Design and Implementation of Controller for Lego Mindstorm Motor

Mehetab Alam Khan, Simone Dal Poz

February 15, 2017

1 Introduction

This report is the presentation of the work done in the course *Advanced Robotics Lab*. The aim of the work was to design a controller for the motor of *Lego MindStrom*. The whole task consisted of the following activities:

1. Collection of the motor output data set (angular position) as a response to a step input (Percentage of maximum power to the motor)
2. Design of a filter for the above dataset
3. Motor parameter identification for creating a suitable mathematical model
4. Design of a feedback controller to control the power output of the motor
5. Simulation of the controller in *Simulink*
6. Implementation of the above controller in the *Lego Mindstorm* brick

2 Data Collection and Partition

The step response data set has been collected through the *Brofist* Bluetooth client running on the *Lego MindStorm* brick. All readings were taken with a sampling frequency of 250 Hz and each reading has a duration of 30 seconds. The input to the motor is the percentage of the maximum power that can be generated by the *Lego Brick*. The whole span (0% to 100%) has been divided into 20 steps so as to provide inputs with increments of 5% for consecutive readings.Collected data has been arbitrarily partitioned into two sets. The training set [Amplitude Interval: 10% - 100%, Increment: 10%] has been used for the identification of the model parameters, while the test set [Amplitude Interval: 5% - 95%, Increment: 10%] has been used for validating the model design.

3 Butterworth Filter Design

3.1 Bandwidth and Cutoff Frequency Identification

The spectrum of the signal is obtained by performing a *Discrete Fourier Transform*. As expected, the harmonic content of the signal is concentrated in the interval [0 - 10 Hz], and the noise is distributed around three peaks: 102 Hz, 110 Hz and 118 Hz respectively (Figure: 1).
The chosen filter type is *Butterworth*, due to the flatness of the response below the cutoff frequency. In this case, a *Butterworth* filter has a overall better performance than a *Chebyshev* one, although it has a worse attenuation of the harmonic content above the cutoff frequency.
The cutoff frequency has been determined through a simulation. The filtered signal has been computed for each integer frequency in the interval [1 - 10]. The frequencies below 6 are not shown for clarity.
As shown in figure 2, low cutting frequencies lead to a visible, delayed overshoot and they also cut some of the useful signal components. Whereas, the higher frequencies make the impulse response flatter and faster,but with ringing (due to the presence of noise components). A 5 Hz cut-off frequency is the best compromise between adding too much overshoot due to filtering and allowing too much noise.

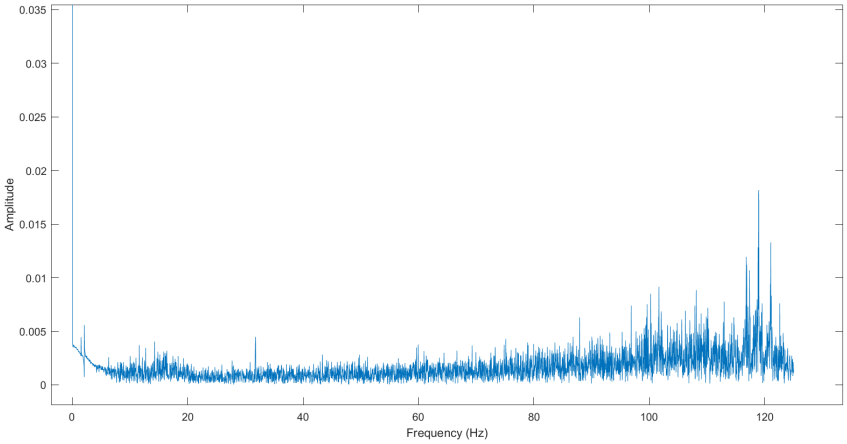

Figure 1: Signal Spectrum of Step Response (Amplitude 100%)

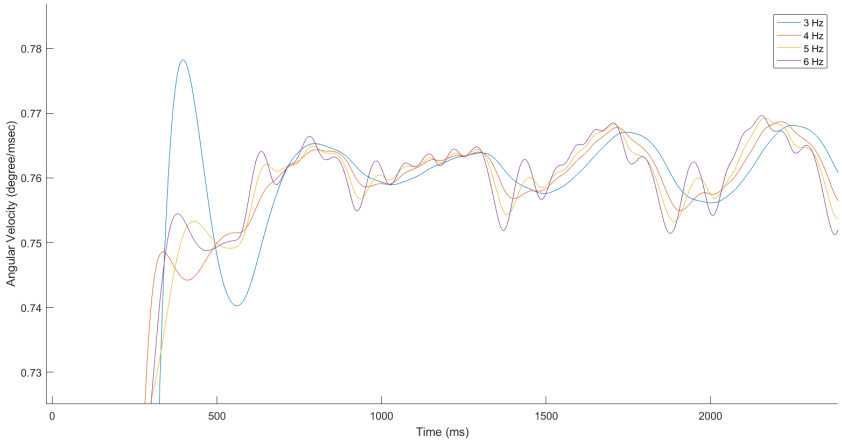

Figure 2: Buterworth Filter Output for Different Cut-off Frequencies (Filter Order 4, Amplitude 100%)

