POLITECNICO DI BARI

Dipartimento di Ingegneria Elettrica e dell'Informazione

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

Robotics 1

Prof. Paolo Lino

Report:

Modeling and control of a two degree of freedom manipulator arm

Studenti:
Angelo Mauro De Pinto
Vincenzo Zinfollino

ANNO ACCADEMICO 2022-2023

Contents

Abstract

Robots have played an important role in the automation of processes over these few past decades. In this report we analyze the phases of building a 2 degree of freedom (DoFs) robot from scratch. Starting from the model of the plant to the derivation of the control laws and discussing the choices we have made to optimize the processes. The challenge was to use as much as possible off the shelf products to keep the price the lowest possible without degrading the performance.

1 Introduction

Today, robots can be found in many different fields, from toys to manufacturing processes or in medicine application. The design and implementation of a robot manipulator is a challenging project because it involves several different disciplines. Every robot needs a physical platform that is purely mechanical, an actuation part that is composed by electrical motors and their drivers and a software part which can be itself divided into a low level part (where the firmware belongs) which implements the control laws and a high level part that implements the trajectory planning. The objective of this project is the modeling and control of 2 DoFs manipulator robot arm that can be capable of drawing on a white board a series of geometric primitives. The drawing trajectory are sent to the robot using a trajectory planner software that runs on a PC. The robot was originally modelled using Matlab [1] and Simulink [2], after that the manipulator was designed and realized using 3D printing and CNC machined aluminium. The control system is implemented on a STM32 nucleo board that communicates with a PC. The communication between the PC and the microcontroller board was realized using the serial port. The Trajectory planner software generate the setpoints for the manipulator.

The requirements that this project needs to satisfy are:

- *Kinematic and Dynamic modelling and simulation of the manipulator :* The first step of this project is to derive the physical properties and the relationships between them. Using the Denavit-Hartemberg rules and Lagrangian method, the manipulator's kinematic model and dynamics are to be derived on top of which the whole system can be built. Afterwards the accuracy and feasability of the relationships that were found are to be verified: the simplest method is using the Matlab and Simulink software to simulate the manipulator's behavior.

- *Realization of a trajectory planner software:* the trajectory planner software is a key element given that the user needs to interact with the manipulator. A simple and effective GUI is fundamental for a better user experience. It follows that this software has to implement the trajectory algorithm and has to handle the communication with the microcontroller, too.

- *Realization of a trajectory generation algorithm:* When the user specifies the trajectory in the operational space, using a series of primitives, another piece of software is needed to translate the operational space trajectory into a joint space trajectory. This algorithm generates the operational space setpoints.

- *Realization of a communication protocol:* The control software that runs on the micro controller board and the trajectory planner software need to communicate with each other to exchange setpoints and commands.

- *Mechanical design:* The mechanical structure needs to be as simple as possible and modular as well. This constraints improve the repeatability of the project and are necessary for develop a "future-proof" system, making it easier to maintain.
- Implementation of the control laws: to ensure the correct movement we need to implement a position controller. In this case we have chosen to use joint space control instead of the operational space one, we can implement a different PID controller for every joint.

2 Control Scheme

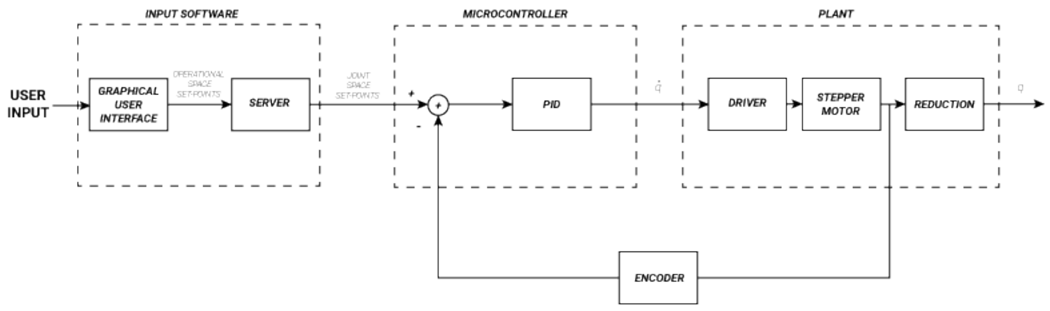


Figure 1: This scheme explains the control loop.

Given the complexity of the project, the control scheme is made up of many heterogeneous elements. To make the control system work properly, some inputs have to be defined. To ensure that the user can simply describes the trajectories that the manipulator will have to execute, a software was created which represents the HMI. With the help of this software the user can specify in a simple way, through primitives, the trajectories in the operating space that the manipulator will have to execute. Once the trajectory in the operational space has been acquired, it must be subdivided to determine the trajectory in the joint space. The subdivision into points is performed by the HMI back-end server, to which the primitives drawn on the GUI of the HMI are passed. At this point of the control scheme, each primitive is divided into points according to a cycloidal timing law. After the subdivision the inverse kinematic algorithm is applied to determine the set points in the joint space to be supplied to the control system. Once the points have been determined they are sent thought the serial port from the host PC to the micro-controller board. When the point are acquired by the board, a callback is used to process the incoming data, afterwards data is processed and stored in a circular buffer, to be then used in the main loop to actuate the driver and then the stepper motor. To guarantee the correct motion of the system, a closed loop control must be used. The chosen control system is a decentralized joint space control, this is achieved using two separate PID controllers. The PID controllers were synthesized starting directly from the frequency domain form, using the Tustin transformation. To close the position loop a magnetic encoder was used to read the position of the motor shaft, this position was corrected multiplying with the correct gear factor, to estimate the correct joint position. In the main loop the control action was calculated in the PID method and than applied, the output of the PID is a velocity value that is converted in a PWM value used to drive the motor driver. During the motion, a timer is used to acquire at a given time the readings of the position: this ensure the correct estimation of the velocity and acceleration.

In the following pages, each component of the project will be described in detail.

3 Plant Model

The two-arm planar manipulator is a type of industrial robot characterized by two arms that move within an horizontal plane. The manipulator has two degrees of freedom as it is designed to operate in a two-dimensional environment. Each arm has a rotary joint that allows it to rotate in space and also connects the two arms together. Each arm is driven by a motor that provides torque, enabling motion at the rotary joints. However, each arm can move independently of the other or synchronize to perform coordinated movements, allowing it to cover a wide workspace area and reach precise positions.

At the end of the arm, there is the end-effector, which is the tool or device used to perform a specific operation or task. In this project, it will be a pen, which, through the movements of the arms, will draw on a whiteboard (Fig 2).

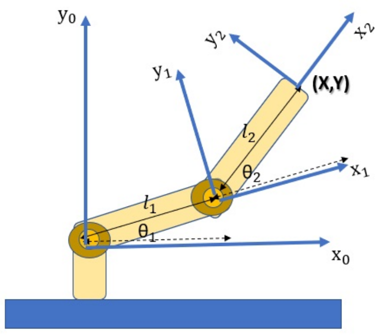


Figure 2: Schematic representation of a two-arm planar manipulator.

