Position Control and Gravity Compensation of a Basic Robotic Link

Eli Ferin (eli.ferin@studenti.unipd.it)
Filippo Semenza (filippo.semenza@studenti.unipd.it)
Giulio Rebecchi (giulio.rebecchi@studenti.unipd.it)

23 February 2024

Contents

1 Introduction

In this experience we recreate what is commonly done in the context of robotics: a position controller combined with a gravity compensation mechanism.

The position control part is needed in almost every application to move the robot to a desired location, for example in a pick and place task.

On the other hand, the gravity compensation mechanism is implemented with two purposes:

- Improve the performance of the controller by cancelling the nonlinear torque produced by gravity. In a robotic system the link's weight and geometry is known a priori, and it can be encoded in the control loop to speed up the response. By doing so the system becomes linear and it allows for an easier controller design. This technique is also called *feedback linearization* for this reason.
- Allow an operator to effortlessly move the robot to a desired position. In this case the role of the controller is simply to cancel gravity, so that the robot feels weightless to handle. This is useful for example when the operator wants to "teach" the robot a position without having to define such position analytically through the study of the kinematics of the machine.

In our project we implemented these concepts in a simplified scenario, where the robot is made of a single revolute joint and a straight link. Nonetheless, this experience can be easily scaled to a much more complex system such as a fully-fledged robotic arm.

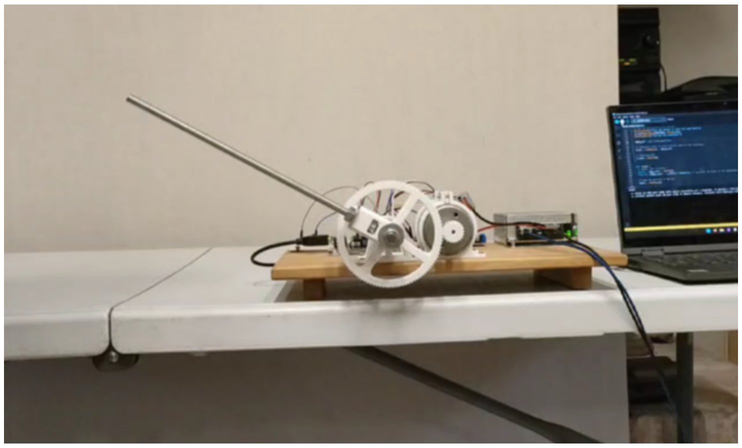


Figure 1: The project during operation.

2 Description of the setup

2.1 Motor

The actuator we chose for our project is a DC motor.

The peculiarity of this motor is the configuration of the coils in the rotor. By disassemblying it we observed that, differently from a typical DC motor, the windings are only 3, and so are the contacts that touch the two brushes of the rotor. With further investigation we observed that the windings formed a delta configuration, as illustrated in Figure 2.

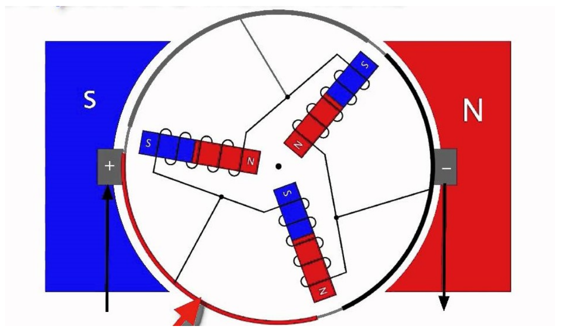


Figure 2: Scheme of a 3 windings DC motor

2.2 Controller

In this project, we used the Arduino Uno (SMD version), a versatile and compact microcontroller. The chip inside is a ATMega328P, a highly reliable and functional solution. It served as the brain of our project, seamlessly interfacing with motor, driver and sensor, through the use of a breadboard and various kinds of jumper cables.

2.3 Driver

The driver we used is the L298N. It's an integrated dual H-bridge motor driver IC to control the motor. It can handle high current and voltage, making it suitable for our purpose. The H-bridge configuration allows the control of both motor direction and speed by adjusting the voltage and polarity. It's connected to the Arduino board, that sends control signals to adjust the motor's behavior according to our desired values.

```
// Take a float dutycycle in [-1,1], and "write" the correct
//configuration of the l298n driver
int setPWM(float dutycycle) {
  // Throw an error if saturation is detected.
  // Saturation must be done outside!
  if (abs(dutycycle) > 1) {
    Serial.println("Saturation detected!");
    setPWM(0);
    exit(0);
  }

  // Set the correct direction
  if ( dutycycle >= 0 ) {
    digitalWrite(INB, 0);
  }
  else {
    digitalWrite(INB, 1);
    dutycycle = 1+dutycycle;  // We are using forward-brake
  }

  // Compute the int value of the pwm in the range [0,255]
  int pwm = map( abs(dutycycle), 0.0, 1.0, 0, 255);
  analogWrite(PWM_PIN, pwm);

  return pwm;
}
```

2.4 Sensor

To control the motor we needed a sensor to measure the position of the rod.

At first we considered a typical full quadrature encoder but we had trouble in mounting it correctly, in particular with the alignment of all the components, so we decided to use a magnetic encoder instead.

A magnetic encoder is made of a set of Hall effect sensor, positioned in a circular array. A diametral magnet (a cilindircal magnet where the magnetic field is oriented along its diameter) is attached to the rotor, and the sensor is positioned in front of it at a distance of about $2\,mm$. When the rotor rotates, the magnetic field rotates aswell. Such rotation is detected by the Hall effect sensors, and by combining their readings the position is detected. The output of the sensors is amplified and converted to a digital signal with an ADC.

