Winter Work Documentation, 2015
Control Systems Team, Team AGV
IIT Kharagpur

Content Writer and Co-Worker:
Siddharth Jha

Editor:
Ayush Pandey

From Datasheet of MMP-S22-346C

Back EMF Constant/Torque Constant (K)	2.3x10^-3 kg m
Moment of Inertia (J)	4.2369x10^-5 kgm^2
Static Friction Torque	3.53077x10^-2 Nm
Armature Inductance (L)	0.06mH
DC Armature Resistance (R)	0.28ohm

DC Motor Concepts (Ref. University of Michigan Controls Tutorials)

In general, the torque generated by a DC motor is proportional to the armature current and the strength of the magnetic field. In this example we will assume that the magnetic field is constant and, therefore, that the motor torque is proportional to only the armature current *i* by a constant factor *Kt* as shown in the equation below. This is referred to as an armature-controlled motor.

$$T = K_t i$$

The back emf, *e*, is proportional to the angular velocity of the shaft by a constant factor *Ke*.

$$e = K_e \dot{\theta}$$

In SI units, the motor torque and back emf constants are equal, that is, *Kt* = *Ke*; therefore, we will use *K* to represent both the motor torque constant and the back emf constant.

From the figure above, we can derive the following governing equations based on Newton's 2nd law and Kirchhoff's voltage law.

$$J\ddot{\theta} + b\dot{\theta} = Ki$$

$$L\frac{di}{dt} + Ri = V - K\dot{\theta}$$

Transfer Function

Applying the Laplace transform, the above modeling equations can be expressed in terms of the Laplace variable *s*.

$$s(Js + b)\Theta(s) = KI(s)$$

$$(Ls + R)I(s) = V(s) - Ks\Theta(s)$$

We arrive at the following open-loop transfer function by eliminating *I*(*s*) between the two above equations, where the rotational speed is considered the output and the armature voltage is considered the input.

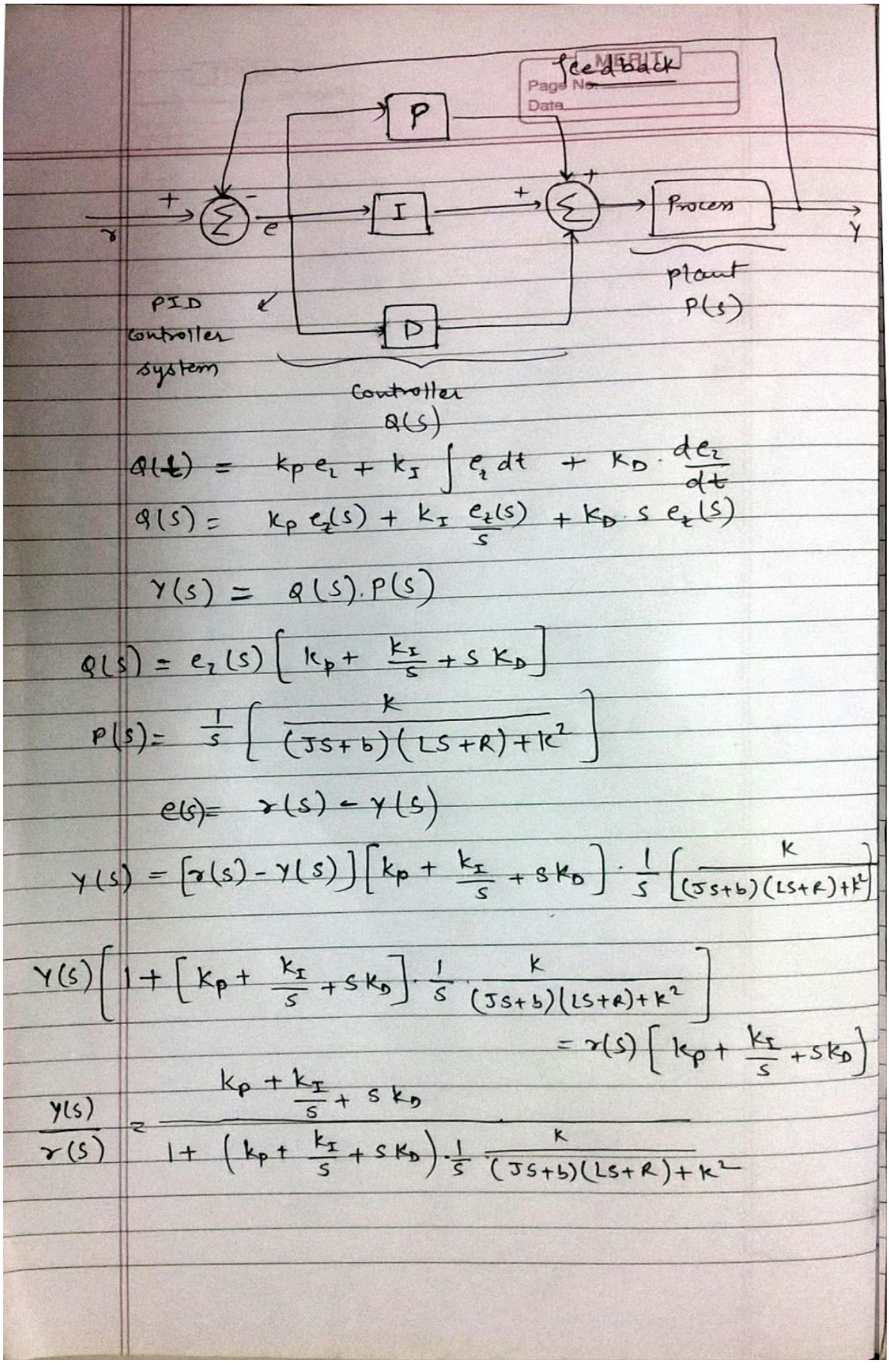
$$P(s) = \frac{\dot{\Theta}(s)}{V(s)} = \frac{K}{(Js + b)(Ls + R) + K^2} \qquad [\frac{rad/sec}{V}]$$

$$P(s) = \frac{k}{(J * s + b) \times (L * s + R) + k^2}$$

We currently don't have b so we are letting it as it is

$$\frac{2.3}{(4.2369 \times 10^{-2} * s + 1000 * b)(0.6 * s + 280) + 5.29 \times 10^{-3}}$$

The State Space form of the transfer function has been derived as below.



$$Q(t) = k_p e_t + k_I \int e_t \, dt + k_D \cdot \frac{de_t}{dt}$$

$$Q(s) = k_p e_t(s) + k_I \frac{e_t(s)}{s} + k_D \cdot s \, e_t(s)$$

$$Y(s) = Q(s) \cdot P(s)$$

$$Q(s) = e_t(s) \left[k_p + \frac{k_I}{s} + s k_D \right]$$

$$P(s) = \frac{1}{s} \left[\frac{k}{(Js+b)(Ls+R)+k^2} \right]$$

$$e(s) = r(s) - y(s)$$

$$Y(s) = [r(s) - Y(s)] \left[k_p + \frac{k_I}{s} + s k_D \right] \cdot \frac{1}{s} \left[\frac{k}{(Js+b)(Ls+R)+k^2} \right]$$

$$Y(s) \left[1 + \left[k_p + \frac{k_I}{s} + s k_D \right] \cdot \frac{1}{s} \cdot \frac{k}{(Js+b)(Ls+R)+k^2} \right] = r(s) \left[k_p + \frac{k_I}{s} + s k_D \right]$$

$$\frac{y(s)}{r(s)} = \frac{k_p + \frac{k_I}{s} + s k_D}{1 + \left(k_p + \frac{k_I}{s} + s k_D \right) \cdot \frac{1}{s} \cdot \frac{k}{(Js+b)(Ls+R)+k^2}}$$

MERIT
Page No.
Date

$$\frac{y}{V} = \frac{k}{s^3 + \left(\frac{JR+Lb}{JL}\right)s^2 + \left(\frac{bR+k^2}{JL}\right)s}$$

$$\frac{y}{v} \cdot \frac{v}{V} = \frac{k}{1} \cdot \frac{1}{\left(s^3 + \left(\frac{JR+Lb}{JL}\right)s^2 + \left(\frac{bR+k^2}{JL}\right)s\right.}$$

