to be

$$l = \sqrt{{y_{cg}}^2 + {z_{cg}}^2}. \tag{11}$$

In this study, I was found to be relatively small compared to ml^2, and was neglected to simplify analysis.

Using this definition of roll energy, the roll energy error is defined to be

$$\tilde{E} = E - E_{des} = E - mgl, \tag{12}$$

because at tip-up, $\theta = 0$ and $\dot{\theta} = 0$, so $E_{des} = mgl\cos\theta = mgl$. Taking the derivative of the roll energy error yields

$$\dot{\tilde{E}} = \dot{E} = -mgl\dot{\theta}\sin\theta + ml^2\dot{\theta}\ddot{\theta}. \tag{13}$$

The equations of motion for the cart-pole yield

$$ml\cos\theta\,\ddot{x} = ml^2\ddot{\theta} + ml\sin\theta. \tag{14}$$

Combining (13) and (14) yields

$$\dot{E} = ml\dot{\theta}\cos\theta\,\ddot{x}. \tag{15}$$

Using the control law

$$\ddot{x} = \ddot{x}_d = -K_1\dot{\theta}\cos\theta\,\tilde{E}, \tag{16}$$

where K_1 is a gain, results in

$$\dot{E} = -K_1 ml\dot{\theta}^2\cos^2\theta\,\tilde{E}, \tag{17}$$

a stabilizing controller for E. To stabilize at the unstable equilibrium point of tip-up then requires

$$\ddot{x}_d = \begin{cases} -K_1\dot{\theta}\cos\theta\,\tilde{E}, & |\psi| \geq \psi_{transition} \\ K_2\dfrac{g}{\tan\varphi_{internal}}, & |\psi| < \psi_{transition} \end{cases}, \tag{18}$$

where $K_2 > 1$ to produce an acceleration sufficient to tip up the vehicle.

In an attempt to slow the process of rollover and improve its controllability, a countersteering term was added of the form

$$\ddot{x}_{cs} = -K_{cs}\dot{\theta}\cos\psi. \tag{19}$$

The roll dependence is used to reduce the effect of the countersteer term as the vehicle nears the balancing angle. The countersteer and energy shaping terms are then combined to produce a desired lateral acceleration, which is then fed to the lateral acceleration controller.

Results

I. Experimental Results

The control system outlined in the section above was implemented on the vehicle hardware. During these experiments, the throttle command to the vehicle was controlled by the operator via the radio transmitter. While the control system was able to tip the vehicle onto two wheels, it was unable to support the vehicle there for more than 1 second. Figure 13 shows the roll angle during an unsuccessful run.

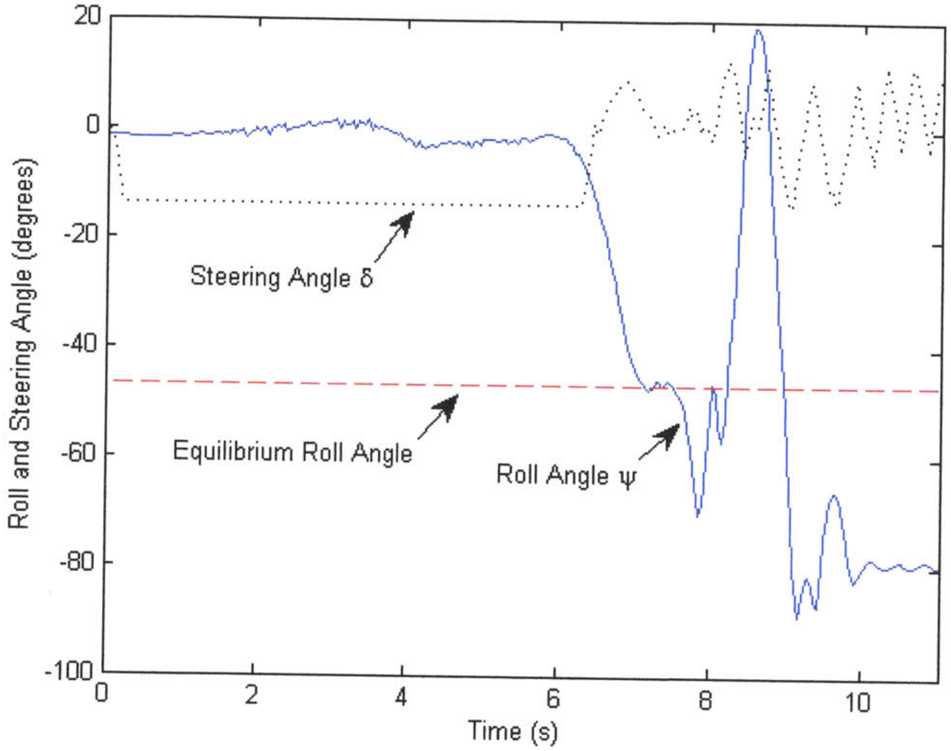

Figure 13: Vehicle steering and roll angles during a tip-up attempt. The vehicle rolls to one side and stabilizes briefly about the balancing angle before rolling back down on to four wheels.

Initially, the vehicle is given a desired acceleration command sufficient to initiate tip-over. As a result, the vehicle begins driving in a tight clockwise circle. The throttle command is then gradually increased by the operator until the vehicle began to tip up. At t=6s, the vehicle's lateral acceleration becomes sufficiently large and it begins to tip up. The roll angle rises smoothly to the balance angle of 47°, and the vehicle pauses there momentarily. However, the vehicle then begins to roll further, prompting a countersteer that brings the vehicle back down on four wheels. The ensuing bounce of the vehicle off the ground causes it to roll over onto its side. The vehicle's roll energy during this maneuver can be seen in Figure 14:

