Self-balancing two-wheeled robot

Brian Bonafilia, Nicklas Gustafsson, Per Nyman and Sebastian Nilsson

***Abstract*— This work describes the design and implementation of a self-balancing two-wheeled robot. The system is similar to the classical unstable, non-linear mechanical control problem of an inverted pendulum on a cart. This paper derives the linearized system dynamics equations and approaches the control problem, of stabilizing the robot, using a Linear Quadratic Regulator for state feedback. Simulation results, using a Kalman Filter for state estimation, show that the controller manages to reject disturbances and stabilize the system using only a gyroscope and an accelerometer. The control algorithm is implemented using both Simulink- and C-programming. A complimentary filter is used for state reconstruction in the final implementation. A Kalman filter implementation is also approached, however it does not perform as well as the complementary filter. The resulting implementation manages to stabilize the pendulum in an upright position and reject disturbances such as gentle pushes. An alternative controller also returns the robot to its initial position in response to a disturbance.**

TABLE I

LIST OF PARAMETERS AND NUMERICAL VALUES USED

Parameter	Value	Description
M_b	0.595 kg	Mass of robot
M_w	0.031 kg	Mass of the wheels,
J_b	0.0015 kgm^2	Moment of inertia about CoM
r	0.04 m	radius of wheels
J_w	5.96e-05 kgm^2	Moment of inertia for the wheels
L_m	0.08 m	Distance from wheel axle to CoM
K_e	0.468 Vs/rad	EMF constant
K_m	0.317 Nm/A	Motor constant
R	6.69 Ohm	Motor resistance
b	0.002 Nms/rad	Viscous friction constant
g	9.81 m/s^2	Gravitational constant

I. PROJECT OVERVIEW

This report documents the design and implementation of a self-balancing robot, which is an unstable system; the basic model is that of an inverted pendulum balancing on two wheels. This work details the derivation of the model of the system and lays out the framework of the robot's control system. It also shows the full implementation of a control system stabilizing the robot.

The robot is built using Lego Mindstorm, an educational product first released by Lego in 1998. The Lego Mindstorm kit is equipped with a 32-bit ARM7 microprocessor with a bootloader modified to run the nxtOSEK real-time operating system.

Brian Bonafilia, Nicklas Gustafsson, Per Nyman and Sebastian Nilsson are with Chalmers University of Technology, in the Department of Signals and Systems {bbrian, nigus, nper, sebnil}@student.chalmers.se

In order to devise a control scheme for the robot, a mathematical model of the physical system was derived from free body diagrams and basic equations of motion. The mathematical model is then linearized around the operation point, namely a tilt angle of 0 degrees.

The control scheme used is a Linear Quadratic Regulator (LQR) for state-feedback control. The LQR is setup to drive the tilt angle to zero. The initial focus is on stabilizing the unstable system and rejecting disturbances. A complementary filter is used to reconstruct an angle estimate to use in the LQR controller. The information used to estimate the current system states comes from three primary sensors: a 3-axis accelerometer, a 1-axis gyroscope, and motor encoders.

II. RELATED WORK

The self-balancing robot is a system similar to the classical mechanical system of an inverted pendulum on a cart. It is an interesting system to control since it is relatively simple, yet non-linear and unstable. The literature is extensive from hobbyists and academic projects [3, 4, 7]; to advanced commercial products such as the Segway [5].

Optimal control with the LQ controller is often used to solve the problem of stabilizing the unstable robot system without the need for manual tuning of the state-feedback controller [1]. The balancing robot in this paper follows this common technique.

Various strategies have been used in similar projects in the past to combine the information from a gyroscope and an accelerometer. A Kalman filter is also used in [1] to arrive at the best estimate for the angle from the available signals.

For control of forward velocity of full-sized balancing vehicles, several techniques have been tested. Among them are setting a weighting on the control in the LQR and designing the controller in such a way that it is unable to eliminate the steady-state error caused by the rider leaning on the platform, forcing the controller to apply constant power to the motors which results in forward motion [1]. The robot in this paper does not have the benefit of a rider providing a constant input disturbance to control forward motion, but some similar technique of using a constant steady-state error to motivate forward motion is being considered.

Data gathering via Bluetooth has been used in similar projects in order to validate system models [2]. The strategies used to compare simulation results with real system performance will be explored as this project advances.

III. REQUIREMENTS AND SPECIFICATIONS

The system requirements are divided into two categories, depending on whether they are considered needed or desired. The robot need to:

- stabilize in an upright position
- reject disturbances, such as gentle pushes
- display information useful for debugging

It is desired that the robot has:

- modes for homing and free drive, i.e. a mode for returning to the initial position and a mode for not caring about which position the robot ends up in.

IV. Design

This section presents the robot's software- and hardware architecture.

A. Software architecture

The software architecture is divided into a digital control system, a Human Machine Interface (HMI) and an initialization module, shown in Fig. 1. A sampling module reads sensory information at 250 Hz. The sampled data is used by a state estimator that produces state estimates at the same rate. A LQR controller generates an input signal to the DC-motors by using state feedback.

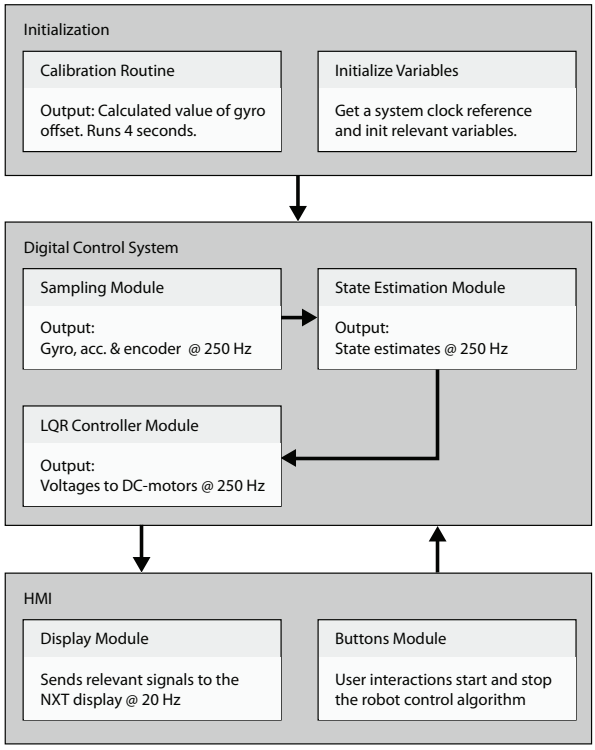

Fig. 1. A digital control system is responsible for the keeping the robot in an upright position. The angle and angular velocity is sampled at 250 Hz. A kalman filter estimated the system's states, which are used by an LQR controller to achieve a state feedback. A HMI displays relevant data and responds to user input.