$=$

$$x = \begin{bmatrix} \ddot{v} \\ \dot{v} \\ v \end{bmatrix} \qquad \dot{x} = \begin{bmatrix} \dddot{v} \\ \ddot{v} \\ \dot{v} \end{bmatrix}$$

$$\begin{bmatrix} \dddot{v} \\ \ddot{v} \\ \dot{v} \end{bmatrix} = \underbrace{\begin{bmatrix} -\left(\frac{JR+Lb}{JL}\right) & -\left(\frac{bR+k^2}{JL}\right) & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_{A} \begin{bmatrix} \ddot{v} \\ \dot{v} \\ v \end{bmatrix} + \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}}_{B} V$$

$$\frac{y}{v} = k \rightarrow y = kv = Cx + DU$$

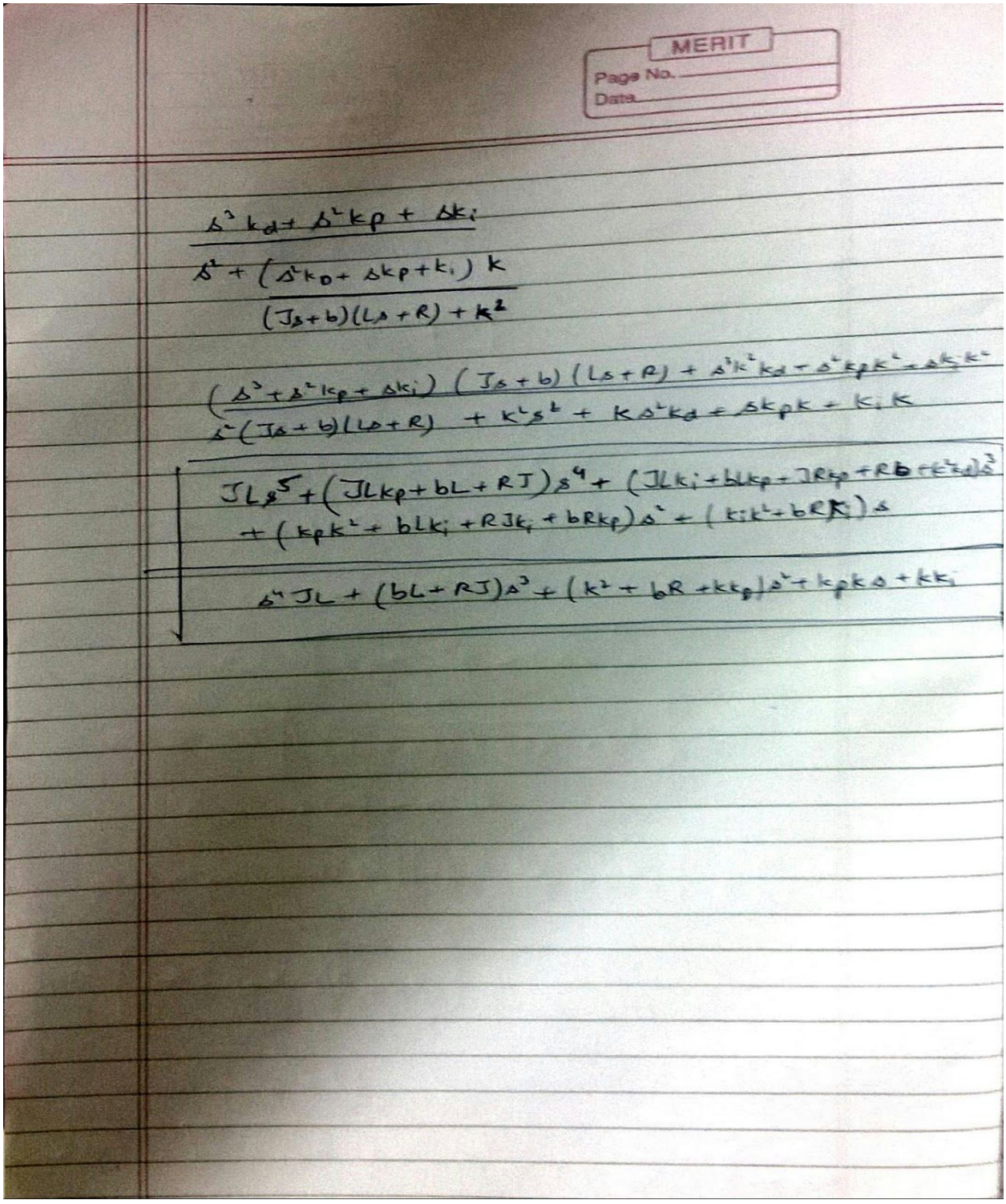
$$= \underbrace{\begin{bmatrix} 0 & 0 & k \end{bmatrix}}_{C} \begin{bmatrix} \ddot{v} \\ \dot{v} \\ v \end{bmatrix} + \underbrace{[0]}_{D} U$$

$$A = \begin{bmatrix} -28268 & -110145503 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & .0023 \end{bmatrix} \qquad D = [0]$$

30.11.15

Below: The second transfer function in expanded form



01.12.15

The PI transfer function for proper controller is given as below

Now I converted all of this in the form a MATLAB script. Then I evaluated the plant, controller, open loop system, closed loop system. The results were as follows:

```
plant =

                0.0023
  -------------------------------------
  2.538e-09 s^3 + 1.784e-05 s^2 + 0.0280s

Continuous-time transfer function.


control =

  16 s + 3
  --------
     s

Continuous-time transfer function.


plant poles are

ans =

         0
    0.0280
    0.0000
    0.0000


closedloop =

                           0.0368 s + 0.0069
  ------------------------------------------------------------------
2.538e-09 s^4 + 1.784e-05 s^3 + 0.02801 s^2 + 0.002576 s + 0.000483
```

Calculating the A,B,C,D matrices for the complete stable system.

A =

```
  1.0e+07 *
 -0.0007   -1.1036   -0.1015   -0.0190
  0.0000         0         0         0
       0    0.0000         0         0
       0         0    0.0000         0
```