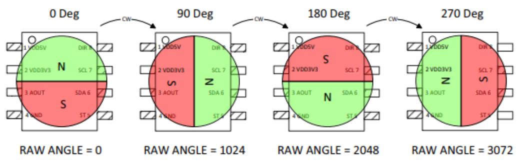


Figure 3: Raw angle measurements example

The advantages of a magnetic encoder over a traditional encoder is that it can be much more precise, does not require contact with the rotor, and does not have any limit on the rotation count. The only drawback we encountered is that the magnet must be aligned correctly with the rotor and the sensor to have a usable reading.

In our application we used an AS5600 magnetic encoder, which requires a 5 V supply (compatible with the Arduino board). This particular sensor provides 4096 pulses per rotation (12 bits), with a sensibility of about 0.1°. The board provides all the necessary components (Hall effect sensors, amplifier, ADC) and the position can be polled using the i2c protocol. For this purpose we used the AS5600 library published by Rob Tillaart.

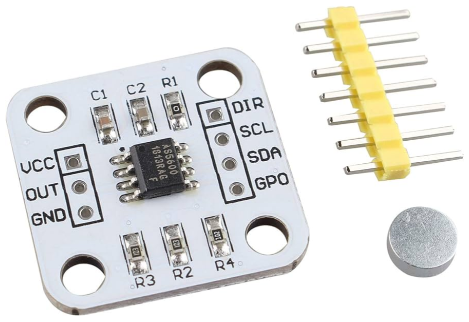


Figure 4: AS5600 encoder

The function we implemented to measure the position returns an angle in the $[-\pi, \pi]$ range, where 0 is the downward position. To get such a range, we had to:

- Measure the raw cumulative position of the motor (this reading can be any real number, up to infinity);
- Calculate the load position by considering the gearbox ratio;
- Calculate the absolute position of the load in the range $[-2\pi, 2\pi]$ by "removing" complete revolutions from the cumulative angle;
- Calculate the absolute position in the range $[-\pi, \pi]$;

```
// Return the angle in radians in a range [-pi,pi],
// where 0 is the downward position
// We are measuring the motor's position, but we want to control
// the load => we need to take into account the gearbox
float readTheta() {
  // Measure the cumulative position of the motor
  motor_position_cumulative = (encoder.getCumulativePosition()
                                * AS5600_RAW_TO_RADIANS);

  // Cumulative position of the load
  theta_cumulative = motor_position_cumulative / GEARBOX_RATIO;

  // Remove complete revolutions from the measurement
  float theta = fmod(theta_cumulative, 2*PI); // in [-2*PI, 2*PI]

  // We want the measurement to be in the [-PI, PI] range
  if (theta > PI) theta -= 2*PI;
  else if (theta < -PI) theta += 2*PI;

  return theta;
}
```

2.4.1 Error Calculation

We observe that when the load is in the upward position, the angle could be either $-\pi$ or π. To avoid problems in the control loop, we took into account this discontinuity in the calculation of the error. Specifically, we impose that the error always remains in the range $[-\pi, \pi]$. By doing so we also ensure that the load always chooses the shortest path (clockwise or counter-clockwise) towards the reference angle, with the change happening once the rod passes the point that sits opposite the reference angle.

```
error = theta_ref - theta;
if (error > PI) error -= 2*PI;
else if (error < -PI) error += 2*PI;
```

2.5 Control Button

To interact with the controller we used a simple push-button, with a pull-down configuration.

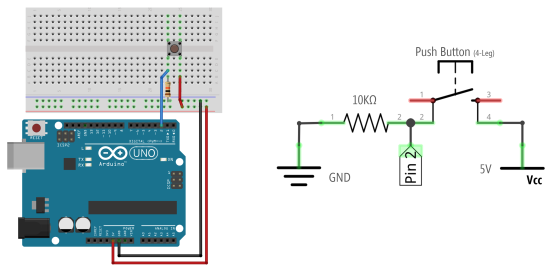


Figure 5: Pull-down configuration for the control button

2.6 Gearbox

Without the gearbox we only managed to use a very light and weak plastic tube as load. To make the project more interesting, we decided to introduce a simple gearbox.

Taking into account the torque capabilities of the motor, we designed the gearbox so it would be able to work with a 25 cm M8 threaded rod. Specifically we wanted the motor to hold it in place at any angle, and at the same time being able to lift an additional weight of about 50 g.

A good compromise between these requirements and the physical dimensions of the gears is a 0.2 gearbox ratio. The gearbox consists of two spur gears, realized with a 3D printer.

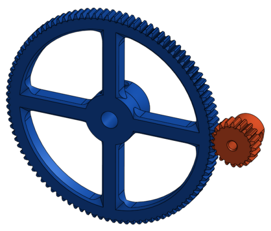


Figure 6: 3D model of the gearbox. The red gear is attached to the motor, the blue gear to the load

2.7 Power Supply

For the power supply we chose a simple AC-DC converter. The output is 11.7 V up to 5 A. Both these specifications are compatible with and sufficient for our motor and driver.

2.8 Threaded rod

The load applied to the motor is a M8 threaded rod, cut at a length of abut $25\,cm$. We chose such load because it is easy to work with thanks to the thread, and its weight provided a sufficient challenge to the motor and the controller.

