ISSN 0280-5316
ISRN LUTFD2/TFRT--5777--SE

Control of a Gantry-Tau Structure

Benoît Brochier

Department of Automatic Control
Lund University
July 2006

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
July 2006

Document Number
ISRNLUTFD2/TFRT--5777--SE

Author(s)
Benoît Brochier

Supervisor
Rolf Johansson Automatic Control in Lund and
Isabelle Boulay Institut National Polytechn. de Grenoble, France

Sponsoring organization

Title and subtitle
Control af a Gantry-Tau Structure (Reglering av en Portal-Tau-Robot)

Abstract
Todays, industrial robots don't correspond to the needs of the Small and Medium sized Enterprises (SME). Indeed, industrial robots are in many cases too hard to program or too expensive to be used by this kind of enterprises. In order to respond to this need, a European project was created, regrouping five major European robot manufacturers and five leading research institute and universities, named SMErobot. Its main task consists of exploiting the potentials of industrial robots, because they constitute the most flexible existing automation technology. This project set to create a radically new type of robot system, a whole family of SME-suitable robots.
A new structure of parallel robots was hence designed to respond to these needs, named a Gantry-Tau structure. A prototype of this parallel kinematic structure was created in the Lund Robotics Laboratory, to test its characteristics and the different control methods that can be applied to it. Moreover, it can be use as a demonstration tool. The thesis project concerns the velocity and position control of the small-scale linear actuators which are used for a prototype of the Gantry-Tau robot.

Keywords
Parallel structure, Gantry-Tau, cascade controller, PID, Kalman filter, backlash.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
107

Recipient's notes

Security classification

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I want to express my appreciation and gratitude to my coordinator, Dr Rolf Johansson, for his help and advices during this master thesis, and to have given me the opportunity to work at the control department laboratory of Lund University.

Thanks also to Anders Robertsson, Isolde Dressler and Toivo Henningsson for their technical help and many helpful suggestions, and to Rolf Braun, who shared his office with me, and gave me all the mechanical help I needed with the table robot.

I finally want to thanks Eduardo Mendes, who allowed me to make this master thesis in Sweden, Christian Commault and Isabelle Boulay for their help during this master thesis.

1 INTRODUCTION

More than 228 000 manufacturing SMEs (Small and Medium sized Entreprises) are working in the entire EU (European Union). They are a crucial factor in Europe‘s competitiveness, wealth creation, quality of life and employment.

In order to enable the EU to become the most competitive region in the world, lots of research efforts was done to strengthen knowledge-based manufacturing in SMEs.

However, existing automation technologies have been developed for capital-intensive large-volume manufacturing, resulting in costly and complex systems, which typically cannot be used in an SME context. Therefore, manufacturing SMEs are today caught in an “automation trap”: they must either opt for current and inappropriate automation solutions or compete on the basis of lowest wages.

So, a new need of affordable and flexible robot automation technology, which meets the requirements of SMEs, was created.

In order to respond to this need, a European project was created, regrouping five major European robot manufacturers and five leading research institute and universities, named SMErobot (**Figure 1**). It main task consist of exploiting the potentials of industrial robots, because they constitute the most flexible existing automation technology. This project set to create a radically new type of robot system, a whole family of SME-suitable robots.

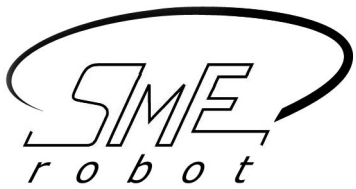


Figure 1 : SMErobot logo

The SMErobot initiative offers an escape out of the automation trap through:

- Technology development of SME robot systems adaptable to varying degrees of automation, at a third of today‘s automation life-cycle costs.
- New business models creating options for financing and operating robot automation given uncertainties in product volumes and life-times and to varying workforce qualification
- Empowering the supply chain of robot automation by focusing on the needs and culture of SME manufacturing with regard to planning, operation and maintenance.

This master thesis was done at Lund University, which is part of this European project, on a parallel kinematics structure.

The task of this master thesis being the control of a Gantry-Tau structure (which is a parallel structure) we will see in this report an introduction about what a parallel structure is, and a description of the robot and the model used during this master thesis. We will see next a presentation of the university, the tender specification, the work done and it results, and finally we will conclude about what was done and what can be done in the future.

2 PARALLEL ROBOTS

2.1 Introduction

As described in [1] [2], the idea of a robot, a mechanical structure able to move by itself, with or without the help of a human, dates at least as far back as the origin of the classical mythology. Indeed, in classical mythology, the god of metalwork (Vulcan or Hephaestus), created some mechanical servants that could move under their own power.

The first autonomous mechanical structure was created by Pierre and Henry-Louis Jaquet-Droz [3] (**Figure 2**).

Figure 2 : The 3 Jaquet-Droz androids

The 3 androids, The Draughtsman, The Writer and The Lady Musician, are able to draw different pictures previously loaded on it (by some mechanical mechanism), write some wanted text and play different musical song respectively (**Figure 3**). They were built in 1770 and displayed from 1774. In this time, these "mechanical puppets" was named androids, a mechanical structure that looks like a human.

Figure 3 : Hands of each android: Draughtsman, Writer and Lady Musician

The word "robot" was introduced by a Czech writer Karel Capek in his play R.U.R. (Rossum's Universal Robots) which was written in 1920 and played the first time in 1923. It is derived from the Czech word "robota", meaning heavy, monotonous or forced labour work. A robot may act under the direct control of a human (ex: the robotic arm of the space shuttle) or autonomously under the control of a pre-programmed computer. Robots may be used to perform tasks that are too dangerous or difficult for humans (ex: the space shuttle arm) or may be used to do repetitive tasks (ex: in automobile production).

The first generation of industrial robot was created in 1960 named Unimate 2000, and was a serial mechanical structure (**Figure 4**).

Figure 4 : Unimate 2000

The principal characteristic of a serial robot is to have an open kinematics chain, meaning that all arms are connected from the origin of the tool points by a serial connection [4] (**Figure 5**). Since this time, the use of serial robots has grown exponentially and nowadays it's possible to find one for almost each industrial application.

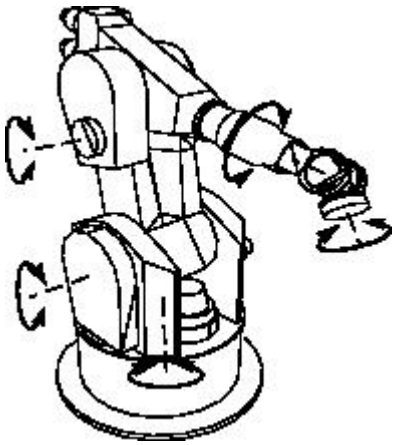


Figure 5 : A classical serial structure (Kuka-160 here)

But serial structures have some problems. The fact that the joints are connected each over with a serial connection reduces the precision (an error on a joint will be added to the error of the next joint, and so on until the tool point) and increases the weight of the structure (each arm need its own actuator and need to support the precedent arms). So it results in reduction of the maximum speed allowed.

Regarding these disadvantages, more and more research was done in the last decades about a new type of robot structure: the parallel structure.

A parallel structure is a closed kinematics chain, meaning that all arms are connected from the origin of the tool point by a parallel connection (**Figure 6**). That connection allows a higher precision (an error on a joint can be compensated, and not added, by an error on an other joint) and a higher velocity (most of the time the actuators are on a static platform, which reduces the weight of the arms, and so are easier to move).

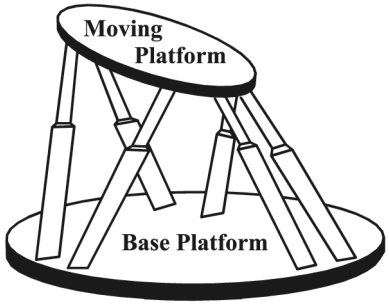


Figure 6 : Standard parallel structure (Stewart platform)

But, even if the parallel structures have a lot of advantages compare to the serial ones, there are still some disadvantages. The two biggest ones are the small working area and that the direct kinematics is more difficult to express computationally.

Some parallel structures, like the Stewart platform [5] or the ABB robot (**Figure 7**), are used today in very specific applications (ex: flight simulators or "pick and place" applications), but there is still lots of work and research to do to make the parallel structure as used as the serial one.

Figure 7 : ABB parallel robot

The parallel robot used in this master thesis is a Gantry-Tau structure [6], with 3 translational degrees of freedom (DOF), that can allow a big working area compared to the other structures (**Figure 8**). This structure has 6 joints, in a 3-2-1 configuration, representing how many joints are on each kinematics group of the robot.

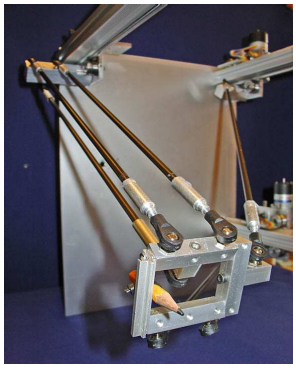


Figure 8 : Gantry-Tau robot

As we wanted to make some work on this structure, we used a table robot (a small version of the real big robot) to prevent some damage that could be done during the tests. Moreover, the table robot can become a demonstration model, easier to transport than the real size one, made using [7].

2.2 The table robot

Initially, the work was done with the full table robot. But after some experiments, it was found that the sensors for the position estimation were not good enough (especially to obtain the velocity signal) and there was a lot of backlash. So, a model of one track of the robot was created to correct these problems.

2.2.1 Description of the full table robot

The table robot consists of three parallel linear tracks which are attached to a plate at each end. One of the end plates has been reduced to an L-shape so that the platform can move freely in this area. On the other end plate, the interface boards between tracks and PC are attached. **Figure 9** shows the table robot from different points of view.

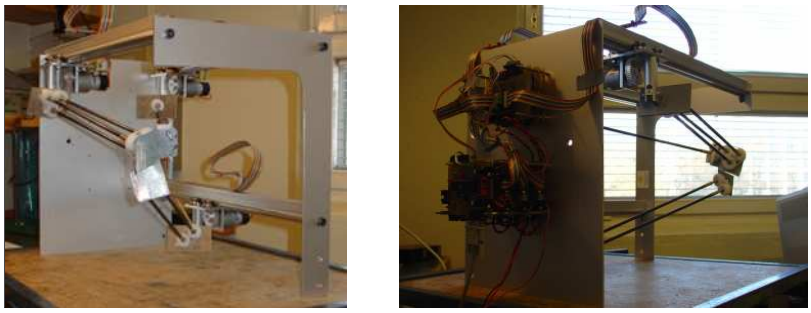


Figure 9 : Photos of the table robot

2.2.1.1 Linear tracks

Figure 10 shows a picture of the track. The cart moves on a toothed belt, driven by a DC motor which is coupled to a gear wheel. The tracks are about 50 cm long and fastened to the end plates with one screw at each end, so that the robot is easy to transport and assemble.

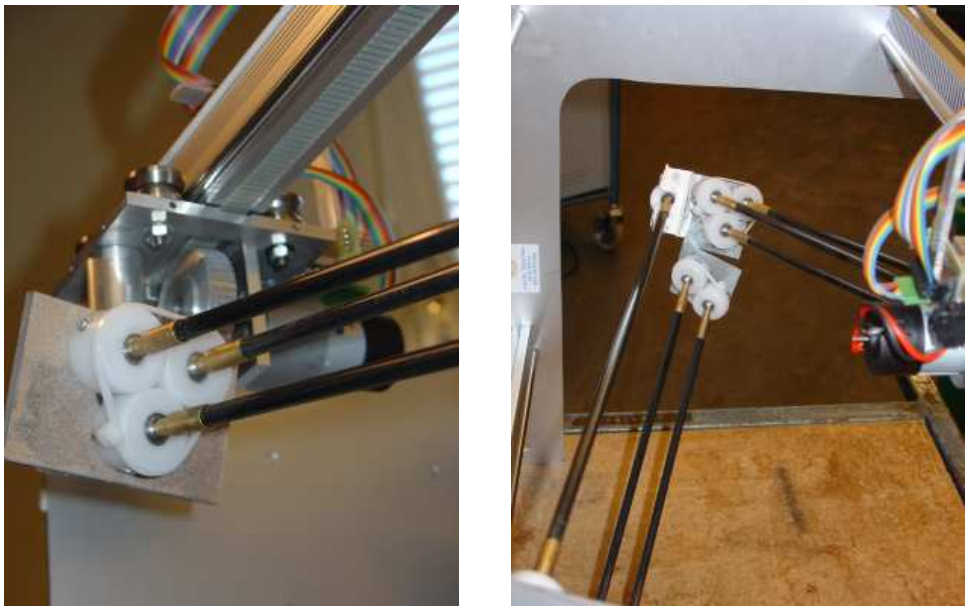


Figure 10 : Linear track with cart and arms with platform

2.2.1.2 Arms, joints and platform

The bars have a diameter of 6 mm. In order to make the robot reconfigurable in an easy way, the spherical joints are magnetic and adhere only by magnetic force. However, this causes problems

when moving the robot too violently. The platform consists of a plate which is folded in a way that is optimized for the small angular range of the spherical joints. When the platform is in the middle of the workspace, the plates on which the joints are attached should be perpendicular to the arms, so that the working range of the spherical joints is better exploited.

2.2.1.3 Track controllers

Each linear track is controlled by one ATMEL ATmega16 microcontroller [8]. The microcontroller sends the control signal from the PC to the motor driver. It also samples signals from analog encoders for position and velocity measurements, extracts the position and velocity and sends them back to the PC.

2.2.1.4 PC interface

The track controllers are attached through an I2C bus to a master microcontroller, which in turn handles the communication with the PC through a standard RS232 serial interface (**Figure 11**). Individual track controllers can also interface directly to the PC. Communication drivers are available on the PC side for Matlab/Simulink and Java.

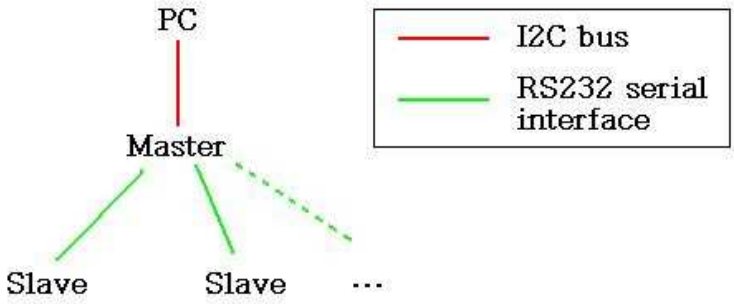


Figure 11 : Communication PC - Microcontrollers

2.2.2 Model of one arm with 2 motors

As described in the introduction of this chapter, the table robot was changed to improve it behaviour. Instead of working with the 3 tracks at the same time, we only worked on one (**Figure 12** and **Figure 13**), so it was easiest to manipulate and cheap if the change wasn't good after all.

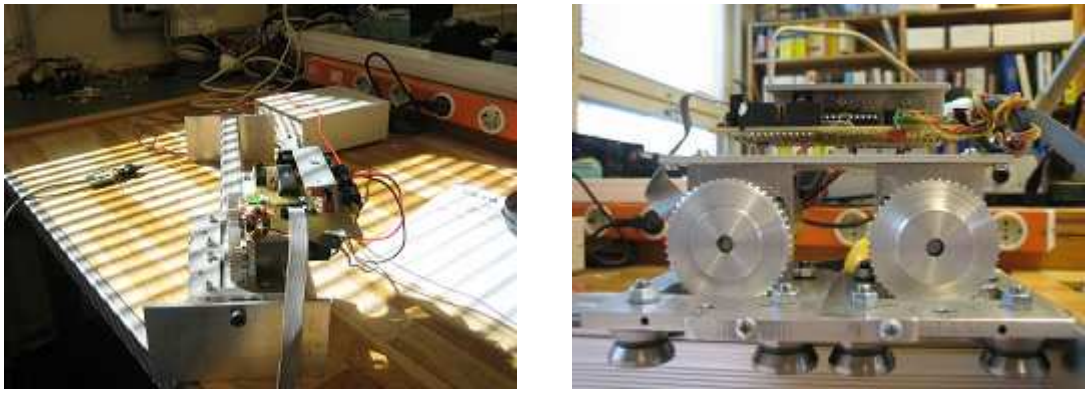


Figure 12 : Photos of the one arm model with 2 motors

Figure 13 : Front view of the track

This new version of one track of the table robot has better position sensors and 2 motors. But, why use 2 motors instead of only 1?

The main utility to have 2 motors instead of one is to allow backlash compensation. Indeed, when the track is moving in one way, the motor make turning the gear wheel, and with the help of the toothed belt, make the track moving. But when it wants to change it direction, the motor starts to turn in the other way, which makes the gear wheel turning, but not the track moving (**Figure 14**). It can be explained by the fact that the tooth thickness of the gear wheel is not the same than the one of the toothed belt, because of tooth clearances. These clearances are necessary to prevent gear teeth jamming due to manufacturing errors or thermal expansion.

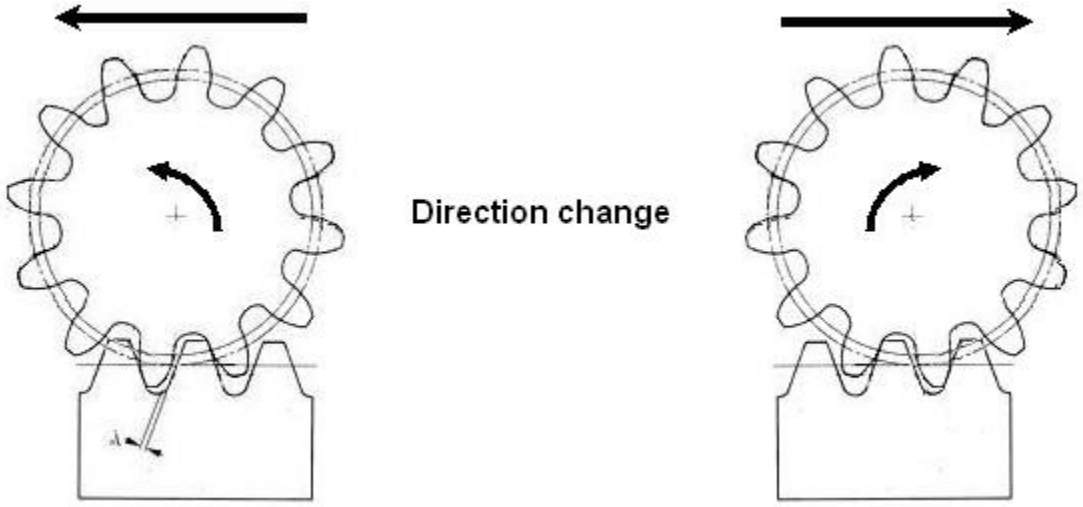


Figure 14 : Origin of the backlash problem

So we have a gap between the time we send the changing direction order and the time when the cart really changes it direction. It introduces discontinuity, uncertainty and makes accurate control of the tool centre point difficult.

Of course, we could have used some better transmissions to reduce the backlash, but these transmissions would have been more expensive. So the solution chose to reduce the backlash without increasing too much the cost of the table robot was to use 2 motors. Moreover, if we use both motors in the same direction, we can increase the maximum speed we can have for the track without burning them.

We will see how to use the motors to reduce the backlash in the section 3.4 of this report.

2.2.3 Communication PC - Microcontroller

To make a communication between a PC and a Microcontroller, we need that each one can send some information to the other and that these data can be fully understand. So we need to make a communication protocol to be sure that on each message sent, the data will always be at the same place, we can know for whom is this message, etc...

This protocol was created by Anders Blomdell, research engineer at Department of Automatic Control, Lund University, on May 2005. With it, it is possible to send up to 32 digital channels of 1 byte, and 31 analog channels of size between 2 and 6 bytes. The analog channel 32 is used as a configuration channel.

As the length of an analog channel depend on the length of the data we send, we need to read the message until the end before knowing when to stop (there is nothing to indicate the size of the message when we send it).

Digital in/out and poll command are sent in one byte. The first 5 bits are the channel on which we want to send the message (5 bits = 2^5 = 32 channels). The 2 next bits represent the action we want to do as shown below:

```
+-+-+-+-+-+-+-+-+
| 0 | 0 0 |  channel  |    Bit clear
+-+-+-+-+-+-+-+-+

+-+-+-+-+-+-+-+-+
| 0 | 0 1 |  channel  |    Bit set
+-+-+-+-+-+-+-+-+

+-+-+-+-+-+-+-+-+
| 0 | 1 0 |  channel  |    Bit get
+-+-+-+-+-+-+-+-+

+-+-+-+-+-+-+-+-+
| 0 | 1 1 |  channel  |    Channel clear
+-+-+-+-+-+-+-+-+
```

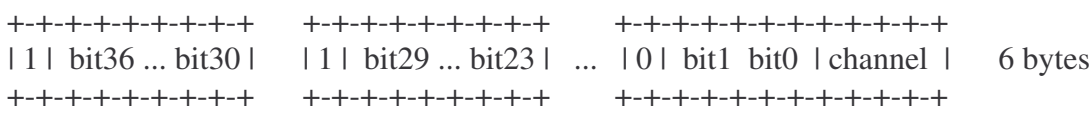
The last bit of the message is put to 0 to indicate that the message is sent on a digital channel.

For analog channels, we need to send the channel, but also the data we want to send. So we need some place to store the data in the message, and so analog channels are sent as 2 to 6 bytes, depending on the resolution:

```
+-+-+-+-+-+-+-+-+    +-+-+-+-+-+-+-+-+-+-+-+
| 1 |   bit8 ... bit2  |    | 0 |  bit1  bit0  | channel |          2 bytes
+-+-+-+-+-+-+-+-+    +-+-+-+-+-+-+-+-+-+-+-+

+-+-+-+-+-+-+-+-+    +-+-+-+-+-+-+-+-+    +-+-+-+-+-+-+-+-+-+-+-+
| 1 |  bit15 ... bit9  |    | 1 |   bit8 ... bit2  |    | 0 |  bit1  bit0  | channel |       3 bytes
+-+-+-+-+-+-+-+-+    +-+-+-+-+-+-+-+-+    +-+-+-+-+-+-+-+-+-+-+-+
```

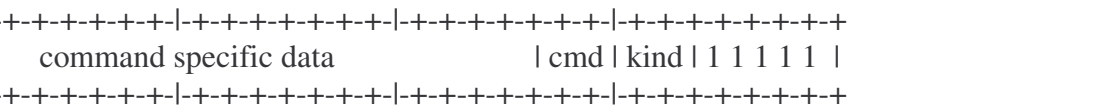
...



With this configuration, we can have a data of a maximum length of 4 bytes.

The last bit of each byte, except the last one, has to be set to 1 to indicate that the message is sent on an analog channel. The last bit of the last byte need to be set to 0 to indicate that it's the end of the message. We can thus know that the 5 first bits describe the channel and are not a part of the value sent.

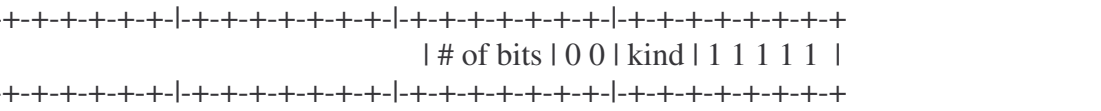
As said before, channel 32 is special, as it serves as the configuration channel. It is send on 4 bytes and can have the following layout:



The field “kind” corresponds to the kind of configuration we want to send:

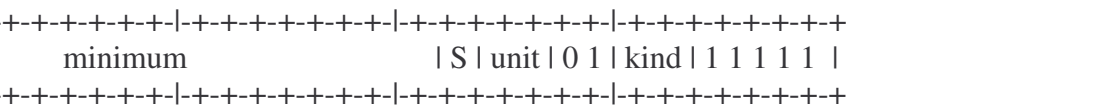
000 => end of configuration
001 => digital in
010 => digital out
011 => analog in
100 => analog out
101 => counter in

If the field “cmd” is 0, we send the resolution as shown:



The field “# of bits” corresponds to the resolution we want (1 to 32).

If the field “cmd” is 1, we send the minimum value limit as shown:



The field “S” corresponds to the sign of this value:

0 => +
1 => -

The field “unit” is the unit for this value:

000 == V
001 == mV
010 == uV
100 == A

If the field “cmd” is 2, we send the maximum value limit as shown:

```
+-+-+-+-+-+-+-+-|-+-+-+-+-+-+-+-|-+-+-+-+-+-+-+-|-+-+-+-+-+-+-+-+
|          maximum                      | S | unit | 1 0 | kind | 1 1 1 1 1  |
+-+-+-+-+-+-+-+-|-+-+-+-+-+-+-+-|-+-+-+-+-+-+-+-|-+-+-+-+-+-+-+-+
```

The field “S” corresponds to the sign of this value:

0 => +
1 => -

The field “unit” is the unit for this value:

000 == V
001 == mV
010 == uV
100 == A

2.2.4 Position and velocity estimators

2.2.4.1 Position estimation

The position estimation is made by 2 analog sensors (light leds). Each one measures the light reflected by the black and white stripes situated perpendicularly of the track direction (**Figure 15**).

