# Development of a Control System for Stabilizing a LEGO Segway Model

Niklas Lidström
niklas.lidstrm@gmail.com

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

Research Academy for Young Scientists
July 10, 2013

**Abstract**

The purpose of this project was to develop a control system that could stabilize and control a segway, a two wheeled, self-balancing electric vehicle. LEGO Mindstorms NXT 2.0 technology was used to build a small model segway, which was programmed with the language RobotC. With a gyroscopic sensor and digital encoders in the motors, the angle and position were measured and compared to a desired state. Using these readings, a controller calculated the power to be applied to the motors in order to keep the segway upright. The model was able to balance independently for an indefinite time, react to external forces and drive in pre-defined paths.

# Contents

**1 Introduction** **1**

1.1 Background . . . . . 2

**2 Method** **3**

2.1 Controllers . . . . . 4

2.2 Positioning and turning . . . . . 5

2.3 Determining the gains . . . . . 7

**3 Results** **8**

**4 Discussion** **10**

4.1 Stability and control . . . . . 10

4.2 Controller design . . . . . 11

4.3 Limitations . . . . . 11

4.4 Future research . . . . . 12

**Acknowledgements** **14**

**A Controller settings** **16**

**B Segway code** **16**

# 1 Introduction

Robotics is a massively expanding field, with an increasing number of industries using robots to automate various processes. By the end of 2011 there were approximately 1,153,000 operational industrial robots in use, and this number is expected to reach around 1,575,000 by 2015 [1]. However, not only industries employ robots. A self-driving subway train is also a type of robot. Often, the train conductor only controls the train doors. Self-driving cars are currently being developed by Google and other major companies. They are anticipated to provide many benefits compared to manually driven cars, such as increased safety, speed and fuel efficiency, amongst others [2, 3].

Another example is the segway, which is a two wheeled, self-balancing electric vehicle. The driver stands upright on the segway so that his center of mass is above the wheels. Mechanically, this system is comparable to an inverted pendulum with a movable base. That is, a pendulum with its center of mass above the pivot. Because an inverted pendulum is inherently unstable, it has to be dynamically stabilized by moving the base. The goal is to keep the center of mass right above the wheel axis.

To achieve this, gyroscopic sensors and accelerometers are used, along with embedded computers. This equipment must have a very fast reaction time in order to stabilize the vehicle properly. In this project, a model of the segway will be constructed using a LEGO Mindstorms NXT 2.0 control unit and two motors of the same brand, along with a gyroscope.

The aim is to develop the control system necessary for stabilizing and controlling a segway-type robot. That is, stabilizing without needless oscillation as well as possessing an ability to react to external disturbances. It should also be able to turn and drive both forwards and backwards near maximum speeds.

## 1.1 Background

In most cases, a system needs to produce a specific *output*. For example, a heating system for a building would need to make sure that the temperature in the building is constant. This temperature would be the system output. Depending on the temperature outside, the same amount of energy to the heating system could produce a different internal temperature. To ensure that the system output is the same as the desired output a controller is used, A controller is a type of feedback loop mechanism, which is named as it is because it feeds the output back to the input. An example of a feedback loop can be seen in Figure 1. It calculates an *error* based on the desired and current output, and the controller then tries to minimize this error by adjusting the inputs of the system. In the heating example, the controller would measure that the internal temperature is dropping and apply more energy, and vice versa. The desired reference output is commonly denoted $r(t)$, the error $e(t)$ and the control signal $u(t)$, which is the energy in the previous example. How the inputs and outputs are denoted depends on what is measured. Since temperature is measured in the example, it would be denoted $T(t)$.

Control systems are traditionally divided into sensors, controllers and actuators. Today, only the sensors and actuators have proper hardware components. The controller itself is often in the form of a chip or computer where the control algorithms are run. In this paper the controller for the model segway is represented mathematically. The main controller that is used is called a *PID controller*, seen in Figure 2 and Equation 3. An error controller based on PID is also used (Equation 1). The PID controller uses three separate parameters to calculate the control signal: the Proportional, the Integral, and the Derivative. Each of these terms are based on the output error, and are multiplied by coefficients $K_P$, $K_I$ and $K_D$, respectively. Within control theory, these coefficients are called *gains*.