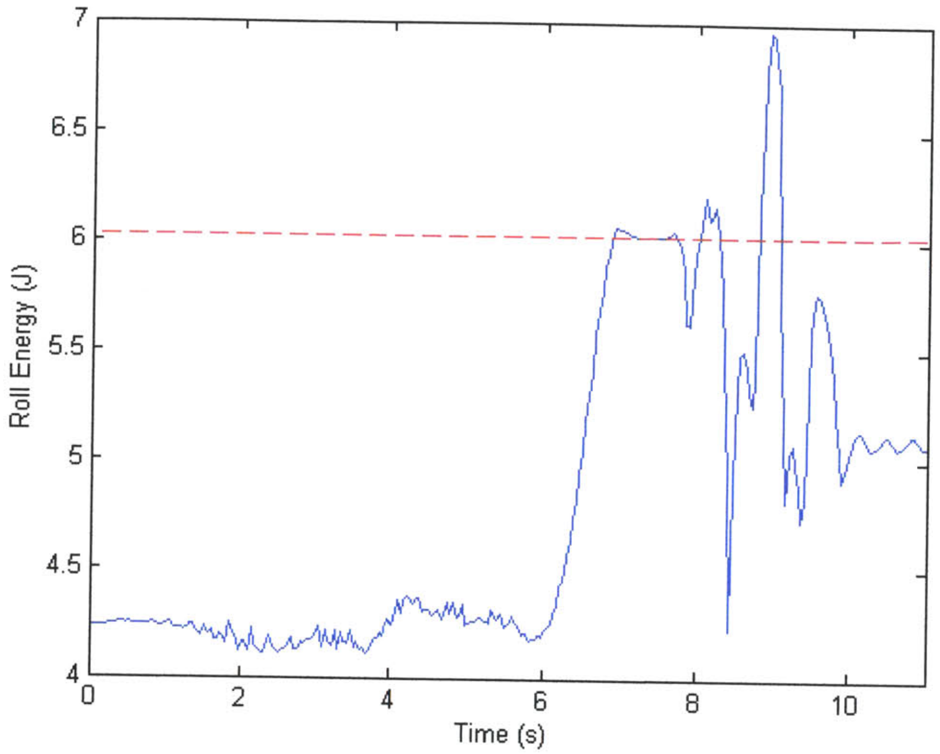

Figure 14: Vehicle roll energy during the trial shown in Figure 13. The dashed line represents the equilibrium energy value when the vehicle is balanced.

Initially, the energy fluctuates as the vehicle drives in a tight circle. Then, as the vehicle begins to tip up the energy rises rapidly until it is stabilized at the equilibrium energy value by the controller. The controller maintains this value for about a second before the vehicle begins to roll further just before t=8s. The controller then over-compensates, causing the energy level to overshoot the desired value before dropping sharply as the vehicle returns to four wheels.

A plot of roll rate against roll angle reveals additional information, as shown in Figure 15.

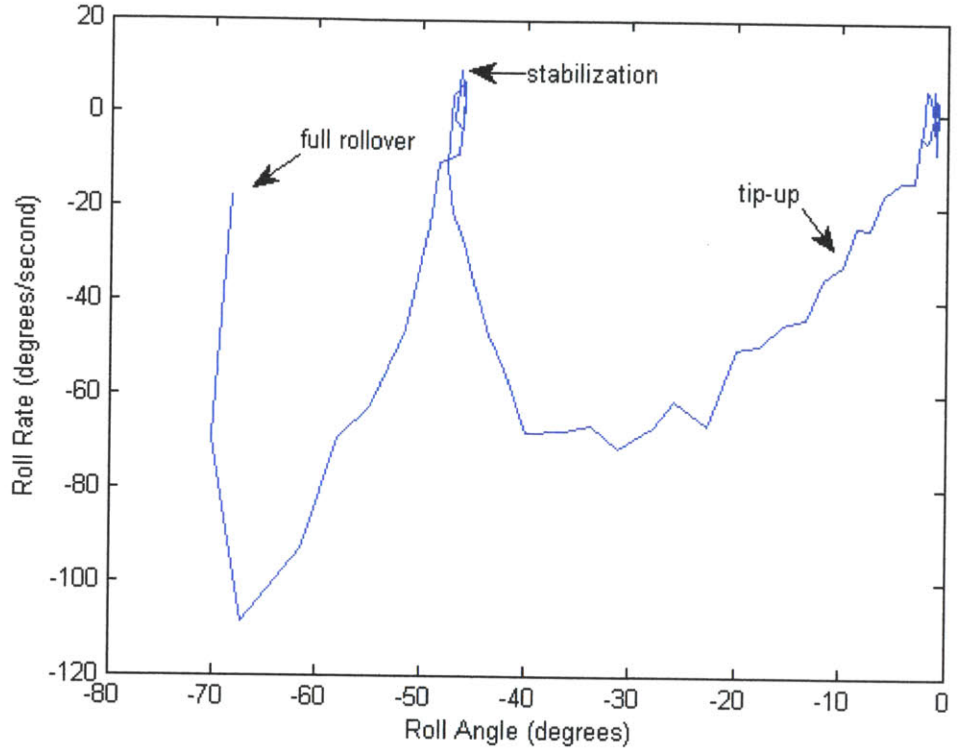

Figure 15: Roll angle and roll rate during the tip-up attempt shown in Figure 13. For clarity, this figure only spans the time from t=6 to t=8.

The vehicle begins in at the right hand side of the figure. As tip-up begins, the roll and roll rate begin to increase. When the roll angle reaches approximately -30°, the roll rate levels off, then drops, stabilizing the vehicle at the equilibrium roll angle of -47°. When control diverges, the roll angle again increases until the vehicle rolls over entirely.

While testing the vehicle it was noted that the controller's behavior seemed to be influenced by the state of the battery. Switching to a freshly charged battery after the run shown in Figures 13 and 14 reduced the controller's effectiveness, as seen in Figure 16.