3.2 Filter Order Identification

A simulation is conducted to evaluate the effects of the filter order on the output signal. The range of the examined possibilities varies from order 1 to order 10. The simulation was set to use the phase-shifting digital filtering. As can be seen from the figure 3, higher the filter order higher is the added overshoot and delay. So a 4th order filter has been chosen as a compromise between quality of the filtering, added overshoot and delay. Some of the results are not shown in the plot for clarity.

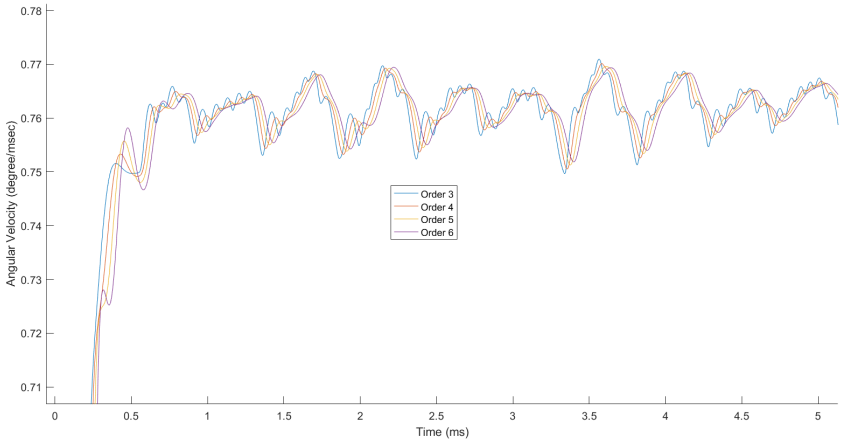

Figure 3: Filter order simulation - Cut-off frequency 3 Hz

4 Model

4.1 Parameters Identification for Training Data set

For each of the measures the following parameters have been calculated:

Table 1:

	Rise Time (ms)	**Sett. Time (ms)**	**Sett. Min**	**Sett. Max**	**Overshoot %**
10%	147.97	436.79	0.067	0.080	7.52
20%	153.58	422.56	0.148	0.174	5.93
30%	151.67	421.40	0.223	0.262	5.93
40%	170.40	665.34	0.305	0.339	0.37
50%	155.75	423.33	0.381	0.443	5.22
60%	158.30	423.73	0.460	0.534	4.78
70%	158.00	416.64	0.536	0.619	4.57
80%	160.96	410.15	0.620	0.712	4.09
90%	159.97	411.11	0.694	0.799	4.05
100%	161.60	414.59	0.780	0.895	4.02

4.2 Model Verification

For the verification phase the following procedure has been applied:
For each voltage step:

1. The settling value has been computed through a zero-order polynomial fitting of the signal after the settling time. The considered threshold for calculating the settling time has been set to 5% of the maximum signal amplitude.

2. A normalization parameter q has been computed by dividing the n-th signal steady state value by the n-th input power in the range [-1,+1].

$$q = q_n/p_n$$

3. The damping factor ζ has been calculated.

$$\zeta = \sqrt{\frac{(ln\, O)^2}{\pi^2 + (ln\, O)^2}}$$

where O is the overshoot value.

4. The natural frequency ω_n has been calculated.

$$\omega_n = \frac{ln(0.03) - ln\left(\frac{1}{\sqrt{1-\zeta^2}}\right)}{-\zeta * ST}$$

where ST is the settling time.

After computing steady state value, damping ratio and natural frequency for each power level of the training set, the final model parameters have been computed with a trimmed mean, excluding 20% of the outliers. Since the data set represents different probability distributions, considering a trimmed mean does not affect the calculation of the output. The resulting parameters are:

$$q = 0.7442 \qquad \zeta = 0.7484 \qquad \omega_n = 19.0673$$

Finally, a transfer function has been computed to simulate the behavior of the system when stimulated with step functions of different amplitude.

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

4.3 Model Validation

After comparing the simulated output results against the training data set the test data set has been used to validate the model. The results are shown in figure 4.