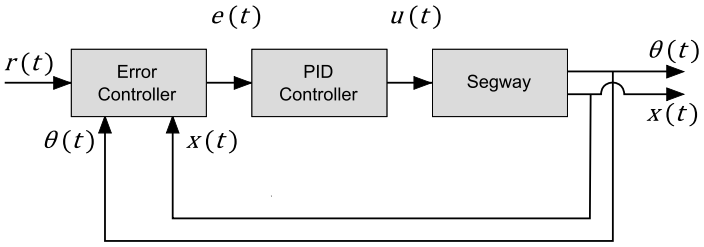


Figure 1: Flowchart of the feedback loop.

# 2 Method

The basis for the design in this project was the LEGO Mindstorms NXT 2.0 Intelligent Brick (from here referred to only as *brick*). It is an embedded computer with 4 sensor ports and 3 motor ports, which can be connected to a computer via USB. Using this and LEGO parts, a segway-like vehicle was constructed which was approximately 15 cm tall. The motors were mounted so that the bottom of the brick was approximately 4 cm from the wheel axis. The wheels were narrow with a soft rubber tyre, and had a diameter of around 7 cm.

Two electrical servo motors were used, belonging to the LEGO Mindstorms NXT 2.0 kit. The sensors used were: i) a HiTechnic gyroscope mounted on the back, which allowed for angular velocity measurements of up to 360°/s; ii) built in digital encoders in the motors, allowing accurate measurement of the movement of the wheels.

Programming the LEGO segway was done using RobotC, a C-based language designed specifically with robotics in mind. It allows programs to run multiple tasks at once, which was used in this project for calculating the reference position, turning, and balancing at the same time. The compiled program was transferred from the computer and executed on the brick.

To develop a stable system, some basic control theory was applied to this problem. A feedback loop was utilized to control the segway. The flowchart seen in Figure 1 illustrates the structure of the control system. The sensors measured the system output at regular intervals. The goal is to as low of a sampling time as possible to decrease the amount of

possible external disturbances between the samples. For this system the sampling period used was 10 ms, denoted by $T$. The angular velocity $\frac{d}{dt}\theta(t)$ and position $x$ was measured. These were fed into the error controller, where a compound error was calculated. The PID controller then used this error to calculate the power to apply to the motors.

## 2.1 Controllers

The error controller calculated a compound error for the system by using a *PD controller*, which is a PID controller without the integral term. This controller, which can be seen in Equation 1, used both angle and position values, as well as their derivatives. The gains $k_\theta$, $k_{\dot{\theta}}$, $k_x$ and $k_{\dot{x}}$ were multiplied by these terms, respectively. These four errors were summed and passed to the PID controller as the compound error $e(t)$.

$$e(t) = k_\theta \theta(t) + k_{\dot{\theta}} \frac{d\theta(t)}{dt} + k_x x_e(t) + k_{\dot{x}} \frac{dx(t)}{dt} \tag{1}$$

From the angular velocity, the angle $\theta$ was approximated numerically using Equation 2. During each sampling period, the angular velocity was measured and multiplied by $T$ to give an approximate value for the difference in angle. This value was then added to the previous angle. The reason for using this method over more precise numerical integration techniques is the requirement for a fast reaction time. Using an interpolating method creates a delay of time $T$.

