Optimal Control Project

Štěpán Bláha, Divij Babbar,
Iram Gallegos, Kubička Matěj

June 9, 2012

LQR control

SELF-BALANCING TWO-WHEELED ROBOT BUILT WITH LEGO MINDSTORMS

Chapter 1

Introduction

The goal of this project was to design, simulate, create, test and measure a self-balancing two-wheeled robot with LQR regulator. Mechanically, the robot is based on LEGO NXT construction kit. The robot is modelled in Matlab Simulink with ECRobot API for LEGO Mindstorms. For our purposes, we have utilized already existing example called NXTway-GS, which is included in ECRobot library.

1.1 Lego NXT Brick

The NXT brick is a programmable device focused in robotics, and was developed in released by Lego in 2006. It has a series of products designed to work with it achieving some robotics tasks. Its main part is the NXT Intelligent Brick, which can take up to four sensors as an input and control 3 actuators using RJ12 cables. The brick has a 100x60 pixel monochrome LCD display, and 4 push buttons which can be used to interact with the SW loaded inside. A speaker is also part of the system, and can play sound files at around 8 kHz. The brick used by our group has a Li-Ion rechargeable battery, but there's also the option of using 6 AA batteries of 1.5 V each.
The device can be used in a variety of ways depending on which peripherals are attached to it and the control models which are executed with it. In this document we will present the work done in the NXTway-GS robot.

1.2 Matlab, Simulink and RTW-EC

The whole system is described in Matlab and Simulink, both for simulation and for the code generation that must be uploaded to the NXT brick in order to do the real execution. It is necessary to download the *Embedded*

Figure 1.1: The LEGO NXT Brick

Coder Robot NXT from (ecrobotnxt12) as an environment for our models files. Along with this, it is necessary to have the following Matlab toolbox requirements:

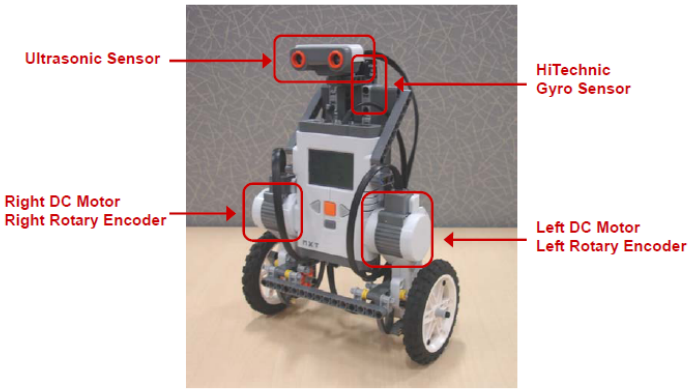
Product	Version	Release
MATLAB®	7.5.0	R2007b
Control System Toolbox	8.0.1	R2007b
Simulink®	7.0	R2007b
Real-Time Workshop®	7.0	R2007b
Real-Time Workshop® Embedded Coder	5.0	R2007b
Fixed-Point Toolbox (N1)	2.1	R2007b
Simulink® Fixed Point (N1)	5.5	R2007b
Virtual Reality Toolbox (N2)	4.6	R2007b

A group of codes and model files were given to us in order to design the Linear Quadratic Regulator control of the system and use it in the environment. These files, which will be analyzed further in this document, are the following. Also, together with Matlab and Simulink, RTW-EC is used to generate the files which are uploaded to the NXT brick. RTW-EC is a subset of RTW (which is a real time workshopp developed by Mathworks) containing the Embedded Coder, which can generate C code directly from the Simulink Models. Further information is given in the Installation section of this document.

File	Description
iswall.m	M-function for detecting wall in map
mywritevrtrack.m	M-function for generating map file (track.wrl)
nxtway_gs.mdl	NXTway-GS model (It does not require Virtual Reality Toolbox)
nxtway_gs_controller.mdl	NXTway-GS controller model (single precision floating-point)
nxtway_gs_controller_fixpt.mdl	NXTway-GS controller model (fixed-point)
nxtway_gs_plant.mdl	NXTway-GS plant model
nxtway_gs_vr.mdl	NXTway-GS model (It requires Virtual Reality Toolbox)
param_controller.m	M-script for controller parameters
param_controller_fixpt.m	M-script for fixed-point settings (Simulink.NumericType)
param_nxtway_gs.m	M-script for NXTway-GS parameters (It calls param_***.m)
param_plant.m	M-script for plant parameters
param_sim.m	M-script for simulation parameters
track.bmp	map image file
track.wrl	map VRML file
vrnxtwaytrack.wrl	map & NXTway-GS VRML file

1.3 NXTway-GS

Along with the brick, we have a series of peripherals which are part of our whole system. We have an ultrasonic sensor, a HiTechnic Gyro sensor, Right and Left rotary encoders and DC motors, which can be shown in the next figure [4].



The values for the constants that will be later used in the mathematical analysis of the system are shown below.

The values for J_m, n, f_m and f_W are taken as seem appropriate because

$g = 9.81$	$[m/\sec^2]$	:	Gravity acceleration
$m = 0.03$	$[kg]$	:	Wheel weight
$R = 0.04$	$[m]$	:	Wheel radius
$J_W = mR^2/2$	$[kgm^2]$	:	Wheel inertia moment
$M = 0.6$	$[kg]$	:	Body weight
$W = 0.14$	$[m]$	:	Body width
$D = 0.04$	$[m]$	:	Body depth
$H = 0.144$	$[m]$	:	Body height
$L = H/2$	$[m]$	:	Distance of the center of mass from the wheel axle
$J_\psi = ML^2/3$	$[kgm^2]$	:	Body pitch inertia moment
$J_\phi = M(W^2 + D^2)/12$	$[kgm^2]$	:	Body yaw inertia moment
$J_m = 1\times10^{-5}$	$[kgm^2]$	:	DC motor inertia moment
$R_m = 6.69$	$[\Omega]$	:	DC motor resistance
$K_b = 0.468$	$[V\sec/rad]$	:	DC motor back EMF constant
$K_t = 0.317$	$[Nm/A]$	:	DC motor torque constant
$n = 1$		:	Gear ratio
$f_m = 0.0022$		:	Friction coefficient between body and DC motor
$f_W = 0$		:	Friction coefficient between wheel and floor.

Figure 1.2: A list of variables, taken from [2]

they are difficult to measure, as well as the values from R_m, K_b and K_t, which were taken from a previous project [2]

1.4 Software requirements - Installation

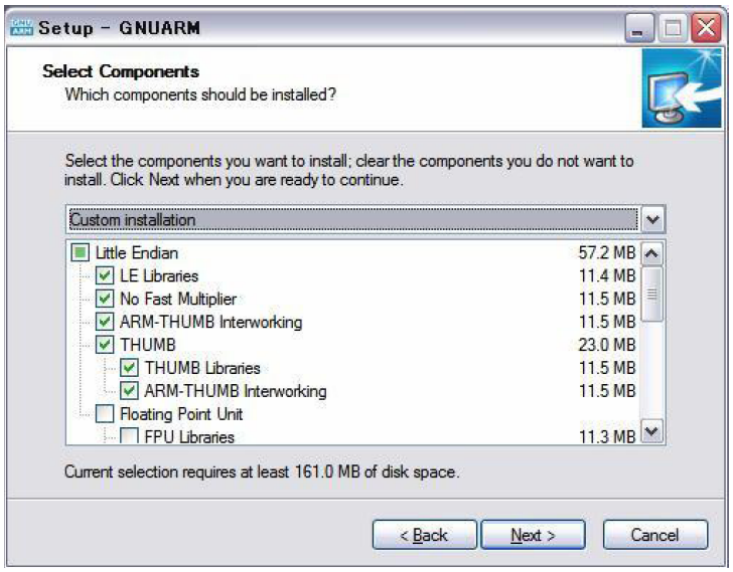
There are several references about the installation of the *Embedded Coder Robot NXT*. We needed to use more than one in order to have a fully functional application, so here we present the steps we followed for this installation.

After downloading *Embedded Coder Robot NXT*, it's recommended to create a "'MATLAB" folder in the hard disk of the computer (C: in our case) and save the "'ecrobotnxt"' folder in it (which is downloaded with *Embedded Coder Robot NXT*). It's also needed to add the folder in which our models are saved ("'models"' in our case).