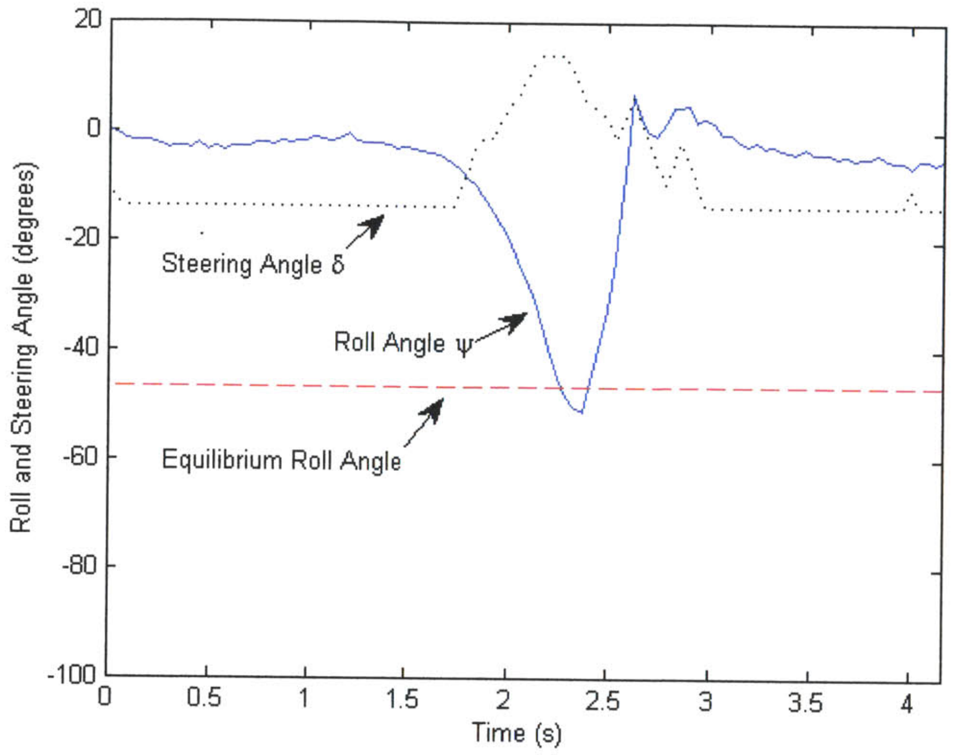

Figure 16: Vehicle steering and roll angles during another tip-up attempt. The vehicle tips up briefly before rolling back down on to four wheels.

In this trial, the vehicle once again begins to tip up at t=1.5s. The roll angle increases smoothly up to and past the stabilizing roll angle of 47°, but a strong countersteer forces the vehicle back down on to four wheels. The vehicle roll energy during this trial can be seen in Figure 17.

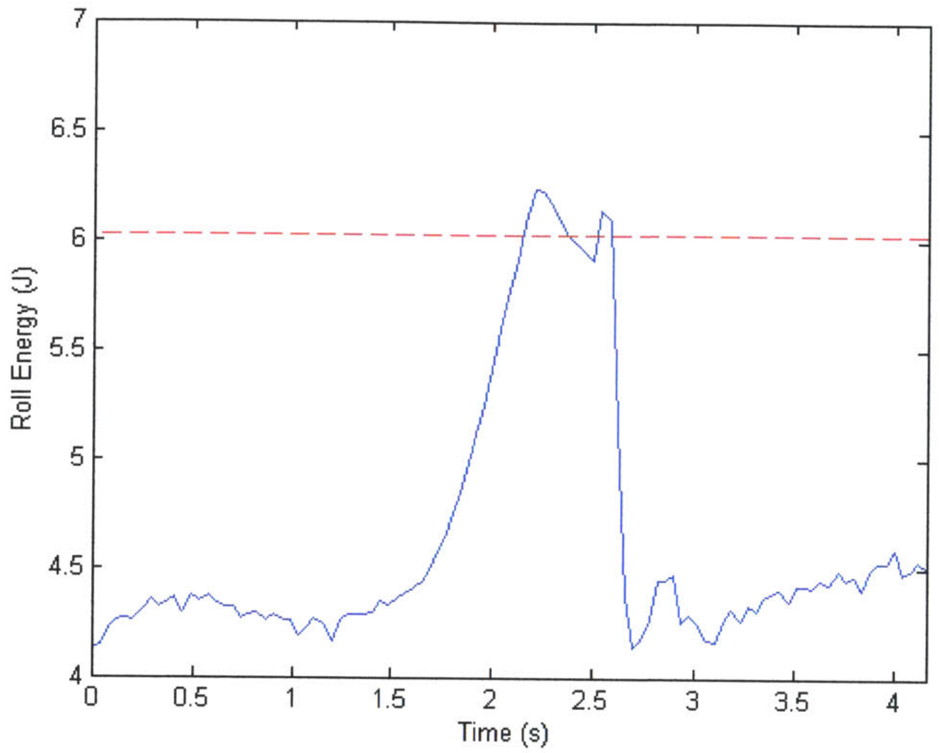

Figure 17: Vehicle roll energy during the trial shown in Figure 16. The dashed line represents the equilibrium energy value when the vehicle is balanced.

In this case, the controller does not stabilize at the desired energy value, but instead overshoots, oscillates, and then drops back down to baseline levels.

Figure 18 shows the roll rate plotted against roll angle for this trial.

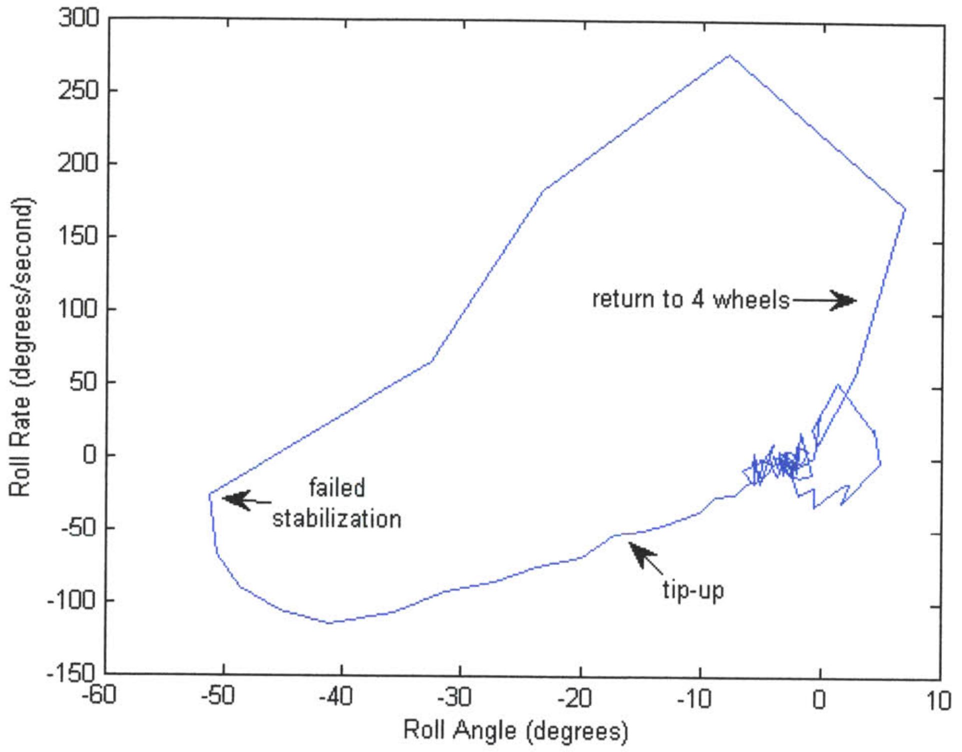

Figure 18: Roll angle and roll rate during the tip-up attempt shown in Figure 16.