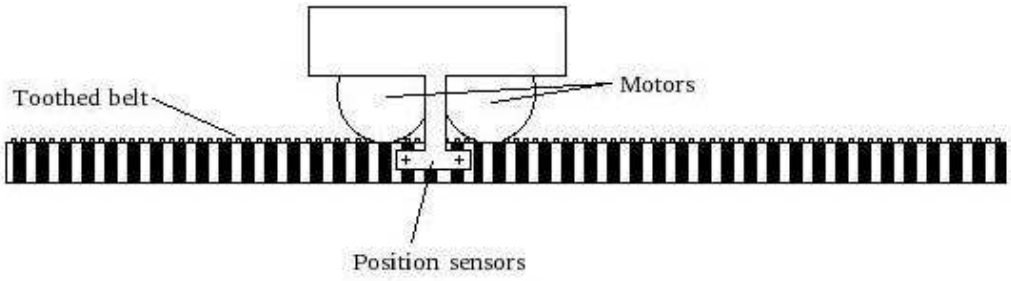


Figure 15 : Position estimation

As the width of the sensor is at least as big as the width of one stripe, we get a signal almost sinusoidal, in function of the position on the track, as we can see on **Figure 16**.

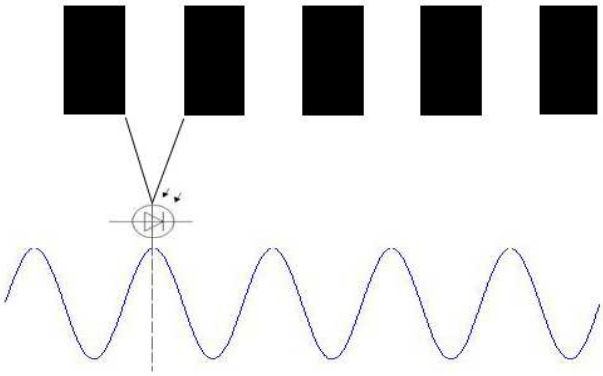


Figure 16 : Position signal

If we would have used only one sensor, it wouldn't have been possible to guaranty the same precision everywhere along the track. Indeed, if we look at the position signal, we can see that when we are on a "slow slope" area, we loose a lot precision. A little change in the position of the track won't be caught. And it's also impossible to know in which direction the track is moving.

To correct these problems we have 2 sensors, arranged to give signals that are shifted 90° between each other. So by looking at the position on both signals we can know the direction of the track, and we remove the problem of the "low slope" areas just by using the signal from the other sensor, as you can see in **Figure 17**.

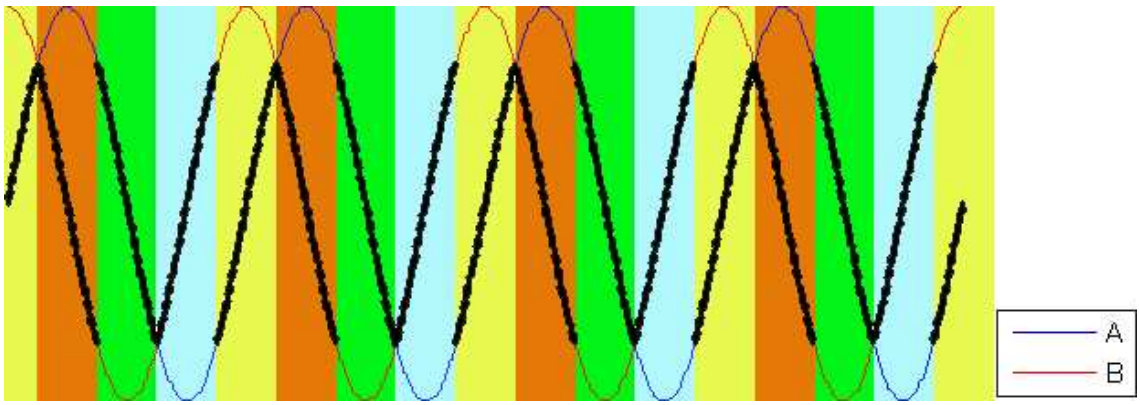


Figure 17 : Signal used for the position estimation

A and B are the 2 signals from the sensors, 90° out of phase. You can easily see on this picture all the different areas we have to obtain the best precision possible (4 different ones, in yellow, orange, green and blue). These areas correspond to the 2 "high slope" areas for signal A (one with a positive slope and one with a negative slope), and the 2 for signal B. In black is the signal we use in each area.

The signal from the sensors is very sensitive to a change in the position or the orientation of them. To have a good precision in the position estimator we need to know the real shape of the wave we get from the sensors.
To obtain it, we made a position calibration. We put a constant force on the motors and we send the signals from the sensors to the PC. There, we can find out on which area we are, and calculate an approximation of the signal curve in this area (the black signal in Figure 14) by a 3rd degree polynomial:

$$B = f(\theta) \tag{2-1}$$

where B is a sensor sampled signal, θ is the position estimate and $f(\theta)$ is polynomial of order 3.

Finally we take the inverse of this polynomial to have the position as function of the sensor's signal:

$$\theta = g(B) \tag{2-2}$$

To keep these parameters for the next use of the robot, we send and store them on the EEPROM of the microcontroller.

With this configuration we obtain 3278 values per mm, so we achieve a resolution of 0.3 µm. This resolution is very high, even too high. With this resolution, just a sun ray or a vibration can change the measured position of the track. And moreover we have some noise on this signal, so it's even worse and make the control of the robot impossible. So we decided to reduce this resolution to 0,15 mm, which is still a good resolution.

In order to always have a positive position, when the robot is powered on, the track move to the right until it reaches the end of the track. At this point, the intern position is set to 0.
Hence, the position maximum we can have with this track is 470, corresponding to the left track end and the minimum one is 0 for the right track end (left and right corresponding to **Figure 14**).

2.2.4.2 Velocity estimator

To facilitate better velocity estimates, analog derivative filters of the sensors signals are also available.

When the position is known, the velocity can easily be obtained from the analog derivative of the same sensor signal that was used for the position estimate. Indeed, this time we have a simple relationship between velocity and analog derivative:

The analog derivative can be expressed, using (2-1), as:

$$\dot{B} = \frac{\partial f(\theta)}{\partial \theta} \cdot \dot{\theta} \tag{2-3}$$

where $\dot{B}$ is the analog derivative of B, and $\dot{\theta}$ is the velocity estimate.

As θ is scalar, it is thus easy to expressed velocity as function of the analog derivative:

$$\dot{\theta} = \left(\frac{\partial f(\theta)}{\partial \theta} \right)^{-1} \dot{B} \tag{2-4}$$

Using (2-2), (2-4) can be rewritten as:

$$\dot{\theta} = g'(B) \cdot \dot{B} \tag{2-5}$$

If the speed is too high, the velocity signal saturates. At these speeds the velocity is obtainable from numerical derivatives. The inversions are implemented in fixed point by evaluation of low order polynomials.

Figure 18 shows a plot of the position and velocity estimates during a test run. The velocity estimate contains some high frequency measurement noise, and the peaks present in the plot are more a programmation problem (seems to be an overflow in a variable) than a physical problem.
A limitation code was added to removed this peaks, but it is just a "quick" solution, and the code need to be studied more in details to remove them instead of limit them.

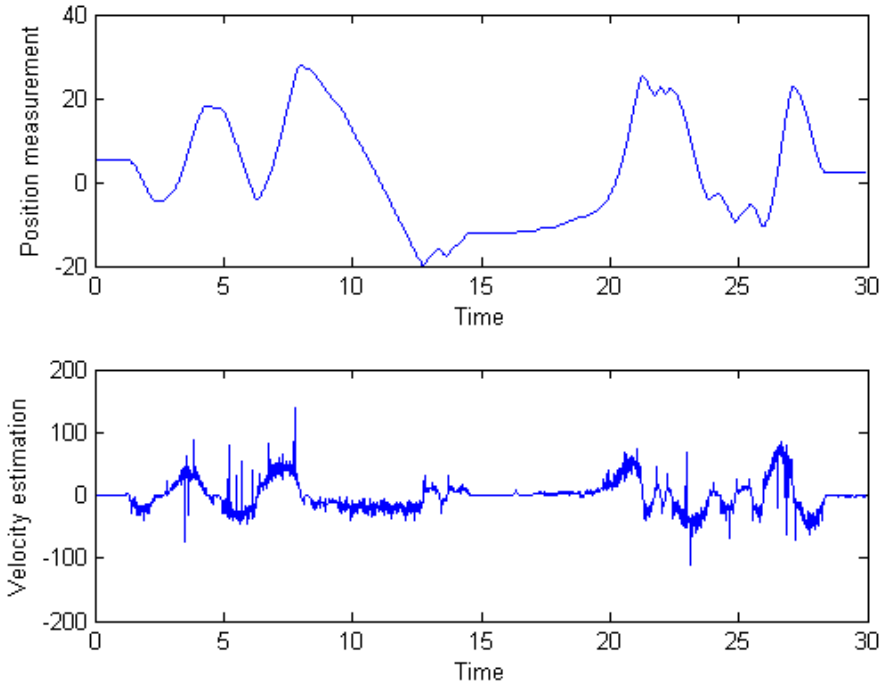


Figure 18 : Position and velocity signals

2.2.5 Microcontroller programming

In order to facilitate the programming of the microcontroller (and in case of modification), it is easier to make several programs than only one [9]. All the programs communicate between each other, as shown on **Figure 19**. Most of the programs are composed by a C file (.c), which contains all that the program have to do, and another library file (.h) with the same name, which contains the definition of all the functions and variables that can be used by another program.

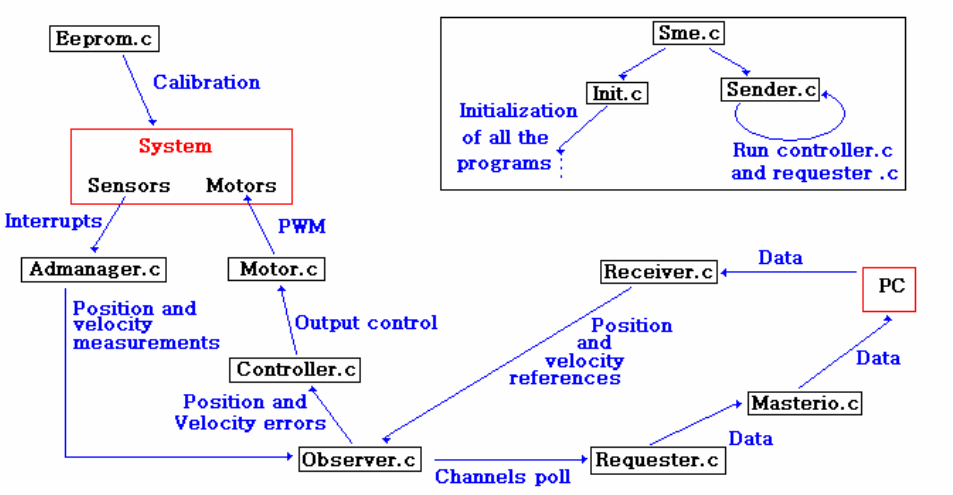


Figure 19 : Programs hierarchy

The function of each program can be described as follows:

- Admanager.c (appendix C.1): Handles the AD conversion interrupt. Takes turns to let the observer, requester and controller ask for channels to sample.

- Channels.h (appendix C.3): Defines the channels.

- Controller.c (appendix C.4): Computes the cascade controller and the backlash compensation.

- Eeprom.c (appendix C.6): Handles the system calibration and load/write on the eeprom.

- Init.c (appendix C.8): Initializes hardware and software.

- Masterio.c (appendix C.10): Handles the communication with the master (PC or master microcontroller). Messages from master via UART or TWI are passed to receiver. Messages to master are sent through UART or TWI depending on which way the last message was received.

- Motor.c (appendix C.12): Handles the motor supply, using the PWM.

- Observer.c (appendix C.14): Handles the position and velocity estimate.

- Receiver.c (appendix C.16): Processes messages from master. Setpoint changes are effected, polls are passed on to requester. Configuration commands are routed to the respective modules.

- Requester.c (appendix C.18): Accepts channel polls. Requests AD manager to sample at next option if necessary, marks available data for delivery at next time sending to master.

- Sender.c (appendix C.20): Takes turns to let controller and requester send to master. Initializes the track reference position to the right end of it.

- Sme.c (appendix C.22): Entry point. Initialization and some configuration. Passes control to sender.

3 MASTER THESIS WORK

3.1 Introduction

As described in section 2.2, the work in the beginning was done on the complete table robot (with 3 arms).
Isolde Dressler and Toivo Henningsson had done the communication PC – microcontroller and the control of the robot in Matlab/Simulink. But the results were quite poor due to some problems in the velocity estimation. Indeed, after some tests, we saw that two arms had their analog derivative sensors non-working. So the velocity estimation was only available on one arm.

Hence, the first work on this master thesis was done to improve the velocity estimation of the only arm where it was working. But as the other arms were broken, the model of the robot was changed and the work was done after on the new table robot with 2 motors and better sensors.

Once the work on the velocity finished, we began to work on the control of the track, first using both motors, and at the end we studied how to make backlash compensation with 2 motors.

3.2 Velocity estimation improvement

As the results for the velocity estimation improvement for both the full table robot and the model of one arm are quite the same, we will only describe in this part the work done on the model of one arm of the robot.

As described in section 2.2.4.2, the velocity is estimated by an efficient way, with good results. So the problem on the velocity estimation is not a problem in the way to calculate it.

As for every signal, we have some added noise on the velocity one. And moreover, it is a signal obtained with the analog derivative of another one, itself noisy, what it makes the velocity signal so bad.

The only way to improve it, as the calcul is good enough, is to filter the signal. Hence, we were able to remove the high frequencies which made the noise, with first a low-pass filter and finally a Kalman filter [10] [11].

3.2.1 Low-pass filters

3.2.1.1 Introduction

In a general way, the filtration consists of obtaining the best estimation of a signal that has been damaged by lots of different types of noise.

A low pass filter is a filter which keeps the low frequency and reduces the high frequency, above its cut-off frequency f_c, as shown on the frequency response on **Figure 20**.

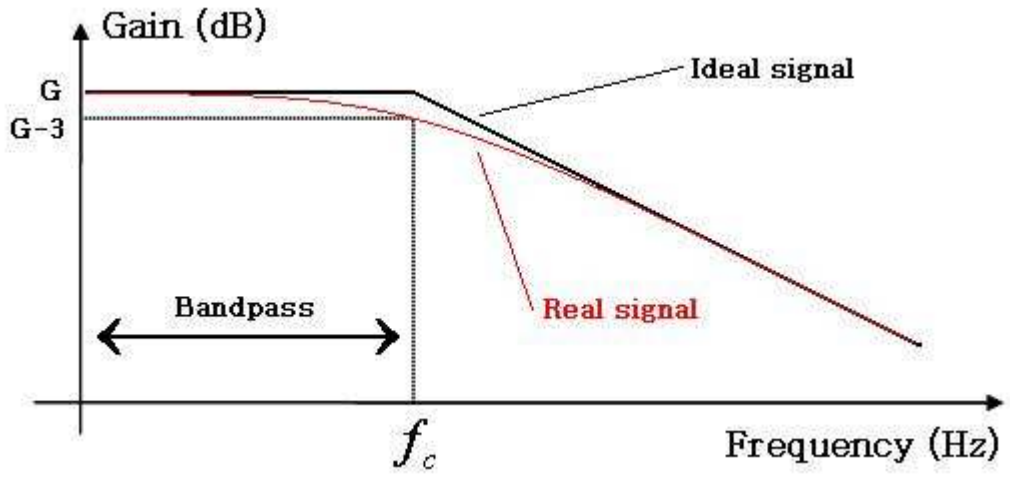


Figure 20 : Frequency response of a low pass filter

Usually, the gain G for frequencies before f_c is 0 (in dB), to keep the signal as it was before the filter. The real filter response starts to decrease for frequencies close to f_c, to reach $G-3dB$ at f_c, as shown on **Figure 20**.

3.2.1.2 Experimentation of the filters

The filters were selected to be Butterworth filters, due to its flat frequency response in the passband. But the problem is that our system sent data that are not analog but digital. And of course it's not possible to use an analog filter on a digital signal.

So we needed to find the corresponding digital filter, and to achieve that, the filters were created using the Matlab function [B,A] = BUTTER(N,Wn). For the parameters, N is the order of the filter and Wn is the cut-off frequency, with 0.0 < Wn < 1.0, with 1.0 corresponding to half the sample rate and the output are the filter coefficient (B numerator and A denominator).

f_c was chosen by looking at the noise when the velocity is constant, and trying to find its frequency. The highest frequency in the noise seemed to be around 10 Hz, and the highest frequency of the velocity obtained with the tests was around 2 Hz. So f_c was chosen as 5 Hz, to remove the noise but not the useful signal.

We chose to implement only a first order and a second order low pass filter because with a third order, the results would not have been so much better compared to the complexity added.

The application of these filters on the velocity signal gave the following results :

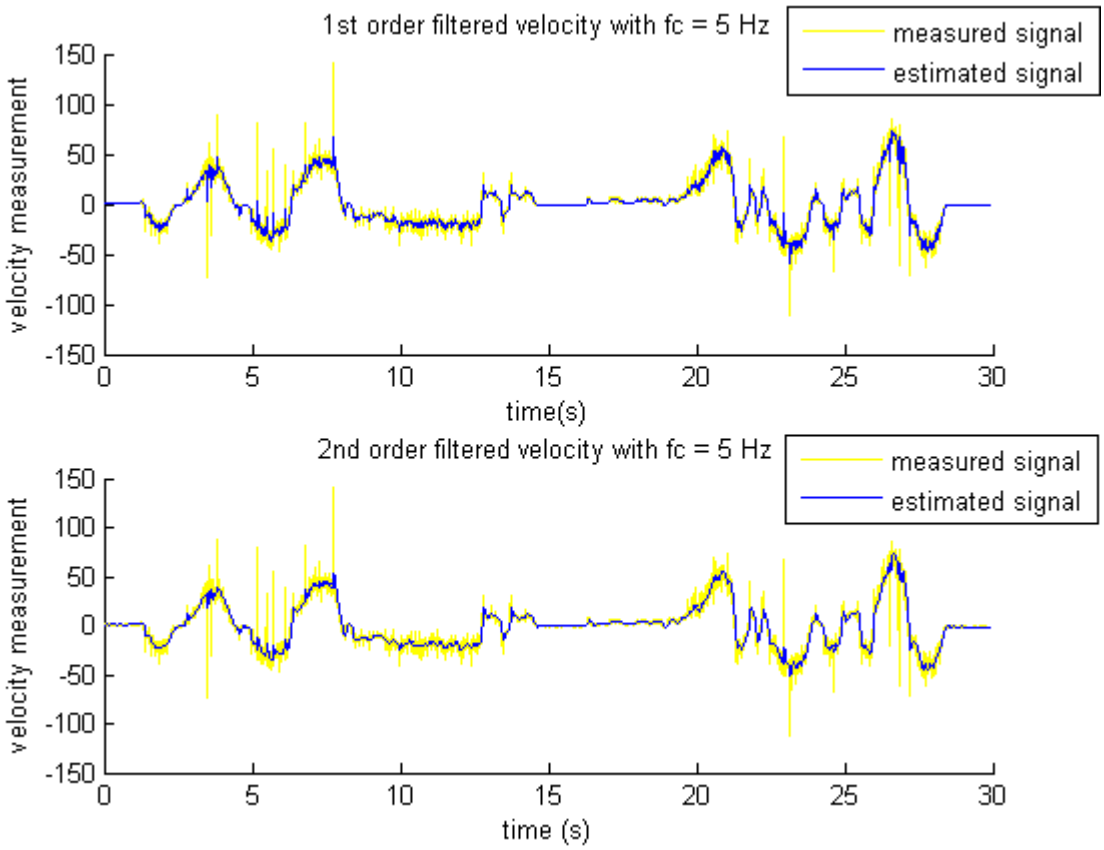


Figure 21 : Velocity filtered by order 1 and order 2 low-pass filters

We can easily see that the high frequencies of the noise are cut and that the order 2 low pass filter gives better results.

But the signal, even after these two filters, was not good enough. The problem is that such filters are based on a frequency analysis. To have better results, it's necessary to use a "model based" filter, as the Kalman filter, which is described in the following section.

3.2.2 Kalman filter

3.2.2.1 Introduction

A Kalman filter is a set of mathematical equations that provide an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error for some cost function. The filter is very powerful in several aspects: it supports estimations of past, present, and future states, and it can do it even if the precise characteristics of the modelled system are unknown.

Compared to the other filters seen before in section 3.2.1, this filter is not based on a frequency analysis but on state estimation. It means that, with a mathematical model of the system, it will be possible to estimate the next sample to come.
So, by comparing this estimate sample and the real sample, we will obtain a signal very close to the real signal without the noise.

3.2.2.2 Kalman filter formulation

The Kalman filter can be computed efficiently in a recursive formulation which, for every sample

time k, generates an optimal estimate of the mean and covariance of the state vector based on :

- the previous mean and covariance estimates at time k–1
- the process inputs and measurements at time k

The filter is really a two-stage algorithm: it first generates a "time update" which is a prediction of the state mean and covariance based on the previous state and process inputs. Then it does a "measurement update" which generates a new estimate which is based on the "time update" estimate and the sensor measurements. The relative influence of the time update estimate and the measurements on the measurement update depends on the "Kalman Gain," K, which is a time-varying parameter that indicates the relative reliability of the information sources based on the current state-covariance, P, and the known sensor covariance, V. The parameter W is the process covariance, corresponding to the relative reliability of the model of the system.

The filter algorithm is expressed below using notation from [12]:

TIME UPDATE

$$\overline{x}(k+1) = \Phi\hat{x}(k) + \Psi u(k) \qquad (3\text{-}1)$$

$$P(k+1) = \Phi\hat{P}(k)\Phi^T + W \qquad (3\text{-}2)$$

MEASUREMENT UPDATE

$$K(k+1) = P(k+1)C^T\left[V + CP(k+1)C^T\right]^{-1} \qquad (3\text{-}3)$$

$$\hat{P}(k+1) = P(k+1) - K(k+1)\left[V + CP(k+1)C^T\right]K(k+1)^T \qquad (3\text{-}4)$$

$$\hat{x}(k+1) = \overline{x}(k+1) + K(k+1)\left[z(k+1) - C\overline{x}(k+1)\right] \qquad (3\text{-}5)$$

where z is the position sampled signal, for $0 \le k \le N$ (N number of samples), with initial conditions $\hat{x}(0) = x_0$ and $\hat{P}(0) = P_0$.

3.2.2.3 Model identification

In order to use the Kalman filter, as described before, a model of the system is needed. There are different ways to identify a model of a system, as described in [13].

We used Matlab due to its powerful identification toolbox. Matlab was created by *MathWorks* and is a high-level language and interactive environment that enables users to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, or Fortran.

Moreover, Matlab offers the possibility of using another tool named Simulink. Simulink is a platform for multidomain simulation and Model-Based Design for dynamic systems. It provides an interactive graphical environment and a customizable set of block libraries, and can be extended for

specialized applications.

To start the identification with this toolbox, it is necessary to have some data from the system (the input and the corresponding output). In order to have a model which can represent the behavior of the system for a large range of frequencies, we need to have the response of the real system at these frequencies (ideally all the frequencies).
In the same way, if we need know that the system will only be used on a small range of frequencies, it is possible to choose to only make the system react at these frequencies. Hence we will have a very precise model, but only for that range of frequencies, the behavior out of these are unknown and can be totally wrong.

As we wanted a general model of the system, which can be used all the time, we chose to create a signal which contains the maximum range of frequency, as a pseudo random binary sequence (PRBS).

This input signal was created in Matlab/Simulink and put on the system. The received data was checked in order to remove any outliers, aliasing effects or trends that affected it.

There are many steps in using the identification toolbox of Matlab. The first one is, of course, to import the data of the system (the input and the corresponding system response). As these data will be used for the system identification and for its validation, the best thing to do is to cut them in half and to use one half for identification and the other half for validation.

Once the input data is imported and checked, it is possible to begin the identification process. Some preliminary experiments can be done, like the impulse response (**Figure 22**), to get some idea of time constants, by correlation analysis.

