UPPSALA
UNIVERSITET

LegoWay: Fault Detection

Project Report

Dirk Stober

January 25, 2021

Contents

1 Project Description

The goal of the project is to develop a Lab for a new course: *Safety and Security in Control Systems*. The Lab should demonstrate safety and security principles exemplified by a self-balancing robot called LegoWay. This should be done by injecting different faults into the system, that could be caused both by cyber-attacks and hardware faults, and demonstrates ways of timely detection. The LegoWay is a two-wheeled model of a Segway with a motor and a rotation sensor at each wheel. In addition there is a gyroscopic sensor to measure angular velocity. The system is controlled by a model-based controller that reads the sensors and manipulates the actuators to keep the robot in a vertical position. The project work can be split into three main parts, first there is the problem to define a set of possible faults as well as finding approaches and eventually designing a framework to detect those faults. This includes understanding the current model and simulating the framework in MATLAB/SIMULINK and finally testing it on the LegoWay. The second part is the implementation of both the controller and the fault detection scheme on the LegoWay. Both the model of the System and an algorithm to generate the mathematical representation of the controller are already present. The controller and simulation framework were implemented more than a decade ago, so there is the need to evaluate the current implementation and possibly program them differently. The last part is to combine the previous two parts into a Lab that teaches the strategies to students. This part will have the lowest priority, as it heavily depends on the two other parts being completed.

2 LegoWay

2.1 Construction

The robot is build using LEGO MINDSTORMS NXT and a HiTechnic Gyro sensor following an online building instruction [1]. In this project I won't use the ultrasonic sensor nor the Gamepad. The LegoWay uses the wide thread as the constant $W = 0.08m$ used in the model describes half the width of the robot and not the full width. The left motor should be connected to port A of the Lego Brick and the right motor to port B. The gyroscopic sensor is connected to port 2.

2.2 Model

The original model [2] was derived using Euler/Lagrange equations. It consists of 3 differential equations for the x, ϕ and ψ coordinate, using the following constants:

g	$9.82\frac{m}{s^2}$	Gravitational acceleration constant
m_b	$0.58kg$	Body mass
m_w	$0.03kg$	Wheel mass
R	$0.04m$	Wheel radius
L	$0.09m$	Distance to center of gravity from wheel axle
W	$0.08m$	Half the body width
I_p	$7.12 * 10^{-4}kgm^2$	Body pitch inertia
I_j	$0.13 * 10^{-2}kgm^2$	Body jaw inertia
I_{xx}	$0.14 * 10^{-2}kgm^2$	Body inertia moment about x axis
I_w	$2.66 * 10^{-5}kgm^2$	Wheel inertia moment
J_m	$5.00 * 10^{-4}kgm^2$	DC motor rotor inertia moment
R_a	6.86Ω	DC motor resistance
K_b	$0.47V\frac{s}{rad}$	DC motor back EMF constant
K_t	$0.32\frac{Nm}{A}$	DC motor torque constant
B_m	$0.11 * 10^{-2}Nms$	Damping ratio of mechanical system

$x - coordinate$:

$$\begin{aligned}\left[m_b + 2m_w + 2\tfrac{I_w+J_m}{R^2}\right]\ddot{x} + \frac{2}{R^2}mc_1\dot{x} + (m_bLcos(\psi) - 2\frac{J_m}{R})\ddot{\psi} - \\ \frac{2}{R}mc_1\dot{\psi} - m_bLsin(\psi)\dot{\psi}^2 = \frac{mc_2}{R}(V_{right} + V_{left})\end{aligned} \tag{1}$$

$\phi - coordinate$:

$$\begin{aligned}\left[2(m_w + \tfrac{I_w+J_m}{R^2})W^2 + I_{xx}sin^2(\psi) + I_Jcos^2(\psi) + m_bL^2sin^2(\psi)\right]\ddot{\phi} + \\ 2\left[\left[m_bL^2 + I_{xx} - I_J\right]sin(\psi)cos(\psi)\dot{\psi} + \tfrac{W^2}{R^2}mc_1\right]\dot{\phi} \\ = \frac{W}{R}mc_2(V_{right} - V_{left})\end{aligned} \tag{2}$$