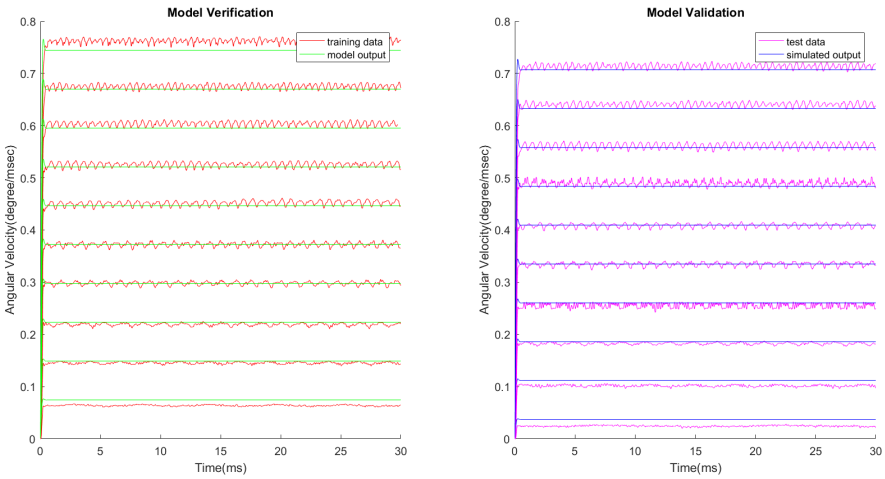

Figure 4: Simulation with Training and Test data

The Root Mean Square Error has been computed for every power level in the test set. The results are reported in table 2.

$$\sqrt{\frac{\sum_{n=0}^{N}(e_e - e_m)}{N}}$$

It can be easily observed that the model has a good accuracy: experimental and simulation results are extremely close, and the RMSE never exceeds 3%. The RMSE variation suggests that the amplitude and the applied power are not linearly dependent, but such approximation does not introduce a significant mismatch.

Table 2: RMSE values for different power levels

	RMSE %
10%	1.06
20%	0.65
30%	0.99
40%	1.01
50%	1.29
60%	1.26
70%	1.62
80%	2.81
90%	2.15
100%	2.38

5 Controller Design

5.1 Root Locus

From previous calculations the final form of the model transfer function, or plant, has been obtained:

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} = \frac{0.7442}{0.002751s^2 + 0.0785s + 1}$$

The next goal to achieve is designing a feedback system (Figure 5) such that the closed loop transfer function

$$\frac{C(s)G(s)}{1 + C(s)G(s)}$$

satisfies certain design requirements, namely:

1. Overshoot percentage less than 10.
2. Settling time less than 500 ms.
3. Marginal Stability.
4. Steady state error for step input response equal to 0.

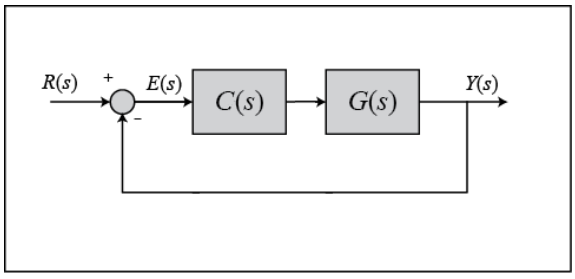

Figure 5: Feedback System

The chosen design method is the Root Locus Analysis, which allows the designer to translate the given constraints into mathematical/geometrical conditions, respectively:

1. $\zeta < \sqrt{\frac{(ln\,0.1)^2}{\pi^2 + (ln\,0.1)^2}} \implies \zeta < 0.591155$
2. $\omega_n < \frac{ln(0.03) - ln\left(\frac{1}{\sqrt{1-0.591155^2}}\right)}{-0.591155 * 0.5} \implies \omega_n < 12.5907$
3. The real part of every pole must be less or equal than 0 in order for the system not to diverge.
4. There must be a pure integrator (one pole in the origin)

The input of the Root Locus Analysis is the open loop transfer function $C'(s)G'(s)$ with $C(s) = k_c C'(s)$ and $G(s) = k_g G'(s)$. The zeros and poles of the controller are chosen in order to satisfy the constraints of the controlled plant system. The resulting controller is in the following form.

$$C'(s) = \frac{(s + 35.99)}{s}$$

The low value of the controller gain ($k = 0.2176$) leads the response towards over-damping and hence a flatter curve. Figure 6 shows the root locus for the open loop system (controller and plant).