B =

```
    1
    0
    0
    0
```

C =

```
  1.0e+07 *

    0         0    1.4500    0.2719
```

D =

```
    0
```

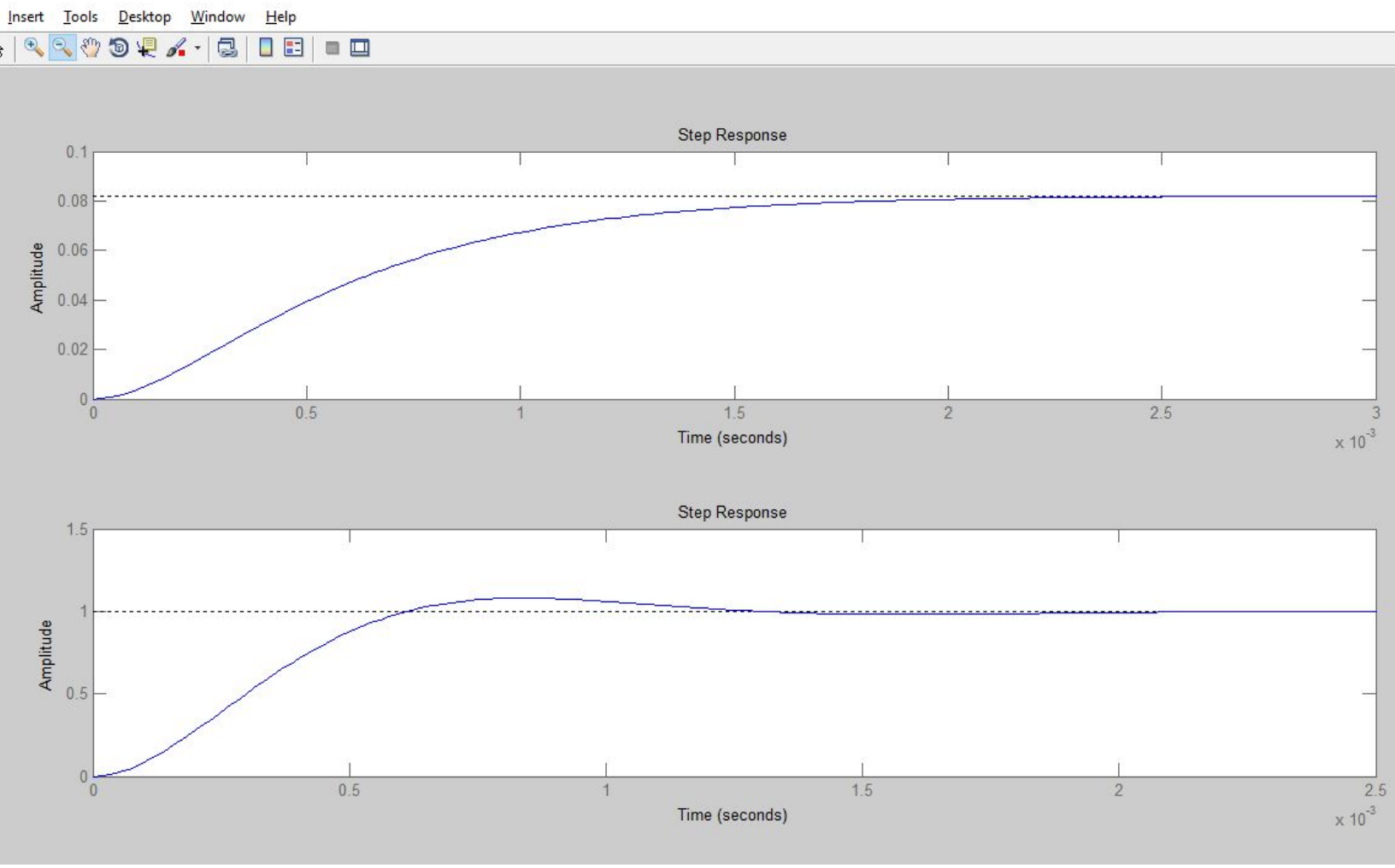
The Script:

```
k=0.0023;
b=0.1;
J=4.23e-05;
R=0.28;
L=6e-5;
plant=tf([k],[J*L,b*L+R*J,b*R+k^2]);
display(plant);
```

```
display(isstable(plant));
control=tf([kp,ki],[1,0]);
display(control);
openloop=control*plant;
closedloop=feedback(openloop,1);
display(closedloop);
display(isstable(closedloop));
subplot(211), step(plant);
subplot(212), step(closedloop);
[num,den] = tfdata(closedloop,'v');
[A,B,C,D]=tf2ss(num,den)
```

Upon performing stability analysis, we find out that both the transfer functions are stable. The ‘v’ in tfdata ensures that the resulting num and den are 1D vectors.

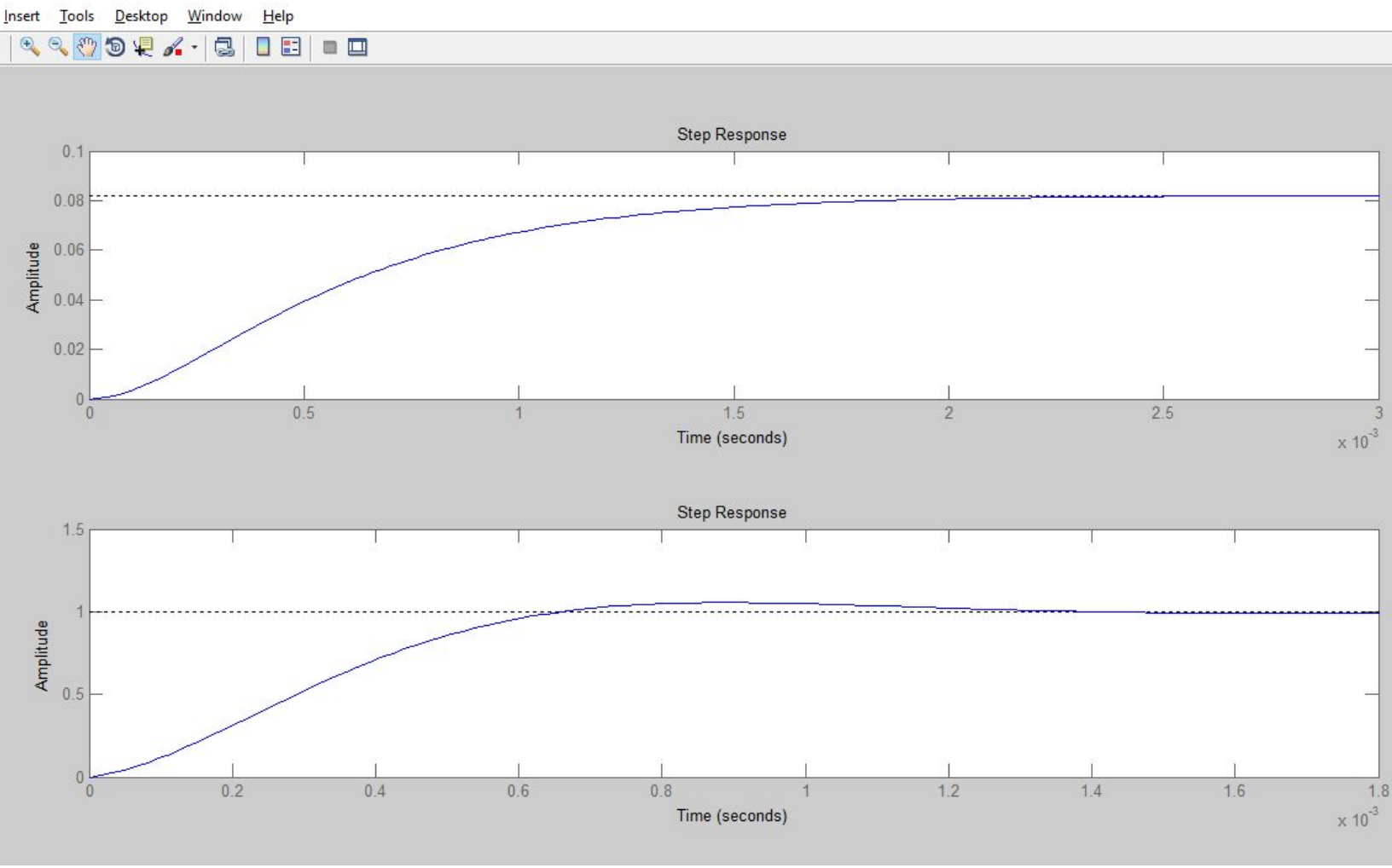
Now using the built in tool ‘pidtune’ of MATLAB, the values of kp and ki were automatically calculated and used to plot graphs. The new values are as follows:



kp=22.3
ki=4.28e+04

We see that although with overshoot, the rise time has gone down considerably, owing to the presence of large integral component.

The addition of ‘kd’ calculated using pidtune methods doesn’t affect the plot much as it is really small. (kd=0.000554)



The syntax for pidtune command is as follows.

```
F=pidtune(plant,'PI');
display(F);
G=pidtune(plant,'PID');
display(G);
```

PID Tuning Reference Table:

CL RESPONSE	RISE TIME	OVERSHOOT	SETTLING TIME	S-S ERROR
Kp **Ki** **Kd**	Decrease Increase Small Change	Increase Increase Decrease	Small Change Increase Decrease	Decrease Eliminate No Change

02.12.2015

Now we connected the Roboteq to our PC and measure the open loop gain of the motor by giving it a value of 1000, but as expected, the motor just kept spinning and did not stop as it was open loop condition. We had to e-stop promptly.

Now we had to measure the transfer function of the entire system in real life. For that we took values of the motor command to Roboteq between -1000 and 1000 and also the values of the feedback, the actual motor rotation value between -1000 to 1000. The corresponding values were plotted too for the PID parameters of 16,3,0 of Kp,Ki,Kd respectively. We saw the response was good enough, the log file is available here.