$\psi - coordinate$:

$$\begin{aligned}\left[m_b L^2 + I_p + 2J_m\right]\ddot{\psi} + \left[m_b L cos(\psi) - 2\tfrac{J_m}{R}\right]\ddot{x} + 2mc_1\dot{\psi} - \\ \frac{2}{R}mc_1\dot{x} - \left[m_b L^2 + I_{xx} - I_J\right] sin(\psi)cos(\psi)\dot{\phi}^2 - m_b g L sin(\psi) \\ = -mc_2(V_{right} + V_{left})\end{aligned} \tag{3}$$

with $mc_1 = B_m + \frac{K_t K_b}{R_a}$ and $mc_2 = \frac{K_t}{R_a}$
They can be merged using $q = \begin{pmatrix} x & \phi & \psi \end{pmatrix}$; $u = \begin{pmatrix} V_{left} & V_{right} \end{pmatrix}$ in the following equation:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \tau(u) \tag{4}$$

The non-linear model of the robot has been linearized around the equilibrium point $\psi = 0$. The liniearization of the non-linear model can be represented by a s state-space model:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{5}$$

$$y(t) = Cx(t) \tag{6}$$

with $x(t) = \begin{pmatrix} q \\ \dot{q} \end{pmatrix}$ and $y(t)$ the sensor outputs. The system can also represented in its discrete form, which will be used for fault detection:

$$x(k+1) = A_d x(k) + B_d u(k) \tag{7}$$

$$y(k) = Cx(k) \tag{8}$$

2.3 LQG-Controller

The controller can be generated using the function `lqg_servo.m`, which creates a state controller and a kalman filter based on a set of weighting matrices and returns them as one state space representation. I tried a couple of different weightings, but did not accomplish any improvements and thus resulted in using the default weighting matrices. The controller uses the linearized model and allows for the setting of a reference velocity $\dot{x}$ and a desired yaw angle ϕ. Due to the limited time the LegoWay was only used with both reference values being set to 0. The MATLAB model of the LQG controller was used to automatically generate RobotC code. This generator was not used in the project, as I decided to go with a different software Implementation, which will be discussed in a later section.

2.4 PWM controlled motors

One issue I stumbled upon was the difference between the motor control in the model and on the LegoWay. The original report describes the motor as being controlled by voltages between $-8V$ and $+8V$ while the hardware only allows for control using a PWM signal between -100 and $+100$. The first approach to solve

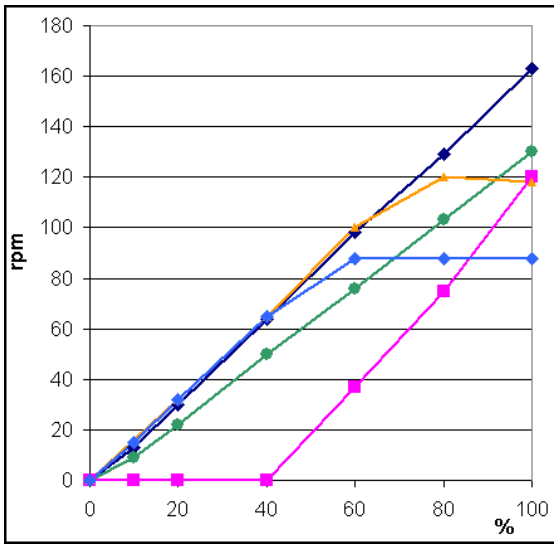

Figure 1: Relationship of PWM signal and RPM for voltage and power control. Pink shows the motor behaviour with no motor control and $11.5Ncm$ load applied.

this problem was to multiply the voltage level by $\frac{100}{8}$. This exposed a deadzone inside the PWM signals, where the wheels only started turning at around $|u| \geq 40$. The website [3] is a collection of different measurements on the nxt motor dynamics, which also includes a graph (Fig 1) plotting the relationship between the PWM value and RPM (11.5N.cm load) with and without power control. This also shows a deadzone when Power control is not used. I thus proceeded to scale the voltage signals by $\frac{60}{8}$ and compensate the deadzone using an offset. This procedure is not very precise and as discussed later, might have an influence on fault detectability.