3.1 Kinematic and Dynamic modelling

First of all, determining the Denavit-Hartemberg parameters is the first step to derive the manipulator's kinematic. The coordinate systems used is shown in Fig.2 . The DH parameters are:

Joint	a_i [m]	α_i [rad]	$d_i[m]$	$\theta_i[rad]$
1	$l_1 = 0.17$	0	0	θ_1
2	$l_2 = 0.15$	0	0	θ_2

The direct kinematic equation is derived by applying the DH rules is:

$$T(q) = \begin{bmatrix} c(\theta_1) + c(\theta_2) + s(\theta_1) + s(\theta_2) & -c(\theta_1)s(\theta_2) - s(\theta_1)c(\theta_2) & 0 & l_2c(\theta_1)c(\theta_2) - l_2s(\theta_1)s(\theta_2) + l_1c(\theta_1) \\ s(\theta_1)c(\theta_2) + c(\theta_1)s(\theta_2) & -s(\theta_1) + s(\theta_2) + c(\theta_1) + c(\theta_2) & 0 & l_2c(\theta_2)s(\theta_2) + l_1c(\theta_1)s(\theta_2) + l_1s(\theta_1) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

After deriving the direct kinematic equation, the inverse kinematic equation has to be derived as well. This relationship helps us determine the joints variables values knowing the end-effector position.

$$\theta_1 = atan2(y, x) - atan2(l_2 \sin(\theta_2, l_1 + l_2 \cos(\theta_2)) \tag{1}$$

$$\theta_2 = \frac{\arccos(x^2 + y^2 - l_1^2 - l_2^2)}{(2l_1l_2)} \tag{2}$$

3.2 Model planning and simulation

After the determination of the kinematic and dynamic models, the main focus of the project became the mechanical design of the manipulator. One of the critical aspect of the dimensioning process is finding a simple and reliable procedure useful to compute a set of data with a physical meaning and with which a payload and actuation system can be estimated. The project methodoloy followed [3] consists in an iterative process that can be schematized this way:

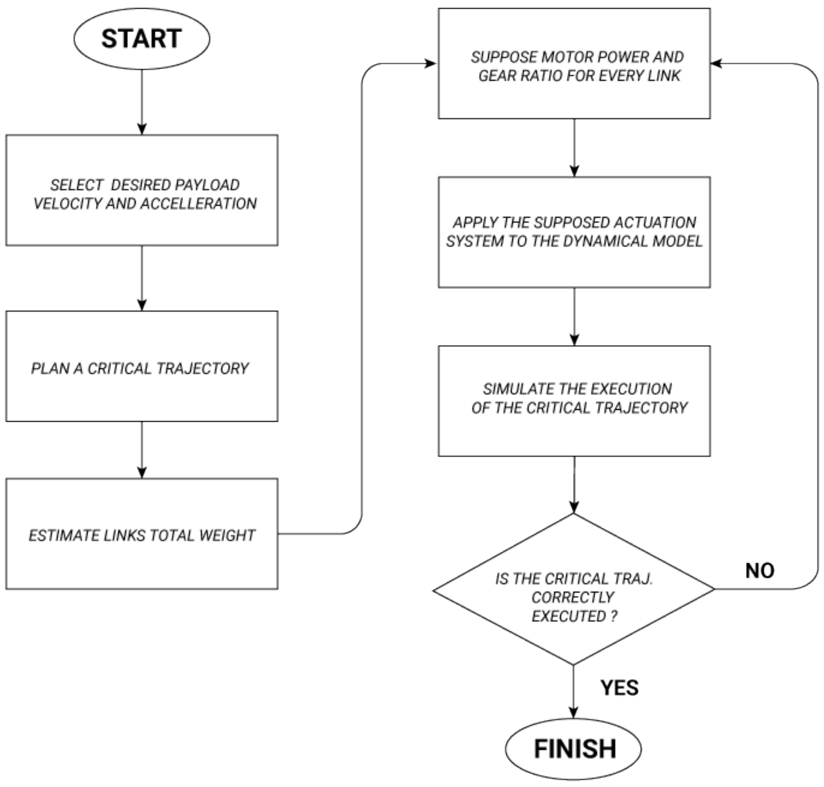

START
SELECT DESIRED PAYLOAD VELOCITY AND ACCELLERATION
PLAN A CRITICAL TRAJECTORY
ESTIMATE LINKS TOTAL WEIGHT
SUPPOSE MOTOR POWER AND GEAR RATIO FOR EVERY LINK
APPLY THE SUPPOSED ACTUATION SYSTEM TO THE DYNAMICAL MODEL
SIMULATE THE EXECUTION OF THE CRITICAL TRAJECTORY
IS THE CRITICAL TRAJ. CORRECTLY EXECUTED ?
NO
YES
FINISH

Figure 3: Schematic representation of the iterative process followed.

The critical trajectory chosen is a third degree polynomial trajectory, computed with the constraint on the final time, that will be maximum 4 seconds, the position will go from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ with maximum acceleration of 1.05 $[\mathrm{rad}/s^2]$. The same trajectory is used for the two joints. The polynomial trajectory is chosen for it's simplicity of computation, this aspect is crucial to save time when a constraint is changed.

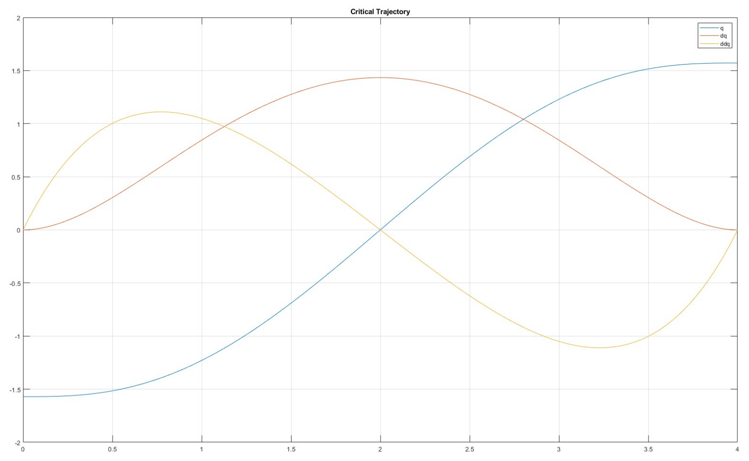


Figure 4: Rappresentation of the critical trajectory.

To be able to perform the Dynamic Simulation of the manipulator, first of all a model of the system is needed, with the help of *Peter Corke Robotics Toolbox* [4] the Lagrangian model is defined. After that using Simulink the inverse dynamic control scheme is implemented and after several iteration, performing different changes, an acceptable set of parameter is defined. The inverse dynamic control system was chosen because compensating the whole dynamics of the system made it easier to compare the output trajectories to the desired ones.

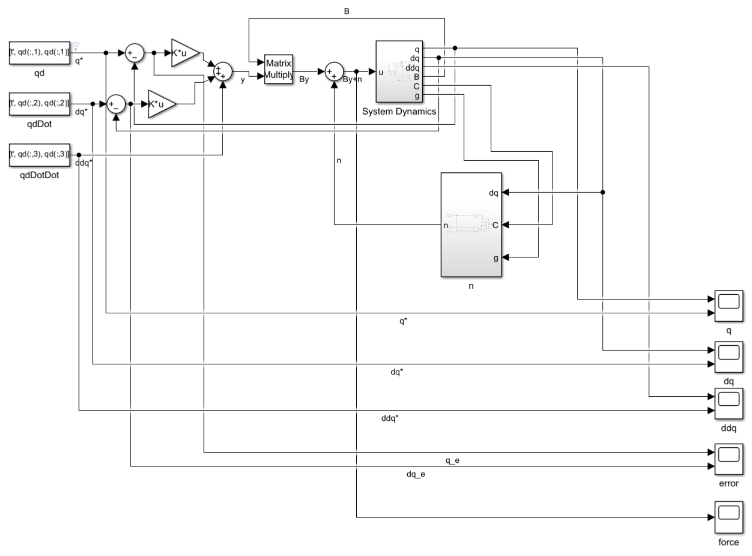


Figure 5: Simulink model used to evaluate the critical trajectory tracking.

3.3 Mechanical Design

When the all necessary parameters were defined, the physical modelling process can start. In the first modelling iteration some hypothesis on the material and the mechanical structure were made. To ensure the production the CAD model was built. With the CAD model the prototyping phase results easier, the first iterations of the CAD model were produced using a 3D printer with PETG filament, but the stiffness of the material was too low, making the arms bend: to overcome this issues, the arms were built by machining them directly from 6060 aluminum. For the milling process, the CNC tool-path was determined using the Fusion 360 software. The arms were machined in both of the sides, for the first arm the motor was mounted with the rotor shaft upside down, to connect the planetary gearbox to the motor shaft a 1:1 pulley transmission system was used.

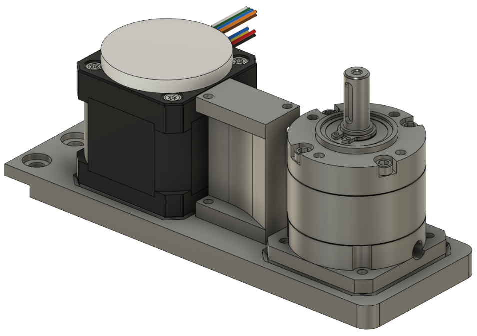


Figure 6: CAD model of the first robot arm.