Later, the computer needs to have *Cygwin* with *GNU Make* installed. We used version 1.5.24, which can be downloaded from (vinschen12) . This software should also be installed to an immediate directory in the hard drive,

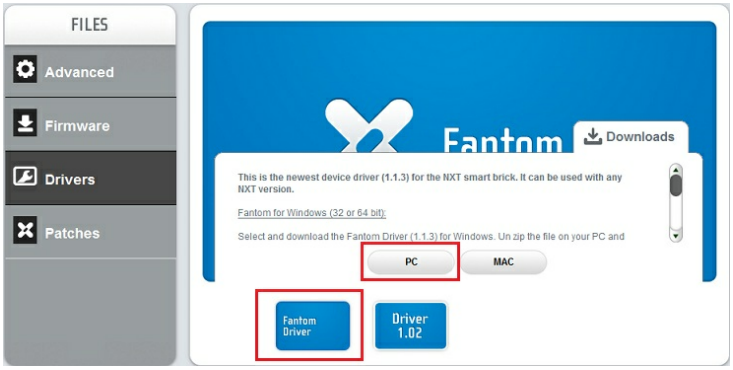
just like C:/cygwin . We must also be sure of selecting as "Default Text File Type" the "Unix/Binary"' option. Then, in the Devel tree node on screen, we must be sure to select "make 3-81-1" in order to have *GNU make*.

Then we need to install *GNU ARM*, for which we need to download GCC-4.0.2 or newer from (gnuarm) . The installation directory should be inside the *Cygwin* folder. The dollowing option need to be selected at the moment of installation:



The wizard should ask to install *Cygwin* DLLs, but this option should be unchecked since they will be already installed with the previous step. At the end of installation, it is asked to add the installation folder to a Windows Environment Path, but this is not needed.

The next step is to install the *Lego Standard USB Driver*, which can be downloaded from (fantom), and it's called *Fantom Driver*. It's important that before installing any Lego firmware, all ATMEL Sam-Ba software must be uninstalled from the PC.



After, we need to install the enhanced *NXT firmware* and *NeXTTool*. So we download the first in (nxtfirmware) . After, we get *NeXTTool* at (nexttool), and we install it inside the *Cygwin* folder. Keep the *NXT firmware* inside the same folder.

A driver which can recognize the NXT connected to the USB port of the computer must also be included in the installation process. So we must download the latest version of *LibUSB* from (libusb). It's crucial to install this dll into a directory with no spaces or special characters in the path, so the best option is to install it in "C:/libUSB". We must connect the NXT with a USB cable to any USB port from the PC, and add it as a specific driver assigned to a peripheral in the LibUSB application.

After this, all is set up to install the entire Matlab environment by running a Matlab script. So we just go to Matlab and run the *ecrobotnxtsetup.m* script and follow the instructions of its wizard. The successful installation should give us the possiility of working freely with the model and simulations.

Once we have a satisfactory set of models and we want to load it to the NXT brick, it's first needed to set it to HW reset mode. This can be achieved by pressing the button in the left "shoulder" of the NXT for more than 4 seconds. Doing this successfully will start a ticking similar to a metronome, indicating that the NXT is now in HW reset mode.

We can now connect the NXT to the PC using the USB cable. We go to the *nxtway_gs_controller.mdl* file and we click in "Generate code and build the generated code". After the code is generated, we can upload it to the NXT using the "Download (SRAM)" button. The "Download (NXT Enhanced Firmware)" button is used when no firmware has already been uploaded to the NXT, although we didn't make use of this in our tests.

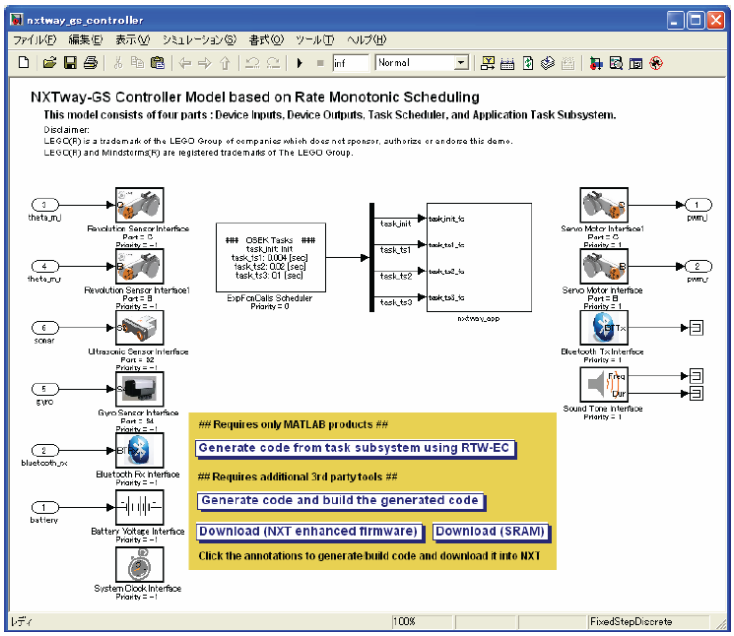


Figure 1.3: NXTway-GS Controller

Chapter 2

Simulink model, methodology

We used mathematical software Matlab and its extension for modeling - Simulink. Simulation of robot movement and all necessary parts were based on the Simulink model, which was already developed by someone else. Nonetheless, there were several changes. The most important was to implement reference path generator inside controller, so we could measure results for the same inputs in both simulation and real model. Another important change compared to original model was to create block for accumulating measurement data and sending them via bluetooth.

For measuring robot movement we chose three types of paths (three kinds of tests). In the first version, the trajectory just forced a robot to stay and balance on the same place. In the second version, we forced the robot to run in circles. Finally in the third version we measured response of impulse on robot's wanted velocity. It basically means to balance in-place for given time, then go forward and after given time delay start again to balance in-place.

For comparison of simulation results with real model of a robot we needed to extract data from both simulation model and real robot. To do that we made a few changes in the model. We added a few global variables and stored them into the workspace of Matlab for post-processing (i.e. creating output graph for comparison with real data). These global variables are also used for collecting real data from real model. Simply said, we kept track of input (resp. output) values from all the sensors (resp. actuators).

Measured variables:

- Revolution sensor in left motor $\dot{\theta}_{m_l}$
- Revolution sensor in right motor $\dot{\theta}_{m_r}$
- Gyroscopic sensor (calibrated) $\dot{\psi}$

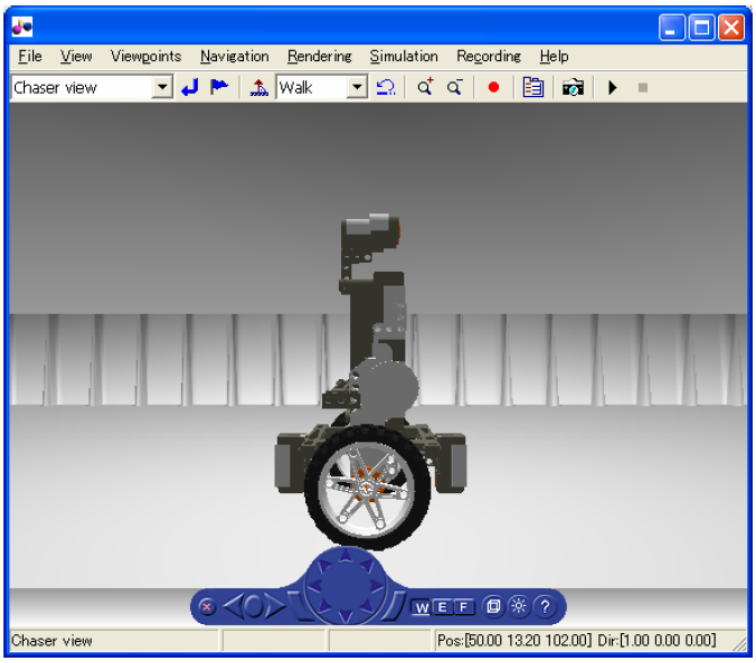


Figure 2.1: Simulation 3D visualisation view

- PWM duty cycle for left and right motor
- Wanted translational velocity $\dot{\theta}$
- Wanted rotational velocity $\dot{\phi}$
- Body angle (already calculated by robot MPU)
- CPU time register (for time scaling purposes)

2.1 Creating reference path

As outlined above, we had to design reference path generators. This is one of the most important blocks, which are used in the model. The Reference generator simulates a reception of commands over Bluetooth. Basically, it is an input for robot's balance & drive control regulator. In original model, the robot was supposed to be controlled via Bluetooth from a gamepad connected to the computer. We do not need, nor want, a real remote control, it would bring human-factor errors to the mneasurements. We just needed to force the robot to do specific movement in specific time, so we can compare measurement between simulation and real robot.