The velocity $\frac{d}{dt}x(t)$ was calculated using the position measured from the encoders. It was approximated with $\frac{d}{dt}x(t) \approx \frac{x(t)-x(t-T)}{T}$. Let the positional error $x_\mathrm{e}$ be the deviation from the temporary reference position $x_\mathrm{ref}$, that is $x_\mathrm{e} = x - x_\mathrm{ref}$.

$$\theta(t) = \int_0^t \frac{d}{dt}\theta(\tau)d\tau \approx \sum_{n=0}^{t/T} T \frac{d}{dt}\theta(nT) \tag{2}$$

The PID controller seen in Equation 3 and Figure 2 uses the error calculated in the previous step. This error is used for three terms, proportional, integral and derivative. They are multiplied with a gain factor, summed, and subsequently used as the motor

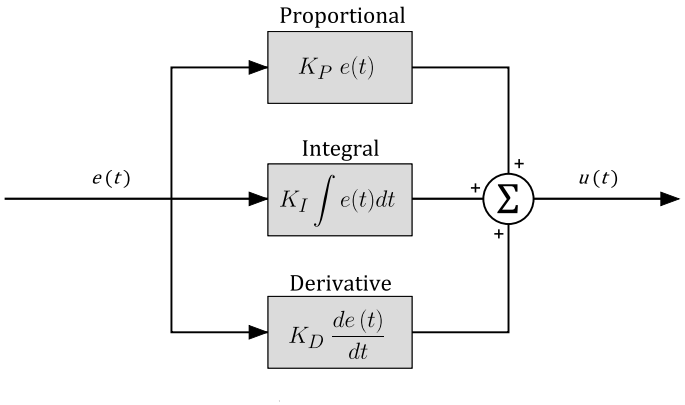


Figure 2: Flowchart of the PID controller which calculates the motor power.

power. In the case of the equipment used in this study, this power is a relative value between -100% and 100%, represented in RobotC as an integer in the range of -100 to 100.

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \frac{de(t)}{dt} \tag{3}$$

## 2.2 Positioning and turning

Part of the aim of this project was to program the segway to move at high speeds. The position and movement of the robot is decided by the reference position. To move the segway to some other position, the reference position was changed during runtime, and the segway moved to the new reference position.

However, since the position term $k_x x_e$ of the error is proportional to the deviation from the reference position, there would be some issues if the reference position was suddenly moved a large distance. The position error would become very large, and the motor power would increase to the maximum and the segway would fall over. To circumvent this, a temporary reference position is created and slowly moved toward the final reference position. For each sampling cycle the temporary reference position is adjusted by a small amount until it reaches the final reference position. It is between this temporary reference

position and the current segway position that the positional error is calculated.

All gyroscopes have some bias, which is a reading of angular velocity when the gyroscope is not moving. This was compensated for by calculating an offset by averaging 100 gyroscope readings at the start of the program and then subtracting this offset from the sensor value. This is called zeroing the gyroscope. However, this calibration was not exact. After zeroing, the gyroscope would still have a small amount of bias. Gyroscopes also have bias drift, which is the change in bias over time. This cannot be compensated for because gyroscope drift changes depending on external factors such as temperature. Over time, the bias would cause problems with the calculated angle value which was integrated from the angular velocity. The calculated angle would continually increase or decrease, even when the physical segway as upright. This angular drift also caused positional drift, where the segway offset the angular error by moving away from the reference position and creating a positional error.

Since the gyroscope cannot be recalibrated when the segway is running, this was accounted for in another way. The simplest way to solve this is to shift the calculated angle toward some true angle value $\theta_{\text{offset}}$. When the segway was standing still the angle averaged over time remained near zero. When moving, the angle was dependent on the speed of the segway. To simplify, this was estimated as a linear relationship. This shift can be seen in Equation 4. A value for $\alpha$ was chosen so that the gyroscope would contribute with the majority of the angular changes during short time periods, but not so close to 1 that angular drift would occur. A value of 0.995 was chosen for $\alpha$.

$$\theta = \alpha \cdot \theta_{\text{gyro}} + (1 - \alpha) \cdot \theta_{\text{offset}} \tag{4}$$

To determine the relationship between the speed and $\theta_{\text{offset}}$, the segway was driven straight forward at full speed for approximately 5 meters. This was performed three times. Data was recorded from the parts of the movement where the velocity was constant. A straight line was fitted to the measured angular values in MATLAB and the y-intercept was used

as the average velocity, because the slope of the lines were negligibly small. The average y-intercept from the three measurements was approximately $\theta = 3.1°$ The relationship between the speed and the angle was then calculated using $k = \frac{x'(t)}{\theta}$, where $x'(t)$ was the average velocity.