In this trial, the roll angle and roll rate once again increase in magnitude smoothly as the vehicle begins to tip up. However, unlike in Figure 15, the roll rate does not level off, but instead only begins decreasing when the roll angle nears -40° degrees. As a result, the vehicle fails to stabilize at the equilibrium angle and returns to 4 wheels.

II. Discussion

In the example shown in Figures 13 and 14, energy and roll angle stabilization were achieved for a period of about 1 second before instabilities caused the vehicle to fall back on to four wheels. As the energy shaping controller is designed to drive the vehicle's roll energy to a fixed value, but not to stabilize about the equilibrium point, it was expected that the vehicle would not be able to remain balanced in the tipped-up position.

Once possible explanation for the controller performance is the presence of outside disturbances. Owing to the nature of the controller, the vehicle would drive in a circle until tip-up began, at which point it would drive in an approximately straight line as it attempted to drive the roll energy to the desired value. Due to this algorithm, the vehicle had to be tested outdoors in an open area. Attempts to test indoors failed as the floor surface did not supply sufficient friction to cause tip-up. Although the driving surface outdoors was relatively flat, fluctuations in the surface roughness may have transmitted disturbances to the vehicle. Additionally, the vehicle used tires with a relatively large tread pattern, which may have resulted in periodic disturbances to the vehicle proportional to the vehicle's speed.

Another explanation is that the controller was insufficiently robust, owing in part to the limited number of sensors aboard the vehicle. Inspection of the Figures 13 and 16 reveals large steering angles prior to the divergence of the vehicle from the desired roll angle. Possibly, the gains for the countersteer term or steering controller are too large, resulting in the rapid steering that causes the vehicle to roll back down on to four wheels. However, experiments with lower values for these gains resulted in the vehicle rolling over entirely. In particular, the countersteering term in (19) is necessary to slow the tip-up process so that the controller may react before rollover. The roll-rate/roll-angle plots in Figures 15 and 18 in particular suggest that the controller may need to react sooner in order to reduce the roll rate to allow stabilization at the equilibrium angle.

The lack of sensors aboard the robot may also contribute to the instability of the controller. From the perspective of the microcontroller, the RC hobby servo used for the steering actuator is an open-loop controller, as a steering pulse is commanded with no indication of the actual steering angle. Currently, the microcontroller cannot sense if the commanded steering angle is being accurately reproduced by the steering actuator. The addition of a potentiometer to sense the actual steering angle would allow for improved estimation of the vehicle's behavior.

Another place in which sensing could be improved is the estimation of the vehicle's forward velocity, as evidenced by the change in behavior of the controller when the vehicle's battery was changed. The vehicle's forward velocity is currently estimated based on the throttle pulse sent to the vehicle's motor controller. However, the correlation used was derived empirically from a limited number of trials, and did not take into account changing battery voltages. In particular, although the vehicle's motor controller is labeled as an "Electronic Speed Controller", the device actually operates by outputting a square wave with a duty cycle dependent on the command pulse. As a result, the motor controller produces an output voltage that is proportional to the supplied battery voltage and does not actually control speed in a closed-loop manner. As the magnitude of lateral steering forces generated depends on the magnitude of the vehicle's forward velocity, an inconstant forward velocity can cause the control algorithm to fail. One possible remedy is the addition of a tachometer to measure the rotational speed of the wheels; due to the vehicle's construction, mounting a Hall-effect or optical tachometer to the vehicle's longitudinal driveshaft may be easiest. This simple change would allow for a significantly more accurate measurement of the vehicle's forward velocity, improving the effectiveness of the controller. Additionally, the current motor controller and steering may be replaced with one that does not interface with the FEZ Domino via the RC hobby PWM standard, allowing for more responsive control.

Conclusions

An instrumented 4-wheeled robot was built using the chassis of a 1/12-scale radio control vehicle. The robot included a 3DM-GX2 inertial measurement unit and a FEZ Domino microcontroller, which read PWM signals sent from a radio transmitter, logged sensor data, and implemented control laws by outputting commands to the vehicle's motor and steering actuator. A steering control law and energy shaping controller for tip-up stabilization were developed and implemented on the experimental hardware. Initial experimental results demonstrate that the controller is able to drive the roll energy and roll angle to stabilizing values and maintain them for up to one second. Analysis of data collected during tip-up attempts suggests that instability in the controller may be caused by a need for additional sensing aboard the vehicle. Further work could include the addition of steering angle and forward velocity sensors, as well as an extension of the controller algorithm to allow balancing once the vehicle has stabilized in the tipped-up position.

References

[1] Arndt, David; Bobrow, James E.; Peters, Steven; Iagnemma, Karl; and Dubowsky, Steven (2009). "Two-Wheel Self-Balancing of a Four-Wheeled Vehicle [Applications of Control]". *IEEE Control Systems Magazine*, vol. 31, no. 2, pp.29-37, April 2011.

[2] Peters, Steven C.; Bobrow, James E.; and Iagnemma, Karl (2009). "Stabilizing a Vehicle near Rollover: An Analogy to Cart-Pole Stabilization". IEEE International Conference on Robotics and Automation, 2010.

[3] http://www.microstrain.com/3dm-gx2.aspx

[4] http://www.ghielectronics.com/catalog/product/133

[5] Wong, J. Y. *Theory of Ground Vehicles*. New York : John Wiley, 2001.

