Ólafur ÞÓRSSON
François BARNOUIN
Émilie TRAN

Project 3 - Dynamic controller behavior

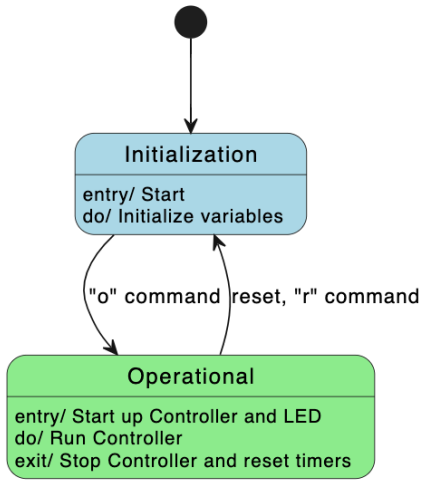
Part 1

For this part, we just implemented the Initialization and Operational states. Here is how we proceed in each state:

Initialization state:
- We indicate that we have entered the initialization state
- We initialize all the variables/values
- We can press on the keyboard “o” to go to the operational state

Operational state:
- We indicate that we have entered the operational state and turn the LED on and we initialize all the timers used by the motor (see extract of Proj2 report for more details about the timers at the end of the file)
- We can press on the keyboard “r” to reset and go to the initialization state
- We reset all the timers before exiting

Here is the corresponding UML diagram:



Part 2

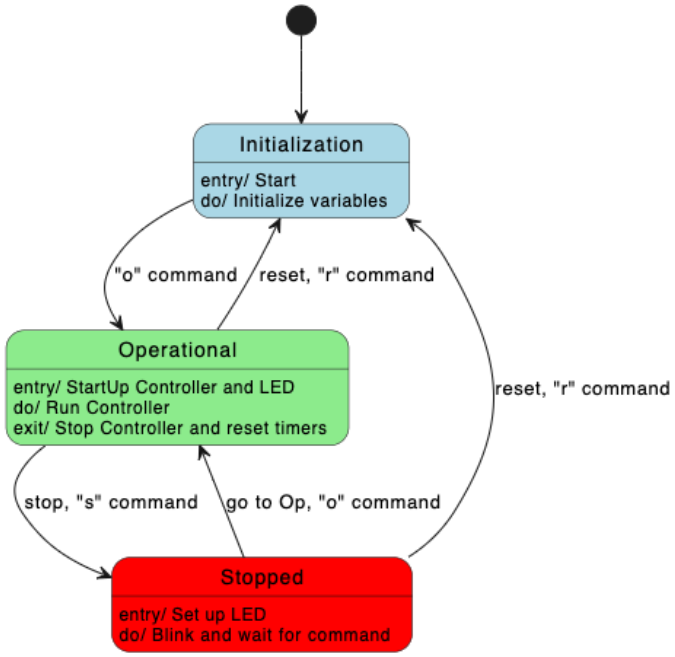
First we added the Stopped state in the following way:
- We indicate that we have entered the stopped state
- We toggle the LED at a 2Hz frequency
- We wait for a command (“o” for operational, “r for reset”)

For the operational state, we added a fault detection: we read the fault pin and if its value changes to low, we go from operational state to stopped state. We also added an emergency

button that uses the same character “s” for the stopped state that ensures the transition from operational to stopped.

To test this, we manually connected the FLT to the ground and the motor stopped (see video).

Here is the corresponding UML diagram:



Part 3

We added the PI_Controller class using polymorphism. We override the update function by adding the new parameters ini the given control formula.

$$u(t) = K_p e(t) + K_i E(t) = K_p \left(e(t) + \frac{1}{T_i} E(t) \right)$$

$$E(t) = \int_0^t e(\tau)\,\mathrm{d}\tau \approx \sum_{k=0}^{t} e_k\, T$$

Normal Flow: The motor controller adjusts the speed based on the error between the reference and actual speed, staying within the motor's operational limits.