As you can see in figure below, the reference generator have two inputs. More specifically it has two changable parameters. The first one - in figure signed as rdot - represents forward or backward movement coeficient. If it is set to plus one, it is maximum forward velocity. On the other hand if it is minus one, it is maximum backward velocity. The second parameter of the generator - in figure signed as thetadot - represent rotation coeficient. As it was with previous parameter, the same hold with thetadot - value plus one represents maximum rotation velocity in positive sense, minus one is analogically the same in negative sense.

Please note that thetadot is actually parameter $\dot{\phi}$ and rdot is $\dot{\theta}$. There is no hidden sense in this. This is a mistake which was done by our predecessor, and we have decided to keep it. We haven't changed reference generator block, we have only used different input values rdot and thetadot for different tests.

As was said before the first type of trajectory was just simple standing and balancing on the place (we call it in following text as stationary balancing). To achieve this, we just simply put zeros for both parameters - zero velocities, angular and translational.

Figure 3.2 below shows whole reference generator block with gain coeficients and simulated packing for bluetooth (the robot controller expects a 32-byte packet coming from bluetooth).

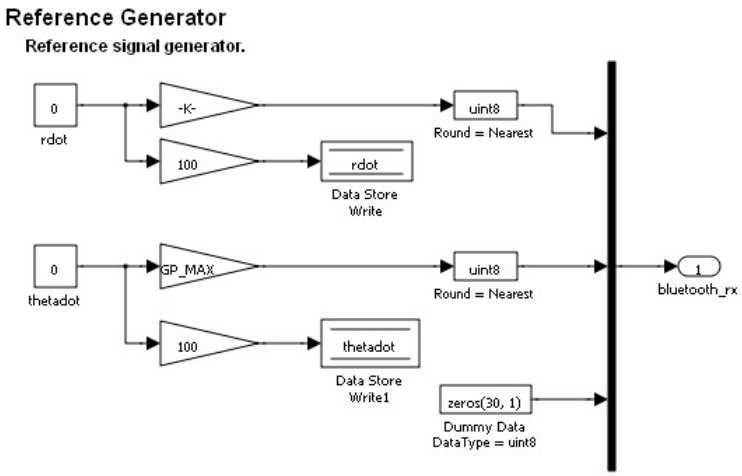


Figure 2.2: Reference path generator for stationary balancing

In the second version, when the trajectory was a circle, the parameters of reference generator are, again, constant. We have set translational velocity to 0.6. which is experimentally tested maximum safe speed and rotational velocity to 0.1. This setting caused the robot to make cca 80 centimeters wide circle per each 17 seconds.

Last version required time-varying translational velocity. We had to create an impulse - movement forward with stationary balancing before and after selected time window. For making this trajectory it was necessary to create a subsystem, which uses internal CPU time. Used ARM7TDMI processor has CPU time register, which is basically a counter of oscillator ticks after PLL multiplication. The ECRobot toolbox utilizes this as a Simulink block with time normalized to miliseconds (which is only useful considering that the CPU can run on different frequencies). We have used two constants - impulse start time and period in miliseconds. The impulse generator subsystem is depicted in figure 3.3. It replaces constant block rdot used in the reference generator from figure 3.2

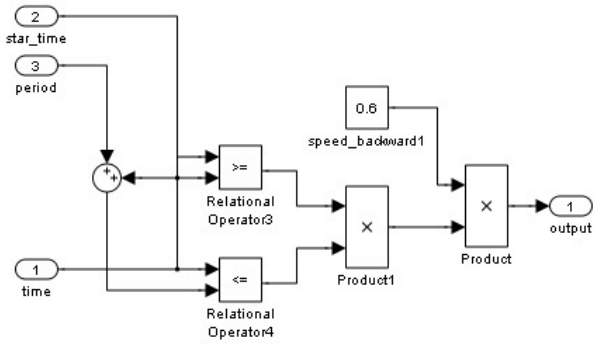


Figure 2.3: Impulse generation subsystem

It is made simply from two condition blocks and two multiplication blocks. The condition blocks are used for time window detection. Internal time has to be greater than starting time and also it has to be smaller than sum of starting time and period. Output product of this subsystem is coeficient 0.6 for forward movement in given interval, otherwise 0. As was said before, 0.6 is maximum feasible velocity.

2.2 Data acquisition

Another important change in the model was data acquisition - we needed to add a process to save current sensor and actuator data in given time steps. We have used different approaches for simulation and for real measurement. In the simulation, we have stored the values with period 4 miliseconds (period of balance & drive control task). The real measurements were sent via Bluetooth with period of 20 miliseconds. How the data were collected is described in following subchapters.

2.2.1 Simulated data generation

As was stated before, we changed the model to store current values of selected variables into standalone global variables. Those variables were saved to workspace using "export to workspace" block in standard Simulink block set. Look at figure 3.4 to see how exactly are data exported from model to workspace.

The task which did the logging was executed every 4 miliseconds, it makes about 250 measurements per second. Various variables are tracked, for details see the list at the beginning of this chapter. After the simulation, we saved results into extra file by issuing following command

```
save('simulation.mat')
```

The command stores whole workspace into a binary file simulation.mat. This file was later used for data processing (see next section).

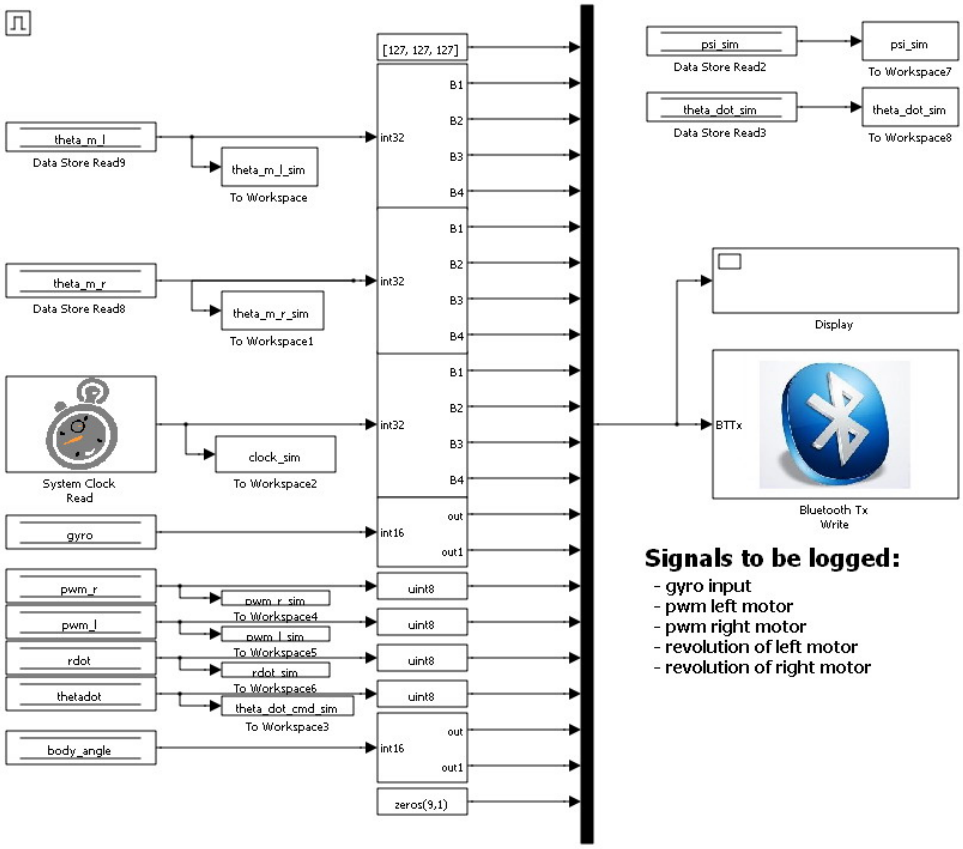


Figure 2.4: New Simulink Block - over-air data logger

2.2.2 Real data generation

We have considered two methods for real-time data collection. The first one was counting on locally saved data, to be downloaded from NXT Brick *a posteriori*. Second approach counted on other (remote) device who collects the measurements, so the robot just needs to send current data with some reasonable period. We have chosen latter. It is simpler and lighter on NXT Brick memory. If we would use the first approach, we would need to figure out how to save complex datatypes into SRAM or EEPROM memory. Also we would need a JTAG programmer to download the test result (as far as authors know, SAM-BA bootloader doesn't support SRAM/EEPROM download). Second approach requires only packing of current data in data-packets. Packed data were sent over-air using Bluetooth module embedded inside the NXT brick. On the other end of communication channel was a computer, storing all received data into a file.

