Contents

List of Figures

List of Tables

Listings

Chapter 1

Mathematical Modeling and Control

1.1 Modelling and Control

1.1.1 Introduction

Most of the robots today are driven by electric motors. For large industrial robots typically are used brushless servo motors while small laboratory or hobby robots use brushed DC motors or stepper motors.

Electric motors are compact and efficient. They do not produce very high torque but they can rotate at very high speed. To increase the torque, reduction gearboxes are used to tradeoff speed with increased torque. The disadvantage though is that gearboxes are increasing cost, weight but most important friction and mechanical noise.

1.1.2 Mathematical Modelling of DC Motor

The analysis following is mainly a review of [**?**].

Figure 1.1 shows the schematic of the drive train of a typical robot joint. The torque developed on the motor shaft is directly proportional to the field flux and the armature current. The relationship among the developed torque, flux ϕ, and current i_a is

$$T_m = K_m \phi i_a \tag{1.1}$$

where T_m is the motor torque (N-m, lb-ft or oz-in), ϕ the magnetic flux (webers), i_a the armature current (amperes), and K_m is a proportional constant.

In addition, when the conductor moves in the magnetic field, a voltage is generated across its terminals. This voltage, the **back emf**, which is proportional to the shaft velocity, tends to oppose the current flow. The relationship between the back emf and the shaft velocity is,

$$e_b = K_m \phi \omega_m \tag{1.2}$$

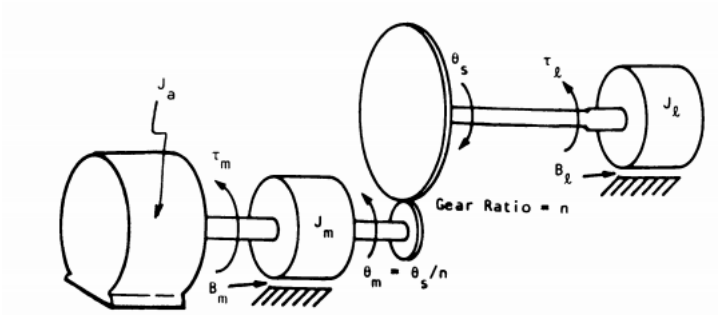


Fig. 1.1: Schematic representation of actuator-gear-load assembly for one joint.

where e_b denoted the back emf (volts), and ω_m is the shaft velocity (rad/sec) of the motor. The equations 1.1 and 1.2 form the basis of the dc-motor operation.

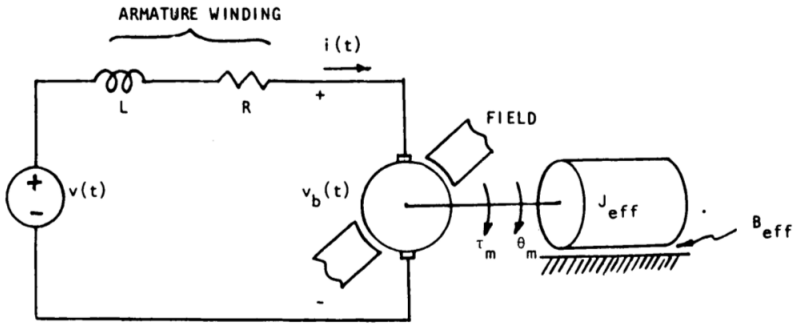


Fig. 1.2: Schematic diagram for electrical drive system

With reference to the circuit diagram of Figure 1.2, the control of the dc motor is applied at the armature terminals in the form of the applied voltage $e_a(t)$. For linear analysis, we assume that the torque developed by the motors is proportional to the air-gap flux and the armature current. Thus

$$T_m(t) = K_m \phi i_a(t) \tag{1.3}$$

If we assume that ϕ is constant Equation 1.3 is written

$$T_m(t) = K_i i_a(t) \tag{1.4}$$

where K_i is the **torque constant** in N-m/A, lb-ft/A or oz-in./A.

Although a dc motor by it self is basically an open-loop system, the state diagram and the block diagram show that the motor has a *"built-in"* feedback loop caused by back emf. Physically, the back emf represents the feedback of a signal that is proportional to the negative of the speed of the motor. As seen from Equation 1.2, the back-emf constant K_b represents an added term to the resistance R_a and the viscous-friction coefficient B_m. Therefore, *the back-emf is equivalent to an "electric friction", which tends to improve the stability of the motor, and in general, the stability of the system.*

Refer to Figure 1.1, the schematic representation of an actuator-gear-load assembly for a single joint, in which

J_a actuator inertia of one joint,
J_m manipulator inertia of the joint fixtures at actuator side,
J_l inertia of the manipulator link,
B_m damping coefficient at actuator side
B_l damping coefficient at load side
f_m average friction torque
τ_g gravitational torque
τ_m generated torque at actuator shaft
τ_l internal load torque,
θ_m angular displacement at actuator shaft,
θ_s angular displacement at load side.

N_m, N_s number of teeth of the gears at the actuator shaft and load shaft, respectively,
r_m, r_s pitch radii of the gears at the actuator shaft and load shaft, respectively

Then

$$n = r_m/r_s = N_m/N_s \leq 1 \tag{1.5}$$

is the gear ratio. As indicated in Fig. 1.2, the force F is transmitted from the actuator to the load at the contacting point of the mating gears. Thus,

$$\tau_l^{'} = equivalent\, internal\, load\, torque\, at\, actuator\, shaft = Fr_m \tag{1.6}$$

and

$$\tau_l = Fr_s \tag{1.7}$$

which leads to

$$\tau_l^{'}/\tau_l = r_m/r_s = n \tag{1.8}$$

or

$$\tau_l^{'} = n\tau_l \tag{1.9}$$

From Figure 1.3 , the pitch angle of one tooth is

$$\theta_m = 2\pi/N_m \tag{1.10}$$

$$\theta_s = 2\pi/N_s \tag{1.11}$$

so that

$$\theta_s = \theta_m N_m \; N_s = n\theta_m \tag{1.12}$$

By taking time derivatives on both sides of (1.12), one obtains relations between angular velocities and accelerations as

$$\dot{\theta}_s = n\dot{\theta}_m \tag{1.13}$$

$$\ddot{\theta}_s = n\ddot{\theta}_m \tag{1.14}$$

From Figure 1.1, it is seen that the internal load torque t_l is required to overcome the link inertia effect $J_l\ddot{\theta}_s$ and damping effect $B_l\dot{\theta}_s$. Using D'Alembert's principle "$\sum torque = \sum J\ddot{\theta}$", one obtains

$$\tau_l - B_l\dot{\theta}_s = J_l\ddot{\theta}_s \tag{1.15}$$

Apply the same principle at the actuator shaft to obtain

$$\tau_m - \tau_l^{'} - B_m\dot{\theta}_m = \left(J_a + J_m\right)\ddot{\theta}_m \tag{1.16}$$

If we wish to express the torque relation at the actuator shaft, first substitute (1.13) and (1.14) into (1.15), and then combine the result with (1.9) to obtain

$$\tau_l^{'} = n^2\left(J_l\ddot{\theta}_m + B_l\dot{\theta}_m\right) \tag{1.17}$$

Combining (1.16) and (1.17) yields

$$\tau_m = \left(J_a + J_m + n^2 J_l\right)\ddot{\theta}_m + \left(B_m + n^2 B_l\right)\dot{\theta}_m \tag{1.18}$$

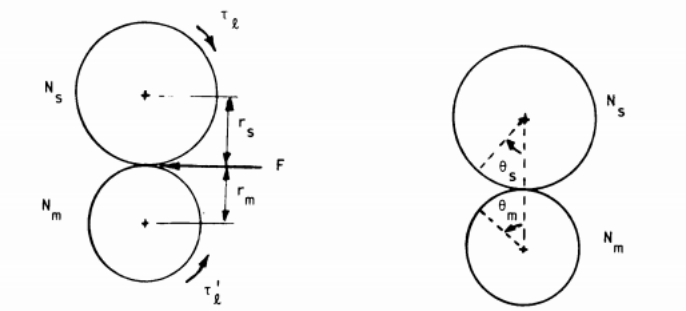


Fig. 1.3: Schematic of gear ratio

where we can assign $J_{eff} = \left(J_a + J_m + n^2 J_l\right)$ is the effective inertia and $B_{eff} = \left(B_m + n^2 B_l\right)$ is the effective damping coefficient at the actuator shaft.

To express the torque relation at the load shaft, eliminate $\ddot{\theta}_m$ and $\dot{\theta}_m$ among (1.13), (1.14) and (1.18) to obtain

$$\tau_m/n = \left[\left(J_a + J_m\right)/n^2 + J_l\right]\ddot{\theta}_s + \left(B_m/n^2 + B_l\right)\dot{\theta}_s \tag{1.19}$$

where $\left[\left(J_a + J_m\right)/n^2 + J_l\right]$ is the effective inertia and $\left(B_m/n^2 + B_l\right)$ is the effective damping coefficient at the load shaft. τ_m/n is the equivalent generated torque at the load shaft.

The electrical actuators that used in industrial robots are armature controlled and their schematic diagram is shown in Figure 1.4 (and in Figure 1.2). In this figure, e_b is the back electromotive force (EMF) in volts in the armature windings which can be represented by

$$e(t) = K_b \dot{\theta}_m(t) \tag{1.20}$$

where K_b is the back EMF constant. Let L and R be the inductance of the motor armature windings, respectively. Then by applying Kirchhoff's voltage law to the armature circuit, one obtains

$$v(t) - e(t) = L\frac{di(t)}{dt} + Ri(t) \tag{1.21}$$

which has an equivalent equation in frequency domain through Laplace transformation

$$V(s) - K_b s\Theta_m(s) = (Ls + R)I(s) \tag{1.22}$$

where s is the complex frequency in rad/s. The dc motor is operated in its linear range so that the generated torque is proportional to the armature current. the relation in the frequency domain is

$$T_m(s) = K_I I(s) \tag{1.23}$$

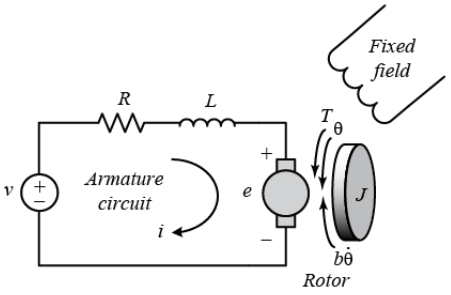


Fig. 1.4: DC motor electrical diagram

where K_I is the torque constant. The motor shaft is mechanically connected to the actuator-gear-load assembly, as indicated in Figure 1.4, with an effective inertia J_{eff} and effective damping coefficient B_{eff} at the actuator shaft. The relation among the mechanical components is described by (1.18) which has Laplace transform equivalence

$$\begin{aligned} T_m(s) &= \left[\left(J_a + J_m + n^2 J_l\right) s^2 + \left(B_m + n^2 B_l\right)\right] \Theta_m(s) \\ &= \left(J_{eff} s^2 + B_{eff} s\right) \Theta_m(s) \end{aligned} \tag{1.24}$$

Eliminating $T_m(s)$ and $I(s)$ in (1.22)-(1.24)

$$\frac{\Theta_m(s)}{V(s)} = \frac{K_I}{s\left[L J_{eff} s^2 + \left(R J_{eff} + L B_{eff}\right) s + \left(R B_{eff} + K_I K_B\right)\right]} \tag{1.25}$$

which is the transfer function, from the applied voltage to the dc motor (input), to the angular displacement of the motor shaft (output).

1.1.3 Relation between K_I and K_b

Although functionally the torque constant K_I and the back-emf constant K_b are two separate parameters, for a given motor, their values are closely related. To show the relationship, we write the mechanical power developed in the armature as

$$P = e(t)i_a(t) \tag{1.26}$$

The mechanical power is also expressed as

$$P = \tau_m(t)\dot{\theta}_m(t) \tag{1.27}$$

where, in SI units, $T_m(t)$ is in $N \cdot m$ and $\omega_m(t)$ is in rad/sec. Now substituting Eqs. 1.18 and 1.20 in Eq. 1.26, we get

$$P = \tau_m(t)\dot{\theta}_m(t) = K_b\dot{\theta}_m(t)\frac{\tau_m(t)}{K_I} \tag{1.28}$$

for which we get

$$K_b(V/rad/sec) = K_i(N \cdot m/A) \tag{1.29}$$

1.2 System Identification

System Identification Toolbox provides MATLAB functions, Simulink blocks, and an app for constructing mathematical models of dynamic systems from measured input-output data. It lets you create and use models of dynamic systems not easily modeled from first principles or specifications.

The toolbox provides identification techniques such as maximum likelihood, prediction-error minimization (PEM), and subspace system identification. To represent nonlinear system dynamics, you can estimate Hammerstein-Weiner models and nonlinear ARX models with wavelet network, tree-partition, and sigmoid network nonlinearities. The toolbox performs grey-box system identification for estimating parameters of a user-defined model. You can use the identified model for system response prediction and plant modeling in Simulink. The toolbox also supports time-series data modeling and time-series forecasting.

1.2.1 Data acquisition

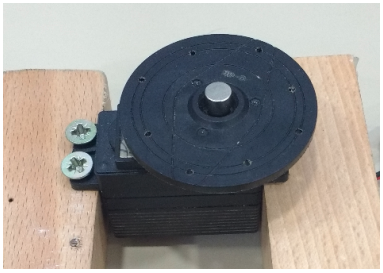
The most important thing in any system identification is the data someone has in order to use them in the estimation and validation process. In order to collect this data, various experiments were conducted. The input of the system (Voltage) as well as the output (Position) were measured. To help the process of collecting data, codes providing communication between MATLAB and Arduino are provided.

The setup is simple. Arduino and MATLAB are communicating through the *Serial* interface. The sensor used to measure the *position* is the *AS5145*. Its functionality is described with details in Section 2.1. In short, it is a magnetic encoder that senses the rotation of a bipolar magnet on top of it. The magnet is attached on the shaft on the outer part of the servo motor as shown in Fig. 2.7a. The sensor is positioned in a close distance to the magnet in a kind of random alignment as shown in Fig. 1.5d. That way it is able to test also the misalignment limits of the sensor (Section 2.1.9). The motor is driven using the *VNH5180A-E* full bridge, Fig. 1.5b. The use of the driver is described in Section 2.2. "*Dummy*" boards for both the driver and the sensor were designed in order to connect them easily with the Arduino board. In Fig. 2.7 the whole setup is presented. The input of the system, the applied voltage, is calculated from the applied PWM signal.

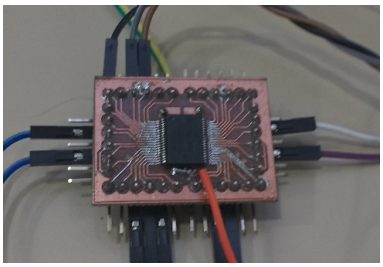
The Arduino code as well as the MATLAB code are provided with this report. The "concept" of the implementation is briefly explained.

