Development of a Stable Control System for a Segway

Dennis Jin
`adennisjin@gmail.com`

under the direction of
Mr. Patricio Esteban Valenzuela Pacheco
Mr. Niclas Blomberg
Mr. Niklas Everitt
Control System Engineering Department
Royal Institute of Technology

Research Academy for Young Scientists
July 10, 2013

Abstract

In this research project, a model used to stabilize and control a segway is developed and analyzed with the help of a Lego Mindstorms NXT 2.0 kit and the RobotC IDE. PID-controller gains were determined through Zielger-Nichols method along with some manual tuning, which lead to a stable closed-loop control system that could balance a segway. Finally, an improved method of tuning is suggested.

Contents

1 Introduction

Robotics is currently a rapidly growing branch of technology due to the high demand for seamless and automatic systems [1]. During recent years, robotics have also made a breakthrough in personal transportation with the segway [2]. It is a flexible vehicle that balances on two wheels and can be controlled through the motion of tilting in the direction one wishes to travel. Conceptually, it can be considered a modern adaption of the *inverted pendulum,* which is an upside-down pendulum, with its center of mass located above its pivot point. The goal of the classical inverted pendulum problem is to keep the pendulum stable by applying a correct amount of force or torque to its base, canceling out any downwards acceleration that the gravitational force has on the pendulum. However, unlike the classical inverted pendulum, the segway, with the help of an *embedded control system* and a movable base, is also able to freely drive and turn in in all directions and maintains balance at all speeds. This is achieved with the help of responsive sensors and advanced control algorithms. Because the segway and its technology is relatively new (and expensive), logically, there is still lots of room for improvement.

This research project aims to develop, analyze and improve upon a model used to stabilize and control a segway, with the help of an already available set of development tools, the Lego Mindstorms NXT 2.0 kit and RobotC.

1.1 Theory

A proportional-integral-derivative-controller (PID-controller) is a widely used controller in control systems [3]. Its goal is to minimize the error in the system (denoted by $e(t)$), relative to a certain reference point that is determined manually. In this case, in order for the segway to balance, the ideal reference point is a point, where the angular deviation (denoted by θ), angular velocity (denoted by $\dot{\theta}$), displacement (denoted by y) and velocity (denoted by $\dot{y}$), is zero. The PID-controller can be described as following:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau) d\tau + k_d \frac{de(t)}{dt} \tag{1}$$

where $u(t)$ is the output to the motors, determined by the PID-controller. The proportional term of the PID-controller establishes a general transient stability for the system. The integral term determines the error in the system that is accumulated over a certain amount of time, t, which allows for elimination of any steady-state errors. The derivate term may increase the overall responsiveness of the controller because it feeds the loop at the rate the error changes. It is important to note that the system is the most stable when the output signal is close to zero. In order for that to happen, the *gains* k_p, k_i, k_d are individually given an unique *weight* that can amplify the error and influence the total output.

Ziegler-Nichols tuning (Z-N tuning) is a method used to determine the different PID gains without the need of any mathematical model [4]. The tuning can be done by setting k_i and k_d to zero and only running the proportional part of the PID-controller. Then by manually adjusting k_p, the system will eventually reach a point where it oscillates with a stable amplitude. The gain that gives the system an oscillating character is denoted by k_c (*critical gain*) and the period of the oscillations is denoted by T_c (*critical period*). It is then possible, through k_c and T_c, to determine the remaining PID gains through a Z-N tuning chart (see Table 1.1).

k_p	k_i	k_d
$0.6k_c$	$\frac{2k_p}{T_c}$	$\frac{k_pT_c}{8}$

Table 1: Ziegler-Nichols Tuning Chart.

2 Method

2.1 Equipment

The main component of the segway was the *NXT Intelligent Brick* (NXT brick), that featured a 32-bit ARM7 microprocessor (48 Mhz) and 64 kB of RAM with a total of 4 input ports for sensors and 3 output ports for motors. The segway in this research project was equipped with HiTechnic's gyroscope, that could measure angular velocity up to $\pm 360°$ per second, and two *digital motor encoders* that could measure up to $\pm 360°$ of rotation per second for each motor, respectively. A common problem that occurs when dealing with sensors and encoders is that they do not log continuous data, but rather single data points at a constant sampling rate. Therefore, in order to compensate for the data loss, the sampling rate must be raised to capture important changes in the system. Note that the risks that come with increased sampling rate is instability in hardware performance and because of that it is important to find a good balance between sampling rate and hardware performance. In this research project, the sampling rate was set to 10 ms which from now on is denoted by t.