3 Fault Detection

3.1 General Approach

Fault detection aims at finding faults or unusual behaviour of the system. This is usually done by using redundant information about the process. One option is to use additional hardware sensors to compare the different results with one another. An alarm is raised if the outputs differ above a certain threshold. This is however not always possible as sensors are expensive and the fault detection might be applied after the system has already been designed. Another option is to use knowledge about the model/system to predict the output of the sensors. This approach is called observer based fault diagnosis and can use a lot of the principles already developed for state and other observers. One example would

be to use a state-observer, predicting the output in the following form:

$$\begin{aligned}\dot{\hat{x}}(t) &= A\hat{x}(t) + Bu(t) - L(Cx(t) + Du(t) - y(t)) \\ \hat{y}(t) &= C\hat{x}(t) + Du(t) \\ r(t) &= V(y(t) - \hat{y}(t))\end{aligned} \tag{9}$$

The matrix V can be added as a filter to reduce the effect of disturbances and other faults onto the residual r. The state observer is a relative simple observer based residual generator and can rely on a lot of research in the field of state observers. There are other more sophisticated approaches to observing the output. But as I have used a parity space approach, I will not go further into explaining the observer based approaches.

3.2 Parity Space Residual Generator

The parity space residual generator uses a different kind of model-based approach compared to the observer ones. It aims at finding parity equations that evaluate to zero at all times. These equations can describe relationship between current sensors, or relationships of sensor and input values over time. A benefit of using parity space approaches over observer based residual generators is their more simple design, as they can be created using mostly linear algebra. Furthermore, they only depend on a finite amount of inputs making the residuals a finite impulse response, in comparison to the infinite impulse response of an observer one. To create a parity space residual generator the following approach [4] was used for a discrete linear system with disturbances d and the fault vector $f(k)$.

$$\begin{aligned}x(k+1) &= Ax(k) + Bu(k) + E_d d(k) + E_d f(k) \\ y(k) &= Cx(k) + Du(k) + F_d d(k) + F_d f(k)\end{aligned} \tag{10}$$

The residual can be generated using s amount of past measurements of both inputs and outputs.

$$y_s(k) = \begin{pmatrix} y(k-s) \\ y(k-s-1) \\ \vdots \\ y(k) \end{pmatrix}, \qquad u_s(k) = \begin{pmatrix} u(k-s) \\ u(k-s-1) \\ \vdots \\ u(k) \end{pmatrix} \tag{11}$$

The value s is a design variable, where an increase in s generally increases performance but also the computational complexity. To generate the residual the following two matrices are needed:

$$H_{o,s} = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^s \end{bmatrix}, \qquad H_{u,s} = \begin{bmatrix} D & 0 & \dots & 0 \\ CB & D & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ CA^{s-1}B & \dots & CB & D \end{bmatrix} \tag{12}$$

These matrices describe how the past internal state $x(k-s)$ and the input vector $u_s(k)$ influence the output vector:

$$\begin{aligned} y_s(k) &= H_{o,s}x(k-s) + H_{u,s}u_s(k) \\ 0 &= y_s(k) - H_{o,s}x(k-s) - H_{u,s}u_s(k) \end{aligned} \tag{13}$$

This means we have found $s+1$ equations that resolve to zero, they are however dependent on the internal state of the system. It is possible to decouple the internal state by multiplying the equation with a matrix v that is in the left null space of $H_{o,s}$.

$$v_s H_{o,s} = 0 \tag{14}$$

Inserting this into equation 13 and using the result as a residual:

$$\begin{aligned} 0 &= v_s(y_s(k) - H_{o,s}x(k-s) - H_{u,s}u_s(k)) \\ r(k) &= v_s(y_s(k) - H_{u,s}u_s(k)) \end{aligned} \tag{15}$$

The residual now evaluates to zero for all input and output signals, if there are no faults or disturbances. The residual now solely depends on faults and disturbances.

$$r(k) = v_s(H_f f(k) + H_d d(k)) \tag{16}$$

3.3 Optimized Parity Space Approach

The parity space residual generator generator creates multiple residuals, one approach is to choose a residual manually and assign it to a fault. A better approach [5] is to linearly combine the residuals using a matrix T. The matrix can be calculated using an optimization function trying to:

- Increase residual sensitivity for a specific fault
- Decrease sensitivity to other faults
- Decrease sensitivity to disturbances