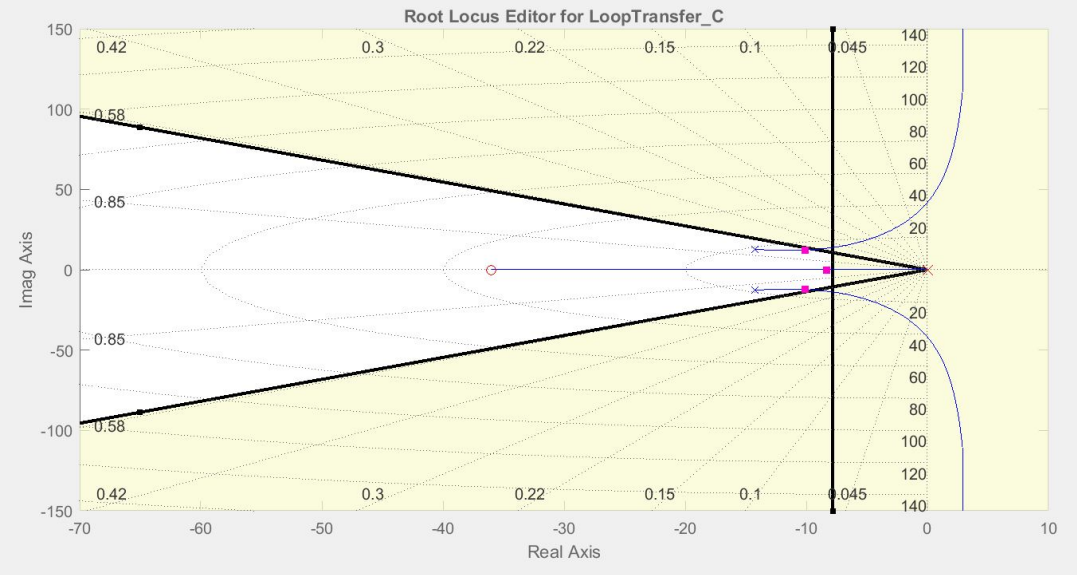

Figure 6: Root Locus of the open loop system C'(s)G'(s)

5.2 Controller simulation

A Simulink simulation of the closed loop system has been conducted (Figure 7). The ideal continuous system has been compared against a more physically accurate model. While the ideal system is composed only by the continuous transfer functions of the controller and the plant, plus a saturation block which limits the voltage input of the system to 100%, the real robot has to deal with the signal discretization and quantization, the presence of a position sensor and the signal filtering process.

More in detail the input of the system is a target speed in the form of a step function. The implementation of the controller requires the controller itself to be discrete for reasons which will be better explained in the next section. A sample and hold block is added for the transition between the sampled output signal of the controller and the continuous nature of the physical system. Again, a saturation block is added to limit the power requests calculated by the controller. The output of the system (the current velocity) is then integrated, quantized and discretized to simulate the readings of a position sensor with a sampling frequency of 500 Hz and a minimum step of 1 deg. An exponential weighting average with a forgetting factor of 0.9 is used to filter out the unwanted components of the signal. The exponential average is used instead of a Butterworth filter because it does not introduce delay and it is not computationally demanding. Finally, the error between current speed and target speed is calculated by the subtraction block in order for the controller to compute the next power value to feed the system with.