For turning, power is simply added or subtracted to the outer or inner motor, respectively (see the example from the source code below). This makes the outer wheel turn faster than inner wheel and makes the segway turn. The larger the value, the tighter the turn.

```
motor[motorA] = motor_power + turn_power;
motor[motorB] = motor_power - turn_power;
```

To set the final reference position and timing, an array is used. Table 1 represents a positioning array with example values to better explain this implementation. Each column represents one instruction. The first row represents the position and the second row represents the wait time. Looking at the first column, the segway is instructed to stay at position $x = 0$ m for 5 seconds. The second instruction orders it to move 2 m forwards. Since the wait time is set to zero this movement will be completed as fast as possible. Once it has moved 2 m, column 3 is read, which instructs it to remain still at $x = 2$ m for 10 seconds. Finally, the segway moves back to the original position as fast as possible. See appendix B for the RobotC implementation of this concept.

Table 1: Tabular representation of the position and timing matrix. Row 1 represents the position in meters from the starting position. Row 2 represents the wait time in seconds.

| | | | | |
|---|---|---|---|---|
| $x$ | 0 | 2 | 2 | 0 |
| $t$ | 5 | 0 | 10 | 0 |

## 2.3 Determining the gains

Since both a PD error controller and a PID controller were used, the amount of gain coefficients that needed to be determined also increased. In this implementation, there

were 4 gains for the error controller and 3 gains for the PID controller. Each of these gains affect the system in specific ways [4]. To determine them, trial and error was utilized. The final values and an additional description of how they are determined can be found in Appendix A.

The process began by first letting the integral gain $K_I$ be 0 and the other two PID gains be 1. Then, $k_\theta$, $k_{\dot{\theta}}$, $k_x$ and $k_{\dot{x}}$ were decided one at a time, in the listed order. They were increased or decreased until the system was somewhat stable. For example, slow reaction time to falling would warrant an increase in $k_{\dot{\theta}}$ and a twitching behaviour would suggest a decrease. These gains were adjusted using the table in Appendix A, disregarding the $KI$ row. This tuning process was mostly systematic trial and error.

Then, $K_P$, $K_I$ and $K_D$ were adjusted according to Ziegler-Nichols method [5]. Ziegler-Nichols method is a method for determining approximate gains for an automatic control system. The tuning began by setting all PID gains to 0. $K_P$ is increased until the control signal $u(t)$ oscillates at a constant amplitude. This value is called the ultimate gain, denoted $K_u$. The oscillation period was measured and denoted $T_u$. The values usedwere: $K_u = 0.056$, $T_u = 0.25$. Using Table 2 the gains can be approximated to $K_P = 0.0336$, $K_I = 0.2688$ and $K_D = 0.00105$. From here, the gains were changed one by one to see if the stability was positively influenced. $K_D$ needed to be decreased to 0.000504 to achieve stability.

Table 2: Ziegler-Nichols method.

| $K_P$ | $K_I$ | $K_D$ |
|---|---|---|
| $0.60K_u$ | $2K_p/T_u$ | $K_pT_u/8$ |

# 3 Results

Combining a PID controller with a PD error controller enabled the segway to balance independently until the batteries were depleted. The segway could stay in approximately the same position for a longer period of time, and move a predetermined distance with an

accuracy of ±0.1 m after 4 m of movement. This test was carried out three times. When standing still, some oscillation took place. The segway would change motor turn direction with a frequency of approximately 1.5 Hz, averaged from three 60 s long samples.

The segway also reacted well to external forces. It regained stability by moving backwards in the direction of the force as well as tilting in the opposite direction. After a short period of time the segway returned to its reference position. See Figure 4 for a graph of the position and angle during external disturbance.

When driving, the segway did not move at a constant speed, but rather alternated between fast and medium speed. It was not stable to external forces in the driving direction and would easily fall over if disturbed. It also turned slightly to the left when it was supposed to drive straight forward.