The "Arduino side", is receiving the desired input (PWM *duty cycle* and *direction*), it applies it to the system, measures the current position and sends it back to the "MATLAB side". The communication is initiated every time from Arduino, as in a microcontroller environment it is possible to achieve accurate fixed sampling time. The way to achieve this accurate timing is described in Section 2.3.2.

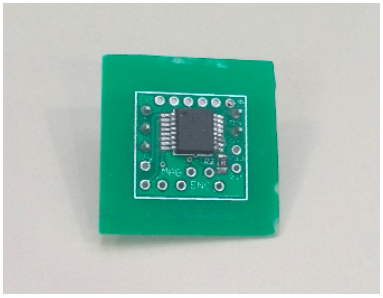
The "MATLAB side" is firstly loading a Simulink file, in which the user can create its own input signals. Then translates the desired signal (in every sampling time) to two bytes namely, the PWM *duty cycle* and the *direction* byte. Afterwards it open the Serial port and waits for "Arduino side" to initiate the communication.



(a) Driver test board



(b) Driver test board, populated



(c) Sensor test board

(d) Sensor test board

Fig. 1.5: The setup of the experiment

The files that are needed to conduct someone an experiment are:

- collect_data.m
- collect_data_buffer.m
- simuling_signal_generator.slx
- plot_data.m
- collect_data_v1_0.ino

The user must first upload the *collect_data_v1_0.ino* file to the Arduino board and then (assuming signals were created in the *simuling_signal_generator.slx*) run the *collect_data.m* file in MATLAB. After the end of the experiment he can run the *plot_data.m* to observe the results. The communication is not optimised as it was made only to serve for collecting data. After every experiment, the Arduino needs to be restarted and before re-run the *collect_data.m* file the user must "*clear all*" the MATLAB variables.

1.2.2 Data preparation

A set of data is also provided with this report, even though someone can perform its own experiments, as described in Section 1.2.1. The experiments for the provided data were conducted at $7.4V$ and *sampling time* at $20ms$.

There were used three different input signals in order to test different -realistic- dynamics. Fig. 1.6 shows the three input/output data sets.

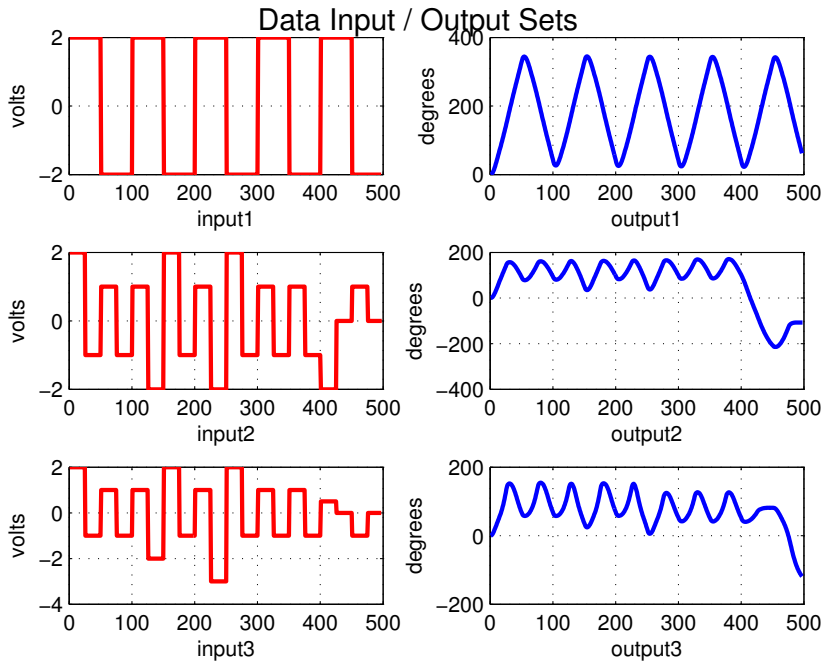


Fig. 1.6: Input-Output Data set

The first data set was selected in order to estimate the transfer function. The data are in the *collect_data2.mat* file,

Listing 1.1 Load the data

```
% Load the input/output data
load('estimation_data');
```

which contains the *input1* and *out_rel_deg* matrices.

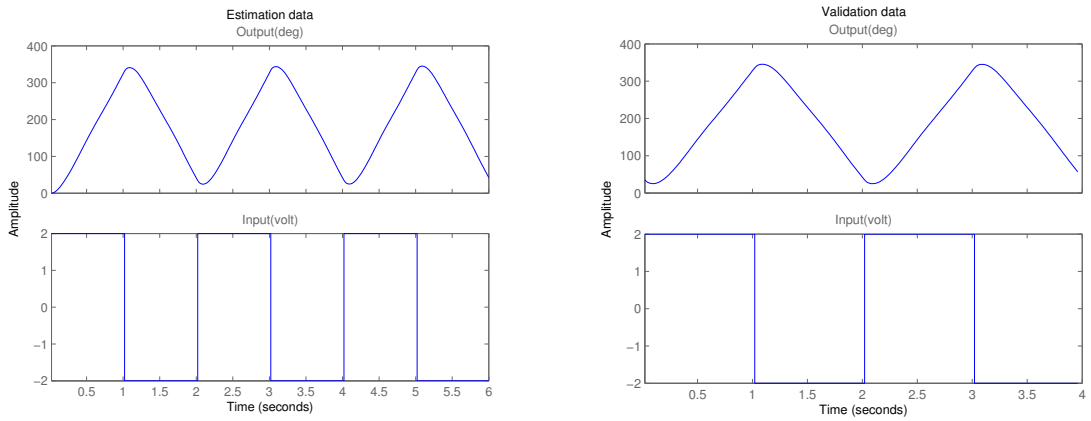
The System Identification Toolbox data object iddata, encapsulate data values and data properties into a single entity. The System Identification Toolbox commands can be used, to conveniently manipulate these data objects as single entities. One part of the data will be used for the estimation and the rest for validation.

Listing 1.2 Creating iddata

```
%Create identification iddata
ze = iddata(out_rel_deg(1:300,1),input1(1:300,1),Ts);
%Properties of ident. data
set(ze,'InputName','Input(volt)','OutputName','Output(deg)',...
    'InputUnit','Volt','OutputUnit','degrees','TimeUnit','seconds');
```

```
%Create validation iddata
zv = iddata(out_rel_deg(301:end,1),input1(301:end,1),Ts);
%Properties of validation data
set(zv,'InputName','Input(volt)','OutputName','Output(deg)',...
    'InputUnit','Volt','OutputUnit','degrees','TimeUnit','seconds');
%Plot
figure; plot(ze); title('Identification data');
figure; plot(zv); title('Validation data');
```

And the resulting plots,



(a) Estimation Data **(b)** Validation Data

Fig. 1.7: The data set seprated

1.2.3 Estimating the Empirical Step Response

Frequency-response and step-response are nonparametric models that can help someone understand the dynamic characteristics of the system. These models are not represented by a compact mathematical formula with adjustable parameters. Instead, they consist of data tables. To estimate the step response from the data, first estimate a non-parametric impulse response model (FIR filter) from data and then plot its step response.

```
%% Estimating the Empirical Step Response

% model estimation
Mimp = impulseest(ze);
% empirical step response
figure, step(Mimp);
```

As we can see from Fig. 1.8 , the response of the model shows that it might be a first order system or an overdamped function.

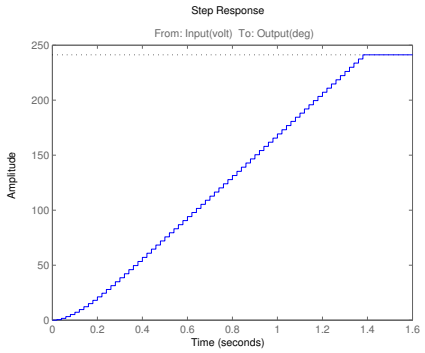


Fig. 1.8: Empirical step response

1.2.4 Estimating Input/Output delays

To identify parametric black-box models, the input/output delay must be specified as part of the model order. If the input/output delays for the system are not known from the experiment, the System Identification Toolbox software can be used, to estimate the delay.

Listing 1.3 Estimation of Input-Output delays

```
%Estimate delay
estimated_delay = delayest(ze) %ans=1 -> 1*Ts = 20ms delay
```

As it was expected, the result is $1 * Ts$ delay.

1.2.5 Estimate Transfer Function

At this point the data are prepared for the estimation of the transfer function. The only choice left, is the number of poles. For $np = 3$ the result is as shown in Fig. 1.9.

Listing 1.4 Transfer function estimation

```
%% ESTIMATE TRANSFER FUNCTION
Opt = tfestOptions('Display', 'on');
% # of poles
np = 3;
% delay
ioDelay = estimated_delay*Ts;
% Estimate the transfer function
mtf = tfest(ze, np, [], ioDelay, Opt);
figure, step(mtf);

figure, compare(zv, mtf)
```

The estimated transfer function is,

$$mtf = \frac{67.56s^2 + 1893s + 4.252e^4}{s^3 + 27.94s^2 + 231.4s + 4.586e^{-11}} \tag{1.30}$$

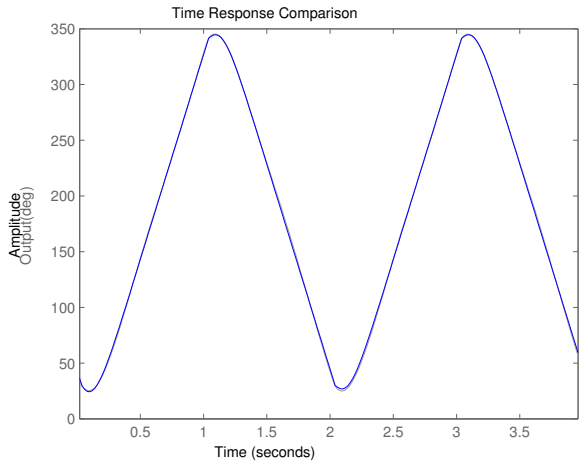


Fig. 1.9: Validation Data fit to the transfer function

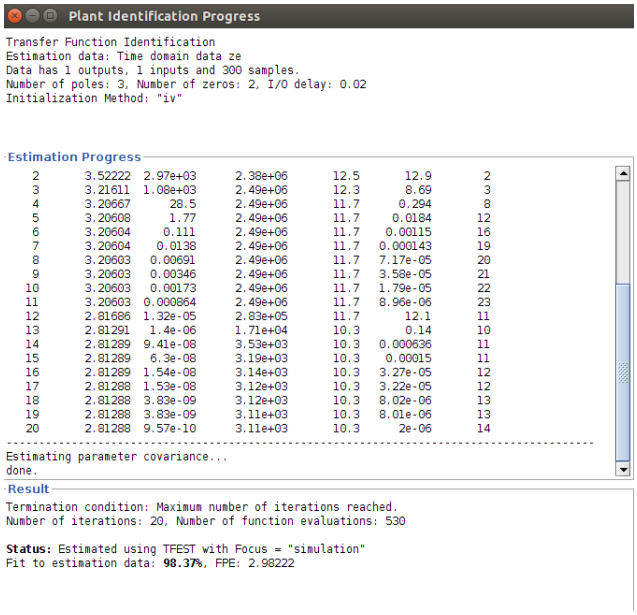


Fig. 1.10: Estimation process

In Fig. 1.9 and 1.10 it is observed that the fit of the validation data reaches the 98.37%. The reason for this very good result is that the dynamics of the validation data are the same as the estimation data. In order to test the transfer function with different dynamics, a simple Simulink model was created, only this time, the second input data was selected as the input to the transfer function. Fig. 1.11 shows the model.

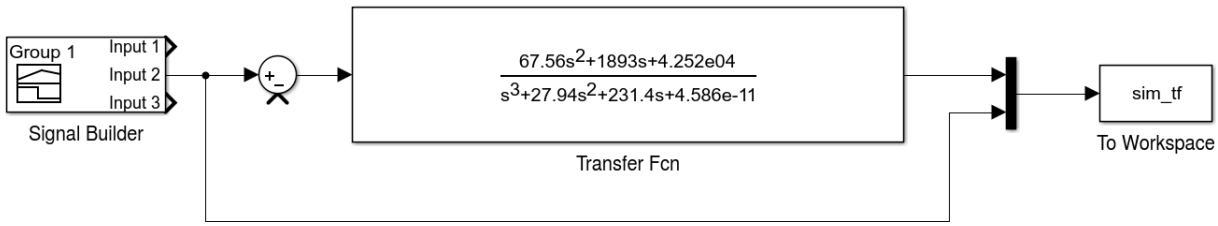


Fig. 1.11: Simulink mode to validate the estimated transfer function

In Fig. 1.12 the comparison between the experiment's output and simulation's output is shown. The estimated transfer function it may not follow the dynamics as well as with the previous data set but the result is still good and suggests, that the estimated transfer function can be used for a controller design.

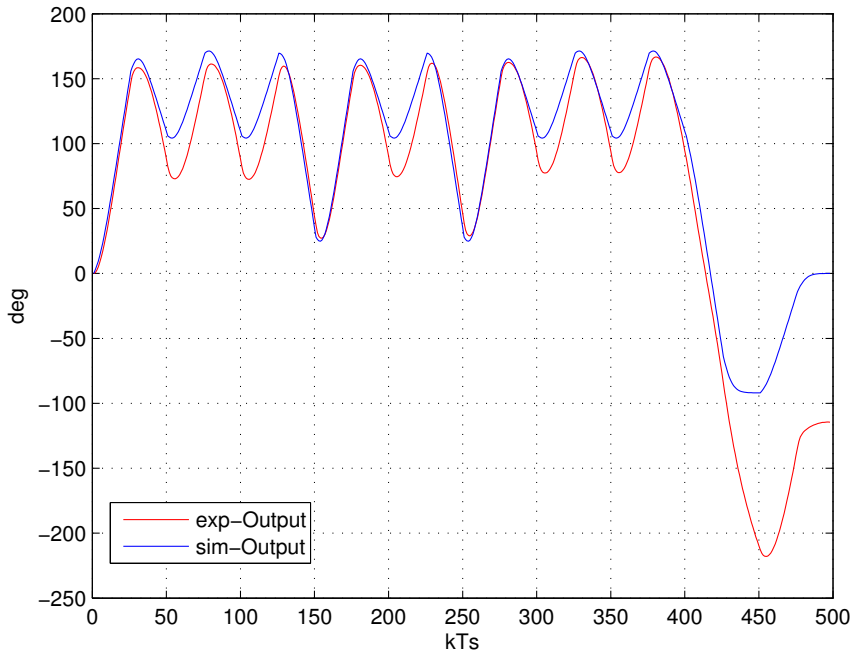


Fig. 1.12: Validation of estimated transfer function

1.3 Parameter Estimation

As it was shown in Section 1.2 it was possible to someone estimate a transfer function considering the system as a black box. If the mathematical model of the system is known, it is possible to estimate the parameters of the model, using again input-output data. On this section, this process will be described.