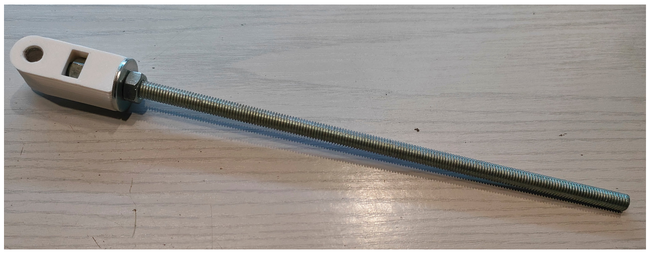
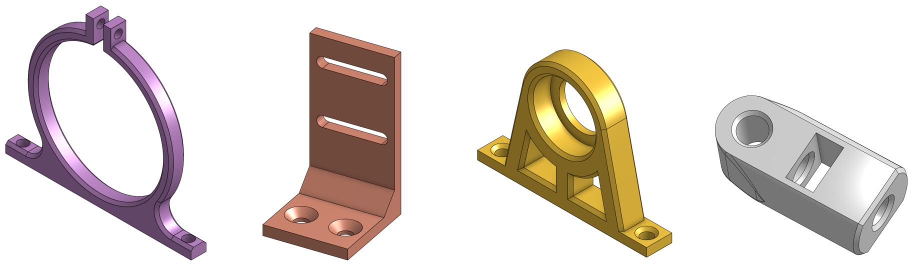


Figure 7: Photo of the rod

2.9 3D Printed Parts

To assemble the project we needed a series of 3D printed parts to position and keep the objects in place. In particular we designed and printed these pieces:

- Two supports for the motor
- A bracket for the encoder
- The gears of the gearbox
- Two supports for the bearings that hold the load shaft
- A connector to attach the rod to the load shaft



(a) Motor support (b) Encoder bracket (c) Bearing support (d) Rod connector

Figure 8: 3D models of the parts

All these parts have been printed with white PLA plastic.

2.10 Other Hardware

To connect and assemble the setup these parts were used:

- Wood board to serve as a base for the project
- Short screw to mount the supports to the board
- M8 Threaded rod used as shaft for the load
- 2 608ZZ Bearings to support the load shaft
- M8 Nuts and washers to mount the load shaft and attach the load connector to the shaft
- M3 Nuts and bolts to attach the gears to the shafts, and to mount the encoder to its bracket

2.11 Assembly

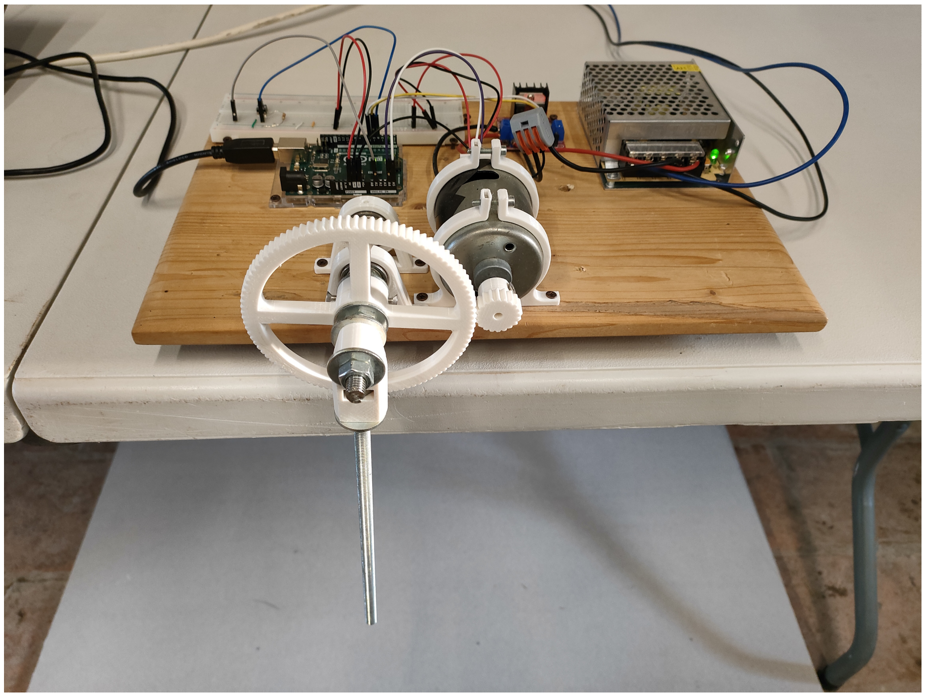


Figure 9: Photo of the assembled setup

3 Measurements

3.1 Resistance

To estimate the armature resistance we blocked the motion of the motor with a 3D printed bracket, applied a known voltage and measured the current. We repeated the experiment for a set of voltages, and then calculated the estimated resistance using a linear regression model

$$V = RI \quad \Rightarrow \quad \frac{1}{\hat{R}} = (V^T V)^{-1} V^T I$$

where V is the vector containing the input voltages and I is the vector containing the output currents.

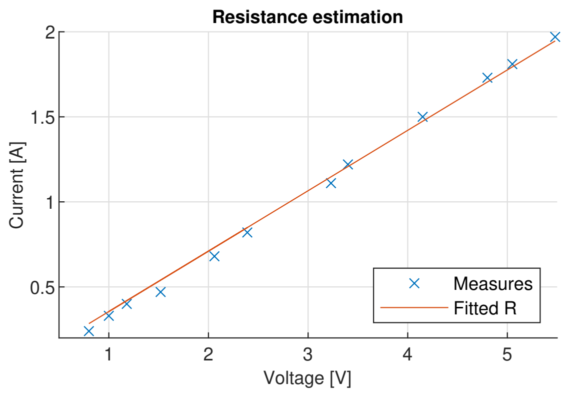


Figure 10: Comparison between measurements and prediction with the estimated resistance

3.2 Motor Constant

To estimate the motor constant we performed a similar experiment as the one for the resistance, but this time we let the motor spin and we measured the rotor velocity aswell.