B. Hardware architecture

An illustration of the hardware used in this work is depicted in Fig. 2. Two DC-motors are rigidly attached to a NXT microprocessor unit. Gyro- and an accelerometer sensors are mounted on the NXT unit, between the wheels connected to the DC-motors. The motors have encoders built in, that enables readings of the position of the robot.

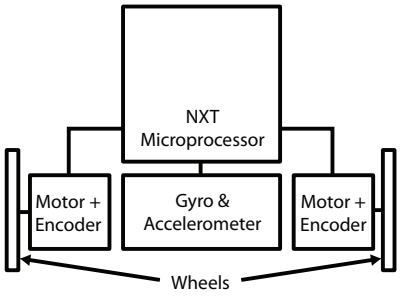

Fig. 2. Hardware architecture of self-balancing robot. Three types sensors are used; encoders, a gyro and an accelerometer.

V. System model

This section describes modelling and simulation of the self-balancing two-wheeled robot.

A. Electrical sub system

The robot's DC-motors can be modelled separately by the electric circuit in Fig. 3, where R and L represent resistive and inductive parameters of the stator windings respectively. u represents the input voltage controlling the motors. The

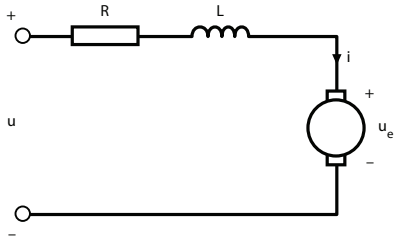

Fig. 3. The electric DC-motors are modelled separately. The winding of a motor is lumped to a resistive- (R) and an inductive (L) part. In addition, a back EMF u_e models an induced voltage that is proportional to the angular velocity generated by the motors.

back EMF u_e is assumed proportional, to the angular velocity $\dot{\theta}_w$, by a constant K_e as

$$u_e = K_e(\dot{\theta}_w - \dot{\theta}_b)$$

The torque T_1 produced by one of the DC-motors is assumed to be proportional to the winding current i, by a constant K_m.

$$T_1 = K_m i \tag{1}$$

Applying KVL yields the following equation for the electric system in Fig. 3.

$$u = Ri + L\frac{di}{dt} + u_e \tag{2}$$

B. Connecting the mechanical sub system

The mechanical system can be divided into sub systems of the wheels and the upper body (pendulum), as illustrated in Fig. 4 and Fig. 5 respectively.

Since the mechanical system dynamics is considered slow, compared to the electrical system, the current transients can

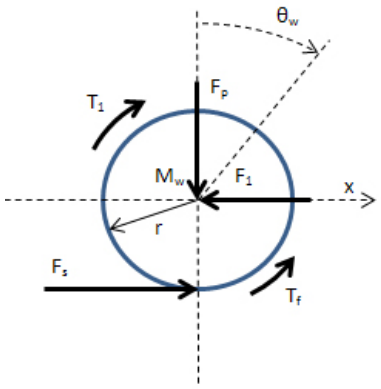

Fig. 4. An free body diagram of the torque and forces applied to one of the robot's wheels, with radius r and mass M_w.

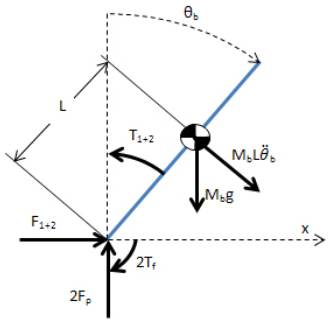

Fig. 5. The robot's upper body can be modelled by an inverted pendulum with length $2L$ and mass M_b.

be omitted. Hence, the derivative term in Eq. 2 goes to zero. Solving for i yields

$$i = \frac{u-u_e}{R} = \frac{u}{R} - \frac{K_e}{R}(\dot{\theta}_w - \dot{\theta}_b).$$

Putting this into Eq. 1 gives an expression for the torque applied by one of the DC-motors

$$T_1 = \frac{K_m}{R}u - \frac{K_e K_m}{R}(\dot{\theta}_w - \dot{\theta}_b). \tag{3}$$

The term $(\dot{\theta}_w - \dot{\theta}_b)$ is used in calculation of the back emf to give the relative rotation speed between the rotor and the stator in the motor.

The basic equations of motion for the robot's wheels are given by

$$J_w\ddot{\theta}_w = T_1 - rF_s - T_f, \tag{4a}$$
$$M_w\ddot{x}_w = F_s - F_1. \tag{4b}$$

J_w includes the inertia for the wheel, axle, and motor. The term F_s is the static friction force due to contact between the wheels and the surface the robot drives on. T_f is a friction torque, proportional to the angular velocity. Substituting T_1 in Eq. 3 into Eq. 4a returns the value of the unknown static friction force

$$F_s = -\frac{J_w}{r}\ddot{\theta}_w - \frac{T_f}{r} - \frac{K_e K_m}{rR}(\dot{\theta}_w - \dot{\theta}_b) + \frac{K_m}{rR}u. \tag{5}$$

It is assumed that there is no slipping in the wheels, hence the rotation of the wheels can be expressed in terms of their displacement (and likewise for other derivatives) as

$$\ddot{x}_w = \ddot{\theta}_w r.$$

The equations of motion for the robot's upper body, in the horizontal direction, can be written as

$$M_b\ddot{x}_b = (F_1 + F_2) - M_b L\ddot{\theta}_b\cos\theta_b, \tag{6}$$

where the last term is the tangential force's horizontal component. For the sake of generality, the subscript 1 represents the right wheel and the subscript 2 represents the left wheel.