The segway was programmed using the RobotC IDE, a C-based programming language, that features a powerful application programming interface for accessing the native functions of the NXT brick. The actual program was transferred through a USB cable from a computer to the NXT brick and ran as a part of its embedded control system. The angular deviation, displacement and power output of the segway was logged in the debug stream of RobotC and the data was analyzed and graphed in MATLAB (seen in Figure 2 under section 3).

2.2 Implementation

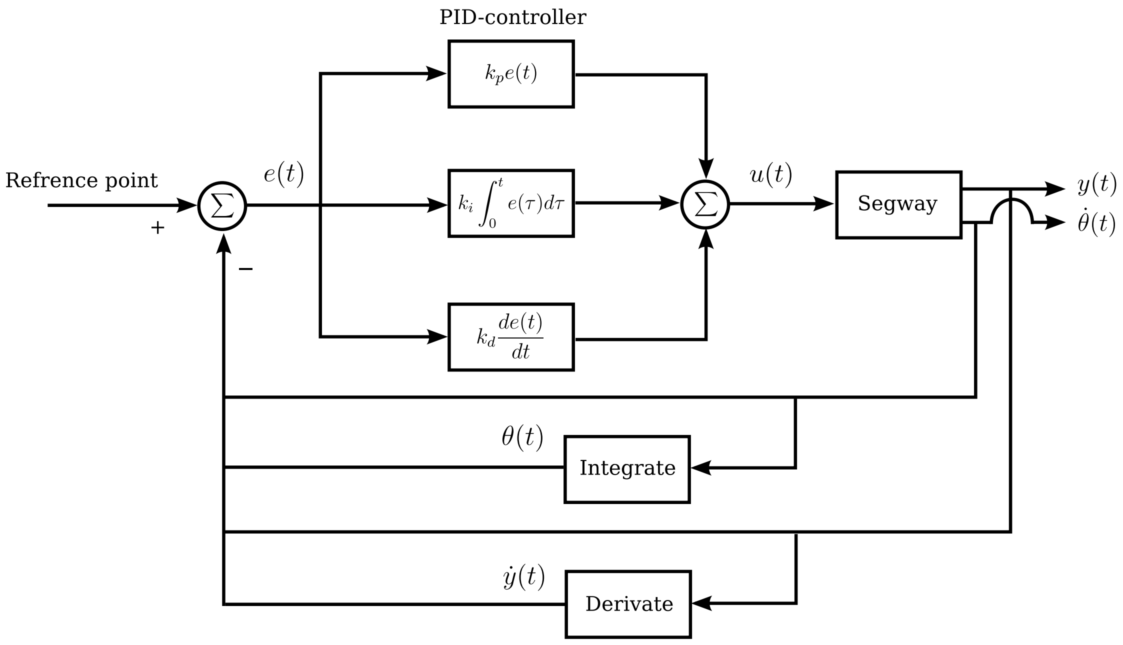

Figure 1: The complete closed-loop control system for the segway.

Figure 1 above depicts the closed-loop system of the segway in this research project. By comparing the sensor readings (two signals sent from the segway) with a reference point and then feeding a controller with the difference, the system can dynamically adjust itself with help of the output $u(t)$ to maintain stability. New feedback is then sent from the sensors for the system to compute and this continuous cycle is what makes this a closed-loop control system.