Appendix A: Source code
What follows below is the C# source code for the program used to implement the tip-up and stabilization routine on the vehicle.

```
using System;
using System.IO; // allows directories
using System.Threading;
using System.Text;

using Microsoft.SPOT;
using Microsoft.SPOT.IO;
using Microsoft.SPOT.Hardware;

using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.System;
using GHIElectronics.NETMF.USBHost;

namespace USBTest
{

    class Program
    {

        static USBH_SerialUSB serialUSB;
        static Thread readUSBThread;

        static PersistentStorage sdcard;
        static bool devConnected; // device connected?
        static bool writing; // not currently recording data
        static bool doneLogging; // finished log?
        static DateTime lastpress;

        static OutputPort LED;


        // drive control variables
        // steering: Di10
        // Throttle: Di9

        // steering in (ch2): Di1
        // throttle in (ch1): Di0
        static PWM servo;
        static PWM throttle;
        static uint servstate;
        static uint throtstate;
        static DateTime servpulsup;
        static DateTime throtpulsup;
        static uint servopulse = 1482;
        static uint throttlepulse = 1500;
        static uint servoread; // read-in servo pulse from receiver
        static uint throttleread; // read-in throttle pulse from receiver
        static uint twentyms = 20 * 1000 * 1000;

        static float pi = (float) System.Math.PI;

        public static void Main()
        {
            // Subscribe to USBH events.
```

```
USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;
USBHostController.DeviceDisconnectedEvent += DeviceDisconnectedEvent;
devConnected = false;



// set up LED
LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, false);

// set up LDR button
InputPort LDR = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, true,
                                  Port.ResistorMode.PullUp);


// filter adjust
// need this or else the filter will filter out all the servo pulses!
Microsoft.SPOT.Hardware.Cpu.GlitchFilterTime = new
                                  TimeSpan(TimeSpan.TicksPerMillisecond);

// set up servo and throttle
servo = new PWM((PWM.Pin)FEZ_Pin.PWM.Di10); // steering servo
throttle = new PWM((PWM.Pin)FEZ_Pin.PWM.Di9); // throttle


// set up reading for steering
InterruptPort servin = new InterruptPort((Cpu.Pin)FEZ_Pin.Interrupt.Di1,
                                  true, Port.ResistorMode.PullUp,
                                  Port.InterruptMode.InterruptEdgeBoth);
servin.OnInterrupt += new NativeEventHandler(servo_pulse_get);


// set up reading for throttle
InterruptPort throtin = new InterruptPort((Cpu.Pin)FEZ_Pin.Interrupt.Di0,
                                  true, Port.ResistorMode.PullUp,
                                  Port.InterruptMode.InterruptEdgeBoth);
throtin.OnInterrupt += new NativeEventHandler(throttle_pulse_get);

servopulse = 1500;
throttlepulse = 1500;


// Sleep forever
//Thread.Sleep(Timeout.Infinite);

writing = false;
doneLogging = false;

while (!devConnected)
{
    Debug.Print("waiting for device...");
    Thread.Sleep(1000);
}

// neutral pulse
servo.SetPulse(twentyms, 1500 * 1000);
throttle.SetPulse(twentyms, 1500 * 1000);

// wait for button press
while (LDR.Read() == true)
{
    Thread.Sleep(20);
}
```

```
    LED.Write(true);
    Thread.Sleep(1000);
    writing = true;
    Debug.Print("beginning write!");

    // writing
    while (LDR.Read() == true)
    {
        //Debug.Print("pretending to write data");
        Thread.Sleep(20);
    }
    writing = false;
    doneLogging = true;

    // neutral pulse
    servo.SetPulse(twentyms, 1500 * 1000);
    throttle.SetPulse(twentyms, 1500 * 1000);

    Debug.Print("done!");

    Thread.Sleep(3000);
    LED.Write(false);
    readUSBThread.Suspend();
}

static void DeviceConnectedEvent(USBH_Device device)
{
    Debug.Print("Device connected...");
    Debug.Print("ID: " + device.ID + ", Interface: " + device.INTERFACE_INDEX
                                   + ", Type: " + device.TYPE);


    USBH_Device siLabs = new USBH_Device(device.ID, device.INTERFACE_INDEX,
                                USBH_DeviceType.Serial_SiLabs,
                                device.VENDOR_ID, device.PRODUCT_ID,
                                device.PORT_NUMBER);
    serialUSB = new USBH_SerialUSB(siLabs, 115200, System.IO.Ports.Parity.None,
                                8, System.IO.Ports.StopBits.One);
    serialUSB.Open();
    readUSBThread = new Thread(SerialUSBReadThread);
    readUSBThread.Start();

    devConnected = true;

}

static void DeviceDisconnectedEvent(USBH_Device device)
{
    Debug.Print("Device disconnected...");
    Debug.Print("ID: " + device.ID + ", Interface: " +
    device.INTERFACE_INDEX + ", Type: " + device.TYPE);


    //readUSBThread.Suspend();
    devConnected = false;
}




static void SerialUSBReadThread()
```

```
{
    byte[] accbuf = new byte[31]; // buffer for reading accelerations
    byte[] eulerbuf = new byte[19]; // buffer for reading euler angles
    byte[] stabbuf = new byte[43]; // buffer for reading stabilized acc.
    byte[] cmd_getAcc = { 0xC2 }; // command to send to get accelerations
    byte[] cmd_getEuler = { 0xCE }; // command to send to get euler angles
    byte[] cmd_getStab = { 0xD2 }; // command to get gravity vector
    byte[] timebuf = new byte[4]; // buffer used to convert time for logging
    byte[] cmdbuf = new byte[4]; // buffer used to convert steering and
                                        // throttle commands for logging
    byte[] convbuf = { 0, 0, 0, 0 }; // buffer for conversions

    float accelX=0, accelY=0, accelZ=0;
    float roll=0, pitch=0, yaw=0;
    float roll_offset = 0, pitch_offset = 0, yaw_offset = 0;
    float rollrate=0, pitchrate=0, yawrate=0;
    float stabAccX=0, stabAccY=0, stabAccZ=0;

    float compAccX=0, compAccY=0, compAccZ=0; // subtracted out gravity

    // control law variables
    float theta_true; // true tilt angle of the vehicle's COM
    float theta_err;

    float PHI_INT = 43*pi/180.0f; // "angle" of internal pendulum of the
                                        //vehicle's COM
    float L_INT = 0.19825f; // internal "radius" of vehicle's COM
    float L_WB = 0.24f; // length of wheelbase between axles
    float ROLL_MARGIN = 5*pi/180.0f; // angle of tilt before tip-up routine
                                                // kicks in