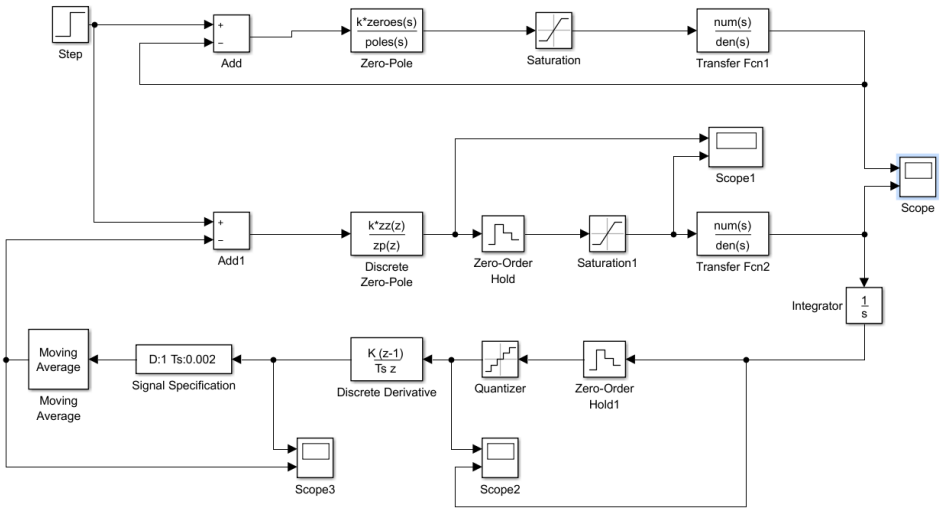

Figure 7: Feedback System

5.3 Simulation Results

Due to the processes described above, the second system is not ideal. For this reason the constraint on overshoot and settling time may or may not be respected depending on the target speed input. For lower target speeds (Figure 8) the parameters sometimes exceed the maximum overshoot and settling time (Figure 9), while high target speeds on the contrary result in faster and more robust responses, almost matching the ideal model response (Figure 10). Ideal model outputs are shown in yellow, realistic model ones in blue.

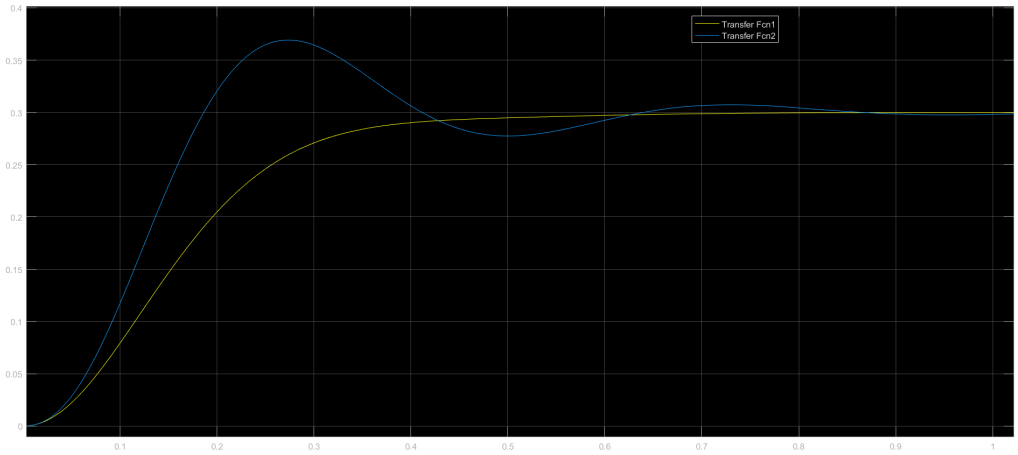

Figure 8: Simulated Output (Target Speed: 0.3 deg/msec, Overshoot: 22.0%, Settling Time: 449 ms)

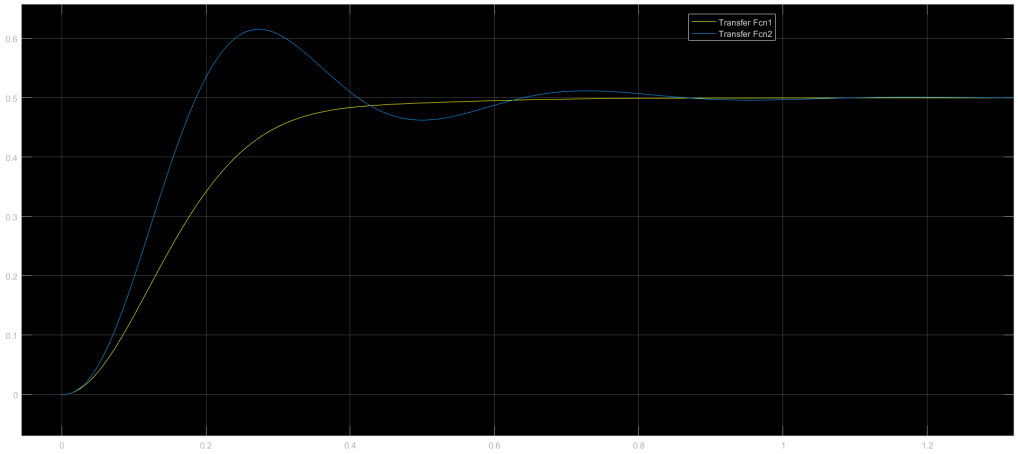

Figure 9: Simulated Output (Target Speed: 0.5 deg/msec, Overshoot: 23%, Settling Time: 800 ms)