For the second arm the motor shaft and the gearbox was mounted upward and the same transmission system was used.
To connect the first arm and the second arm, a self centering lock mechanism was used, this piece was crucial to guarantee a correct relative rotation without misalignment between the joints, without inducing undesired elliptical motion.

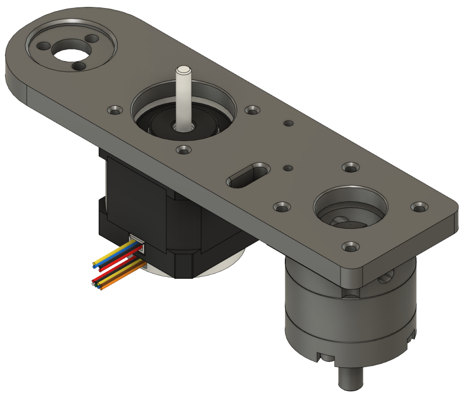


Figure 7: CAD model of the second robot arm.

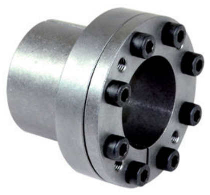


Figure 8: Image of the RCK80 loker built by Chiaravalli Group.

The most iterated part was the base, the first time the base had the same side size as the first arm, but this was not enough to guarantee the stability of the robot, manly during the motion, causing lot of vibrations. To overcome this issue, the latest design was developed doubling the support area and it was printed in a single block, adding inside of the print the nuts for bolting the first arm. Another improvement was made by adding a clamping system to anchor the manipulator to the corner of the working area, increasing the overall stability.

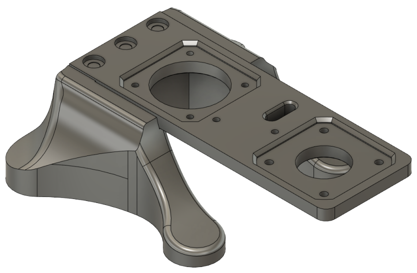


Figure 9: CAD model of the last iteration of the base.

4 Components

As to be expected from a project of this complexity, each functionality was implemented by using multiple software and hardware components made to complement each other and work along side one another. Software wise, all components can be categorized into two macro sets: high level components and low level components; hardware wise the components will be divided into mechanical and electrical components.

4.1 Hardware Components

The connection between the element can be summarized in the subsequent scheme:

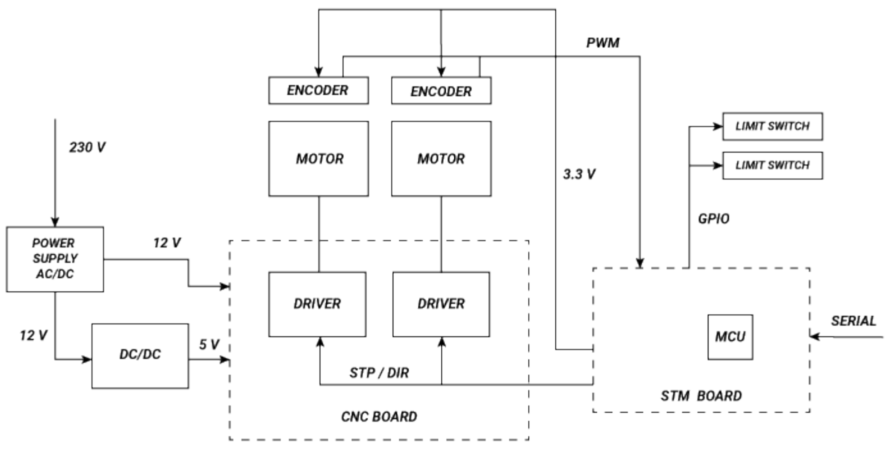


Figure 10: In this scheme were displayed the component and how they are interconnected

4.2 Stepper

The choice of the NEMA 17 motor, a popular option for many small robotics and automation applications, was motivated by several reasons, primarily because it offers a good balance between performance and cost.

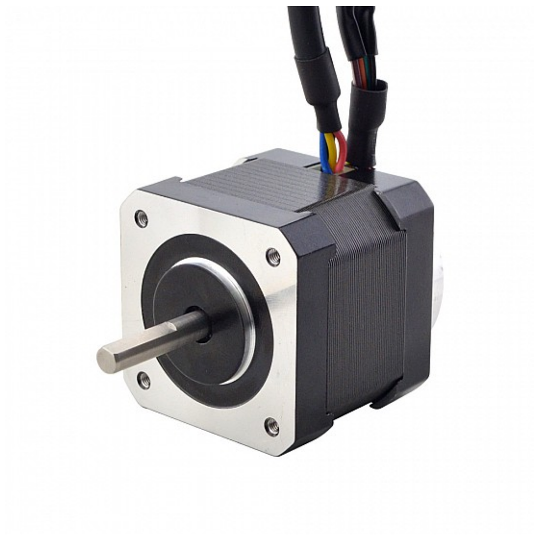


Figure 11: Image of the stepper motor used

The first main reason was that this motor model has an attached magnetic incremental encoder above the stepper motor, making the entire motor system more compact. This aspect was particularly advantageous in this project where space efficiency was crucial.

However, after some tests, it was observed that this configuration can create problems during PID position control; in particular, the problem was found for the last arm but could also be for the first one, although not clearly visible. If an arm lock occurs, the encoder, connected to the previous arm, will register that the error has been corrected, but in reality the arm will continue to move due to blacklashing of the gearbox. Therefore, a better configuration for these problems, sacrificing compactness, might be to place the encoder directly above the arm and attach it to the gears, so that the readings taken are the actual position changes of the gearbox, and thus of the arm. In the configuration used, the actual position is not seen because the encoder reads the rotor displacements. In this way, even with the same gearbox problem, the encoder could accurately detect the actual error and take more effective action. The position of the stepper motor would remain unchanged.

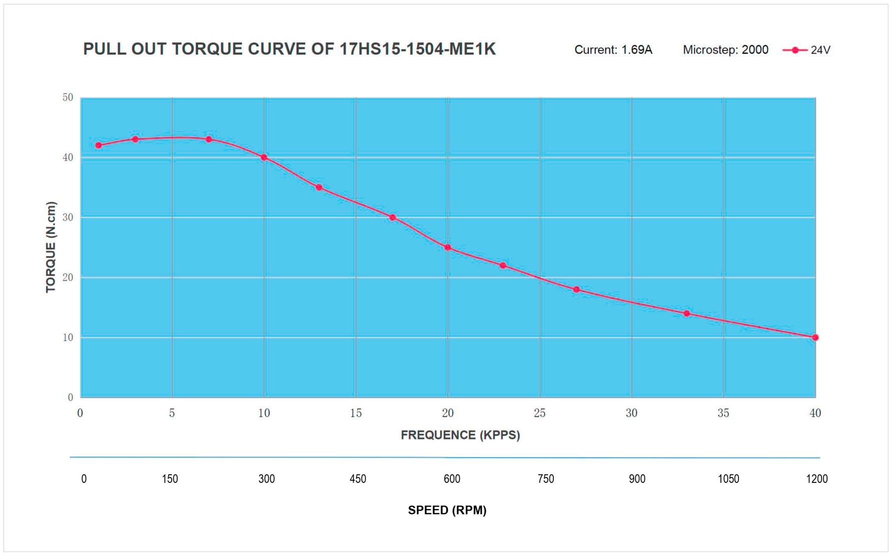


Figure 12: Torque Curve of the stepper Motor

Other reasons for choosing this motor include its strong torque despite its compact size and the precision control enabled by its high number of steps per revolution

4.3 Encoder

The magnetic encoder uses a permanent magnet on a rotating body, like a motor shaft, to sense magnetic field changes. It has a magnet and sensor on a PCB board. When the shaft rotates, the sensor detects changing magnetic fields, enabling the encoder to measure shaft position and speed. A rotating magnetic field comes from the magnet on the shaft. A Hall element in the sensor senses field changes and turns them into electrical signals. As it senses only in one direction, two Hall elements are needed.
In fact, the encoder has two outputs called A and B where a squared waveform is generated. If the rising edge of the channel A comes before the rising edge of the channel B the encoder is rotating clockwise, otherwise it's rotating counterclockwise.