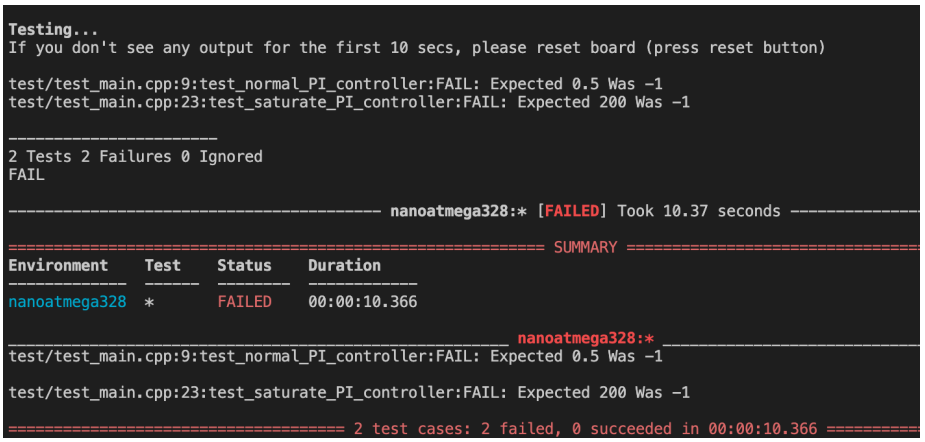
Alternate Flow: The motor controller encounters a situation where the control signal exceeds the motor's limits and it saturates (due to integral-windup for instance). We stop integrating to ensure that the error does not increase without bounds.

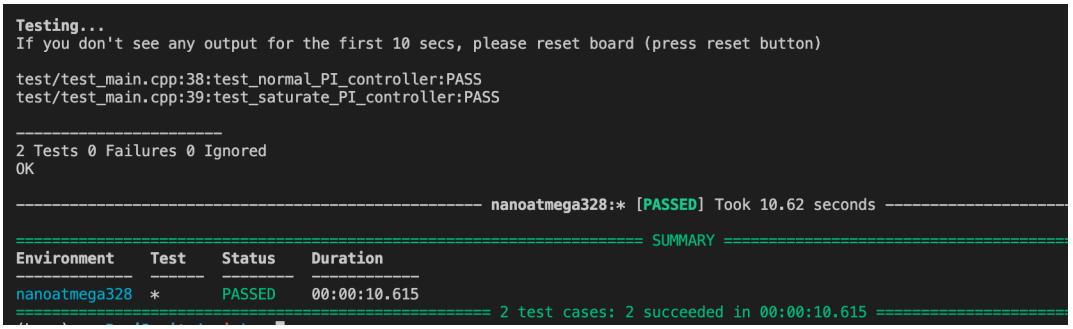
To check these two flows, we implemented two tests units in our code:

test_normal_pi_controller: We verify if the motor adapts the value of the control output correctly for progressive updates where the error stays within the boundaries of the motor's capabilities.

test_saturate_PI_controller: We verify the behavior when the control output saturates by checking if the integral sum stops increasing in this case.

Here are the results before and after implementation of the pi_controller class:



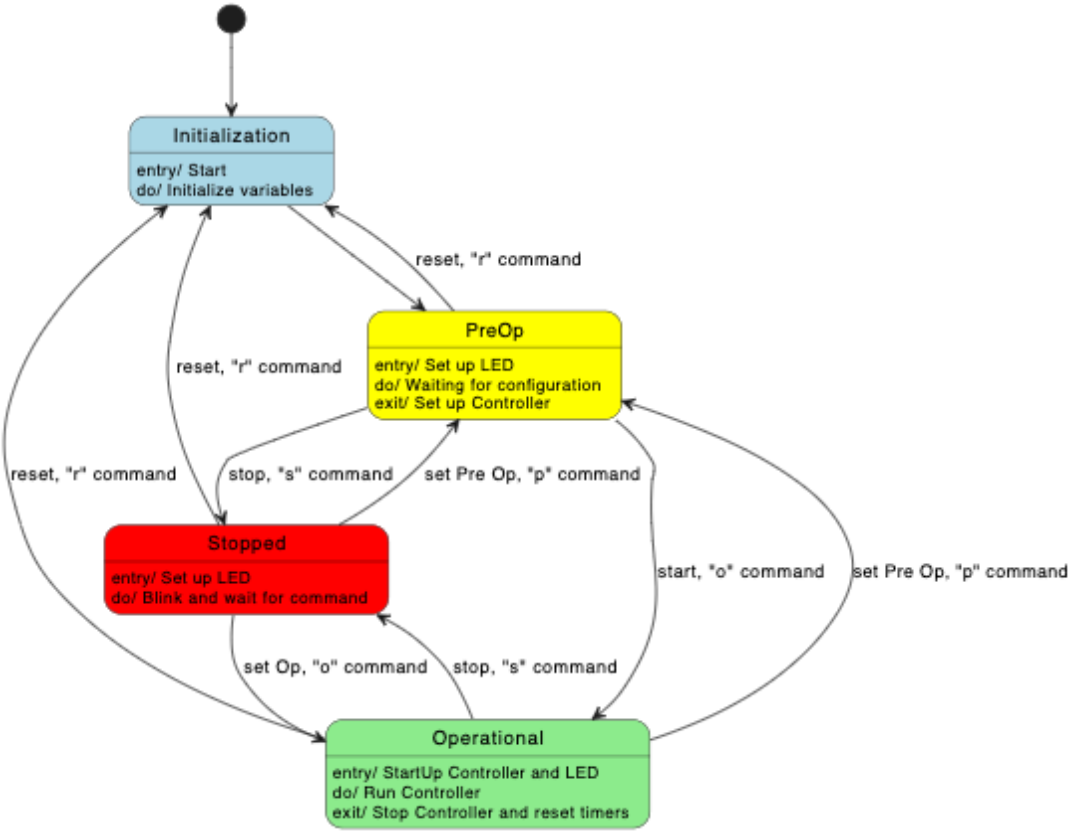


Part 4

We added the Pre-operational state in the following way:

- We toggle the LED at a 1Hz frequency
- We indicate that we enter the pre-op state and we offer two choices for the user: type "b" to have a basic p controller or type "a" to have an advanced pi controller
- We listen to keyboards inputs for the user to set up a value for Kp (and optionally for Ti if the user chose a pi controller)
- We wait for a command for the transition "o" for operational, "r" for reset, and "s" for stopped

We also added to the previous states the necessary transitions to the Pre-operational state
Here is the final UML diagram:



Ziegler-Nichols method to tune the parameters:

By using only the proportional controller, we found out an “ultimate gain” Ku of 0.03 and an associated value of Tu of 2s.
It induces:

Kp = 0.45 * Ku = 0.0135
Ti = 0.83 * Tu = 1.66

We chose to use a sample time T of 0.01s to maintain an accurate stable and quick control.

For the plots, we could not print more frequently than every 1 second because the serial monitor overflood everything.