2.2.1 Calculating the error

As previously mentioned, a perfectly stable system has close to no angular deviation from the vertical position, no angular velocity, no displacement and no velocity. Because of that, our reference point must be zero. The total error in the system can therefore be described as a sum of all deviations in the system, as seen in Eq.2. How much influence each deviation may have on the error can be determined by giving each constant $k_\theta, k_{\dot{\theta}}, k_y, k_{\dot{y}}$ a weight. In this research project, the constants were determined through systematic trial and error, described further in Section 2.2.2.

$$e(t) = k_\theta \theta + k_{\dot{\theta}} \dot{\theta} + k_y y + k_{\dot{y}} \dot{y} \tag{2}$$

The error is then fed into the PID-controller as seen in Figure 1. Conceptually, this is the same as sending each and every one of the four deviations individually to its own PID-controller and then combining them. This can be proven through simple factorization:

$$u_1(t) = k_p k_\theta \theta(t) + k_i k_\theta \int_0^t \theta(\tau) d\tau + k_d k_\theta \frac{d\theta(t)}{dt} \tag{3}$$

$$u_2(t) = k_p k_{\dot{\theta}} \dot{\theta}(t) + k_i k_{\dot{\theta}} \int_0^t \dot{\theta}(\tau) d\tau + k_d k_{\dot{\theta}} \frac{d\dot{\theta}(t)}{dt} \tag{4}$$

$$u_3(t) = k_p k_y y(t) + k_i k_y \int_0^t y(\tau) d\tau + k_d k_y \frac{dy(t)}{dt} \tag{5}$$

$$u_4(t) = k_p k_{\dot{y}} \dot{y}(t) + k_i k_{\dot{y}} \int_0^t \dot{y}(\tau) d\tau + k_d k_{\dot{y}} \frac{d\dot{y}(t)}{dt} \tag{6}$$

$$u_1 + u_2 + u_3 + u_4 = k_p e(t) + k_i \int_0^t e(\tau) d\tau + k_d \frac{de(t)}{dt} \tag{7}$$

where θ can be approximated numerically with Eq.(8)

$$\theta(t) = \int_0^t \dot{\theta}(\tau)d\tau \approx \dot{\theta}(0) + t\dot{\theta}(t) \tag{8}$$

and $\dot{y}$ can approximated numerically with Eq.(9).

$$\dot{y}(t) \approx \frac{y(0) - y(t)}{t} \tag{9}$$

2.2.2 Determining the gains

Determining the correct gains for the PID-controller is one of the most important parts when producing a stable system. The weight for each gain, k_p, k_i, k_d, were determined through Z-N tuning, which is a well known method used for tuning PID-controllers. Further adjustments were done through manual tuning based on intuitive understanding of the system. For example, if the segway was overall unstable, k_p would have to be increased; If the segway was unable to maintain its position, k_i would be increased in order to amplify the error that can accumulate over time and if the segway was not responsive (or too responsive), the k_d gain would be adjusted. The weight of each constant in the error was determined through a systematic process of trial and error, where the constants were adjusted one at a time until the controller gave a desired output. See Appendix A for a list on all constants and gains that were determined.

2.2.3 Steering

Steering involves changing the reference point of the system, directing the segway to re-balance at a new point. However, because the PID-controller is prone to drastically large changes in the proportional and derivate term, the reference point has to be adjusted in small increments to prevent it from falling over. In addition, because the RobotC code runs *synchronously*, it forces the segway to prioritize balancing so steering the segway at a constant speed becomes difficult. An acceleration phase was therefore added to soften the steering process and allows the segway to travel at a constant speed. The corresponding code is seen below. For clarification purposes, the code is slightly different from the actual code seen in Appendix C.

```
y_ref = y_ref + v + acceleration * Math.pow(0.010, 2);
// where Math.pow(x, y) is the same as x^y
```

Once the segway reaches the desired speed, the acceleration shifts between being positive and negative every other time in the code loop, in order to keep a constant speed. Furthermore, for the segway to turn one motor has to apply more power than the other and this can easily be done by adjusting the output to the right or left motor, respectively. And the bigger the output difference is, the sharper the turn.

2.2.4 Error tolerance

Because the output value from the gyroscope is a non-zero value, the gyroscope readings had to be adjusted. This was done by sampling the gyroscope and then calculating the mean offset, while in a stationary state, and then subtracting it from the readings.