The first challenge was to detect the presence of any fault and thus sensitivity to other faults is not taken into account, leaving the following cost function that should be minimized:

$$H_{f,s} = \begin{bmatrix} F_f & 0 & \dots & 0 \\ CE_f & F_f & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ CA^{s-1}E_f & \dots & CE_f & F_f \end{bmatrix} \tag{17}$$

$$H_{d,s} = \begin{bmatrix} F_d & 0 & \dots & 0 \\ CE_d & F_d & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ CA^{s-1}E_d & \dots & CE_d & F_d \end{bmatrix} \tag{18}$$

$$f(T_i) = \frac{1}{c_1 J_1(T_i)} + c_2 J_2(T_i); \tag{19}$$

$$J_1(T_i) = (T_i * v_s * H_f * f)^2 \tag{20}$$

$$J_2(T_i) = (T_i * v_s * H_d) R_f (T_i * v_s * H_d)^T \tag{21}$$

The disturbance and measurement noise covariance matrix R_f is not perfectly known and can be changed as a design parameter. The values c_1, c_2 can be used to choose between increased fault sensitivity and decreased disturbance sensitivity.

4 Software Implementation

4.1 Simulation

As mentioned previously the model was implemented in MATLAB using text based equations. To simplify the simulation and extension of the system the model was ported to SIMULINK, which is a graphical extension for MATLAB. This allows for easy experimentation as blocks of equations can be easily connected. I refrained from exploring other frameworks, as I was already familiar with MATLAB/SIMULINK and it is the software usually used in control system courses.
The model in SIMULINK is created using a MATLAB function solving:

$$\ddot{q} = M(q)^{-1}(-C(q,\dot{q})\dot{q} - G(q) + \tau(u)) \tag{22}$$

The state vector $x = \begin{pmatrix} x & \phi & \psi & \dot{x} & \dot{\phi} & \dot{\psi} \end{pmatrix}^T$ can then be calculated using two integrators for each element of q. A stop block is also included to stop simulation if the LegoWay falls to the ground $|\psi| \geq \frac{\pi}{2}$. The model uses the vector of the two wheel voltages $u = \begin{pmatrix} V_{right} & V_{left} \end{pmatrix}^T$ as input and outputs the internal states and the actual sensor output $y = Cx$. Gaussian white noise is added to both the internal states and the sensor outputs to model measurement noise and modelling errors. The controller is inserted as a discrete state space block and connected to the model.

4.2 Residual Generation

The residual generator generator is also implemented in MATLAB as the linearized model of the system can be directly used, without translating it into a different representation for a different programming language. The residual generation will be done by reading live values from the device and analyzing them in MATLAB, eliminating the need for students to deal with the old programming frameworks for the LegoWay. To simulate the residual generator inside the model they are transformed into a discrete state space representation. This state space representation has only the current values as inputs and stores the

past values of y and u internally. The residual generators can than be inserted into the SIMULINK model by connecting the input and output vectors.
When analyzing the signal in real-time the past values are stored in a vector and directly multiplied with the matrix $rC = \begin{bmatrix} v_s & v_s H_{u,s} \end{bmatrix}$, this reduces the amount of multiplications as time is crucial when dealing with live data. The data is received in blocks of 10 time-blocks and read and processed continuously.

4.3 Host Bluetooth Communication

Data from the LegoWay is read using a serial bluetooth connection. To connect to the device the script `btconnect` from the RWTH NXT toolbox can be used. After starting the program on the device, it enters an initial screen where you can start the application. If data is desired to be read, the host has to connect successfully to the device before continuing. In MATLAB data can be read from the device, using the `bt_com.m` script. First the script opens a file handle to `rfcomm0` and then starts reading packages. Each package starts with a byte that can be discarded followed by an integer time stamp. Afterwards the 50 integers storing input and output data can be read. The scripts reads data until the device is disconnected after which the file handle is closed.

5 Firmware

5.1 Programming Language

My first approach was to try to implement the controller on the host machine in MATLAB and send and receive commands over bluetooth to and from the device. This would have made running and understanding the implementation of the controller very simple. However, the system's instability and thus its requirement for fast responding control were too much for the high latency of an over bluetooth connected controller. As a consequence it is necessary to use a programming language that runs directly on the device
There are multiple different programming frameworks to program NXT lego mindstorms with different capabilities. To implement the discrete controller correctly it is required for the system to have at least simple timing options. One option is to use RobotC, which was used in the original implementation of the LQG controller. I choose to use the nxtOSEK framework instead, as it has in comparison to RobotC at least unofficial Linux support and is Open Source. As nxtOSEK is a RTOS it also has all required timing capabilities.