The equation for motion about the center of mass is given by

$$J_b\ddot{\theta}_b = -(F_1+F_2)L\cos\theta_b - (T_1+T_2) + 2F_pL\sin\theta_b + 2T_f \tag{7}$$

The terms are multiplied by 2 here because it is assumed that the reaction force and friction force are the same for each wheel. Motion perpendicular to the pendulum is given by

$$M_b\ddot{x}_b\cos\theta_b = (F_1+F_2)\cos\theta_b - M_bL\ddot{\theta}_b - 2F_p\sin\theta_b + M_bg\sin\theta. \tag{8}$$

The wheel and body systems are linked together by finding an expression for the unknown reaction forces (F_1+F_2) from Eq. 6, 5 and 4b, under the assumption that any difference in rotational velocity between the two wheels is negligible

$$M_b\ddot{x}_b + M_bL\ddot{\theta}_b\cos\theta_b = -2(M_w + \frac{J_w}{r^2})\ddot{x}_w - \frac{2K_eK_m}{r^2R}\dot{x}_w + \frac{2K_eK_m}{rR}\dot{\theta}_b + \frac{2K_m}{rR}u - \frac{2T_f}{r}$$

Substituting Eq. 7 into Eq. 8 yields an expression for translation of the pendulum in terms of its rotation

$$(J_b + M_bL^2)\ddot{\theta}_b = M_bgL\sin\theta - M_b\ddot{x}_bL\cos\theta_b + 2T_f - (T_1+T_2),$$

where T_1 and T_2 is given by Eq. 3 and

$$T_f = b(\dot{\theta} - \frac{\dot{x}}{r}).$$

Note that this is a simplified form and omits the centripetal force of the pendulum entirely (it is considered to be very small). Further simplification follows when $\ddot{x}_b$ is taken to be equal to $\ddot{x}_w$ as a single variable $\ddot{x}$. The system is linearized and put on state-space by introducing small angle approximations, i.e. $\sin\theta = \theta$ and $\cos\theta = 1$. The linearized system dynamics can be written in terms of the system states and the input as

$$\dot{\xi} = A\xi + Bu,$$
$$y = C\xi + Du,$$

where

$$\xi = \begin{bmatrix} x & \dot{x} & \theta & \dot{\theta} \end{bmatrix}^T$$

is the state vector, y is the system's output and u is the DC-motors' voltage input. The A- and B matrices are given by

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \alpha & \beta & -r\alpha \\ 0 & 0 & 0 & 1 \\ 0 & \gamma & \delta & -r\gamma \end{bmatrix}$$
$$B = \begin{bmatrix} 0 & \alpha\varepsilon & 0 & \gamma\varepsilon \end{bmatrix}^T$$

where

$$\alpha = \frac{2\left(Rb - K_e K_m\right)\left(M_b L^2 + M_b r L + J_b\right)}{R\left(2(J_b J_w + J_w L^2 M_b + J_b M_w r^2 + L^2 M_b M_w r^2) + J_b M_b r^2\right)}$$

$$\beta = \frac{-L^2 {M_b}^2 g r^2}{J_b\left(2J_w + M_b r^2 + 2M_w r^2\right) + 2J_w L^2 M_b + 2L^2 M_b M_w r^2}$$

$$\gamma = \frac{-2\left(Rb - K_e K_m\right)\left(2J_w + M_b r^2 + 2M_w r^2 + L M_b r\right)}{Rr\left(2(J_b J_w + J_w L^2 M_b + J_b M_w r^2 + L^2 M_b M_w r^2) + J_b M_b r^2\right)}$$

$$\delta = \frac{L M_b g\left(2J_w + M_b r^2 + 2M_w r^2\right)}{2J_b J_w + 2J_w L^2 M_b + J_b M_b r^2 + 2J_b M_w r^2 + 2L^2 M_b M_w r^2}$$

$$\varepsilon = \frac{K_m r}{Rb - K_e K_m}$$

C. Simulation and control

As a first approach, realistic parameter values were selected. The parameters used for the DC-motors were found in [7]. Table I shows all the parameters used. They give the following system matrices

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -126.3 & -12.8 & 5.1 \\ 0 & 0 & 0 & 1 \\ 0 & 1446.9 & 213.6 & -57.9 \end{bmatrix}$$
$$B = \begin{bmatrix} 0 & 10.7 & 0 & -122.6 \end{bmatrix}^T$$
$$D = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$$

By examining the eigenvalues of the A matrix, it is clear that the linearized system has unstable poles. However, the system is stabilizable since the controllability matrix

$$W_c = \begin{bmatrix} B & AB & \cdots & A^{n-1}B \end{bmatrix}$$

has full rank. The gyro- and accelerometer sensors are used to measure θ and $\dot{\theta}$ respectively, hence the C matrix is given by

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As a result, the observability matrix given by

$$W_o = \begin{bmatrix} C & CA & \cdots & CA^{n-1} \end{bmatrix}^T,$$

where n is the order of the A matrix, has rank $< n$. This implies that the final state-space model is not fully observable. However, no state is dependant upon the unobservable state x. Thus, a minimum realization of the system can be used to reduce it to a third-order, fully controllable- and observable system. A Kalman filter, shown in Fig. 6, is used to produce an estimate $\hat{\xi}$ of the states. The Kalman gain is chosen to be

$$L = \begin{bmatrix} 0.0290 & -0.9072 \\ 0.1395 & 1.9804 \\ 0.9902 & 29.1690 \end{bmatrix}.$$