From the model of the motor we know that, at steady-state, the following equation holds

$$k\Phi = \frac{V - RI}{\Omega_m}$$

Using the estimated resistance $\hat{R}$ we calculated before, we estimated the motor constant using a linear regression model, where the regressor is defined as $X = V - \hat{R}I$:

$$\frac{1}{\hat{k\Phi}} = (X^T X)^{-1} X^T \Omega$$

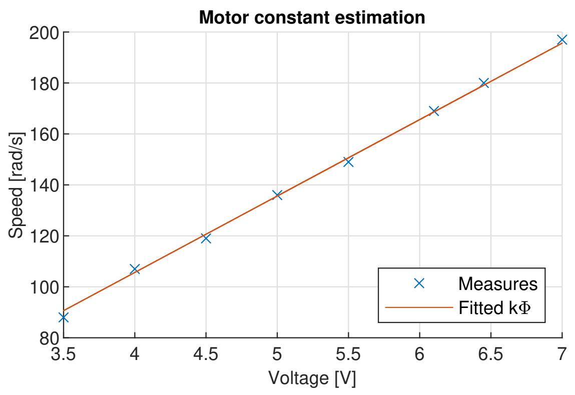


Figure 11: Comparison between measurements and prediction with the estimated motor constant

3.3 Friction

To estimate the static friction of our system, we prepared the setup with the gearbox in place, but without attaching the load to the shaft. Then, starting from $0\,V$ we increased the voltage applied to the motor until it started moving.

We observed that the threshold voltage required to overcome static friction is at about $2.45\,V$, corresponding to a dutycycle $\delta = 0.21$.

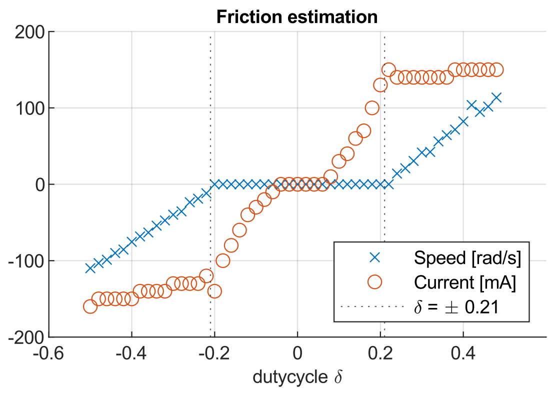


Figure 12: Measurements of speed and current for various values of δ

4 Control

As described earlier, the aim of the project was that of achieving positional set-point control with disturbance rejection of the rod attached to our DC motor, including and highlighting in the control strategy a feedforward gravity compensation term. This gravity term allows the user to activate an alternative control mode, where the rod no longer tries to remain at its previous set-point, but instead can be effortlessly moved to a new position by hand as if it were weightless, without falling due to gravity.

The control was performed with an Arduino controller by using its loop function as the control loop, and thus in a discrete time manner with a chosen sampling time of $T_s = 5000\ \mu s$ (this value was selected by experimenting with a time-out function).
The overall discrete time positional control action is composed by the sum of:

- A PID Controller operating as the main Feedback loop on the positional error given by the encoder;
- A Gravity Compensation term;
- A Friction Compensation term;

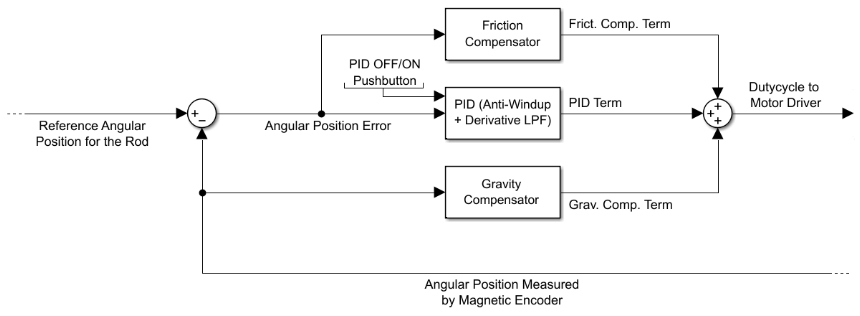

Friction Compensator
Frict. Comp. Term
PID OFF/ON Pushbutton
PID (Anti-Windup + Derivative LPF)
PID Term
Dutycycle to Motor Driver
Reference Angular Position for the Rod
Angular Position Error
Gravity Compensator
Grav. Comp. Term
Angular Position Measured by Magnetic Encoder

Figure 13: Controller diagram with all the components.

4.1 PID Controller

We chose a PID Controller because of its ease of use and flexibility. The presence of the integrator allows the the system to not only track reference angle set-points with no steady-state error, but also be able to reject constant or impulse disturbances such as an added load hanged on the end of the rod, or a hand hitting and moving the rod away from its reference.
The PID has been implemented in a discrete time fashion (sampling time T_s) by using the following calculations methods for each of its three terms in the Arduino loop:

- The Proportional term P at time t is simply based on the current angular error θ_e measured at that same time, so that

$$P(t) = K_P \, \theta_e(t)$$

- The Integral term I at time t is calculated by first computing the integral of the error up to time t with the backward method and then using it, so:

$$\int_0^t \theta_e(r)\, dr = \int_0^{t-1} \theta_e(r)\, dr + T_s \cdot \theta_e(r) \qquad I(t) = K_I \int_0^t \theta_e(r)\, dr$$

- The Derivative term D at time t is calculated by first computing the derivative of the error with the backward method and then using it, so:

$$\dot{\theta}_e(t) = \frac{\theta_e(t) - \theta_e(t-1)}{T_s} \qquad D(t) = K_D \, \dot{\theta}_e(t)$$

  Note that this update to the Derivative term only happens if the difference between the two errors at the numerator is less than π.
  This is done to avoid problems with the error discontinuity introduced by the mechanism that makes the rod follow the shortest roation sense to its reference, as previously described in the Subsection 2.4.1.