Actually, encoder used has three channel, which the third is for the zero index but it hasn't been used, because there was no need to know when the motor passed the zero point. For this type of reading, the STM used in the project has a hardware module to read encoders.

It could be set directly in timer configuration, choosing Encoder mode.

This module configures a register with a counter that increments or decrements based on the rising edge's direction. We can access this register to obtain position readings and to estimate the velocity and acceleration.

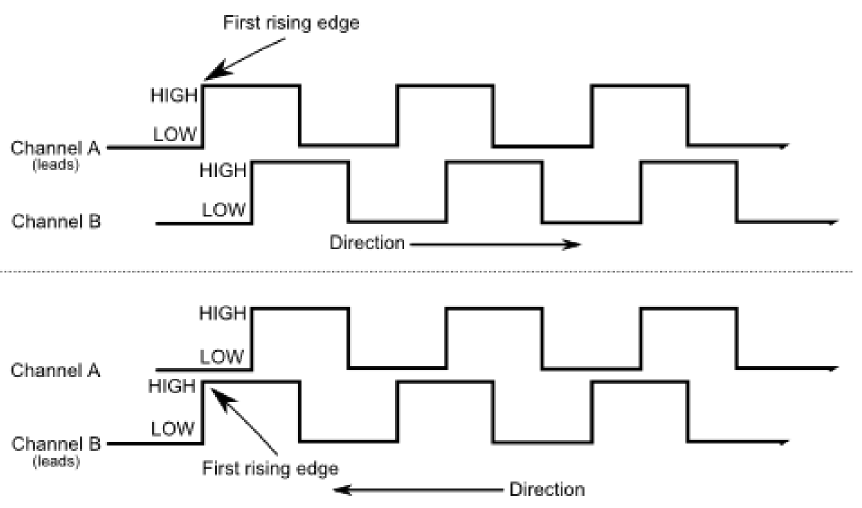
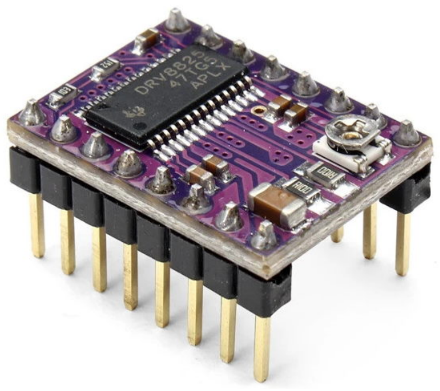


Figure 13: Output signal of the encoder

4.4 Motor Driver



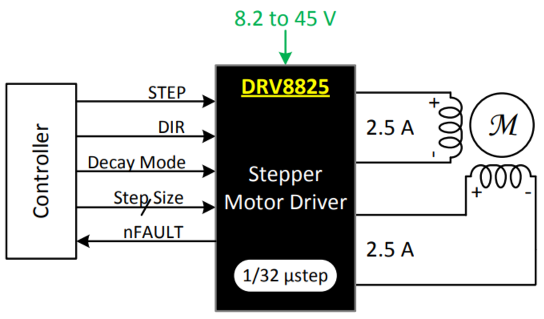


Figure 14: TI DRV8825 Driver, with pinout diagram

- Simple step and direction control interface;
- Six different step resolutions: full-step, half-step, 1/4-step, 1/8-step, 1/16-step, and 1/32-step;
- Can interface directly with 3.3 V and 5 V systems;
- Adjustable current control lets you set the maximum current output with a potentiometer, which lets you use voltages above your stepper motor's rated voltage to achieve higher step rates;

The motor driver chosen to control the motor is the DRV8825 due to its high performance, integrated protections, easy configuration, and broad compatibility. It is particularly suitable for applications that require precise and reliable control of stepper motors.

Moreover, this driver allows microstepping, allowing higher resolutions than the number of motor steps by enabling intermediate step locations. These intermediate step locations are achieved by energizing the coils with intermediate current levels. To select one of the six possible step modes, the three dedicated resolution pins must be correctly connected. For more information, please refer to the driver datasheet [5]. In the project has been used the 1/16 step micro-stepping resolution, that allows to control the position of the manipulator with more precision because there are more available steps and allows to have enough range in terms of max speed affordable by the motor. In fact, Nema 17 has normally 200 steps for revolution, but with micro-steps, there will be 3200 micro steps for revolution. To control the motor, only two pins are needed: one for the direction and one for the step. The direction is simple, logic level 0 is clockwise rotation, while logic level 1 is counterclockwise (according to the motor wires connections). To trigger a step on the step pin, a transition from 0V to 5V is required. Therefore, if a continuous rotation is desired, a square wave signal needs to be generated. However, with this driver is not possible to do a direct position control but only a velocity control, because it takes as input only the frequency of how many steps for seconds it has to do, and the direction. So, the frequency of this signal is proportional to the wanted speed. To impose the velocity, it has been necessary to use a general-purpose timer for the Pulse Width Modulation (PWM) generation. Once the desired velocity is known, there will be some operations to do for setting the properly the registers:

- Auto-Reload Register (ARR): responsible of the period of the waveform;
- CCR: responsible of the duty cycle. The duty cycle has always been set to 50% (half value of ARR). It is set to 0% only to stop the motor.

4.5 Reductions

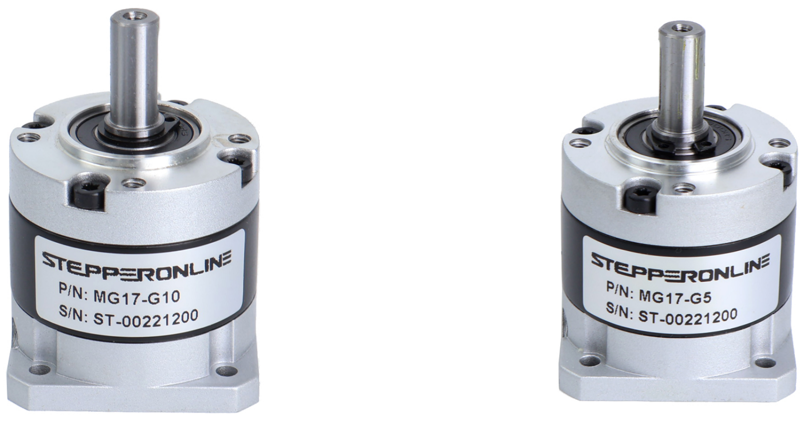


Figure 15: MG17-G10 and MG17-G5 reduction gears

Planetary gearboxes were chosen for the project due to their numerous structural advantages. In fact, they have a simple structure but allow for almost perfect load distribution and high efficiency. In fact, in this way it is possible to transmit high torques with high efficiency even with a compact structure.

A planetary gearbox has a central sun gear surrounded by evenly spaced cylindrical sprockets that rotate concentrically. The reduction ratio is determined by how the sun gear interacts with the crown gear. A smaller sun gear results in a higher ratio.

The decision to employ 10:1 and 5:1 reduction gears for the first and second arms respectively was based on simulation tests. Through these tests, it was determined that utilizing these specific reduction gears, along with their associated ratios, enabled the manipulator to achieve proper and effective motion.

4.6 STM32F446RE MCU

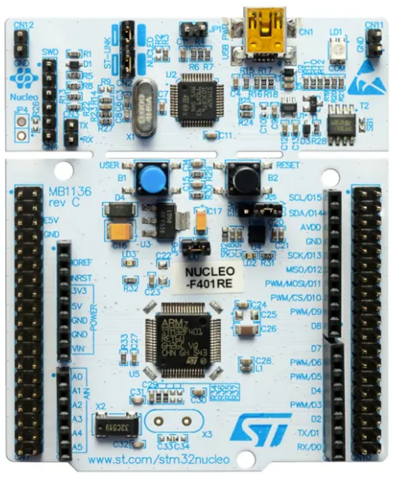


Figure 16: STM32F411RE Microcontroller

The STM32F446RE [6] board was initially intended to be used for the project, as it offers a higher CPU clock frequency and has more timers and ADCs. However, during the purchase period, this board was unavailable on the market due to the chip crisis. As a result, the STM32F411RET6 board was used, which offers everything necessary for the project to continue properly.