Band-limited white noise is added to the measured signal

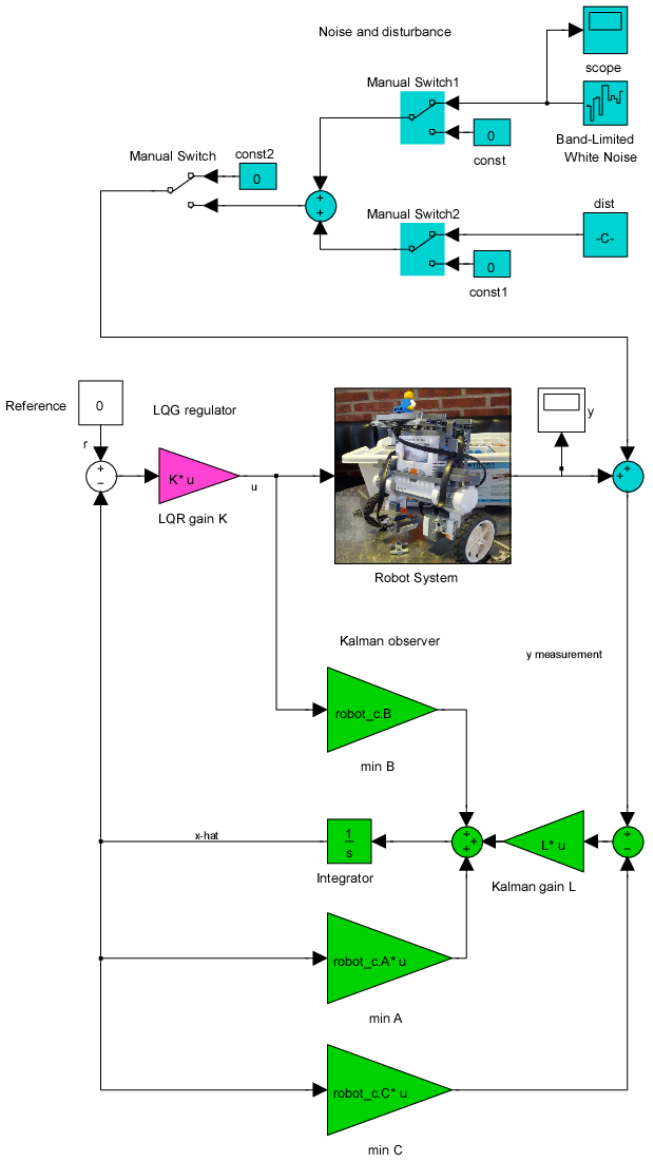

Fig. 6. A Simulink model of robot dynamics (image), sensors (blue) and LQG control system (green and purple).

to model the sensors. In addition, a constant bias is added to the angular velocity measured, to model the drifting property of the gyro sensor.

A Linear Quadratic Regulator (LQR) is used for optimal state feedback control, Fig. 7 shows how the system's states are brought to zero from an initial state

$$\xi_0 = \begin{bmatrix} 0 & 0 & .25 & -.15 \end{bmatrix}^T.$$

It is clear that the system can reject disturbances (modelled

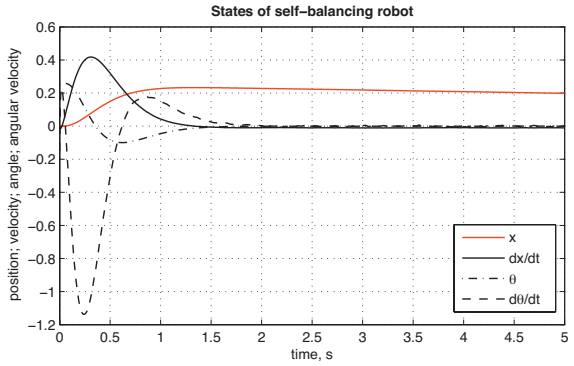

Fig. 7. The controller set-up used manages to regulate the system's observable states to zero from an initial perturbation.

by an initial perturbation). Figure 8 shows the state estimate $\hat{\xi}$, which converges to the true (observable) states from an initial guess of

$$\hat{\xi}_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T.$$

Note that the position state estimate is omitted since it is not included in the minimal realization.

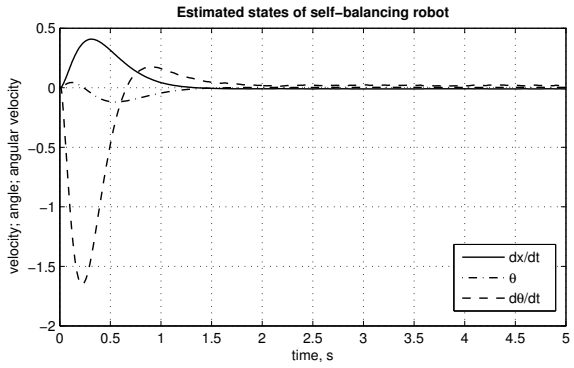

Fig. 8. The LQE algorithm produces estimates of the system's observable states. The estimates converge to the true states, from an initial guess of all states being equal to zero.

It is of interest to validate the LQG controller further, by connecting it to the non-linear model of the system dynamics. Figure 9 shows a comparison between a simulation of the linearized (red) system and the non-linear system (black). It is clear that there are some minor differences between the systems' responses, they are however considered small.

VI. Implementation

This section describes the control system implementation using Simulink and c-programming.

A. Simulink

The simulink model from which code was generated is setup as in Fig. 10. It consists of a block gathering sensor data, a small calibration routine, a state reconstruction block, a control block, and an actuator/output block.

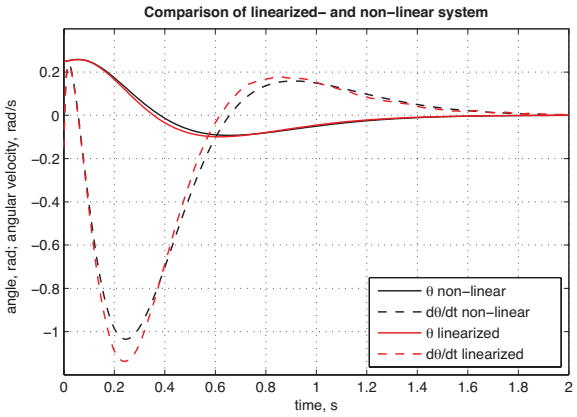

Fig. 9. A comparison between the LQG controller acting on a non-linear- (black) and a linearized (red) model of the robot. The differences between the two systems' states are considered small.

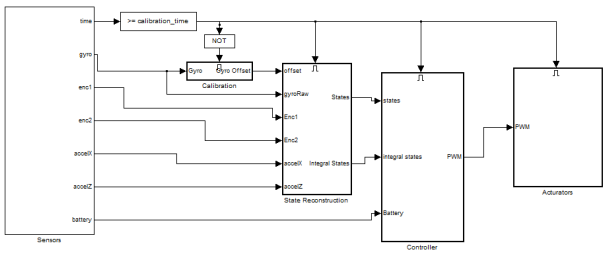

Fig. 10. The Simulink model (top level) used for control of the self-balancing robot.

1) Sensors: The first block in the Simulink model consists of the sensor input data.

The best performance was observed when the internal timer, accelerometer, gyroscope, battery level indicator, and motor encoders were used as sensor inputs. Previous attempts successfully balanced the robot with only gyroscope and encoder combinations, but the stability of the system over time was compromised by gyroscope drift. It was not possible to develop a control system that worked at different levels of battery charge without feedback from the battery indicator.