5.2 nxtOSEK

nxtOSEK is a RTOS that allows programming of the NXT controller, using a gcc toolchain and an enhanced firmware. The toolchain compiles C and C++ code. As a RTOS it allows for multithreading and event based synchronization. An example of how to install the toolchain on Arch Linux based systems can be found online [1], where I have written a guide based on [6]. The toolchain requires two files, a `.oil` file defining the tasks and their properties, and a `.c` file defining the actions of the tasks. The programming of the controller is kept rather simple due to the time constraint of the project.

5.3 Tasks

5.3.1 Main Task

The controller is split into four different task (Fig. 2). The main task `TaskM` starts by initializing the LegoWay. This includes the setting of the different fault modes, which is done sequentially. The user can change the values by turning the wheel connected to port A. After setting all fault values the gyroscopic sensor has to be initialized, as it has an offset. This is done by keeping the LegoWay still and reading 1000 gyro values, taking the average as the offset for the current run. Afterwards, the task stalls until the enter button is pressed again signalling the start of the controller. The main tasks then enters a permanent loop, where it first checks the current runtime, storing it to time each iteration of the controller. Then `TASKReadSensors` is started using an event. Following a synchronization event by the task, reading the sensors, `TASKWriteMotors` is started and the next

[1] https://github.com/DirkStober/nxtOSEK

internal controller state controller state calculated using `lqg_lego_x_next()`. Every tenth iteration the bluetooth task is started. After each iteration the time is stopped again to check whether the 2 ms time interval was accomplished. The main task is also responsible to set and reset the fault active flag once the fault start/stop times are reached.

5.3.2 Read Sensors & Write Motors

Both these tasks are driven by events from the main task. `TASKWriteMotors` starts by calculating the output values using the `lqg_lego_y()` function. Then it writes the results to the motors after adding an compensate the deadzone that occurs when controlling the motors using PWM signals. `TASKReadSensors` reads the sensor values, subtracting the offset, set during initialization, of the gyroscopic sensor. Also due to the possibility of connecting the gyro sensor in two different direction the gyro value is negated to work with the controller. If the fault active flag is set the faults are injected to the motor and sensor values respectively. Afterwards the Sensor/Motor values are written into a data array for the bluetooth communication.

5.3.3 Bluetooth Communication

The `TaskBT` controls the bluetooth communication between the device and the host. To reduce the impact on the other control task, only blocks of data are send. In this case the values of the past 10 iterations, which are 50 integers plus one integer for the current runtime of the first data block. One package is thus 204 bytes big . As the control and sensor values don't need the full range of one integer, it is possible to reduce package size, which was not done due to the latency between device and host being acceptable. The data is written by the sensor and motor tasks into an array of 102 integers. Half of the area is data that is sent currently and the other half is written too. The pointers to each half are switched for each iteration of the bluetooth task. Similar to the other auxiliary the bluetooth task is synchronized using events.

5.4 Faults

The used faults are relatively limited and focused on addition and multiplication to the sensor and actuator values. There are three faults for each sensor and actuator. All faults have a predefined start value and stop value. The first is an offset in the value, this a static valued added to the value at each iteration. The second value is a drift, where the offset increases over time $f_d(k) = FD(k - k_0)u(k - k_0)$ with FD being the size of the fault. The last fault is a static gain that is multiplied to the sensor/actuator value. In the simulation there is also a time delayed fault, that delays actuator signals by a static amount of time. The different variable names can be found in table 1, on the device interface the variables are set with different precision, which is displayed after the fault name with the indicator being shown after a vertical line.