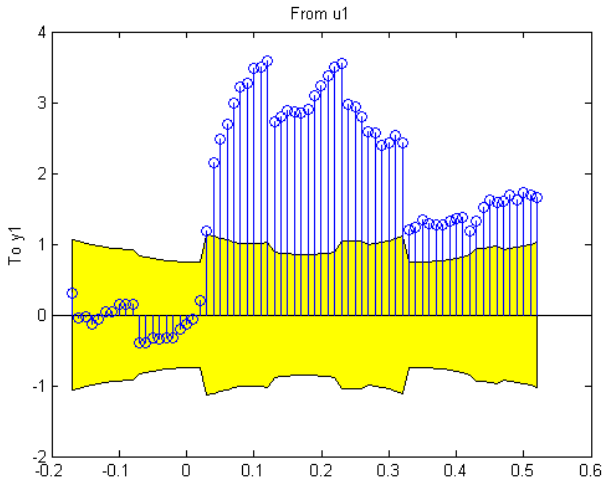


Figure 22 : Impulse response of the system

On this plot, the yellow area makes a confidence region corresponding to three standard deviations (99.9%). Thus it is easy to see if there is a time delay in the system. Here, we can see that 3 samples are needed to get out of this confidence region. The system has thus 3 samples time delay.

After the preliminary experiments, we need to select a model structure. The model structure determines the set in which the model estimation is performed. For example, a very simple such model set is the set of static gains K mapping the input to the output. That is the input-output model $y(t) = Ku(t)$. The complexity of the model structure, of course, affects the accuracy with

which the model can approximate the real process. Few dynamical systems can be well approximated by the model $y(t) = Ku(t)$.

The most general parametric model structure used in the System Identification Toolbox is given by:

$$A(q)\,y(t) = \frac{B(q)}{F(q)}u(t-n_k) + \frac{C(q)}{D(q)}e(t) \qquad (3\text{-}6)$$

where y is the output sequence, u is the input sequence and e is a white noise sequence with zero mean value. The polynomials A, B, C, D, F are defined in terms of the backward shift operator.

The general structure is rarely used, but some special forms, where one or more polynomials are set to identity, are more often used:

- AR model

$$A(q)\,y(t) = e(t) \qquad (3\text{-}7)$$

- ARX model

$$A(q)\,y(t) = B(q)u(t-n_k) + e(t) \qquad (3\text{-}8)$$

- ARMAX model

$$A(q)\,y(t) = B(q)u(t-n_k) + C(q)e(t) \qquad (3\text{-}9)$$

- Output-Error (OE) model

$$y(t) = \frac{B(q)}{F(q)}u(t-n_k) + e(t) \qquad (3\text{-}10)$$

- Box-Jenkins (BJ) model

$$y(t) = \frac{B(q)}{F(q)}u(t-n_k) + \frac{C(d)}{D(q)}e(t) \qquad (3\text{-}11)$$

- State-space model

$$x(t+Ts) = Ax(t) + Bu(t) + Ke(t) \qquad (3\text{-}12)$$

$$y(t) = Cx(t) + Du(t) + e(t) \tag{3-13}$$

It is not necessary that a model with more parameters or more freedom (more polynomials) is better. Finding the best model is a matter of choosing a suitable structure in combination with the number of parameters.

After some tests, the most relevant model structure was the ARX model and the State-space model. The model estimation method used for the ARX models was the least squares method, which gave the bests results, and the N4SID method for the State-space models, each with 3 samples delay and the focus on prediction (as the model will be used in the Kalman filter algorithm).

In order to compare each model, a validation plot is available in the System Identification Toolbox. This validation is made by simulating the calculated model with the input, and comparing the result with the validation. This technique is called the cross validation, due to the fact that the data for model identification and validation are not the same. **Figure 23** shows the results of the model obtained.

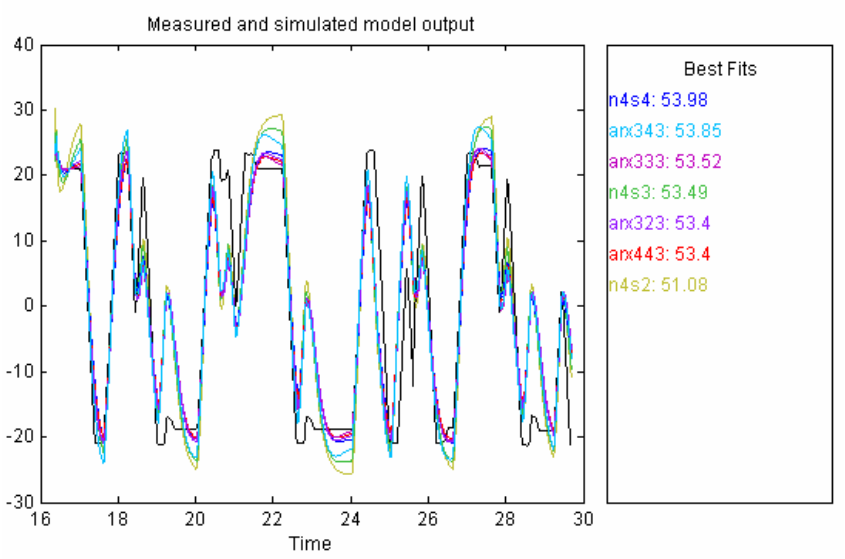


Figure 23 : Measured and simulated model output comparison

We can see that 3 different state-space models and 4 ARX models was identified and gave almost the same results. Just by looking at the fitting percentage, we can see that for the state-space models, higher order of model gave a better fit, meaning that with a low order model all the dynamics of the system are not captured. But with the ARX method, the 4th order model doesn't give a better fit. This means that all the freedom we added is just negligible compared to the other coefficient.

But just by looking at this figure is not enough. We need more information to find which one of these models is the best one. A way to compare the model to the real system is to compare their transient responses (impulse and step responses), and their bode diagram.

Figure 24 represents the step responses of each models compare to the one of the real system, **Figure 25** represents the impulse response and **Figure 26** represents the bode magnitude diagram. Each plot was done with the LTI viewer module of the Matlab toolbox.

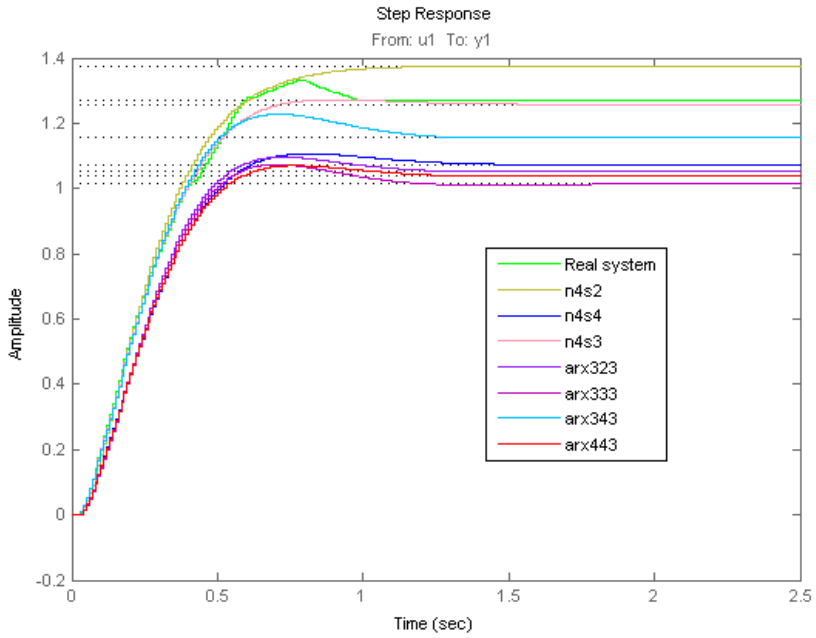


Figure 24 : Step responses of the models and the system

We can see on this figure that no model step response is the same that the model step response. But we can notice that the one which give the best fit is obtained with the model n4s3, even if the stationary gain is not exactly the same.

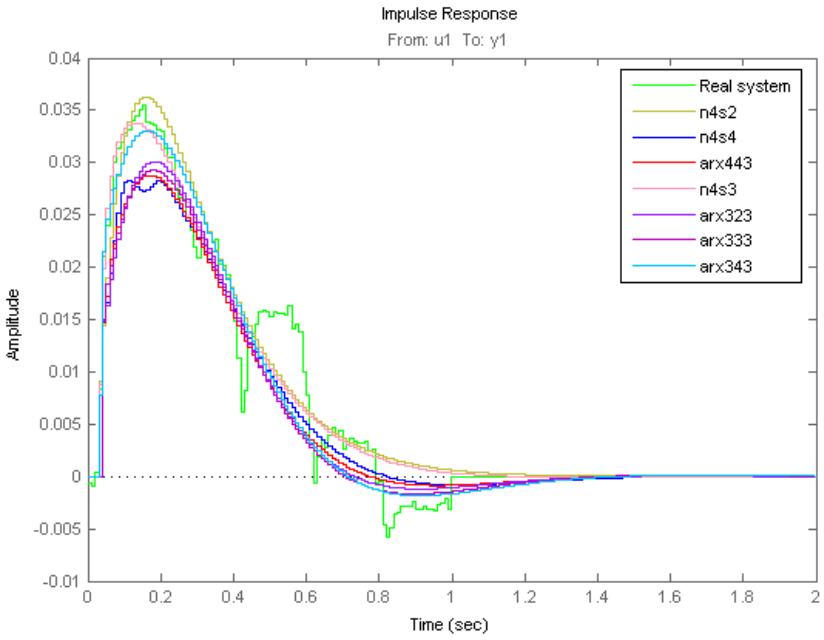


Figure 25 : Impulse responses of the models and the system

Figure 25 confirms what have been saw on the step response: the best fit for the impulse response is obtained with the model n4s3.

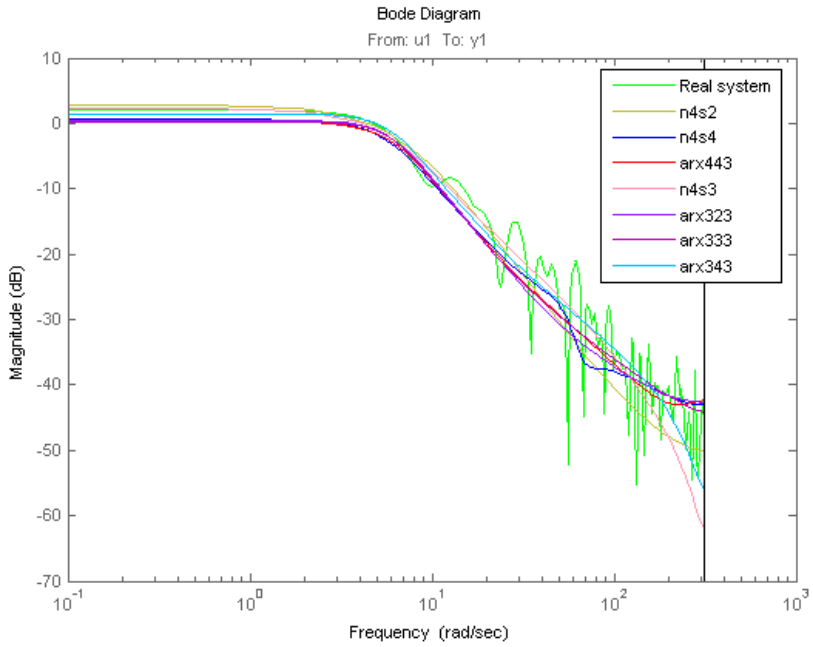


Figure 26 : Bode magnitude diagram of the models and the system

We can see here that all the models have a bode diagram similar to the real system.

Hence, after comparing many of the models obtained, the one that produced the best fit was a n4s3 third order model based on parametric state space estimation. The loss function for this model estimation is 0.0257981 and the Final Prediction Error (FPE) is 0.0262665, for a sampling interval of 0.01s.
The fit percentage for this model (see **Figure 23**) is only around 53,49%, which is not very good, but it was the best result we managed to have.

In order to try this filter on the on-line system, we needed to create a function that can be used in Matlab/Simulink. This function was created as a S-function to be implemented in a Simulink block, as shown on **Figure 27**. A description of this S-function, called *kalmanfiltern.m*, can be find in appendix A.

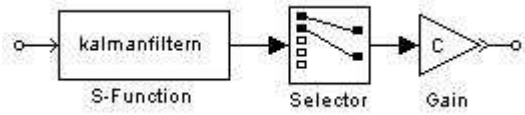


Figure 27 : Simulink Kalman filter

As the *kalmanfiltern*.m function gave the states as output of the system and the predicted covariance matrix $\hat{P}$, we put a selector to get just the states and they were then multiplied by the matrix C to obtain the estimate sensor output.

3.2.2.4 Kalman filter experimentation

The Kalman filter was tested on the real system, with running both motors in the same direction. The input was a step-like signal which alternated between low and high track movement.

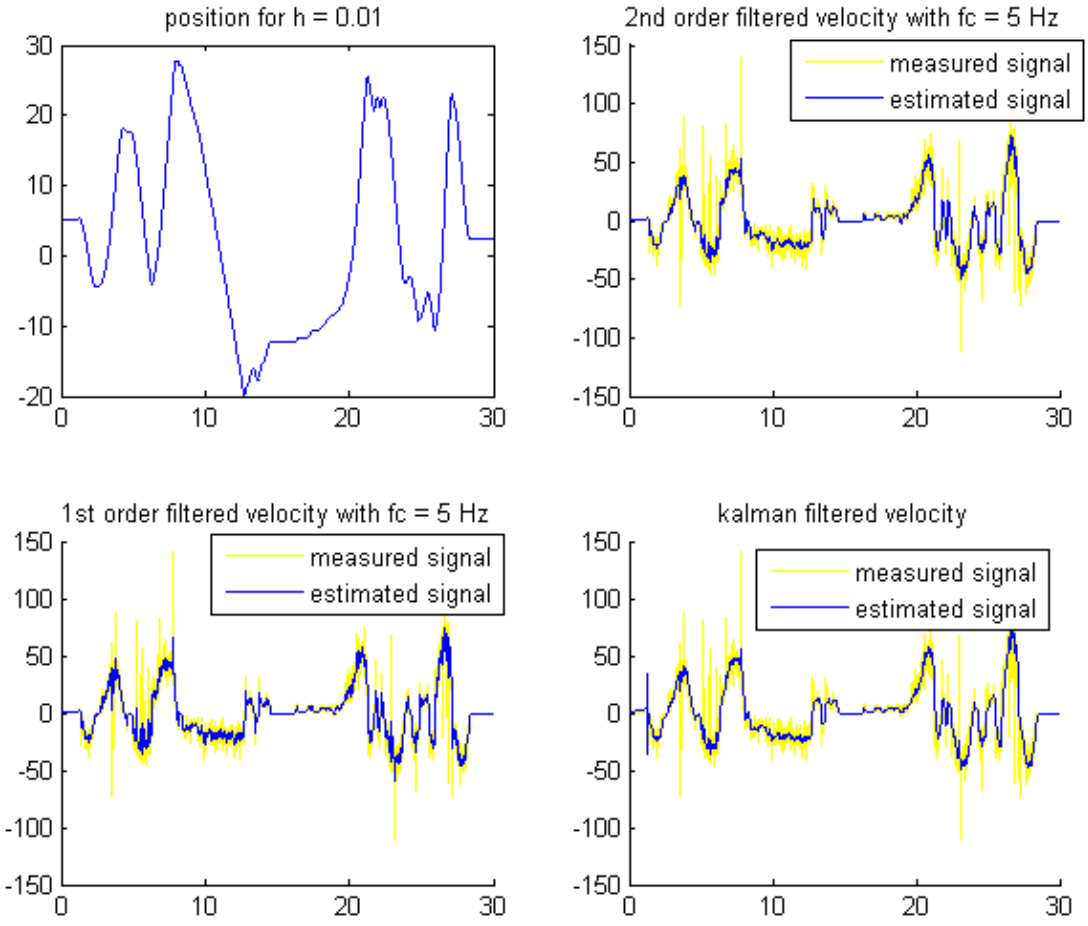


Figure 28 : Comparison between the velocity filters

Figure 28 shows a comparison between two filters. On first look, it seems that the Butterworth second order filter and the Kalman filter give the same results. But if we look more closely to the curve when the velocity sign changes, as shown on **Figure 29**, we can see that the Kalman filter gives a more smooth response accordingly with the position variations.

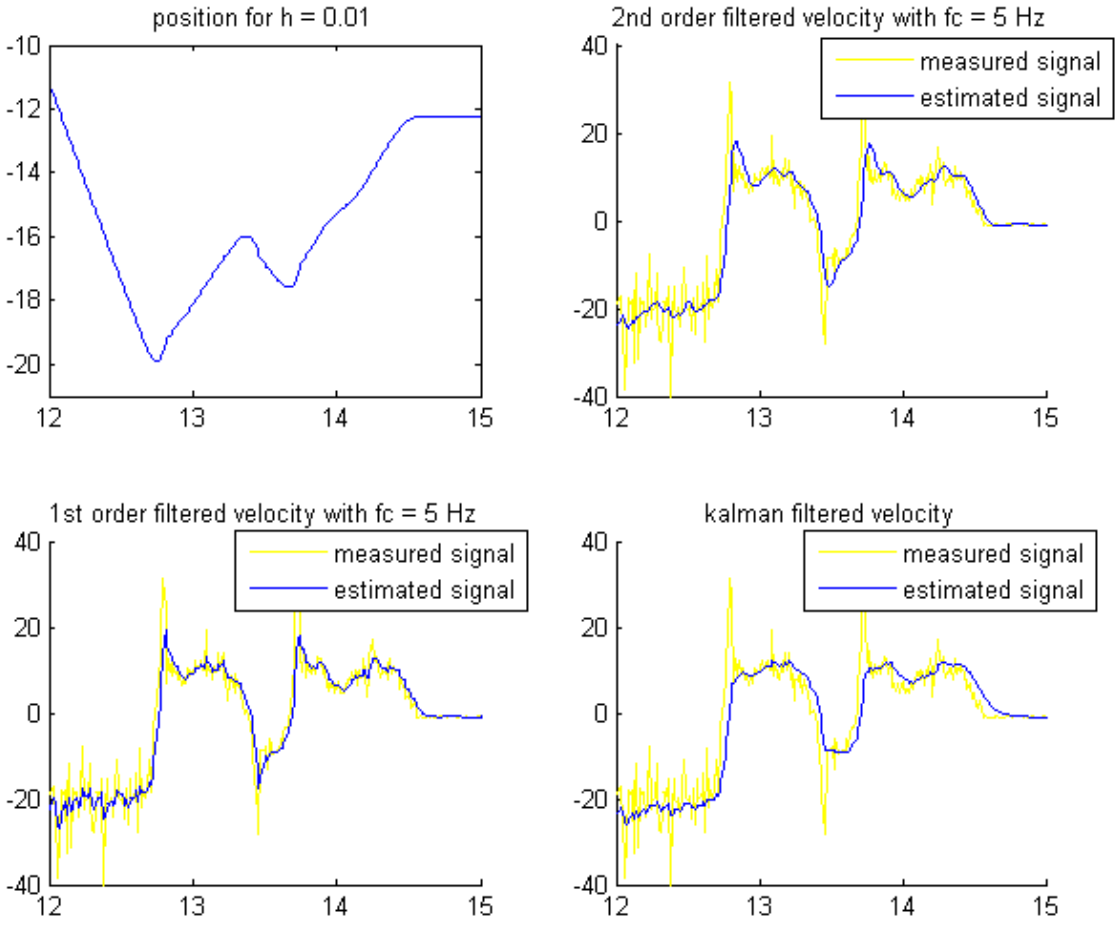


Figure 29 : Close comparison between the velocity filters

Hence, as expected, the Kalman filter gave better results, due to its more efficient calculation method.

3.2.2.5 Implementation on the microcontroller

As the Kalman filter algorithm is quite complex, it needs a lot of memory (for the code and for the variables). So when we tried to put it on the microcontroller, it appears that we didn't have enough memory on it, even with removing all that was possible.

3.2.2.6 Alternative solution

An alternative filter was created, very simple, to correct the velocity signal. This one was a mean filter which calculates the estimated output with the following equation:

$$y(k) = \frac{x(k-3) + x(k-2) + x(k-1) + x(k)}{4} \qquad (3\text{-}14)$$

where $x(k)$ is the output measurement.

We also add a limitation on the next sample value to prevent some measurement error.

3.2.3 Conclusion

We have seen in this part that the best filter found for the velocity estimate was the Kalman filter.

But its complexity prevented us to implement it on the microcontroller due to a lack of memory.

We decided to implement the alternative solution instead, to keep as much as memory possible for the controller part.

However, when the implementation of the Kalman filter will be possible, it could be good to make a new system identification to find a better model, and so increase the efficiency of this filter.

3.3 Control design

3.3.1 Introduction

Once the communication between the PC and the microcontroller was done, the objective was to make the track move to a desired position, by itself.

Used without any controller, the track will move to the desired position (also named a reference position) and stop when it will reach it. But the system is not ideal, so the communication between the sensors and the microcontroller took some time, the track have it own inertia, etc… All this things will make the track going further the desired position and never go back.

So to reduce this problem (and trying to remove it at all), we use controllers, which basically try to reduce the difference between the measured position and the desired position to zero, as shown on the equation below:

$$e(k) = \left(y_m(k) - y_d(k) \right) \to 0 \qquad (3\text{-}15)$$

where $e(k)$ is the error, $y_m(k)$ is the measured position and $y_d(k)$ is the desired position.

There are lots and lots of different kind of controller, but the most used is probably the PID controller, and is the one which was used on this master thesis.

All the tests were done with the same step reference with a java program (SerialIO.java, which can be found on appendix B). This java interface, very simple, allows the sending of a position and a velocity reference. As soon as a reference is sent to the system, the java program gets the data from the microcontroller (basically the time, the position measurement and the velocity estimate) and stores them on a Matlab file (named curve.m).
In order to facilitate the use in Matlab of this file (and so loose less time), a drawing command is added at the end of each, allowing the plot of the curves for a time in ms and a position in mm.

The motor input sent by the microcontroller is a PWM (Pulse Width Modulation) signal.

3.3.2 Without any controller

Before implementing any controller, it is needed to check the system and identify any problem that are physically or conceptual problems. Hence, for all the tests after, we will be able to know if it is a problem in the program (the tuning of the parameters, some coding errors, etc…) or a “usual” physical problem.

So, at the beginning, we just moved the track with a constant low speed and looked at the position and velocity estimate, as shown on **Figure 30** , in order to see if the system response is the same along the whole track, or if there is some “problem areas”.

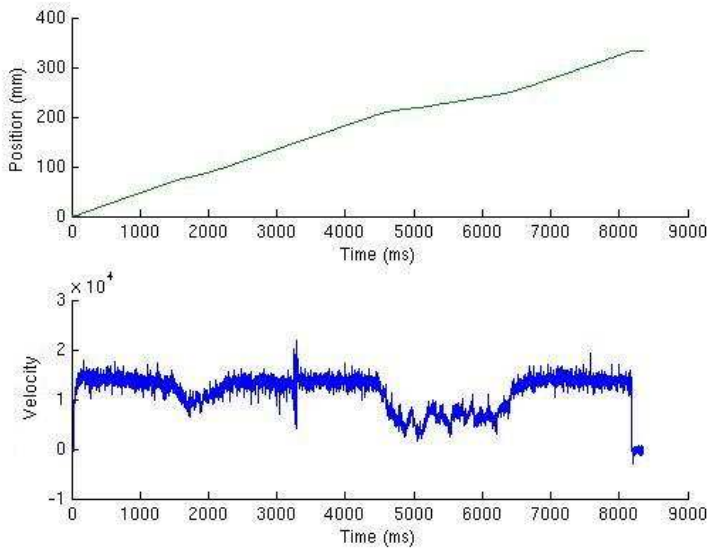


Figure 30 : Test without any controller

We can see on this figure that with a constant force on the motors, the velocity is not constant. This means that the track is not physically the same everywhere.

Hence, we can identify some areas where it is harder to move the track at positions 90 to 110 and 220 to 290.

Moreover, the peaks we can see on the velocity plot show a sensor problem at this position (position 190). After a close study of the track, we have seen that it was a problem with the stripes. Indeed, the black and white stripes, used for the position estimate (see section 2.2.4.1 for more details), was glued on the track structure. But at this position, the 2 stripes are not really stick, as show on **Figure 31**, and so the sensors value will be wrong, which have a bigger influence in the velocity estimate (use the analog derivative of the sensors signal) than in the position estimate.