The sampling frequency used (which was the same frequency used for updating the controller) was 250 Hz. The sampling frequency was chosen based on the rise time of the closed-loop system with a continuous time controller in simulation, which was observed to be .005 s for the fastest output signal. Based on the relationship

$$bandwidth = \frac{0.35}{risetime}$$

A bandwidth for the closed-loop system is calculated to be about 70 Hz. From the Nyquist criteria, the sampling rate should be at least 140 Hz. It was observed that anything below 200 Hz struggled with disturbance rejection and was only marginally stable. The final sampling rate of 250 Hz was determined through testing as the slowest sampling frequency which would give suitably robust performance.

The difference between this and the calculated 140 Hz rate is believed to be an error introduced by inaccuracy of model parameters or by linearization of the plant.

2) State reconstruction: It was found that the states could be adequately reconstructed from sensor data without the need of model-based state estimation, as shown in Fig. 11.

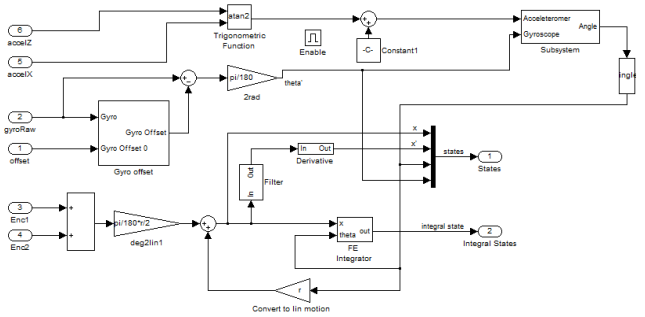

Fig. 11. State Reconstruction Block. The four system states are derived from sensor output without using model-based state estimation.

To determine the angular velocity, the signal from gyroscope was used directly after filtration and correction for drift. To reconstruct the angle from this, the gyroscope signal was integrated and combined with an estimate of the angle from the accelerometer using a complementary filter. The formation of the complementary filter is given in Fig. 12.

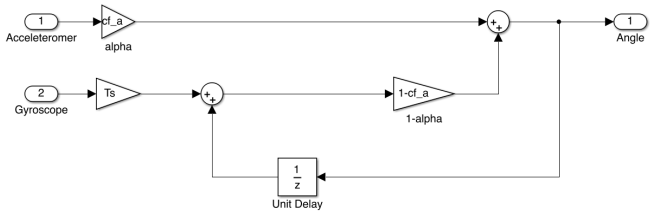

Fig. 12. A Complementary Filter Implemented in Simulink. It has the effect of passing the accelerometer signal through a low-pass filter, passing the gyroscope signal through a high pass filter, and summing the signals.

The estimate of the angle from the accelerometer was taken from the function

$$\theta = \arctan_2\left(\frac{accel_z}{accel_x}\right),$$

where $accel_z$ and $accel_x$ are the raw A/DC output of the accelerometer along the z- and x-axes. The $\arctan_2$ function takes the sign of its two inputs into account individually to return the appropriate quadrant of the angle. No problems were experienced when using the trigonometric function in the calculation.

The accelerometer angle signal was multiplied by α, a filter constant. The lower the value of α, the less weighting was put on the signal from the accelerometer. With no weighting on the accelerometer ($\alpha = 0$), the angle estimate suffered from gyroscope drift and was reliant upon an initial angle of zero at startup. With larger values of α, the acceleration of the robot began to interfere with the angle estimates and led to instability.

Through trial and error, a value of $\alpha = .005$ was found to correct the angle estimates' drifting problem without introducing any errors due to acceleration. Figure 13 shows the difference in angle recorded by the robot while balancing around an unstable equilibrium point for values of $\alpha = .005$ and $\alpha = 0$.

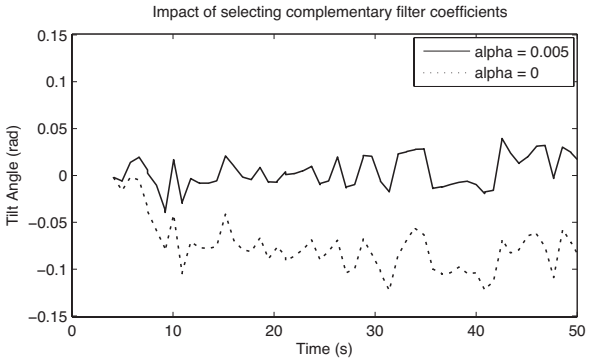

Fig. 13. Using the accelerometer to remove the bias in the gyroscope's estimate of the current angle as a result of non-zero starting conditions and measurement drift.

The signals from the motor encoders and battery level indicator were passed through a low-pass filter. The derivative of the wheel position signal from the encoders was used to determine the wheel speed.

Reconstructing proper states was crucial to the success of the control algorithm. Numerous attempts at filtration of the gyroscope signal were made. Ultimately, a method of determining an estimate of the angle from the gyroscope signal was found in [6]. By passing the gyroscope signal through a low-pass filter to determine the offset continuously during operation a stable angular velocity estimate could be made which was sufficient for both angular velocity and angular position states.

Nevertheless, in addition to the complementary filter, model-based state estimation was approached by using a Kalman filter. Figure 14 and 15 shows the estimates produced by the Kalman filter, compared to results achieved using the complementary filter. It is clear that the filters gives similar results and that the Kalman filter produces states estimates with more "spikes". Note that the Kalman implementation does not use as many sensors as the complimentary; it does not use the encoders to produce its estimates.

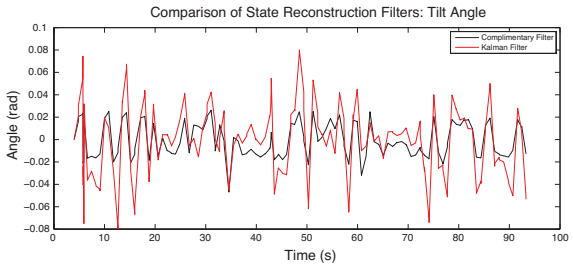

Fig. 14. Tilt angle state generated by Kalman- and complementary filter respectively.