    float ARR_MARGIN = 15 * pi / 180.0f; // angle of tilt when countersteer
                                                // kicks in

    float energy = 0; // energy associated with the roll axis
    float energy_err; // difference between actual and desired energy
    float energy_prev; // roll energy on the previous iteration
    float e_dot_pred = 0; // predicted rate of change of energy
    float MGL = 3.1f * 9.8f * L_INT; // desired energy = m * g * l
    float ML2 = 3.1f * L_INT * L_INT; // ML^2 = m * l * l

    float smoothedLat = 0; // smoothed lateral force
    float prevlat = 0; // previous value of lateral force, used to calculate
                            // rate of change
    float smoothRatio = 0.5f; // smoothing average for lateral force
                            // calculation
    float ades_prev = 0;
    float ades = 0; // desired lateral acceleration
    float ades_dot = 0; // change in desired lateral acceleration
    float smoothed_yawrate = 0; // smoothed yaw rate
    float smoothed_rollrate = 0;


    float v_est = 0; // estimated forward velocity
    float v_est_prev = 0; // previous estimated forward velocity

    float delta = 0; // steering angle in radians
    float DELTA_MAX = 18 * pi/180; // maximum steering angle
    float DELTA_DOT_MAX = 5 * pi / 180; // maximum rate of change of delta
                                    // (estimated from servo datasheet)

    float delta_dot_des=0; // desired rate of change of steering angle
    float delta_naive = 0; // desired steering angle based on no-slip model
    float Kd = 2f;// gain for anti-rollover
float simAc = 0; // simulated lateral force

float smoothedAc = 0; // smoothed true centripital acceleration
float smoothedAc_prev = 0; // previous smoothed Ac

float mergeAc_prev; // previous mergeAc;
float mergeAc = 0; // better estimate for lateral force, by merging simAc
                            // with smoothedAc
float mergeAc_dot = 0; // change in mergeAc


float K1 = 5; // energy gain for tip-up

float atip = 11.5f; // acceleration to tip-up = 1.1*9.8/tan(phi)


float ctrl_mode = 0; // which mode is the control in

bool logging;
// create SD card to write to
if (PersistentStorage.DetectSDCard())
{
    sdcard = new PersistentStorage("SD");
    sdcard.MountFileSystem();
    Debug.Print("Mounted SD card");
    logging = true;
}
else
{
    Debug.Print("ERROR: No SD card detected. Skipping log.");
    logging = false;
}

// create log file
FileStream fs;
if (logging)
{
    string rootdir = VolumeInfo.GetVolumes()[0].RootDirectory;
    string[] files = Directory.GetFiles(rootdir);


    // assume there are only IMU_log files
    string logname = @"\SD\IMU_log" + (files.Length+1).ToString() +
                                                                ".dat";

    Debug.Print("New file shall be: " + logname);
    fs = new FileStream(logname, FileMode.Create);

    Debug.Print("created file");

}
else
{
    //logfile = null;
    fs = null; // avoids complaints
}


Thread.Sleep(2000);

// set euler angle offsets
if (getData(cmd_getEuler, eulerbuf, 19) == 0)
```

```
{
 // successfully read euler angles

    // pitch
    convbuf[0] = eulerbuf[4];
    convbuf[1] = eulerbuf[3];
    convbuf[2] = eulerbuf[2];
    convbuf[3] = eulerbuf[1];
    GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                            pitch_offset, convbuf, 0);

    // roll
    convbuf[0] = eulerbuf[8];
    convbuf[1] = eulerbuf[7];
    convbuf[2] = eulerbuf[6];
    convbuf[3] = eulerbuf[5];
    GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out  roll_offset,
                                            convbuf, 0);

    // yaw
    convbuf[0] = eulerbuf[12];
    convbuf[1] = eulerbuf[11];
    convbuf[2] = eulerbuf[10];
    convbuf[3] = eulerbuf[9];
    GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out yaw_offset,
                                            convbuf, 0);
}



DateTime logstart = DateTime.Now; // begin timing

TimeSpan diff;
float currsecs;


//byte[] testingbuf = new byte[4*9+10];

while (true)
{
    //Thread.Sleep(10); // sampling rate = 1000/sleep

    if (writing)
    {


        if (getData(cmd_getEuler, eulerbuf, 19) == 0)
        {
              // successfully read euler angles
            // pitch
            convbuf[0] = eulerbuf[4];
            convbuf[1] = eulerbuf[3];
            convbuf[2] = eulerbuf[2];
            convbuf[3] = eulerbuf[1];
            GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                                 pitch, convbuf, 0);
            pitch = pitch - pitch_offset;

            // roll
            convbuf[0] = eulerbuf[8];
            convbuf[1] = eulerbuf[7];
            convbuf[2] = eulerbuf[6];
        convbuf[3] = eulerbuf[5];
        GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                                  roll, convbuf, 0);
        roll = roll - roll_offset;

        // yaw
        convbuf[0] = eulerbuf[12];
        convbuf[1] = eulerbuf[11];
        convbuf[2] = eulerbuf[10];
        convbuf[3] = eulerbuf[9];
        GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out  yaw,
                                                  convbuf, 0);
        yaw = yaw - yaw_offset;
    }


    if (getData(cmd_getAcc, accbuf, 31) == 0)
    {
            // successfully read accelerations
        // accelX
        convbuf[0] = accbuf[4];
        convbuf[1] = accbuf[3];
        convbuf[2] = accbuf[2];
        convbuf[3] = accbuf[1];
        GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                              accelX, convbuf, 0);

        // accelY
        convbuf[0] = accbuf[8];
        convbuf[1] = accbuf[7];
        convbuf[2] = accbuf[6];
        convbuf[3] = accbuf[5];
        GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                              accelY, convbuf, 0);

        // accelZ
        convbuf[0] = accbuf[12];
        convbuf[1] = accbuf[11];
        convbuf[2] = accbuf[10];
        convbuf[3] = accbuf[9];
        GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                              accelZ, convbuf, 0);


        // pitchrate
        convbuf[0] = accbuf[16];
        convbuf[1] = accbuf[15];
        convbuf[2] = accbuf[14];
        convbuf[3] = accbuf[13];
        GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                              pitchrate, convbuf, 0);

        // rollrate
        convbuf[0] = accbuf[20];
        convbuf[1] = accbuf[19];
        convbuf[2] = accbuf[18];
        convbuf[3] = accbuf[17];
        GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                              rollrate, convbuf, 0);

        // yawrate
        convbuf[0] = accbuf[24];
        convbuf[1] = accbuf[23];
    convbuf[2] = accbuf[22];
    convbuf[3] = accbuf[21];
    GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                    yawrate, convbuf, 0);
    yawrate = -yawrate; // sign needs to be flipped

}


if (getData(cmd_getStab, stabbuf, 43) == 0)
{
       // successfully got stabilized accelerations?
    // stabAccX
    convbuf[0] = stabbuf[4];
    convbuf[1] = stabbuf[3];
    convbuf[2] = stabbuf[2];
    convbuf[3] = stabbuf[1];
    GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                    stabAccX, convbuf, 0);

    // stabAccY
    convbuf[0] = stabbuf[8];
    convbuf[1] = stabbuf[7];
    convbuf[2] = stabbuf[6];
    convbuf[3] = stabbuf[5];
    GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                    stabAccY, convbuf, 0);

    // stabAccZ
    convbuf[0] = stabbuf[12];
    convbuf[1] = stabbuf[11];
    convbuf[2] = stabbuf[10];
    convbuf[3] = stabbuf[9];
    GHIElectronics.NETMF.System.Util.ExtractValueFromArray(out
                                    stabAccZ, convbuf, 0);
}


diff = DateTime.Now - logstart;
currsecs = ((float)diff.Ticks) / TimeSpan.TicksPerSecond;


compAccX = accelX - stabAccX;
compAccY = accelY - stabAccY;
compAccZ = accelZ - stabAccZ;



// ========= CONTROL LAW GOES HERE =======================

prevlat = smoothedLat; // update
smoothedLat = accelX * smoothRatio + smoothedLat *
       (1 - smoothRatio); // calculate smoothed lateral force

smoothedAc_prev = smoothedAc; // update
smoothedAc = 0.8f * smoothedAc + 0.2f * 9.8f *
       (compAccX * cos(roll) - compAccZ * sin(roll));
              // calculate smoothed centripital acceleration

// smooth yaw rate
smoothed_yawrate = 0.8f * smoothed_yawrate + 0.2f * yawrate;

// smooth roll rate
```

```
smoothed_rollrate = 0.6f * smoothed_rollrate + 0.4f * rollrate;

// delta from servo pulses
delta = servopulse*(DELTA_MAX / 140f) - 1445*(DELTA_MAX / 140f);

throttlepulse = throttleread; // echo throttle input
// scale throttle according to tilt
throttlepulse = (uint) (1540 + 0.5f*(1+cos(roll))*
                                    (throttleread - 1540f));


// calculate theta_true ( true roll angle)
if (roll > 0) theta_true = roll + PHI_INT;
else theta_true = pi + roll - PHI_INT;

// roll error
theta_err = theta_true - pi / 2;

// estimate forward velocity
v_est_prev = v_est; // update
v_est = v_est + 0.08f * (1.5f * ((float)throttlepulse - 1540f) /
      170f - v_est);  // = prev_v + 0.1 * (est. maxv - prev_v)

// get an estimate of the lateral force
simAc = v_est * v_est * sin(delta) / 0.24f; //v^2*sin(delta)/l
mergeAc_prev = mergeAc; // update
mergeAc = 0.3f * simAc + 0.7f * smoothedAc; // merge with
                                            // measurement
mergeAc_dot = mergeAc - mergeAc_prev;

// calculate roll energy
energy_prev = energy; // update
energy = MGL * sin(theta_true) + 0.5f * ML2 * smoothed_rollrate *
                    smoothed_rollrate; // with smoothed rollrate
energy_err = energy - MGL;
// predict the rate of change of energy
e_dot_pred = energy - energy_prev; // discretized

ades_prev = ades;
//ades = -4; // constant
//ades = -6 * sin(2*pi*currsecs); // sinusoidal 1/2 second period
//ades = -2 * (int)(currsecs%2.0); // stepwise 2 second period



// energy controller here
if (roll > ROLL_MARGIN || roll < -ROLL_MARGIN) // tipping up
{
    // calculate desired energy using smoothed roll rate
    ades = -K1 * smoothed_rollrate * sin(theta_true) * energy_err;

    // anti-rollrate term
    if (roll > ARR_MARGIN || roll < -ARR_MARGIN)
    {
        ades += -Kd*cos(roll) * smoothed_rollrate;
    }

    ctrl_mode = 1;
}
else // not yet tipped up; circling
{
    ades = -atip; // circle to the right
    ctrl_mode = 0;
```

```
}

ades_dot = ades - ades_prev;


// steering controller
if (v_est != 0)
{
    delta_dot_des = 0.1f * (ades - mergeAc);

    // delta naive prediction
    delta_naive = L_WB * ades / (v_est * v_est);
    delta_naive = clamp(delta_naive, -DELTA_MAX, DELTA_MAX);

    // average delta_dot_des and naive delta
    delta_dot_des = (delta_dot_des + delta_naive - delta) / 2.0f;

    // clamp steering rate
    delta_dot_des = clamp(delta_dot_des, -DELTA_DOT_MAX,
                                                DELTA_DOT_MAX);

    delta += delta_dot_des;
}
else
{
    // set to zero
    delta = 0;
}




// clamp steering angle
delta = clamp(delta, -DELTA_MAX, DELTA_MAX);

// convert back to pulses
servopulse = (uint)(1445 + 140 * delta / DELTA_MAX);


// clamp throttle
if (throttlepulse > 1960) throttlepulse = 1960; // clamp
else if (throttlepulse < 1000) throttlepulse = 1000; // clamp



// clamp steering output
if (servopulse > 1665) servopulse = 1665;
else if (servopulse < 1225) servopulse = 1225;

// echo control pulses
servo.SetPulse(twentyms, servopulse * 1000);
throttle.SetPulse(twentyms, throttlepulse * 1000);


// =========== END CONTROL LAW ===========================




if (logging)
{
```

```
                        GHIElectronics.NETMF.System.Util.InsertValueIntoArray(currsecs,
                                                                     convbuf, 0);
                        // flip endian for time
                        timebuf[0] = convbuf[3];
                        timebuf[1] = convbuf[2];
                        timebuf[2] = convbuf[1];
                        timebuf[3] = convbuf[0];
                        fs.Write(timebuf, 0, 4); // write the time

                        // convert steering command

GHIElectronics.NETMF.System.Util.InsertValueIntoArray(servopulse, convbuf, 0);
                        // flip endian
                        cmdbuf[0] = convbuf[3];
                        cmdbuf[1] = convbuf[2];
                        cmdbuf[2] = convbuf[1];
                        cmdbuf[3] = convbuf[0];
                        fs.Write(cmdbuf, 0, 4); // write the steering command

                        // convert throttle command

GHIElectronics.NETMF.System.Util.InsertValueIntoArray(throttlepulse, convbuf, 0);
                        // flip endian
                        cmdbuf[0] = convbuf[3];
                        cmdbuf[1] = convbuf[2];
                        cmdbuf[2] = convbuf[1];
                        cmdbuf[3] = convbuf[0];
                        fs.Write(cmdbuf, 0, 4); // write the throttle command