The board's datasheet can be consulted for further details.

To write the code and flash it, the manufacturer's IDE, STM32CubeIDE, was used, as it was already familiar from the course tutorials. The IDE features theSTM32CubeMX tool, which allows the board to be adjusted at a higher level.

4.7 Software Components

Considering the complexity of the project, was impossible to use one single software component or stack, for reducing the overall complexity the software was divided in two different parts an "High Level" ,that runs on a PC, and a "Low level" part, mostly composed by the firmware that runs on the micro-controller board.

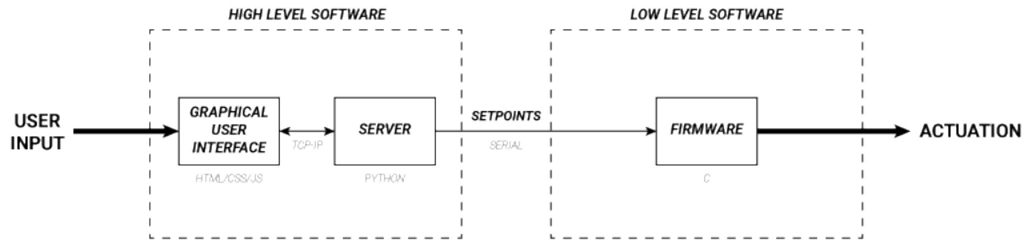


Figure 17: This scheme visualizes the software components and the interconnection between them.

4.7.1 High Level Software

On the highest software level it is possible to find the *trajectory creation tool*, which was made using Web Technologies (HTML[7] and Javascript[8]) and Python[9]. It uses the Eel Framework[10] to host a local webserver that allows for the communication between a Python backend and a HTML/Javascritp fontend via *Function Exposition*. Through this framework, it was possible to create a simple GUI using HTML, CSS and Javascript that allowed the user to draw with the mouse the trajectory that they wanted the manipulator to follow, to then use Python to handle the trajectory data, converting it into a list of joint space setpoints to be sent to the microcontroller.

These set-points are sent via the `"TRJ"` command message, structured in the following way:

$$TRJ : q_1 : q_2 : \dot{q}_1 : \dot{q}_2 : \ddot{q}_1 : \ddot{q}_2 : penup\backslash n \tag{3}$$

where $q_i, \dot{q}_i$ and $\ddot{q}_i$ are the positions, velocities and accelerations of each motor, while *penup* is a single bit used in case a pen is attached to the end-effector, signaling whether the pen should write on the paper or not. each variable (a `Python float`) is converted into its IEEE 754 standard hexadecimal representation before being sent: this ensures that each message has always the same size independently from the actual values each variable may take.

Trajectory Tracing The trajectory creation tool lets the user trace on a canvas the desired trajectory for the manipulator; this is accomplished by using two geometrical primitives: the *line* and the *circumference*. By pushing a button, the user can switch between the line tool and the circle tool to create a continuous trajectory made of multiple segments, called *patches*. By pressing the "Send Data" button the trajectory is split in its patches that then undergo a *slicing phase*, during which, by using a cycloidal timing law (Figure 18), they are sampled into multiple points.

Listing 1: Cycloidal function definition: it is highly dependant on the duration of the motion and the starting and ending values of the cycloidal

```python
def cycloidal(q:list[float], ddqm:float = 1.05, tf:float=None) ...
    -> tuple[list[function], float]: # return the function ...
    handles for q, dq and ddq
     if tf is None:
         tf = sqrt(2*pi*abs(q[1]-q[0])/ddqm)
     qt  = lambda t: q[0]+(q[1]-q[0])*(t/tf-sin(2*pi*t/tf)/(2*pi))
     dqt  = lambda t: (q[1]-q[0])*(1-cos(2*pi*t/tf))/tf # ...
         derivative of q
     ddqt  = lambda t: 2*pi*(q[1]-q[0])*sin(2*pi*t/tf)/(tf**2) # ...
         2nd derivative of q
     return ([qt, dqt, ddqt], tf)
```

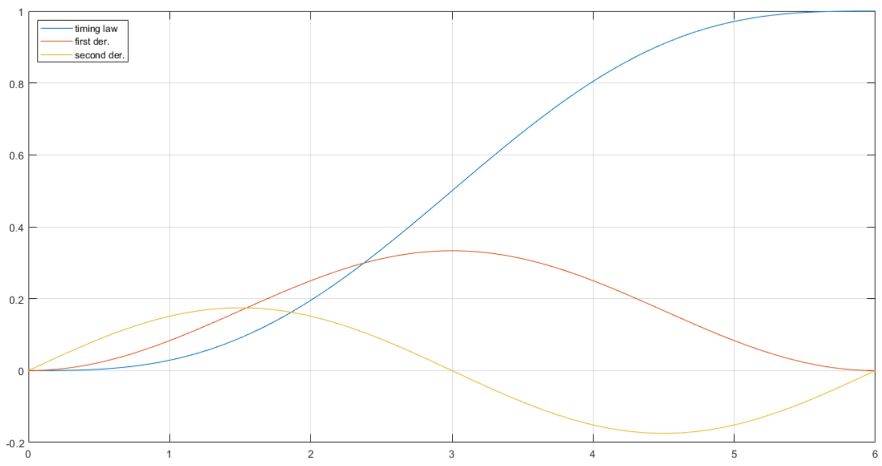


Figure 18: Cycloidal timing law (in blue) used during the slicing phase with its first and second derivative (respectively in red and yellow)

These points are then used as setpoints in the operational space that then are converted into joint space setpoints via *inverse kinematics* with the geometric laws described by equations 1 and 2.

The cycloidal timing law used changes between 0 and 1 in a time period that can be specified by the user: the value generated can be used as "percentage" that dictates how far into the trajectory patch the manipulator is. Once the timing law is created based on the movement duration, it is sampled with a constant sampling time and then used to generate the operational space setpoints as follows:

$$\textit{line primitive}: \ \bar{p}_0 + (\bar{p}_1 - \bar{p}_0) \cdot s, \ \forall s \in [0,1] \tag{4}$$

$$\textit{circle primitive}: \ \bar{c} + (\bar{p}_0 - \bar{c}) \angle [(\theta_1 - \theta_0) \cdot s], \ \forall s \in [0,1] \tag{5}$$

where $\bar{p}_0, \bar{p}_1$ are the initial and final point vectors of the trajectory patch, $\bar{c}$ is the center vector of the circumference and s is the resulting value from the timing law:

$$s(t) = \left(\frac{t}{t_f} - \frac{sin(2\pi \cdot \frac{t}{t_f})}{2\pi} \right), \ \forall t \in [0, t_f] \tag{6}$$

As it can be seen from the equations 4 and 5 above, the points of the trajectory are obtained by moving the starting point of each patch for a certain distance (defined by s in percentage) along the path defined by the patch itself: the points of a line patch are obtained by translation, the points of a circle patch, on the other hand, are obtained by rotation (represented by the $\angle$ symbol).