1.3.1 Simulink model

The mathematical model that was derived in Section 1.1.2 is going to be used. The implementation of this model in Simulink is pretty straightforward,

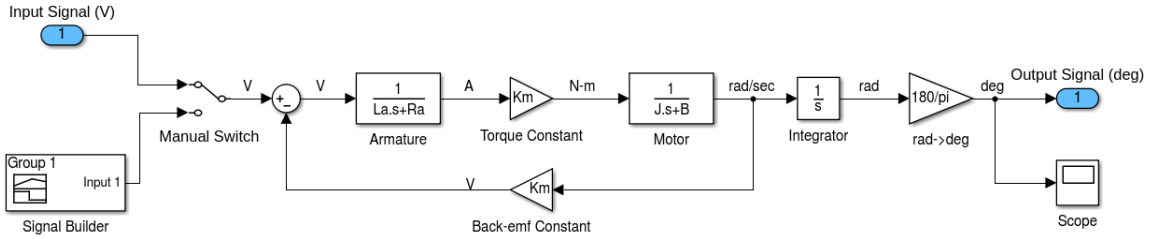


Fig. 1.13: Mathematical model of the motor

The parameters of the system that we want to estimate are:

B	effective damping coefficient, $N–m\ s/rad$,
J	effective inertia, $N–m\ s^2/A$,
K_m	torque constant, $N–m/A$,
L_a	armature inductance, H
R_a	armature resistance, Ω.

In order to be able to continue we need to define initial values for these parameters in the workspace.

```
>> B = 0.008;
>> J = 5.7e-07;
>> Km = 0.0134;
>> La = 6.5e-05;
>> Ra = 1.9;
```

The same data set will be used, as in Section 1.2, namely *collect_data2*. The data set contains 6 matrices.

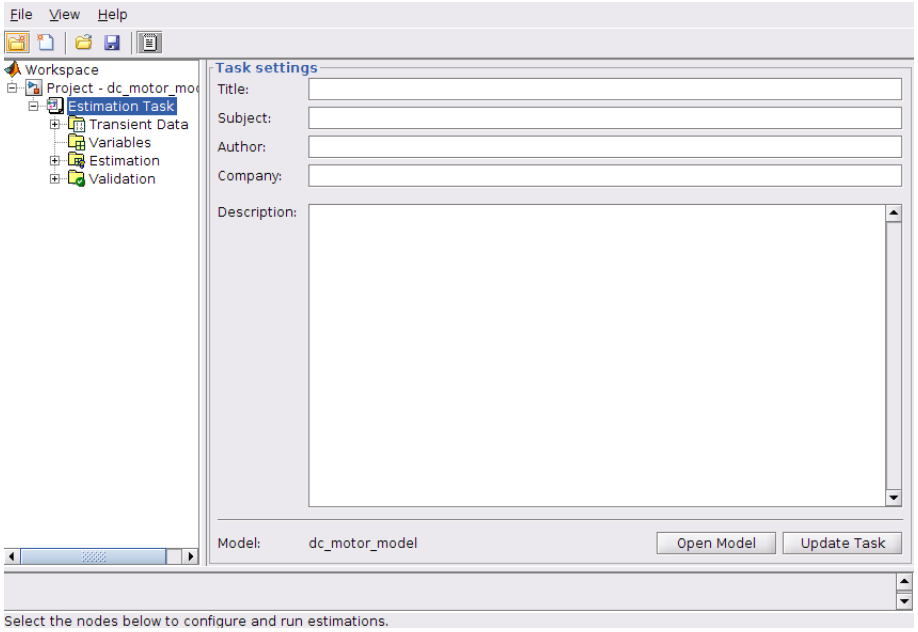
inp_duty	The duty cycle of the pwm signal.
inp_v_pwm	The duty cycle translated in $[0-5]$ volts.
inp_volt	The output voltage of the driver, the input voltage to the motor.
out_abs	The absolute output of the system expressed to ticks per revolution.
out_rel	The relative output of the system expressed to $[0-4098]$ ticks per revolution.
out_rel_deg	The relative output of the system expressed to $[0-360]$ degrees per revolution.

Table 1.1: Data set matrices

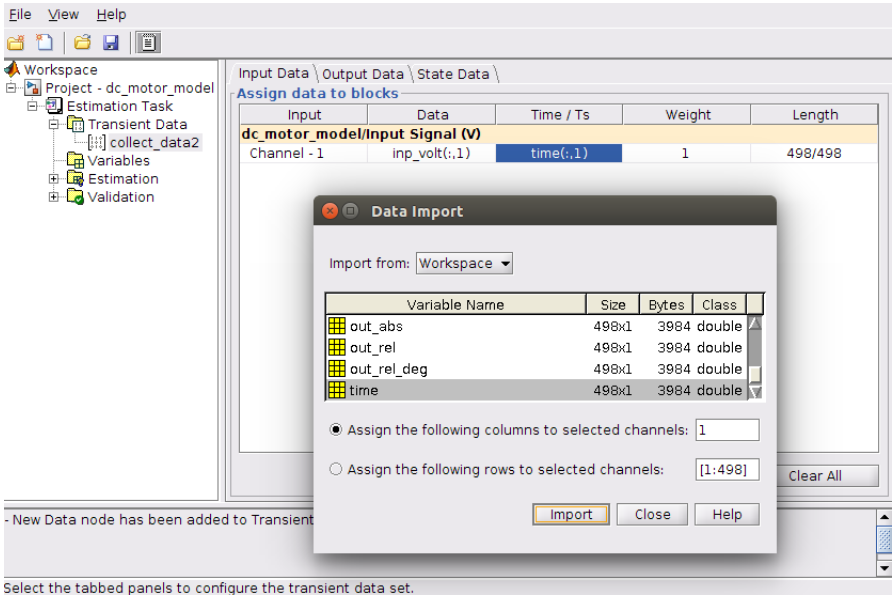
1.3.2 MATLAB Parameter Estimation Toolbox

The input of the model 1.13 is in voltage and this voltage is the output of the driver of the motor. Therefore the *input data* that will be used will be the *inp_volt*. The chosen output is in degrees therefore the selected *output data* is *out_rel_deg*.

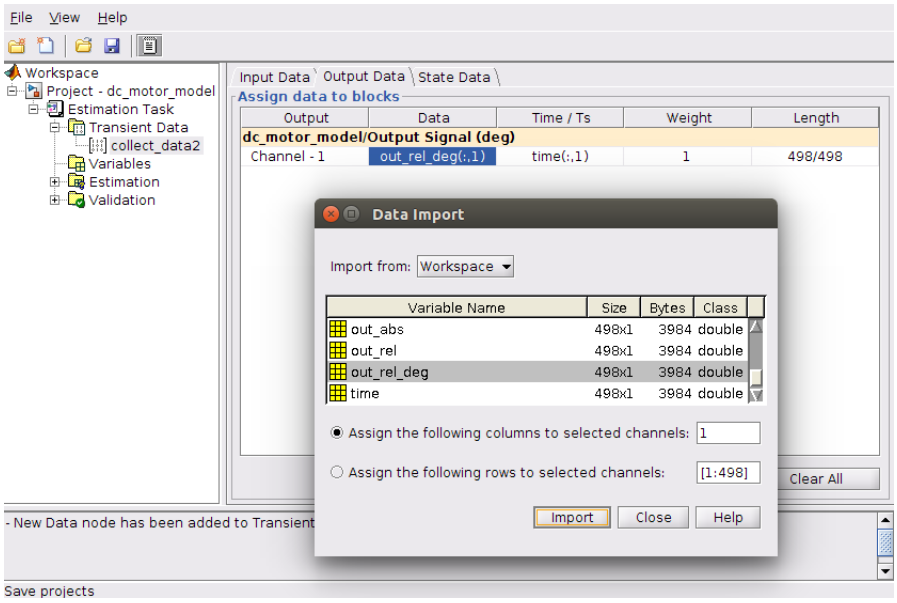
By selecting *Analysis* → *ParameterEstimation* it opens the *Parameter and Estimation Tools Manager* main window. In the first tab the user can add some information such as the title and author of the project.



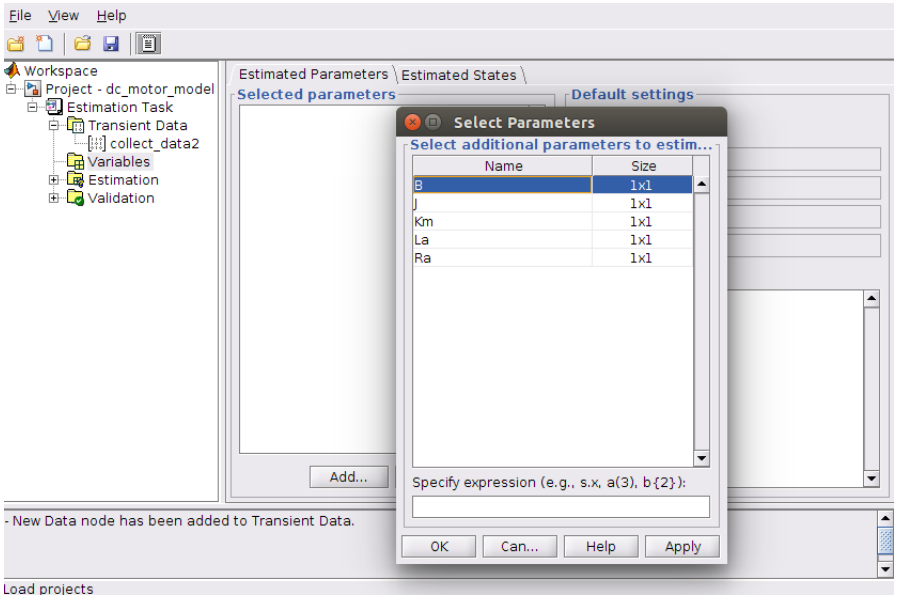
In the *Transient Data tab* a new *data* was created named *collect_data2*. And now the user is able to add the desired input/output data. First the input,

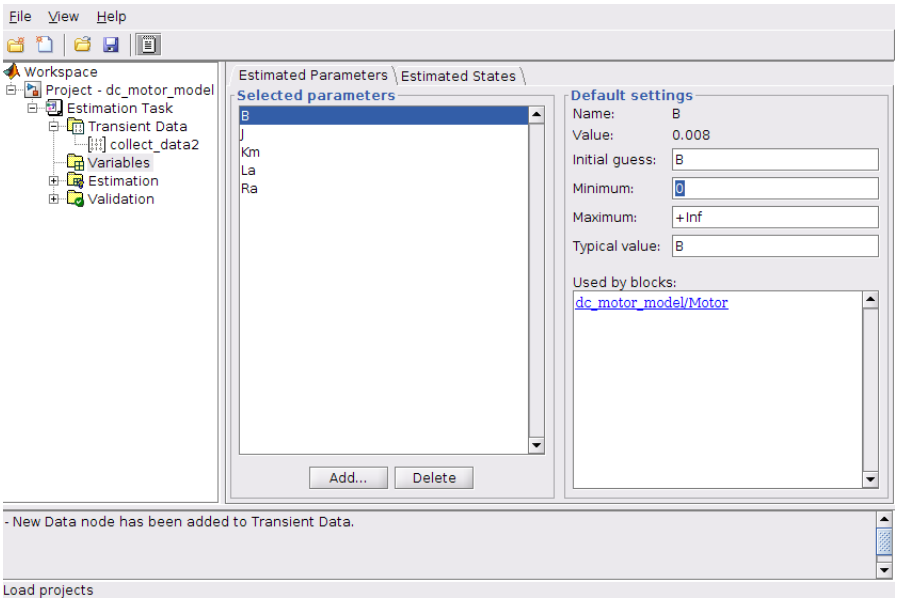


and then the output

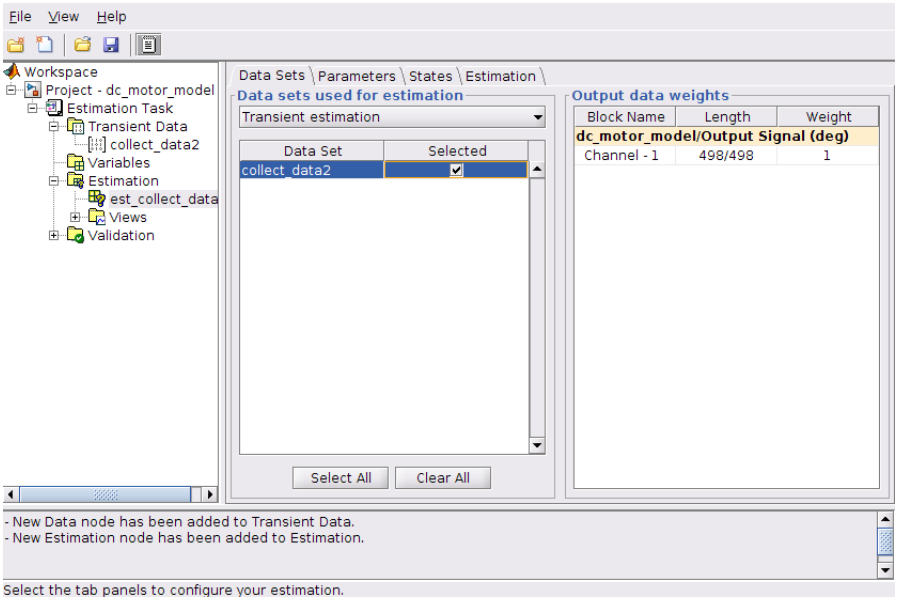


Next tab is *Variables*, where by selecting *Add* the user can choose which parameters wants to estimate. Since all these parameters are physical quantities, their minimum values are set to zero, as they cannot have negative values.

