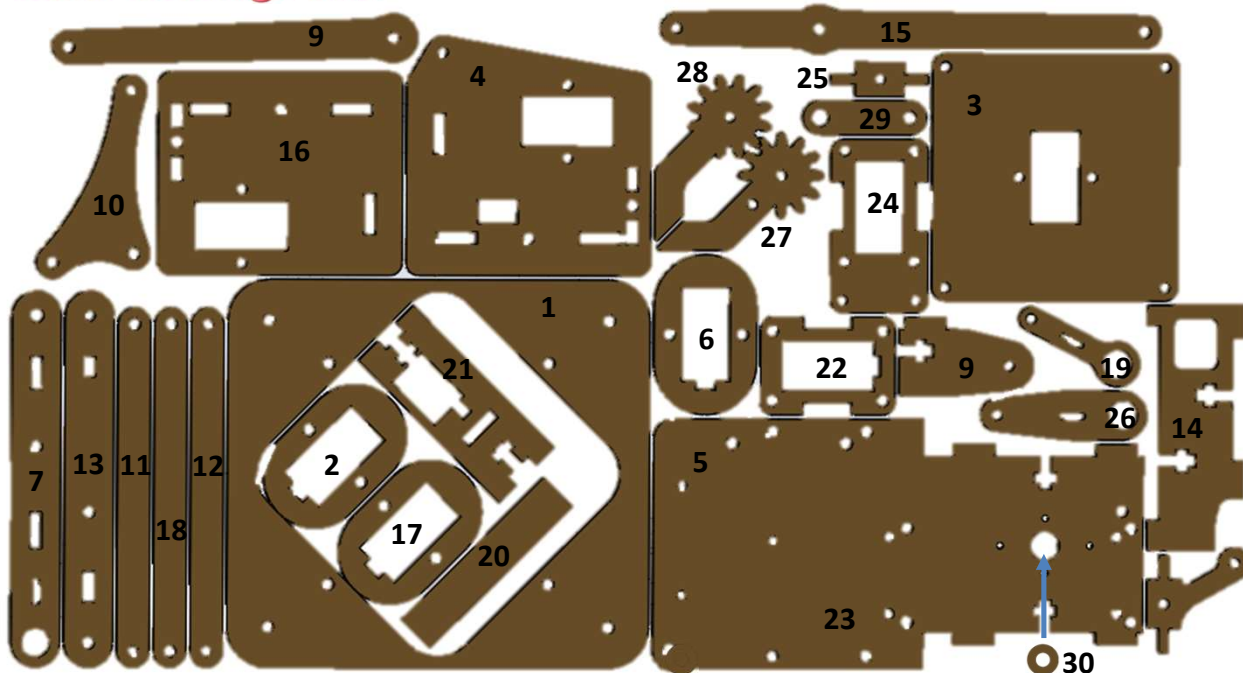
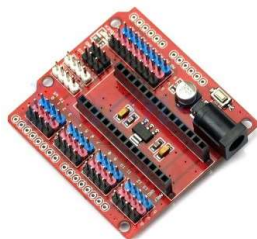