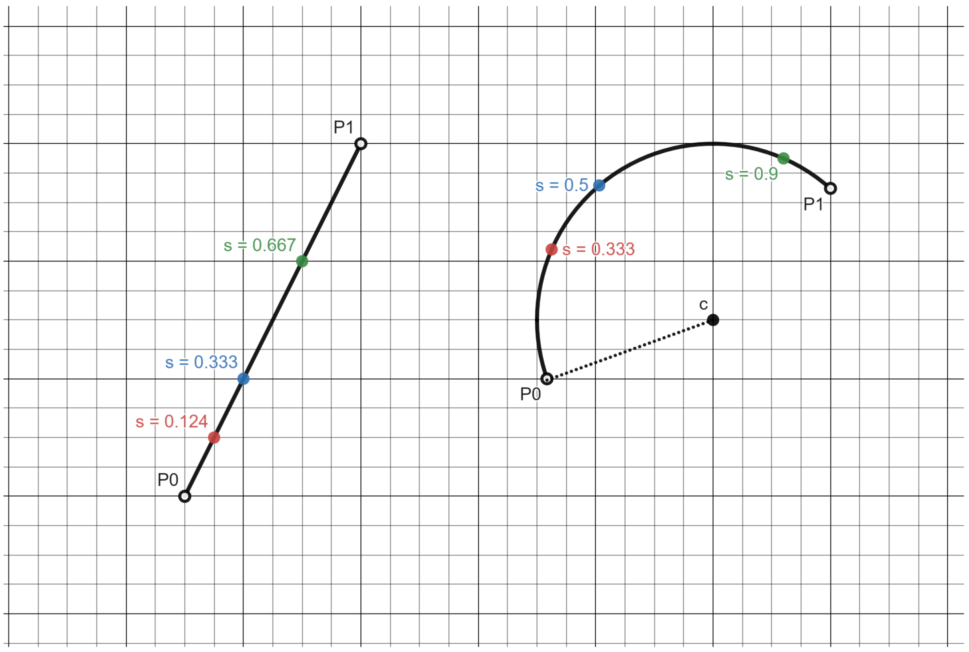


Figure 19: Example of how the primitives are sliced: each colored point represents the point that is found by slicing the trajectory with a specific value of s

Once the slicing phase is completed for each patch, the list of points that is found is converted into a list of setpoints in the joint space, which is sent to the micro-controller.

4.7.2 Low Level Software

Signal conditioning To guarantee usable measurements for the feedback, a limited noise filtering techniques was used. First of all, using the encoder the position of the rotor is measured. From this data, velocity and acceleration were calculated using discrete derivation (with Euler's Method). The first step of the filtering is using a moving average of the position; this is done because planetary gearboxes are used, and given that the encoders have high resolution, every little vibration causes a displacement reading causing very high increments in the velocity. This series of spikes in the velocity value and acceleration can be mitigated with this position averaging technique.
Using Matlab [1] the filter was designed in the frequency domain, after that to implement it correctly on the micro-controller a discrete transformation was applied, in the discrete time domain the filter formulation is:

$$y(k) = \beta x(k) + (1 + \beta)y(k - 1) \tag{7}$$

Where $\beta = e^{-w_0 T}$, w_0 is the cutoff frequency and T is the sampling time.

An important aspect to highlight is the fact the $y(k-1)$, in our system, represents an average value and in our iteration we notice that considering the right time elapsed during the acquisition of the data is crucial for having meaningful results.

Ringbuffer To store the data received from the HMI back-end and from sensors, a simple circular buffer data structure, called *Ringbuffer*, was used.

Listing 2: Data structure of the ringbuffer

```c
typedef struct ringbuffer {
    uint8_t tail;               /* where data will be pushed */
    uint8_t head;               /* where data will be popped */
    uint8_t length;             /* number of elements */
    rbelement_t buffer[RBUF_SZ]; /* actual buffer */
} ringbuffer_t;
```

The `Ringbuffer` data structure follows the *FIFO* access method: two indexes point to the tail and the head of the structure, relatively the input and output indexes; these indexes are incremented when data is inserted or removed, making this ringbuffer a moving circular buffer structure. To access data stored within the data structure, the `rbpush(ringbuffer_t *buffer, rbelement_t data)` and `rbpop(ringbuffer_t *buffer, rbelement_t *data)` methods have been implemented, which let the user push and pop data from the buffer. The `rbpop` method returns the least recent element inserted in the buffer and deletes it from the data structure, but in certain cases the element might still be needed elsewhere: to avoid data loss, the methods `rbpeek(ringbuffer_t *buffer, rbelement_t` `*data)`, `rblast(ringbuffer_t *buffer, rbelement_t *data)` and `rbget(ringbuffer_t *buffer, int8_t i, rbelement_t *element)` were implemented to allow access to the data without deleting the accessed element. Lastly, the `rbclear(ringbuffer_t *buffer)` method is used to completely reset the ringbuffer, initializing it to a known state.

The aforementioned data structure was used to store data received from the serial wire (the set-points) and from the sensors (motor shaft positions), the latter than used to compute the speed of the motors also stored within a ringbuffer. All this data regarding the manipulator was stored within a `manipulator` data structure, containing all the aforementioned ringbuffers.

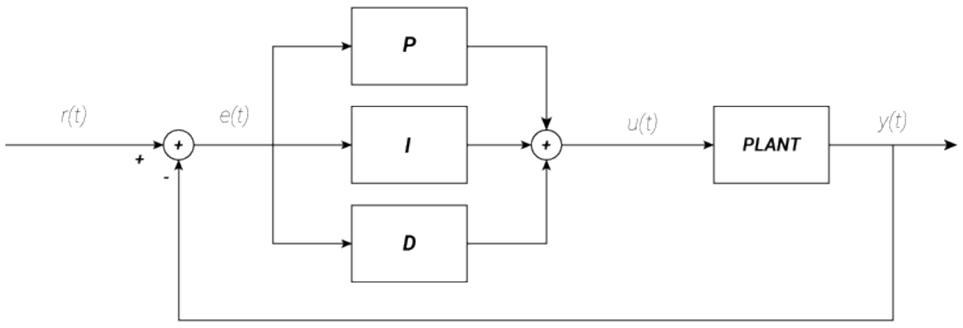


Figure 20: Schematic representation of PID.

PID Controller In order to manage the control loop of the planar manipulator, in the project was used the proportional–integral–derivative controller (PID controller or three-term controller), one of the most widely used control techniques in the fields of automation and dynamic systems control. This technique is fundamental for regulating and maintaining the performance of a wide range of industrial processes, systems, and devices.

A PID controller continuously calculates an error value e(t) as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction based on three terms:

- *Proportional* (P): Responds to the current error by applying a corrective action proportional to the error's size, quickly reducing the error;
- *Integral* (I): Accumulates errors over time, gradually increasing control action to eliminate persistent errors, improving long-term accuracy;
- *Derivative* (D): Considers the rate of error change, helping anticipate future changes and enabling the controller to preemptively adjust to enhance stability.

Appropriate use of the PID requires tuning of the parameters: K_p, T_i and T_d. For tuning, a Ziegler-Nichols heuristic method was used in Matlab in which different values were obtained depending on the type of controller chosen (PI or PID) and motors.

Starting from the the system itself, the position (in open loop) of each motor was logged into a `csv` file together with the timestamp at which the measurement was done: `TIMESTAMP, VALUE`. Once this data was collected, MATLAB [1] was used to analyze the output and understand how the motors behaved in an open loop configuration. This data is fundamental to compute the PID parameters using the aforementioned Ziegler-Nichols heuristics, again using MATLAB for ease of use and precise computations. The resulting values where then set up in the firmware as macro constants to be used in the PID implementation of this project:

PI: First and second motor:

- $K_p = 0.291$;
- $T_i = 0.64$;

PID (Derivative component not used):
First motor:

- $K_p = 20$;
- $T_i = 4$;

Second motor:

- $K_p = 20$;
- $T_i = 4$;

PI was used for speed control as it could offer better results than PID, which was only used for position control.

The need for two types of controllers (or better, 4 different PID controllers) has risen from the homing problem: to make sure that the position read from the encoder was actually the correct one, it is mandatory to take into account the initial offset error (resulting from the fact that the encoders used are relative and not absolute) that prevents the use of position control. Two PI controller are used to implement speed control for the motors during the homing procedure, which is used to compute the position offset to then correct it. The other two PID controllers are used during the normal operations of the manipulators to implement position control.

For the project, the continuous-time PID was discretized using Tustin, in fact the formula obtained from the discretization is:

$$z = \frac{1 + s\frac{T_S}{2}}{1 - s\frac{T_S}{2}} \tag{8}$$

The data of a PID controller, which can be divided into four blocks, is found within a structure called `pid_controller_t`. The first block contains the gain parameters described earlier. The second contains the previous error and measurement values, plus the proportional and derivative action values. The third is dedicated to the limits for the controller output and the integrating element. The last is the output, which is the control action of the PI or PID.

In addition, there is the variable type which is used to indicate the type of controller: PI or PID.