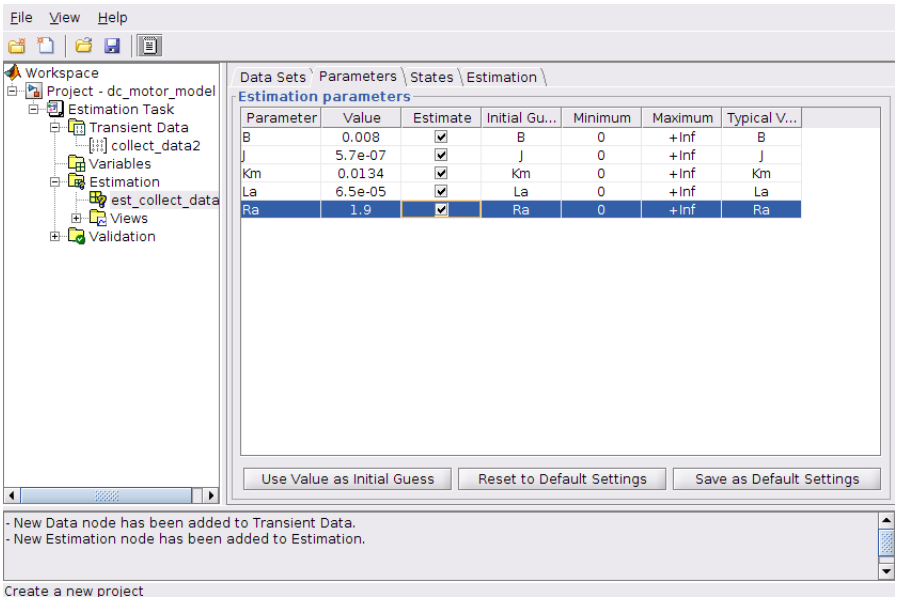




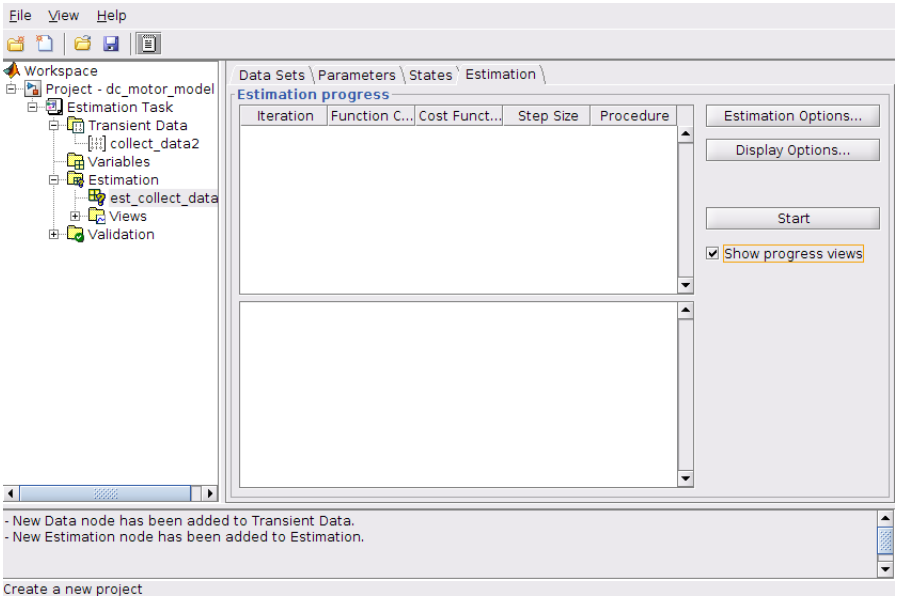
Now that the data are imported and the variables to estimate are chosen, in the *Estimation* tab a new entry created called *est_collect_data2*. There, the imported data are chosen,



as well as the parameters,



and then everything is ready for the estimation process to begin. Before of that, by selecting the *Show progress views*, the user can see the parameters trajectory and the result of the estimation during the process.



The result of the *Parameter Estimation* is shown in Fig. 1.14 and Fig. 1.15. It is shown that the trajectories converged relatively fast and the fit to the data is very good.

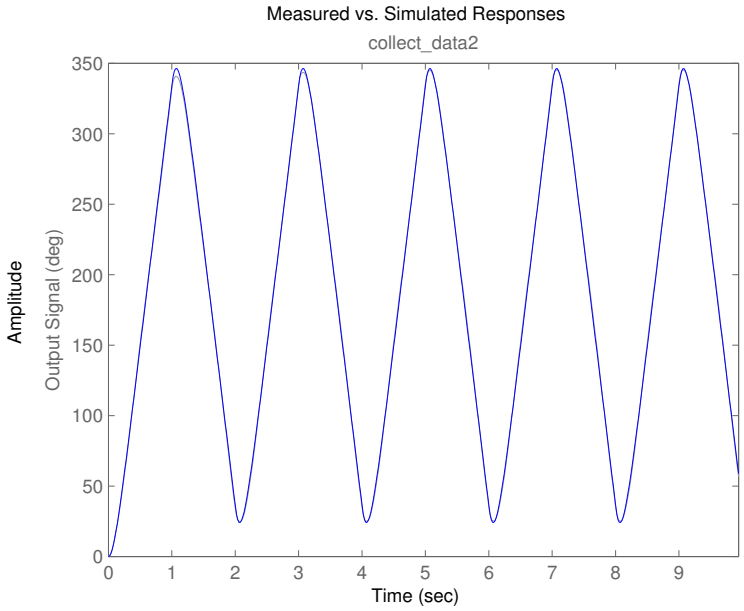


Fig. 1.14: Simulation vs. Measured Responses

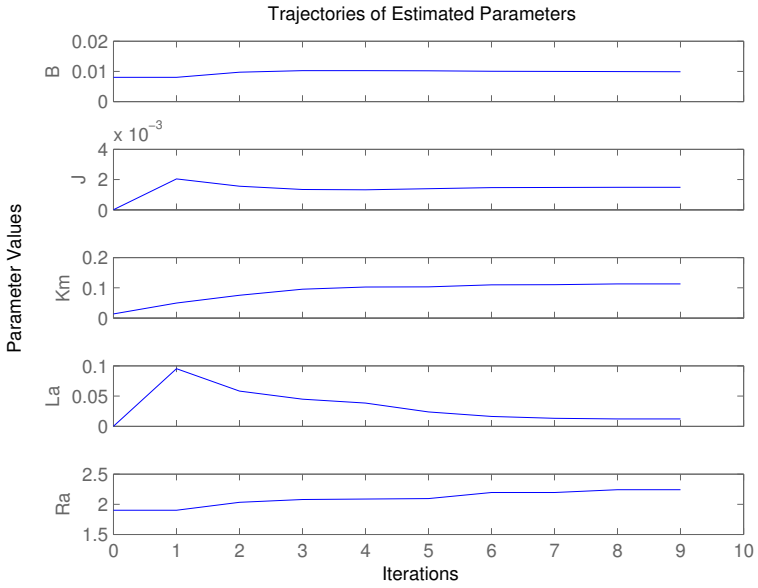


Fig. 1.15: Trajectories of Estimated Parameters

The estimated parameters are,

B	0.0098949
J	0.0014829
Km	0.11262
La	0.012109
Ra	2.2397

Table 1.2: Estimated parameters

1.3.3 Validation

In order to validate the estimated parameters, the user only has to connect the *Manual Switch* to the second input of the *Signal Builder* (which is the same one used for the data acquisition) and run the simulation, as shown in Fig. 1.16.

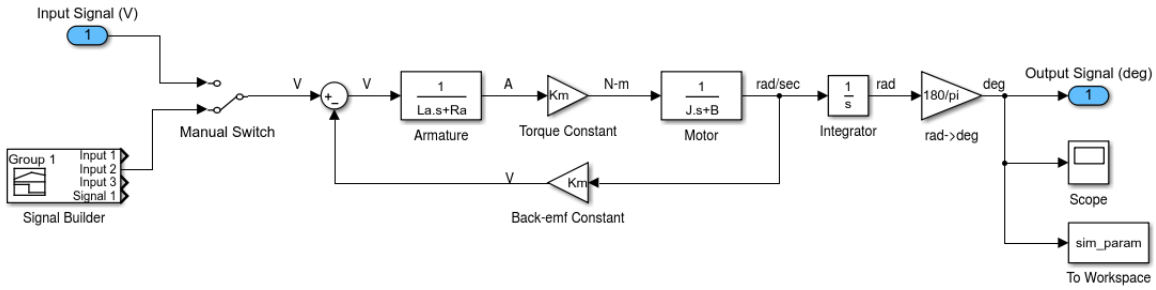


Fig. 1.16: Validation model

The result is compared with the *out_rel_deg* of data set *collect_data_2_1.mat*. As it can be seen in the Fig. 1.17 the result is not that good as in the case of transfer function identification. One of the reason for that, could be the unmodelled frictions in the motor. Also, the input of the system, is the *input_volt*. This voltage, that is the output of the driver of the motor, is not measured but calculated from the known PWM duty cycle. Finally, the simulink model does not contain the gear ratio. The estimated parameters are such as the effect of the gears is included in the model. All these result in not a such good fit on the measured dynamics but still good enough for someone to design a controller.

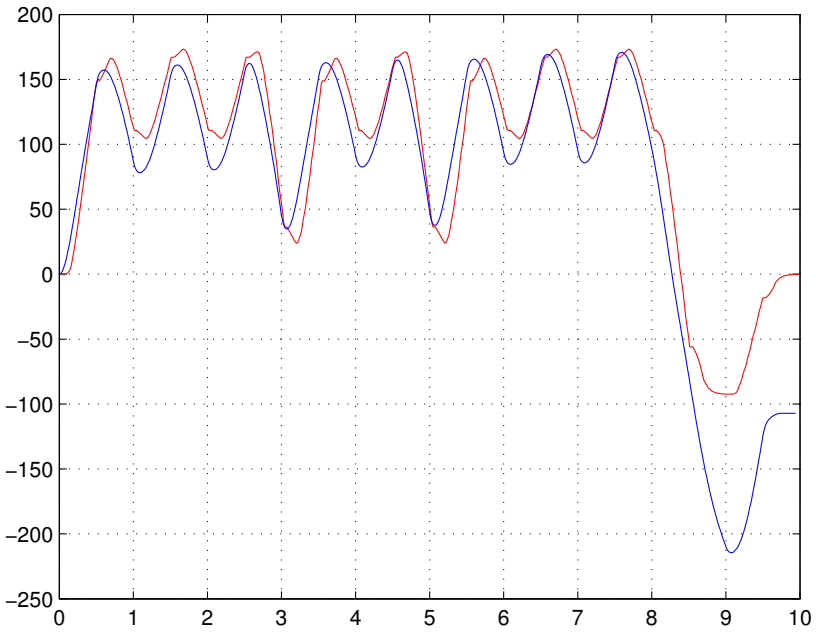


Fig. 1.17: Validation of the estimated parameters

1.4 Control

PID controllers have survived many changes in technology, from mechanics and pneumatics to microprocessors via electronic tubes, transistors, integrated circuits. The microprocessors has had a dramatic influence on the PID controller. Practically all PID controllers made today are based on microprocessors. This has given opportunities to provide additional features like automatic tuning, gain scheduling, and continuous adaptation.

1.4.1 The Algorithm

The basic algorithm will be summarised here as it is described in [**?**] and [**?**].

The "textbook" version of the PID algorithm is described by:

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right) \tag{1.31}$$

where y is the measured process variable, r the reference variable, u is the control signal and e is the control error ($e = y_{sp} - y$). The control signal is thus thus a sum of three terms: the P-term (which is proportional to the error), the I-term (which is proportional to the integral of the error), and the D-term (which is proportional to the derivative of the error). The controller parameters are proportional gain K, integral time T_i, and derivative time T_d.

The transfer function of a PID controller in the s-domain is expressed as:

$$\frac{U(s)}{X(s)} = G_c(s) = K_P + \frac{K_I}{s} + K_D s \tag{1.32}$$

A digital implementation of this controller can be determined by using a ***discrete approximation*** for the *derivative* and *integration*.

Numerical Differentiation

One of the simplest implementation of time derivative, is with the use of the **backward difference rule**

$$u(k) = \frac{1}{T}\left(x(k) - x(k-1)T\right) \tag{1.33}$$

The z-transform of Equation 1.33 is

$$U(z) = \frac{1 - z^{-1}}{T} X(z) = \frac{z - 1}{Tz} X(z) \tag{1.34}$$

Numerical Integration

The integration of $x(t)$ can be represented by the **forward-rectangular integration**

$$u(k) = u(k-1)T + Tx(k) \tag{1.35}$$

The z-transform of Equation 1.35 is

$$U(z) = z^{-1}U(z) + TX(z) \Leftrightarrow \frac{U(z)}{X(z)} = \frac{Tz}{z - 1} \tag{1.36}$$

PID Algorithm Implementation

Putting all together, the z-domain transfer function of the **PID controller** is

$$G_c(z) = K_P + \frac{K_I T z}{z - 1} + K_D \frac{z - 1}{Tz} \tag{1.37}$$

The complete difference equation algorithm that provides the PID controller is obtained by adding the three terms to obtain

$$u(k) = \boldsymbol{K_P} x(k) + \boldsymbol{K_I} \left[u(k-1) + Tx(k)\right] + \left(\frac{\boldsymbol{K_D}}{T} \left[x(k) - x(k-1)\right]\right) \tag{1.38}$$

Equation 1.38 can be easily implemented in a microcontroller using the tools provided in this report. Of course, someone can obtain a *PI* or *PD* controller by setting the appropriate gain equal to zero.

Chapter 2

Implementation

On this chapter it will be discussed the implementation of the electronics of the servo motor that will allow the user to drive the motor and read the position of the rotor using a microcontroller. In first section the sensor for the reading of the position will be presented while in second section it will be presented the driver that was chosen to run the motor. Finally in section three it is presented the microcontroller that is used (Arduino) and the way to use it with the driver and the sensor.

2.1 Magnetic Encoder

In order to read the position of the motor, the rotary magnetic encoder AS5145 from *Austria Microsystems (AMS)* [**?**] was chosen. The AS5145H is a contactless magnetic rotary position sensor for accurate angular measurement over a full turn of 360° and over an extended ambient temperature range of -40°C to 150°C. The *absolute* angle measurement provides instant indication of the magnet's angular position with a resolution of 0.0879° = 4096 positions per revolution via a serial bit stream and as a PWM signal.

2.1.1 General Description

The *AS5145* is a contact-less magnetic encoder for accurate angular measurement over a full turn of 360 degrees. It is a system-on-chip, combining integrated Hall elements, analog front end and digital signal processing in a single device.

To measure the angle, only a simple two-pole magnet, rotating over the center of the chip, is required. The magnet can be placed above or below the IC. The *absolute angle measurement* provides instant indication of the magnet's angular position with resolution of 0.0879 deg = 4096 positions per revolution. This digital data is available as a serial bit stream and as a PWM signal.

2.1.2 Key Features

- Contact-less high resolution rotational position encoding over a full turn of 360 degrees.
- Two digital 12-bit absolute outputs:
  - Serial interface
  - Pulse width modulated (PWM) output
- Three incremental outputs
- User programmable zero position
- Failure detection mode for magnet placement, monitoring, and loss of power supply
- Red-Yellow-Green indicators display placement of magnet in Z-axis
- Serial read-out of multiple interconnected AS5145 devices using Daisy-Chain mode
- Tolerant to magnet misalignment and gap variations

2.1.3 Interfacing with Arduino

In order to read the position of the rotor, the **S**ynchronous **S**erial **I**nterface (SSI) was used. The schematic of the IC in Figure 2.1, shows which pins need to be used.