Fault type:	MATLAB	C-code	Device
Start	f_t(1)	F_A_S_T[0]	FT0
Stop	f_t(2)	F_A_S_T[1]	FT1
Offset	f_*_offset(i)	F_*_OFFSET[i]	F*O...| i
Drift	f_s_drift(i)	F_S_DRIFT[i]	FSD...| i
Gain	f_a_gain(i)	F_*_GAIN[i]	F*G...| i
Delay	f_a_delay(i)	-	-

Table 1: Table of implemented fault variables in MATLAB and on the device with their variable name. ($*$ means both actuator and sensor fault)

5.5 Code generation

The Controller requires two equations in each iteration, one to calculate the next internal state and one to calculate the output values. The values of the matrix multiplications can change depending on the chosen design matrices for both the kalman filter (R1,R2) and the LQG-controller (Q1,Q2). A way to eliminate the need to rewrite parts of the function and manually optimize the matrix multiplication is to generate the C-code using the MATLAB coder. One MATLAB function for each equation has been implemented, which also convert the PWM motor control signals back to the voltage levels between $-8V$ and $8V$ and saturate the output value at $|u| \leq 60$. The MATLAB coder combines both functions into one source and header file (`lqg_lego_controller.c/h`). These are then used by the tasks of the main inside `lqg.c` (Fig. 3).