Arduino Materials

Laser Cutting Parts



1x Arduino + USB Cable



1x Extension Shield



4x Servo motor



1x Li-Ion 16340 Battery



1x DIY 16340 Powerbank



Spacers, Screws and Nuts



Selftap screws Ø2mm

Tools



1x Hot-glue gun

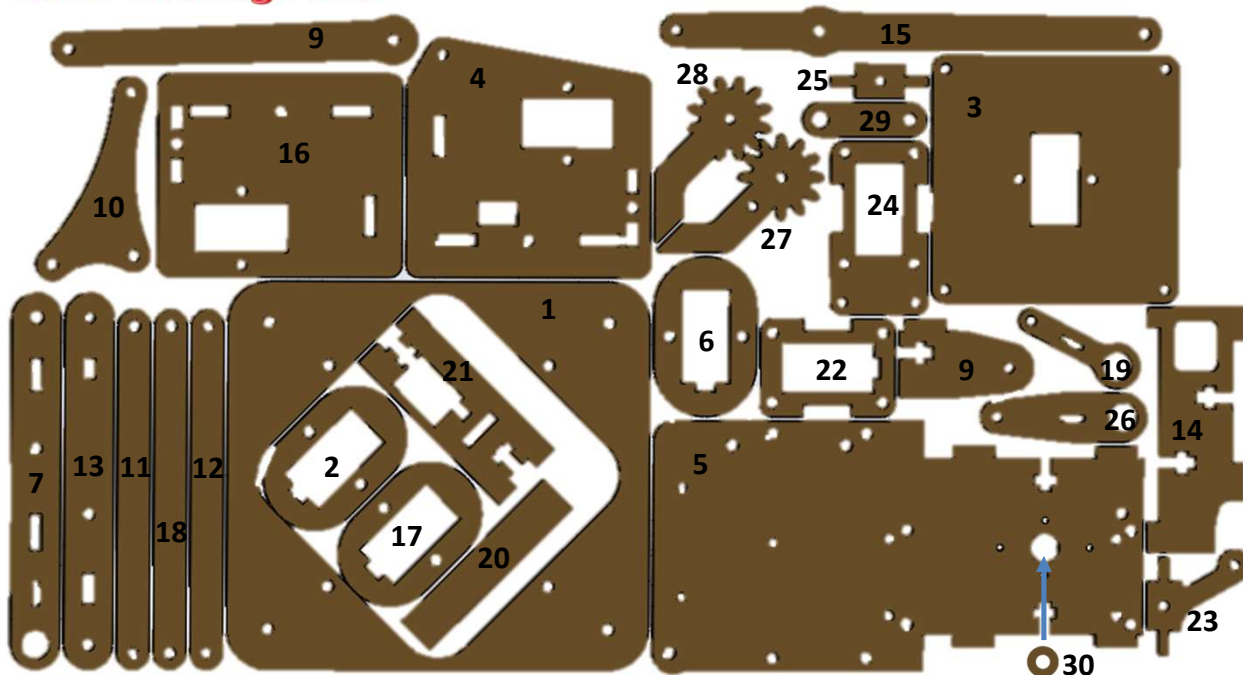


Pliers y screwdriver

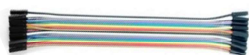


ESP-01S Materials

Laser Cutting Parts



1x ESP-01S with
USB Programmer



10x DuPont Cables F-F



1x Servo Controller PCA9685



4x Servo motor



1x DIY 16340 Powerbank



1x Li-Ion 16340 Battery



Spaces, Screws, Nuts



Selftap screws
Ø2mm

Tools



1x Hot-glue gun



Pliers and scwdriver

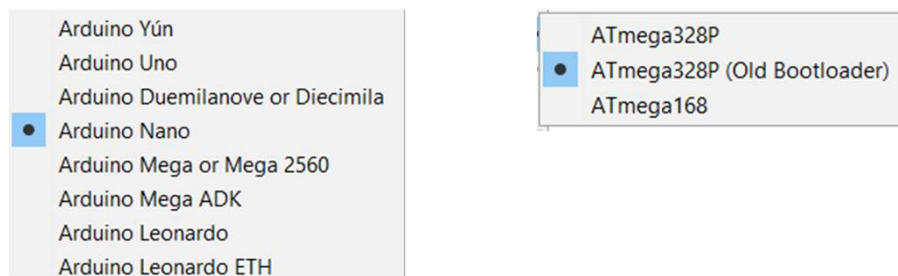


Arduino IDE and Arduino

Arduino IDE

Arduino IDE (<https://www.arduino.cc/en/main/software>) is the programming environment that we will use to program the robot, although you can use others. The program includes all the libraries that we will use by default, so it does not need a special configuration.

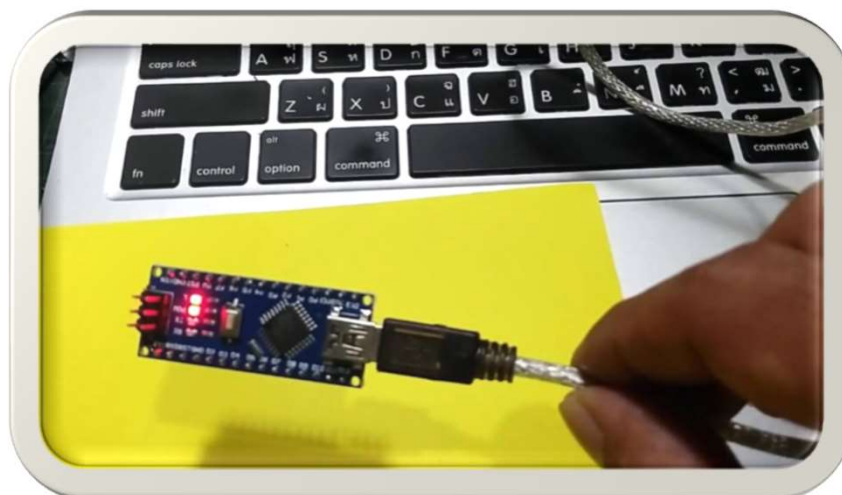
Select the Arduino Nano board in **Tools** → **Board**. In **Tools** → **Processor**, you must take into consideration that if you are using a cloned version, you must use the ATmega328P (old bootloader) version. If the Arduino board is original, then you must use ATmega328P.



Once the board and the processor have been selected, you must select the communication port in **Tools** → **Port**. Connect the Arduino to the PC and a new serial port will appear that you must select. If no new port appears, then it probably means that you have not properly installed the driver. In the cloned versions of Arduino, the CH340G chip is used, so you must install the driver for this chip following the instructions indicated here, if that's the case: <https://sparks.gogo.co.nz/ch340.html>

Uploading Code through USB Port

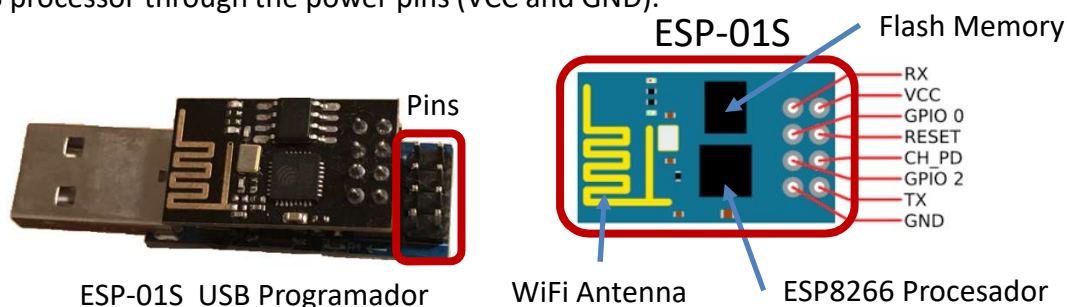
Connect the USB cable, with an empty program (with the "setup" and "loop" functions that come by default), try uploading the code to the Arduino board by accessing **Program** → **Upload**. Usually, it is convenient to previously compile the program, to verify that you do not have any errors in the code. In that case, to compile the program (do not upload the code), go to **Program** → **Verify/Compile**.



Arduino IDE and ESP-01S

Programming with the ESP-01S

The ESP-01S is a development board based on the ESP8266 processor. In order to program this development board it is convenient to use a USB programmer. The ESP-01S must be connected to the USB programmer as shown in the figure. The USB programmer has duplicated the connector, so that on the one hand it allows you to connect the ESP-01S to the programmer and at the same time it allows you to connect the device to other devices through the processor signals (RX, TX, GPIO0 and GPIO2) and to supply external power the ESP-01S processor through the power pins (VCC and GND).