The sum of the three PID components constitutes the base of the control action, to which the resulting outputs from the Gravity and Friction Compensation calculations (shown in detail later on) will be added to give the final dutycycle δ, used by the Arduino to send the correct PWM command to the driver, as shown in Subsection 2.3.

The dutycyle δ is a real number in the interval of $[-1, +1]$, with a safety measure to simply saturate the value inside these limits should it surpass them.

Since the Gearbox has an even number of gears (two), it is of essential importance to invert the final duty cycle before sending it to the driver function, to account for the inverted sense of rotation of the rod with respect to the motor and all the motor side measurements we gathered and used for the calculations.

4.1.1 Ziegler-Nichols PID Tuning Method

The estimated model of the motor turned out to be quite inaccurate, due to the non-negligible static friction and other non-linearities of our drive. This means that we couldn't easily and reliably obtain a satisfactory controller using theoretical methods starting from such model.

To overcome this problem, we used an experimental process described by Ziegler and Nichols [Optimum Settings for Automatic Controllers]. Assuming that the PID controller is already implemented, we can find good values for the proportional, integral and derivative gains by observing the behaviour of the closed loop system. The procedure is the following:

1. Set all the gains to 0.
2. Increase the proportional gain K_P until the system reacts to a small perturbation with a stable and consistent oscillation. This is the critical point of the system: if we increase the gain, then it becomes unstable.
3. Set the *Ultimate Gain* $K_u = K_P$, and the *Ultimate Period* T_u equal the period of the oscillation.
4. Compute the empirical PID gains from K_u and T_u.

There are multiple ways to compute the gains, based on the desired dynamics of the closed loop system. We chose the configuration that gives a minimal overshoot, which is:

$$K_P = 0.2\,K_u \qquad K_I = 0.4\,\frac{K_u}{T_u} \qquad K_D = 0.066\,K_u\,T_u$$

In our case the explained tuning method was applied after having already set the Gravity and Friction Compensation Terms (described later), obtaining the two values:

$$K_u = 10 \qquad T_u = 0.220\,s$$

4.1.2 Anti-Windup on the Integral Term

In order to reduce overshoot, we introduced an anti-windup mechanism. The goal of the anti-windup is to prevent the integral of the error from growing too much when the control signal is saturating. Without it, during the rising edge (for example in a step response) the integral increases and causes a large and long overshoot. In our application, the saturation threshold is met when the dutycycle is set to and absolute value of 1. Since we used a quite large integral gain, without the anti-windup the system becomes unstable.

We also observed that the overshoot is further reduced if we enable the integral term only in a neighbourhood of the reference angle. We chose an interval of about 30° on either side of the reference.

```
// Integral with anti-windup
if (abs(dutycycle) <= 0.99 && abs(error) < 0.5) {
  error_integral += error * (Ts/1000000.0);
}
```

4.1.3 Low-Pass Filter on the Derivative Term

The introduction of a low-pass filter for the derivative term is necessary to mitigate the effects of noise in the measurements. By implementing this filter we managed to reduce significantly large variations in the derivative, leading to a much smoother control action.

The low-pass filter consists of a weighted average of the previous filtered value of the error derivative and the new computed error derivative. The smoothing factor r determines the weights. When $r = 1$ the filter is disabled, and if we decrease it, the smoothing action is stronger. Note that as we decrease r, we introduce a delay in the derivative term of the PID.

$$\dot{\theta}_{e\,filt}^{(t)} = (1 - r)\,\dot{\theta}_{e\,filt}^{(t-1)} + r\,\dot{\theta}_{e}^{(t)}$$

```
// Low pass filter
float lowPass(float x, float yPrev, float r) {
  float y = (r-1)*yPrev + r*x;
  return y;
}
```

4.2 Gravity Compensation

Gravity compensation is a technique that is used to cancel the nonlinear term produced by gravity in a mechanical system.

In our case, this term is the torque produced by the weight of the rod. Since the rod acts as a physical pendulum, the torque it generates is proportional to its weight and its position.

Let m be the mass of the rod, and l its length. Assuming the mass is evenly distributed, and attaching the rod to the shaft on one of its extremities, we have that the torque produced is

$$\tau_G(\theta) = -mgd\sin(\theta)$$

where g is the gravity acceleration, d is the distance of the baricenter from the rotation axis and θ is the angle displacement from the downward position.

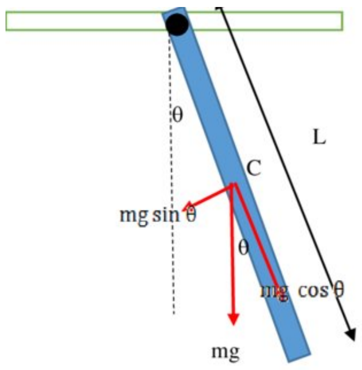


Figure 14: Drawing of gravity acting on a physical pendulum

At motor side, this torque is multiplied by the gearbox ratio

$$\tau_{G,m}(\theta) = -mgd\sin(\theta)\,k_g$$

Note that, for simiplicity, in our program we combined the parameters into a single variable `gravity_comp_constant`. Such variable has been hand-tuned starting from the theoretical value calculated on paper.