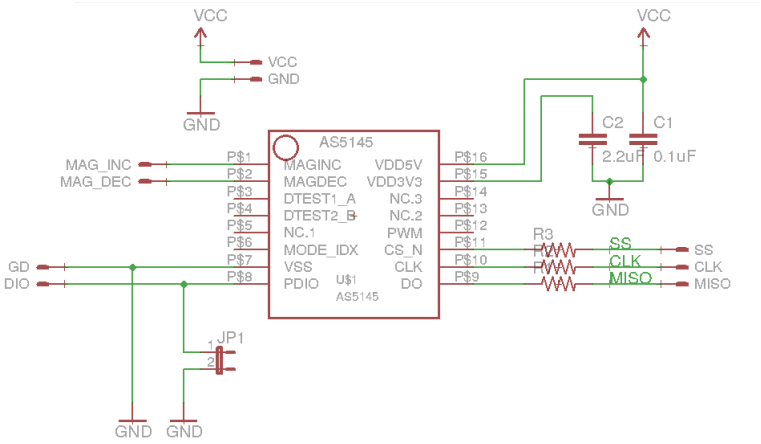


Fig. 2.1: Schematic of AS5145

Where the pins of interest are,

VDD5V	5V power supply pin.
CSn	Chip Select. Active low.
CLK	Clock input of SSI.
DO	Data Output of SSI.
VSS	GND

And the connection between AS5145 pins and Arduino pins are,

AS5145		Arduino
CSn	↔	D10
DO	↔	D12 (MISO)
CLK	↔	D13

where, for example, D12 stands for Digital Pin 12.

2.1.4 Synchronous Serial Interface (SSI)

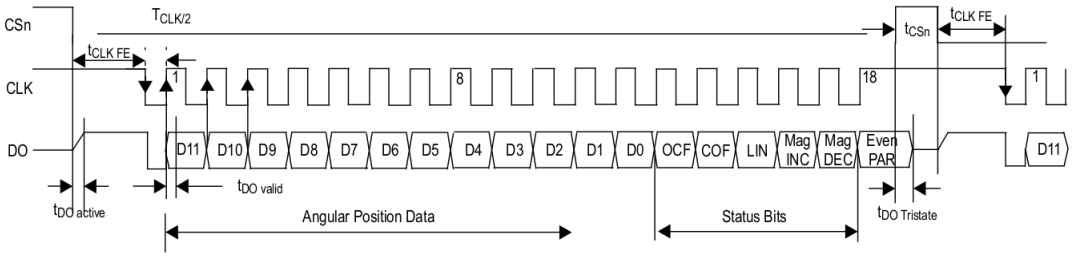


Fig. 2.2: SSI Interface

If CS_n changes to logic low, Data Out (DO) will change from high impedance (tri-state) to logic high and the read-out will be initiated.

- After a minimum time t_{CLKFE}, data is latched into the output shift register with the first falling edge of CLK.
- Each subsequent rising CLK edge shifts out one bit of data.
- The serial word contains 18 bits, the first 12 bits are the angular information $D[11:0]$, the subsequent 6 bits contain system information, about the validity of data.
- A subsequent measurement is initiated by "high" pulse at CSn with a minimum duration of t_{CSn}.

2.1.5 Data Content

- **D11:D0** absolute position data (MSB is clocked out first
- **OCF** (Offset Compensation Finished), logic high indicates the finished Offset Compensation Algorithm.
- **COF** (Cordic Overflow), when the bit is set, the data D11:D0 is invalid. The absolute output maintains the last valid angular value. This alarm can be resolved by bringing the magnet within the X-Y-Z tolerance limits.
- **LIN** (Linearity Alarm), logic high indicates that the input field enerates a critical output linearity. When the bit is set, the data D11:D0 can still be used, but can contain invalid data. This alarm can be resolved by bringing the magnet within the X-Y-Z tolerance limits.
- **EVEN PARITY** bit for transmission error detection of bits 1...17 (D11...D0, OCF, COF, LIN, MagINC, MagDEC).

Data D11:D0 is valid, when the status bits have the following configurations

OCF	COF	LIN	MagINC	MagDEC	Parity
1	0	1	0	0	Even check sum of bits 1:15
			0	1	
			1	0	
			1	1	

2.1.6 Read the position

This is the function to read the sensor.

Listing 2.1 Arduino function to read the position of the rotor

```
uint32_t readSSI () {
    uint32_t data;
    //Pulse to initiate new transfer
    digitalWrite(10,HIGH);
    digitalWrite(10,LOW);
    //Receive the 3 bytes (AS5145 sends 18bit word)
    for (u8byteCount=0; u8byteCount<3; u8byteCount++){
       u32result <≤ 8;  // left shift the result so far - first time shifts ...
    0's-no change
       SPDR = 0xFF;  // send 0xFF as dummy (triggers the transfer)
       while ( (SPSR & (1 << SPIF)) == 0);  // wait until transfer complete
       u8data = SPDR;  // read data from SPI register
       u32result |= u8data;  //store the byte
    }
    // Print only the data no check of flags!
    u32result >≥ 12;
    data = u32result;
    u32result = 0;
    return data;
}
```

And an example of calling it,

Listing 2.2 readSSI() call example

```
uint_32t pos = readSSI();
```

2.1.7 Selecting Proper Magnet

Typically the magnet is $6mm$ in diameter and $2.5mm$ in height. Magnetic materials such as rare earth AINiCo/SmCo5 or NdFeB are recommended. The magnetic field strength perpendicular to the die surface has to be in the range of $\pm 45mT ... \pm 75mT$ (peak). The magnet's field is verified using a gauss-meter. The magnetic field Bv at the given distance, along a concentric circle with radius of $1.1mm$ (R1) is in the range of $\pm 45mT ... \pm 75mT$ (see Figure 2.3)

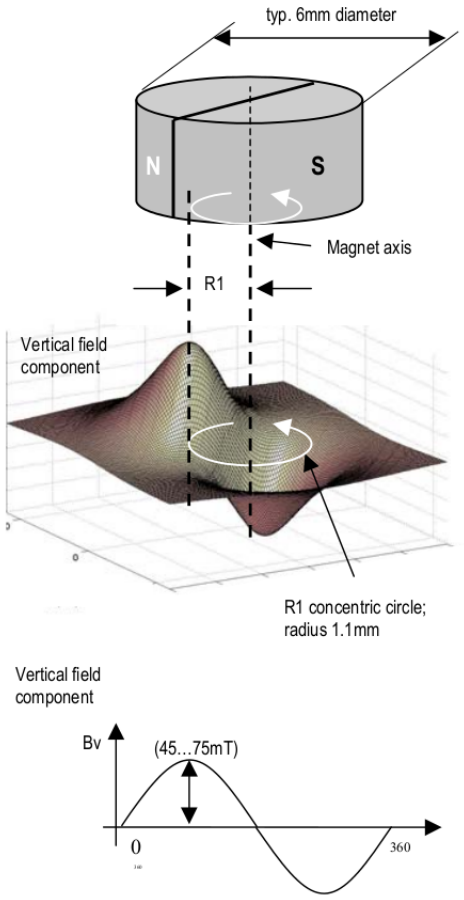


Fig. 2.3: Typical magnet (6x3) and Magnetic Field Distribution

2.1.8 Physical Placement of the Magnet

The best linearity can be achieved by placing the center of the magnet exactly over the defined center of the chip as shown in the drawing below:

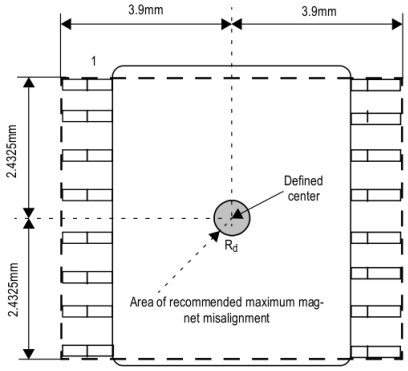


Fig. 2.4: Defined Chip Center and Magnet Displacement Radius

The magnet's center axis must be aligned within a displacement radius R_d of $0.25mm$ from the defined center of IC. The magnet can be placed below or above the device. The distance can be chosen such that the magnetic field of the die surface is within specified limits. The typical distance "z" between the magnet and the package surface is $0.5mm$ to $1.5mm$, provided the use of the recommended magnet material and dimensions ($6mm \times 3mm$).

2.1.9 Alignment Mode

The alignment mode simplifies centering the magnet over the center of the chip to gain maximum accuracy.

Alignment mode can be enabled with the falling edge of *CSn* while *PDIO = logic high*. Afterwards, there are two ways to check if the magnet is proper placed.

- In alignment mode, the Data bits D11:D0 of the SSI change to a 12-bit displacement amplitude output. A high value indicates large X or Y displacement, but also higher absolute magnetic field strength. The magnet is properly aligned, when the difference between highest and lowest value over one full turn is at a minimum. Under normal conditions, a properly aligned magnet will result in a reading of less than 128 over a full turn. Stronger magnets or short gaps between magnet and IC will show values larger than 128. These magnets are still properly aligned as long as the difference between highest and lowest value over one full turn is at a minimum.
- Under normal conditions, the *MagINCn* and *MagDECn* indicators will be equal to 1 when the alignment mode reading is less than 128. At the same time, both hardware pins MagINCn

(pin 1) and MagDECn (pin 2) will be pulled to VSS. A properly aligned magnet will therefore produce a $MagINC = MagDEC = 1$ signal throughout a full 360 deg turn of the magnet.

The Alignment mode can be reset to normal operation by a power-on-reset (disconnect/reconnect power supply) or by a falling edge on CSn with PDIO = low.

2.2 H-Bridge motor driver

To drive the motor, the ***VNH5180A-E*** fully integrated H-bridge motor driver from STMicroelectronics, was selected [**?**]. The two main reasons behind this choice, were the output current of 8 A, that covers the need of most of the hobby RC-servos in the market and, the integrated current sensor.

2.2.1 General Description

The *VNH5180A-E* is a full bridge motor driver intended for a wide range of automotive applications. The device incorporates a dual monolithic high-side driver and two low-side switches. Both switches are designed using STMicroelectronics' well known and proven proprietary $VIPower$ M0 technology that allows to efficiently integrate on the same die a true Power MOSFET with an intelligent signal/protection circuitry. The three dies are assembled in PowerSSO-36 TP package on electrically isolated leadframes. This package, specifically designed for the harsh automotive environment offers improved thermal performance thanks to exposed die pads. Moreover, its fully symmetrical mechanical design allows superior manufacturability at board level.

The input signals IN_A and IN_B can directly interface to the microcontroller to select the motor direction and the brake condition. The DIAG_A/EN_A or DIAG_B/EN_B , when connected to an external pull-up resistor, enables one leg of the bridge. Each DIAG_A/EN_A provides a feedback digital diagnostic signal as well. The normal operating condition is explained in the truth Table 2.2. The CS pin allows to monitor the motor current by delivering a current proportional to its value when CS_DIS pin is driven low or left open. When CS_DIS is driven high, CS pin is in high impedance condition. The PWM, up to 20 KHz, allows to control the speed of the motor in all possible conditions. In all cases, a low level state on the PWM pin turns off both the LS_A and LS_B switches.

2.2.2 Key Features

- Output current: 8 A
- 3 V CMOS compatible inputs
- Undervoltage shutdown
- Overvoltage clamp
- Thermal shutdown
- Cross-conduction protection
- Current and power limitation
- Very low standby power consumption
- PWM operation up to 20 KHz

- Protection against loss of ground and loss of V_{CC}
- Current sense output proportional to motor current
- Output protected against short to ground and short to V_{CC}

2.2.3 Interfacing with Arduino

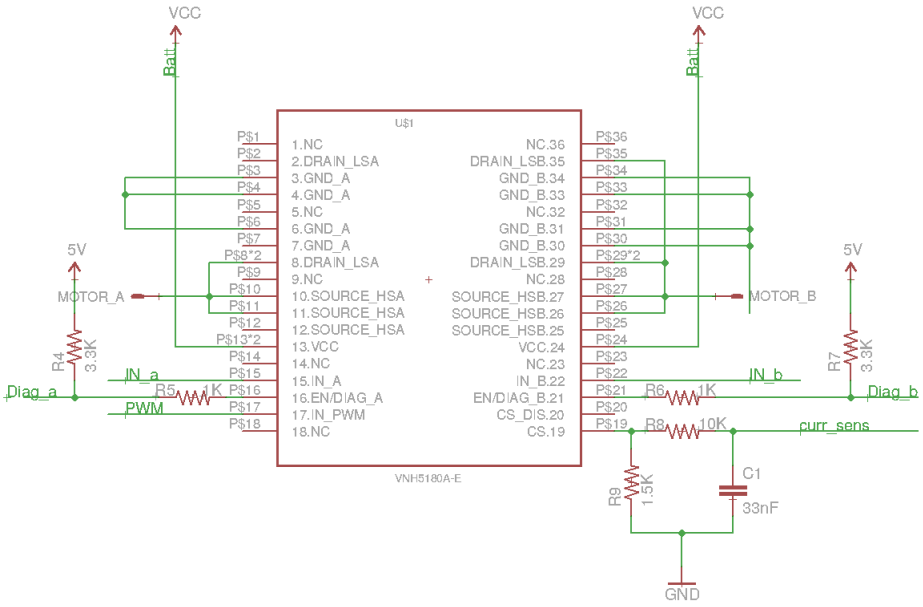


Fig. 2.5: Schematic of the VNH5180A-E

They key points in order to connect with Arduino are:

- The pins *SOURCE_HSA/B* (high side mosfet's source) and *DRAIN_LSA/B* (low side mosfet's drain), must be connected to each other. Of course the junction between them is where the motor cables are also connected.
- In normal operating conditions the *EN/DIAG_A* and *EN/DIAG_B* pins are considered as inputs by the device. They must be connected to an external pull up resistor.

Apart of these points, the connection with the Arduino is straightforward.

Table 2.1: Pin connection between Driver and Arduino

Driver		Arduino
IN_A	↔	D4
IN_B	↔	D7
IN_PWM	↔	D3

where D4 and D7 are the digital pins we chose to control the direction of the motor. The possible combination of all the pins in *normal operation* conditions are shown in Table 2.2

Table 2.2: Truth table in normal operating conditions

IN_A	IN_B	$DIAG_A/EN_A$	$DIAG_B/EN_B$	OUT_A	OUT_B	Operating mode
1	1	1	1	H	H	Brake to V_{CC}
	0				L	Clockwise (CW)
0	1			L	H	Counterclockwise (CCW)
	0				L	Brake to GND

2.2.4 Run the motor

In order to run the motor all it needs to be done, is choose the direction (CW/CCW) and apply the PWM in the proper pin. An example code is show in Listing 2.3.

Listing 2.3 Arduino code to run the motor

```
// Make sure you don't hae cross - conduction (even though
// chip has protection about it)
digitalWrite(7,LOW);  digitalWrite(4,LOW);  //Brake
digitalWrite(4,HIGH);
OCR2B = 100;
```

where *OCR2B,* is the 8-bit register of the timer is used to create the PWM signal at pin D3, as explained in Section 2.3.2

2.3 ATmega328 microcontroller - Arduino

For this application we chose the *Atmel* microcontroller *ATmega328P-AU*, which is also used to *Arduino UNO* boards [**?**]. That way the user can take advantage of all the functionalities of Arduino, such as the IDE and the libraries. The only difference with an Arduino board is that the code is uploaded through ISP instead of USB.