Arduino IDE

Arduino IDE (<https://www.arduino.cc/en/main/software>) is the programming environment that we will use to program the robot, although you can use others. We must previously configure Arduino IDE to be able to program the ESP8266 processors. To do this, you must access **File** → **Preferences** and within the **Additional Boards Manager URLs**, add the following URL: http://arduino.esp8266.com/stable/package_esp8266com_index.json. Then, in **Tools** → **Board** → **Board Manager**, find and install the latest version of the ESP8266 boards.

esp8266 by ESP8266 Community
Tarjetas incluidas en este paquete
Generic ESP8266 Module, Generic ESP8285 Module, ESPDuino (ESP-13 Module), Adafruit Feather HUZZAH ESP8266, Invent One, XinaBox CW01, ESPresso Lite 1.0, ESPresso Lite 2.0, Phoenix 1.0, Phoenix 2.0, NodeMCU 0.9 (ESP-12 Module), NodeMCU 1.0 (ESP-12E Module), Olimex MOD-WIFI-ESP8266(-DEV), SparkFun ESP8266 Thing, SparkFun ESP8266 Thing Dev, SparkFun Blynk Board, SweetPea ESP-210, LOLIN(WEMOS) D1 R2 & mini, LOLIN(WEMOS) D1 mini Pro, LOLIN(WEMOS) D1 mini Lite, WeMos D1 R1, ESPino (ESP-12 Module), ThaiEasyElec's ESPino, WifiInfo, Arduino, 4D Systems gen4 IoT Range, Digistump Oak, Wifiduino, Amperka WiFi Slot, Saeed Wio Link, ESPECTRO Core, Schirmilabs Eduino WiFi, ITEAD Sonoff, DOIT ESP-Mx DevKit (ESP8285).
[More Info](#)

2.7.4 ▾

In order to use the Servo controller board, you need to install Adafruit PWM Servo Driver library. You can install this library from **Program** → **Include Library** → **Manage Libraries** and search for '**PWM Servo**'.

Adafruit PWM Servo Driver Library by Adafruit Versión 2.4.0 **INSTALLED**
Adafruit PWM Servo Driver Library Adafruit PWM Servo Driver Library
[More info](#)