We have added into our simulink model a block depicted at figure 3.4. On the left, all tracked variables are read. This collection corresponds to the list given in the beginning of this chapter. Then the data goes into conversion blocks. They are the tricky part. The multiplexer creates an array of 32 bytes, or, in Simulink terminology, vector of 32 fields of uint8. Now, we have variables of uint16, int32, int8 and uint8 datatypes. All non uint8 datatypes needs to be converted to a set of uint8's (bytes). For conversion from signed to unsigned datatype we have just utilize standard conversion blocks. For example int8 is converted to uint8 simply with conversion block.

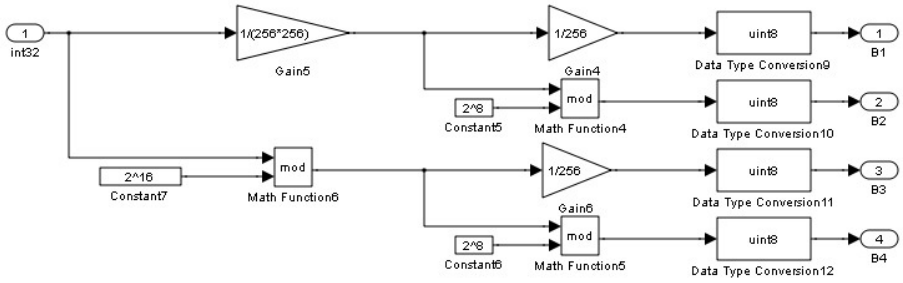


Figure 2.5: Spliting a 32-bit integer into its four inner databytes

Long datatypes (uint16, uint32) had to be splitted to its inner databytes. Obviously, uint16 consist of higher and lower value of uint8 datatype. The higher byte can be found by dividing the value by 2^8. Lower byte is retrieved by calculating modulo of 2^8. If we want to convert even bigger datatypes (in our case uint32), we can cascade process of dividing the value into higher and lower halves. Simulink submodel used for converting int32 into four

databytes is shown above on figure 3.5

It is fair to say, that we had run into problems with exportation of this subsytem into C (by using RTW-EC). In simulation, block on figure 3.5 was working perfectly, in reality, the highest databyte is always 0x00, even though it should be at least 0x80 for all negative numbers. We haven't found solution to this problem, but bypassed it by ignoring highest byte. When processing the data, we consider only lower 24 bits, which is more than enough for our purposes.

2.3 Data processing

This chapter describes methods used to collect, process and visualise measured data. It doesn't give any comments on the results itself (for that see next chapter), it just describes methodology - how we came to our results.

2.3.1 Bluetooth datapacket format

We have designed extremely simple format while keeping some nice properties of the information source. First of all, we were afraid that data will be partially scrambled and that there will be some parts missing, which would mess up byte orderings. It is very unprobable that any kind of transmission error will happen over air, because data are heavily guarded with CRC checksums (for more information see freely available Bluetooth stack description). Nonetheless, it can happen between MPU and Bluecore module on the NXT Brick printed circuit board. There are two wires of UART bus and data are not guarded. Also, two motors (EMF sources) are close to NXT brick.

The protocol is as follows. Each packet starts with a preambule {`0x7F, 0x7F, 0x7F`}, followed by all the variables, packed as shown on the figure 3.4. The preambule allows us to detect beginning of the next datapacket. It is very unprobable that we get somewhere else 3 bytes with same values. It comes from principle of two's complement, from ordering of variables in the datapacket and from error probabilities.

With the preambule, we can always lock to the beginning of next datapacket and as a consequence, we don't need to care about proper byte ordering. If there are some missing/added data, they will be ignored. If unfortunate sequence same as preambule appear in the data, we will se that on the result as an outlier. Nonetheless, we haven't seen a single mistake of this kind in our processed data.

2.3.2 Real data collection

Process of collecting real data is described only to help our followers to setup what is necessarry. The robot has to pose as Bluetooth slave. That means in the context of Bluetooth stack that the robot do not start connection. That is in contrast to many other network standards who describe slave as the node which cannot send data without being asked to. This is not the case, Bluetooth slave can send data to master freely.

Before we start the robot, we have to connect to it via Bluetooth. This is usually done on the NXT's operating system level, before the program is started. The matlab model comes with NXT OSEK operating system, which is based on OSEK OS. We have used a Linux box utilizing Xubuntu 12.04 with Bluez protocol stack and Blueman Bluetooth manager for data collection.

The NXT Brick support SPP Bluetooth profile (Serial Port Protocol), which works like a virtual serial port between the robot and the computer. The connection is, of course, tunelled via Bluetooth. Virtually any Bluetooth device support this kind of functionality, even the SPP profile is not available. A RFCOMM protocol is a low level protocol in the Bluetooth protocol stack which utilizes at most about 50 virtual serial links between all the Bluetooth nodes around. Support for SPP profile merely says that we can control directly a serial port created by RFCOMM protocol.

Blueman application on the Linux box offers a GUI to start connection and to create a serial port accessible from `/dev/rfcomm0`. To store all the data from the robot, just issue following command:

```
sudo cat /dev/rfcomm0 > datalog
```

The `sudo` makes the program cat running with superuser rights. If you are using another linux box, you can use command `su` before calling cat. The result is written into file named `datalog` in current folder. See following listing for the hexadecimal dump of a typical data log.

```
matej@matej-M50Vc:~$ hd /home/matej/datalog -s 2000 -n 112
000007d0  00 00 00 00 00 00 20 00  7f 7f 7f 00 00 01 47 00
000007e0  00 01 22 00 01 42 65 02  11 3b 40 3c 0a 7e b8 00
000007f0  00 00 00 00 00 00 00 00  20 00 69 42 01 00 45 40
00000800  35 1d 00 00 00 00 24 01  00 00 49 01 00 00 2e f3
00000810  00 00 00 00 00 00 ff ff  ff ff 20 00 7f 7f 7f 00
00000820  00 01 50 00 00 01 2a 00  01 42 79 02 00 33 38 3c
00000830  0a 7e d0 00 00 00 00 00  00 00 00 00 20 00 7f 7f
```

2.3.3 Loading a binary log file into matlab

A Matlab function read_bt was designed to load the data from the log file. It takes as an argument a path to the file, then loads the binary content as one large vector A. The algorithm goes through the vector A and stores a new record always when it detects an preambule sequence.

There are few things to notice. System time, which is part of each data sample, is recalculated to 0 when program starts. The databytes of variables which utilize more than one 8 bits are decoded into a single (original) value. Gyroscopic measurement is subtracted with constant `512`, because the result is a 10-bit value from ADC, where `0` equals to $-512\,^\circ/s$, `512` to 0/s and `1023` to $511\,^\circ/s$ (nonetheless the fact that it's dynamical range is $\pm 300\,^\circ/s$).

```
1  % Function READ_BT
2  % \input  realdata Path to a binary file with log of real
3  %                  robot measurements
4  % \output theta_ml Vector with the samples of revolution
5  %                  sensor on left motor
6  % \output theta_mr Vector with the samples of revolution
7  %                  sensor on right motor
8  % \output clock    Time axis reference
9  % \output gyro     Vector with gyroscopic measurements
10 % \output psi      Vector with calculated body angle
11 %                  propagation in time
12 function [theta_ml, theta_mr, clock, gyro, psi]=read_bt(path)
13
14   % Load the file into matrix A
15   A = fread(fopen(realdata));
16
17   % Reset all the variables
18   theta_ml = [];
19   theta_mr = [];
20   clock = [];
21   gyro = [];
22   psi = [];
23   pwm_l = [];
24   pwm_r = [];
25   r_dot = [];
26   theta_dot_cmd = [];
27   base_clock = 0;
28
29   % For all the databytes
30   for i=1:size(A)-32
31     % Detect the preambule
32     if((A(i)==127)&&(A(i+1)==127)&&(A(i+2)==127))
33
```

```
34        % load current clock
35        clock_curr =
36         (2^24).*A(i+11)+(2^16).*A(i+12)+(2^8).*A(i+13)+A(i+14);
37
38        if((base_clock==0) && (clock_curr>0))
39          % Setup clock offset
40          base_clock = clock_curr;
41        end
42
43        if(base_clock>0)
44          % Store the clock, with proper offset
45          clock = [clock, (clock_curr-base_clock)/1000];
46
47          % Decode & store theta_ml (24-bit signed number)
48          val = (2^16)*A(i+4) + (2^8)*A(i+5) + A(i+6);
49          if(A(i+4)>127)
50            valinv = bitxor(val, 2^24-1);
51            theta_ml = [theta_ml,-1*(valinv-1)];
52          else
53            theta_ml = [theta_ml,val];
54          end
55
56          % Decode & store theta_mr (24-bit signed number)
57          val = (2^16)*A(i+8) + (2^8)*A(i+9) + A(i+10);
58          if(A(i+8)>127) %treat as negative
59            valinv = bitxor(val, 2^24-1);
60            theta_mr = [theta_mr,-1*(valinv-1)];
61          else % treat as positive
62            theta_mr = [theta_mr,val];
63          end
64
65          % Store the gyroscopic measurement, set zero at 512
66          gyro = [gyro,(2^8).*A(i+15)+A(i+16)-512];
67
68          % Store body angle with precision of 0.01
69          psi = [psi,((2^8).*A(i+21)+A(i+22)-2^15)/100];
70
71          % NOTE: pwm_l, pwm_r, thetadot_cmd, rdot_cmd unused
72        end
73      end
74    end
75  end
```