2.3.1 Peripheral Features

Some of the peripheral features of the micro-controller are,

- Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
- One 16-bit Timer/Counters with Separate Prescaler, Compare Mode and Capture Mode
- Real Time Counter with Separate Oscillator
- Six PWM Channels
- 8-channel 10-bit ADC in TQPF and QFN/MLF package Temperature Measurement
- 6-channel 10-bit ADC in PDIP package Temperature Measurement
- Programmable Serial USART
- Master/Slave SPI Serial Interface
- Byte-oriented 2-wire Serial Interface (Philips I^2C compatible)

2.3.2 Setup

The system clock is at 16Mhz. The connection with the sensor and the driver is already described in subsections 2.1.3 and 2.2.3 respectively.

Arduino IDE, is using a main loop which is repeating every time as soon as the code inside it is executed. For any typical digital control system, there is the need of a fixed sampling time. In order to achieve that, *Timer0* was used, to trigger an interrupt in the desired sampling time. This time will be referred as ***control loop*** from now on.

Note! The use of *Timer0* interrupt interferes with the arduino library that uses the ***delay()*** function. If the interrupt routine is used the user shouldn't use the *delay()* function any more. Even though the compiler will not find any error, the accuracy of the timing of the *delay()* command is lost.

Timer 0 for Control Loop

Timer0 is used in the *"Clear Timer on Compare Match"* or **CTC** mode. The timer has an 8-bit register called *OCR0A*. It also has a counter, *TCNT0* that, if the timer is active, it increases its value every timer-clock cycle. Whenever $OCR0A = TCNT0$ the counter goes to 0 again (on the same clock) and an interrupt is triggered.

The timer-clock can be configured from the *TCCR0B* register. The configuration of these registers give the user the choice to choose the control loop frequency.

Listing 2.4, shows the configuration of *Timer0* for this application,

Listing 2.4 Setup of Timer0 registers

```
TCCR0A = 0;
TCCR0B = 0;

TCCR0A |= B01000010;
TCCR0B |= B00000101;
// to be able to use the interrupt
TIMSK0 |= B00000010;
loop_flag = 1;

OCR0A = 78;  //with CS00:2 = 101 -> period = 0.01
// Enable global interrupts
sei();
```

The interrupt routine must be as fast as possible. All it does is to raise a boolean flag. In the control loop, after the execution of the code, this flag is turned back to LOW.

Listing 2.5 Timer0 interrupt routine.

```
/* Counter0 compare match interrupt - for control loop*/
ISR(TIMER0_COMPA_vect) {
  if (loop_flag == 0){
    loop_flag = 1;
  }
}
```

And an example of a control loop,

Listing 2.6 Example of control loop

```
void loop() {
  // loop_flag was set 1 in setup
  if (loop_flag == 1) {

    /* Your code here */
```

```
      loop_flag = 0;
    }

  }
```

After the execution of "*Your code*", the microcontroller will stay in the *loop()* doing nothing, until the *loop_flag* will be raised again from the interrupt routine, which happens in a fixed -sampling-time. For that reason "*Your code*" must be executed before the end of the *control loop*. If the user wants to check if the code exceeds this time, he can use an *else* statement in the interrupt routine.

Timer 2 for PWM generation

Apart of the use of *Timer0* for the control loop, *Timer2* is also used to generate the PWM signal that will be used to drive the motor.

Timer2 is used in the *"Fast PWM"* mode. This mode provides a high-frequency PWM waveform and the reason for that is its single-slope operation. The counter counts from BOTTOM to TOP and then restarts from BOTTOM. BOTOM is equal to 0 while TOP can be configured from the timer registers. This high frequency makes the fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), and therefore reduces total system cost.

Every time the timer overflows it toggles the state of OCOB which is the *Digital Pin 3*. Listing 2.7, shows the setup of the timer,

Listing 2.7 Timer2 setup for Fast PWM

```
/* ---------- Timer 2 - Configuration (FAST_PWM) ------------ */
  TCCR2A = 0;
  TCCR2B = 0;
  TCCR2A |= B00100011;
  TCCR2B |= B00000111;
/* ---------------------------------------------------------*/
```

The value of the 8-bit register ***OC2B*** corresponds to the PWM duty cycle. So if a PWM signal, with 50% duty cycle is required, someone could use the command of Listing 2.8.

Listing 2.8 Setup of PWM duty-cycle

```
OCR2B = 128;
```

By setting the *OCR2B* register the PWM signal generation starts.

2.4 Motor Control Board

A board that is hosting all the components described in this chapter was designed. The design of the **P**rinted **C**ircuit **B**oard was done using the *Eagle* software. The goal of this design was to replace the one that already exist in a commercial DC servo motor. Fig 2.7 shows the servo motor with its initial board and the two boards (old an new) next to each other.

Some of the advantages, compared to the old board, of this implementation are

- Higher accuracy. Before the position was sensed with a potentiometer while now with the magnetic encoder it has accuracy of 0.088 degrees.
- Wide range of applications. The new driver can deliver up to 9A which suggest that can be used for a large variety of DC servo motors.
- Current sensing through the driver chip.
- Fully programmable microcontroller.
- I2C communication allowing up to 400 kHz *Data Transfer Speed* and the option to connect up to 128 motors to the bus.

2.4.1 Main Board

The main board is the one that host the *Driver* and the *Arduino*. Fig 2.6 shows the schematic of the system.

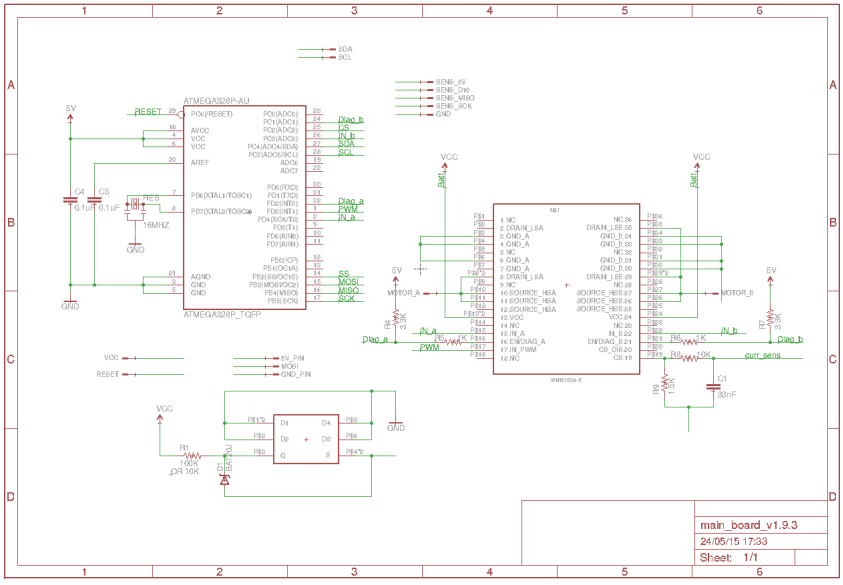
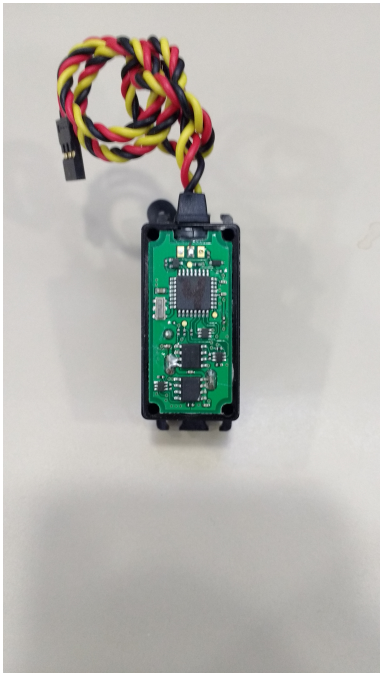


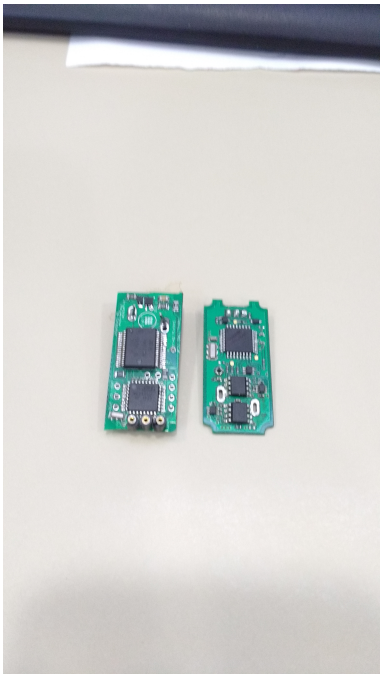
Fig. 2.6: Schematic of the main board

(a) The DC servo motor



(b) The original board

(c) The board removed



(d) Old and new board

Fig. 2.7: The original and modified servo motor

It also contains a mosfet that acts as protection from reverse connection of the power supply or battery. Fig. 2.8 shows the *top* side of the designed board.

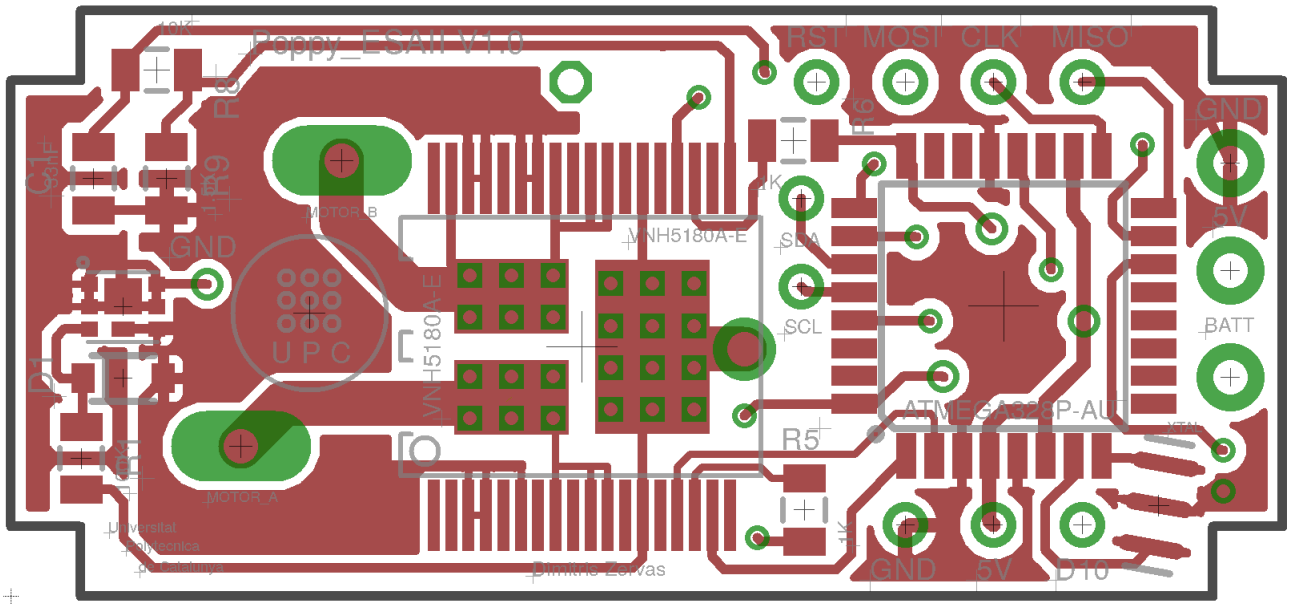


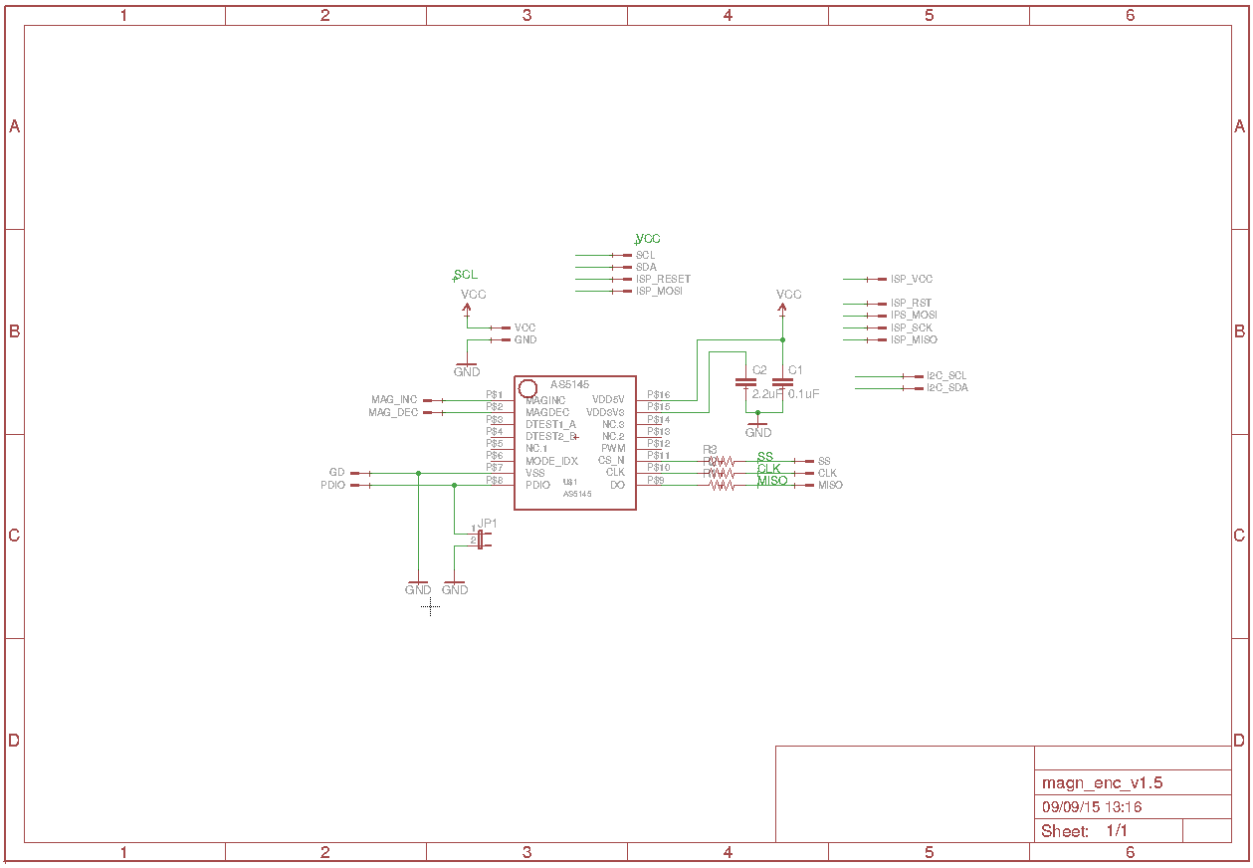
Fig. 2.8: Top side of the main board

The board, has on its left the three power supply pins, namely *VCC*, *5V* and *GND*. The board does not contain a voltage regulator to supply the microcontroller and the sensor. 5 volts must be supplied externally. Around the place of the microcontroller (on the right of the top side of the board), there are the pins that the *sensor board* will be attached.