Then we used the system identification tool of MATLAB to calculate the transfer function. The corresponding code used:

```
dat3=iddata(y1,u1,1/12.5);
sys4=tfest(dat3,4);
display(sys4);
[num2,den2]=tfdata(sys2,'v');
roots(den2);
```

Each function is briefly explained as below:

iddata is the function used to generate a data structure that holds ordered data with timestamps and intervals specified. The transfer function estimation of MATLAB uses this to estimate a continuous time transfer function. u2 is the input data vector and y2 is the output data vector, and 1/12.5 is the sampling period of the data. That means data was measured every 0.08 seconds by Roborun.

tfest is the main function used to estimate a transfer function (continuous time) that best fits the given data. On my system, it took about 6-7 seconds to identify a TF from about 250 samples of

input output data with 90% accuracy. The 4 in the arguments is the measure of the number of poles in the transfer function we desire. As we are estimating one of a DC motor, using a PI controller, we estimated that 4 poles would a suitable approximation.

The result of the code was as follows:

```
sys4 =

  From input "u1" to output "y1":
      0.6407 s^3 + 65.4 s^2 + 0.7498 s + 74
  -----------------------------------------
  s^4 + 11.07 s^3 + 68.76 s^2 + 12.78 s + 78.04

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 4   Number of zeros: 3
   Number of free coefficients: 8
   Use "tfdata", "getpvec", "getcov" for parameters and their
uncertainties.

Status:
Estimated using TFEST on time domain data "dat3".
Fit to estimation data: 90% (simulation focus)
FPE: 1198, MSE: 1066
```

Now we checked for system poles, stability and step response.

```
>> isstable(sys4)

ans =

     1

>> pole(sys4)

ans =

  -5.5341 + 6.0810i
  -5.5341 - 6.0810i
  -0.0001 + 1.0744i
  -0.0001 - 1.0744i
```

The next challenge lies in identifying the plant from the system TF.

//TODO: FIND A STABLE TF FROM THE ABOVE DATA. THE TRADITIONAL METHOD IS PROVIDING AN UNSTABLE TF.

Now we move on to tuning the PID constants of the front hub motor of Eklavya 4.0. It is a generic hub motor extracted from an electric scooter and by no means were we able to gather a reputable datasheet for the same. Right now we are using a generic BLDC motor driver that we assume is one with good performance. More information on working of the BLDC motor can be found on page 3 of this document.

Right now the input value to the PID used in the vx_pid node written in C++ is the error value between the target vx (referred to as vt from now on) and actual vx (va from now on).
Hence, Error = vt-va
The output value is the value of 12 bit DAC which is given to the generic BLDC motor driver to drive the motor. This has two main problems, firstly we have tuned the PID manually without knowing the transfer function and also we are currently assuming the performance of the motor driver to be perfect.

While nothing can be done for the second problem as of now, we can still incorporate some of the errors of the controller and also rectify the first problem completely using the approach used for the DC motor previously.

Procedure for Taking Values from ROS Nodes

I first edited the codes in order to ensure that the loop rates of the publishers are equal. The loop rates were all set to 20, and then took a ROSbag that contained the data. The publishers were split into two codes, namely vx_pid.cpp and modeswitcherdue.cpp and even after changing the loop rates to 20, for some reason the number of observations came out to be different. So now I have changed all the publishers to be inside vx_pid.cpp. Hopefully now the number of observations should be the same.

The bash codes for this process are:

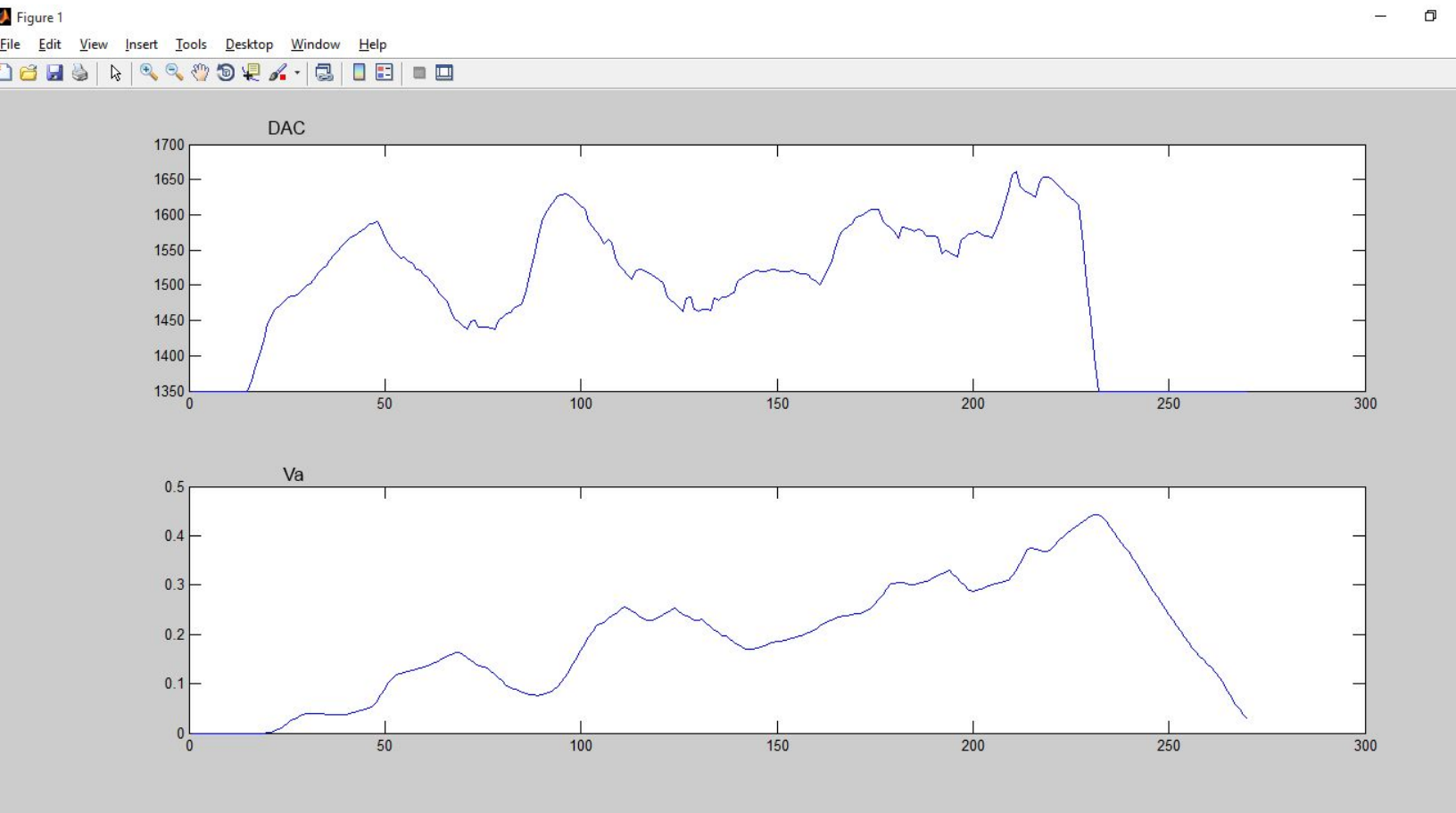
```
roscore
//open new terminal
rosbag record -a
//press Ctrl+C to abort. Keep varying the values of vt.
rostopic echo -b file.bag -p /topic > data.txt
//all three topics have to be stored in separate files
```

The PID has been tuned manually for now. The earlier response was pretty bad, but now it's okay. Although some jerks are still there, it's perfectly acceptable.

//rqt_plot of PID here.

03.12.15 and 04.12.15

Then for some time, I manually tried to get a relation between the DAC and Va in a closed loop tuned system as ours. Graphs, such as the ones below were obtained. The data although was quite good enough to run the robot, had some strange relation between the two variables.