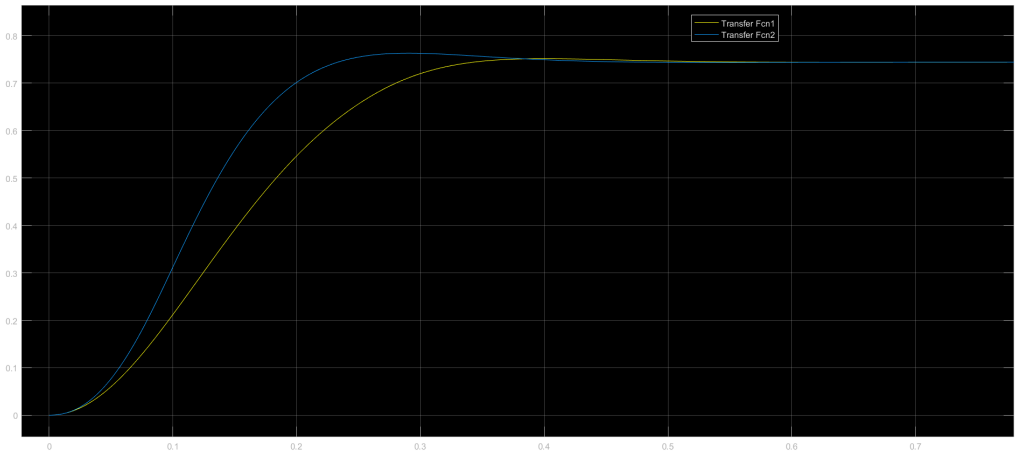

Figure 10: Simulated Output (Target Speed: 0.8 deg/msec, Overshoot: 9%, Settling Time: 400ms)

Note: As shown in figure 11 a significant part of the controller power output is cut out by the saturation block, so having an excessively high controller gain would be unnecessary. Moreover, despite the lower gain, the real model tends to be more reactive than the ideal one.

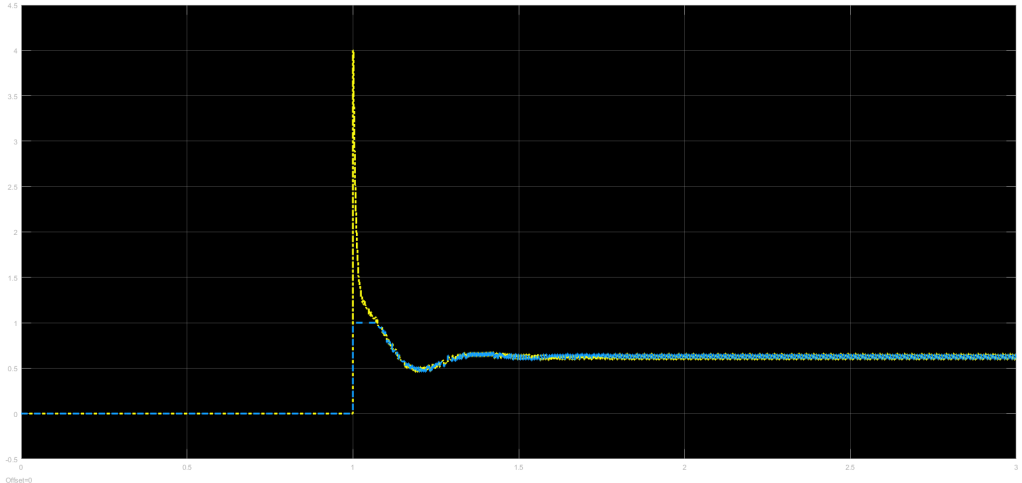

Figure 11: Controller output before (yellow) and after (blue) the saturation block

5.4 Controller Discretization

A fundamental step before implementing the solution on the Lego Brick is the discretization of the controller. This process is needed for the Brick to be able to recursively calculate the current power output level given the history of power output values and angular velocity input values at every sampling interval.
First of all the continuous model has been approximated to a discrete one using the Tustin's trapezoidal rule:

$$z = e^{sT} \approx \frac{1 + \frac{sT}{2}}{1 - \frac{sT}{2}} \implies s \approx \frac{2(z-1)}{T(z+1)}$$

$$C(s) = 0.21763 * \frac{s + 35.99}{s} \implies C_z(z) = 0.2333 * \frac{z - 0.8657}{z - 1}$$

It is possible to write the controller equation in the following form:

$$C_z(s) = \frac{Y(z)}{U(z)} = 2.333 * \frac{1 - 0.8657z^{-1}}{1 - z^{-1}}$$

$$Y(z) = 0.2333 * (U(z) - 0.8657z^{-1}U(z)) + z^{-1}Y(z)$$

Finally, by applying the delay property of the zeta-transform

$$z^{-h}U(z) = Z[u(t-h)]$$

and by doing the inverse transform, an equation is obtained:

$$y(t) = 0.2333 * (u(t) - 0.8657u(t-1)) + y(t-1)$$

This equation can be recursively solved by an algorithm running on a computing system.