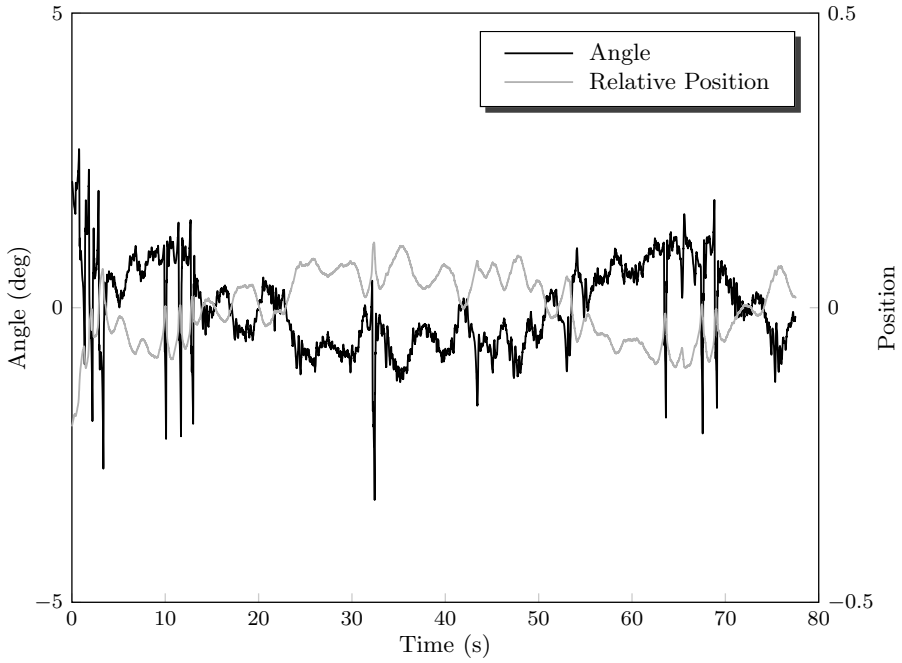


Figure 3: Segway position and angle over time while balancing without external disturbances.

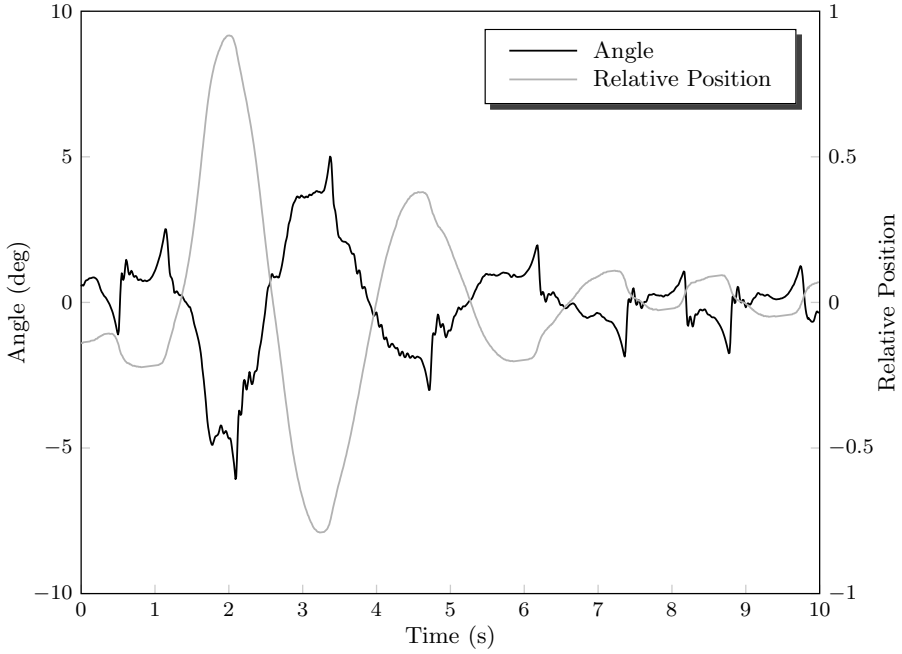


Figure 4: Segway position and angle over time while balancing while being disturbed. An external force is applied at $t \approx 1.5$ s.