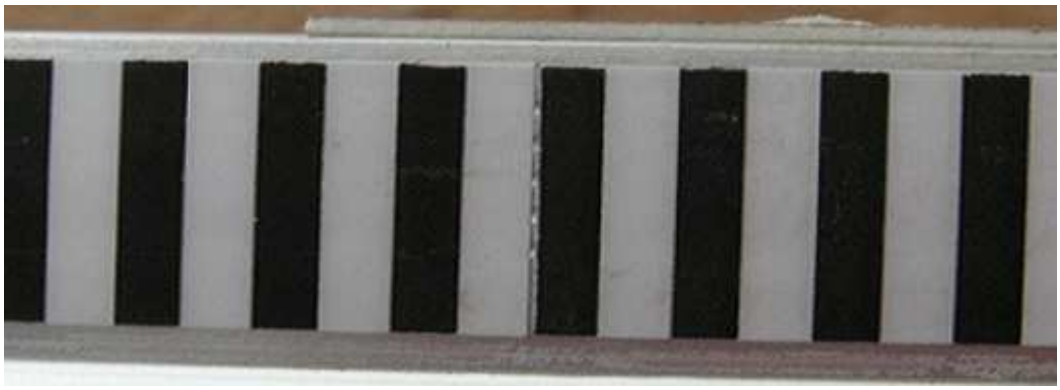


Figure 31 : Photo of the stripes problem

We can also identify another peak that is not the noise at position around 300. This one is caused by a black stripe which is a little bit damaged at this position.

3.3.3 Position control

As explained in the introduction of this section, a position controller is needed to keep the cart in the desired position.

In order to do that, a basic controller, a proportional controller P and a proportional and integral controller PI was tried.

3.3.3.1 Basic controller

The first controller we made was done by sending a constant PWM to the system, depending on the sign of the error when it is not zero. Hence, if the measured position is smaller than the reference position, the error is negative (see (3-15)), so the PWM sent to the motors is negative, and became positive when the cart cross over the reference position.

This controller can be expressed as:

$$\begin{cases} u(k) = PWM \quad \text{for} \quad e(k) > 0 \\ u(k) = -PWM \quad \text{for} \quad e(k) < 0 \end{cases} \tag{3-16}$$

where $u(k)$ is the control signal.

The PWM value is expressed in the system as a hexadecimal number, with the notation 0x…

Figure 32 shows the results of this test for a reference value of 305, and PWM = 0x100, 0x120 and 0x160.

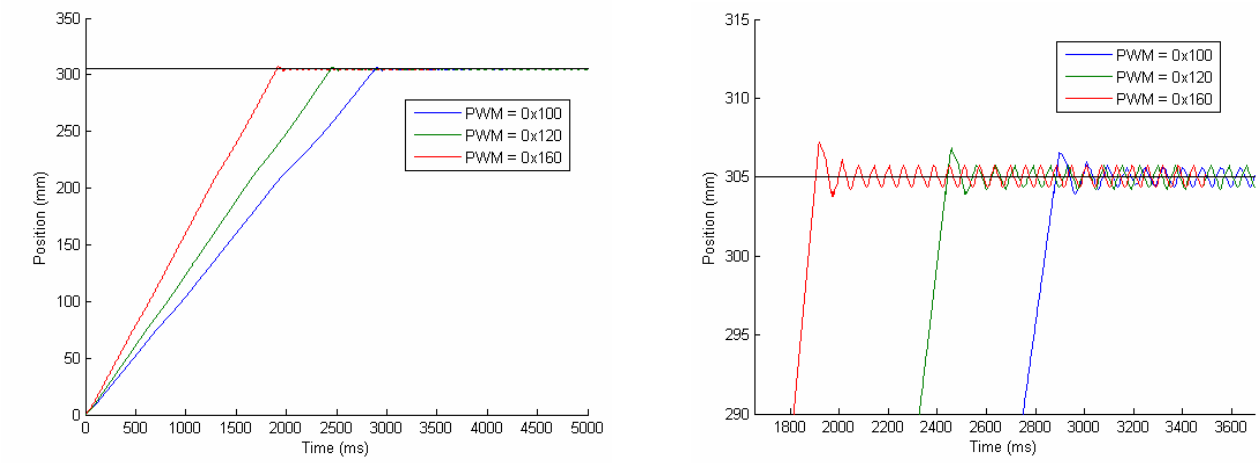


Figure 32 : Step response without any controller

We can see on this plot that the 3 curves have almost a constant slope until they reach the reference position. As expected, the slope is as big as the PWN is. We can also see on the zoom plot (the figure on the right) that the cart oscillates at the reference position, which is very harmful for the system. So we need to implement some controllers that can reduce the speed when the track is close to its reference position.

3.3.3.2 P controller

A P controller is a controller where the output signal is directly proportional to the error $e(k)$, as expressed on the following equation :

$$u(k) = K \cdot e(k) \qquad (3\text{-}17)$$

where K is the proportional gain.

In order to limit the PWM (to prevent to damage the motors) and to compensate the friction, we inserted a saturation on the output signal, which had the following shape (**Figure 33**):

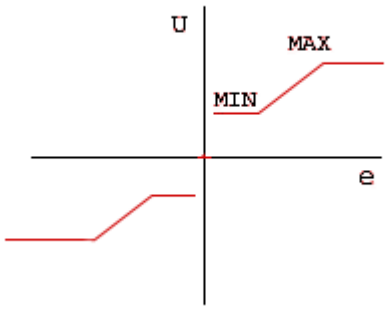


Figure 33 : Limitation of the output signal in function of the error

Hence, when the error is very big, meaning when the track is far from the reference position, its speed will be saturated to the maximum (corresponding to the field MAX_VEL in the controller program, in appendix C).
In the same way, when the error is very small (the track is close to its reference position), the velocity will be saturated to the minimum value (MIN_VEL field in the controller program, in appendix C), so the friction will be compensated and the track will continue to move.
Between these 2 saturation parts, the relation between the output signal and the error is linear, depending on K.

3.3.3.2.1 Choice of MAX_VEL

The following test was done for $K = 2$ and MIN_VEL = 0x80, with a reference value of 305. The value for MAX_VEL was 0x100, 0x200, 0x300 and 0x3FF. The value 0x3FF corresponds to an internal saturation already present in the motor.c program (which can be found in appendix C). So we did not try to increase this limit, to not damage the motors.

We can see on **Figure 34** that the only difference is in the first part of the curve, where the error is big enough to saturate the output at MAX_VEL value. The bigger MAX_VEL is, the faster we reach the reference position.
We can also see on the zoom plot that for each value of MAX_VEL, the overshoot is the same (~0.22%), so the value chose for MAX_VEL was 0x3FF.

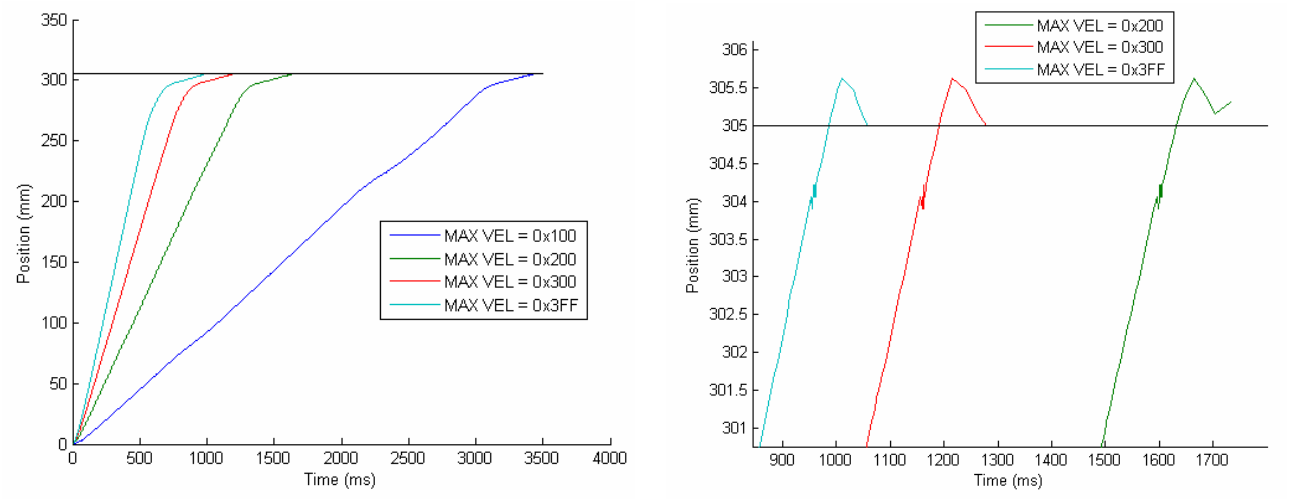


Figure 34 : Tuning of MAX_VEL

3.3.3.2.2 Choice of MIN_VEL

The following test was done for $K = 2$ and MAX_VEL = 0x300, with a reference value of 305. The value for MIN_VEL was 0x50, 0x60, 0x70, 0x80 and 0x90.

We can see on **Figure 35** that the only difference is in the last part of the curve, where the error is small enough to saturate the output at MIN_VEL value. For the value 0x50, we never reach the reference position because the force sent to the motors is not big enough to compensate the friction. For the value 0x90, the cart oscillates around its reference position, which is unacceptable. For the other values, the track reaches the reference position and don't oscillate, or just a bit. So the value we chose for MIN_VEL was the one which gave the fastest response of the system, 0x80.

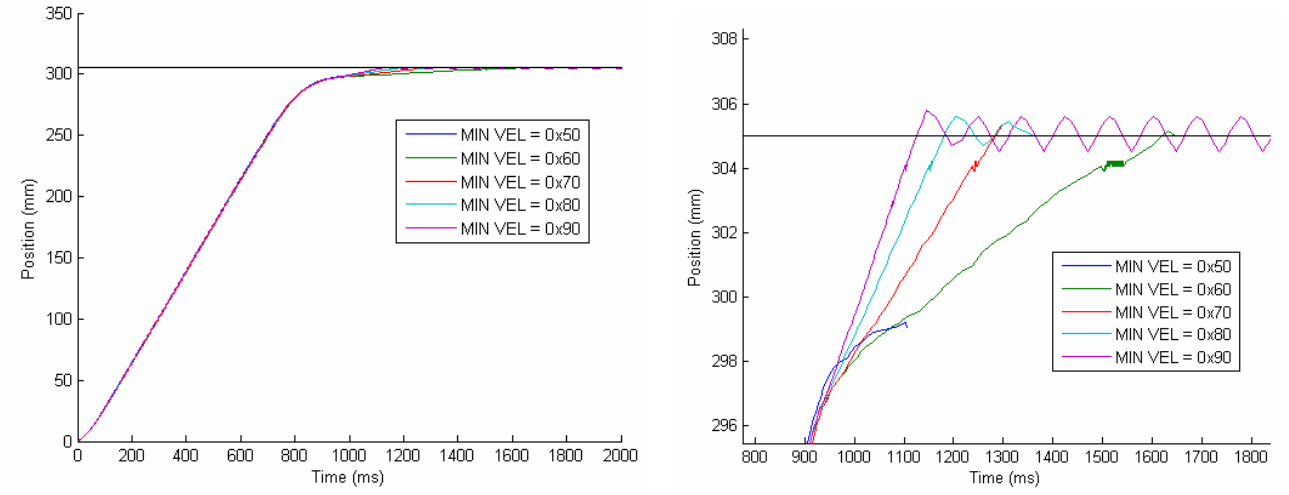


Figure 35 : Tuning of MIN_VEL

3.3.3.2.3 Tuning of K

The following test was done for MAX_VEL = 0x3FF and MIN_VEL = 0x80, with a reference value of 305. The value for K was 1, 2, 3, 4 and 5.

We can see on **Figure 36** that the difference is in the top part of the curve, when the error began small enough to have an output signal between MAX_VEL and MIN_VEL. For $K = 1$, the system

response is much too slow, and for $K = 4$ and $K = 5$ the overshoot is too big (~0.7% and 1.31% respectively).
For $K = 2$ and $K = 3$ the overshoot is the same (~0.26%), but the system response is faster with $K = 3$.

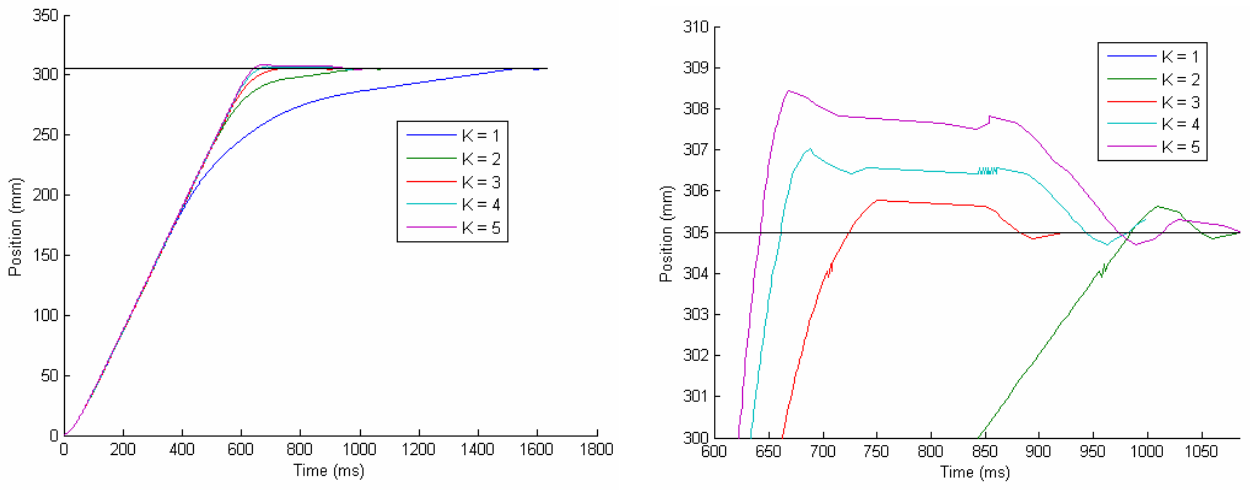


Figure 36 : Tuning of K

The best P controller was so obtained with the following parameters: MAX_VEL = 0x3FF, MIN_VEL = 0x80 and $K = 3$. This controller was used as a base for the PI controller.

3.3.3.3 PI controller

To improve the behaviour of a P controller, it is possible to add an integral part (I) to this controller, which thus becomes a PI controller.

The integral part corresponds to the accumulated sum of the errors in the process. So this part can become very big when the sign of the error does not change. It is why we need to reduce this part with a coefficient T_i, named integral coefficient.

A PI controller can be expressed as:

$$u(k) = K \cdot e(k) + \frac{1}{T_i} \sum e(k) \qquad (3\text{-}18)$$

In the controller program (controller.c, see appendix C), the integral coefficient T_i is modified using the variable TI, by the following relation :

$$T_i = 2^{TI} \qquad (3\text{-}19)$$

This relation was used to reduce the microcontroller calculation time, in the way that (3-19) can be done by a binary shifting (move the bits of a number to the right or the left), which is faster than a multiplication (or a division) in the microcontroller.

3.3.3.3.1 Basic PI controller

This PI controller was implemented in the microcontroller, and we can see on **Figure 37** the step response of the system with this controller.

The following test was done with the same parameters as before for the proportional part (MAX_VEL = 0x3FF, MIN_VEL = 0x80 and K=3), with a reference value of 152.5 (and not 305 as before to be sure that the cart will not hit a track end in case of oscillation). The value for *TI* was 8, 9 and 10.

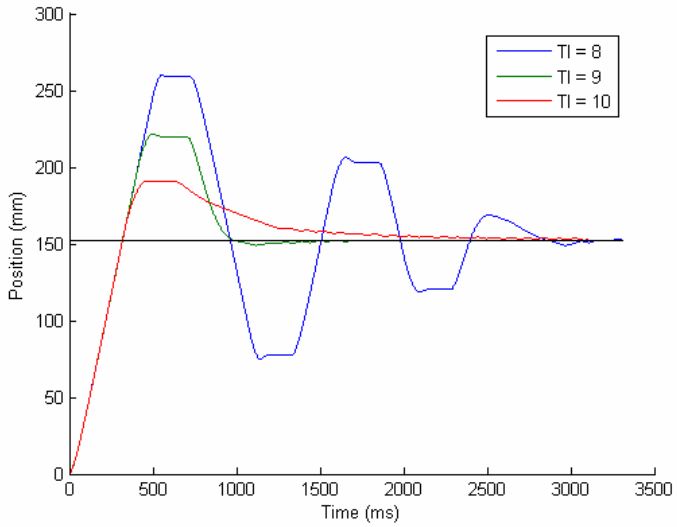


Figure 37 : Step response with a basic PI controller

The overshoot with this controller is much bigger than before (71% for $TI = 8$, 45% for $TI = 9$ and 26% for $TI = 10$). The reason of this behaviour is that the integral part is so big when the system reaches its reference position that it takes a long time to reduce and cancel it.

Of course, it is possible to reduce the overshoot by having a bigger *TI* . But in that case, to have a "correct" behaviour, *TI* would have been so big that the integral part would become negligible compared to the proportional part.

So, to keep the advantage of the PI controller and reduce the overshoot, it is possible to reset the integral part when the error becomes zero, or implementing an anti-windup correction, which limit the growth of the integral part.

3.3.3.3.2 PI controller with cancellation of the integral part

A way to reduce the overshoot we have with a basic PI controller is to reset the integral part when the error is null, hence the system does not have to wait until the integral part cancel itself.

The following test (**Figure 38**) was done with the same controller parameters, but with the integral part cancellation, with a reference value of 305. The value for *TI* was 16, 12, 10 and 7.

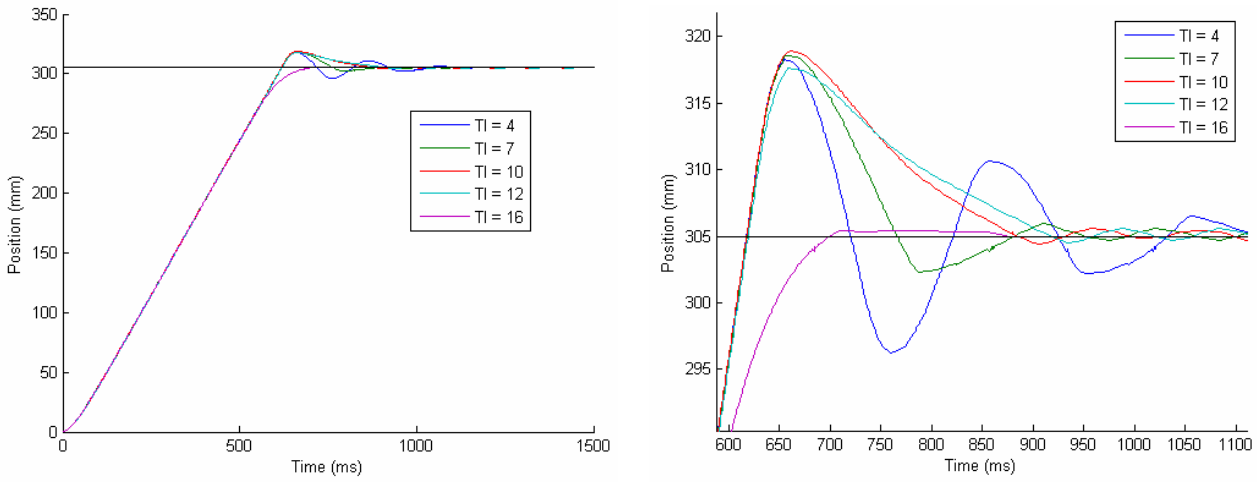


Figure 38 : PI controller with integral part cancellation

We can see on this figure that the results are better than the ones before (see **Figure 37**), with an overshoot of about 4.5% for $4 < TI < 12$. The curve which give the best response is obtained for $TI = 16$, but it is only because the integral part in this case is so small that the controller can be considered as a P controller.

But even if the overshoot is smaller than before, the results are not good enough. Indeed, we can see on the zoom plot that when the cart reaches the reference position the first time, the curve is linear at this time, meaning it is at the maximum velocity (MAX_VEL). This is due to the integral part which is strong enough to keep the output saturated even when the error begins very small, and it is called the "windup" effect [14].

3.3.3.3.3 PI controller with anti-windup correction

In order to remove this "windup" effect, we need to limit the integral action when the output saturates. Hence, the system will be able to cancel this integral part faster and so reduce the overshoot.

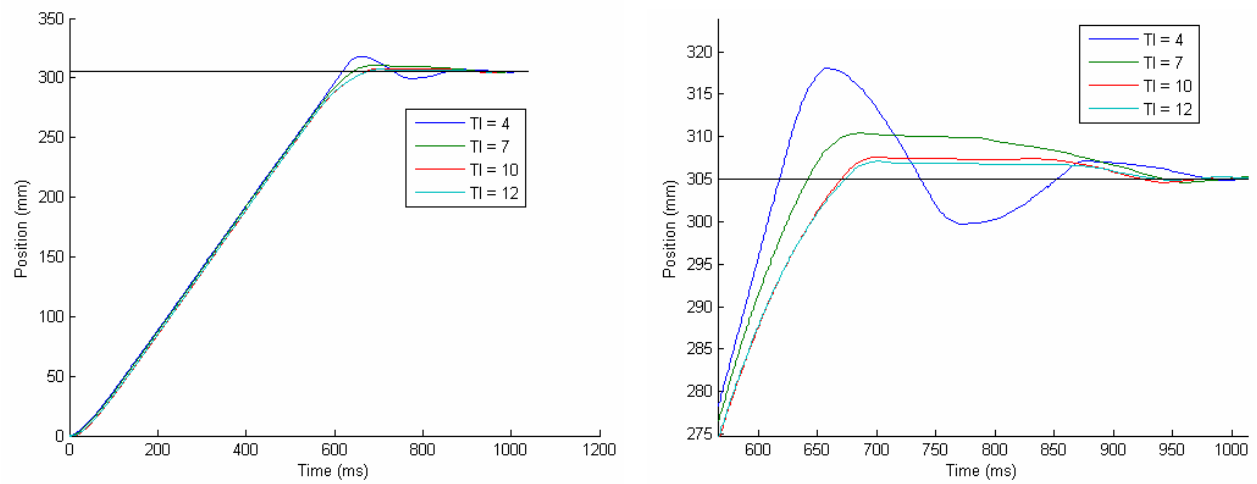


Figure 39 : PI controller and anti-windup correction

Figure 39 was obtained with an anti-windup correction for the integral part, for $TI = 4,\ 7,\ 10$ and 12.

We can notice that thanks to the anti-windup correction, the overshoot this time, is reduced as much as TI is big, to be around 0.65% for $TI = 12$, due to the speed that is lower that before when we reach the reference position (the integral action is smaller than before, so the output signal also). The best PI controller with anti-windup correction was obtained for $TI = 10$, which gave a response a little bit faster than for $TI = 12$, as we can see on **Figure 40**.

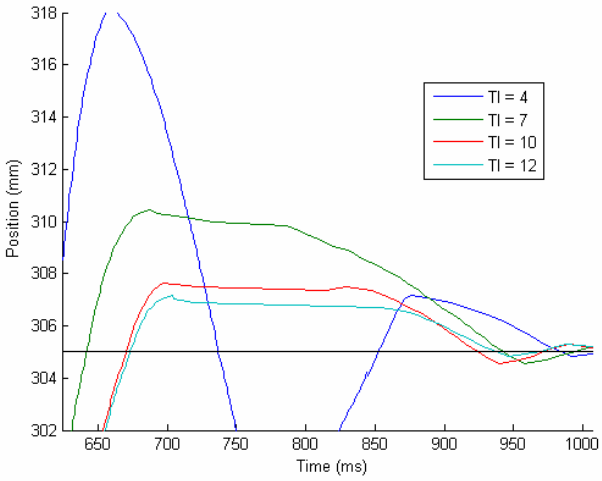


Figure 40 : PI controller and anti-windup correction (zoom)

We have seen so far that the results are much better with an anti-windup correction. But there is still a problem in our algorithm: the MIN_VEL saturation. Indeed, this saturation limit was necessary with the P controller to compensate the friction when the proportional part is too small to make the cart moving (i.e., when the error is small enough). But due to that saturation, we have a non-linear relation between the error and the output signal. So we need to remove it to have a linear relation.

3.3.3.3.4 PI controller with a linear error-output relation

With the PI controller described before, it is now possible to remove the MIN_VEL saturation limit. Indeed, if the output began too small to compensate the motors friction, the integral part will grow until the cart starts to move.