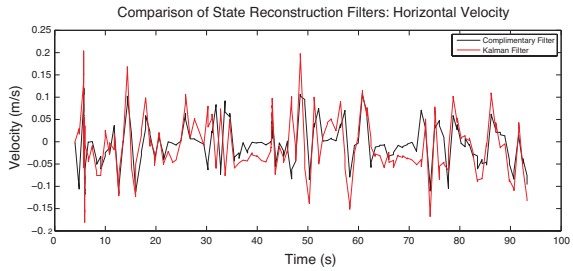

Fig. 15. Linear velocity state generated by Kalman- and complementary filter respectively. Note that the complementary filter uses one extra sensor (encoder) to achieve its reconstruction.

With this in mind, the linear velocity state estimate in Fig. 15 looks quite good.

The states given by the complementary filter showed to be more useful when implementing a control algorithm. It is hence used throughout the remaining sections of this report. However, the Kalman filter could possibly be tuned, by adjusting the Kalman gain, to give a better result.

3) Controller: The control algorithm used was an LQR based on the state information provided from the sensors. Additionally, integral states on position and angular position were calculated, resulting in a new state space

$$\bar{A} = \begin{bmatrix} A & 0_{4x1} & 0_{4x1} \\ C_1 & 0 & 0 \\ C_2 & 0 & 0 \end{bmatrix}$$
$$\bar{B} = \begin{bmatrix} B & 0 & 0 \end{bmatrix}^T,$$

where the sub-matrices are

$$C_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$
$$C_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$$

Adding the integral states on both position and angle allows offset-free control of the robot. With proper weighting on the position state and the integral of position state the robot can return to its original position in response to a disturbance. With weighting only on angle and integral of the angle, the robot will balance itself without regard to the final position it comes to rest in.

The integral states are passed from the State Reconstruction block to the Controller block, which is shown in Fig. 16. In the Controller block, the battery voltage is used as a divider to determine the required PWM signal. The input returned by the LQR is the voltage which should be applied to the motors; the battery voltage level is what voltage will be applied at 100% duty cycle for the PWM signal. Including the battery voltage level here guarantees that the outgoing PWM signal corresponds to the same voltage target provided by the LQR, regardless of charge level (up to saturation, of course).

Two control schemes were developed. The first would stabilize the robot at its original position. This was accomplished by putting larger penalties on the position and integral of the position states, as shown in the follow weight

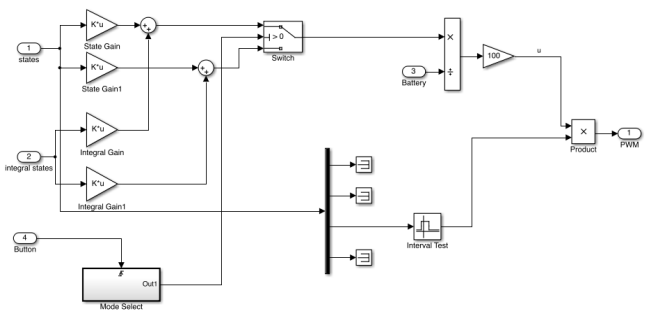

Fig. 16. The control algorithm for the self-balancing robot. Full-state feedback is used on all four states as well as on the integral of the position and tilt angle. To save battery life and prevent damage, the controller is setup to stop once the robot tilts beyond 40 degrees, a point at which it is unable to stabilize itself. Note that this figure contains a block to switch between two different controllers.

matrix

$$Q_1 = \begin{bmatrix} 6 \cdot 10^4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 \cdot 10^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
$$R = 200$$

The resulting state feedback gains are

$$k_f = \begin{bmatrix} -18.5 & -33.0 & -36.7 & -3.8 \end{bmatrix}$$
$$k_i = \begin{bmatrix} -1.4 & 0.0 \end{bmatrix}$$

The second scheme allowed the robot to balance itself without concern as to where it ended up. If pushed it would roll forward to stabilize itself and remain stationary wherever it came to a stop. Penalties were put on the angle and integration of the angle only

$$Q_2 = \begin{bmatrix} 10^{-3} & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 \cdot 10^4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10^{-3} & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 \cdot 10^2 \end{bmatrix}$$
$$k_f = \begin{bmatrix} -0.3 & -28.5 & -36.9 & -3.3 \end{bmatrix}$$
$$k_i = \begin{bmatrix} 0.0 & 0.1 \end{bmatrix}$$

It is observed that the gains applied to the horizontal velocity, angular velocity, and angle are approximately the same in both cases. This gain is a function of both the Q matrix weighting as well as the linearized model of the system, which establishes the relationship between the states. That the control algorithm in both cases results in what is expected is taken as evidence that the linear model derived earlier is correct.

4) Actuators: The final block in the Simulink implementation delivers the PWM signal to the DC motors. The current setup drives the motors with the same voltage input. Modifications to allow turning and control of steering would instead provide different signals to the motors, under the

condition that the average driving voltage between the two motors be sufficient to stabilize the system.

The DC motors are highly nonlinear components. They feature a significant backlash as well as a very large deadzone. Attempts were made to account for the deadzone in the control algorithm by increasing the PWM duty cycle by 45% in the appropriate direction, but this resulted in very shaky behavior and led to frequent stalling when the robot approached an equilibrium point. With suitable controller gains, it was not necessary to account for the motor deadzone to get adequate controller performance.

B. C-programming

The Simulink controller implementation, in Section VI-A, was reproduced by programming the robot in C code. The final C implementation includes state estimation, using a complimentary filter, and control using LQR state feedback. No major difference in performance was found when running a qualitative test, comparing the C- and Simulink implementations. The robot was able to stabilize itself and managed to reject disturbances such as gentle pushes.

The implementation is found in the Appendix. Three tasks are used in the implementation. A background task is responsible for updating the NXT display with variables used during debugging, the background task is assigned a low priority. A medium priority task, responsible for initialization, calibration and control, is run at 250 Hz. A third task, also run at 250 Hz, reads sensor values and updates the states using a complementary filter.

VII. Validation

Figure 17 shows the results of two runs, using different Q-matrices. The system is exposed to perturbations, applied as gentle pushes, in both runs. It is clear that a state feedback controller, with higher penalties on the position state, makes the robot want to go back to where it started. In addition, it is also clear that both controllers manage to reject the disturbances to stay in an upright position.

VIII. Discussion

In the initial part of the project a number of requirements were formulated for the robot. The requirements were categorized into either a needed requirement or a desired requirement, where the second category was to be done depending on time and success of the needed requirements.

It was defined that the robot needed to be able to stand in upright position and reject disturbances. As seen in previous section both these requirements were fulfilled.