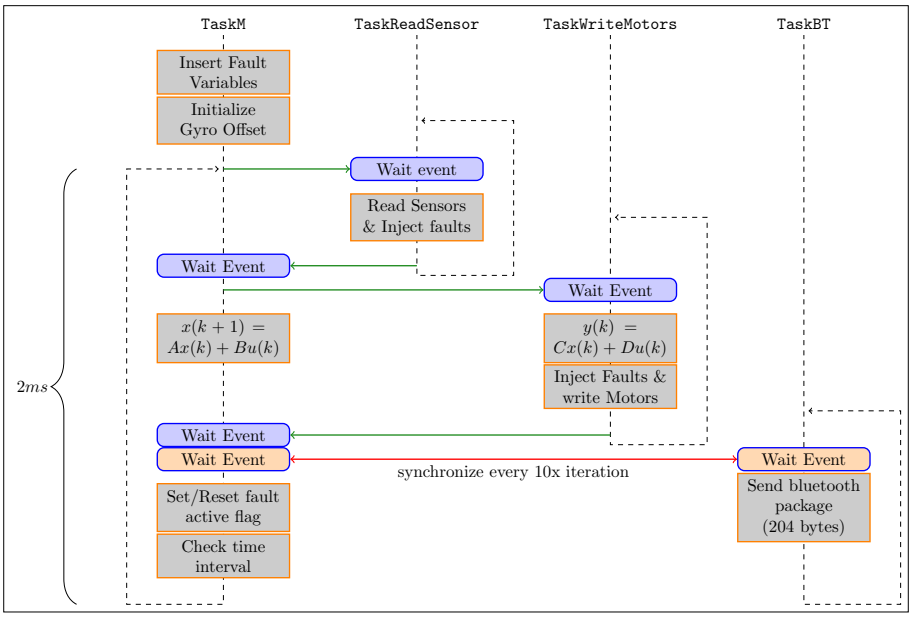

Figure 2: Different Tasks and their execution flow

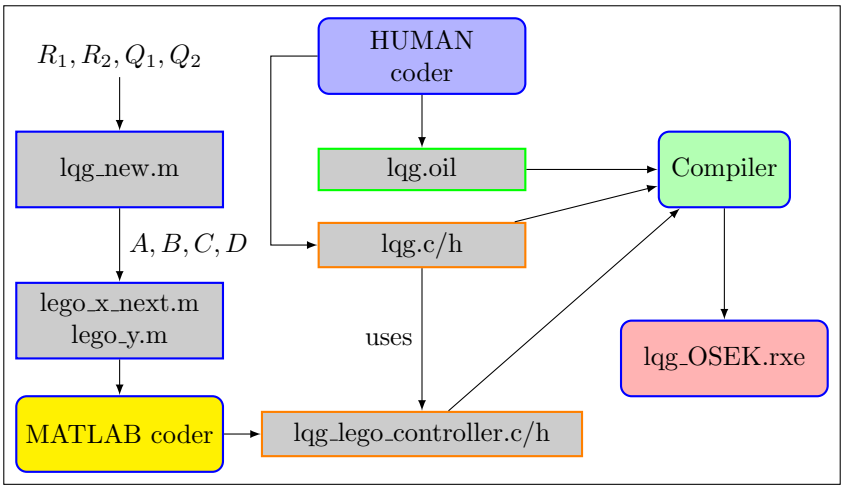

Figure 3: Diagram of software compilation

6 Results

As the motor sensors are quite precise it is sufficient to create a residual that just takes their output as input. These residuals trigger a spike, if there offset of one, that is detectable both in simulation and when working on real data. As a drift is a series of small offsets it is also detectable. Figure 6 clearly shows spikes above the threshold once the drift on sensor 1 starts (3000). The gyroscopic sensor varies a lot more than the rotation sensors and it is thus not possible to detect a fault by just looking at the single residual.
Although at first it looked like one of the residuals using all sensors was able to pick up the fault, it turned out to be just due to a programming error. Detecting actuator faults has proven more difficult, as none of the single residuals could detect any faults in the actuators. I thus proceeded to apply an optimization function, as described previously. The resulting residual is than filtered using a moving average and then the cumulative sum is calculated. This allows the detection of gain faults in the SIMULINK model (Fig. 5), however it was not possible to detect actuator faults in the real data (Fig. 4). The weights and disturbances matrices for the optimization function have been chosen arbitrarily and due to time limitations different configurations could not be tested. The main reason why faults could not be detected in real data is that the motors are not modelled correctly. This can be supported by the fact that the fault residual for the motors already start moving significantly before the fault is present (Fig. 4). As gyroscopic sensor faults could not be detected using a single residual, I also applied the optimization function. Here I was successful in detecting a drift in both the SIMULINK model and using real data. It is interesting to note that the actuator residuals also react to the gyro fault but to a lesser degree (Fig. 7), this is caused by the optimization function not taking decoupling from other faults into account.

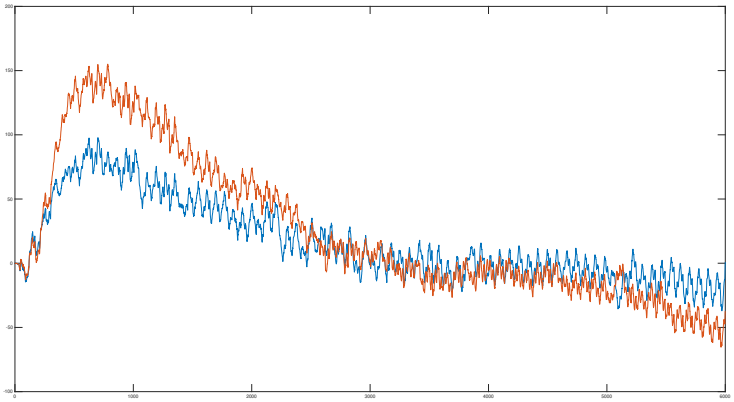

Figure 4: Fault residual for both actuators with a fault starting at 3000 using real data.

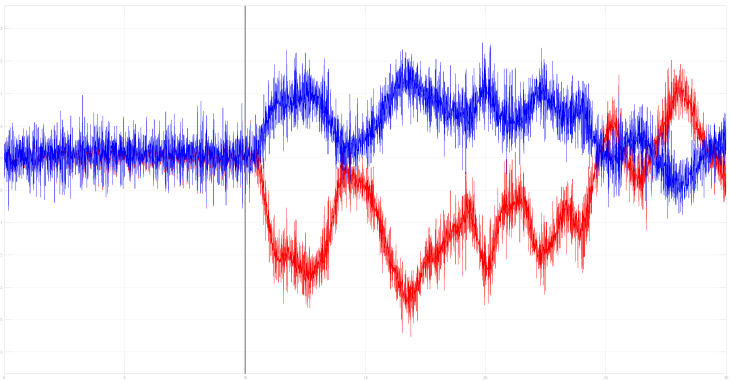

Figure 5: Fault residual for both actuators with a fault starting at the black line in SIMULINK.