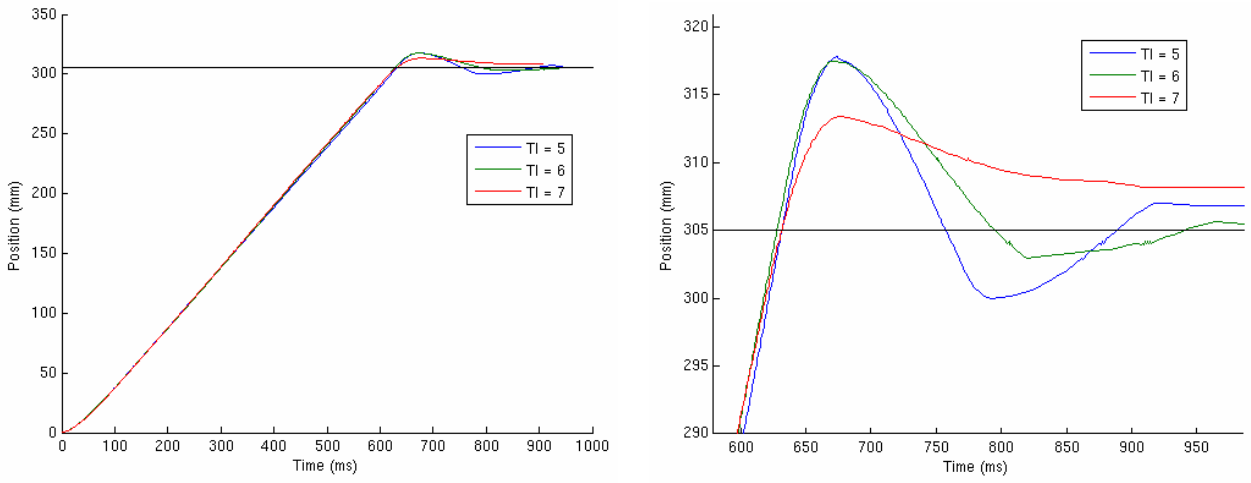


Figure 41 : PI controller with linear error-output relation

As shown on **Figure 41**, for $TI = 7$ the integral part is not strong enough and the cart never reaches its reference position (it will, but in a very long time). For $TI = 5$, we can see some oscillation

around the reference position, which is not very good. The best result with this PI controller was obtained for $TI = 6$.

3.3.3.4 Comparison of the controllers

We have seen in this part different ways to control the position. The first controller implemented was a P controller, and the best one was obtained with the following parameters: MAX_VEL = 0x3FF, MIN_VEL = 0x80 and $K = 3$.
The second controller was a PI controller with anti-windup correction, which gave the best results with the following parameters: MAX_VEL = 0x3FF, MIN_VEL = 0x80, $K = 3$ and $TI = 10$.
But to have a linear relation between the error and the output signal, and hence remove the non-linearity, we made a third controller, without the MIN_VEL saturation limit, which gave the best step response with the following parameters: MAX_VEL = 0x3FF, $K = 3$ and $TI = 6$.

Figure 42 show a comparison between these 3 controllers:

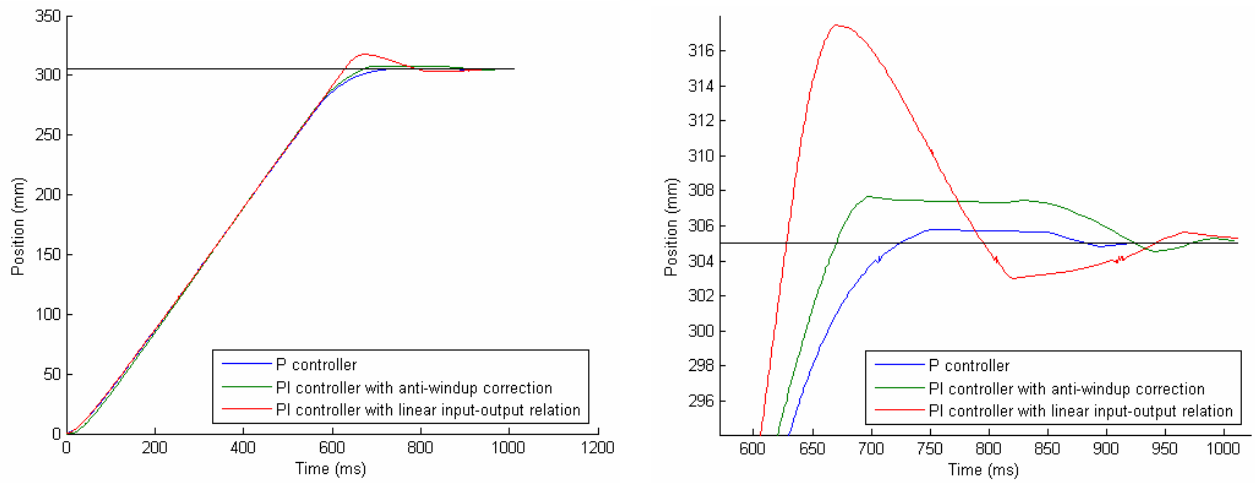


Figure 42 : Comparison of the controllers

We can see that the best controller we have is obtained with a P controller. But to remove the non-linearity (with a linear relation between error and output signal), we need to use the last PI controller we have seen, even if it behaviour is not the best one.

Moreover, for each controller we have an overshoot in the position control, which is not acceptable if we want to do some very precise work with the robot. We so need to remove this overshoot by some other control methods.

3.3.4 Velocity control

One explanation of the overshoot in the position control is that the velocity is too big when the track reaches it reference position, and due to the inertia of the system, the cart has not the time to brake before going past the reference position.

So in order to remove this, we need to implement a velocity control to improve the behaviour of the system, with first a P controller and finally a PI controller.

3.3.4.1 P controller

A P controller for the velocity was implemented in the same way of the P controller for the position. The proportional coefficient is the variable KV and was tested with the value 6 on **Figure 43**, 7 on **Figure 44** and 9 on **Figure 45**. The tests were done by sending a constant velocity of value 1000

(slow velocity), and after some time the reference changed and became 2000 (high velocity).

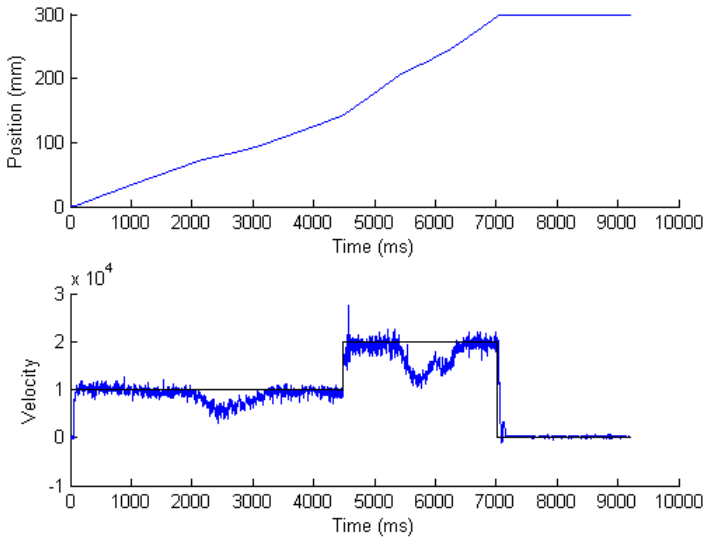


Figure 43 : P velocity controller for KV = 6

On this figure we can see that the velocity signal is controlled, but the proportional part is not strong enough to remove the track problems seen in section 3.3.2.

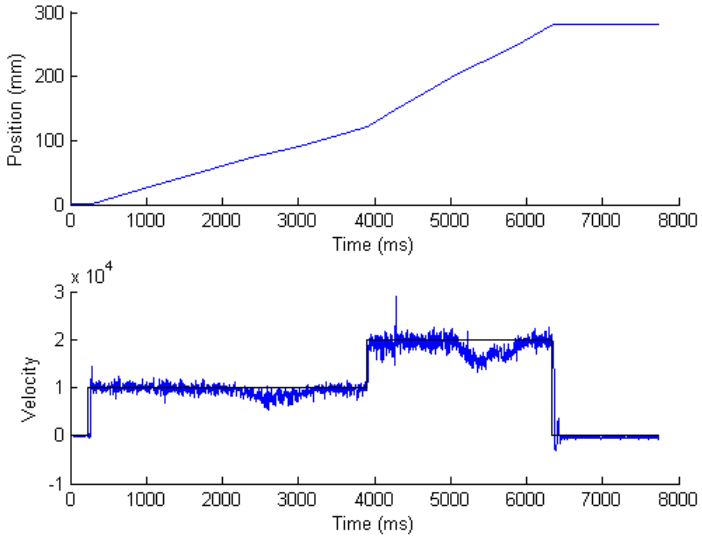


Figure 44 : P velocity controller for KV = 7

This time the controller is stronger, so we can see that the result is better, but we still have some problem around the position 250.

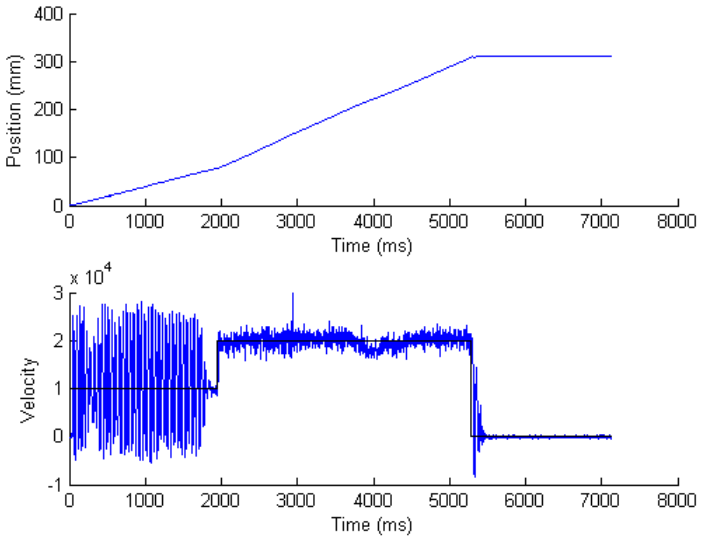


Figure 45 : P velocity controller for KV = 9

With $KV = 9$ the track problem around the position 250 is almost compensated, but this controller is too strong when the velocity is small, and so the system is unstable in that case.

So we have seen that the P controller is not good enough to control the velocity correctly, so we need to implement a PI controller which can give us better results.

3.3.4.2 PI controller

The PI controller was implemented as before for the position, with an anti-windup correction and a linear error-output relation.

The tests were done in the same way as before, using the precedent P controller for $KV = 7$, and adding an integral part where KIV was the integral coefficient and had the value 13 (**Figure 46**), 14 (**Figure 47**) and 15 (**Figure 48**).
The reference was set to 1000 only to see the behaviour in low velocities where the control is the more important. For high velocity the behaviour will be only better due to the inertia.

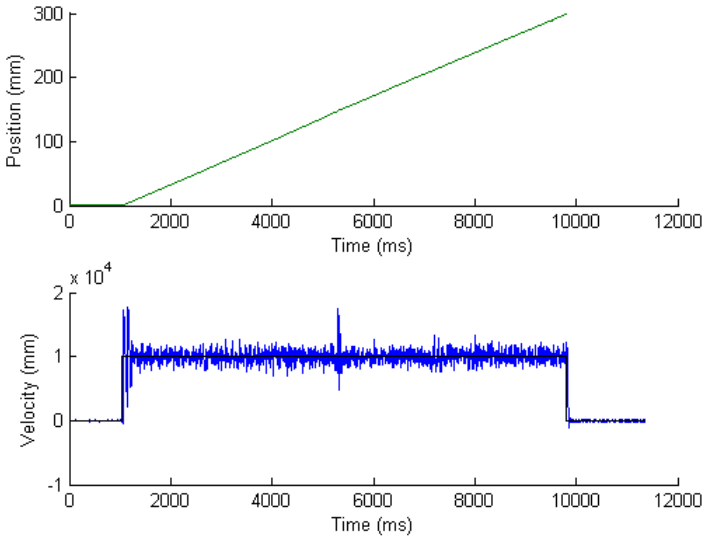


Figure 46 : PI velocity controller for KIV = 13

We can see on this figure that the result is much better that with a simple P controller. The integral part compensates the track problem when it is needed and hence helps to give a smooth velocity response. However, we can see some oscillation at the beginning, due to the integral part which have been saturated, but still need some time to be cancelled.

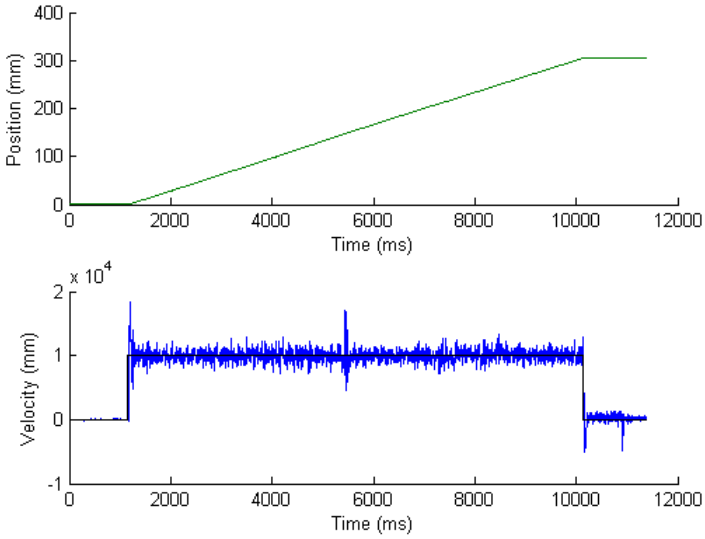


Figure 47 : PI velocity controller for KIV = 14

This time the integral part is smaller than before, which give less oscillation at the beginning, but still a little bit.

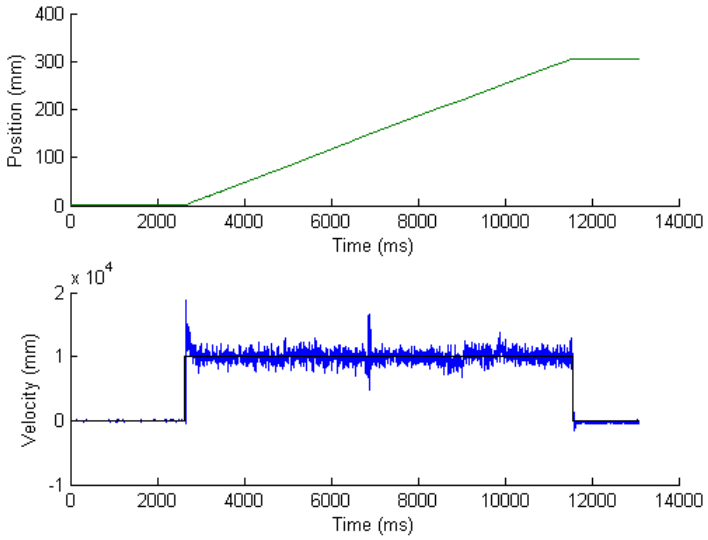


Figure 48 : PI velocity controller for KIV = 15

With this controller we haven't any oscillation at the beginning of the velocity plot, only an overshoot.

3.3.4.3 Results

We have seen in this part that a velocity controller is needed to improve the position control. No deep studies are needed to see that the best results were obtained with a PI controller with an anti-windup correction.

The controller that achieves the best of these characteristics is the PI controller with $KIV = 14$, as shown on **Figure 47**. Indeed, we have some oscillation, just what is needed to decrease the time response with keeping a stable response. So this controller was chosen to be used in the next step: the cascade controller.

3.3.5 Cascade controller

A cascade controller is the cascade connection of two controllers: one for the position, and the other one for the velocity, as shown on the following block diagram (**Figure 49**):

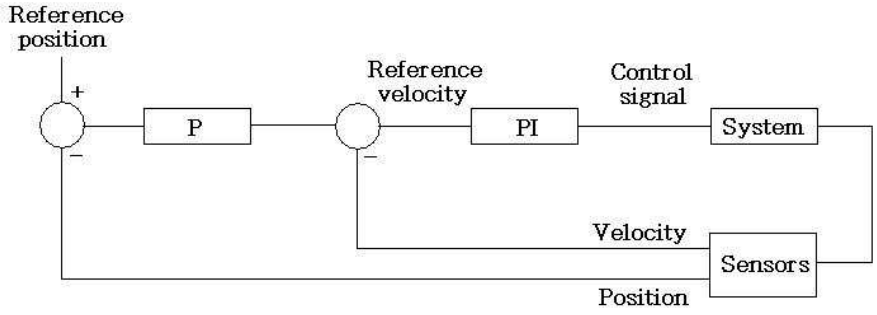


Figure 49 : Cascade controller block diagram

The position P controller will sent a velocity reference to the PI velocity controller depending on the position error. Hence, both of the position and the velocity are controlled.

The following tests were done with the previous PI velocity controller ($KV = 7$ and $KIV = 14$), and the P position controller was tuned with $K = 7$ (**Figure 50**), $K = 8$ (**Figure 51**), $K = 9$ (**Figure 52**), $K = 10$ (**Figure 53**) and $K = 11$ (**Figure 54**). The reference was set to 30 to simulate a small move, and after some time was set to 305.

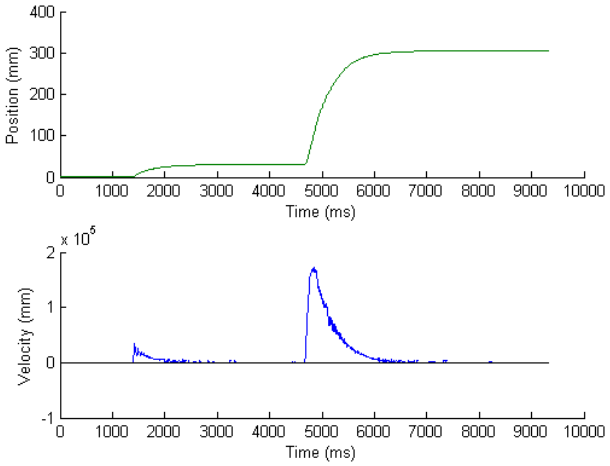


Figure 50 : Cascade controller for K = 7

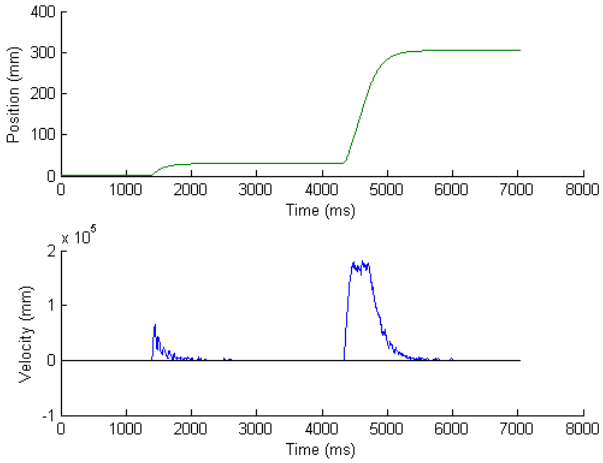


Figure 51 : Cascade controller for K = 8

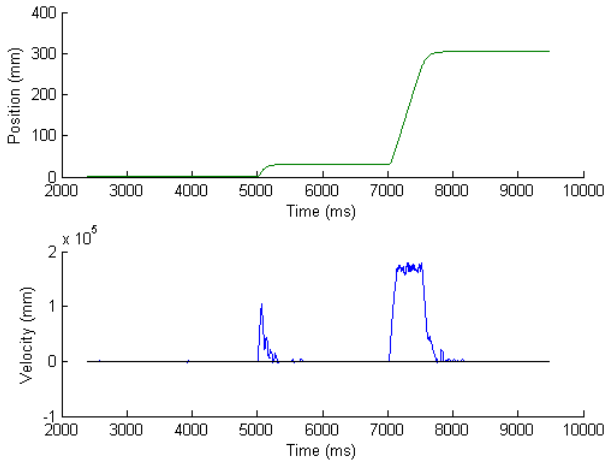


Figure 52 : Cascade controller for K = 9

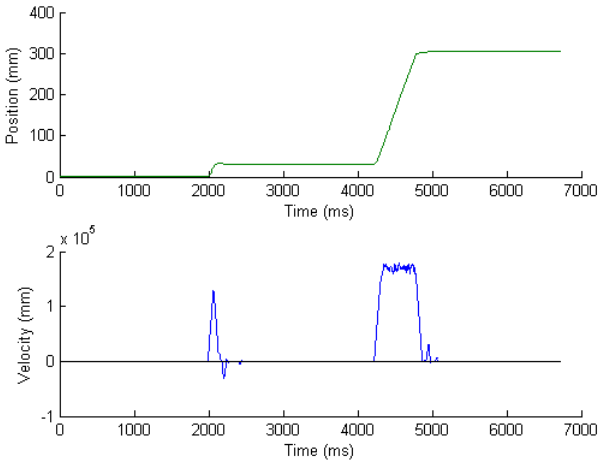


Figure 53 : Cascade controller for K = 10

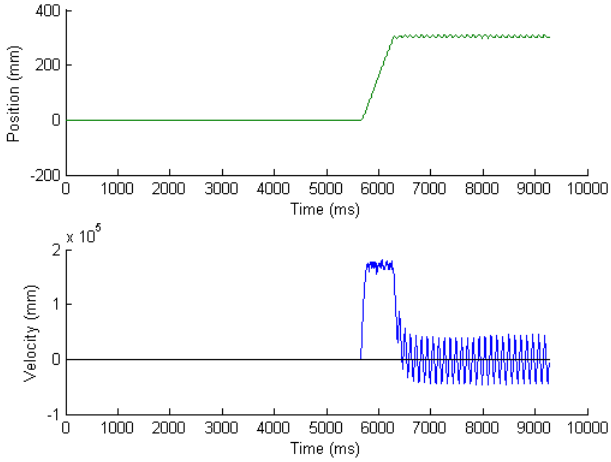


Figure 54 : Cascade controller for K = 11

If we look at all these controllers, we can directly remove the cascade controller for $K = 11$, which give an unstable response. After, if we look closely to the **Figure 53** for $K = 10$, we can see an overshoot in the response for the small reference, characterized by the sign change of the velocity around the reference position.
So this controller is good when the reference position is far from the actual position, but a little bit too strong when the reference position is quite close to the actual position.

Finally, all the other controllers are good, giving a position control without any overshoot. The best one is obtained for $K = 9$.

3.3.6 Comparison between the controllers

We have seen so far two controllers which allow a position control: the position P controller and the cascade controller with inner PI velocity controller and outer P position controller.

Figure 55 show a comparison between these filters:

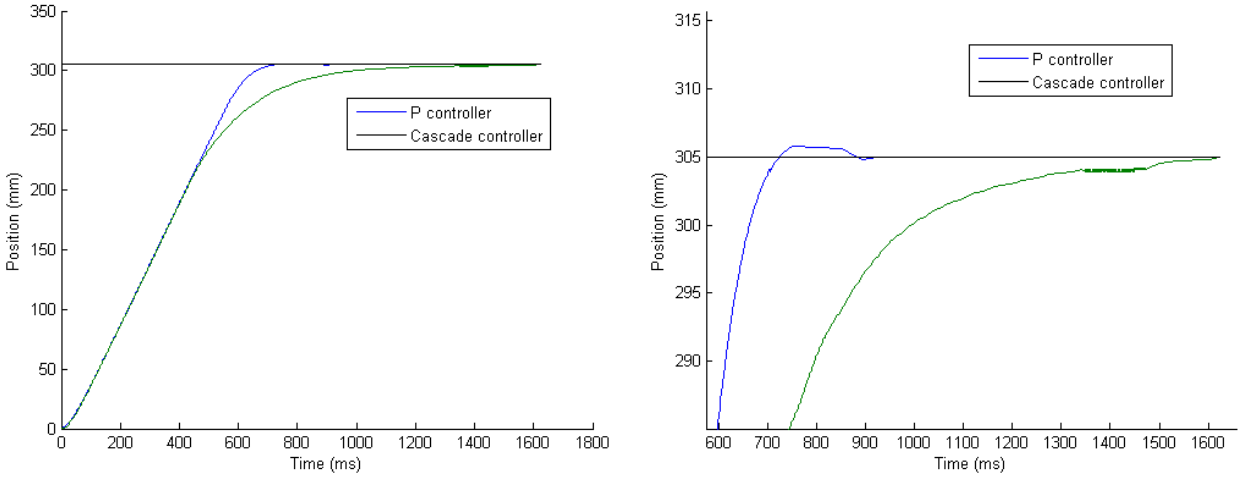


Figure 55 : Comparison P controller – cascade controller

We can see on his figure that the P controller is faster than the cascade controller, but give an overshoot on the response which is removed with the cascade controller.