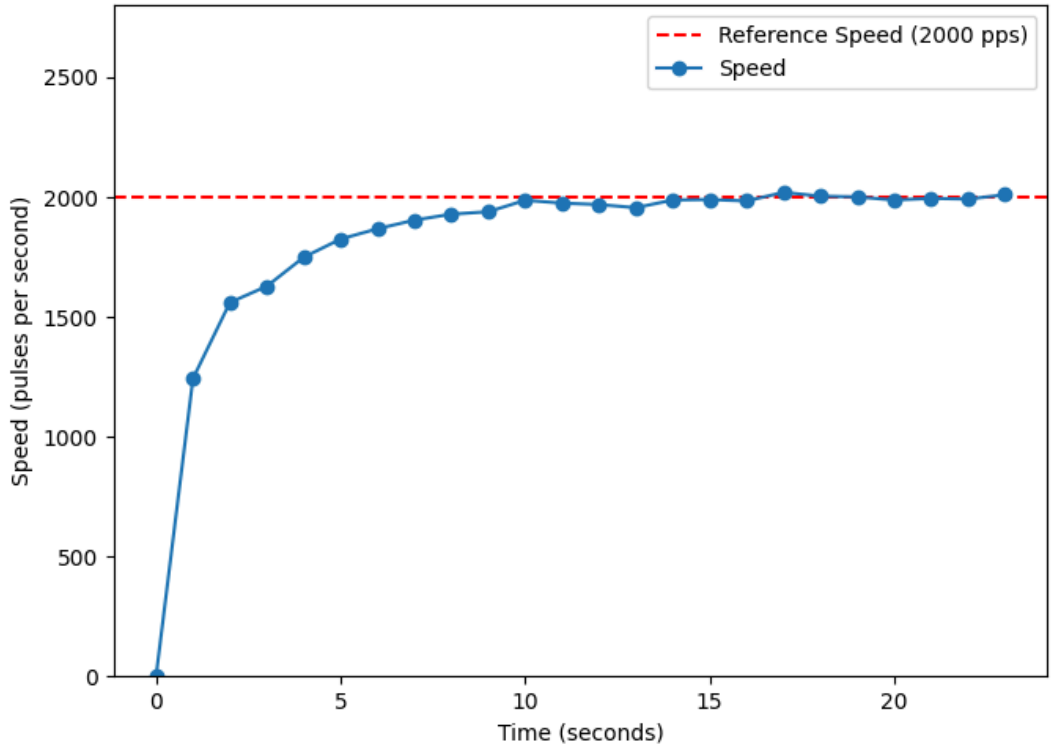


Figure1: **PI** Controller step response **without** friction

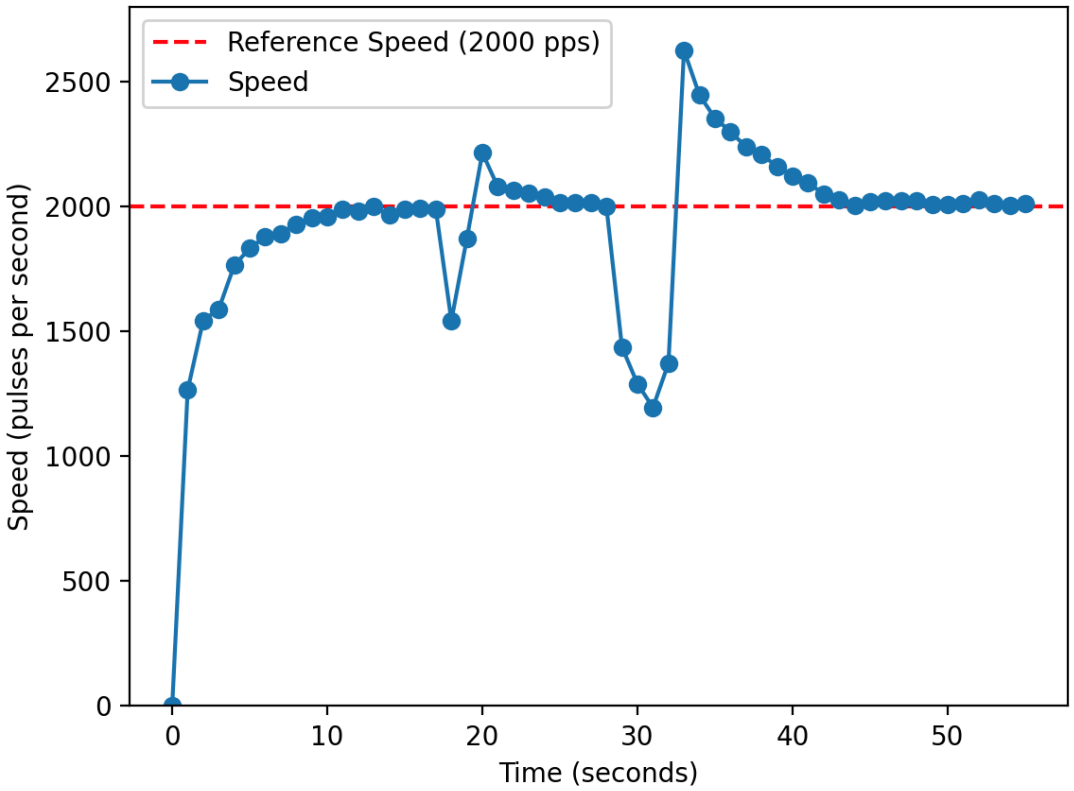


Figure2: **PI** Controller step response **with** frictions

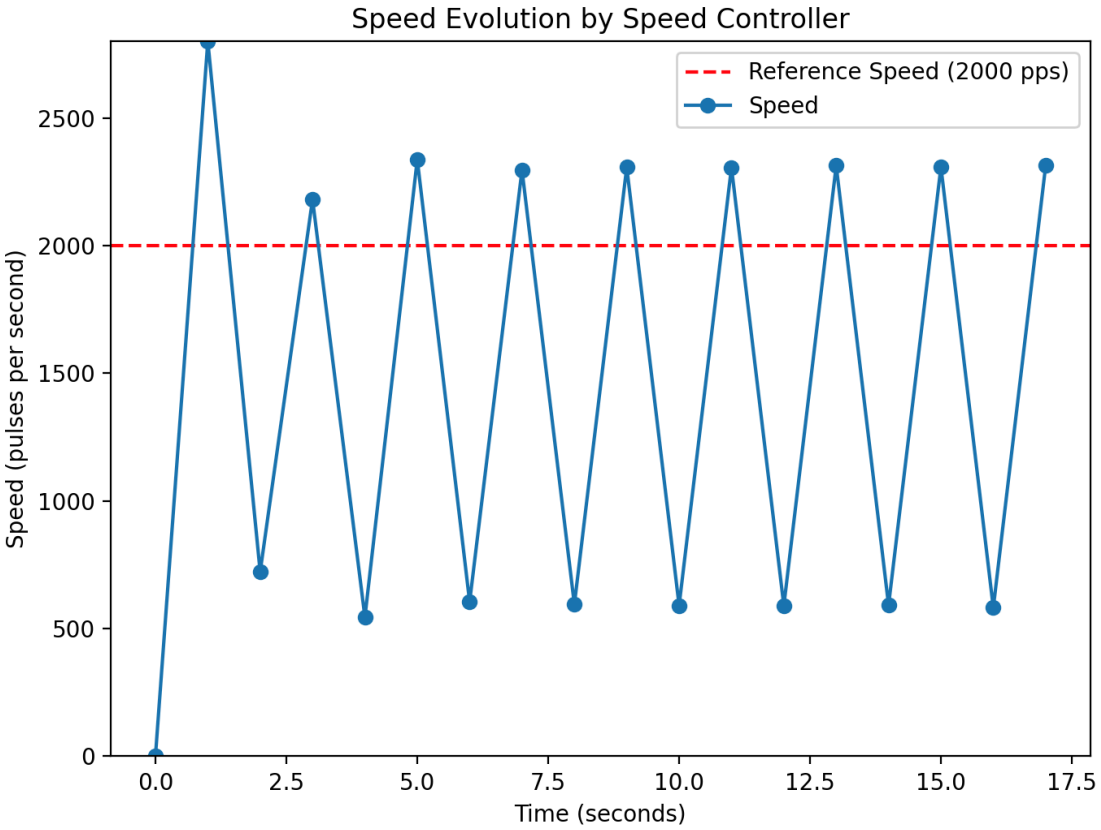


Figure3: **P** Controller step response **without** friction

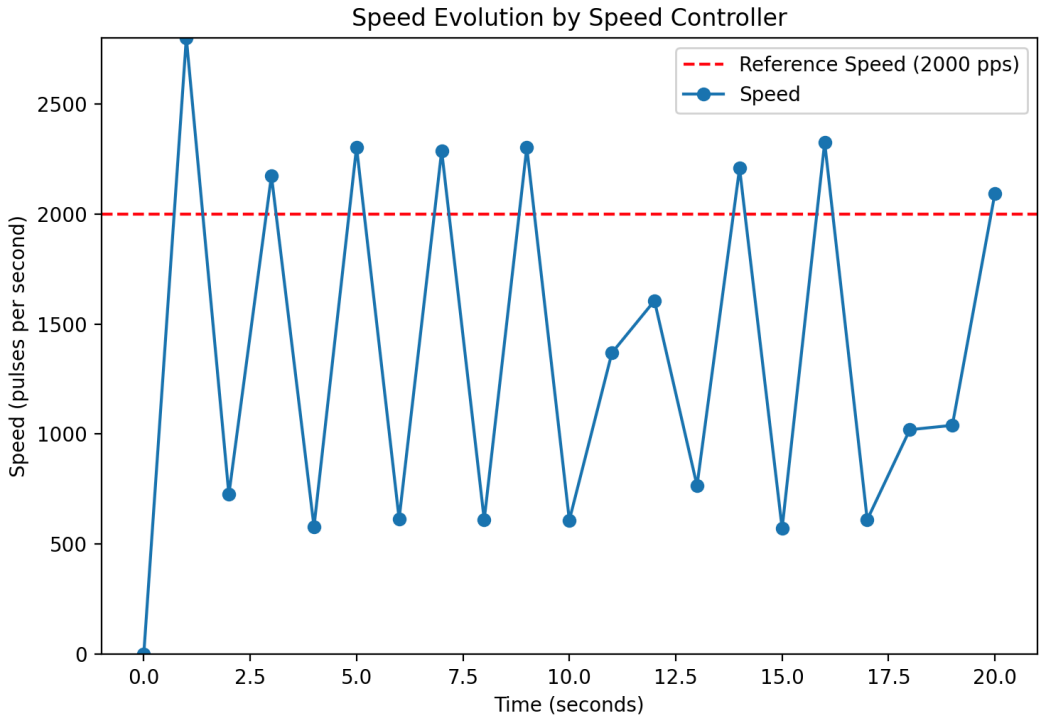


Figure3: **P** Controller step response **with** frictions

As we can see, without the Integration part of the controller we can't reach the desired stability because the error is too big, that is either too positive or too negative. Which results in the motor going full speed in the first case or just stopping in the second case.

Note: We also tried with different reference speeds. The PI controller managed to reach them and stabilize every time when the reference value was above 1000 pps. However