The above figure shows smoothened out data, sans any peaks. The original data on the other hand had a lot of peaks that were due to incorrect placement of the encoders on our vehicle. That was a small issue. So I used the following code in MATLAB to render the above graphs. The va value was too spiky so I had to apply it twice.

```
u1=[1350,1350……1603,1605,.....1350]; //from
rosbag
y1=[0,0,.....0.92324,0.9221,.....
.....0]; //from rosbag
//above data is representative only.
uu1=smooth(u1);
yyy1=smooth(smooth(y1));
dat1=iddata(yyy1,uu1,0.05);
tfest(dat1,3);
```

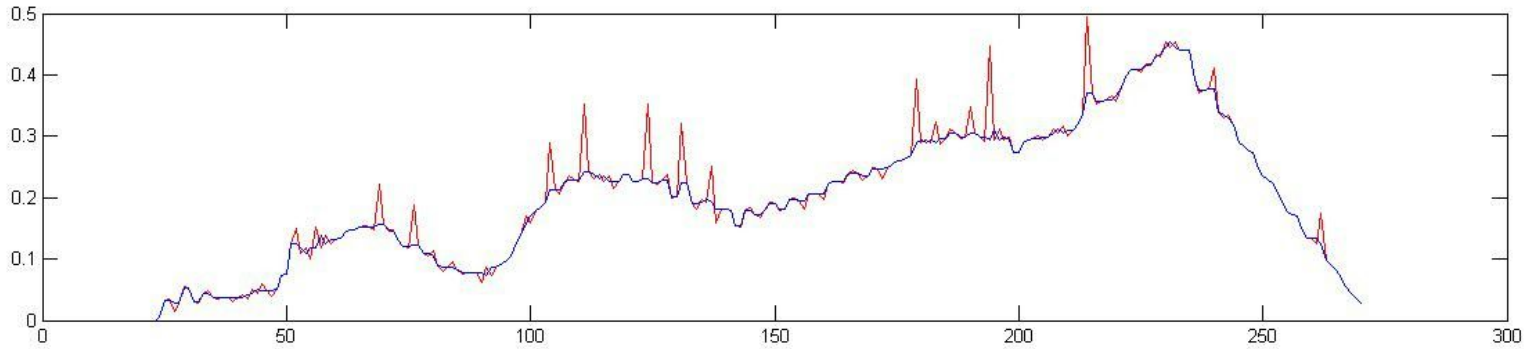
The output was not one with very good accuracy.

```
ans =

  From input "u1" to output "y1":
  -5.396e-05 s^2 + 0.0007774 s + 0.0001342
  ----------------------------------------
    s^3 + 0.9427 s^2 + 4.408 s + 0.4833

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 2
   Number of free coefficients: 6
   Use "tfdata", "getpvec", "getcov" for parameters and their
uncertainties.

Status:
Estimated using TFEST on time domain data "dat1".
Fit to estimation data: 55.66% (simulation focus)
FPE: 0.00377, MSE: 0.002798
```

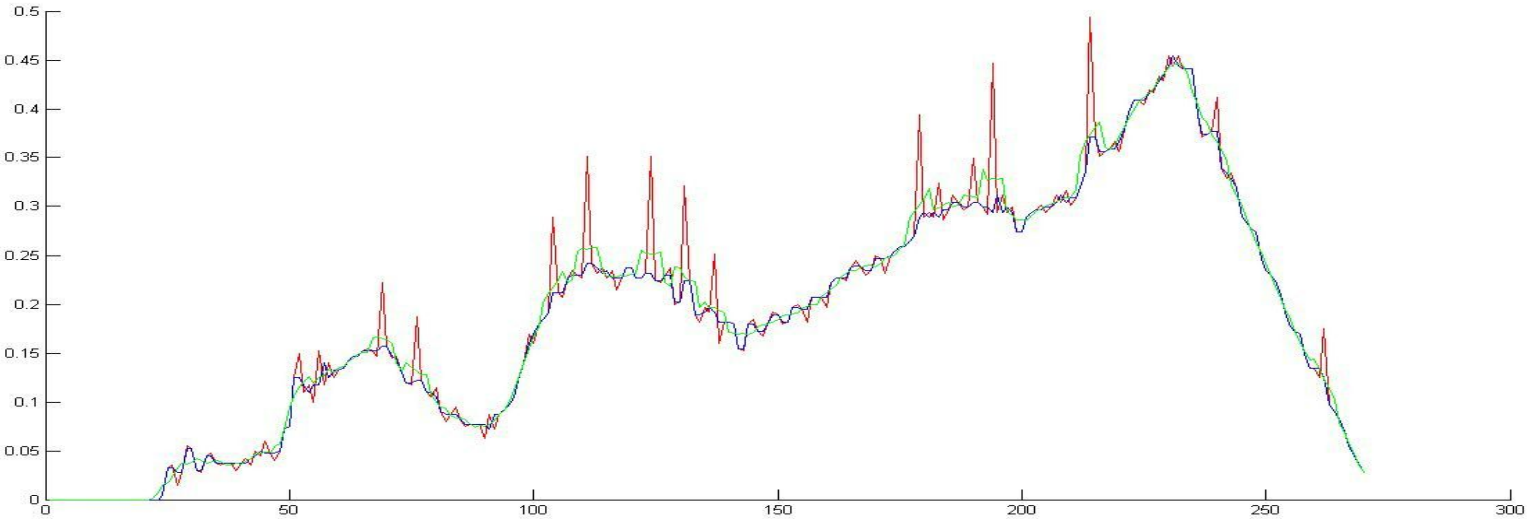
Now I learnt about another filter called Median Filtering. The output generally has lesser spikes than the input and it doesn't manipulate the input as much as the Smooth function. The syntax is:

```
dat_med=medfilt1(dat,3);
/*here we are taking the median of the value of 3 to the left and 3
to the right values of each value*/
```

Below is the comparison plot between the input value (in red) and median filtered value (in blue)



Below a comparison of original (red), median (blue) and smooth (green) is shown



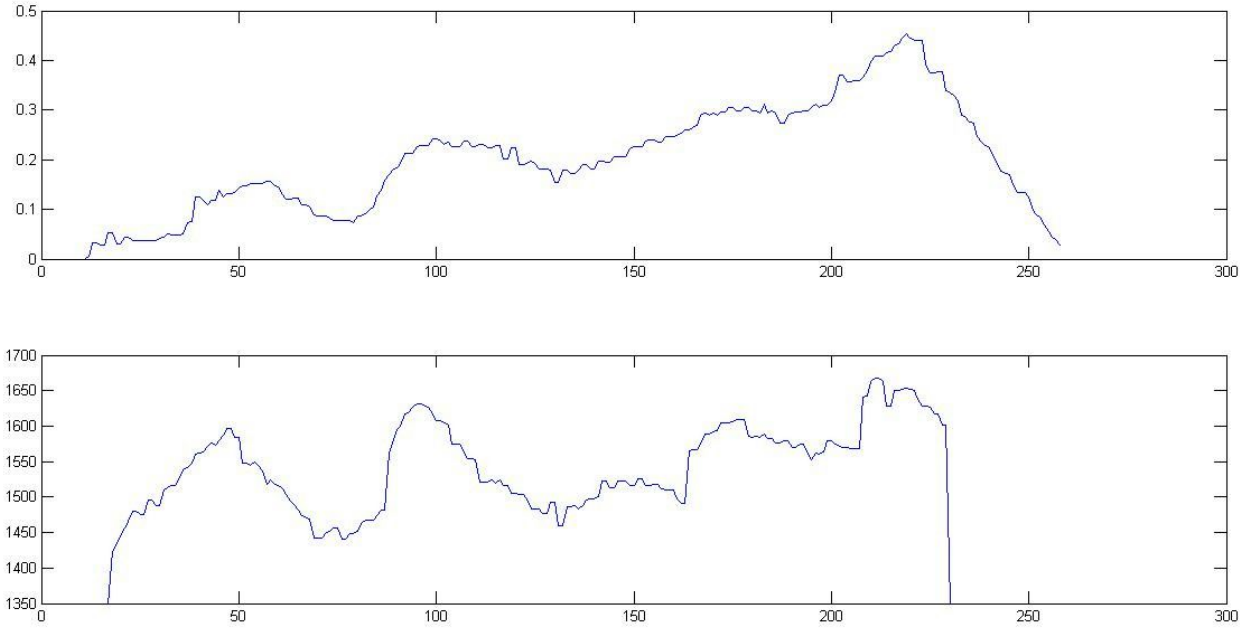
But even after median filtering, only 56.85% accuracy could be achieved

```
tf_closed =

  From input "u1" to output "y1":
  -8.401e-05 s^2 + 0.0009076 s + 0.0001253
  ----------------------------------------
    s^3 + 0.9727 s^2 + 4.533 s + 0.4209

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 2
   Number of free coefficients: 6
   Use "tfdata", "getpvec", "getcov" for parameters and their
uncertainties.

Status:
Estimated using TFEST on time domain data "dat_med".
Fit to estimation data: 56.85% (simulation focus)
FPE: 0.003722, MSE: 0.002587
```