3.3.7 Evaluation

We have seen in this part that in order to use the robot, we needed a position control. After different control methods, it appeared that we had two ways to control the system in position: the P controller and the cascade controller with inner PI velocity controller and outer P position controller.

In order to have a very precise position control, an overshoot in the position response is not allowed, even if the time response is a bit longer.

So, as a conclusion, we can say that to have a position control without any overshoot, a cascade controller need to be used, and it gives good results.

Moreover, it would have been interesting to implement some different control procedures, not just the heuristic P or PI controllers. We could have implemented a LQG (Linear Quadratic Gaussian) control, for example, which use an optimization-based approach. But we hadn't the time during this master thesis to try such control procedures.

3.4 Backlash compensation

3.4.1 Introduction

As said in section 2.2.2, a model of the robot was created with 2 motors in order to make backlash compensation. Indeed, when we have 2 motors, we can use both in the same direction to increase the maximum speed, but we can also use one in one direction, and the other in the other direction.

Of course, if we put the same force on each motor but not in the same direction, the track will not move, and worse, it can damage the system. But we can use one motor in "full" power to move the system, and the other with a small PWM in the opposite direction. Hence, it will create a small torque on the other direction of the motion, which will be used as a brake.

With this simple method, we have always a tooth of the gear wheel of both motors which is in contact with a tooth of the toothed belt. Hence, immediately after a changing direction order, the track will be able to move, without any backlash.

But as simple as this method is, it had to be implemented on the system. In order to find which motor ran in which direction, and to check if both motors had the same behaviour, we made a comparison between each motors, in function of the controller used and the direction of the track.
Figure 56 show the Matlab/Simulink controller structure used for these tests.

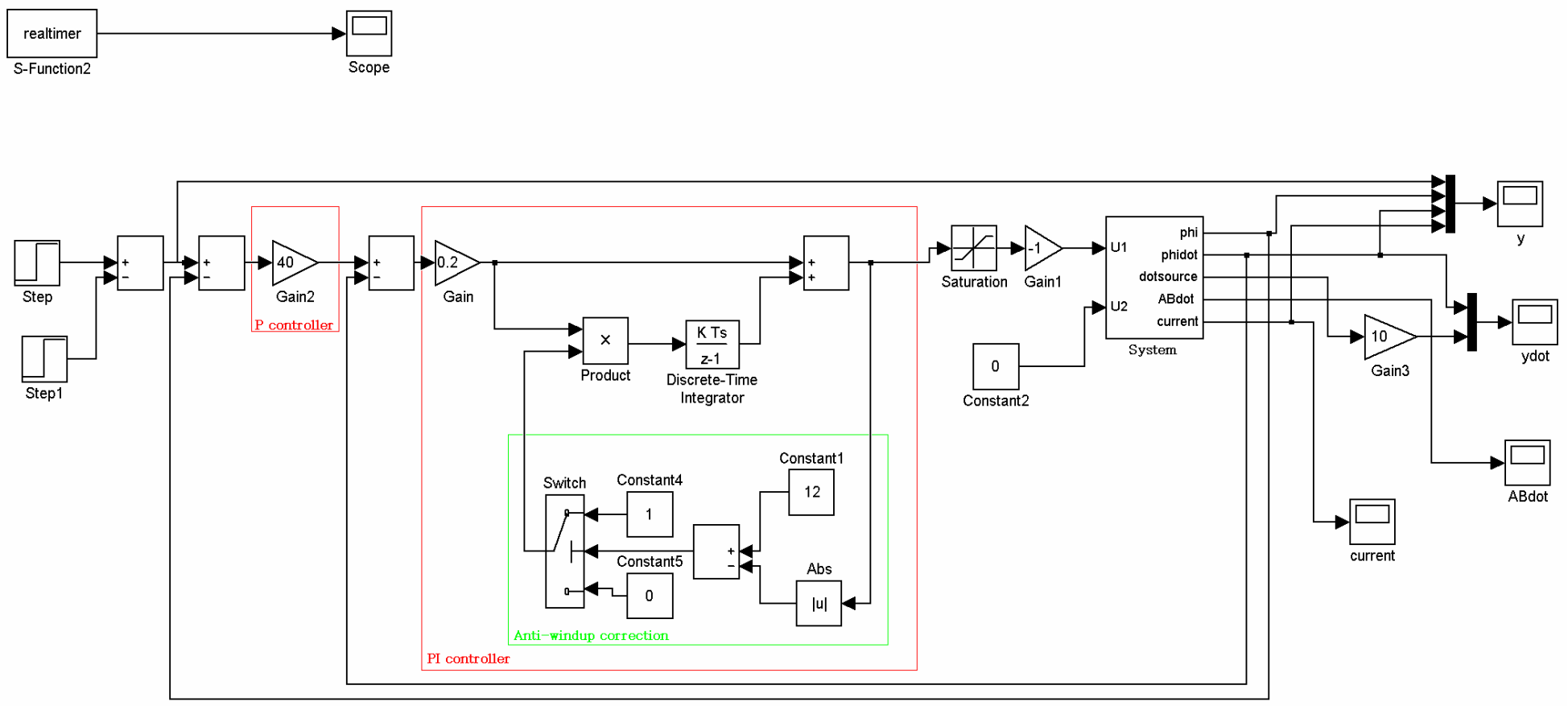


Figure 56 : Simulink controller structure

3.4.2 Motors comparison

The motors which made the table robot are the same, in a theoretical point of view. But in reality, they have not exactly the same behaviour. So we need to find what are the differences, and see if they will have an influence in the backlash control or not.

And, moreover, even if they are the same, we need to find which motor to use in which direction. Indeed, just by looking at the robot, it is impossible to say if the motor on the right (right and left define from **Figure 13**) has to be used to move the track on the right or on the left.

As we just wanted to compare the motors behaviour in function of the direction of the track, a basic P position controller was enough. **Figure 57** shows the steps response of the system, for a reference to 50 for the first step, and 10 for the second one. Hence, both directions were tested. Their control signals were also compared.

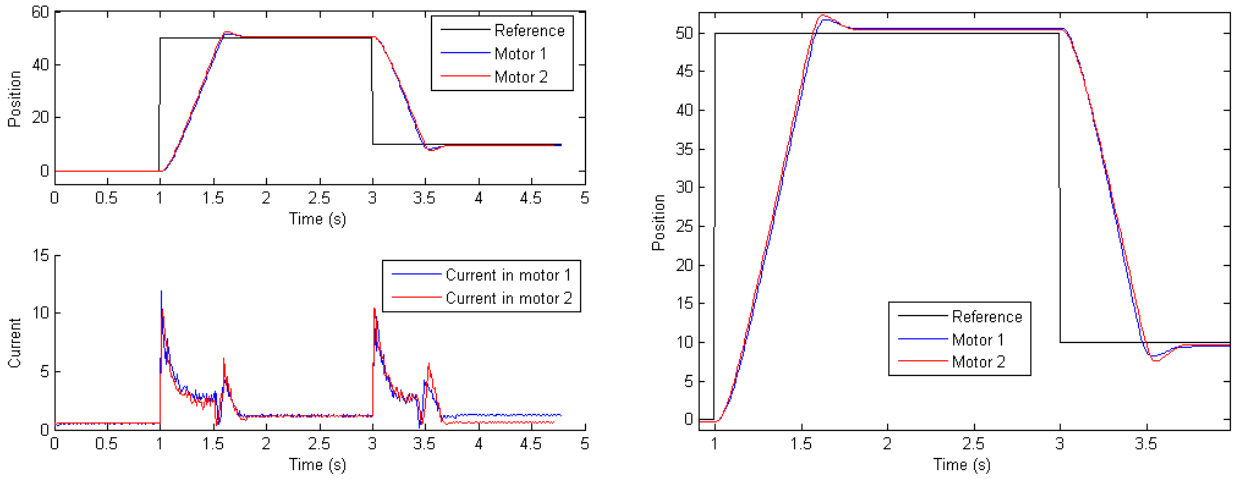


Figure 57 : Motor comparison for a P position controller (P = 2)

We can see on this figure that, as expected, both motors have not the same behaviour. In a general way, we can notice that motor 2 have a bigger overshoot than motor 1. If we look at the rising saturation part of the curve (almost between 1 and 1.5 s), we can see that the current in motor 2 is less than the current in motor 1. So, we can deduce that it is easier for motor 2 to move the track than motor 1 when the track goes to the left.

When the track goes to the right (from reference position 50 to reference position 10), we can notice that motor 1, even if it start a bit later than motor 2, reach the reference position first. Moreover, in the downward saturation part of the curve, we see that the slope is not the same, even if the current on the motors are equivalent. We can deduce here that it is easier for motor 1 to move the track than motor 2 when the track goes to the right.

Hence, we can conclude that for an optimal use of the motors in the backlash compensation, motor 1 have to be used when the track goes to the right and motor 2 when the track goes to the left.

3.4.3 Backlash compensation mechanism

As we have seen in the previous section, one motor are better than the other for one direction. In order to create backlash compensation, we need to use one motor in “full” power and the other one in brake (as explained in 3.4.1). So we can, for example, define the brake in function of the position error. Thus, when the error is big enough, we don’t use the brake, and when the error start to be smaller, the brake is used as much as the error is small.

This backlash compensation mechanism will work, but can still be improved, as shown on **Figure 58**: when the error is big enough, both motors are used to move the track, and to increase the maximum speed, and when the error is smaller than a limit, only one motor move the track and the other act as a brake. When the error is zero, both motor are in brake, but in opposite directions.

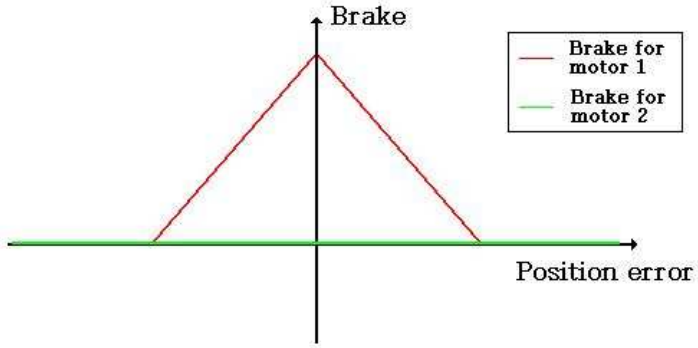


Figure 58 : Brake signal when the track goes to the left

Hence, above an error limit (chosen at 48, corresponding to an error of 7.5mm between the track position and the position reference) both motor control signals are used.
Below the error limit, depending on the direction, one motor keeps its full control signal, and the other motors brake signal increases (to reach the value BRAKE in the *controller.c* program when the error is zero).

3.4.4 Results

Figure 59 shows the backlash compensation algorithm in action for a cascade controller. The parameters for the cascade controller are the same as in section 3.3.6.

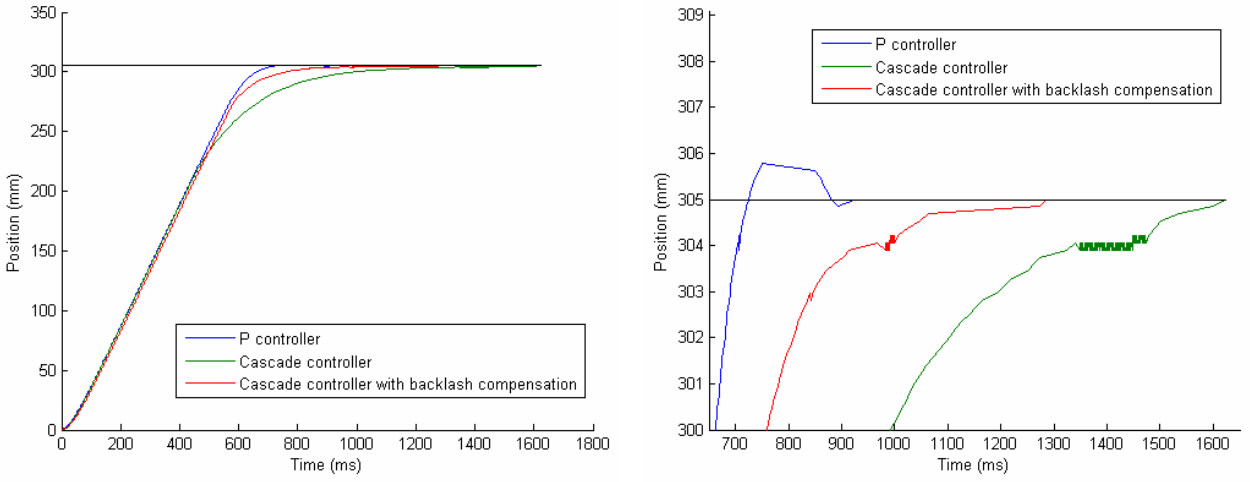


Figure 59 : Comparison of the controllers with and without backlash compensation

We can see on this figure that for the same parameters, the cascade controller with backlash compensation is much better than the one without. But it still is slower than the P controller.

3.4.5 Evaluation

As we have seen before, backlash in the gear wheels introduces discontinuity, uncertainty and makes accurate control of the tool centre point difficult. So it needs to be compensated.

The way used in this master thesis work was to use 2 motors, and to control them in order to use one in brake when it is necessary. So with that mechanism, the cart had always one tooth of the gear in contact with on tooth of the toothed belt, so the backlash was reduced.

Moreover, we have also seen that this backlash compensation mechanism improved the global behaviour of the system, by giving a faster position step response, still without any overshoot.

4 EVALUATION AND FUTURE WORK

We have seen in this report the control of a Gantry-Tau structure, with some basic P and PI controllers, directly implemented on a microcontroller. In order to give a position step response without overshoot, we have done and tuned a cascade controller with inner PI velocity controller and outer P position controller.

Moreover, thanks to the robot model with 2 motors we have made a backlash compensation, which also improved the step time response of the system.

But as the memory on the microcontroller was limited, we were not able to implement a Kalman filter algorithm for the velocity signal. This filter was only tested using Matlab/Simulink and gave good results.

So, in order to improve the global behaviour of the system, it will be interesting to add some extra memory to the microcontroller to implement the Kalman filter algorithm.

Moreover, it will be judicious to implement a current control, in the same way that was implemented the velocity control.

Another thing that can be tested is to implement different control procedures, such as LQG control for instance, which use an optimization-based approach.

Finally, as the control was done for one arm, the last thing to implement is the communication between the microcontrollers of each arm, in order to make the full table robot.

So, as a conclusion, we can say that the control of the Gantry-Tau structure implemented gave good results, but not as good as expected due to microcontroller memory limitation, and can still be improved.

REFERENCES

[1] Craig, J. J., 2005. “Introduction to Robotics”. Upper Saddle River, Prentice Hall, New Jersey.

[2] Bonev, I. A., 2003. “The True Origin of Parallel Robots”. ParalleMIC Reviews.

[3] Bolmsjö, G., 2005. “Introduction to robotics”. Seminar presentation, Department of Mechanical Engineering, Lund Institute of Technology, Lund, Sweden.

[4] Pieper, D. L., 1968. “The kinematics of manipulators under computer control”. Ph.D. thesis, Department of Mechanical Engineering , Stanford University, Stanford, USA.

[5] Stewart, D., 1965-66. "A platform with six degrees of freedom". Proceedings of the IMechE, pp. 371-386.

[6] Johannesson, L., V. Berbyuk and T. Brogårdh, 2003. “Gantry-Tau, a new Three Degrees of Freedom Parallel Kinematic Robot”. Proceedings of the Mekatronikmöte2003, pp. 1-6, Göteborg, Sweden.

[7] St Clair, M., 2004. “Build a Prototype Parallel Robot”. ITEE demonstration day, School of Information Technology and Electrical Engineering, The University of Queensland, Australia.

[8] Atmel Corporation, 2003. “AVR ATmega16 microprocessor manual”.

[9] Olsson, G., and G. Piani, 1992. “Computer System For Automation and Control”. Prentice Hall International, London.

[10] Venema, S.C., 1994. “A Kalman Filter Calibration Method for Analog Quadrature Position Encoders”. MSEE thesis Th010, Department of Electrical Engineering, University of Washington, Washington, USA.

[11] Welch, G. and G. Bishop, 2004. “An Introduction to the Kalman Filter”. Ph.D. thesis TR 95-041, Department of Computer Science, University of North Carolina, Chapel Hill, USA.

[12] Bryson, A.E. Jr., 1993. “Dynamic Optimization with Uncertainty”. Pre-publication draft, Department of Aeronautics and Astronautics, Stanford University, Stanford, USA.

[13] Johansson, R., 1993. “System Modeling and Identification”. Englewood Cliffs, Prentice Hall, New Jersey.

[14] Åström, Karl J. and Tore Hägglund, 1988. “Automatic Tuning of PID Controllers”. Instrument Society of America, Pittsburgh, USA.

APPENDIX A

Kalman filter function

```
function [sys, x0] = kalmanfiltern(t,x,u,flag,A,C,V,W,xi,h)

% S-function for kalman filter

% The system is a state-space system

% x(k+1) = A x(k) + B u(k) + w(k)
%   y(k) = C x(k) + D u(K) + v(k)

% where w ~ N(0,W) meaning w is gaussian noise with covariance W
%       v ~ N(0,V) meaning v is gaussian noise with covariance V

% The states of this S-function are x = [xhat ; Phatv]
%                 where xhat is the predicted estimate xhat(k)
%                       Phatv is a vector containing the elements of the
%                       predicted covariance matrix Phat

% The parameters of this S-function are A, C, V, W, xi, h
%                 where A is the state transition matrix
%                       C is the observation matrix
%                       V is the measurement noise covariance
%                       W is the process noise covariance
%                       xi is the initial state
%                       h is the sample time of the system

% The outputs are the states and the input is the measurement u

% calculate the number of state
n = length(A);

if flag == 2,    % return next state

  % create xhat
  xhat = x(1:n);

  % prevent unexistant value (at time 0)
  if isnan(u)
    y = 0;
  else
    y = u;
  end

  % create Phat matrix
  ni = 1;

  for i=1:n
      Phat(i,1) = x(n + ni);
      id(i) = n + ni;
      ni = ni + i;
  end


  for j = 2:n
      ni = 1;
      for i=1:j-1
          Phat(i,j) = x(id((j-2)*n + i) + j-1);
          id((j-1)*n + i) = id((j-2)*n + i) + j-1;
          ni = ni + i;
```

```
        end
        for i=j:n
            Phat(i,j) = x(id((j-2)*n + i) + 1);
            id((j-1)*n + i) = id((j-2)*n + i) + 1;
            ni = ni + i;
        end
    end

    % compute kalman algoritm
    % time update
    xbar = A*xhat;
    P = A*Phat*A' + W;

    % measurement update
    temp = C*P*C' + V;
    K = P*C'*inv(temp);
    Phat = P - K*temp*K';
    xhat = xbar + K*(y - C*xbar);

    % Construct the state vector for output : x = [xhat ; Phatv]
    x(1:n) = xhat;

    % position in the state vector
    ind = n+1;
    for i=1:n
        for j=1:i
            x(ind,1) = Phat(j,i);
            ind = ind + 1;
        end
    end

    % return output
    sys = x;

elseif flag == 3,    % return output
    sys = x;

elseif flag == 4,    % return next sample time
    sys = t + h;

elseif flag == 0,    % return sizes

    % calculation of the number of states and the number of outputs
    n = length(A);
    nb_states = n*(n+3)/2;
    nb_outputs = nb_states;

    sys = [0 nb_states nb_outputs 1 0 0];
    x0  = xi;

else     % no need to return anything
    sys = [];

end
```

APPENDIX B

Java programs

B.1: SerialIO.java

```
import java.io.IOException;
import java.io.*;

public class SerialIO implements Serial2002ChannelHandler
{
	Serial2002Driver serial;
	FileWriter f;
	exGui gui;
	long init = 0;
	int init2 = 0;
	long reference = 0;
	int limit_vel = 170000;
	int position_stop = 350;

	// constructor
	public SerialIO(String port, int baudrate) throws Exception {
		// initialize serial communication
		serial = new Serial2002Driver(port, this, baudrate);

		// create the measurement Matlab .m file
		f= new FileWriter("curve.m");
		f.write("array = [");

		// create the gui
		gui = new exGui(this);
	}

	// send the reference position to the microcontroller
	public void setPositionNumber(int value){
		try {
			serial.setChannel(21,value);
		}
		catch (IOException e) {}
	}

	// send the reference velocity to the microcontroller
	public void setVelocity(int value){
		value += limit_vel;
		if (value < 0) value = 0;
		else if (value > limit_vel*2) value = limit_vel*2;
		reference=value - limit_vel;
		try {
			serial.setChannel(22,value);
		}
		catch (IOException e) {}
	}
```

```
// serialio callbacks
public void handleBit(int index, int value) throws IOException {
	System.out.println("Digital channel #" + index + ": received " + value);
}

// catch channel sent by the microcontroller
public void handleChannel(int index, long value) throws IOException {
	// position signal
	if(index == 7){
		f.write(" " + value);
		// in case of velocity control, stop the track before it collides the end
		if (value > position_stop) setVelocity(0);
	}

	// velocity signal
	else if (index == 31) {
		if (init2 == 0) {
			f.write(" 0");
			init2 = 1;
		}
		else f.write(" " + value);
	}

	// time signal
	else if  (index == 30){
		if (init == 0)init = value;
		f.write(" " + (value-init));
		f.write(" " + (reference) + ";\n");
	}

	// print the other channel on the screen
	else System.out.println("Analog channel #" + index + ": received " + value);
}

// finalisation of the measurement Matlab .m file
public void endFile(){
	try{
		f.write("]; \n array(:,3)=array(:,3)/4.612;\n");
		f.write("\n c=[305*ones(size(array,1),1),array(:,3)];\n");
		f.write("subplot(2,1,1)\n");
		f.write("hold all\n");
		f.write("plot(array(:,3),array(:,1)) \n plot(c(:,2),c(:,1),'k') \n");
		f.write("plot(array(:,3),array(:,1)) \n");
		f.write("xlabel('Time (ms)'); \n ylabel('Position (mm)');\n");
		f.write("title('Position');\n");
		f.write("subplot(2,1,2)\n");
		f.write("hold all\n");
		f.write("plot(array(:,3),array(:,2),array(:,3),array(:,4),'k') \n");
		f.write("xlabel('Time (ms)'); \n ylabel('Velocity');\n");
		f.write("title('Velocity');\n");
```

```
                        f.flush();
                }
                catch (IOException e){}
        }

        // entry point
        public static void main(String[] args) {
                try {
                        new SerialIO("/dev/ttyS0", 115200);
                } catch (Exception e) {}
        }
}
```

B.2: exGUI.java

```
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;


public class exGui extends JFrame implements ActionListener {
	private static final long serialVersionUID = 0x10001L;

	SerialIO sioex;


	// gui elements
	Integer steps = new Integer(0);

	JLabel positionLabel = new JLabel("Enter the reference position (mm) :    ");
	JTextField positionField = new JTextField(20);
	JLabel velocityLabel = new JLabel("Enter the reference velocity :");
	JTextField velocityField = new JTextField(20);
	JLabel stopLabel = new JLabel("Stop the measurement   ");
	JButton stopButton = new JButton("Stop");

	// constructor
	public exGui(SerialIO sioex) {
		this.sioex = sioex;

	addWindowListener(new WindowAdapter() {
		public void windowClosing(WindowEvent e) {
			System.exit(0);
		}
	});

		positionField.addActionListener(this);
		velocityField.addActionListener(this);
		stopButton.addActionListener(this);

		// layout
		Container content = getContentPane();
		GridBagLayout layout = new GridBagLayout();
		content.setLayout(layout);
		GridBagConstraints c = new GridBagConstraints();
		c.fill = GridBagConstraints.HORIZONTAL;

		c.gridx = 0;
		c.gridy = 0;
		c.insets = new Insets(10,0,0,0);
		content.add(positionLabel,c);
		c.gridx = 1;
		c.gridy = 0;
```

```
                content.add(positionField,c);
                c.gridx = 0;
                c.gridy = 1;
                content.add(velocityLabel,c);
                c.gridx = 1;
                c.gridy = 1;
                content.add(velocityField,c);
                c.gridx = 0;
                c.gridy = 2;
                content.add(stopLabel,c);
                c.gridx = 1;
                c.gridy = 2;
                content.add(stopButton,c);
            setSize(480, 200);
            setVisible(true);
        }


        // gui callbacks
        public void actionPerformed(ActionEvent e) {
                if (e.getSource() == positionField)
sioex.setPositionNumber(steps.parseInt(positionField.getText()));
                if (e.getSource() == velocityField)
sioex.setVelocity(steps.parseInt(velocityField.getText()));
                if (e.getSource() == stopButton) sioex.endFile();
        }
}
```