                        fs.Write(eulerbuf, 1, 12); // write euler angles
                        fs.Write(accbuf, 1, 24); // write accelerations and angles
                        fs.Write(stabbuf, 1, 12); // write stabilized accelerations
                        fs.Flush();

                        // write ades
                        GHIElectronics.NETMF.System.Util.InsertValueIntoArray(ades,
                                                                 convbuf, 0);
                        // flip endian
                        cmdbuf[0] = convbuf[3];
                        cmdbuf[1] = convbuf[2];
                        cmdbuf[2] = convbuf[1];
                        cmdbuf[3] = convbuf[0];
                        fs.Write(cmdbuf, 0, 4); // write the ades command

                        // write ctrl_mode

GHIElectronics.NETMF.System.Util.InsertValueIntoArray(ctrl_mode, convbuf, 0);
                        // flip endian
                        cmdbuf[0] = convbuf[3];
                        cmdbuf[1] = convbuf[2];
                        cmdbuf[2] = convbuf[1];
                        cmdbuf[3] = convbuf[0];
                        fs.Write(cmdbuf, 0, 4); // write the ades command


                    }
                    else
                    {

                    }

                }
                else
                { // not writing

                    // Debug.Print("waiting for button");
                     if (doneLogging)
                     {
                         Debug.Print("done logging!");
                         if (logging)
                         {
                             sdcard.Dispose();
                             //logfile.Close();

                             fs.Close();
                         }
                         break;
                     }
                }


            } // end of while loop

        }





        // writes cmd to USB, then reads bytesToRead bytes into databuf. Returns -1 on
write failure, -2 on read failure
        static int getData(byte[] cmd, byte[] readbuf, int bytesToRead)
        {
            int retval = 0;
            try
            {
                retval = serialUSB.Write(cmd, 0, cmd.Length);
            }
            catch (System.Exception) // occurs after disconnect
            {
                retval = 0;
            }

            if (retval == cmd.Length) // successful write
            {
                retval = serialUSB.Read(readbuf, 0, bytesToRead);
                if (retval == bytesToRead) // successful read
                {
                    return 0;
                }
                else // read failure
                {
                    return -2;
                }
            }
            else // write failure
            {
                return -1;
            }
        }
        // shorthand for sin
        static float sin(float f)
        {
            return ((float)GHIElectronics.NETMF.System.MathEx.Sin(f));
        }

        // shorthand for cosine
        static float cos(float f)
        {
            return ((float)GHIElectronics.NETMF.System.MathEx.Cos(f));
        }

        // absolute value function
        static double abs(double d)
        {
            return (d > 0) ? d : -d;
        }

        // clamping
        static float clamp(float f, float min, float max)
        {
            if (f < min) return min;
            else if (f > max) return max;
            else return f;
        }


        // read servo pulse
        static void servo_pulse_get(uint port, uint state, DateTime time)
        {

            if (state == 1 && servstate == 0) // rising edge
            {
                servpulsup = time; // start timing
            }
            else if (state == 0 && servstate == 1) // falling edge
            {
                long temp = ((time.Ticks - servpulsup.Ticks) * 1000 /
TimeSpan.TicksPerMillisecond); // end timing, measure pulsewidth
                // filter  1000 < pulsewidth < 2000
                if (temp > 1000 && temp < 2000)
                {
                    //Debug.Print("servo pulse of " + servopulse + " us");
                    servoread = (uint)temp;
                }
            }
            servstate = state;

        }

        // read throttle pulse
        static void throttle_pulse_get(uint port, uint state, DateTime time)
        {

            if (state == 1 && throtstate == 0) // rising edge
            {
                throtpulsup = time; // start timing
            }
            else if (state == 0 && throtstate == 1) // falling edge
            {
                long temp = ((time.Ticks - throtpulsup.Ticks) * 1000 /
TimeSpan.TicksPerMillisecond); // end timing, measure pulsewidth
                // filter  1000 < pulsewidth < 2000
                if (temp > 1000 && temp < 2000)
                {
                    //Debug.Print("throttle pulse of " + throttlepulse + " us");
                    throttleread = (uint)temp;
                }
            }
            throtstate = state;

        }


        // SD insert/remove
        static void RemovableMedia_Insert(object sender, MediaEventArgs e)
        {
            Debug.Print("Storage \"" + e.Volume.RootDirectory + "\" is inserted.");
            Debug.Print("Getting files and folders:");

            sdcard = new PersistentStorage("SD");
            sdcard.MountFileSystem();

            if (VolumeInfo.GetVolumes()[0].IsFormatted)
            {
                string rootDirectory = VolumeInfo.GetVolumes()[0].RootDirectory;
                Debug.Print("Root directory: " + rootDirectory);

                string[] files = Directory.GetFiles(rootDirectory);
                string[] folders = Directory.GetDirectories(rootDirectory);

                Debug.Print("Files available on " + rootDirectory + ":");
                for (int i = 0; i < files.Length; i++)
                    Debug.Print(files[i]);

                Debug.Print("Folders available on " + rootDirectory + ":");
                for (int i = 0; i < folders.Length; i++)
                    Debug.Print(folders[i]);
            }
            else
            {
                Debug.Print("Storage is not formatted. Format on PC with FAT32/FAT16
first.");
            }

            sdcard.Dispose();
        }

        static void RemovableMedia_Eject(object sender, MediaEventArgs e)
        {
            Debug.Print("Storage \"" + e.Volume.RootDirectory + "\" is ejected.");
        }
    }
}
```