I tried this with different data too, but low accuracy values like 22% and 20% were coming up. So I came to a conclusion that the data has some kind of a delay. The response of the robot isn't instantaneous and as apparent from the DAC,Va subplot, there is a noticable delay between the two peaks.

Then I decided to shift the data to the left and the right, deleting irrelevant values, so that the final peaks turn up at the same position. I deleted 12 initial and 12 final values respectively from the two plots and the resulting data (median filtered) was as given in the plot below.

There is almost coincidence between the peaks and trenches, and I expected the tf to come out to be quite good in accuracy.



To my surprise, not much change at all (actually a drop in accuracy).
That's when I decided to change the approach.

```
ans =

  From input "u1" to output "y1":
  0.0002509 s^2 + 0.001033 s + 0.0001741
  ---------------------------------------
    s^3 + 1.227 s^2 + 5.451 s + 0.6706

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 2
   Number of free coefficients: 6
   Use "tfdata", "getpvec", "getcov" for parameters and their
uncertainties.

Status:
Estimated using TFEST on time domain data "dat_coin".
Fit to estimation data: 51.82% (simulation focus)
FPE: 0.00407, MSE: 0.002977
```

05.12.15

Now I started on a completely different approach. I earlier was varying the v_target using the Xbox controller while in closed loop. Now I wrote a modified C++ node called vx_pid_bypass that completely bypassed the PID control system and manually sent in values like 1350, 1700 etc. for DAC in an open loop and measured the resultant values of encoders, in order to calculate the velocities.

A small problem was that I couldn't simply use the delay functions as they would affect the loop rate of the ROS node. The solution had to lie in some kind of interrupt call. The function that had to be called was called time(&now) that gives the time in seconds since January 1,1970. The function had to be read once at the start of the code, and then the values taken on each looping, and when the values lie in our desired range, we call the respective functions.

```
#include<ctime>

start=time(NULL); // at the start
now=time(NULL); // in each loop
difftime(now,start); /*calculates basically the time for which the
code has been running for*/
```

After a lot of meaningless time spent on debugging, I finally managed to get the function running. The data was collected for three type of signals:

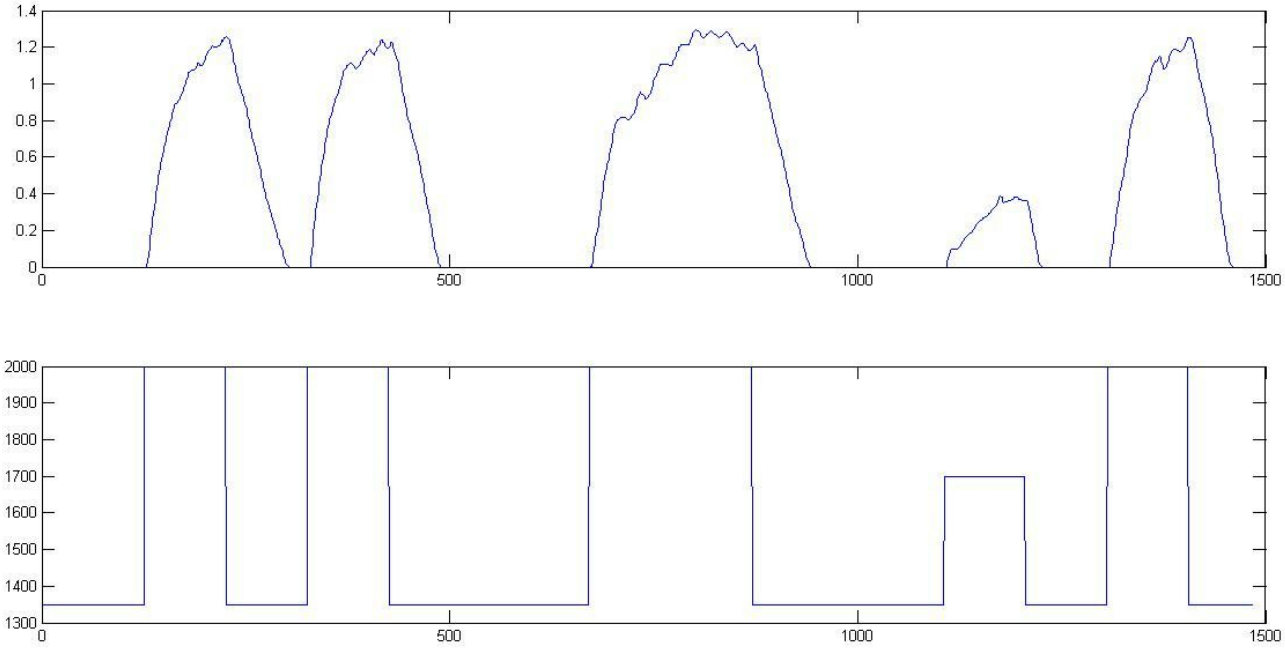
- Single Step
- Double Step with equal amplitudes
- Double Step with different amplitudes.

The values were then placed in the vectors in MATLAB after median filtering and the transfer function was estimated from the same.

```
tfest(dat4,3)

ans =

  From input "u1" to output "y1":
  0.001011 s^2 + 0.0006967 s + 1.801e-06
  -------------------------------------
   s^3 + 1.16 s^2 + 0.3351 s + 0.007859
```

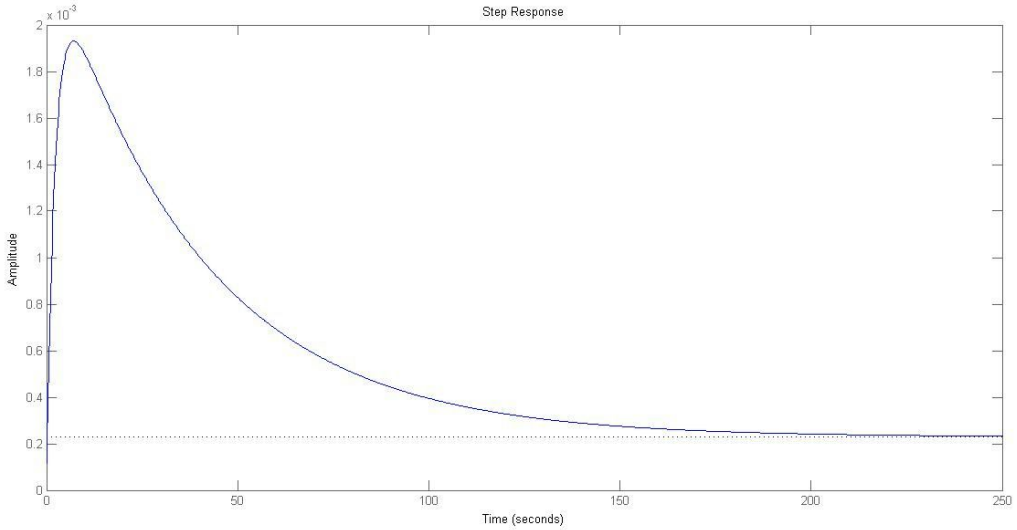
```
Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 2
   Number of free coefficients: 6
   Use "tfdata", "getpvec", "getcov" for parameters and their
uncertainties.

Status:
Estimated using TFEST on time domain data "dat4".
Fit to estimation data: 78.61% (simulation focus)
FPE: 0.01103, MSE: 0.01091
```

The corresponding Graph:



06.12.15

The step response of the open loop controller as we had calculated was as follows:



Which as we can appreciate is stable but has really high overshoot. As the input variable was the digital value feeded to DAC (in 1000s) and the output was a velocity term (in scale of e-01) , we won't expect the step response to settle anywhere near 1 and that is what it did. So basically if we give a DAC value of 1, the velocity will settle down to 0.22*10^-3 m/s, both of which are completely meaningless. So the above plot has no real significance.