# 4 Discussion

## 4.1 Stability and control

The stability of the segway was overall very satisfactory. Oscillation when balancing was not an issue because the angular and positional deviation was very small. As can be seen in figure 3, the angular deviation from 0 during undisturbed balancing is normally within the range $\pm 1°$. At some points the absolute value of the angle exceeded 3°, but returned to normal within a very short time period. After an external disturbance the segway rapidly returned to its stable position, as seen in Figure 4. At $t \approx 7$ s the segway is fully stable after a disturbance at $t \approx 1.5$ s.

As was mentioned in the Results section, the segway was not resistant to external disturbances while moving. If a force was applied to the segway in the direction of the movement, the segway would often fall. Because the motors were running near full speed, the motors could not run any faster forward to compensate for the increase in angle. This

problem cannot be avoided unless the segway is driven at lower speeds. Another issue while moving was the fact that the segway would turn to the left. One explanation for this is different friction forces in the motors, causing the left wheel to turn slightly slower even though the same power was applied to both motors. To solve this, a feedback loop using the wheel encoders to measure the rotation of the wheels could be implemented.

## 4.2 Controller design

The control system that was used for this project is very unorthodox. Normally, only a PD controller is used (a PID controller without the integral term), and the error is simply $\theta_e = \theta - \theta_{ref}$. During this project, attempts were made over 5 days to stabilize the segway using a PD controller, but they were unsuccessful. A PID controller was also tried (using the position error for the integral term) as well as only using a PD controller similar to the one which is now used as the error controller. When combining the PD and PID controllers, good results were achieved.

Determining the gains through trial and error is also a very inefficient method. Within in the industry, most PID tuning is done by commercial software packages. Mathematical loop tuning often used, which simulates a system and disturbances, and approximates the gains numerically.

## 4.3 Limitations

The angle, because it was integrated from gyroscope readings, had some noise and a delay of time $T$. However, the main problem is drift, where the measured angle changed over time even though the true angle remains constant. This causes the angle to cumulatively increase or decrease. An accelerometer could be used to measure the angle, but its readings contained too much noise to be usable when sampled frequently. This is in part due to the fact that the accelerometer cannot sense the difference between horizontal acceleration from the motors and the component of gravity which is perpendicular to the segway.

To circumvent the problem with drift, a low pass filter for the accelerometer was tried in combination with a high pass filter for the gyroscope integrated angle. This so called complementary filter can be seen in Equation 5.

$$\theta = \alpha \cdot \theta_{gyro} + (1 - \alpha) \cdot \theta_{accel} \tag{5}$$

It is used to smooth out short term changes of the accelerometer angle by setting $\alpha$ to a value close to (but less than) 1. Values between 0.95 and 0.999 were tried. However, when the accelerometer was used the segway would run the program too slowly. The LEGO Mindstorms brick has some computational limitations. Due to its low clock frequency of 48 MHz and 64 kB Random Access Memory performance is poor in some areas. Because of this, it was chosen that only the angle measurements from the gyroscope should be used in the segway.

The sampling time was chosen mainly based on some of the limitations of the motors and the brick. However, the processing power of the brick as well as the motor response time did not allow for periods under 10 ms. Thus, a sampling time of 10 ms was chosen. This should be sufficient for our LEGO model, as well as for real segway vehicles.

## 4.4 Future research

Currently, the segway can only drive in predefined paths. Because of how RobotC works, the only way to receive new information when the program is running is by using bluetooth. Thus, it would be possible to control the segway during runtime from a computer or mobile phone.

The next step onwards from this project is to simplify the control algorithm. The current implementation is needlessly complicated to adjust because it uses 7 gains and two controllers. Some of the gains have very small values and their impact is minuscule. Compensating for their removal should be possible by increasing the other gains. It is possible to stabilize a segway using only a PID controller with 3 gains, which would make

it easier to optimize the gains to minimize the segway oscillations and increase stability.