2.3.4 Processing & visualising the results

We have developped another Matlab function called `process_measurement` for processing and visualising the results. This function loads simulation and real data, then crops them so only a selected time window is processed (there can be an offset between time window for real and for simulation, which helps to synchronize the data)

For visualisation, raw measurements needs to be converted its representatives. First of all, an average `theta` is calculated as average of `theta_ml` and `theta_mr` variables. `theta` is then used to calculate travel distance and a velocity. `distance` is calculated as `theta` multiplied by a constant given by travel distance when a wheel turns by one degree. `velocity` is found by calculating derivative (difference) of `distance`. Simulation data are processed analogically. They are usually identified by postfix `_sim`.

```
1  function process_measurement(real, sim, start, stop, offset)
2
3    % setup constants
4    wheel_diameter = 5.6; % [cm]
5
6    % collect the data
7    [theta_ml, theta_mr, clock, gyro, psi]=read_bt(real);
8    load(sim);
9
10   % crop the real data
11   [~, start_pos] = min(abs(clock-start));
12   [~, end_pos] = min(abs(clock-stop));
13   clock=clock(start_pos:end_pos);
14   theta_ml=theta_ml(start_pos:end_pos);
15   theta_mr=theta_mr(start_pos:end_pos);
16   gyro=gyro(start_pos:end_pos);
17   psi=psi(start_pos:end_pos);
18
19   % crop the sim data with some offset
20   clock_sim = (double(clock_sim)/1000)+offset;
21   [~, start_pos] = min(abs(clock_sim-start));
22   [~, end_pos] = min(abs(clock_sim-stop));
23
24   clock_sim=clock_sim(start_pos:end_pos);
25   theta_m_l_sim=double(theta_m_l_sim(start_pos:end_pos));
26   theta_m_r_sim=double(theta_m_r_sim(start_pos:end_pos));
27   psi_sim_f=psi_sim_f(start_pos:end_pos);
28
29   % calculate average theta and filter out quantization
30   % noise of the encoder
31   theta_ = (theta_ml + theta_mr)/2;
```

```
32    theta_sim = (theta_m_l_sim + theta_m_r_sim)/2;
33    theta = filter(ones(1,2)/2,1,theta_')';
34
35    % approximate theta difference to get real theta_dot
36    % (units are degrees per sec)
37    theta_dot_real = [diff(theta),0];
38
39    % calculate 1-D distance
40    distance = (theta./360).*(pi*wheel_diameter);
41    distance_sim = ((theta_sim./360).*(pi*wheel_diameter))';
42
43    % calculate velocity in cm/s
44    velocity = [diff(distance),0];
45    velocity_sim_ = [diff(distance_sim),0];
46    velocity_sim = filter(ones(1,4)/4,1,velocity_sim_')';
47
48    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
49    % CALCULATE 2-D DISTANCE MAP
50
51    % get theta derivatives on left and right motor
52    theta_dot_ml = [diff(theta_ml), 0];
53    theta_dot_mr = [diff(theta_mr), 0];
54
55    % recalculate in terms of distance
56    theta_dot_ml_cm = (theta_dot_ml/360).*(pi*wheel_diameter);
57    theta_dot_mr_cm = (theta_dot_mr/360).*(pi*wheel_diameter);
58
59    dcm = theta_dot_ml_cm-theta_dot_mr_cm;
60    phi_dot_real = asin(dcm / L);
61
62    phi_real = cumsum(phi_dot_real);
63    theta_dot_real_cm=(theta_dot_real/360).*(pi*wheel_diameter);
64
65    x_real = cumsum(theta_dot_real_cm.*cos(phi_real));
66    y_real = cumsum(theta_dot_real_cm.*sin(phi_real));
67
68    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
69    % PRINT THE FIGURES
70    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
71
72    close all;
73    set(0,'DefaultFigureWindowStyle','docked');
74
75    % Distance 1-D and the velocity
76    figure; hold on;
77    subplot(2,1,1);
78    plot(clock,distance,clock_sim,distance_sim,'LineWidth',2);
79    grid on;
80    subplot(2,1,2);
```

```
81    plot(clock,velocity,clock_sim,velocity_sim,'LineWidth',2);
82    grid on;
83
84    % Gyro and the psi
85    figure; hold on;
86    subplot(2,1,1);
87    p=plot(clock, psi, clock_sim, psi_sim_f, 'LineWidth', 2);
88    grid on;
89    subplot(2,1,2);
90    plot(clock, gyro, 'LineWidth', 2);
91    title('Gyro and psi');
92    grid on;
93  end
```

The position map was calculated as follows. We have used measurement of θ_l and θ_r in time. First, we needed to calculate $\dot{\theta}$ and $\dot{\phi}$, which is change of position in polar coordinates, then we have transformed $(\dot{\theta}, \dot{\phi})$ to cartesian coordinate system.

$$\dot{\theta} = \frac{\Delta \frac{(\theta_l + \theta_r)}{2}}{\Delta t} \;\; ; \;\; \dot{\phi} = \frac{\Delta(\theta_l - \theta_r)}{\Delta t}$$

$$dx = \dot{\theta}_t \cos \dot{\phi}_t$$

$$dy = \dot{\theta}_t \sin \dot{\phi}_t$$

This gave us two equations for two differences (dx, dy). Finally, we have summed all the samples of both dx, dy over time. As a result we get a single point on a position map in chosen time T.

$$x = \sum_{t=1}^{T} dx = \sum_{t=1}^{T} \dot{\theta}_t \cos \dot{\phi}_t$$

$$y = \sum_{t=1}^{T} dy = \sum_{t=1}^{T} \dot{\theta}_t \sin \dot{\phi}_t$$

Please notice, that even with our best effort, we were not able to find hardware specification of used encoder. As a consequence, we do not know how exact the reconstruction is (encoder introduces quantization error, which we cannot estimate, to make some conclusions regarding map reliability). Nonetheless, the results seem to be smooth enough and the position map correspond to our observation.

2.4 Other changes in the NXTway-GS project

There are many changes we did to the NXTway-GS model. They are mostly about removing some unwanted additional features. We have replaced the task for obstacle avoidance completely with datalogger, by which the sonar on the robot became dormant, unused. Also, we have removed various modes like “autonomous mode”, “remote control mode”, we do not need them. At last, support for control over gamepad was completely removed too.

Chapter 3

Results, comments

In this chapter, we present our results and give a number of comments regarding them. We will point out several "weak points", discuss possible causes and offer solutions.

3.1 Stationary balancing

In this test, the robot was supposed to balance in-place. With ideal conditions, he wouldn't move at all. In practice, the robot oscillates back and forth to keep yourself in upright position.

Used regulator doesn't really try to keep robot in place, it just tries to keep angular and translational velocity at zero. This is not the same. For example, imagine situation when the robot starts with biased body angle - he will be able to cancel this error, but he will have to change position to achieve that. The regulator which would minimize planar distance between starting and current position was not used, although it is possible to design such regulation (encoders inside actuators give absolute distance between starting and current position).
In real life conditions we have to deal with several issues - motors have limited torque, batteries cannot give enough current for motors to enforce fast startup times, sensor measurements are noisy, motors don't have same construction, tires are too soft, and so on..

As a result, when two-wheeled robot wants to stay at upright position, he needs to oscillate around it. The oscillation amplitude strongly depends on used regulator, used model, and so on. Additionally, the move back and forth introduces positional offsets in both X and Y directions (considering we are balancing on a flat plane).