APPENDIX C

Microcontroller programs

C.1: Admanager.c

```
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <avr/io.h>
#include "sme.h"
#include "observer.h"
#include "requester.h"
#include "controller.h"

// counts up at each AD interrupt
volatile uint32_t adm_time;

// current channel used to send the signals
volatile static int8_t adm_curr_channel;

// current client (use to communicate with the master avr)
volatile static int8_t adm_curr_client;

// place left in the encoder conversion queue
volatile static uint8_t adm_client_slots_left;
static uint8_t adm_client_slots[NUM_ADM_CLIENTS];

// peek to be able to decide next channel to sample
static void adm_peek_value(uint8_t channel, uint16_t value)
{
  observer_peek_AD_value(channel, value);
  requester_process_AD_value(channel, value);
}

// ask for next channel
static int8_t adm_client_get_next_channel(int8_t client)
{
  switch (client) {
    case ADM_CLIENT_OBSERVER: return observer_get_next_AD_channel();
    case ADM_CLIENT_REQUESTER: return requester_get_next_AD_channel();
    default: return -1;
  }
}

// internal
static uint8_t adm_get_next_channel()
{
  int8_t client = adm_curr_client;
  int8_t channel = -1;
  int8_t num_restarts = 0;

  // find a client that wants an AD conversion
  do {
    if (adm_client_slots_left > 0) {
```

```
      adm_client_slots_left--;
      channel = adm_client_get_next_channel(client);
      if (channel >= 0) break;   // we have a request
    }

    // skip to next client
    client++;
    if (client >= NUM_ADM_CLIENTS) client = 0;
    adm_client_slots_left = adm_client_slots[client];

    if (client == 0) {
      num_restarts++;
    }
  } while (num_restarts < 2);

  adm_curr_client = client;
  if (channel >= 0) return channel;
  else return 0xfe;   // Vbg, -2
}

// interrupt from the sensors signals
SIGNAL(SIG_ADC)
{
  // read AD value
  uint16_t value = ADCW;

  // set next channel
  adm_peek_value(adm_curr_channel, value);
  int8_t next_channel = adm_get_next_channel();
  ADMUX =  0x40 | (next_channel & 0x1f);
  ADCSR = (1<<ADEN) | (1<<ADSC) | (1<<ADIE) | ADC_DIVIDER_EXP; // necessary?

  // update of the integral part of the controller
  absIntPart = absIntPart + velocity_error;

  adm_curr_channel = next_channel;
  adm_time++;
}

// initialisation of all the parameters
void admanager_init()
{
  adm_time = 0;
  adm_curr_channel = -1;
  adm_curr_client = -1;
  adm_client_slots_left = 0;

  int i;
  for (i=0; i < NUM_ADM_CLIENTS; i++) {
    adm_client_slots[i] = 1;
  }
```

```
  // portar
  ADMUX = 0x5e;        // Vcc Vref, right adjust, read 1.23V (Vbg)
  ADCSR = (1<<ADEN) | (1<<ADSC) | (1<<ADIE) | ADC_DIVIDER_EXP;
}

// set a slot number to each client
void admanager_set_client_slots(uint8_t client, uint8_t num_slots)
{
  if (num_slots >= NUM_ADM_CLIENTS) return;
  adm_client_slots[client] = num_slots;
}

// define the configuration
void admanager_handle_conf_command(uint32_t command)
{
  uint8_t client = command & 3;
  uint32_t slots = command >> 2;
  if (client >= NUM_ADM_CLIENTS) return;

  if (slots < 255) {
    admanager_set_client_slots(client, slots);
  }
}
```

C.2: Admanager.h

```
#ifndef __admanager_h
#define __admanager_h

#define NUM_ADM_CLIENTS      3
#define ADM_CLIENT_OBSERVER  0
#define ADM_CLIENT_REQUESTER 2

void admanager_init();
void admanager_set_client_slots(uint8_t client, uint8_t num_slots);
void admanager_handle_conf_command(uint32_t command);

extern volatile uint32_t adm_time;

#endif
```

C.3: Channel.h

```
#ifndef __channels_h
#define __channels_h

// AD channels

#define AD_ENCODER_A_CHANNEL              0
#define AD_ENCODER_B_CHANNEL              2
#define AD_DOT_A_CHANNEL                  1
#define AD_DOT_B_CHANNEL                  3
#define AD_CURRENT_CHANNEL(motor)         6

// channel classification
#define AD_CHANNEL_AXIS(channel)       0
#define AD_IS_OBSERVER_CHANNEL(channel) (((channel) >= 0) && ((channel) <= 3))
#define AD_CHANNEL_ENCODER(channel)    (((channel) & 2) >> 1);
#define AD_IS_ENCODER_CHANNEL(channel)  ((channel & 1)==0)
#define AD_IS_DOT_CHANNEL(channel)     ((channel & 1)==1)

// serialio channels

//   analog channels

//     input
#define MOTOR_INPUT_CHANNEL(motor)

//     output
#define POSITION_OUTPUT_CHANNEL  0
#define ENCODER_A_OUTPUT_CHANNEL 1
#define ENCODER_B_OUTPUT_CHANNEL 2
#define CURRENT_OUTPUT_CHANNEL(motor)   3
#define VELOCITY_OUTPUT_CHANNEL  4

#define DOT_A_OUTPUT_CHANNEL    5
#define DOT_B_OUTPUT_CHANNEL    6

#define CONFIGURATION_CHANNEL        31


//     calibration interface
#define RESET_MESSAGE_CHANNEL         9
#define RESET_MESSAGE               42

#define OBSERVER_CONF_CHANNEL        11
#define ADM_CONF_CHANNEL             12
#define CALIBRATION_CHANNEL          13
```

```
//   digital channels

//     output
#define OVERTEMP_DOUTPUT_CHANNEL     0

#define CHANGE_DESIRED_POSITION_CHANNEL  21
#define DESIRED_VELOCITY                 22

#endif
```

C.4: Controller.c

```
#include "controller.h"

#define K               9           // Proportional position coeficient
#define KV              7           // Proportional velocity coeficient
#define KIV             14          // Integral velocity coeficient
#define MAX_VEL         170000      // Velocity limit
#define MAX_FORCE       0x3ff       // Force limit
#define SAT_LIMIT       0x250       // Saturation limit for the antiwindup
#define BRAKE           0x70        // Brake limit

volatile int64_t absIntPart;        // Current integral part
int64_t lastAbsIntPart;             // Precedent integral part
int64_t intPart;                    // Integal part / 2^TI
int32_t u_position;                 // Position output (velocity reference)
int32_t position_error;             // Position error (current position - desired position)
int32_t velocity_error;             // Velocity error
int32_t u_velocity;                 // Velocity output

// initialisation of the parameters
void controller_init()
{
        absIntPart = 0;
        intPart = 0;
        u_position = 0;
        position_error = 0;
        u_velocity = 0;
        velocity_error = 0;
        lastAbsIntPart = 0;
}

// calculation of the velocity reference
void controller_calculate_force(){
        // P controller
        position_error = observer_compare_positions(0);
        u_position = position_error << K;

        // send the velocity reference
        observer_change_desired_velocity(-u_position);
}

void controller_control_velocity(){
        // conversion velocity – motor force
        controller_calculate_force();

        // PI controller
        velocity_error = observer_compare_velocities();
        int32_t propPart = (velocity_error >> KV);
        intPart = absIntPart >> KIV;
```

```
        // antiwindup
        if((propPart > SAT_LIMIT) || (propPart < -SAT_LIMIT)) absIntPart=lastAbsIntPart;
        else {
                if (propPart > 0)       {
                        if((propPart + intPart) > SAT_LIMIT) absIntPart = SAT_LIMIT - propPart;
                }
                else{
                        if((propPart + intPart) < -SAT_LIMIT) absIntPart = -SAT_LIMIT - propPart;
                }
        }

        intPart = absIntPart>>KIV;

        u_velocity = observer.desired_force - propPart - intPart;

        lastAbsIntPart =  absIntPart;

        // high saturation
        if(u_velocity > MAX_FORCE) u_velocity = MAX_FORCE;
        if(u_velocity < -MAX_FORCE) u_velocity = -MAX_FORCE;

        // send the output
        controller_backlash_compensation(-u_velocity);
}

// Backlash compensation
void controller_backlash_compensation(int32_t u_velocity){
  if (observer_compare_positions(0) > 32 || observer_compare_positions(0) < -32 ) {
    // the error is big enough to use both motors in the same direction
    set_motor_force(0,u_velocity);
    set_motor_force(1,u_velocity);
    }
  else if (observer_compare_positions(0) > 0){
    // right direction, use motor 1 in brake
    set_motor_force(0,u_velocity);
    set_motor_force(1,observer_compare_positions(0)*((u_velocity + BRAKE) >> 5) - BRAKE);
  }
  else if(observer_compare_positions(0) < 0){
    // left direction, use motor 2 in brake
    set_motor_force(0,-observer_compare_positions(0)*((u_velocity + BRAKE) >> 5) + BRAKE);
    set_motor_force(1,u_velocity);
  }
  else {
    // use both motors in different direction when the track is stopped
    set_motor_force(0,BRAKE);
    set_motor_force(1,-BRAKE);
  }
}
```

C.5: Controller.h

```
#ifndef __controller_h
#define __controller_h

#include "sme.h"

// extern variable that need to be use in other program
volatile extern int64_t absIntPart;    // integral part of the PI controller, used in admanager.c
extern int32_t velocity_error;         // control error, used in admanager.c

void controller_init();
void controller_calculate_force();
void controller_control_velocity();
void controller_backlash_compensation(int32_t u_velocity);

#endif
```

C.6: Eeprom.c

```
#include <avr/io.h>
#include "eeprom.h"

// initialisation of all the register to write on the eeprom
unsigned char eeprom_write(unsigned int addr, unsigned char value)
{
  unsigned char result = 0;
  unsigned char sreg;

  sreg = SREG;
  cli();
  if ((EECR & 0x02) == 0 &&
      (SPMCR & 0x01) == 0) {
    result = 1;
    EEARL = addr & 0xff;
    EEARH = addr >> 8;
    EEDR = value;
    EECR = 0x04;
    EECR = 0x06;
  }
  SREG = sreg;
  return result;
}

// initialisation of all the register to read in the eeprom
unsigned char eeprom_read(unsigned int addr)
{
  unsigned char result;
  unsigned char sreg;

  sreg = SREG;
  cli();

  while ((EECR & 0x02) != 0) {};
  EEARL = addr & 0xff;
  EEARH = addr >> 8;
  EECR = 0x01;
  result = EEDR;

  SREG = sreg;
  return result;
}
```

C.7: Eeprom.h

```
#ifndef __eeprom_h
#define __eeprom_h


unsigned char eeprom_write(unsigned int addr, unsigned char value);
unsigned char eeprom_read(unsigned int addr);


#endif
```

C.8: Init.c

```
#include "sme.h"
#include "motor.h"
#include "admanager.h"
#include "observer.h"
#include "controller.h"
#include "masterio.h"

static void init_ports();

// initialisation of all the parameters
void init_sme()
{
  // stop motors
  set_motor_force(0, 0);
  set_motor_force(1, 0);

  // intialize components
  init_ports();
  admanager_init();
  observer_init();
  serialio_init();
  controller_init();
  masterio_init();
}

// initialisation of all the ports for the communication
static void init_ports()
{
  // motor control
  DDRC = 0xf0;         // Bit 4, 5, 6 & 7 output
  PORTC = 0x0c;        // Pull up on overtemp signals

  DDRD = 0x30;         // Bit 4 & 5 output

  // Timer 1 (PWM) 14.7456MHz/1/1024 -> 14.4kHz
  TCCR1A = 0xa3;       // OC1A & OC1B 10 bit phase correct PWM, active high
  TCCR1B = 0x01;       // Clock / 1

  // RS-232
  UCSRA = 0x00;        // USART:
  UCSRB = 0x98;        // USART: RxIntEnable|RxEnable|TxEnable
  UCSRC = 0x86;        // USART: 8bit, no parity

  UBRRH = (USART_DIVIDER-1)>>8;
  UBRRL = (USART_DIVIDER-1)&255;
}
```

C.9: Init.h

```
#ifndef __init_h
#define __init_h

void init_sme();

#endif
```

C.10: Masterio.c

```
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <avr/io.h>
#include <avr/twi.h>
#include "sme.h"
#include "masterio.h"
#include "eeprom.h"
#include "receiver.h"

#define EEPROM_SLAVE_ADDRESS 256

uint8_t masterio_slave_address;
volatile uint8_t masterio_destination;

uint8_t masterio_sr_channel;
uint8_t masterio_sr_count;
unsigned long masterio_sr_value;

volatile uint8_t masterio_st_buffer[5];
volatile uint8_t masterio_st_count;
volatile uint8_t masterio_st_full_count;

void masterio_init()
{
  masterio_destination = 0;    // 0 = rs232, 1 = twi
  masterio_sr_channel = -1;
  masterio_sr_count = 0;
  masterio_sr_value = 0;
  masterio_set_slave_address(eeprom_read(EEPROM_SLAVE_ADDRESS));

  masterio_st_count = 0;
  masterio_st_full_count = 0;
}

void masterio_set_slave_address(uint8_t address)
{
  masterio_slave_address = address&127;

  if (address == 0) {
    TWCR = 1<<TWEN;     // do not reply on TWI, for security's sake
  }
  else {
    TWAR = address<<1;
    TWCR = (1<<TWEA) | (1<<TWEN) | (1<<TWIE);    // hits on TWI, adress and interrupts
  }
}
```

```
uint8_t masterio_get_slave_address()
{
 return masterio_slave_address;
}

void masterio_store_slave_address(uint8_t address)
{
 while (!eeprom_write(EEPROM_SLAVE_ADDRESS, address));
 masterio_set_slave_address(address);
}

void masterio_putbit(unsigned char channel, unsigned char value)
{
 if (masterio_destination==0) serialio_putbit(channel, value);
 else {
  uint8_t temp = SREG;

  while (TRUE) {
   while (masterio_st_count > 0); // block until the data buffer is empty
   cli();
   if (masterio_st_count == 0) break;
   SREG = temp;
  }
  // interrupt-flag is loading new data in buffer
  masterio_st_count = 1;
  masterio_st_full_count = 1;
  masterio_st_buffer[0] = ((value!=0)<<5)|(channel&0x1f);   // set bit

  SREG = temp;
 }
}

void masterio_putchannel(unsigned char channel, unsigned long value)
{
 if (masterio_destination==0) serialio_putchannel(channel, value);
 else {
  uint8_t temp = SREG;

  while (TRUE) {
   while (masterio_st_count > 0); // block until the data buffer is empty
   cli();
   if (masterio_st_count == 0) break;
   SREG = temp;
  }

  // interrupt-flag is loading new data in buffer
  masterio_st_count = 5;
  masterio_st_full_count = 5;
  masterio_st_buffer[4] = (4<<5)|(channel&0x1f); // set channel
  *((unsigned long*)&masterio_st_buffer[0]) = value;
```

```
    SREG = temp;
  }
}

#define TWCR_CONTINUE ((1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE))

// interrupt on TWI signal
SIGNAL(SIG_2WIRE_SERIAL)
{
  uint8_t status = TWSR & 0xf8;

  if ((status >= TW_ST_SLA_ACK) && (status <= TW_ST_LAST_DATA)) {
    // slave transmitter
    if ((status == TW_ST_SLA_ACK) || (status == TW_ST_ARB_LOST_SLA_ACK) || (status ==
TW_ST_DATA_ACK)) {
      // opportunity to send a data byte
      if (masterio_st_count > 0) {
          masterio_st_count--;
          TWDR = masterio_st_buffer[masterio_st_count];
          TWCR = TWCR_CONTINUE;
      }
      else {
          // no more data: send 0xff
          TWDR = 0xff;
          TWCR = TWCR_CONTINUE;
      }
    }
    else {
      // wait for mastern to complete the transfert

      // indicates buffern for a new experiment if the transfer was aborted before scheduled time
      if (masterio_st_count > 0) masterio_st_count = masterio_st_full_count;
      TWCR = TWCR_CONTINUE;
    }
  }
  else if ((status >= TW_SR_SLA_ACK) && (status <= TW_SR_STOP)) {
    // slave reciever
    masterio_destination = 1; // replies on twi

    if ((status >= TW_SR_DATA_ACK) && (status <= TW_SR_GCALL_DATA_NACK)) {
      // data to retreive
      uint8_t data = TWDR;

      if (masterio_sr_count > 0) {
          masterio_sr_value = (masterio_sr_value << 8) | data;
          masterio_sr_count--;

          if (masterio_sr_count == 0) {
            // set channel mottagen
            receive_channel(masterio_sr_channel, masterio_sr_value);
          }
```

```
      }
      else {
          uint8_t type = data >> 5;
          uint8_t channel = data & 0x3f;
          if ((type & 6)==0) {
            uint8_t bit = type&1;
          }
          else if (type == 2) {
            // poll bit
            receive_bit_poll(channel);
          }
          else if (type == 4) {
            // beginning on set channel
            masterio_sr_channel = channel;
            masterio_sr_value = 0;
            masterio_sr_count = 4;  // ta emot 4 bytes data
          }
          else if (type == 6) {
            // poll channel
            receive_channel_poll(channel);
          }
      }
    }
    else {
      // start/seals on transfert
      masterio_sr_count = 0;
      masterio_sr_value = 0;
    }
    TWCR = TWCR_CONTINUE;
  }
  else if (status == TW_BUS_ERROR) {
    TWCR = (1<<TWSTO) |(1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE); //reset the bus
  }
  else {
    TWCR = TWCR_CONTINUE;
  }
}

// interrupt on UART signal
SIGNAL(SIG_UART_RECV)
{
  char ch = UDR;

  masterio_destination = 0;

  switch (serialio_RXC(ch)) {
    case serialio_pollbit: {
      receive_bit_poll(serialio_channel);
    } break;
    case serialio_pollchannel: {
      receive_channel_poll(serialio_channel);
```

```
    } break;

    case serialio_setchannel: {
      receive_channel(serialio_channel, serialio_value);
    } break;

    case serialio_more: {
    } break;
    case serialio_error: {
    } break;
  }
}
```

C.11: Masterio.h

```
#ifndef __masterio_h
#define __masterio_h

#include "sme.h"

void masterio_init();
void masterio_set_slave_address(uint8_t address);
uint8_t masterio_get_slave_address();
void masterio_store_slave_address(uint8_t address);

void masterio_putbit(unsigned char channel, unsigned char value);
void masterio_putchannel(unsigned char channel, unsigned long value);

#endif
```

C.12: Motor.c

```
#include <avr/io.h>

int16_t motor_force[2];

void set_motor_force(int16_t value)
{
  // saturation
  int pwm = value;
  if (value < 0) pwm =  -value;
  if (pwm > 0x3ff) { pwm = 0x3ff; }

  if (value < 0) PORTC |= 0x80;
  else PORTC &= ~0x80;
  OCR1B = pwm & 0x3ff;
  motor_force[0] = value;

}
```

C.13: Motor.h

```
#ifndef __motor_h
#define __motor_h

#include <inttypes.h>

extern int16_t motor_force[2];

void set_motor_force(int16_t value);

#endif
```