# Acknowledgements

I would like to thank my mentors Mr. Niklas Everitt, Mr. Niclas Blomberg, and Mr. Patricio Valenzuela a who helped us with our LEGO segways and computers. They also answered all our questions regarding control theory and the physics of the segway. I would also like to thank the Royal Institute of Technology for supplying us with facilities, laptop computers and LEGO Mindstorms units. Thanks to my project partner Dennis Jin, a great friend and an incredibly talented programmer and person, who contributed with a lot to the segway stabilization algorithm.

I extend my gratitude to Rays for excellence and director Mikael Ingemyr for providing us with an amazing opportunity to do research. Thanks to all other students at Rays for making these weeks such a wonderful experience. Thanks to Dr. Nicole Nova, counsellors Mariam Andersson, Johan Henriksson, and Johannes Orstadius. They have given us many laughs during these weeks, and have taught us a lot.

Finally, I would like to thank ABB and Volvo for allowing Rays to be arranged.

# References


[1] World Robotics. Executive Summary of 2012 Industrial Robots. [Online] 2012. Available from: http://www.worldrobotics.org/uploads/media/Executive_Summary_WR_2012.pdf [Accessed: June 30 2013]

[2] Cowen T. Can I See Your License, Registration and C.P.U.? New York Times. [Online] May 28 2011. Available from: http://www.nytimes.com/2011/05/29/business/economy/29view.html?_r=0 [Acccessed: June 30 2013]

[3] O'Toole R. Gridlock. [Online] Cato Institute. 2009. Available from: http://books.google.se/books?id=1I8Wuv7P13AC [Accessed: July 01 2013]

[4] Zhong J. PID Controller Tuning: A Short Tutorial. [Online] Purdue University. 2006. Available from: http://saba.kntu.ac.ir/eecd/pcl/download/PIDtutorial.pdf [Accessed: July 06 2013]

[5] Ziegler JG, Nichols NB. Optimum Settings for Automatic Controllers. [Online] 1942. Available from: http://www2.eie.ucr.ac.cr/\~valfaro/docs/Ziegler\%26Nichols.pdf [Accessed: July 09 2013]

# A Controller settings

"We are most interested in four major characteristics of the closed-loop step response. They are

1. Rise Time: the time it takes for the plant output y to rise beyond 90% of the desired level for the first time.

2. Overshoot: how much the the peak level is higher than the steady state, normalized against the steady state.

3. Settling Time: the time it takes for the system to converge to its steady state.

4. Steady-state Error: the difference between the steady-state output and the desired output."

| Response | Rise Time | Overshoot | Settling Time | S-S Error |
|---|---|---|---|---|
| KP | Decrease | Increase | NT | Decrease |
| KI | Decrease | Increase | Increase | Eliminate |
| KD | NT | Decrease | Decrease | NT |

NT: No definite trend. Minor change" (Zhong, 2006) [4]

The following gains were used for this system:

$k_\theta$ 25
$k_{\dot{\theta}}$ 0.23
$k_x$ 272.8
$k_{\dot{x}}$ 24.6

$K_P$ 0.0336
$K_I$ 0.2688
$K_D$ 0.000504

# B Segway code

This is the segway code without turning control.

```
#pragma config(Sensor, S3,     SensorGyro,     sensorI2CHiTechnicGyro)
#pragma config(Sensor, S4,     SensorAccel,    sensorI2CCustom)

#include "hitechnic-accelerometer.h"

/* Random constants */

int SAMPLE_COUNT = 100; // Number of gyro samples for calibration
float M_TO_SEG = 3.44; //Conversion factor from meters to segway units



int pos[][] = {{0,3} // Position and timing array
,{1,0}
,{1,10}
,{0,0}};
int nStates = sizeof(pos)/sizeof(pos[0]);
int currState = 0;
long currTime = 0;
int timeDiff = 0;


/* Constants */

float K_TH = 25; // Error controller: Angle gain
float K_TH_D = 0.23; // Error controller: Angular velocity gain
float K_Y = 272.8; // Error controller: Position gain
float K_Y_D = 24.6; // Error controller: Velocity gain
float KP = 0.0336; // PID controller: Proportional gain
float KI = 0.2688; // PID controller: Integral gain
float KD = 0.000504; // PID controller: Derivative gain

float dt = 0.010; // Used in all calculations and events
float ALPHA = 0.995; // Factor to multiply with gyro_angle
float k_speed_angle = 0.0016129; // Relationship between speed and angle


/* Gyro */

float gyro_angle = 0; // Angle
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; // Partial reference goal for the robot
float y_ref_final = 0; // Final reference position we want to achieve
float move_speed = 0.005; // Change in y_ref per iteration
float y = 0; // Actual motor position
float y_e = 0; //Difference between reference position and current position
float y_d = 0; // Actual motor velocity
bool y_moving = false; // Is the machine moving to a new point?
bool y_settime = true;

/* Misc. motor */

const float wheel_radius = 0.070; // Meaning of life
const float degtorad = PI / 180;
float motor_power = 0; // -100 to +100 motor power
unsigned long targetTime = 0;
//int encoder[n_max]; // Array containing last n_max motor positions
int enc=0;
int enc_old=0;

/* Started */

float gyroCalibrate(){
float temp_offset = 0;

nxtDisplayCenteredTextLine(2, "Hold still son...");
wait1Msec(1000);

for(int i = 0; i < SAMPLE_COUNT; i++){
temp_offset = temp_offset + SensorRaw[SensorGyro];
wait1Msec(5);
}
temp_offset = temp_offset / (float)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

nMotorPIDSpeedCtrl[motorA] = mtrNoReg;
nMotorPIDSpeedCtrl[motorB] = mtrNoReg;
nMotorEncoder[motorA] = 0;
nMotorEncoder[motorB] = 0;
eraseDisplay();

while(true){
currTime = nSysTime;

gyro_rate = SensorValue[SensorGyro] - gyro_offset;
gyro_angle = gyro_angle + gyro_rate * dt;

// Calculate motor st05fx

enc = nMotorEncoder[motorA] + nMotorEncoder[motorB];
y_d = (enc-enc_old)/dt * degtorad *wheel_radius;
y = enc * degtorad * wheel_radius;
y_e = y - y_ref;
enc_old = enc;

//
gyro_angle = ALPHA*gyro_angle + (1-ALPHA)*k_speed_angle*y_d;

// Calculate error

e = (K_TH * gyro_angle + K_TH_D * gyro_rate + K_Y * y_e + 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(abs(gyro_angle) > 70){
StopAllTasks();
}

motor_power = pid;
```

```
if(motor_power > 100){
motor_power = 100;
} else if(motor_power < -100){
motor_power = -100;
}

motor[motorA] = motor_power;
motor[motorB] = motor_power;

if(currState < nStates) {

// Slowly move the temporary target position towards the final target position
// Is the segway currently executing a movement command?
if(y_moving) {
if(y_ref<y_ref_final) {
y_ref += move_speed;
} else {
y_ref -= move_speed;
}

} else {
y_ref_final = pos[currState][0]*M_TO_SEG;

// If the segway is waiting for the specified time
if(pos[currState][1] != 0) {
targetTime = nPgmTime + pos[currState][1]*1000;
y_settime = true;
} else {
y_settime = false;
}
y_moving = true;
}

// Time was set
if(y_settime) {
if(targetTime > nPgmTime) {
y_moving = false;
currState++;
}

// No time was set
} else {
if((y_ref_final > 0 && y_ref+move_speed > y_ref_final) ||
( y_ref_final < 0 && y_ref-move_speed < y_ref_final)){
```

```
y_ref = y_ref_final;
y_moving = false;
currState++;

} else {
y_moving = true;
}
}

}
timeDiff = nSysTime - currTime;
wait1Msec(dt * 1000 - timeDiff);
}
}

task main(){

StartTask(hyper_power_terminator_eagle);
}
```