On a figure 4.1 you can see both simulated and real results. The oscilation

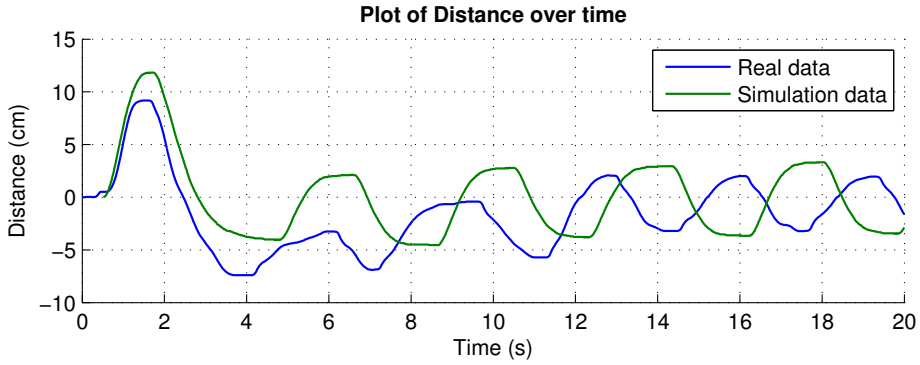


Figure 3.1: Stationary balancing - distance

amplitude is actually somewhat smaller in real measurement than in simulation. It is important to realize that both simulation and real measurement use the same model, which is simple and unprecise. The LQR regulation is optimal around (linearized) unstable equilibrium, but only under assumption that the model is equal to real machine (which is not, not exactly). Best results obtained (on the figure) are with oscillations around 45 milimeters. We couldn't lower this amplitude anymore with used model. There is lower bound for those oscillations given by physical limits of motor, battery and motor control logic. It is probable that with better model we would be able to lower this amplitude.

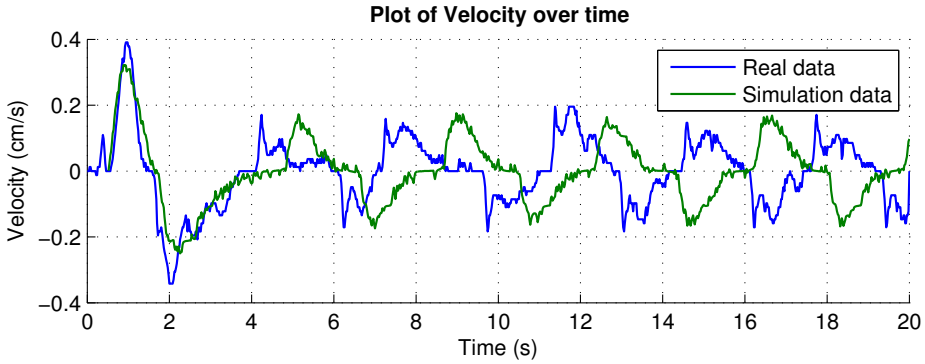


Figure 3.2: Stationary balancing - velocity

In following figure 4.2 we can see that the regulator reacts to threatening body angle by an acceleration peak against the falling. We can see similiarities between the model and the real machine - even the period is different, the curves (for both velocity and distance) are similar.
Figure 4.3 shows real, somewhat noisy, gyroscopic measurement. Body angle

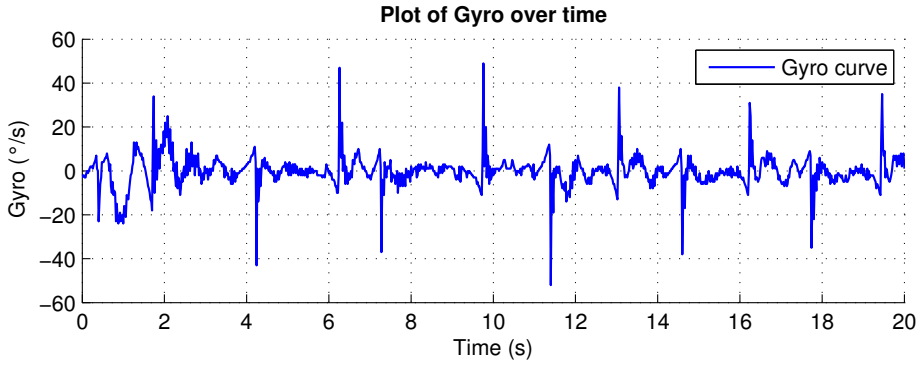


Figure 3.3: Stationary balancing - gyroscope

at figure 4.4 is calculated by cumulative integration of gyroscope measurement. That is only natural, gyroscope gives speed of rotation, which makes body angle its integrand. From the body angle curve we read somewhat periodical behavior with alternating upper and lower extremas.

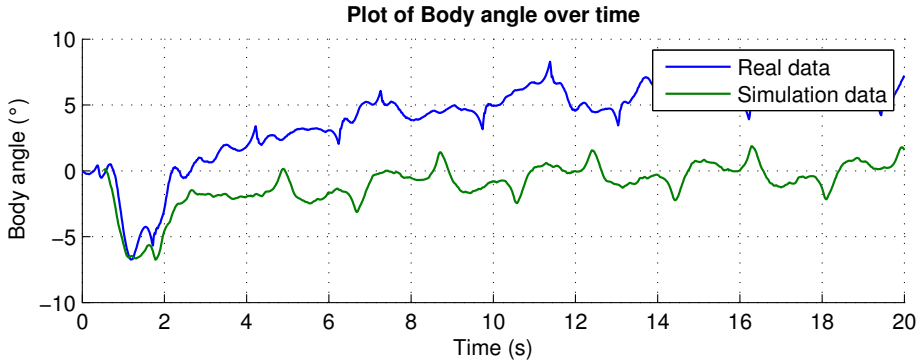


Figure 3.4: Stationary balancing - body angle

There is one important difference between measured and simulated body angle. Measured one is biased by approximately 5 degrees. There can be several reasons for that, but the most probable one is in wrong calibration during startup [12]. Since the gyro is calibrated when we hold the robot in "somewhat upright" position with our own hands, we can easily make a 5 degree mistake. The integrand of measurements then contain biased information. If we assume that the body angle offset is a matter of sensor calibration, not of actual robot body angle bias, then this error doesn't really exists.

Nonetheless, there is other suitable explanation. The robot measures actual body angle, but the model assumes that this is an angle of a point in

center of the robot mass. Since the robot is not balanced, this particular point has some angle offset from the body angle. Hence, the regulator compensates for unbalanced robot body by adjusting body angle.

Finally, there is position map at figure 4.5, where you can see positioning error. As was stated before, the robot oscilates back and forth, which introduces errors in both X and Y directions. Thanks to encoder sensors inside the motors we were able to reconstruct robot path on a 2D plane.

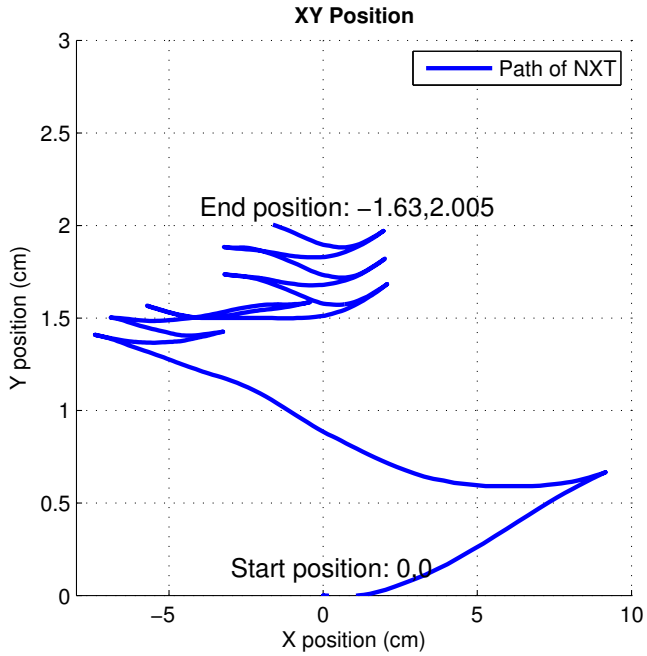


Figure 3.5: Stationary balancing - position offsets in time

The map shows the position offsets propagated in time. You can see, that in the start there is one big swing. This is caused by clumsy operators who did the test - we have had started the robot with biased body angle. As a result, the regulator automatically corrected this error and the swing is its by-product. When the robot finally stabilizes at [-6, 1.5] centimeters from the starting point, it starts the stationary balancing to keep the upright position. Please note that this is what the robot was supposed to do from the beginning.