In order to cancel the gravity torque it is necessary to measure the position of the rod and to calculate the voltage to apply to the motor to mantain equilibrium. From the dynamical model of the DC motor we know that the transfer function from torque to voltage is (ignoring the electrical dynamics)

$$\tau(s) = \frac{k\Phi}{R}V(s) - \frac{k\Phi^2}{R}\Omega_m(s)$$

If we assume the position to be constant, this relation is simplified to the static equation

$$\tau(s) = \frac{k\Phi}{R}V(s) \quad \Rightarrow \quad V_G(\theta) = \frac{R}{k\Phi}\tau_{G,m}(\theta)$$

```
// Constant that describes the maximum load torque
float gravity_comp_constant = 0.0047;

float gravityCompensation(float theta) {
  float V_gc = sin(theta) * gravity_comp_constant * R/kphi;
  return V_gc / V_MOT_MAX;
}
```

The gravity compensation mechanism has two consequences:

- It mantains the system in equilibrium even without a controller (assuming no disturbance is applied and perfect knowledge of the parameters);
- It makes the system linear by canceling the only non-linear term in the dynamic equation (ignoring static friction).

4.3 Friction Compensation

While testing the performance of the motor, we observed that the static friction of the rotor and the gearbox are not negligible, especially since we want to perform position control.

As we observed in Section 3.3 the motor did not move until the dutycycle reached the value $\delta = 0.21$. After that, the speed of the motor increased linearly with the voltage.

To compensate it, we implemented a feedforward action. The idea is to apply immediately the threshold voltage. By doing so, the motor reacts as expected to any additional voltage (for example the control voltage), resulting in a significantly quicker response. Since we need to move the motor in a direction that is dependent on the reference position, we initially set the feedforward voltage to the threshold multiplied by the sign of the error.

The problem with that implementation is that when the error is very small and is in a neighbourhood of 0, it changes sign very often, resulting in a very noisy control signal (when the error changes its sign we have an instant variation of about $4.7\,V$) and a lot of vibrations. So, we decided to set the friction compensation voltage as proportional to the position error, until it reaches the threshold and saturates (Figure 15). This resulted in a much smoother response, with no vibrations.

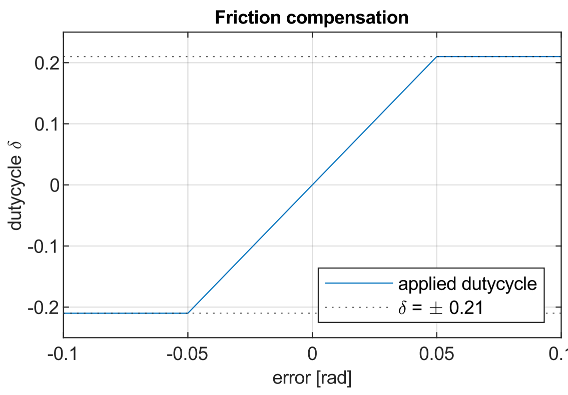


Figure 15: Friction compensation as a function of the error

```
// Voltage required to overcome static friction
float friction_dutycycle = 0.21;
// Error at which the compensation saturates
float error_threshold = 0.05;

float frictionCompensation(float error) {
  float dutycycle = 0;

  if ( abs(error) > error_threshold) {
    dutycycle = sign(error) * friction_dutycycle;
  }
  else {
    dutycycle = error / error_threshold * friction_dutycycle;
  }

  return dutycycle;
}
```

4.4 Button Interactions

The control button serves two purposes:

- After the setup, the program waits for the button to be pressed once to activate the control loop. This is useful to collect data and to observe particular behaviours of the system when reaching a given set-point.

```
// Wait for the button to be pressed
Serial.println("Press the button to start the experiment");
while(!digitalRead(BUTTON)); delay(1000);
Serial.println("Starting the experiment");
```

- The other role of the button is to disable the PID controller and keep active only the gravity compensation term. This allows the user to move the rod and change the reference angle by keeping the button pressed. When the button is released, the PID is enabled once again and the controller keeps the rod where the user last left it.

```
// If the button is pressed, refresh the reference to be the
// current position
// This basically disables the controller
if (digitalRead(BUTTON)) {
    theta_ref = theta;
    error_integral = 0;
}
```

5 Challenges

5.1 Problems with the 3-Coils Configuration of the Rotor

In the balancing motor, we encountered challenges associated with the 3-coils configuration of the rotor. One prominent issue was the misalignment of the coils, leading to a ”jump” while moving the rod. This discrepancy has been catalogued as a non-linearity of the motor.

5.2 Asymmetric Behaviour of the Motor

The motor is able to generate a smooth and stable torque when rotating clockwise, but it stutters and is inconsistent when rotating counter-clockwise. We tried to investigate on the control loop, on the driver wiring, on the power supply, on the soldered cables, the brushes and other motor internals.

Our conclusion is that the brushes are slightly asymmetrical, and the contact is not perfect when the motor rotates counter-clockwise, causing inconsistent electrical connection with the rotor. Nevertheless, the behaviour is too inconsistent to have a precise analysis of the problem.

5.3 Gravity Compensation

In an ideal situation, the gravity compensation should provide the exact torque required to cancel the weight of the load. In our project, the combination of friction, motor asymmetry and the fact that the rotor has only 3 coils resulted in a gravity compensation that is too strong in some positions, and too weak in others. The gravity compensation constant we used in the program is the one that minimizes this effect.