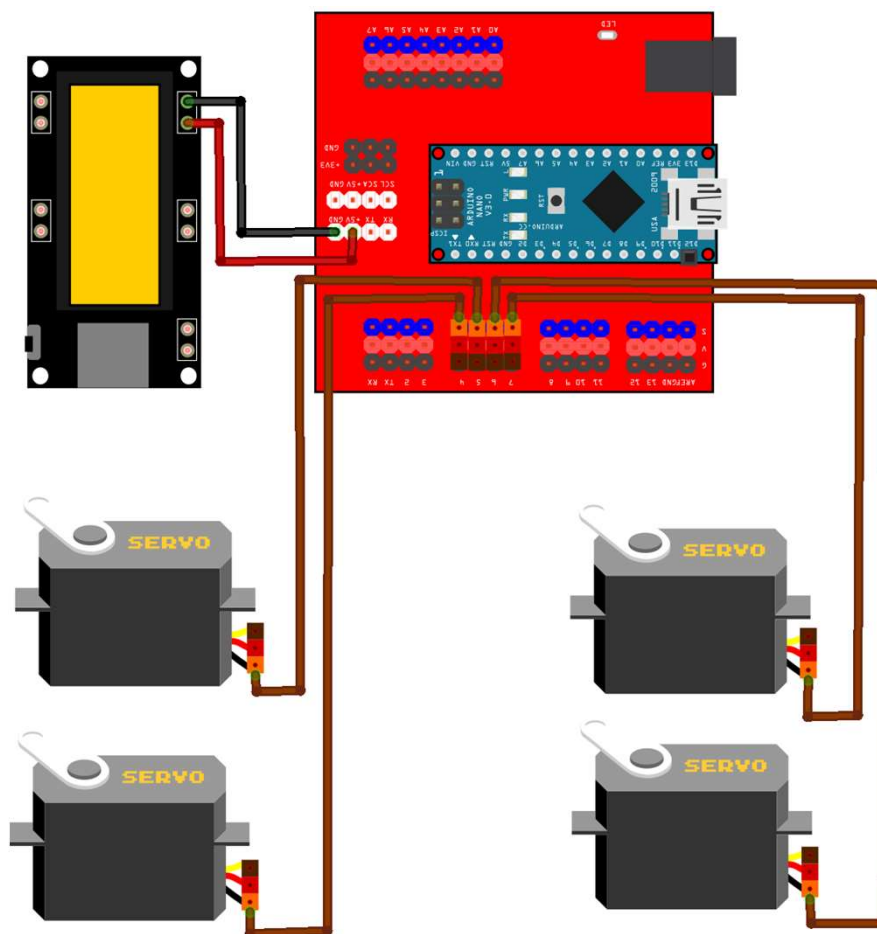
Seleccione versión ▾

Uploading Code through USB Port

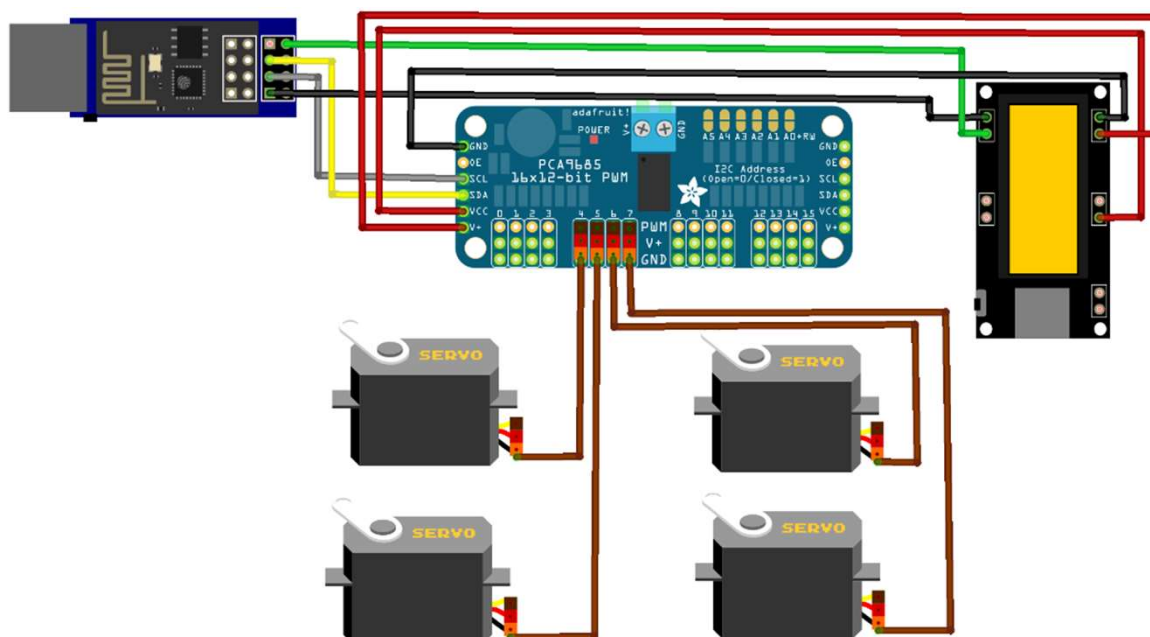
Connect the USB cable, with an empty program (with the "setup" and "loop" functions that come by default), try uploading the code to the Arduino board by accessing **Program** → **Upload**. Usually, it is convenient to previously compile the program, to verify that you do not have any errors in the code. In that case, to compile the program (do not upload the code), go to **Program** → **Verify/Compile**.

Electronics Connection

Arduino Connection Diagram



ESP-01S Connection Diagram



Electronics Connection

Arduino Connection

Component	Arduino PIN
Servo motor 1	7
Servo motor 2	6
Servo motor 3	5
Servo motor 4	4
Powerbank (5V and GND)	+5V
	Gnd

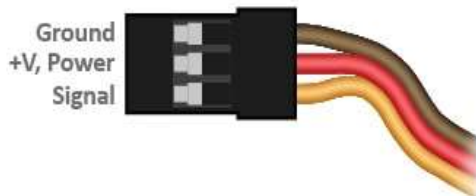
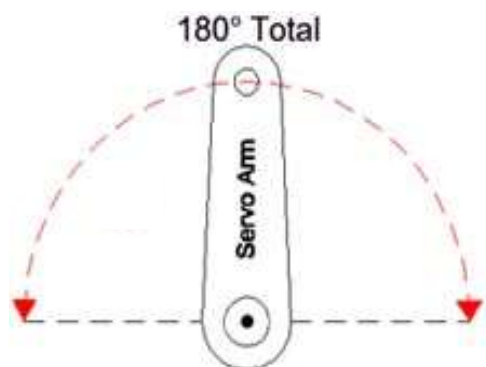
PCA9685 and ESP-01S Connection

PCA9685 Connection	PIN
Servo motor 1	7
Servo motor 2	6
Servo motor 3	5
Servo motor 4	4
I2C (serial bus connected to ESP-01)	SCL
	SDA
5V Power for PCA9685 IC (from Powerbank)	VCC
5V Power signal to servos	V+
Ground (any GND signal of the Powerbank)	GND

ESP-01S Connection	PIN
I2C (bus serie con PCA9685)	GPIO0/SDA
	GPIO2/SCL
3.3V Power (from Powerbank)	Vcc
Ground (any GND signal of the Powerbank)	GND

Servos

Previous to assembling the robot, it is convenient to set all servos at 90° position. This is an intermediate position between 0° to 180° servo range. Brown cable is for ground signal and yellow for signal, while red is for Vcc (a voltage between 4V and 6V approx.).



If you need to move the servo horn manually, please, be very careful, because the servo gears are usually made of plastic and be easily broken if a high torque is applied to move the horn.

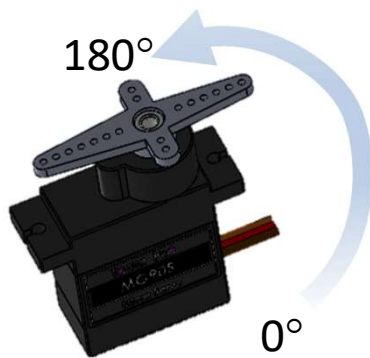
Servo Control

Controlling Servos Using the Servo Library

It is a well-known library integrated in Arduino IDE to control servos. To include the library in your code, use the **#include** directive with the header file "*Servo.h*".

To control a servo, you must create an instance of a global variable with type **Servo**, and define the pin that it is attached to with the **attach** method. If you want to disconnect the servo (to avoid unnecessary power consumption), you can use the **detach** instruction.

To move the servo from one position to another one, you can use the **write** method, indicating the position in degrees where we want to move the servo. Look at the picture to understand how servo rotate.