Then the PID was tuned in MATLAB only. As there was close to 80% accuracy in the tfest data, we expected the PID constants to be fairly correct.

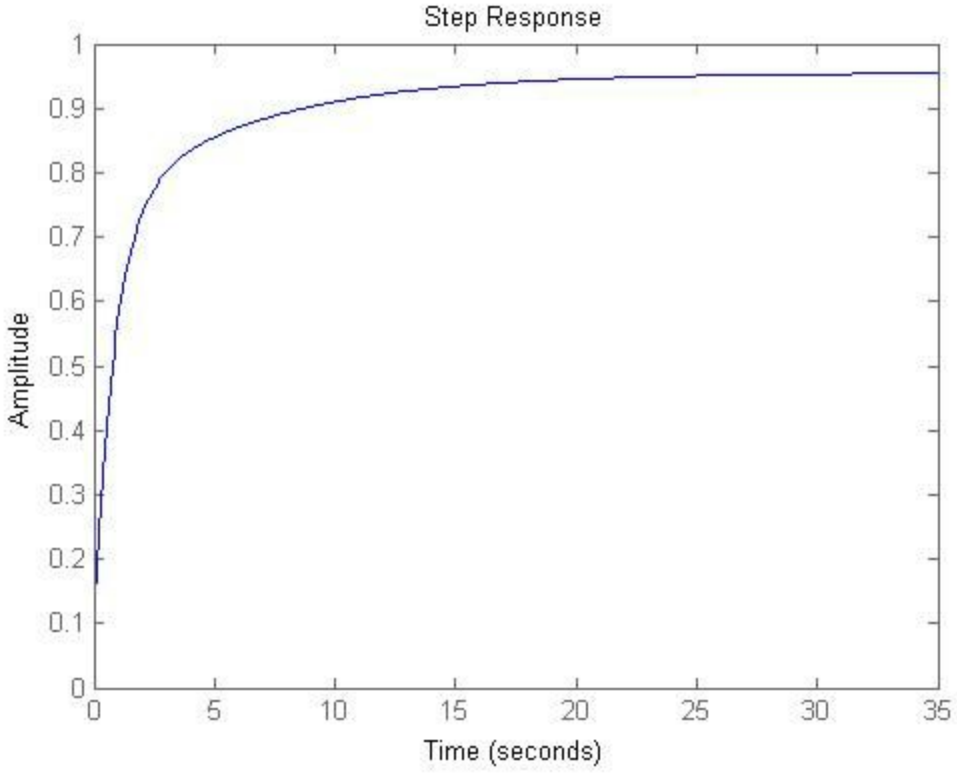
The result was:

```
ans =

             1
  Kp + Ki * --- + Kd * s
             s

  with Kp = 1.35e+03, Ki = 243, Kd = 1.86e+03

Continuous-time PID controller in parallel form.
```

We tested the robot using these constants and it did not work that well.

Then we tried to tune the PID ourselves and check out whether the constants we identified are good or not. Then we plotted the step response, considering feedback to be 1, as we already assume in the code

The step response we finally got was great!



It is still a bit on the slower side, but we can safely assume that we have found a respectable well rounded transfer function for the motion component of our robot!

So I finally conclude by saying that

```
0.001011 s^2 + 0.0006967 s + 1.801e-06
--------------------------------------
 s^3 + 1.16 s^2 + 0.3351 s + 0.007859
```

is the transfer function of the robot's forward motion component…

Now I checked MATLAB's inbuilt PID tools for optimal tuning.

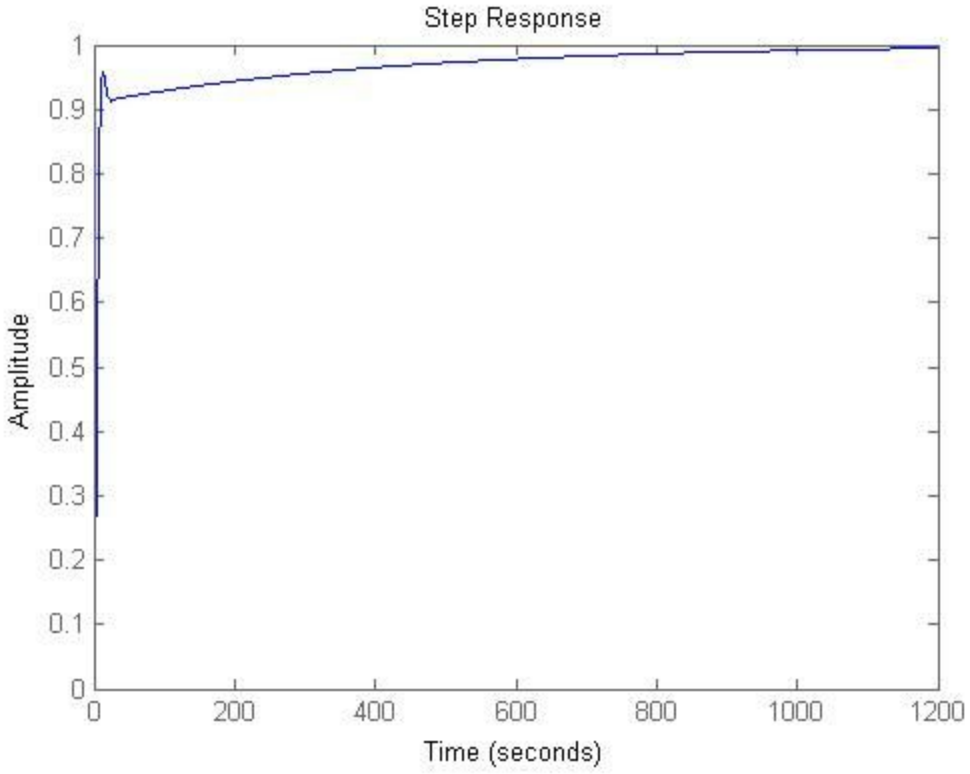
First I used the command pidtune, which has the syntax

```
pidtune(open_loop,'PID');
```