Listing 3: PID data structure implementation

```c
	typedef struct{
		int type;

		/* GAINS */
		float Kp;
		float Ti;
		float Td;
		float N;

		/* MEMORY */
		float prev_err;
		float prev_meas;
		float integrator;
		float derivative;

		/* LIMITS */
		float lim_out_min;
		float lim_out_max;

		float lim_integ_min;
		float lim_integ_max;

		/* OUTPUT */
		float out;
} pid_controller_t;
```

Notice the lim_out_min, lim_out_max, lim_integ_min and lim_integ_max members of the struct: these values are used for a simple implementation of an *anti-windup mechanism*, that simply sets a maximum and minimum limit value for both the output value of the PID controller and its integral value, which avoids the saturation of both the integrator and the actuator that will receive the PID output.

For controller initialization, two functions have been defined which are called inside the main, before the main loop. The first is PID_init(pid_controller_t *pid, float KP,float TI, float TD, float N, int Controller_type) for the controller values and set_limit(pid_controller_t *pid, float lim_out_min, float lim_out_max, float lim_integ_min,float lim_integ_max) to initialize the maximum and minimum limits of the output and integrator.

After initialization, the PID_update(pid_controller_t *pid, float set_point , float measure, float T_C) method is used, which updates the value of the controller relative to the error value, calculated as the difference between the setpoint and the output measurement.

Depending on the type of controller, the output control action is calculated, with the derivative term if it is a PID controller or without for a PI.

This method is used in position and velocity control methods.

Listing 4: distinction between PI and PID

```
    if (pid->type>0){
        u=proportional+pid->integrator;
    }else{
        /*derivative contribute*/
        pid->derivative = (2*(pid->Kp)*alpha*error
        - pid->derivative*(1-(2*alpha)/pid->N))/
        (1+(2*alpha)/pid->N);
        u=proportional+pid->integrator+0*pid->derivative;
    }
```

The PID_controller_position(man_t *manip, pid_controller_t *pid1, pid_controller_t *pid2, float *u) method is used to compute the speed control output for the motors when executing trajectories. Even though it is a position control, it is necessary to input the speed required for the motors to reach the desired position, since the motor driver accepts a STEP/DIR input, which represents a velocity command.

The first step in the method is to calculate distance between the reference position and the actual measured position, then to convert from a position input to a velocity input, a discrete derivation method was used (Euler method) using as the time period the equivalent time that a cycloidal trajectory would have taken to reach the desired position. This approach results into a slower but safer trajectory, that does not have discontinuities, with a relatively simple code implementation:

Listing 5: time period computation in apply_position_input for the first motor

```
1        /* if the movement is too small, ignore it -> avoids ...
            vibrations */
2       /* find the time duration of the movement based on the ...
           cycloid */
3       if (ABS(u[0]-pos[0])<0.01){
4           tc0= 1000000;
5       }else{
6           tc0 = sqrt(2*M_PI*ABS(u[0]-pos[0])/0.3);
7       }
```

if the resulting movement of the manipulator is too small (an arbitrary threshold of 0.01 was chosen empirically), then instead of using the relationship between cycloidal and max acceleration (seen at line 6 of listing 5) the time period is set to a constant value, large enough to make speed approach zero, avoiding any motor movement.

Speed output is determined by the ratio of distance to time.

In the homing procedure, on the other hand, the `PID_controller_velocity(man_t *manip, pid_controller_t *pid1,pid_controller_t *pid2, float *u)` method was used to perform a velocity check. This method will simply read the velocity, reference and current values in order to update the PI control action.

Firmware The complexity of the firmware makes its description difficult, this is why the firmware will be described by following the logical steps of how the entire system is supposed to function.

When the microcontroller is started (after connecting it to the PC), the initial setup funtions are called to setup all the data structures, the timers used to read the encoders and to actuate the motors: `start_timers(...)`, `setup_encoders(...)`. Afterwards, just before the main loop, the first *data reception request* is done to start the set-point reception: `HAL_UART_Receive_DMA(...)`.

Once the user defined set-points are converted into joint space set-points and then sent via serial communication to the micro-controller, the `HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)` callback associated to the reception interrupt is called, where the data will be read from the reception buffer and inserted into the corresponding ringbuffers depending on the value received (position, speed or acceleration). These buffers will be accessed within the main loop when needed.

Given the set-points, to close the control loop the encoder data is needed, which is read within the TIM10 callback `HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)`, which is called with a time period defined by the `T_S` constant macro.

Within the main loop, the `update_speeds(man_t *manip)` method is called, which given the encoder readings computes the actual speeds and accelerations of the motors (both saved within their relative ringbuffers).

Afterwards, the PID_controller_position(man_t *manip, pid_controller_t *pid1,pid_controller_t *pid2, float *u) method is called: this method computes the control action (speed) to actuate with the motors, which will be stored in the variable u (an array of floats of size 2).

Once the control action is computed, it is applied with the apply_velocity_input (TIM_HandleTypeDef *htim1, TIM_HandleTypeDef *htim2, float *u) method which converts the speed input into the frequency of the PWM signals that will be sent to the STEP pins of the drivers driving the motors. Afterwards, the rate_sleep (rate_t *rate) method is called to wait a time period long enough to have a consistent control time, defined by the T_C constant macro.

Outside the main loop, the timers used for the encoders and the motors are stopped before the firmware reaches its end.

For further implementation details, check the documentation attached with this report or the github repo.

5 Peripheral and system configuration

Regarding the peripherals and system configuration, most of the details are covered in the configuration report: the following paragraphs will briefly describe how the system and its peripherals are configured.

5.1 Communication Peripherals and Protocols

The protocol used for communication is the *USART* protocol, which used the serial communication wire also used to flash the device (`usart2`). It was set up with a baudrate of 115200 with a word length of 8 bits (without parity checking) with 16 oversampling samples. The communication was set up to be used in DMA mode.

5.2 Timers

A set of 5 timers are used to handle sampling time, encoder readings and motor control.

- *TIM2 and TIM5*: these two timers are used to generate the PWM signals to control both the motors of the arms. To control the PWM signals, the ARR, CCR and PSC registers are set programmatically. The Prescaler is set to a fixed value of 5200 for the first motor and 8400 for the second one, to ensure that the maximum frequency of the PWM signal fed to the DRV8825 driver does not exceed the maximum value of `250 kHZ`, as stated on the official documentation (pag. 7).
  The ARR register is set relative to the velocity set point:

$$ARR = \frac{RESOLUTION \cdot f_{clk}}{(|v| \cdot reduction \cdot 16 \cdot prescaler)} \tag{9}$$

  where $RESOLUTION$ is the motor step resolution, f_{clk} is the clock frequency, v is the velocity input, *reduction* is the reduction ratio, *prescaler* is the prescaler value and 16 is the number of microsteps set on the CNC board to which the driver is connected. The CCR register is set as half of the ARR register, meaning that the PWM as a constant duty cycle of 50%, which is actually unimportant given that the velocity at which the motor turns only depends on the frequency of the PWM signal.
  Notice how in the equation 9 the sign of the speed v is ignored, that's because the direction of the movement of the motor is controlled by setting to `HIGH` or to LOW the direction pin of the driver.

- *TIM3 and TIM4*: these timers are used to read and decode the signals outputted by the encoders attached to the motors. The timers are set to `Encoder Mode` and their ARR registers are set to fixed values: respectively 40000 for the first encoder and 20000 for the second encoder. These values where chosen to take into account the characteristics of each motor-reduction pair, to have the correct position value (in radians), following this formula:

$$ARR = mode \cdot CPR \cdot reduction \tag{10}$$

where *mode* is the the timer encoder mode ($4x$ mode is used in this project), CPR is the Count per Revolution parameter (1000 for the motors used) and *reduction* is the reduction ration (10 for the first motor and 5 for the second).

- *TIM10*: this timer is used to time correctly the instant when the encoders will be read: the ARR and PSC registers are fixed based on the chosen sampling time:

$$ARR = \frac{T_S \cdot f_{clk}}{prescaler} \tag{11}$$

with T_S is the sampling time, $f_c lk$ is the clock frequency and *prescaler* set to 16.