Los servos tienen un tiempo de ciclo de 20ms, con lo que no debéis mandar comandos de movimiento por debajo de este periodo. Utilizad instrucciones de retardo (**delay**) para garantizar que se cumple con este periodo.

```
#include <Servo.h>

Servo servos[12]; //Max number of digital signals

#define SERVO4_PIN          4
#define SERVO4_OFFSET      0
#define SERVO4_MIN_POS     0
#define SERVO4_MAX_POS    180

typedef struct
{
    uint8_t pin;
    int offset;
    int min_pos;
    int max_pos;
} RobotServo_t;

RobotServo_t servo4={SERVO4_PIN, SERVO4_OFFSET, SERVO4_MIN_POS, SERVO4_MAX_POS};

void setup() {
    writeServo(servo4,90); //Sets the position of the servo to 90°
    delay(1000);
    detachServo(servo4);
}

void loop() {
}

void writeServo(RobotServo_t &servo, int angle)
{
    angle=constrain(angle+servo.offset,servo.min_pos,servo.max_pos);
    servos[servo.pin].attach(servo.pin);
    servos[servo.pin].write(angle);}

void detachServo(RobotServo_t &servo)
{
    servos[servo.pin].detach();
}
```

Control de Servos

Servo Control with PCA9685 PWM Controller

The following code sets the position to 90° of a servo connected to pin 4 to the PCA9685 PWM controller. At the beginning, it creates an instance of a **Adafruit_PWMServoDriver** object with the **pwm** variable, which is in charge of communicating with the **PCA9685 PWM Controller**.

The controller is connected through I2C to the processor, in this case, SDA is connected to GPIO0, while SCL is connected GPIO2 of the ESP-01S board.

We have defined a new type **RobotServo_t** with all necessary values to control a servo. Indeed, the variable **servo4** is an instance of this type containing all servo-related information so that the **writeServo** function can move the servo. If we want to disconnect a servo, we can use the **detachServo** function.

```
#include <math.h>
#include <Arduino.h>
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>

Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();

#define I2C_SDA_PIN          0
#define I2C_SCL_PIN          2
#define MIN_PWM              130
#define MAX_PWM              570
#define FREQUENCY            50
#define SERVO4_PIN           4
#define SERVO4_OFFSET        0
#define SERVO4_MIN_POS       0
#define SERVO4_MAX_POS       180

typedef struct
{
    uint8_t pin;
    int offset;
    int min_pos;
    int max_pos;
} RobotServo_t;

RobotServo_t servo4={SERVO4_PIN, SERVO4_OFFSET, SERVO4_MIN_POS,
SERVO4_MAX_POS};
void setup() {
    Wire.pins(I2C_SDA_PIN,I2C_SCL_PIN);
    Wire.begin(I2C_SDA_PIN,I2C_SCL_PIN);
    pwm.begin();
    pwm.setPWMPFreq(FREQUENCY);
    yield();
    writeServo(servo4,90); //Sets the position of the servo to 90°
    delay(1000);
    detachServo(servo4);
}
void loop() {
}

void writeServo(RobotServo_t &servo, int angle)
{
    int pulse_width;
    angle=constrain(angle,servo.min_pos,servo.max_pos);
    pulse_width = map(angle+servo.offset, 0, 180, MIN_PWM, MAX_PWM);
    pwm.setPWM(servo.pin,0,pulse_width);
}

void detachServo(RobotServo_t &servo)
{
    pwm.setPWM(servo.pin,0,0);
}
```

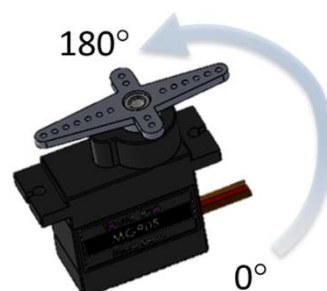
Servos have a cycle time of 20ms, so you should not send movement commands below this period. Use **delay** instruction to ensure this period is met.

Servo Calibration

The provided code includes two constants defining the minimum and maximum pulse width of the servo, **MIN_PWM** and **MAX_PWM**. These values must be calibrated in case of an incorrect operation, because the **pwm** object only understands pulses, while the **writeServo** requires an angle.

Calibration procedure:

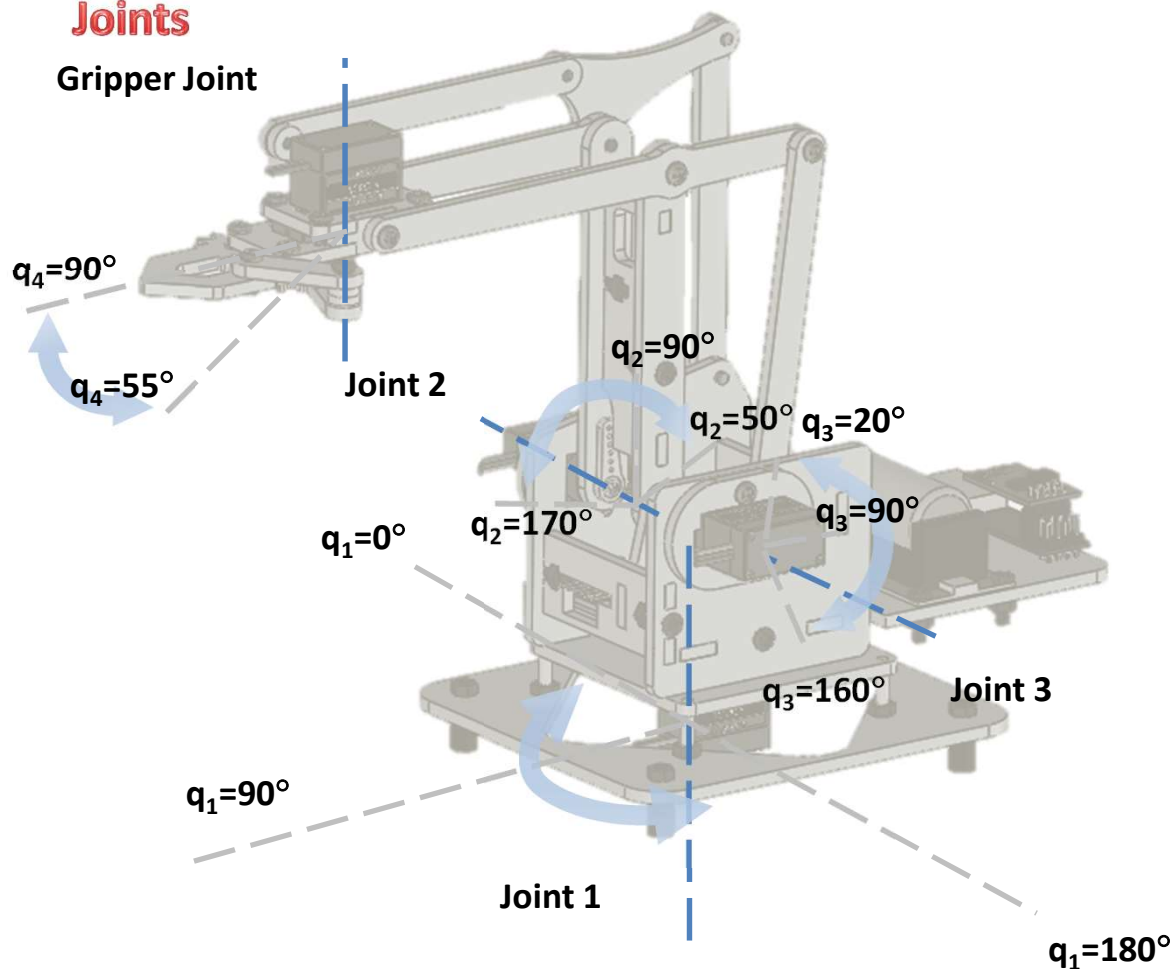
- Set the servo at position 0° and decrease the value of **MIN_PWM** until the servo stops moving from its previous position.
- Set the servo at position 180° and increase the value of **MAX_PWM** until the servo stops moving from its previous position.