C.14: Observer.c

```
#include "observer.h"
#include "channels.h"
#include "eeprom.h"
#include "masterio.h"

#define OBSERVER_INPUT_EXP          6
#define POSITION_DIVIDER            9
#define MAX_LENGTH                  1080000
#define MIN_LENGTH                  0
#define MAX_OSC                     5000        // The max ocilation that we allow to reduce the

observer_t observer;

const uint16_t div_table[] = { 32768l,      (32768l+1)/ 2, (32768l+2)/ 3, (32768l+2)/ 4,
                               (32768l+3)/ 5, (32768l+3)/ 6, (32768l+4)/ 7, (32768l+4)/ 8,
                               (32768l+5)/ 9, (32768l+5)/10, (32768l+6)/11, (32768l+6)/12,
                               (32768l+7)/13, (32768l+7)/14, (32768l+8)/15, (32768l+8)/16};

void observer_set_mask(uint8_t mask)
{
  mask = mask & 17;

  // reset and restart the observer for the activated axes

  uint8_t active=(mask&1)!=0;
  uint16_t flags= (mask&(1<<4)) ? OBSERVER_DOT_ESTIMATION_FLAG : 0;
  observer.active = active!=0;
  observer.sequence = 0;
  observer.flags = flags;
  observer.phase = OBSERVER_PHASE_UNKNOWN;
  observer.position = 0;
  observer.desired_position = -30000;   // to make sure it will get to the left
  observer.last_position = 0;
  observer.desired_velocity = -0x000;
  observer.desired_force = -0x000;
  observer.last1_velocity=0;
  observer.last2_velocity=0;
  observer.last3_velocity=0;
  observer.filt_vel=0;

  observer.region = 0;
  observer.encoder_values[0] = -1;
  observer.encoder_values[1] = -1;
  observer.last_region = -1;
  observer.last_phase = 0;
  observer.calib_counter = 255;
  }
}
```

```
void observer_peek_AD_value(uint8_t channel, uint16_t value){
  if (!AD_IS_OBSERVER_CHANNEL((int8_t)channel)) return;
        // it's an encoder uint8_t type;
        uint8_t encoder = AD_CHANNEL_ENCODER(channel);
        if (AD_IS_ENCODER_CHANNEL(channel)) type=0;
        else if (AD_IS_DOT_CHANNEL(channel)) {
                type=1;
        }
  if (!observer.active) return;
  observer_calibration_t *c = &(observer.calibration);

  if (type == 0) { // regular encoder value
    if (observer.phase == OBSERVER_PHASE_UNKNOWN) {
        // startup: coarse phase approximation, find correct quarter
        observer.encoder_values[encoder] = value;
        int16_t a = observer.encoder_values[0];
        int16_t b = observer.encoder_values[1];
        if ((a >= 0) && (b >= 0)) {
                // we have both encoder values, so estimate phase quarter
                int16_t phase = 0;
                a -= c->origins[0];
                b -= c->origins[1];
                int16_t abs_a = a > 0 ? a : -a;
                int16_t abs_b = b > 0 ? b : -b;
                if (abs_a >= abs_b) {
                        if (a > 0) phase = 0;
                        else phase = 2;
                        }
                else {
                        if (b > 0) phase = 1;
                        else phase = 3;
                }
                observer.phase = phase << (OBSERVER_PHASE_BITS-2);
        }
  }
  else if ((!(observer.region & 1)) == encoder) {
        // save old data for possible differentiation
        observer.last_phase = observer.phase;
        observer.last_region = observer.region;
        observer.last_phase_adm_time = observer.phase_adm_time;
        observer.phase_adm_time = adm_time;

        // we have a measurement from the encoder we wanted
        // evalutate appropriate 3rd degree polynomial
        uint8_t region = observer.region;
        int16_t ymid = c->ymids[region];
        const int16_t *poly = c->polys[region];
        int16_t x = ((value - 512) << OBSERVER_INPUT_EXP) - ymid;

        // evaluate polynomial on Horn form
```

```
        int16_t p = poly[0];
        int i;
        for (i=1; i <= 3; i++) {
                p = (int16_t)((((int32_t)p)*((int32_t)x)) >> OBSERVER_PHASE_BITS);
                p += poly[i];
        }
        // wrap value to range of one period and store
        p = p & OBSERVER_PHASE_MASK;
        observer.phase = p;

        if (c->flags & OBSERVER_CALIB_INVERT_FLAG) p = -p;
        // update full position: choose left or right for smallest step
        int32_t diff = (p - observer.position) & OBSERVER_PHASE_MASK;
        if (diff >= (1 << (OBSERVER_PHASE_BITS-1))) {
                diff -= 1 << OBSERVER_PHASE_BITS;
        }
        observer.position += diff;
    }
  }
else if (type == 1) { // derivative
  if (observer.phase != OBSERVER_PHASE_UNKNOWN) {
     if ((!(observer.region & 1)) == encoder) {
     // we have a derivative from the right encoder
        if ((value >= c->dot_sat_min[encoder]) && (value <= c->dot_sat_max[encoder])) {
          // unsaturated measurement
          uint8_t region = observer.region;

          // evaluate polynomial
          int16_t phase = observer.phase - c->dot_phasemids[region];
          int16_t t = 2*4*phase;
          int16_t *poly2 = c->dot_polys[region];
          int16_t p2 = poly2[0];
          int i;
          for (i=1; i <= 4; i++) {
                p2 = (int16_t)((((int32_t)p2)*((int32_t)t)) >> 15);
                p2 += poly2[i];
          }
          int16_t enc_dot = value - c->dot_offsets[encoder];
          int16_t phase_dot = (int16_t)(( ((int32_t)p2) * ((int32_t)(enc_dot<<5)) ) >> 15);
          int8_t shift = OBSERVER_DOT_BITS - c->dot_exponent - 5;
          if (shift > 0) observer.velocity = (((int32_t)phase_dot) << shift);
          else observer.velocity = (((int32_t)phase_dot) >> (-shift));
          observer.velocity &= ~1; // signal velocity source in LSB
        }
        else {
          // saturated measurement: use numerical differentiation instead
          observer.velocity = 0;
          if (observer.region == observer.last_region) {
                int32_t delta_phase = observer.phase - observer.last_phase;
                delta_phase = ((delta_phase + (1<<(OBSERVER_PHASE_BITS-1))) &
        OBSERVER_PHASE_MASK) - (1<<(OBSERVER_PHASE_BITS-1));
```

```
                int32_t delta_ticks = observer.phase_adm_time - observer.last_phase_adm_time;
                if ((delta_ticks >= 1) && (delta_ticks <= 16)) {
                  delta_phase = (delta_phase*div_table[delta_ticks-1])>>15;
                }
                else {
                  delta_phase = 0;
                }
                observer.velocity = delta_phase << (OBSERVER_DOT_BITS-
          OBSERVER_PHASE_BITS);
                observer.velocity |= 1; // signal velocity source in LSB
                }
          }
      }
    }
  }
}

int8_t observer_get_next_AD_channel()
{
  int8_t encoder = observer_get_next_encoder_channel();

  if (encoder < 0) {
    encoder = observer_get_next_encoder_channel();
  }

  if (encoder==0) return AD_ENCODER_A_CHANNEL;
  else if (encoder==1) return AD_ENCODER_B_CHANNEL;
  else if (encoder==2) return AD_DOT_A_CHANNEL;
  else if (encoder==3) return AD_DOT_B_CHANNEL;
  else return -1;
}

int8_t observer_get_next_encoder_channel() {
  if (!observer.active) return -1;
  if (observer.sequence < 0) {
    observer.sequence = 0;  // leave over so other can access to AD
    return -1;
  }
  if (observer.sequence == 0) {
    // new round
    if (observer.phase == OBSERVER_PHASE_UNKNOWN) {
      //read a sample from A and after one from B
      if (observer.encoder_values[0] == -1) {
          // read from encoder A
          return 0;
      }
      else {
          // read from encoder B
          return 1;
          observer.sequence = -1;  // end the round
      }
```

```
    }
    else {
      // choose the best encoder to sample
      // predict the phase for the next measurement
      int16_t pred_phase = observer.phase & OBSERVER_PHASE_MASK;
      // find the region
      uint8_t region = 0;
      observer_calibration_t *c = &(observer.calibration);
      if (pred_phase < c->regions[2]) {
          // 0, 1 or 2
          if (pred_phase > c->regions[1]) region = 2;
          else if (pred_phase > c->regions[0]) region = 1;
      }
      else {
          // 3 or 0
          if (pred_phase < c->regions[3]) region = 3;
      }
      observer.region = region;
      if (observer.flags & OBSERVER_DOT_ESTIMATION_FLAG) {
          observer.sequence = 1;   // sample the derivative after that
      }
      else observer.sequence = -1;   // end the round
      // choose the encoder
      if ((region & 1) == 0) return 1;   // sample B for the interval (|A|>|B|)
      else return 0;    // or sample A
    }
  }
  else if (observer.sequence == 1) {
    // sample derivate, we take the same encoder we sampled at the end
    observer.sequence = -1; // end the round

    if ((observer.region & 1) == 0) return 3; // sample B for the interval (|A|>|B|)
    else return 2;
  }
  else {
  observer.sequence = 0; // unknown sequence, should never happen return -1;
  }
}

void observer_handle_calibration_command(uint32_t value){
  uint8_t command = (value>>2)&15;
  value >>= 6;
  if (command == 3) {
          masterio_store_slave_address(value);
  }
  else {
          if (command == 1) {
                  observer.calib_counter = 0;
          }
          else if (command == 2) {
                  observer_calibration_t *c = &(observer.calibration);
```

```
                uint16_t *data = (uint16_t *)c;
                if (observer.calib_counter < sizeof(observer_calibration_t)/sizeof(uint16_t)) {
                  data[observer.calib_counter] = value;
                  observer.calib_counter++;
                  }
                }
                else if (command == 7) {
                  if (observer.calib_counter == sizeof(observer_calibration_t)/sizeof(uint16_t)) {
                    uint8_t *src = (uint8_t *)&(observer.calibration);
                    uint16_t base_address = observer.eeprom_offset;
                    uint16_t i;
                    for (i=0; i < sizeof(observer_calibration_t); i++) {
                          while (!eeprom_write(i + base_address, src[i]));
                    }
                  }
                }
        }
}

void observer_init() {
   observer.eeprom_offset = 0;
   observer_set_mask(0);
}

void observer_read_eeprom_config() {
  uint8_t *dest = (uint8_t *)&(observer.calibration);
  uint16_t base_address = observer.eeprom_offset;

  uint16_t i;
  for (i=0; i < sizeof(observer_calibration_t); i++) {
    dest[i] = eeprom_read(i + base_address);
  }
}

int32_t observer_get_position() {
  return (observer.position >> POSITION_DIVIDER)*0.16;
}

int32_t observer_get_velocity() {
   if (observer.calibration.flags & OBSERVER_CALIB_INVERT_FLAG) return -
observer.velocity;
   else return observer.velocity;
  }
}

void observer_change_desired_position(uint32_t value){
        if(value > MAX_LENGTH) value = MAX_LENGTH;
        if(value < MIN_LENGTH) value = MIN_LENGTH;
        observer.desired_position=value+observer.reference_position;
}
```

```
int16_t observer_compare_positions() {
  // convert the position in mm and compare it to the reference position
  int16_t error = (observer.position >> POSITION_DIVIDER)*0.16 - observer.desired_position;
        return error;
}

void observer_change_desired_velocity(int32_t value){
        observer.desired_velocity=value;
        observer.desired_force=observer_calculate_desired_force(value);
}

int32_t observer_compare_velocities(){
        int32_t error = (observer.filt_vel - observer.desired_velocity);

        // measurement sent to the PC in order to plot the graph using the java interface
        masterio_putchannel(7 observer_get_position());
        masterio_putchannel(31,observer.filt_vel);
        masterio_putchannel(30,adm_time);

        return error;
}

void observer_velocity_filter(){
        // disable interrupts
        cli();

        // velocity estimation limit to prevent calculation errors
        if(observer.velocity>observer.filt_vel+MAX_OSC)
observer.velocity=observer.filt_vel+MAX_OSC;
        if(observer.velocity<observer.filt_vel-MAX_OSC) observer.velocity=observer.filt_vel-
MAX_OSC;

        // mean filter
        observer.filt_vel=(observer.velocity+observer.last1_velocity+observer.last2_velocity+obser
ver.last3_velocity)>>2;

        observer.last3_velocity=observer.last2_velocity;
        observer.last2_velocity=observer.last1_velocity;
        observer.last1_velocity=observer.velocity;

        // enable the interrupts
        sei();
}

// convert the velocity reference to a motor force
int16_t observer_calculate_desired_force(int32_t value){
if(value > 10000)  return ((value*11) >> 11)+88;  // force=vel/186+89
else if (value < -10000) return ((value*11) >> 11)-88;  // force=vel/186-89
      else  return ((value*29) >> 11);  // force=vel/70.6
}
```

C.15: Observer.h

```
#ifndef __observer_h
#define __observer_h

#include "sme.h"

#define OBSERVER_PHASE_BITS                     14
#define OBSERVER_PHASE_MASK     ((1 << (OBSERVER_PHASE_BITS)) - 1)
#define OBSERVER_PHASE_UNKNOWN                  0x8000

#define OBSERVER_DOT_BITS                       23
#define OBSERVER_DOT_MASK       ((1l << (OBSERVER_DOT_BITS)) - 1)

#define OBSERVER_CALIB_INVERT_FLAG 1 // negate observer output signal
#define OBSERVER_DOT_ESTIMATION_FLAG 1 // do velocity estimation

// calibration data
typedef struct {
        char id[8];                             // 4
        int16_t origins[2];                     // 2
        int16_t regions[4];                     // 4
        int16_t ymids[4];                       // 4
        int16_t polys[4][4];    // [region][index]      //+16
                                                //=30

        uint16_t flags;                         // 1
        int16_t dot_offsets[2];                 // 2
        int16_t dot_sat_min[2];                 // 2
        int16_t dot_sat_max[2];                 // 2
        int16_t dot_phasemids[4];               // 4
        int16_t dot_polys[4][5];  // [region][index]  // 20
        int16_t dot_exponent;                   //+ 1
                                                //=31
} observer_calibration_t;       //in total: 62 words (should not exceed 128 for EEPROM space)

typedef struct {
        uint8_t active;
        int8_t sequence; // 0 = new round, -1 = end of round, 1 = time for derivative sampling
        uint16_t flags;

        volatile int16_t phase;
        volatile int32_t position;

        volatile int32_t desired_position;
        volatile int32_t reference_position;
        volatile int32_t last_position;

        volatile int32_t velocity;
        int32_t desired_velocity;
```

```
        int16_t  desired_force;

        int32_t last1_velocity;
        int32_t last2_velocity;
        int32_t last3_velocity;
        int32_t filt_vel;

        int8_t region;
        int8_t last_region;
        int16_t last_phase;

        int32_t phase_adm_time, last_phase_adm_time;
        int16_t encoder_values[2];

        observer_calibration_t calibration;
        uint8_t calib_counter;
        int16_t eeprom_offset;
} observer_t;

void observer_init();
void observer_set_mask(uint8_t mask); // set which variables to estimate
void observer_read_eeprom_config();

int32_t observer_get_position();
int32_t observer_get_velocity();

// AD manager interface
int8_t observer_get_next_AD_channel();
void observer_peek_AD_value(uint8_t channel, uint16_t value);
void observer_process_AD_value(uint8_t channel, uint16_t value);

void observer_change_desired_position(uint32_t value);
int16_t observer_compare_positions();
int8_t observer_get_next_encoder_channel();

#endif
```

C.16: Receiver.c

```
#include "sme.h"
#include "serialio.h"
#include "channels.h"
#include "motor.h"
#include "observer.h"

#define VEL_VALUE    170000

void receive_bit_poll(uint8_t channel)
{
  requester_process_digital_request(channel);
}

void receive_channel_poll(uint8_t channel)
{
  requester_process_request(channel);
}

void receive_channel(uint8_t channel, unsigned long value)
{
  switch (channel) {
          case MOTOR_INPUT_CHANNEL(0) : {
                  observer_change_desired_position(0,value);
          } break;

          case MOTOR_INPUT_CHANNEL(1) : {
                  observer_change_desired_position(0,value);
          } break;

          case OBSERVER_CONF_CHANNEL: {
                  observer_set_mask(value & 3);
          } break;

          case ADM_CONF_CHANNEL: {
                  admanager_handle_conf_command(value);
          } break;

          case CHANGE_DESIRED_POSITION_CHANNEL: {
                  observer_change_desired_position(0,value);
          } break;

          case DESIRED_VELOCITY:{
           observer_change_desired_velocity(value-VEL_VALUE);
           } break;
   }
}
```

C.17: Receiver.h

```
#ifndef __reciever_h
#define __reciever_h

void receive_bit_poll(uint8_t channel);
void receive_channel_poll(uint8_t channel);
void receive_channel(uint8_t channel, unsigned long value);

#endif
```

C.18: Requester.c

```
#include "requester.h"
#include "observer.h"
#include "masterio.h"

#define POLL_AXIS1_OVERTEMP      0x0001
#define POLL_AXIS2_OVERTEMP      0x0002
#define POLL_AXIS1_POSITION      0x0004
#define POLL_AXIS1_A             0x0008
#define POLL_AXIS1_B             0x0010
#define POLL_AXIS1_CURRENT       0x0020
#define POLL_AXIS1_ADOT          0x0040
#define POLL_AXIS1_BDOT          0x0080
#ifdef MOTOR_X2
  #define POLL_AXIS2_CURRENT     0x0200
#endif
#define POLL_AXIS1_VELOCITY      0x0400

#define POLL_SLAVE_ADDRESS       0x4000
#define POLL_CONFIG              0x8000

static volatile uint16_t requester_poll_mask;
static volatile uint8_t  requester_AD_mask;
static volatile uint16_t requester_AD_values[8];

int8_t requester_get_next_AD_channel()
{
  uint8_t mask = requester_AD_mask;
  int8_t channel = 0;

  if (mask == 0) return -1;

  while ((mask & 1) == 0) {
    channel++;
    mask >>= 1;
  }

  return channel;
}

void requester_process_AD_value(uint8_t channel, uint16_t value)
{
  if (requester_AD_mask & (1 << channel)) {
    requester_AD_values[channel] = value;

    switch (channel) {
      case AD_ENCODER_A_CHANNEL(0): {
           requester_poll_mask |= POLL_AXIS1_A;
      } break;
```

```
      case AD_ENCODER_B_CHANNEL(0): {
          requester_poll_mask |= POLL_AXIS1_B;
      } break;

      case AD_DOT_A_CHANNEL(0): {
          requester_poll_mask |= POLL_AXIS1_ADOT;
      } break;
      case AD_DOT_B_CHANNEL(0): {
          requester_poll_mask |= POLL_AXIS1_BDOT;
      } break;

       case AD_CURRENT_CHANNEL(0): {
          requester_poll_mask |= POLL_AXIS1_CURRENT;
      } break;

#ifdef MOTOR_X2
      case AD_CURRENT_CHANNEL(1): {
          requester_poll_mask |= POLL_AXIS2_CURRENT;
      } break;
#endif
    }

    // stop asking for this AD channel
    requester_AD_mask &= ~(1 << channel);
  }
}

void requester_process_digital_request(uint8_t channel)
{
  switch (channel) {
    case OVERTEMP_DOUTPUT_CHANNEL(0): { requester_poll_mask |=
POLL_AXIS1_OVERTEMP; } break;
    case OVERTEMP_DOUTPUT_CHANNEL(1): { requester_poll_mask |=
POLL_AXIS2_OVERTEMP; } break;
  }
}

void requester_process_request(uint8_t channel)
{
  switch (channel) {
    case POSITION_OUTPUT_CHANNEL(0):   {
      requester_poll_mask |= POLL_AXIS1_POSITION;
    } break;
    case ENCODER_A_OUTPUT_CHANNEL(0):  {
      requester_AD_mask |= (1 << AD_ENCODER_A_CHANNEL(0));
    } break;
    case ENCODER_B_OUTPUT_CHANNEL(0):  {
      requester_AD_mask |= (1 << AD_ENCODER_B_CHANNEL(0));
    } break;

    // dot
```

```
    case DOT_A_OUTPUT_CHANNEL(0):  {
      requester_AD_mask |= (1 << AD_DOT_A_CHANNEL(0));
    } break;
    case DOT_B_OUTPUT_CHANNEL(0):  {
      requester_AD_mask |= (1 << AD_DOT_B_CHANNEL(0));
    } break;

    case CURRENT_OUTPUT_CHANNEL(0):  {
      requester_AD_mask |= (1 << AD_CURRENT_CHANNEL(0));
    } break;

    case VELOCITY_OUTPUT_CHANNEL(0):   {
      requester_poll_mask |= POLL_AXIS1_VELOCITY;
    } break;

    case CALIBRATION_CHANNEL:
    {
      requester_poll_mask |= POLL_SLAVE_ADDRESS;
    } break;

    case CONFIGURATION_CHANNEL: {
      requester_poll_mask |= POLL_CONFIG;
    } break;
  }
}

// informs master (Simulink) of available channels and their type, range and resolution
static void send_config()
{
  CONF_DIGITAL_IN(0, CONF_RESOLUTION(1));     // Axis1 -- Overtemp
  CONF_DIGITAL_IN(1, CONF_RESOLUTION(1));     // Axis2 -- Overtemp

  CONF_ENCODER_IN( POSITION_OUTPUT_CHANNEL(0)  , CONF_RESOLUTION(25));
// Axis1 -- position

  CONF_ANALOG_IN(  ENCODER_A_OUTPUT_CHANNEL(0) , CONF_RESOLUTION(10));
// Axis1 -- encoder A
  CONF_ANALOG_IN(  ENCODER_A_OUTPUT_CHANNEL(0) ,
CONF_MIN(CONF_POSITIVE_VOLT(0)));
  CONF_ANALOG_IN(  ENCODER_A_OUTPUT_CHANNEL(0) ,
CONF_MAX(CONF_POSITIVE_VOLT(5)));
  CONF_ANALOG_IN(  ENCODER_B_OUTPUT_CHANNEL(0) , CONF_RESOLUTION(10));
// Axis1 -- encoder B
  CONF_ANALOG_IN(  ENCODER_B_OUTPUT_CHANNEL(0) ,
CONF_MIN(CONF_POSITIVE_VOLT(0)));
  CONF_ANALOG_IN(  ENCODER_B_OUTPUT_CHANNEL(0) ,
CONF_MAX(CONF_POSITIVE_VOLT(5)));

  CONF_ANALOG_IN(  DOT_A_OUTPUT_CHANNEL(0) , CONF_RESOLUTION(10));     //
Axis1 -- encoder A
  CONF_ANALOG_IN(  DOT_A_OUTPUT_CHANNEL(0) ,
```

```
CONF_MIN(CONF_POSITIVE_VOLT(0)));
  CONF_ANALOG_IN(  DOT_A_OUTPUT_CHANNEL(0) ,
CONF_MAX(CONF_POSITIVE_VOLT(5)));
  CONF_ANALOG_IN(  DOT_B_OUTPUT_CHANNEL(0) , CONF_RESOLUTION(10));     //
Axis1 -- encoder B
  CONF_ANALOG_IN(  DOT_B_OUTPUT_CHANNEL(0) ,
CONF_MIN(CONF_POSITIVE_VOLT(0)));
  CONF_ANALOG_IN(  DOT_B_OUTPUT_CHANNEL(0) ,
CONF_MAX(CONF_POSITIVE_VOLT(5)));


  CONF_ANALOG_IN(  CURRENT_OUTPUT_CHANNEL(0)   , CONF_RESOLUTION(8));
// Axis1 -- current
  CONF_ANALOG_IN(  CURRENT_OUTPUT_CHANNEL(0)   ,
CONF_MIN(CONF_POSITIVE_VOLT(0)));
  CONF_ANALOG_IN(  CURRENT_OUTPUT_CHANNEL(0)   ,
CONF_MAX(CONF_POSITIVE_VOLT(5)));

  CONF_ENCODER_IN( VELOCITY_OUTPUT_CHANNEL(0)  ,
CONF_RESOLUTION(OBSERVER_DOT_BITS));    // Axis1 -- velocity

  CONF_ANALOG_OUT(0, CONF_RESOLUTION(11));    // Axis1 speed
  CONF_ANALOG_OUT(0, CONF_MIN(CONF_NEGATIVE_VOLT(12)));
  CONF_ANALOG_OUT(0, CONF_MAX(CONF_POSITIVE_VOLT(12)));
}

// returns TRUE if something was sent
bool requester_send_data()
{
  uint16_t poll;

  // take current poll bits and reset
  uint8_t temp = SREG;
  cli();
  poll = requester_poll_mask;
  requester_poll_mask = 0;
  SREG = temp;

  if (poll == 0) return FALSE;

  if (poll & POLL_AXIS1_OVERTEMP) {
    uint8_t overtemp = (PORTC >> 3) & 1;
    masterio_putbit(OVERTEMP_DOUTPUT_CHANNEL(0), overtemp);
   }

  if (poll & POLL_AXIS1_POSITION) {
    // compensate for offset conversion in simulink serialio driver
    masterio_putchannel(POSITION_OUTPUT_CHANNEL(0),
observer_get_position(0)+(((uint32_t)1)<<24));
   }
  if (poll & POLL_AXIS1_VELOCITY) {
```

```
    // compensate for offset conversion in simulink serialio driver
    masterio_putchannel(VELOCITY_OUTPUT_CHANNEL(0),
(observer_get_velocity(0)+(((uint32_t)1)<<(OBSERVER_DOT_BITS-1))) &
OBSERVER_DOT_MASK);
  }

  if (poll & POLL_AXIS1_A) {
    masterio_putchannel(ENCODER_A_OUTPUT_CHANNEL(0),
                        requester_AD_values[AD_ENCODER_A_CHANNEL(0)]);
  }
  if (poll & POLL_AXIS1_B) {
    masterio_putchannel(ENCODER_B_OUTPUT_CHANNEL(0),
                        requester_AD_values[AD_ENCODER_B_CHANNEL(0)]);
  }

  if (poll & POLL_AXIS1_ADOT) {
    masterio_putchannel(DOT_A_OUTPUT_CHANNEL(0),
                        requester_AD_values[AD_DOT_A_CHANNEL(0)]);
  }
  if (poll & POLL_AXIS1_BDOT) {
    masterio_putchannel(DOT_B_OUTPUT_CHANNEL(0),
                        requester_AD_values[AD_DOT_B_CHANNEL(0)]);
  }

  if (poll & POLL_AXIS1_CURRENT) {
    masterio_putchannel(CURRENT_OUTPUT_CHANNEL(0),
                        requester_AD_values[AD_CURRENT_CHANNEL(0)]);
  }

  if (poll & POLL_SLAVE_ADDRESS) {
    masterio_putchannel(CALIBRATION_CHANNEL, masterio_get_slave_address());
  }

  if (poll & POLL_CONFIG) send_config();

  return TRUE;
}

void requester_init()
{
  requester_poll_mask = 0;
  requester_AD_mask = 0;
}
```

C.19: Requester.h

```
#ifndef __requester_h
#define __requester_h

#include "sme.h"


void requester_init();
void requester_process_digital_request(uint8_t channel);
void requester_process_request(uint8_t channel);
bool requester_send_data();


// AD interface
int8_t requester_get_next_AD_channel();
void requester_process_AD_value(uint8_t channel, uint16_t value);

                                                    #endif
```

C.20: Sender.c

```
#include "serialio.h"
#include "requester.h"

// move the robot to the right during the initialisation
// set the reference position to 0 at the extreme right
void get_to_the_right(){

  set_motor_force(0,0x150); //right
  set_motor_force(1,0x150); //right

  int i=0;
  int pos_temp=0;
  observer.position=80000;

  // stay in the loop until the position stop to change during a certain time
  while(i<20000){
          i++;
          if(pos_temp!=(observer_get_position(0) >> 9)){
            pos_temp=observer_get_position(0) >> 9;
            i=0;
          if(observer.position<40000) observer.position=+40000;
          }
  }

  // stop both motors
  set_motor_force(0,0x00);
  set_motor_force(1,0x00);

  // put the reference position where the motors stopped
  observer.reference_position=observer_get_position(0);

  // stay in this position
  observer.desired_position=observer.reference_position;
}

void send_loop()
{
  get_to_the_right();
  while (1) {
    requester_send_data();

    // compute the velocity filter
    observer_velocity_filter();

    // compute the controller
    controller_control_velocity();
   }
}
```

C.21: Sender.h

```
#ifndef __sender_h
#define __sender_h

void get_to_the_right();
void send_loop();


#endif
```

C.22: Sme.c

```
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <avr/io.h>

#include <serialio.h>
#include "sme.h"
#include "channels.h"
#include "init.h"

#include "init.c"
#include "admanager.c"
#include "observer.c"
#include "requester.c"
#include "controller.c"
#include "motor.c"
#include "receiver.c"
#include "sender.c"
#include "eeprom.c"
#include "masterio.c"

int main()
{
  init_sme();

  // send reset message
  masterio_putchannel(RESET_MESSAGE_CHANNEL, RESET_MESSAGE);

  admanager_set_client_slots(ADM_CLIENT_OBSERVER, 2);
  observer_read_eeprom_config();

  observer_set_mask(1|16); // activate position and velocity observer

  sei();   // Global interrupt enable

  send_loop();

  return 0;
}
```

C.23: Sme.h

```
#ifndef __sme_h
#define __sme_h

#include <inttypes.h>

typedef uint8_t bool;
#define FALSE 0
#define TRUE  1

// ADC-clocking
#define ADC_DIVIDER_EXP 7
// ADC-freq = 8.8kHz * 2^ADC_OVERCLOCK_EXP = 8.8kHz * 2^(7-
ADC_OVERCLOCK_EXP)
#define ADC_OVERCLOCK_EXP (7-ADC_DIVIDER_EXP)

// USART-clocking
#define USART_DIVIDER 8    // 115200 @ 14.7456MHz


#endif
```