6 Implementation

6.1 One Wheel Control

The digital controller has finally been implemented using a single task, due to the limitation of the hardware platform. The used variables have been declared static to ensure that the value is maintained among invocations. After the initialization the average speed is calculated through an exponential average. Then the current error is calculated by subtracting the average speed from the target speed input value. The current power is obtained by using the equation calculated in subsection 5.4. The power magnitude is limited to 100%, and the motor is set to use the calculated power. All the samples are shifted before the next iteration.
Following is code snippet of the implementation of the controller in the *Lego MindStrom* brick.

```
TASK(SimulTask)
{
  static boolean initialized = false;
  //notice the use of array indexes: 0 for current 1 for t-1 2 for t-2
  uint32_t elapsed_time = systick_get_ms()-init_time;
  static uint32_t prev_time=0;
  static int rev_count_1[2]={0,0};
  static int rev_count_2[2]={0,0};

  static const float coeffp[3]={1,1,0};
  static const float coeffe[3]={1,-0.8657,0};
  static const float K=0.2333;

  static const float alpha=0.1; //forgetting factor 0.9 (high = better filtering but
  static float avg_speed_1=0;
  static float avg_speed_2=0;
  static float p_1[3]={0,0,0};
  static float p_2[3]={0,0,0};
```

```
static float e_1[3]={0,0,0};
static float e_2[3]={0,0,0};
static float current_speed_1=0;
static float current_speed_2=0;
static float target_speed_1=0;
static float target_speed_2=0;
static int power_1=0;
static int power_2=0;


if(!initialized){
        //input: deg/sec (-900 to +900)
    //
    //target_speed=get_speed(&sim_config,elapsed_time)/1000;
    target_speed_1 = 1.0;
    target_speed_2 = 1.0;
        //speed in deg/msec (range goes from ca -0.9 to 0.9)
        initialized==true;
      }

if (elapsed_time >= get_duration(&sim_config))
  {
    terminate_sim();
    TerminateTask();
  }

else
  {

        //calculating current speed
    rev_count_1[0]=nxt_motor_get_count(MOTOR_PORT_1);
    rev_count_2[0]=nxt_motor_get_count(MOTOR_PORT_2);
        current_speed_1=((float)(rev_count_1[0]-rev_count_1[1]))/(elapsed_time - pre
        current_speed_2=((float)(rev_count_2[0]-rev_count_2[1]))/(elapsed_time - pre
    rev_count_1[1]=rev_count_1[0];
    rev_count_2[1]=rev_count_2[0];

        //moving average (no butterworth, introduces delay)
    avg_speed_1=alpha*current_speed_1+(1-alpha)*avg_speed_1;
    avg_speed_2=alpha*current_speed_2+(1-alpha)*avg_speed_2;
    //current_speed=avg_speed;

        //calculating current error
    e_1[0]=target_speed_1-avg_speed_1;
    e_2[0]=target_speed_2-avg_speed_2;

        //calculating current power
    p_1[0]=coeffp[1]*p_1[1]+coeffp[2]*p_1[2]+(e_1[0]+coeffe[1]*e_1[1]+coeffe[2]*e_1
    p_2[0]=coeffp[1]*p_2[1]+coeffp[2]*p_2[2]+(e_2[0]+coeffe[1]*e_2[1]+coeffe[2]*e_2

    if (p_1[0]>1) {p_1[0]=1;} //saturation
    if (p_2[0]>1) {p_2[0]=1;} //saturation

        if (p_1[0]<-1){p_1[0]=-1;}
        if (p_2[0]<-1){p_2[0]=-1;}
        power_1=(int)(p_1[0]*100);
        power_2=(int)(p_2[0]*100);
        //set current power to motor
```

```
    nxt_motor_set_speed(MOTOR_PORT_1, power_1 , 0);
    nxt_motor_set_speed(MOTOR_PORT_2, power_2 , 0);

        //debugging

    //speed_val = get_speed(&sim_config, elapsed_time);

    //shifting to old samples
e_1[2]=e_1[1];
e_2[2]=e_2[1];
        e_1[1]=e_1[0];
        e_2[1]=e_2[0];

p_1[2]=p_1[1];
p_2[2]=p_2[1];
        p_1[1]=p_1[0];
        p_2[1]=p_2[0];

      prev_time = elapsed_time;


        //send data to pc
    send_buffered_data((int)(elapsed_time), (int)(avg_speed_1*1000));
    TerminateTask();
}
```

Figure 12 shows the theoretical model output compared against the data acquired from the robot after the implementation of the controller.