When the angle was integrated numerically from the angular velocity readings (see Eq.8 and Figure 1), it eventually ended up drifting because of a delay in-between each sample period. To compensate, a *filter* was implemented so that the offset dynamically could adjust itself accordingly based on the angular velocity of the segway as seen in the code below.

```
gyro_offset = gyro_offset * 0.999 + (0.001 * (gyro_rate + gyro_offset));
```

The gyroscopic offset changes in accordance to the angular velocity at the beginning of every sample period, which is enough to cancel out any drift.

3 Results

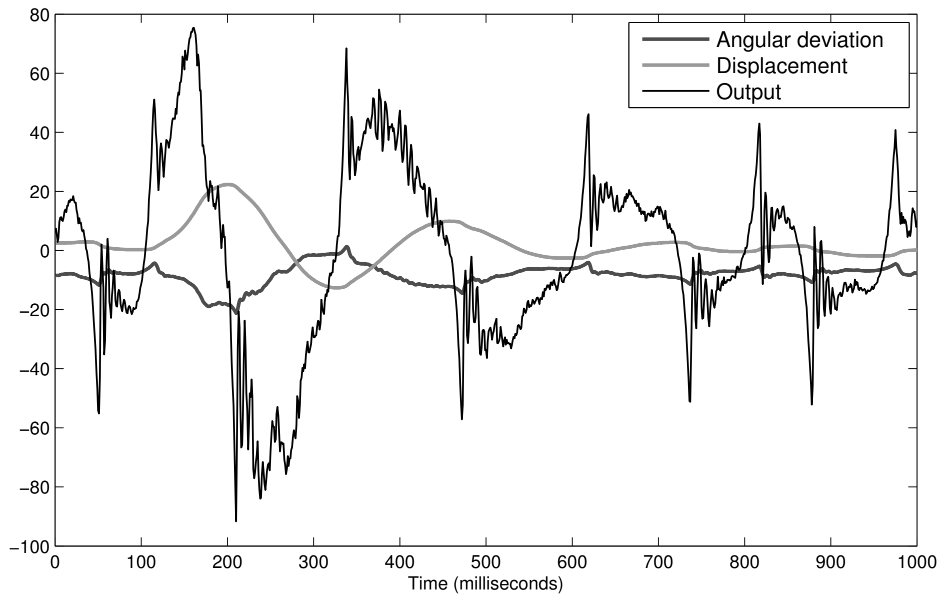

Figure 2: Balance stabilization of the segway after being disturbed.

As seen in Figure 2, by applying an external force to the segway it still managed to stabilize and return to its original reference position. The segway was able to balance for *at least* 20 minutes without problems, unaffected by drifts. In addition, it was also able to drive and turn in all directions at a constant speed.

4 Discussion

This research project has shown that a Lego segway can successfully balance with only a PID-controller in a closed-loop system. Furthermore, it is also shown that it is possible to achieve consistent balancing and steering despite the hardware limitations (64 kB RAM and 48 MHz CPU clock speed). However, a lot of useful calculations had to be discarded. For example, an accelerometer was included in a previous segway build. When it was implemented as a low-pass filter the overall performance of the closed-loop suddenly decreased, increasing the calculation time in-between each sample. The increase turned out to be enough to force the segway into an very unstable state. The accelerometer was therefore excluded in the final build, which turned out to be a small drawback when trying to analyze low-frequency changes in the system. As seen between 800 and 1000 milliseconds on the output graph in Figure 2, noise and oscillations still appeared in the output after the segway had stabilized. However, they did not affect the motors enough to actually make a noticeable impact on steering and/or balancing.

In spite of the general success, there is still a fundamental flaw in the tuning method. Ziegler-Nichols tuning is not always viable in all real-world scenarios because it does not require an extensive knowledge of the system and **does not** guarantee a desired closed-loop behavior [5, 6]. The actual complexity of the physical structure and control system of a segway therefore makes Z-N tuning a less ideal approach. Moreover, Z-N tuning requires manual tuning afterwards, making it a very time consuming process.