when the reference speed is lower the controller has troubles to stabilize because at one given time he has not enough power to turn the wheel so the sum of errors goes up drastically inducing some sort of oscillations around the reference value.

Annex: extract of Project 2 report regarding the timers related to operational state

Task	Timing Requirement	Actual Timing	Method Used	Resource Allocation (Timer/ Interrupts)
Counting Encoder Pulses	Not miss any encoder pulse (done in Project 1)	**280 µs** sampling period	Using **Timer2** to generate an interrupt every **280 µs**	**Timer2 (8-bit)**, Prescaler: **1024**, Interrupt every **280 µs**
Computing Motor Speed	Compute the speed velocity every second	**1 second** (calculated using **125 * 8 ms**)	**Timer0** generates interrupts every **8 ms**, and after **125 counts**, speed is calculated	**Timer0 (8-bit)**, Prescaler: **1024**, Interrupt every **8 ms**
Updating Control Output	Respect the given equation: $\frac{1}{f_{PWM}} \ll \frac{1}{f_{update}} \leq \frac{t}{10}$	**8 ms** (actual **125 Hz** update rate)	**Timer0** interrupt triggers the control law update every **8 ms**	**Timer0 (8-bit)**, Prescaler: **1024**, Interrupt every **8 ms**
Storing/ Transmitting Data	Every second	**1 second** (after each speed calculation)	**Serial.print()** transmits data after speed calculation	**UART** for serial communication to the laptop
Generating Stable PWM Output	Respect the given equation: $\frac{1}{f_{PWM}} \ll \frac{1}{f_{update}} \leq \frac{t}{10}$	**450 µs** PWM period (**2.22 kHz**)	**Timer1** generates PWM at **450 µs** period (duty cycle changes based on control output)	**Timer1 (16-bit)**, Prescaler: **64**, PWM period **450 µs**