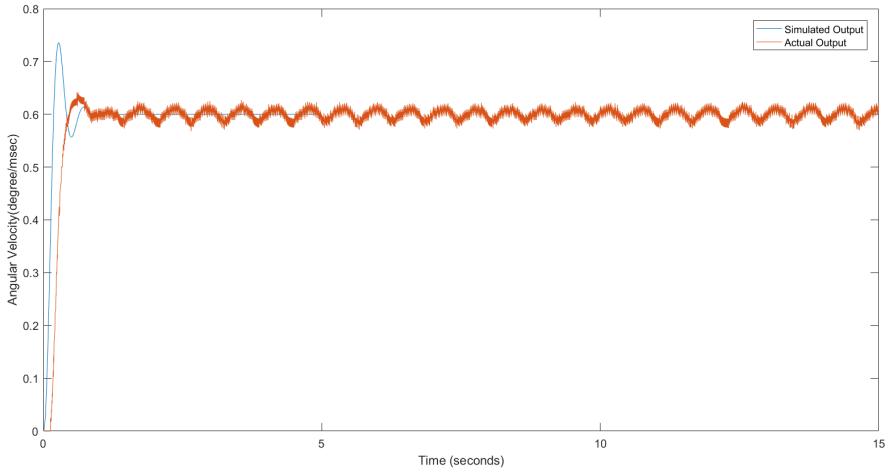

Figure 12: Simulated and Actual Output for a Target Speed of 0.6 deg/msec

6.2 Two Wheels Control

The above controller design is for one of the two motors required to make *Lego MindStorm* robot work. Since both of the motors belong to the same device, the same controller design has been used for the other motor as well. The code snippet provided in the previous section was modified to accommodate both the motors with the same calculations for the controller. The final result were found to be satisfactory as can be seen from the figure 13 and 14

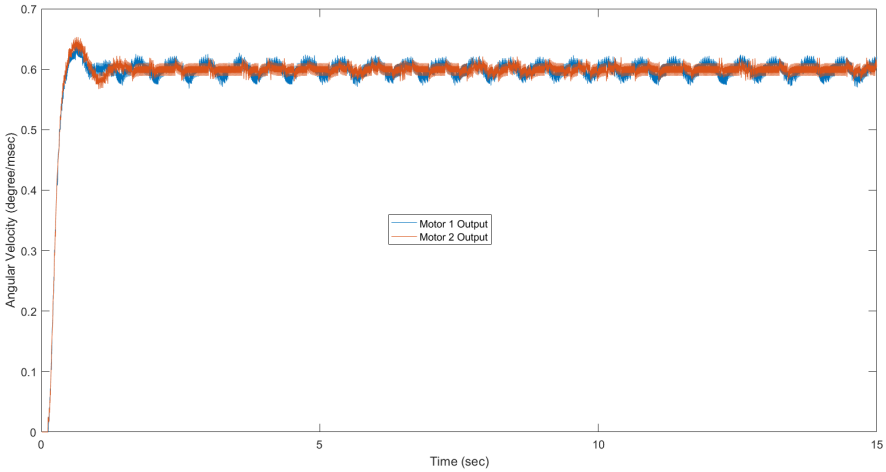

Figure 13: Comparison of Motor 1 and Motor 2 Outputs for a target speed of 0.6 deg/msec

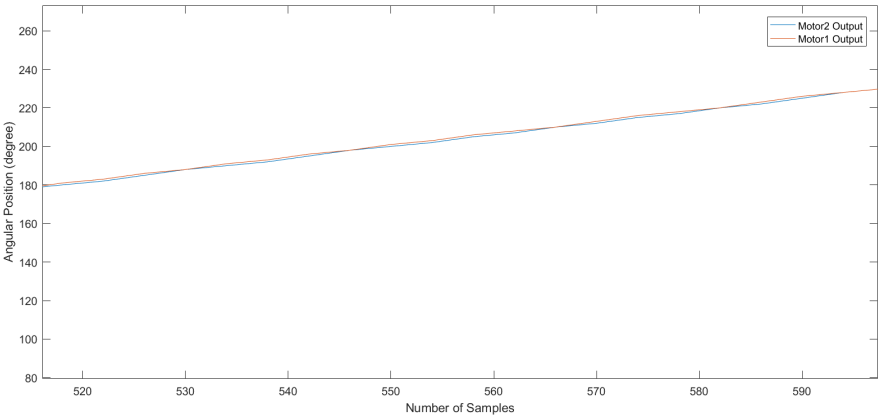

Figure 14: Comparison of Motor 1 and Motor 2 Angle