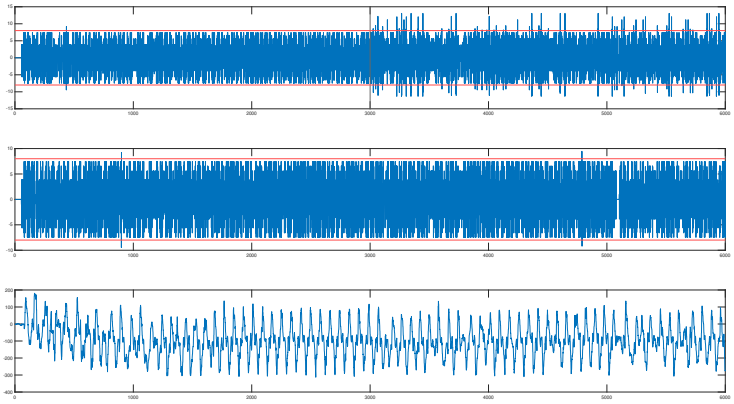

Figure 6: Sensor residuals for a drift in sensor 1, starting at 3000

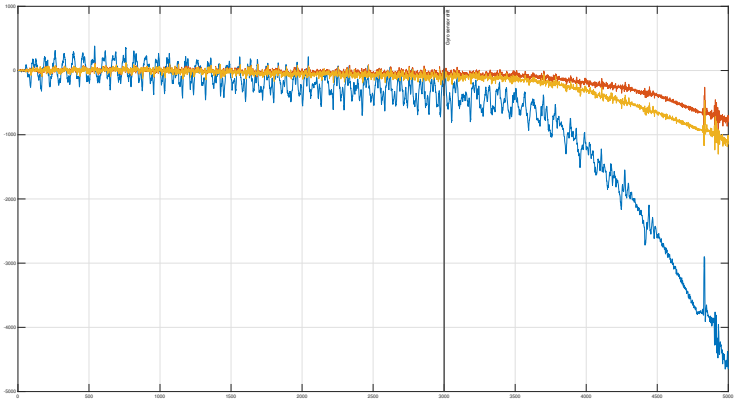

Figure 7: Sensor residual for gyroscopic sensor (blue) and actuators (yellow and red) using real data. The fault starts at the black line.

7 Conclusion

The project resulted in mixed success. Although the target of developing a Lab was not successful, I was able create a foundation from which further work can be done. A separation of the controller implementation using nxtOSEK and the fault detection was achieved, which provides a more user friendly way to test fault detection and isolation schemes. The amount of faults that can be injected are somewhat limited and might have to be extended in the future. The choice of using a parity space approach to generate residuals was used due to its simplicity but other approaches might yield better results. Furthermore, it might be possible to use the internal symmetry of the system to detect faults inside the motors (both motors and rotation sensors should behave similar). If the optimized parity space approach is to be used it is probably necessary to explore the covariance of measurement noise and disturbances to achieve better residuals.
It might have been more successful to first focus solely on generating residuals that detect the faults inside the SIMULINK model, before starting the controller implementation. This has been done due to the time limitations and a priority on the ability to present a running system. Finally, there is the choice of the operating system used on the host-pc. I chose to use Linux as it is in my subjective opinion simpler to program with and I did not have easy access to a windows machine. However it might be desired to run the Lab on Windows machine and although a small change in the bluetooth communication on the host-pc will be necessary I think it will not pose too big of a challenge. To conclude, this project delivers a framework which implements the ability to inject basic faults into the LegoWay and to detect them on a PC using MATLAB.

References

[1] "Lego mindstorms nxtway-gs building instructions," [Online]. Available: http://lejos-osek.sourceforge.net/NXTway-GS_Building_Instructions.pdf.

[2] P. Jonsson, P. Ali, and R. Olov, "Two wheeled balancing lego robot," 2009. [Online]. Available: https://it.uu.se/edu/course/homepage/styrsystem/vt09/Nyheter/Grupper/Rapport_group6.pdf.

[3] (). "Nxt motor internals," [Online]. Available: https://www.philohome.com/nxtmotor/nxtmotor.htm (visited on 2021).

[4] S. X. Ding, *Model-based fault diagnosis techniques: design schemes, algorithms and tools*, English, 2nd. New York: Springer, 2013.

[5] H. M. Odendaal and T. Jones, "Actuator fault detection and isolation: An optimised parity space approach," *Control Engineering Practice*, vol. 26, pp. 222–232, 2014, ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2014.01.013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0967066114000434.

[6] L. Cuvillon. (). "Nxtosek installation in linux," [Online]. Available: http://lejos-osek.sourceforge.net/installation_linux.htm (visited on 2021).