meArmUPV Robot

Joints

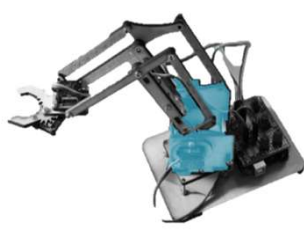
Gripper Joint



Links



Link 0



Link 1



Link 2



Link 3 (left)



4 bar mech.



Parallelogram
(link 2)

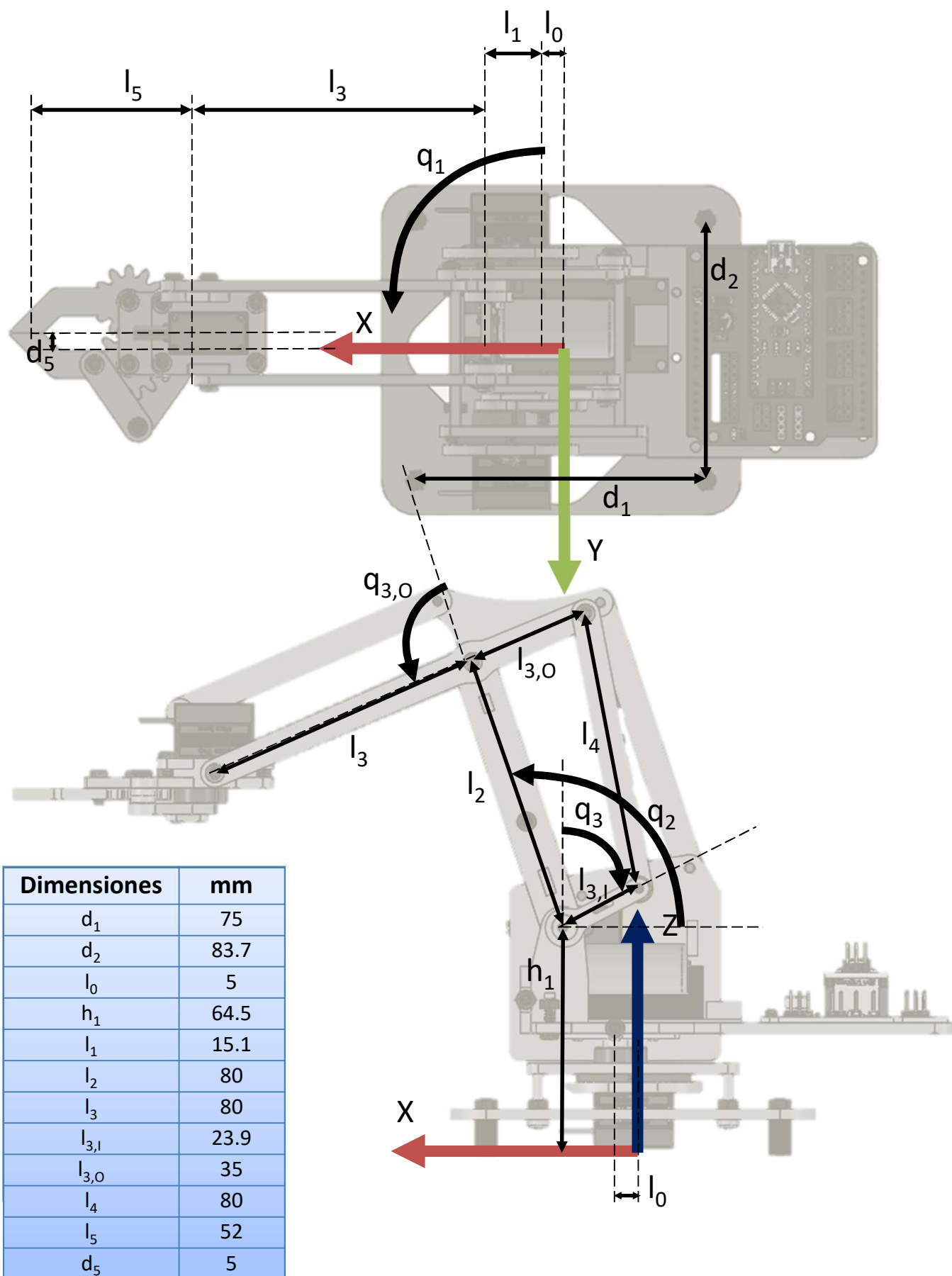


Parallelogram
(link 3)