To investigate further, a time-efficient and streamlined way to approach PID tuning can be done through *black box modelling* (modelling with no prior information about the system) in *system identification.* In its simplest form, system identification is a statistical method used to establish a mathematical model of the system with a set of input and output data, which in this case can be logged directly from RobotC. In an email from P. Valenzuela, PhD(pva@kth.se) in July 2013, the main concept of system identification is described. It is mentioned that relation between the system model (denoted by $G(q;\theta)$) and the input/output of the system can be described as

$$\hat{y}(t) = G(q;\theta)u(t) \tag{10}$$

where $\hat{y}(t)$ is a *predicted output* (angular deviation) and $u(t)$, in this case, is the power input determined by the PID-controller. Because the closed-loop system developed in this research project is unstable, the model can be split into two parts and therefore producing "two predictions" like

$$\hat{y}_1(t) = G_s(q;\theta)u(t) \tag{11}$$

$$\hat{y}_2(t) = G_u(q;\theta)u(t) \tag{12}$$

$$\therefore \hat{y}(t) = \hat{y}_1(t) + \hat{y}_2(t) \tag{13}$$

where $G_s(q;\theta)$ is the stable part and $G_u(q;\theta)$ is the unstable part. In order to determine the parameters of the model, the squared sum of the "prediction error" should be simulated and minimized like

$$\begin{gathered}
\frac{1}{N}\sum_{t=1}^{N}(y(t)-\hat{y}_1(t))^2 = f_1(\theta) \\
\frac{1}{N}\sum_{t=N}^{1}(y(t)-\hat{y}_2(t))^2 = f_2(\theta) \\
f_{\min}(f_1(\theta)+f_2(\theta))
\end{gathered} \tag{14}$$

where $y(t)$ is the real angular deviation measured from the output, N the amount of input or output measurements made, and $f_{\min}$ represents the function `fminunc` in MATLAB. The array of values returned through `fminunc` can then be used to create a model described through z-transform transfer functions. A mathematical model of the segway can then be established by computing the minimal realization of the transfer functions. For results and code, see Appendix B.

To conclude, further steps can be taken in order to come closer to determining more accurate gains. The concept mentioned above is still rough, but will hopefully serve as guidance for future research.

5 Acknowledgements

I would like to thank Patricio Esteban Valenzuela Pacheco being a great mentor and for providing me with invaluable guidance and feedback. In addition, a big thanks to Niclas Blomberg and Niklas Everitt for being very supportive. I would also like to thank Niklas Lidström for being a great project partner, because without your co-operation the project would never have been completed. Furthermore, I would also like to thank Johannes Ostradius for providing feedback and guidance throughout the project. Thanks to ABB and Volvo for supporting Rays. Finally, a giant thanks to Rays and everyone involved, for making this project possible.

References

[1] Toor A. Foxconn begins replacing workers with robots ahead of US expansion . The Verge [Internet]. 2012 Dec 11 [Cited 2013 Jun 29]; Available at: http://www.theverge.com/2012/12/11/3753856/foxconn-shenzhen-factory-automation-manufacturing-US-expansion

[2] ACT. A Review of Segway Use and Commercialisation in the Australian Capital Territory. 2012 Feb [Cited 2012 Jun 29]; Available from: http://www.rego.act.gov.au/assets/PDFs/Segway

[3] Åstrom J, Hägglund T. PID Controllers: Theory, Design, and Tuning 2nd Edition. USA: Instrument Society of America; 1995.

[4] Schmidt K. Control System Design: PID Control and Ziegler-Nichols Method [unpublished lecture notes]. ECE441: Elective Course in Electronic, Cankaya University; 2013.

[5] Skogestad S. Probably the best simple PID tuning rules in the world. Journal of Process Control. Paper presented at: AIChE Annual Meeting; 2001 Nov 04-09; Redo, NV, USA.

[6] Zalm G. Tuning of PID-type Controllers: Literature Overview. Eindhoven: DAF (Netherlands); 22 p. Report no.:51051104-050.

[7] Farkh R, Laabidi K, Ksouri M. Computation of All Stabilizing PID Gain for Second-Order Delay System. Mathematical Problems in Engineering 2009; 2009(2009):17. doi:10.1155/2009/212053.