The sensors were one of the most difficult part of the project to get to function properly and also one of the most important. However, when the filter was tuned properly, the control scheme worked well. Since the complementary filter worked well the more complex Kalman filter was not implemented as initially planned.

Another critical factor in the control scheme was the sampling frequency. With a sampling frequency of 100 Hz, no suitable controller could be developed. When the

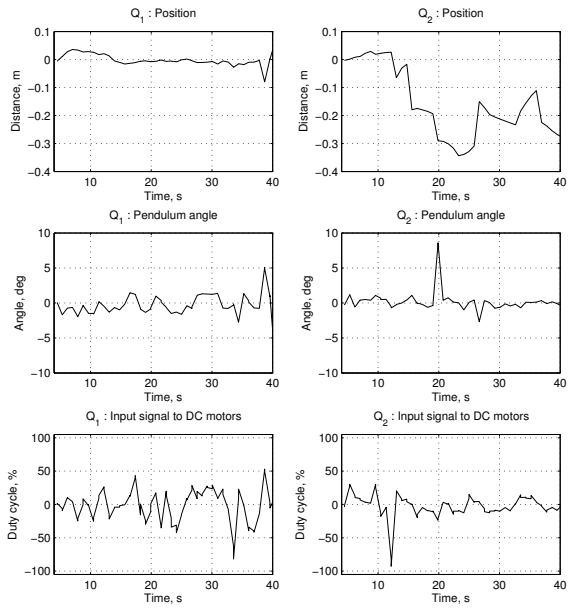

Fig. 17. Logged results of the angle- and position states, together with the corresponding input signal, for two different Q-matrices. The results show that using a Q-matrix with a low penalty (Q_2) on the position state makes the robot not caring about returning to its initial position. The data's low resolution is due to the slow and unreliable Bluetooth connection.

sampling frequency of 250 Hz was decided upon, stabilizing the system became very easy using the LQR. At 200 Hz, the system was marginally stable at best.

The other planned features, such as steering and speed control, have been deemed to be unimportant relative to stabilization and have not yet been implemented.