5.3 GPIO Configuration

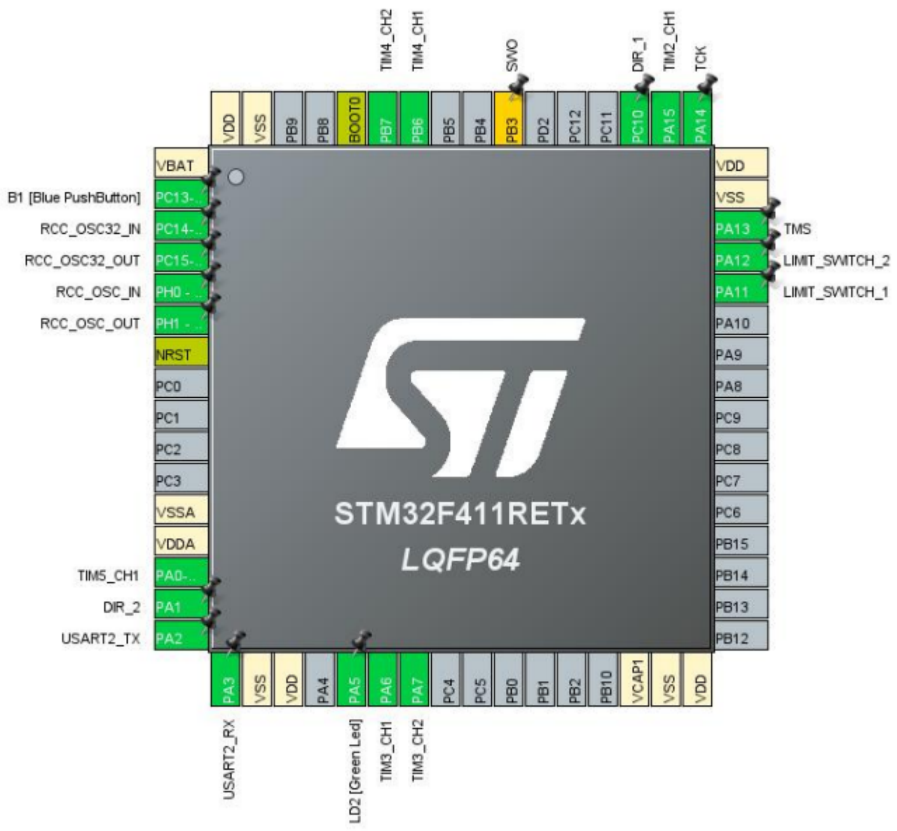


Figure 21

5.4 Timers

- Pin: PA15 (TIM2), PA6 and PA7(TIM3), PB6 and PB7(TIM4), PA0 WKUP(TIM5);
- GPIO mode: Alternate Function Push Pull;
- GPIO pull up/pull down: No pull-up and no pull-down;

5.5 Limit Switch

- Pin: PA11 (limit switch first motor), PA12(limit switch second motor);
- GPIO mode: External Interrupt Mode with Rising edge trigger detection;
- GPIO pull up/pull down: pull-up;
- User Label: LIMIT_SWITCH_1 , LIMIT_SWITCH_2;

External Interrupts are associated with the limit switches to stop the motors when activated (avoids that the arms get outside the workspace).

5.6 Motor Movement Direction

- Pin: PA1 (direction for the second motor), PC10(direction for the first motor);
- GPIO mode: Output Push Pull;
- GPIO pull up/pull down: No pull-up and no pull-down;
- User Label: DIR_1 , DIR_2;

Through these pins, signals are sent to the DRV8825 to indicate which direction the movement of the respective motor should adopt. If the signal is at `HIGH` then the direction is in counterclockwise, if `LOW` the direction is clockwise.

5.7 DMA Configuration

For the project, it was necessary to configure DMA for both reception and transmission because continuous high data exchange was needed. Regarding reception, the data is received from Python and concerns the trajectories to be executed; regarding transmission, the data concerns the values read from the encoder to be sent for data analysis.

The configuration:

- DMA Request: USART2_RX , USART2_TX;
- Stream: DMA1_Stream5 (RX), DMA1_Stream6 (TX);

- Direction: Peripheral To Memory (RX), Memory To Peripheral (TX);
- Priority: Medium;
- Mode: Circular;
- Use FIFO: Disable;
- Peripheral Increment: Disable;
- Memory Increment: Enable;

The controller configuration is described in more detail in the configuration report.

6 Conclusions and Future Developments

This project was really challenging and has forced us to push our knowledge more and more. The first real challenge was understanding how to subdivide the project to be manageable. This was our first approach to a big project, we have mostly used an Agile approach, starting from the control loop and than going deeper and deeper, the first part developed was the HMI interface the last the homing procedure. In this "journey" we have used all of the notion learned on the curse, and in general from our whole academic career.

The most difficult part was learning how to correctly implement the control loop and, when the PID controller was synthesized, how to correctly tune it. Another challenging aspect was applying the correct control input to the stepper motor, translating the position control needed, into a velocity input to drive the motor driver. There are many possible future implementation that can be done: From the mechanical point of view, instead of planetary gearbox, an harmonic drive could have been used. this drive guarantees better performances reducing backlash and improving the overall precision and repeatability. Another mechanical improvement that can be done is a sturdier base, maybe using epoxy granite composite material. This solution, if correctly implemented can reduce the vibration during the motion. Instead, from the actuation point of view, to guarantee better performances, a series of small servos can be used: this can increase the performance of the manipulator not only for the type of motor used but also thanks to the more sophisticated control algorithms that can be implemented on a servo that guarantees an overall increase of the dynamic performance. From the controller side, if the system can be modelled properly, an inverse dynamic controller can be implemented. This algorithm can easily increase the overall dynamic performance.

References

[1] *MATLAB*. https://www.mathworks.com/products/matlab.html?s_tid=hp_ff_p_matlab. Accessed: 2023-07-26.

[2] *Simulink*. https://www.mathworks.com/products/simulink.html?s_tid=hp_ff_p_simulink. Accessed: 2023-07-26.

[3] Morteza Shariatee et al. “Design of an economical SCARA robot for industrial applications”. In: *2014 Second RSI/ISM International Conference on Robotics and Mechatronics (ICRoM)*. 2014, pp. 534–539. DOI: 10.1109/ICRoM.2014.6990957.

[4] *ROBOTICS TOOLBOX*. https://petercorke.com/toolboxes/robotics-toolbox/. Accessed: 2023-07-26.

[5] *DRV8825 driver TI official documentation*. https://tinyurl.com/ti-documentation-drv8825. Accessed: 2023-07-26.

[6] *STM32F411RE MCU*. https://www.st.com/en/microcontrollers-microprocessors/stm32f411re.html. Accessed: 2023-07-26.

[7] *HTML - HyperText Markup Language*. https://developer.mozilla.org/en-US/docs/Web/HTML. Accessed: 2023-07-26.

[8] *Javascript*. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 2023-07-26.

[9] *Python*. https://www.python.org/. Accessed: 2023-07-26.

[10] *Eel Framework - Python*. https://github.com/python-eel/Eel. Accessed: 2023-07-26.

[11] Cheng Zhang and Zhuo Zhang. “Research on Joint Space Trajectory Planning of SCARA Robot Based on SimMechanics”. In: *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. 2019, pp. 1446–1450. DOI: 10.1109/ITNEC.2019.8729547.

[12] Claudio Urrea, Juan Cortés, and José Pascal. “Design, construction and control of a SCARA manipulator with 6 degrees of freedom”. In: *Journal of Applied Research and Technology* 14.6 (2016), pp. 396–404. ISSN: 1665-6423. DOI: https://doi.org/10.1016/j.jart.2016.09.005. URL: https://www.sciencedirect.com/science/article/pii/S1665642316300931.

[13] Mahdi Alshamasin, Florin Ionescu, and Riad Al-Kasasbeh. “Modelling and simulation of a SCARA robot using solid dynamics and verification by MATLAB/Simulink”. In: *Int. J. of Modelling* 15 (Jan. 2012), pp. 28–38. DOI: 10.1504/IJMIC.2012.043938.

[14] *Github Repo*. https://github.com/dede-amdp/robotics1_proj.