A Determined constants

$k_\theta = 25, k_{\dot{\theta}} = 0.23, k_y = 272.8, k_{\dot{y}} = 24.6, k_p = 0.0336, k_i = 0.2688, k_d = 0.000504, k_u = 0.056, T_u = 0.25$

B Identified system model and Matlab code

The model that fit the segway looks like following:

$$G(q;\theta) = \frac{\theta_1 + \theta_2 q^{-1} + \theta_3 q^{-2}}{1 + \theta_4 q^{-1} + \theta_5 q^{-2}} \tag{15}$$

where $\theta_1 = 0.007791, \theta_2 = -0.01039, \theta_3 = -0.0467, \theta_4 = -2.784, \theta_5 = -2.196.$

```
% Summer project, identifying system

% Author: Patricio E. Valenzuela


function J = predic_err(x,y,u,n_trans)


% Number of samples

N = length(u);


% Decomposition of G_mod into stable and unstable part

%G_mod = tf([0 x(1:2)],[1 x(3:end)],1,'variable','z^-1');

%[G_mod_s,G_mod_uns] = stabsep(G_mod);

G_mod_s = tf([x(1:2)],[1 x(5)],1,'variable','z^-1');

G_mod_uns = tf([x(3:4)],[1 x(6)],1,'variable','z^-1');

[B_s,A_s] = tfdata(G_mod_s,'v');
```

```
[B_uns,A_uns] = tfdata(minreal(G_mod_uns'),'v');

% Computation of filtered errors, stable and unstable parts
y_s = filter(B_s,A_s,u);
y_uns = filter(B_uns,A_uns,u(end:-1:1));
y_pred = y_s+y_uns(end:-1:1);

% prediction error
e_pred = y - y_pred;

% Cost function
J = (e_pred(n_trans:end-n_trans)'*e_pred(n_trans:end-n_trans))/(N-2*n_trans);

% Summer poject, identifying system
% Author: Patricio E. Valenzuela

clc
clear all

% loading data
datos = load('super_robot_data.txt');

% Creating variables
u = datos(:,3);
theta = datos(:,1);
```

```
% Optimization step
n_trans = 50;
options = optimset('MaxIter',1e6,'TolFun',1e-8,'TolX',1e-8,'MaxFunEvals',1e6,'Displ
% Initial estimate NMP + MP noise filter
x0 = [1.3,-2,-0.5,3,0.5,-2];

x = fminunc(@(x)predic_err(x,theta,u,n_trans),x0,options);
%x = fminsearch(@(x)predic_err(x,theta,u,n_trans),x0,options);

display('Estimated parameters')
display(x)

% Transfer function
G_mod_s = tf([x(1:2)],[1 x(5)],1,'variable','z^-1');
G_mod_uns = tf([x(3:4)],[1 x(6)],1,'variable','z^-1');
G_mod = minreal(G_mod_s +G_mod_uns);

% poles of G_mod
display(pole(G_mod));

% save results
save result.mat
```