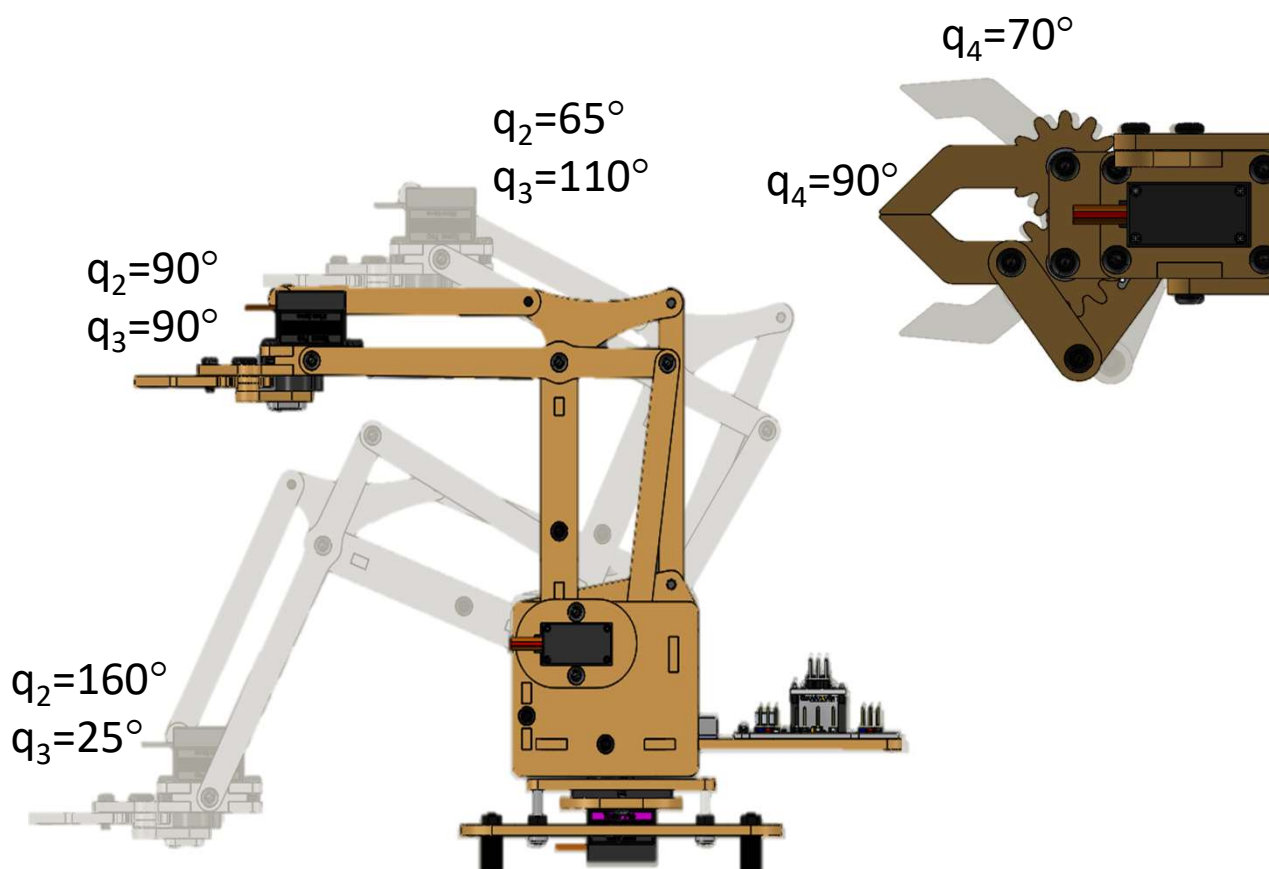
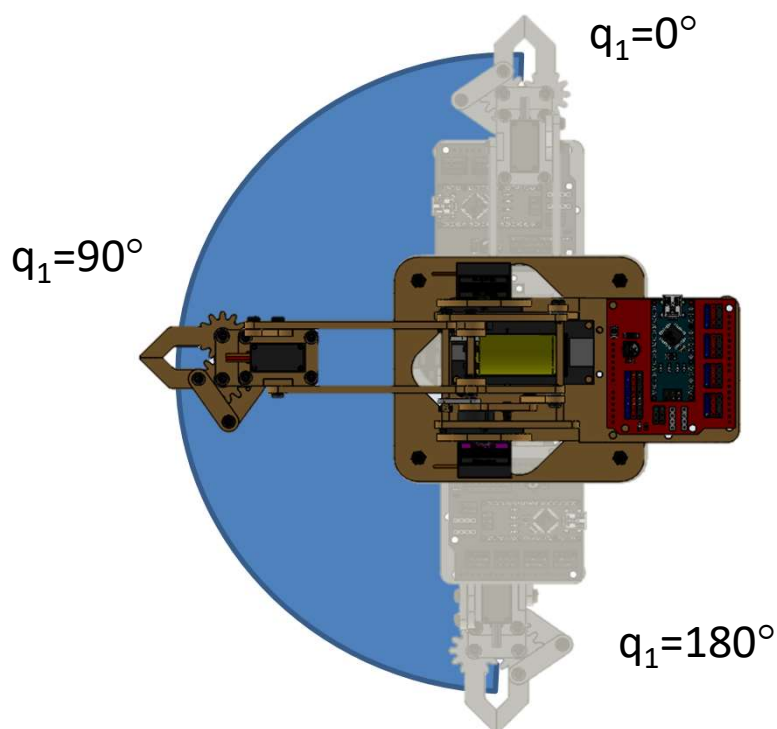


