**Robot Controller System Report**

Last Updated – April, 06, 2021

Group Big Gain, ELEC 391, University of BC, Vancouver, BC, Canada

**Abstract**

Implementing the Controller

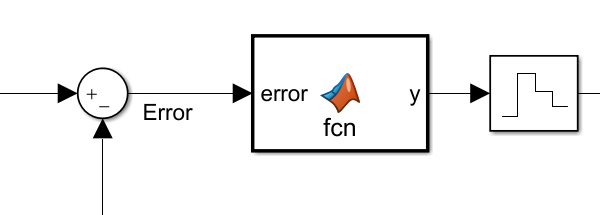
* MATLAB Function
* Arduino Code

Tuning Strategy (each joint)

Robot Kinematics

Path planning

**Implementing the PID Controller (MATLAB Function)**



To set up the implementation of the PID Controller, we first determined the control frequency to be 10 kHz, which gives the PID controller a run time of 0.0001 seconds, which is a bit higher than the ISR time of the Arduino, as we will see later. This is to allow some extra time to make sure we are giving the Arduino code enough time to run and not interrupting it prematurely. This gives a control frequency of .

To implement a filtered derivative, we decide to average 8 samples on each iteration of the controller because this gives a very accurate response and overlaps exactly with the built-in PID block with a step input. The filter pole for the filtered derivative is chosen to be 5000 to average 8 samples because

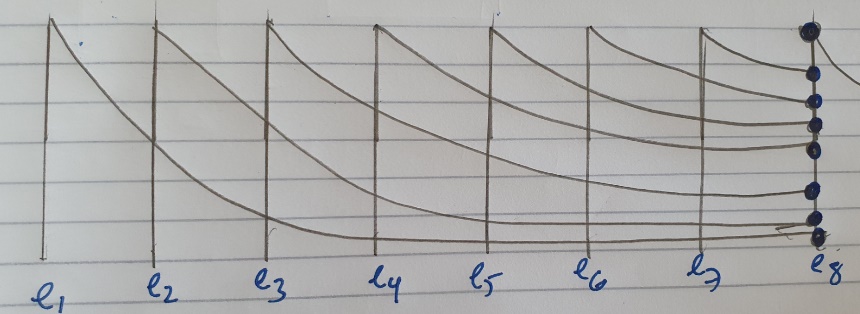
A high filter pole will signify less delay in the response and it is also averaging 8 samples which is sufficient for high accuracy. With these decisions made beforehand, we can proceed with the code, a more detailed explanation of the filtered derivative will be given after displaying the code below:

****

****

The values that we want to retain between the times the controller function is called are assigned as persistent, or global variables in MATLAB. The proportional component scales the error and the integral component is a cumulative sum of all the previous errors up to the present one.

**Filtered Derivative**



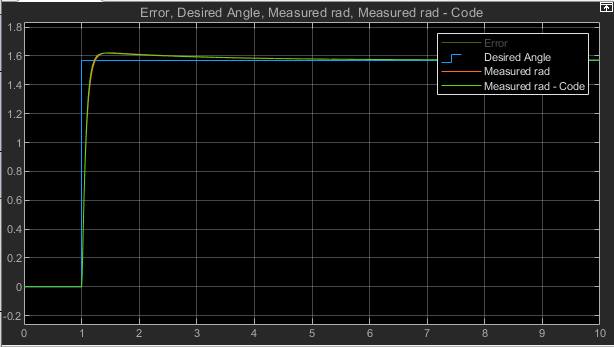
*Figure 1: Averaging 8 samples for the derivative component accounts for the exponential decay contributions from the previous errors*

This weighting is accounted for in the *ept (exponential – pole (filter) – sampling time)* vector of 8 numbers in the MATLAB code.