C RobotC code for the segway

```
#pragma config(Sensor, S2,     SensorGyro,     sensorI2CHiTechnicGyro)
#include "hitechnic-accelerometer.h"


/* Random constants */
float SAMPLE_COUNT = 100;


/* PID constans */
float K_TH = 25;
float K_TH_D = 0.23;
float K_Y = 272.8;
float K_Y_D = 24.6;
float KP = 0.0336;
float KI = 0.2688;
float KD = 0.000504;
float dt = 0.010; // Used in all calculations and events


/* Gyro */
float gyro_angle = 0; // Angle
float gyro_angle_pre = 0; // Angle before filter
float gyro_rate = 0; // Angular velocity
float gyro_offset = 0; // Initial offset


/* Error */
```

```
float e = 0; // e(t)
float e_d = 0; // de(t)/dtb
float e_i = 0; // e(t) * dt
float e_old = 0; // Old e(t)
float pid = 0; // PID controller ==> power with the right constants

/* Motor */
float y_ref = 0; // Reference point we want to achieve
float v = 0; // Motor velocity we want to achieve
float y = 0; // Actual motor position
float y_d = 0; // Actual motor velocity
float y_old = 0;

/* Misc. motor */
const float wheel_radius = 0.042; // Meaning of life
const float degtorad = PI / 180;
float motor_power = 0; // -100 to +100 motor power
const int n_max = 7; // Number of measurements used
int n = 0;
int n_comp = 0; // Intermediate variables. Compute measured motor speed
int encoder[n_max]; // Array containing last n_max motor positions

/* More misc. motor */
int steering = 0;
int acceleration = 50;
int speed = 0;
int last_steering = 0;
int d_pwr = 0;
int straight = 0;

/* Started */
bool started_terminator = true;
  float gyroCalibrate(){
  float temp_offset = 0;
  nxtDisplayCenteredTextLine(2, "Hold still...");
  wait1Msec(1000);
  for(int i = 0; i < SAMPLE_COUNT; i++){
    temp_offset = temp_offset + SensorRaw[SensorGyro];
    wait1Msec(10);
  }
  temp_offset = temp_offset / SAMPLE_COUNT;
  eraseDisplay();
  nxtDisplayCenteredTextLine(2, "We're ready!");
  nxtDisplayCenteredTextLine(4, "Get ready in 1 sec");
  wait1Msec(1000);
  return temp_offset; // Calculate offset
}
task hyper_power_terminator_eagle(){
  memset(&encoder[0], 0, sizeof(encoder));
  gyro_offset = gyroCalibrate(); // Get calibration
```

```
started_terminator = false;
nMotorPIDSpeedCtrl[motorA] = mtrNoReg;
nMotorPIDSpeedCtrl[motorB] = mtrNoReg;
nMotorEncoder[motorA] = 0;
nMotorEncoder[motorB] = 0;
eraseDisplay();
nxtDisplayRICFile(0, 0,"swe.ric"); // Eila Ilmatar Juutilainen from SW
while(true){
  gyro_rate = SensorValue[SensorGyro];
  gyro_rate = (gyro_rate + SensorValue[SensorGyro]) / 2 - gyro_offset;
  gyro_offset = gyro_offset * 0.999 + (0.001 * (gyro_rate + gyro_offset));
  gyro_angle = gyro_angle + gyro_rate * dt;

  // Calculate ref stuffff
  if(v < speed * 10){
    v = v + acceleration * 10 * dt;
  } else if(v > speed * 10){
    v = v - acceleration * 10 * dt;
  }
  y_ref = y_ref + v * dt;

  // Calculate motor st05fx
  n++;
  if(n == n_max){
    n = 0;
```

```
}
encoder[n] = nMotorEncoder[motorA] + nMotorEncoder[motorB] + y_ref;
n_comp = n+1;
if(n_comp == n_max){
  n_comp = 0;
}
y = encoder[n] * degtorad * wheel_radius; // Position
y_d = (encoder[n] - encoder[n_comp]) / (dt * (n_max - 1)) * degtorad * wheel_ra

// Calcualte dem err0rz st0ff
e = (K_TH * gyro_angle + K_TH_D * gyro_rate + K_Y * y + K_Y_D * y_d);
e_i = e_i + e * dt;
e_d = (e - e_old) / dt;
e_old = e;
pid = (KP * e + KI * e_i + KD * e_d) / wheel_radius;

// Gyro sync
if(steering == 0){
  if(last_steering != 0){
    straight = nMotorEncoder[motorB] - nMotorEncoder[motorA];
  }
  d_pwr = (nMotorEncoder[motorB] - nMotorEncoder[motorA] - straight) / (wheel_r
} else {
 d_pwr = steering / (wheel_radius * 10);
}
```

```
    last_steering = steering;
    if(gyro_angle > 60){
      // Note, my segway does not fall backwards.
      StopAllTasks();
    }
    motor_power = pid;
    if(motor_power > 100){
      motor_power = 100;
    } else if(motor_power < -100){
      motor_power = -100;
    }
    motor[motorA] = motor_power + d_pwr;
    motor[motorB] = motor_power - d_pwr;
    wait1Msec(dt * 1000);
  }
}
int time_count = 0;
task main(){
StartTask(hyper_power_terminator_eagle);
  while(started_terminator){} // When this turns false ==> Done calibrating
  // Speed = 0.091m/s
  while(true){
    // E.g speed = 80;
  }
}
```