APPENDIX
C CODE IMPLEMENTATION

```
// nxt_config.h
#ifndef _NXT_CONFIG_H_
#define _NXT_CONFIG_H_

#include "ecrobot_interface.h"

/* NXT motor port configuration */
#define PORT_MOTOR_R NXT_PORT_B
#define PORT_MOTOR_L NXT_PORT_C

/* ENCODER DEF */
#define PORT_ENC_R NXT_PORT_B
#define PORT_ENC_L NXT_PORT_C

/* NXT sensor port configuration */
#define PORT_ACC   NXT_PORT_S3
#define PORT_GYRO    NXT_PORT_S2
/* NXT Bluetooth configuration */
#define BT_PASS_KEY  "1234"
#endif
```

```
// main.c
#include "kernel.h"
#include "kernel_id.h"
#include "ecrobot_interface.h"
#include "nxt_config.h"
#include <math.h>
#define PI 3.14159265

// mode of the robot
typedef enum {
  INIT_MODE,   // init
  CAL_MODE,    // calibrate gyro
  CONTROL_MODE // control system
} MODE;

MODE nxtway_gs_mode = INIT_MODE;

// states
double theta = 0;
double thetadot = 0;
double x[] = {0.0, 0.0};
double xFiltered[] = {0.0, 0.0};
double xdot = 0;
double angle_acc = 0;

// variables
double batteryFiltered[] = {0.0, 0.0};
double batteryDen;
double gyroOffsetFiltered[] = {0.0, 0.0};

// coefficients
static double a_b = .8;
static double a_gd = 0.999;
static double r = .04;
static U32 cal_start_time = 0;

/* hook */
void ecrobot_device_terminate(void) {
  ecrobot_set_motor_speed(NXT_PORT_B, 0);
  ecrobot_set_motor_speed(NXT_PORT_C, 0);
  ecrobot_term_bt_connection();
}

/* hook */
void ecrobot_device_initialize(void) {
   ecrobot_init_bt_slave(BT_PASS_KEY);
}

DeclareCounter(SysTimerCnt);

void user_1ms_isr_type2(void) {
  /* Activate periodical Tasks */
  (void)SignalCounter(SysTimerCnt);
}

/* 4 ms: calibration and control*/
TASK(OSEK_Task_ts1) {
  S8 pwm_l;
  S8 pwm_r;
  double stateFeedback;
  double tmp;

  switch(nxtway_gs_mode){
    case(INIT_MODE):
      pwm_l = 0;
      pwm_r = 0;
      nxt_motor_set_count(PORT_MOTOR_L, 0);
      nxt_motor_set_count(PORT_MOTOR_R, 0);
      cal_start_time = ecrobot_get_systick_ms();
      nxtway_gs_mode = CAL_MODE;
      break;

    case(CAL_MODE):
      // Calibrate the gyro offset
      tmp = gyroOffsetFiltered[0];
      gyroOffsetFiltered[0] = a_b*gyroOffsetFiltered[1] +
           (1.0-a_b) * ((double)ecrobot_get_gyro_sensor(
           PORT_GYRO));
      gyroOffsetFiltered[1] = tmp;

      // count to 4k ms
      if ((ecrobot_get_systick_ms() - cal_start_time) >=
           4000U) {
        // beep
        ecrobot_sound_tone(440U, 500U, 30U); /* beep a
             tone */
        nxtway_gs_mode = CONTROL_MODE;
      }
      break;

    case(CONTROL_MODE):
      // LQR CONTROLLER
      stateFeedback = (18.5001*x[0] + 33.0438*xdot +
           36.7429*theta + 3.7731*thetadot)*100.0/
           batteryDen;

      // saturation
      if (fabs(stateFeedback) > 100.0) {
        stateFeedback = stateFeedback/fabs(stateFeedback)
             *100.0;
      }

      // set motor outputs
      pwm_l = stateFeedback;
      pwm_r = stateFeedback;

      nxt_motor_set_speed(PORT_MOTOR_L, pwm_l, 1);
      nxt_motor_set_speed(PORT_MOTOR_R, pwm_r, 1);
      break;

    default:
      // unexpected mode. bad robot
      nxt_motor_set_speed(PORT_MOTOR_L, 0, 1);
      nxt_motor_set_speed(PORT_MOTOR_R, 0, 1);
      break;
  }

  TerminateTask();
}

/* 4 ms: sample and estimate */
TASK(OSEK_Task_ts2) {
  // Battery denominator calculation for charge
       compensation
  double tmpBat = batteryFiltered[0];
  batteryFiltered[0] = a_b * batteryFiltered[1] + (1.0-a_b
       ) * ((double)ecrobot_get_battery_voltage());
  batteryFiltered[1] = tmpBat;
  batteryDen = 0.001089*batteryFiltered[0] - .625;

  if(nxtway_gs_mode == CONTROL_MODE) {
    // Read encoders
    int enc_r = nxt_motor_get_count(PORT_ENC_R);
    int enc_l = nxt_motor_get_count(PORT_ENC_L);

    // Read gyro and accel.
    double gyroRaw = ecrobot_get_gyro_sensor(PORT_GYRO);
    S16 accValues[3];
    ecrobot_get_accel_sensor(PORT_ACC, accValues);

    // Continue to filter offset with slower filter
    double tmp = gyroOffsetFiltered[0];
    gyroOffsetFiltered[0] = a_gd*gyroOffsetFiltered[1] +
         (1.0-a_gd) * gyroRaw;
```

```
    gyroOffsetFiltered[1] = tmp;

    // Complimentary filter
    const double Ts = 0.004;
    const double alpha = .005;
    angle_acc = atan2(accValues[2],accValues[0])+4.4*PI
         /180.0; // in rad
    thetadot = (gyroRaw - gyroOffsetFiltered[0])*PI/180.0;
          // in rad
    theta = (1.0-alpha)*(theta+thetadot*Ts)+(alpha*
         angle_acc);  // -> in rad

    // Position states
    x[1] = x[0]; // remember last position for
         differentiation
    x[0] = ((double)(enc_r + enc_l))*(PI*r)/(180.0*2.0) +
         theta*r;

    xFiltered[1] = xFiltered[0];
    xFiltered[0] = (1-a_b)*x[0] + xFiltered[0]*a_b;

    xdot = (xFiltered[0]-xFiltered[1])/Ts;
  }
  TerminateTask();
}

// Background task
TASK(OSEK_Task_Background) {
  while(1){
    // Show relevant signals on display (used for
         debugging)
    display_clear(0);
    int i = 0;

    display_goto_xy(0, i++);
    display_int((int)x[0]*180.0/PI, 6);
    display_goto_xy(0, i++);
    display_int((int)xdot*180.0/PI, 6);
    display_goto_xy(0, i++);
    display_int((int)gyroOffsetFiltered[0], 6);
    display_goto_xy(0, i++);
    display_int(angle_acc*180.0/PI, 6);
    display_goto_xy(0, i++);
    display_int(theta*180.0/PI, 6);

    display_update();
    systick_wait_ms(50); /* 50msec wait */
  }
}
```

```
// main.oil : task set-up

#include "implementation.oil"

CPU ATMEL_AT91SAM7S256 {
  OS LEJOS_OSEK {
    STATUS = EXTENDED;
    STARTUPHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = FALSE;
    USEGETSERVICEID = FALSE;
    USEPARAMETERACCESS = FALSE;
    USERESSCHEDULER = FALSE;
  };

  /* Definition of application mode */
  APPMODE appmode1{};

  /* Definitions of a periodical task: OSEK_Task_ts1 */
  TASK OSEK_Task_ts1 {
    AUTOSTART = FALSE;
    PRIORITY = 2;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    STACKSIZE = 512; /* bytes */
  };

  ALARM OSEK_Alarm_task_ts1  {
    COUNTER = SysTimerCnt;
    ACTION = ACTIVATETASK
    {
      TASK = OSEK_Task_ts1;
    };
    AUTOSTART = TRUE
    {
      APPMODE = appmode1;
      ALARMTIME = 1;
      CYCLETIME = 4;
    };
  };

  /* Definitions of a periodical task: OSEK_Task_ts2 */
  TASK OSEK_Task_ts2 {
    AUTOSTART = FALSE;
    PRIORITY = 3;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    STACKSIZE = 512; /* bytes */
  };

  ALARM OSEK_Alarm_task_ts2 {
    COUNTER = SysTimerCnt;
    ACTION = ACTIVATETASK
    {
      TASK = OSEK_Task_ts2;
    };

    AUTOSTART = TRUE
    {
      APPMODE = appmode1;
      ALARMTIME = 1;
      CYCLETIME = 4;
    };
  };

  /* Definition of background task: OSEK_Task_Background
       */
  TASK OSEK_Task_Background {
    AUTOSTART = TRUE
    {
      APPMODE = appmode1;
    };

    PRIORITY = 1; /* lowest priority */
    ACTIVATION = 1;
    SCHEDULE = FULL;
    STACKSIZE = 512; /* bytes */
  };

  /* Definition of OSEK Alarm counter: SysTimerCnt */
  COUNTER SysTimerCnt {
    MINCYCLE = 1;
    MAXALLOWEDVALUE = 10000;
    TICKSPERBASE = 1;
  };
};
```

REFERENCES

[1] Mikael Arvidsson and Jonas Karlsson. Design, construction and verification of a self-balancing vehicle. Online, 2012. http://publications.lib.chalmers.se/records/fulltext/163640.pdf.

[2] Stephan Blaha, Divij Babbar, Iram Gallegos, and Kubica Matej. Optimal control project. Online, 2012. http://www.matejk.cz/zdroje/nxtway-gs-evaluation.pdf.

[3] Mitch Burkert, Taylor Groll, Thao Lai, Tyler McCoy, and Dennis Smith. Segway design project. Online, 2004. http://etidweb.tamu.edu/classes/Entc%20462/Segway/final_report.pdf.

[4] Sebastian Nilsson. Sjalvbalancerande robot. Online, 2012. http://sebastiannilsson.com/k/projekt/selfbalancing-robot/.

[5] Segway. Segway - the leader in personal, green transportation. Online, 2013. http://www.segway.com.

[6] MathWorks Simulink Team. Simulink support package for lego mindstorms nxt hardware (r2012a). Online, 2013. http://www.mathworks.com/matlabcentral/fileexchange/35206-simulink-support-package-for-...lego-mindstorms-nxt-hardware-r2012a.

[7] Yorihisa Yamamoto. Nxtway-gs (self-balancing two-wheeled robot) controller design. Online, 2009. http://www.mathworks.com/matlabcentral/fileexchange/19147.