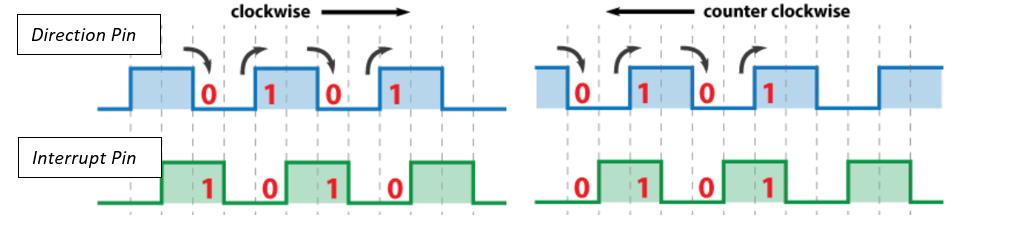
where n = sample number from 1 to 8, p = filter pole (5000) and Ts = 0.0001s

Ideally the with unlimited samples ‘n’ sampled continuously but we are only sampling 8 samples hence there is a discretization error which causes the sum to be 0.883, hence we scale the *ept* vector by 1/0.883 = 1.132 to make the sum of all 8 numbers exactly equal to 1.

We accumulate eight errors (error – previous error) in the *unweighted\_differential\_outputs* vector and do a dot product with the ept vector to output a **filtered** derivative that can then be used in the output. It is important to shift the errors in *unweighted\_differential\_outputs* to the right by 1 each time the PID controller function is called in order to introduce the **new** error to the first index so as to make sure we are doing the dot product with a new *unweighted\_differential\_outputs* vector each time the PID function is called. This is what the for loop at the end of the MATLAB code shown above does.



*Figure 2: MATLAB PID function can exactly track the response generated by the built-in PID block in Simulink with arbitrary tuning values*

**Arduino PID Controller (C-Code)**

For the microcontroller, we need to use the optical encoder to track the real-time change in angle of the motor – this was represented by the feedback in the Simulink model. The pulse() Interrupt service routine in the Arduino Code interrupts the main code every time the *interrupt\_pin* rises and we check another pin that indicates when light passes through the slits of the encoder (whether it is high or low) – this is called the *direction\_pin.* Whenever the Direction pin is high and the interrupt is high, we increment a counter which can then be converted to a positive angle (clockwise rotation) otherwise, if the direction pin is low when the interrupt pin is high (when the interrupt activates) counter is decremented and anticlockwise rotation.

The counter can then be multiplied with the resolution of the arduino which basically indicates how many degrees are traversed by each pulse i.e. 2pi/(Counts\_per\_turn). The encoders we are using have a CPR of 128.

**ISR Frequency**

|  |  |  |
| --- | --- | --- |
| Operator | Instances | Cycle Count |
| = | 25 | 100 |
| pinMode | 4 | 24 |
| attachInterrupt | 1 | 6 |
| For Loop | 2 | 10 |
| Multiplication | 7 | 126 |
| Division | 2 | 40 |
| Addition/subtraction | 10 | 180 |
| Setup() | 1 | 12 |
| If/else | 2 | 21 |
| Return | 2 | 6 |
| Analogread | 1 | 5 |
| Digitalread | 1 | 8 |
| Digitalwrite | 1 | 6 |
| **Total: 554 Cycles** | | |

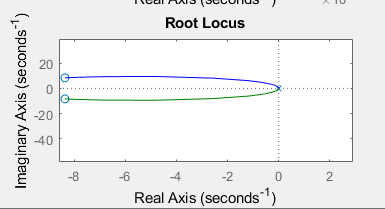
The total run time of the Arduino code can now be calculated – we know that the microcontroller can run cycles in a second with an operating frequency of 16 MHz, hence it takes 1/16MHz to run one cycle. Therefore:

With a sampling time Ts in Simulink of 0.0001 seconds, which is roughly three times more than this value, we are being conservative enough to make sure that we are giving the Arduino code more than enough time to run on each iteration.

****

****

**Tuning Strategy**

To demonstrate how we obtained the initial tuning values (without the cosimulation) just based off the transfer functions, we will consider the DCX22M motor.