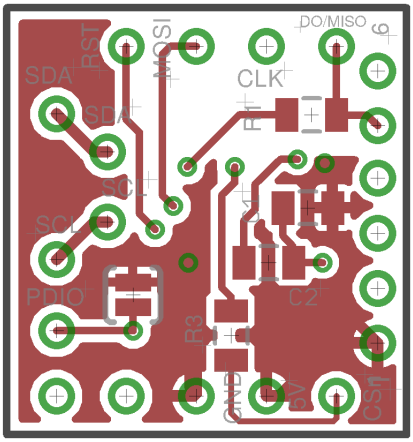
2.4.2 Sensor Board

The sensor was designed on a second board for two reasons. First, because of lack of space on the main board and second, in order for the sensor to be placed closer to the magnet. Fig. **??** shows the schematic and he top side of the *sensor board*. On the right of the *top side* there are the pins for the ICSP cables. Among them, there is also a 5V pin. When the user wants to upload a *sketch* to the Arduino, he must **take care** to not supply both the boards (main and sensor) with 5V as the 5V pin of the *main board* is the same 5V pin on the *sensor board*.

On the left side of the *sensor board* there are the two pins for the I2C communication. **Note!** There are no pull-up resistors for the I2C pins on either of the two boards.



(a) Sensor board schematic



(b) Top side of the sensor board

Fig. 2.9: The sensor board

As noted in Section 2.1

"A properly aligned magnet will therefore produce a $MagINC = MagDEC = 1$ signal throughout a full 360 deg turn of the magnet."

The *sensor board* provides also these two pins, namely *MagINC* and *MagDEC* in order for the user to check if the magnet is properly aligned.

2.4.3 Summary

On this chapter, was presented a short description of each of the "tools" that are needed to successfully control a motor. In addition to that, the necessary code to use each one of them was provided. Finally, the schematic and board files were shown. The final outcome results in a board with 11 cables coming out of it. These are

- 3 for the power supply (VCC, 5V, GND)
- 2 for the I2C (SDA, SCL)
- 6 for the ICSP (5V, GND, MISO, MOSI, SCK, RST)

It is worth reminding here that the *5V* and *GND* pins from power supply and ICSP are interconnected with each other through the board. Also, if the user does not want to upload any new program and just want to interact with the motor through I2C, the 6 cables of the ICSP are not needed any more. Finally, he can connect the I2C cables either on the main board or on the sensor board, at his convenience.

Chapter 3

GUI

3.1 Introducing the GUI

The proper tune of a *PID* controller is most of the times a tedious process. Usually the designer, first creates the mathematical model, then designs the simulation where he is tuning the gains of the controller and finally he applies the controller to the real system. In most of the cases the result of the simulation don't coincide with the result of the real system. Some of the reasons for that could be, the uncertainty of the model or the difficulty to model some physical phenomena, such as friction, backlashes etc.

The solution for that, is that the designer must re-tune the gains empirically based on experimental results or using some other techniques. It is a process that can be time consuming, as the choice of the gains is based purely on the designer's intuition.

In order to help with this process and make it easier, an application was created, where the user can see the output data of the motor *live*, compared to the reference signal and tune the PID gains *"online"*. It also provides the option to create its own reference signals. Figure 3.1 shows the main window.

The application was implemented using *Python*. More specific *PyQt4* was used. *PyQt4* is a set of *Python* bindings for *Qt*. *Qt* is a cross-platform application development framework for desktop, embedded and mobile. Someone can find more informations at [**?**] and [**?**]. The application is communicating through serial communication with an Arduino, on which the driver of the motor is connected, as well as the sensor. For this implementation was used the driver that was chosen and discussed in Section 2.2 and the magnetic encoder as discussed in Section 2.1. Through this section, only some of the *Python* code is presented because of its size. At the end of this section the full Arduino code is presented.

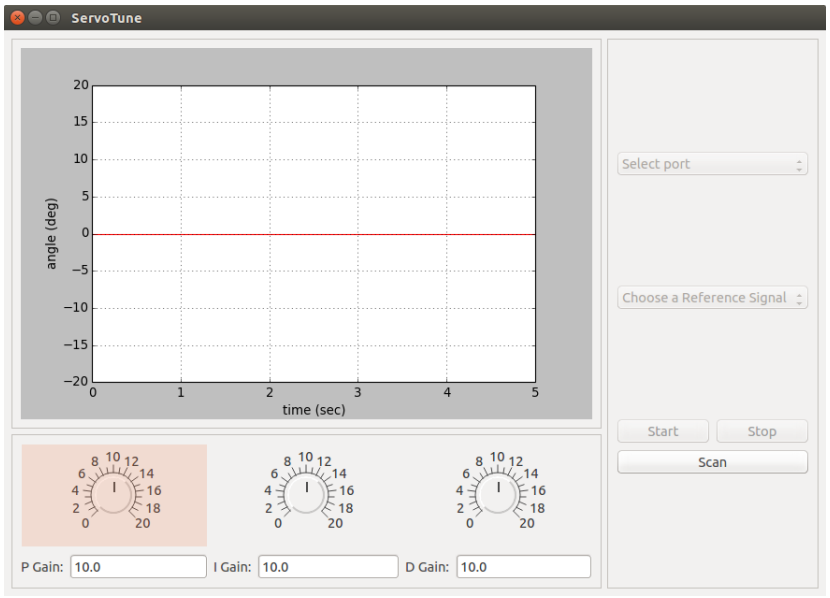


Fig. 3.1: Main Window

In Fig. 3.1 is shown the main window of the application. It consists of 3 parts,

- The figure part, where the output data is compared with the reference signal
- The knobs part, to tune the P/I/D gains
- the configuration part, where you can choose the reference signal you want to apply, the serial port to connect to, and to start/stop the process.

3.1.1 Knobs section

The three knobs are to modify the P/I/D gains accordingly. There are three characteristics for each knob. The ***range***, the ***mid*** value and the ***step***. The range is expressed as $\pm val$ the *mid* value. The *step* refers to the minimum change on the knob value. For example, the original configuration of the knobs are,

Table 3.1: Knobs original configuration

mid	10
range	±10
step	0.1

So with $mid = 10$ the range is $[0 - 20]$ and every movement of the knob will change the value by ± 0.1.

Under of each knob, there is a *"Line Edit"* with original value of 10. This corresponds to the mid value of the knob. For example, with the original configuration described in Table 3.1, if the user enters the value 20 in the *Line Edit*, the press of *enter* will result to a range of $[10 - 30]$ with $step = 0.1$ and of course $mid = 20$. The result is shown in Fig. 3.2b,

(a) knob(P) original configuration **(b)** knob(P) new configuration

Fig. 3.2: knob(P) configuration through *mid* value

The user also has the choice to change the other two parameters (*range* and *step*). Under the *"Tools"* there are three more tabs, *knob(P)*, *knob(I)* and *knob(D)*. By clicking any of them, a new *Dialog* window opens as it is shown in Fig. 3.3. This *input dialog* window is modeless, which means that does not block the main window. Therefore, the user can also reconfigure the knobs while the process is running and not only in advance.

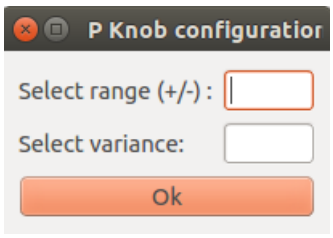


Fig. 3.3: knob(P) configuration window

3.1.2 Creating Reference Signals

Under the *Signal* option in the tool bar, the are two more tabs, ***New Reference Signal*** and ***New Periodic Reference Signal***. The first choice is to create a *custom* signal and the later allows the user to create either a *square* or a *sawtooth* signal.

New Reference signal

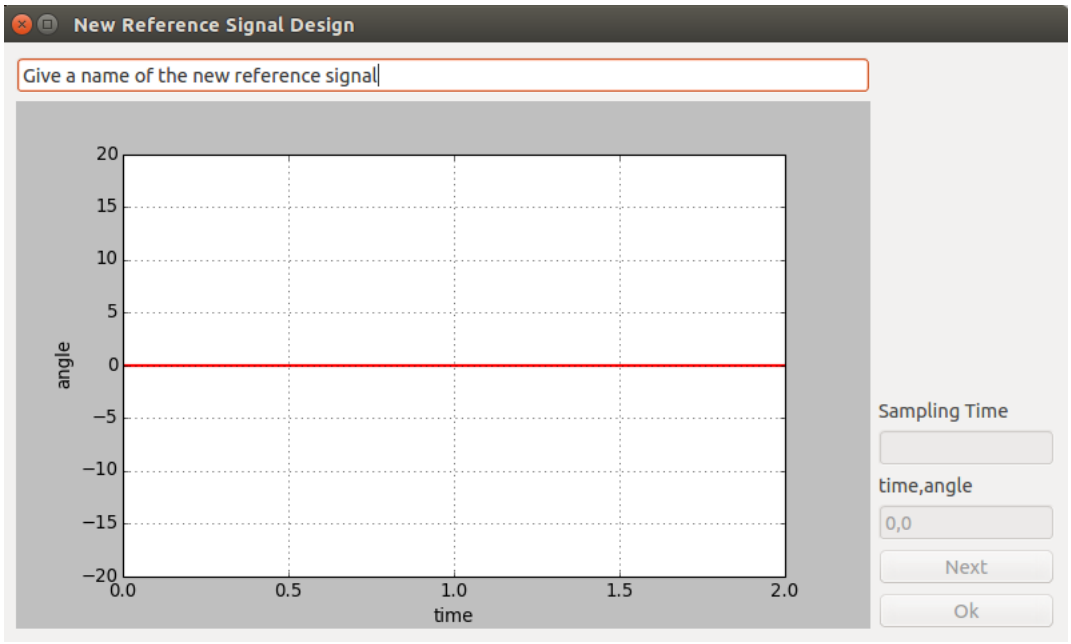


Fig. 3.4: New Reference Signal Main Window

By selecting *New Reference Signal* the window of Fig. 3.4 pops up. This windows contains,

- A *Line Edit* to enter the name of the signal
- The *Figure* that shows the signal as the user creates it
- A *Line Edit* to enter the Sampling Time

- A *Line Edit* to enter a point in the form $time\_val, angle\_val$. *Time* is the x-axis coordinate of the point and *angle* the y-axis coordinate of the point
- A button that allows the user to enter the *next* point
- A button (OK) to finish the process of crating a signal and to return to the main window.

There are some points that need to be noticed. The first one has to do with the name of the signal. In the main window, there is a *combo* box that -will- contain all the signals that the user created (see Fig. 3.1). In the *New reference Signal* window of Fig, 3.4, by clicking the *OK* button the name of the signal will populate the list of the *combo* box in the main window. Therefore, if the user enters a name that already exist, he will not be able to continue, unless he will change the name. The second point has to do with the form of the point(time,angle). Table 3.2, shows a sequence of points and Fig. 3.5 the resulting signal.

Table 3.2: Example of a sequence of points for custom refrence signal

0,15	1,15	1,-20	1.5,-20	1.5,10	3,10	3,0	4,0

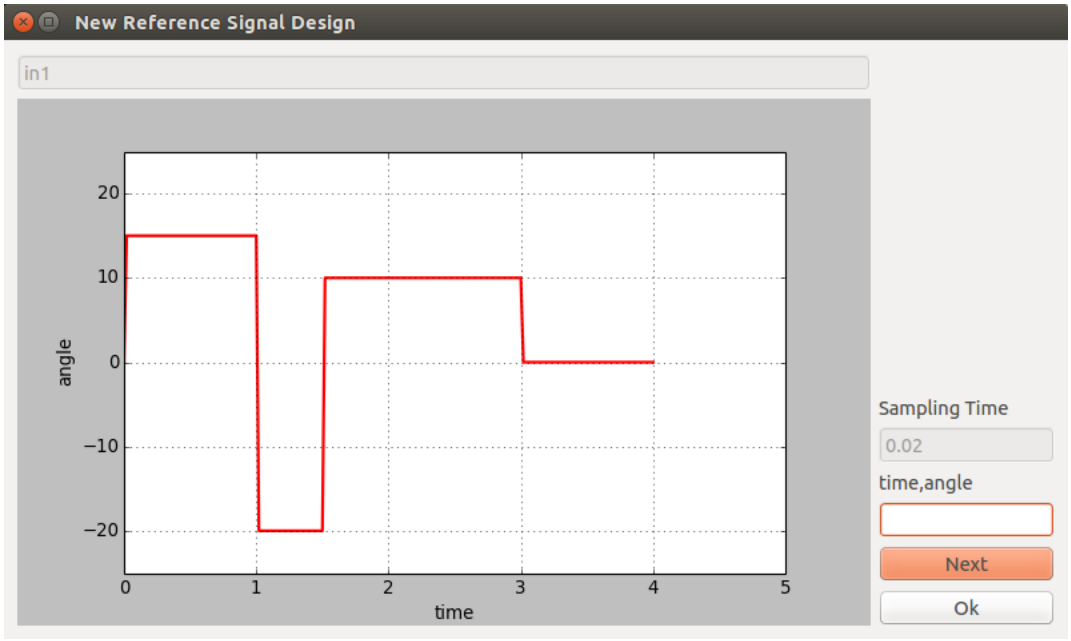


Fig. 3.5: The signal results from the point of Table 3.2

Note!: All the reference signals start from (0,0).

The final point that needs to be noticed, is the *Sampling Time*. By clicking the button OK, except of adding the signal to the *combo* box in the main window, it also samples the line. In the example of Fig. 3.5 the *Sampling Time* is 0.02 and the total duration of the signal is 4 seconds. That means that after the sampling, the signal is a *list* of $4/0.02 = 200$ points. The reason for that is explained in Section 3.2. The value of the *Sampling Time* **must be** equal to the *control loop* as this was defined in Section 2.3.2.

New Periodic Reference Signal

Apart of custom reference signals, the user also has the option to create *square* or *sawtooth* signals. By selecting **Signal→New Periodic Reference Signal** from the tool bar, a new *Dialog* window will appear, as show in Fig. 3.6. For either the *square* or the *sawtooth* signal the parameters for the user to configure are,

- The *Name* of the signal
- The total *Time* of the signal
- The *Sampling Time*
- The *Amplitude* of the signal
- The *Frequency* of the signal

The *Name* of the signal is working with the same way as for the custom reference signal described in Section 3.1.2. If the name already exists, the user cannot proceed unless he change the name. The total *Time* and the *Sampling Time* also work as described in Section 3.1.2. And finally, the *Amplitude* and the *Frequency*, are the actual parameters of the signal. There is also a *combo* box that allows the user to choose the type of the signal.