During the stationary balancing, the robot goes slowly sideways (around 3 centimeters per minute). We can see that on the position map. The question is why - we are not clear on the reason, but we see several possibilities.

We can say for sure that left side of the robot is somewhat slower than the right side. This shouldn't happen, because the simulink model uses a

proportional regulator to compensate differences of the two motors revolution during forward/backward movement (the regulator is turned off when wanted rotational speed is not zero).

Let's assume, for the moment, that the P regulator is not present. Then, generally speaking, there can be two kinds of problems - left motor rotates on the same velocity as the right one, but the tire slips, or the left motor is just slower for the same PWM duty cycle. Please note that in stationary balancing, there is no reason for the regulator to go sideways - the PWM duty cycle is the same for both left and right motor.

The first type of problem (a slipping tire) is possible because of used tires, they are really soft. It is just a piece of rubber, with no soul. As a consequence, the robot body tends to incline to side in response to motor acceleration (the rubber sags). This can cause the tire to slip and in consequence the robot goes sideways. Generally speaking, the combination of motor torque and relatively large body weight doesn't allow the tires to slip without sag even on slippery surface.

The second (and more probable) option states that the problem is somewhere in the left motor. First of all, the left motor have almost definitely slightly different parameters then the right one. Most probably, the inner friction and moment of inertia of left motor with its gearbox is higher than on the other motor. As a consequence the motor gives lower torque.

Third, and also probable option, is that the PWM amplifier ("H" bridge with 4 unipolar transistors) have larger resistance between source and drain, which cause voltage losses in the control logic. Then there is lower voltage for the motor and hence lower velocity around the axle.

There are inidices that the problem lies in control logic though. The curve of stationary balancing seems like the motor is weaker when going forward, than in the other direction. Both inner friction and moment of inertia cannot cause that. The "H" bridge, on the other hand, can cause this, because different pair of transistors is used to go forward and different pair for going backwards.

We believe that the problem is combination of last two causes. Inner friction and moments of inertia are never the same for two motors. The same holds for two transistors put together. To sum up, those inconsistencies probably cause the robot to go sideways. The problem is partially compensated by the P controller, but it is well known that this kind of controller introduces step error. Using PI controller could help to improve the drifts.

3.2 Impulse response

On the figure 4.6 we can see the scenario of this test case. At first, the robot is doing now well known stationary balancing. The change comes 30 seconds after start when the robot starts his journey forward. He keeps doing it for 7 seconds and then stops again. In other words, the wanted velocity is zero from time 0 to 30 seconds and 37 to ∞ seconds. Between 30 to 37 seconds, the wanted velocity is maximum feasible. This forms an impulse in the regulator input.

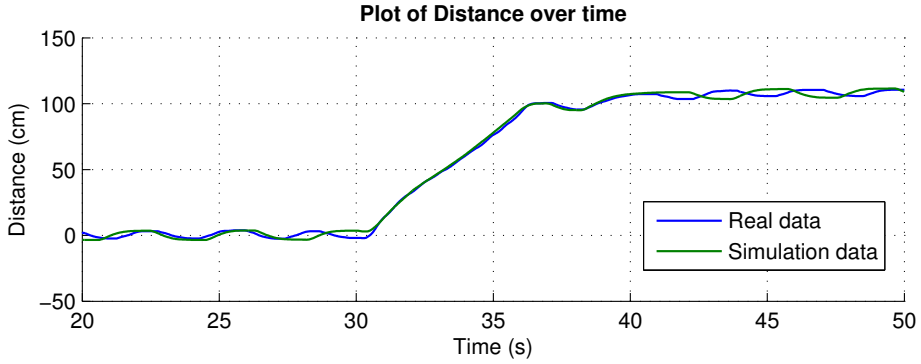


Figure 3.6: Impulse response - distance from starting point

The slope of the impulse was not adjusted, it goes from zero to maximum velocity immediately. On figure 4.7 we can see the response. The regulator reacts in both simulated and real data with quickly rising edge, which causes overshoot at first (specially in real data). Regulator then oscilates around velocity of 20 cm/s.

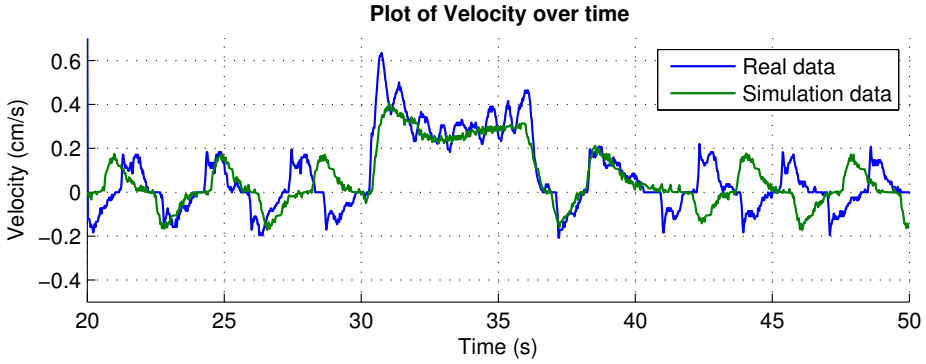


Figure 3.7: Impulse response - velocity

It is not hard to notice that real robot tends to oscillate with quite large amplitude around its mean velocity. It seem like combined efffect of soft tires and of used regulator properties. Soft tires probably cause large overshoot in the begining, because when the motor turns forward with significant torque, both tires sag. Then, the large amplitude oscillation is caused by regulator which tries to achieve two things at once - it wants to go forward, while keeping the robot in upright position. The problem is, that for going forward, the robot needs to incline a bit. As a consequence, those two effects fight each other and cause this oscillation.

Solution for this problem is simple, though. We need to use regulator which will take on input wanted body inclination. When we do this, the robot then move forward and backward without oscillation as a consequence of tracking wanted (non-zero) body angle. This solution is actually quite elegant, but then we dont use wanted velocity as input, but wanted body angle, which is, in a sense, nonsense. We do not know which angle will cause the robot to move with wanted velocity, nor we do not know what is the maximum feasible inclination So, we need at least some more calculation to estimate the given angle for given velocity. (but this robot will not operate on inclined surfaces). For really well working solution, we would need another (wrapping) regulator with velocity feedback.

Figure 3.8: Impulse response - Body angle

Next figure 4.8 depict body angle in real and simulated measurement. The two states (stationary balancing versus movement forward) are clearly separated. The body angle suffers with the same problem with oscillations as the velocity. The reasons are already described in the paragraph above. Second problem is, that real body angle measurement is biased. The reasons are the same as for stationary balancing. Hence, more information on this can be found in previous chapter.

Figure 4.9 show the real gyroscope measurement. We can see that when

Figure 3.9: Impulse response - gyroscopic measurement

real robot moves forward, gyroscope measurement is somewhat smoother, there are no quick, large peaks to both directions. This is because body is already inclined at some degree.

3.3 Running in circle

The last testcase makes the robot running in a meter wide circle. We chose to design this test to measure accuracy of the robot directioning - in ideal conditions, the robot would create exact circle.

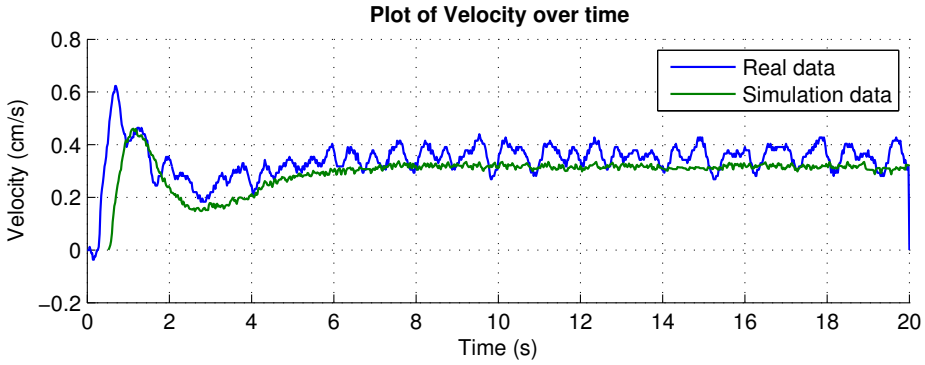


Figure 3.10: Running in circle - velocity

We want to measure the deviation between the ends of full circle. The circle is created simply by giving positive constant on the regulator inputs $\dot{\theta}$ (translational velocity) and $\dot{\phi}$ (rotational velocity). The constants were chosen to make approximately one meter wide circle (it is not very important for our test, though). The travelling distance is then around 3 meters.