5.4 Friction

As we described in Section 4.3, static friction is not negligible in our application. Even though we managed to mitigate it with a feedforward component, such compensation is not perfect and friction is still problematic, especially when the measured position is slightly different from the reference.

6 Results and Conclusions

6.1 Position Control

The motor incorporates a robust position control system, ensuring its ability to autonomously return to the reference position despite external disturbances. This feature highlights the motor's resilience and self-balancing capability in maintaining precise positioning.

6.1.1 Step Response

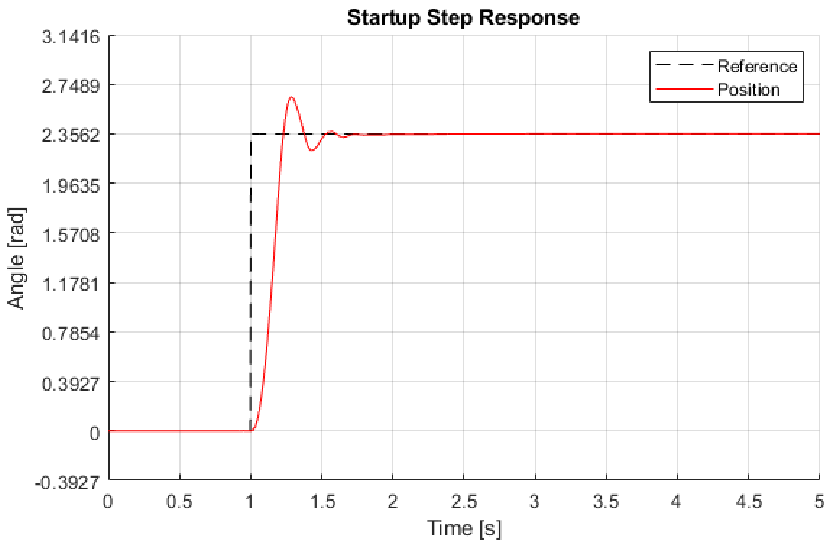


Figure 16: Step response of the controller at startup, reaching an angular position reference of $\theta_{ref} = \pi\frac{3}{4} \approx 2.3562\,rad$ from the downward rest position. Note the quick rise time, the relatively contained overshoot, the rapid settling time, and the perfect asymptotic tracking.

6.2 PID Controller Off

The 'PID Controller Off' functionality provides users with the flexibility to manually reposition the bar when necessary. By pressing and holding the designated button, the motor's balancing capabilities are temporarily deactivated. This allows users to freely move the bar to a desired position without interference from the motor's position control. This feature proves especially useful in situations where manual adjustments or positioning are required.

Upon releasing the button, the motor seamlessly resumes its self-balancing functionality, and if moved again will automatically return the system to the new reference position, which is where it was last left at the button release.

6.3 Robustness

PID Performance

The use of a well-tuned Proportional-Integral-Derivative (PID) controller played a pivotal role in enhancing system stability and mitigating errors. Its contribution ensures smooth and efficient control, allowing the motorized system to adapt seamlessly to different load conditions and disturbances.

Perfect Tracking

The precision of the control system is enhanced by the achievement of perfect tracking, consequence of the use of an Integrator. The motor shows high accuracy, proving the effectiveness of our control algorithms.

The motor can hold additional weight

The system is able to handle also additional weight, while mantaining its balancing functionality. In fact, whether faced with unexpected payloads or intentional variations in load, the system adapts seamlessly, maintaining stability and precise control. This feature not only extends the practical applications of the system, but also demonstrate its reliability in different situations.