and which gave the result

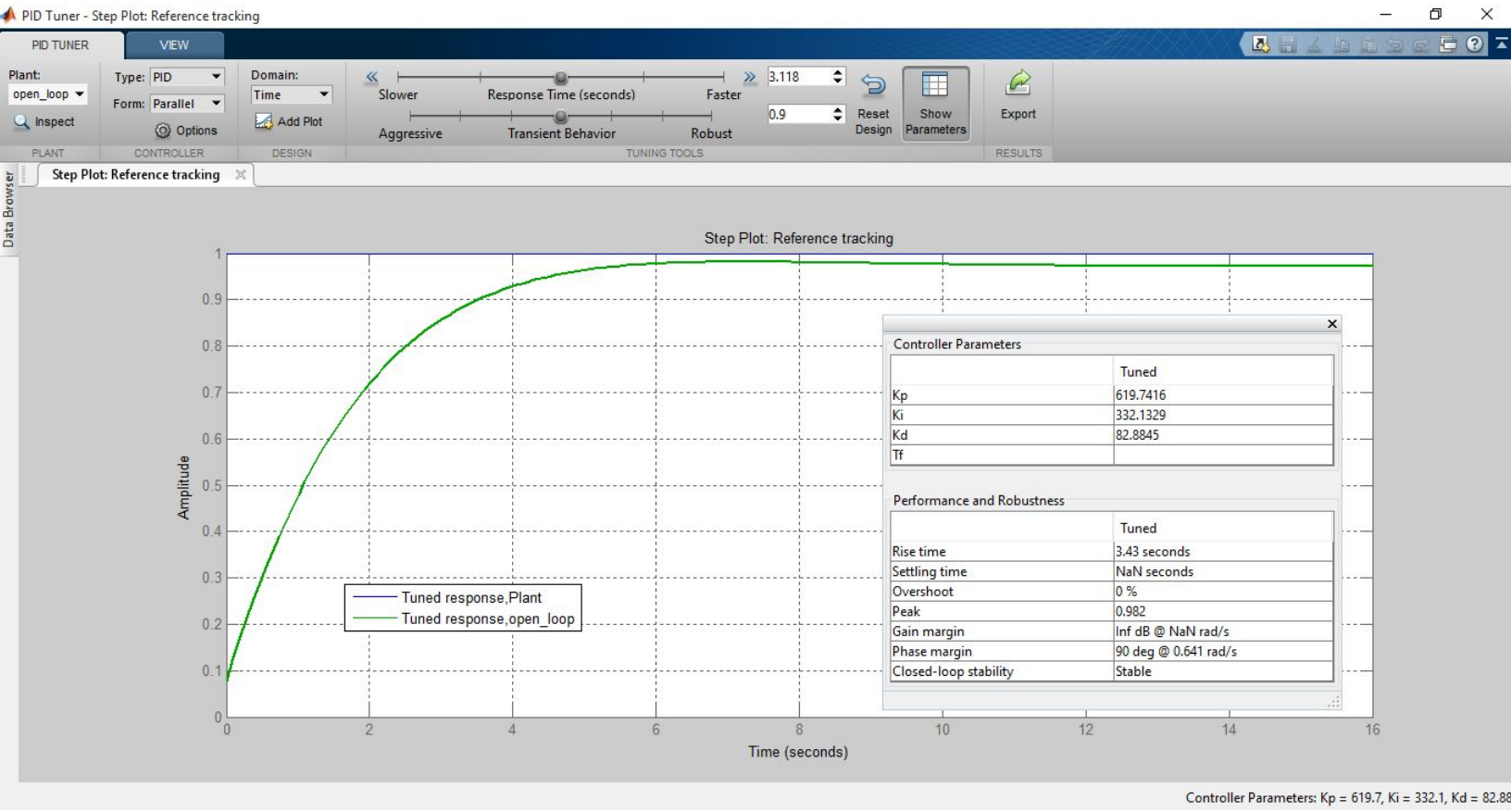
```
ans =

        1
  Ki * ---
        s

  with Ki = 104

Continuous-time I-only controller.
```

Which was quite peculiar as the robot can't run on an I only system that well. I nevertheless decided to plot the response and this is what I got.



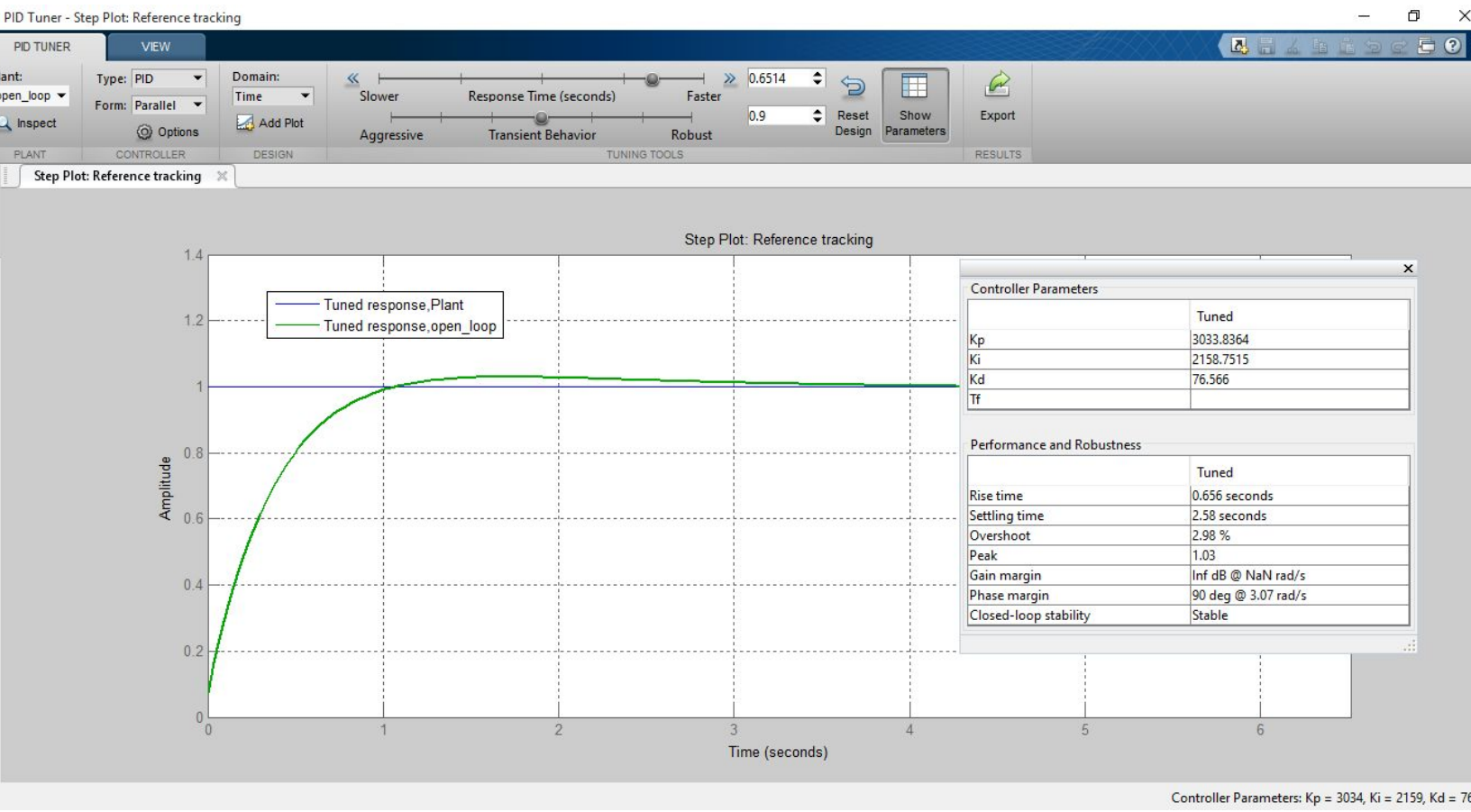
Which was really nice in the rise time and steady state error department. But as we can appreciate, the settling time is close to a 1000 seconds. The controller thus rises really fast and settles really slowly. In addition, absence of Kd leads to a lot of bumps in the real world tests and as expected the robot wasn't able to move well.

Now I tried tuning the PID using the MATLAB's pidtool app. What I finally settled on were really nice values with and rise time both really low, and almost zero overshoot and infinite robustness. The settling time and steady state error were still an issue.



When I tried to decrease the settling time and steady state error, the plots kept becoming better and this one in particular was amazing
But the practical error was that our robot can't afford that high value of Ki and hence can't have that fast a rise time. What basically kept happening for both the above case and the case below was that Ki was above 300, and that meant our robot kept moving in a stop and go fashion. We don't know whether it is a bottleneck for the motor driver, motor or the encoders, but it surely doesn't let the robot behave in a way it should.

So in practicality a steady state error always remains, but the value settles to 0.964 for a step response, which in our case is perfectly acceptable, considering that too many jerky motions already hamper the encoder readings and getting this precision is not with the ROS node.



So here is finally a comparison of the three step responses in a single plot.


Figure 1
File Edit View Insert Tools Desktop Window Help
Open File
Step Response
STEP RESPONSES FOR PLANT
RED: PIDTool Tuned
GREEN: pidtune Tuned
BLUE: Manually Tuned
Amplitude
Time (seconds)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
0
200
400
600
800
1000
1200