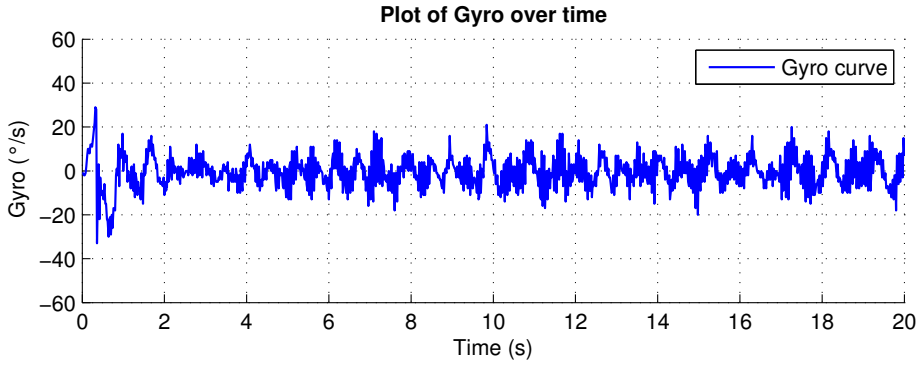


Figure 3.11: Running in circle - gyroscopic measurement

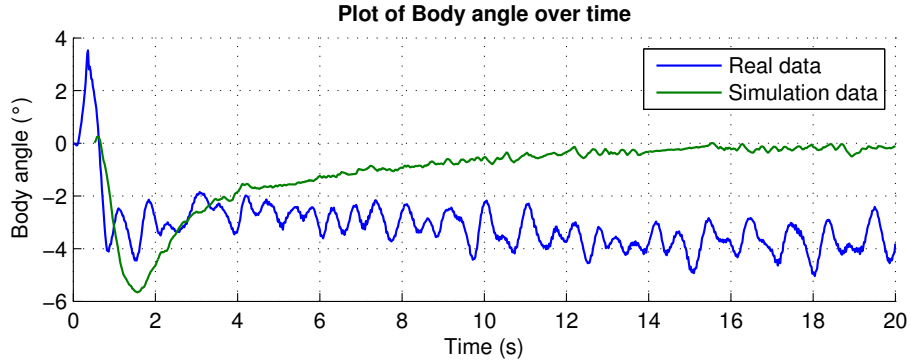


Figure 3.12: Running in circle - body angle

First three figures 4.10 4.12 4.11 show the measured velocity, body angle and gyro output. There is one interesting thing to notice in all three figures - all the curves exhibit some kind of oscillations. We could see with bare eyes why, when commencing the test - the robot tend to swing from side to side. This was a consequence of soft tire with no soul. When running in a circle, more weight was put on outer wheel. That in combination with velocity/inclination oscillations which the robot exhibited already during the impulse response test caused the robot to sway in all four directions - forward, backward, left and right.

That is why we were surprised by the results. The swing from side to side didn't affected the results seriously. They are more precise than expected. When we look at figure 4.13 we can see that the robot kept the circle quite precisely. There is deviation in the beginning, where the robot compensated for start in inclined starting position (we could see that effect in all three tests - it's a consequence of starting in unprecise upright position). After making a full circle, the robot was only a few centimeters on the left from its

original trajectory.

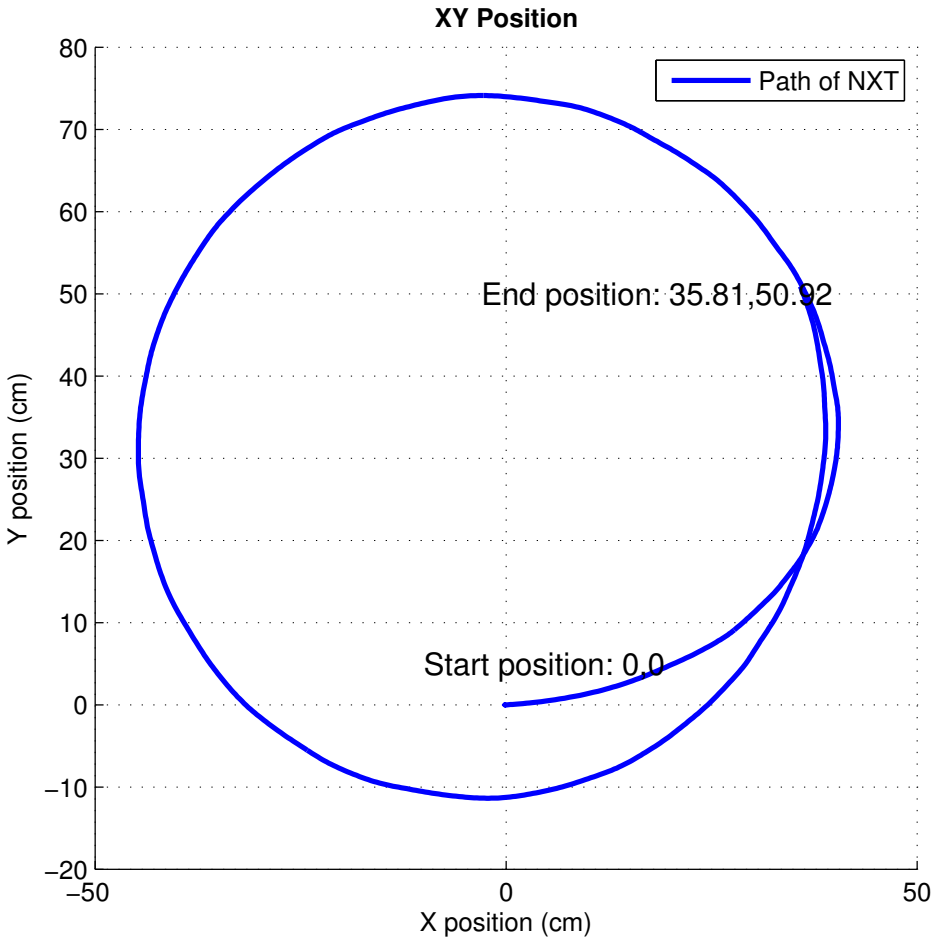


Figure 3.13: Running in circle - position

Bibliography

[1] Yorihisa Yamamoto *"NXTway-GS Model-Based Design - Control of self-balancing two-wheeled robot built with LEGO Mindstorms"*. Applied Systems First Division, Cybernet Systems Co., LTD, Revision 1.4, May 2009

[2] Ryo Watanabe *"Lego Mindstorms NXT"*. Wasedda University http://web.mac.com/ryo_watanabe/iWeb/Ryo's%20Holiday/NXTway-G_files/nxtway-g-1.pdf

[3] Takashi Chikamasa *"Embeded Coder Robot NXT Instruction Manual"*. Revision 1.2, June 2008

[4] nxtOSEK/JSP *"nxtOSEK Installation in Windows XP/Vista/7"*. http://lejos-osek.sourceforge.net/installation_windows.htm

[5] GNU ARM *"GNU ARM toolchain for Cygwin, Linux and MacOS"*. http://www.gnuarm.com/

[6] Takashi Chikamasa *"Embedded Coder Robot NXT Demo"*. Matlab Central, March 1st 2012, http://www.mathworks.com/matlabcentral/fileexchange/13399

[7] Corina Vinschen *"Updated: Cygwin 1.7.15"*. Cygwin, May 10th 2012, http://cygwin.com/ml/cygwin-announce/2012-05/msg00019.html

[8] Lego Mindstorms *"Drivers"*. http://mindstorms.lego.com/Support/Updates/

[9] Bricx Command Center 3.3 *"NXT firmware"*. http://bricxcc.sourceforge.net/lms_arm_jch.zip

[10] Bricx Command Center 3.3 *"Programmable Brick Utilities: NeXTTool"*. http://bricxcc.sourceforge.net/utilities.html

[11] libusb-win32 *"libusb-win32"*. Geeknet, 2012, http://sourceforge.net/apps/trac/libusb-win32/wiki

[12] Looney Mark (July 2010); *"A simple calibration for MEMS gyroscopes"*; Analog Devices. Retrieved on June 10, 2012.
(available online at http://www.analog.com/static/imported-files/tech_articles/GyroCalibration_EDN_EU_7_2010.pdf)