7 Code

```
#include "AS5600.h"
#include "Wire.h"
#include "math.h"

// Arduino UNO i2c pins: SCL:A5 SDA:A4
// Remeber to connect the DIR pin of the encoder to 5V to set
// the direction to counter-clockwise
// If using the gearbox, the motor spins in the opposite
// direction wrt the load => connect DIR to GND
AS5600 encoder;
float motor_position, motor_position_cumulative;
float theta, theta_cumulative;

// L298N driver
#define INA 5
#define INB 6
#define EN 3
int PWM_PIN = INA;

// Control button
#define BUTTON 4

// Time variables [us]
uint32_t Ts = 5000;   // sampling time
uint32_t t_start = 0; // start time of the experiment

// Power supply parameters
#define V_PSU 11.7 // Before the driver [V]
#define DRIVER_DROP 1.8  // Voltage drop of the driver [V]
const float V_MOT_MAX = V_PSU - DRIVER_DROP;

// Motor parameters
#define R 2.8   // [Ohm]
#define kphi 0.0033 // [Vs]

// Gearbox parameters
```

```
#define GEARBOX_RATIO 5.0 // The load spins 5 times slower

// Gravity compensation
float gravity_comp_constant = 0.0047;

// Given the angle compute the dutycycle that compensate
// the gravity torque
float gravityCompensation(float theta) {
  // Voltage [V]
  float V_gc = sin(theta) * gravity_comp_constant * R/kphi;
  // dutycycle in range [0,1]
  return V_gc / V_MOT_MAX;
}

// Controller variables
float error = 0;
float error_integral = 0;
float previous_error = 0;
float error_derivative = 0;
float error_derivative_previous = 0;
float theta_ref = 0;
float step_amplitude = PI*2/4;
float dutycycle = 0;
int pwm = 0;

// Ziegler-Nichols ultimate gain and period
float ku = 10;
float Pu = 0.220;

// PID gains
float kp = 0.2 * ku;
float ki = 0.4 * ku / Pu;
float kd = 0.066 * ku * Pu;

// Overload of map() that uses floats
float map(float x, float in_min, float in_max, float out_min,
          float out_max) {
  return (x - in_min) * (out_max - out_min) / (in_max - in_min)
```

```
          + out_min;
}

// Take a float dutycycle in [-1,1], and "write" the correct
// configuration of the l298n driver
int setPWM(float dutycycle) {
  // Throw an error if saturation is detected. The saturation
  // must be done outside!
  if (abs(dutycycle) > 1) {
    Serial.println("Saturation detected!");
    setPWM(0);
    exit(0);
  }

  // Set the correct direction
  if ( dutycycle >= 0 ) {
    digitalWrite(INB, 0);
  }
  else {
    digitalWrite(INB, 1);
    dutycycle = 1+dutycycle;  // We are in forward-brake!
  }

  // Compute the int value of the pwm in the range [0,255]
  int pwm = map( abs(dutycycle), 0.0, 1.0, 0, 255);
  analogWrite(PWM_PIN, pwm);

  return pwm;
}

// Return the angle in radians in a range [-pi,pi], where 0 is
// the downward position
// We are measuring the motor's position, but we want to
// control the load => we need to take into account the gearbox
float readTheta() {
  // Measure the cumulative position of the motor
  motor_position_cumulative = (encoder.getCumulativePosition()
                              * AS5600_RAW_TO_RADIANS);
  // Cumulative position of the load
  theta_cumulative = motor_position_cumulative / GEARBOX_RATIO;

  // Remove complete revolutions from the measurement
  float theta = fmod(theta_cumulative, 2*PI); // in [-2*PI, 2*PI]

  // We want the measurement to be in the [-PI, PI] range
  if (theta > PI) theta -= 2*PI;
  else if (theta < -PI) theta += 2*PI;

  return theta;
}

// Return the sign of a number
float sign(float x) {
  return x / abs(x);
}

// Low pass filter
// r is the smoothing factor
float lowPass(float x, float yPrev, float r) {
  float y = (r-1)*yPrev + r*x;
  return y;
}

// Given the error, compute the friction compensation term
// If we use a simple sign(error) * friction_dutycycle, the
// input oscillates too much and the br oscillates a lot,
// so we need to smooth it
float friction_dutycycle = 0.21; // Voltage required to overcome
// static friction
float error_threshold = 0.05;  // [rad]

float frictionCompensation(float error) {
  float dutycycle = 0;

  if ( abs(error) > error_threshold) {
    dutycycle = sign(error) * friction_dutycycle;
  }
  else {
    dutycycle = error / error_threshold * friction_dutycycle;
  }

  return dutycycle;
}

//===============================================================
void setup() {
  // Driver setup
  pinMode(INA, OUTPUT);
  pinMode(INB, OUTPUT);
  pinMode(EN, OUTPUT);
  digitalWrite(INB, 0);
  digitalWrite(INA, 0);
  digitalWrite(EN, HIGH);

  // Control button setup
  pinMode(BUTTON, INPUT);

  // Set the baudrate of the serial connection
  Serial.begin(250000);
  Serial.println("\n================\nProgram started");

  // Encoder setup
  Wire.begin();
  Wire.setClock(800000);
  encoder.begin(2);
  Serial.print("AS5600 connect: ");
  Serial.println(encoder.isConnected());

    // Set the offset for the encoder. The rod should
    // be oriented downward!
    encoder.resetCumulativePosition();

  // Wait for the button to be pressed
  Serial.println("Press the button to start the experiment");
  while(!digitalRead(BUTTON)); delay(1000);
  Serial.println("Starting the experiment");

  // Initialize the reference
  theta_ref = step_amplitude;

  // Initialize error to avoid initial spike in the derivative
  error = readTheta() - theta_ref;

  // Starting time
  t_start = micros();
}

void loop() {
  // Update time variables
  uint32_t loop_start = micros();

  // Time past the start of the experiment, in seconds
  float t = (loop_start - t_start) / 1000000.0;

  // Read the position of the bar
  theta = readTheta();

  // If the button is pressed, refresh the reference to be the
  // current position
  // This basically disables the controller
  if (digitalRead(BUTTON)) {
    theta_ref = theta;
    error_integral = 0;
  }

  // PID controller

    // Save the previous value of the error
    previous_error = error;

    // Compute the error
    error = theta_ref - theta;
    if (error > PI) error -= 2*PI;
    else if (error < -PI) error += 2*PI;

    // Integral with anti-windup
    if (abs(dutycycle) <= 0.99 && abs(error) < 0.5) {
      error_integral += error * (Ts/1000000.0);
    }

    // Compute the discrete derivative of the error
    float error_delta = error - previous_error;
    if ( abs(error_delta) < PI )
       error_derivative = (error_delta) / (Ts/1000000.0);
    // Low pass filter applied to derivative
    error_derivative =
       lowPass(error_derivative, error_derivative_previous, 0.8);
    error_derivative_previous = error_derivative;

    // Compute the control dutycycle
    dutycycle = kp*error + ki*error_integral
                    + kd*error_derivative;

  // Gravity compensation
  float g_comp = gravityCompensation(theta);
  dutycycle += g_comp;

  // Static friction compensation
  float f_comp = frictionCompensation(error);
  dutycycle += f_comp;

  // dutycycle saturation correction
  if (dutycycle > 1) dutycycle = 1;
  if (dutycycle < -1) dutycycle = -1;

  // Change direction because we have the gearbox
  // (positive dutycycle -> positive torque)
  dutycycle *= -1;
```

```
  // Set the pwm
  pwm = setPWM(dutycycle);

  // Wait the sampling time
  while( micros()-loop_start < Ts );

}
```