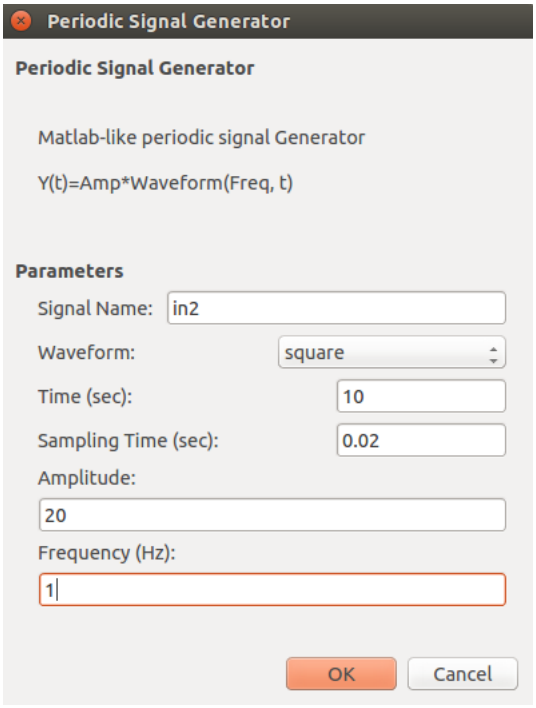


Fig. 3.6: Main window for creating a Periodic Reference Signal

Once all the fields are proper complete, the OK button will become active and the user will return to the main window where, if he choose in the *combo* box the signal just created, he will see the result plotted in the main figure. Fig. 3.7, shows the result of the signal created in Fig. 3.6.

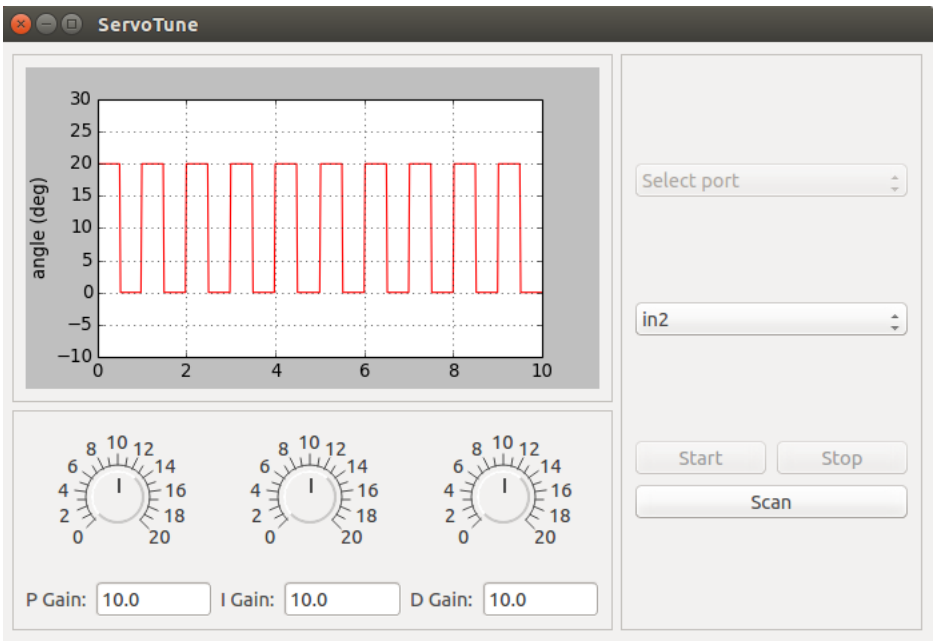


Fig. 3.7: The signal created on Fig. 3.6 plotted in the main figure.

3.1.3 Configuration section

On this part of the main window the user first has to *Scan* for any available port and there is a button for it. The list of all the available ports will appear in the specific *combo* box.

After the creation of at least one reference signal, it must be selected through the according *combo* box. Once selected, the signal will be plotted in the main figure.

After the selection of the port and the reference signal, the *Start* button will become active and the process is ready to begin.

3.2 Communication

The communication between the application (Python) and the controller (Arduino) was implemented through the serial interface. To achieve this, the **pySerial** library was used [**?**].

While it is possible to create specific timed loops in *python*, under any operating system this loop can not be guaranteed to be accurate every time. Something that is easy to achieve in a micro-controller such as Arduino, in a way that was described in Section 2.3.2. For that reason, every communication between Arduino and Python is triggered by Arduino.

By pressing the *Start* button on the main window of the application, Arduino is restarted through the *pySerial* library to ensure that both sides are synchronized. After that, Arduino, for every control loop, sends once the character 's' that stands for start. If Python will receive this character, both sides are ready for the data exchange. Since the communication is 1-on-1 it is

possible for both sides to know exactly what to expect from the other side. That makes simple the process of validating that the data were transferred correctly.

3.2.1 Python side

Since Python-application received the 's' character from Arduino, is ready to send data to it. That data is a fixed serial *word* consists of 10 bytes. Table 3.3, shows an example of this serial word.

Table 3.3: Serial word to be sent to Arduino

`start_cmd` (1 byte)	`reference-float` (4 bytes)	`gain_cmd` (1 byte)	`gain-float` (4 bytes)

The start command can be one of the following bytes,

Table 3.4: Start commands

start_cmd	**Byte**	**Description**
`startCx_f`	`0xf0`	Full word: reference + gain
`startCx_e`	`0xf1`	"Empty" word: reference + zeros
`stopCx`	`0xf2`	Stop word: zeros + zeros

where the gain command can be one of the following bytes,

Table 3.5: Gain commands

Command	**Byte**	**Description**
`P_Gain`	`0xf3`	Update the P gain
`I_Gain`	`0xf4`	Update the I gain
`D_Gain`	`0xf5`	Update the D gain
`No_Gain`	`0xf6`	No gain update

Every time, the data exchange from both sides is happening inside the "time window" of the control loop. Since this loop is at the level of milliseconds, it is impossible for the user to change the value of two knobs on the same time. Therefore, there is no need for the *serial word* to contain the float values of all three gains. If that was the case, then every *Serial word* sent from Python to Arduino would consist of 17 bytes. Instead, with the use of the *gain command* we indicate to Arduino which gain to update every time.

Also, it is obvious that most of the *serial words* that are sent from the application wouldn't contain any change in any gain value, as the user is not able to change values that fast. For that reason, the *"startCx_e"* command was introduced, to indicate to Arduino, that on this word there is only the new reference value, a useful information for the word decomposition from the Arduino side, as it is shown in Section 3.2.2. The *"startCx_f"* is for the case where there is a change on one gain and the *serial word* is "full". And finally, *"stopCx"* indicates to Arduino, that this is a *serial word* with all zeros, which means that the whole reference signal was sent. Listing 3.1, shows the Python code for constructing the *Serial word*.

It is clear now why the reference signal is sampled and why in Section 3.1.2, was pointed out that the *Sampling Time* of the reference signal and the *control loop* **must be** equal. If the reference signal had different sampling time, then the timing of both the signals (reference and output) wouldn't coincide in the plot at the main figure and of course, the output wouldn't be representative of what the user would want.

The process of communication between these two sides contains mainly, data sent, data received and refreshing the main figure. All these processes are time consuming and must be complete before the new time "window" of the next control loop. If the implementation of all these code is done in a "serial" manner, then there would be time "windows" where there would be no time to check any change in the knobs. This would translate to the user as some kind of "lag" (delay) in the movement of the knobs. To solve this issue, the communication part of the code, was originally implemented using the threading interface, [**?**]. It was observed though, that the communication was not following the time "window" (there were times that needed two or even three control loops in order to send the new reference value). The reason of that was, as it is stated in [**?**],

"...due to the *Global Interpreter Lock*, only one thread can execute Python code at once...If you want your application to make better use of the computational resources of multi-core machines, you are advised to use **multiprocessing**."

Hence by using the "multiprocessing" interface [**?**] one processor is used for the communication process allowing to simultaneously exchange data with Arduino, refresh the main figure and change the value of any knob.

The most "*multiprocessing*"-safe way to transfer data between the *process* and the main application, is by use of Queues [**?**]. For that a reason a FIFO Queue was created named `gain_q` . Every time a knob is moved, first the name ('P', 'I' or 'D') of the knob and then the value of the knob are added to the `gain_q`. Therefore if the size of the Queue is equal to 2, that means there is a change to a knob. An information used for the construction of the *Serial word*.

Listing 3.1 Construction of Serial word

```
# the reference value to send
self.ref = self.reference[self.ref_counter]
# value1 is the first float to be send (the reference)
value1 = struct.pack('%sf' % 1, self.ref)

# Update the gains
if self.gain_q.qsize() == 2:
    let = self.gain_q.get()
    self.new_gain = self.gain_q.get()

    if let == 'P':
        self.command_gain = command.p
    elif let == 'I':
        self.command_gain = command.i
    elif let == 'D':
        self.command_gain = command.d
```

```
elif self.gain_q.qsize == 1:    # if size less than 2 (smth went wrong!), ...
    clear the queue
    trash = self.gain_q.get()

# If the knobs didn't change, send just the reference (startCx_e)
if self.new_gain == self.prev_gain:
    command_start = command.startCx_e
    self.command_gain = command.x
    # value2 is the second float to be send (the gain), in that case, 0.
    value2 = struct.pack('%sf' % 1, float(0))
else:
    self.prev_gain = self.new_gain
    command_start = command.startCx_f
    value2 = struct.pack('%sf' % 1, self.new_gain)

# Construct the buffer
buf = bytearray([command_start, ord(value1[0]), ord(value1[1]), ...
    ord(value1[2]), ord(value1[3]),
                self.command_gain, ord(value2[0]), ord(value2[1]), ...
    ord(value2[2]), ord(value2[3])])
```

3.2.2 Arduino side

As it was already mentioned, it is very important the data exchange between *Python* and *Arduino* to be complete in every "time window" defined by Arduino as *control loop*. The *ATmega328* micro-controller (Arduino), has a **U**niveral **S**ynchronous and **A**synchronous serial **R**eceiver and **T**ransmitter (USART). The user can find more details in the datasheet [**?**], but what is need to be clear, is that it receives 8-bit of data in every "step".

In order to achieve the communication as fast as possible, the use of interrupts was necessary. Arduino has already implemented a *Serial* library that can be used, but it is already using the desired interrupts. Therefore, the programming of the USART had to be done manually. In order to explain how the code works, first is necessary to explain the set-up of the variables.

First the incoming data buffer is defined, ***bufferRx***

Listing 3.2 Buffer to store the incoming data

```
volatile unsigned char counterRx = 0;
volatile unsigned char bufferRx[9] = {0,0,0,0,0,0,0,0,0};
```

As it is already mentioned earlier, *Python side* is sending a word of length of 10 bytes every time. The strategy is to receive the full word, and then decompose it accordingly. There is also a counter ***counterRx*** to allow indexing this buffer. The reason these variable are defined as *volatile* is in order to be able to used inside an interrupt routine.

The definition of the floats that are received and sent are shown in Listing 3.3. Here is worth noticing, that for the Arduino environment, *Floating-point* numbers are stored as 32 bits (4 bytes).

Listing 3.3 Floats and their pointers to be sent/received

```
// Number to be sent
float pos = 0;
unsigned char *pointer_pos = (unsigned char *)&pos;
// Numbers to be received
float ref = 0;
float *pointer_ref = (float *)&bufferRx[1];
float gain = 0;
float *pointer_gain = (float *)&bufferRx[6];
// Actual P/I/D gains
float P = 0;
float I = 0;
float D = 0;
```

The *pos* float is used to store the position value that was read from the sensor and will be sent to the *Python side*. As it is mentioned, the structure of the incoming *serial word* is well known (Table 3.3). Therefore it is certain that in $bufferRx[1]$ will be the first byte of the *reference-float*. By defining a float as *ref*, we conserve 4 bytes in the memory. With the use of pointer, we can connect that space of the memory with the specific part of the buffer. With that way, the value of the float *ref*, is defined by the bytes $bufferRx[1:4]$. Similarly the value of the float *gain* is defined by the bytes $bufferRx[6:9]$.

Manual Serial

To start a *Serial* communication, first the USART module needs to be initiated. At the very top of the Arduino code, some definition are made that specify the *BAUDRATE* of the port.

Listing 3.4 BAUDRATE definitions

```
// USART initialization def's
#define FOSC 16000000UL   //clock speed
#define BAUD 115200     //desired baud rate
#define MYUBRR (FOSC/4/BAUD-1)/2
```

Now it is possible to initialize the port,

Listing 3.5 USART_Init

```
void USART_Init (unsigned int ubrr)
{
  UCSR0A = 0;
  UCSR0A |= (1<<U2X0);
  /* Set baud rate */
  UBRR0H = (unsigned char)(ubrr>>8);
  UBRR0L = (unsigned char)(ubrr);
  UCSR0B = B00000000;
```

```
  // Enable receiver and transmitter
  UCSR0B |= (1 << RXEN0) | (1 << TXEN0);
  UCSR0C = B00000000;
  // Set frame: 8data, 1 stp
  UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);

}
```

And a function that can be used to send data,

Listing 3.6 Function to transmit data

```
void USART_Tx (char data)
{
  /* Wait for empty transmit buffer */
  while ( !(UCSR0A & (1<<UDRE0)) ) {
  }
  /* Put data into buffer, sends the data */
  UDR0 = data;
}
```

Every time there are 8 bits that are received successfully, an interrupt is triggered. These byte, is stored to the register **UDR0**. To take advantage of this interrupt, the content of *UDR0* is stored directly to the *bufferRx*, inside the interrupt routine. All it has to be taken care for, is to increase the *counterRx* every time and set it to zero again when needed.

Listing 3.7 Incoming data interrupt routine

```
ISR(USART_RX_vect) {
  bufferRx[counterRx] = UDR0;
  counterRx++;
}
```

Control Loop

There are three boolean flags, namely *com*, *txOn* and *rxOn*. The *com* flag is to indicate if the communication inside the time window of one control loop is complete. The rest two flags, are to indicate which part of the code to execute according to either if it has to receive data ($rxOn==true$) or it has to transmit data ($txOn==true$). The pieces of the code for each of the cases are following,

Listing 3.8 Read and Send data

```
if (txOn==true) {
  /* Controller calculation */
```

```
    // Here you apply your controller, for example,
    pos = ref + P;
    /* --- end of Controller --- */
    for (i=0; i<4; i++) {
        USART_Tx(pointer_pos[i]);
    }
    for (i=0; i<4; i++) {
        USART_Tx(eof[i]);
    }
    txOn = false;
    sendOnce = true;
    com = false;  //finished the com, now wait for the rest of Ts
}

if (rxOn == true) {
  switch (bufferRx[0]) {
        case 0xf0:  //read reference + gain
            rxOn = false;
            ref = (*pointer_ref);
            // Gains update (one at a time)
            if (bufferRx[5] == 0xf3) {
                P = (*pointer_gain);
            }else if (bufferRx[5] == 0xf4) {
                I = (*pointer_gain);
            }else if (bufferRx[5] == 0xf5) {
                D = (*pointer_gain);
            }
            txOn = true;
            break;
        case 0xf1:  //read only reference
            rxOn = false;
            txOn = true;
            ref = (*pointer_ref);
            break;
        case 0xf2:  //stop
            rxOn = false;
            txOn = false;
            break;
  }
}
```