Gripper

Robot Dimensions



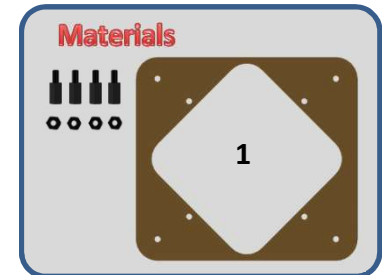
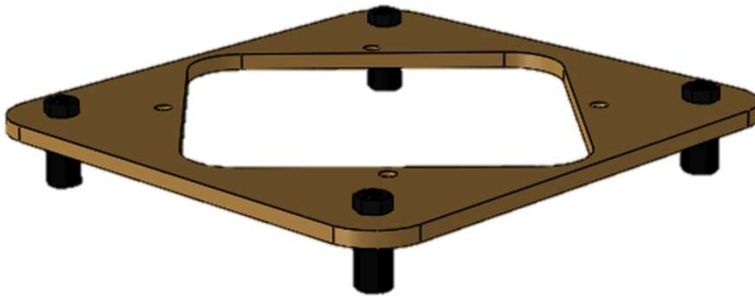
Robot Movements



Assembly Instructions

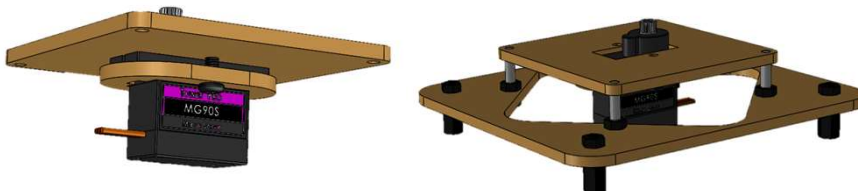
Step 1

Fix four M3x10mm spacers to the base (1) of the robot with M3 nuts.



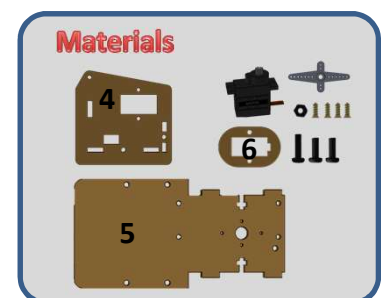
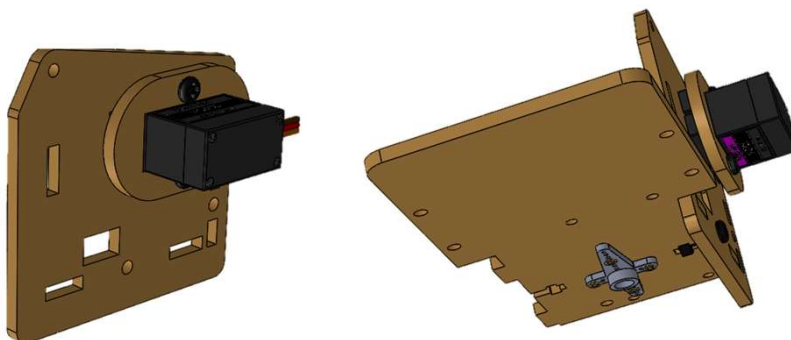
Step 2

Now screw a servo (SG90 or MG90S) to the base of servo 1. Use two M3x8mm screws and a servo bracket (2) for fixing. Then, fix the assembly to the robot base (3) with four M3x16mm (metal) screws and their corresponding M3 nuts. To fix the height, the screws are screwed against the wood itself. **Note that the direction of the servo cable should be at the front of the robot.**



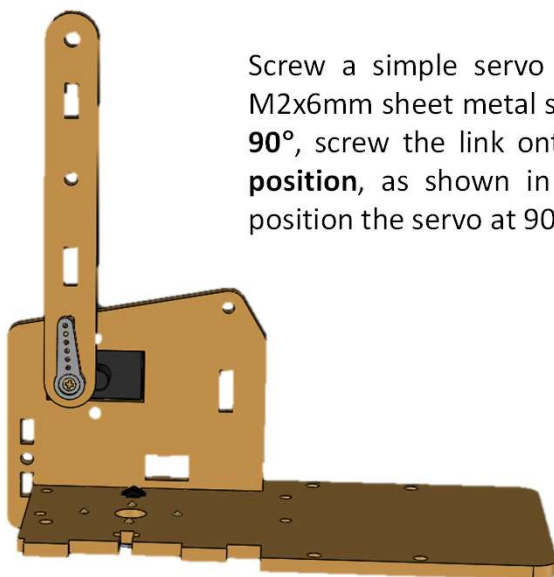
Step 3

Fix a servo (SG90 or MG90S) to the right side of link 1 (4). Use two M3x8mm screws and a servo bracket (6) to fix it (the screws are screwed against the wood itself). **Note that the direction of the servo cable should be as shown in the figure.** Then, screw the base of link 1 (5) to the right side with a M3x10mm screw and a M3 nut. At the bottom of link 1, screw the double servo horn with four M2x6mm sheet metal screws.



Assembly Instructions

Step 4



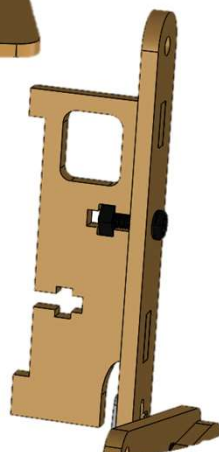
Screw a simple servo handle to the right link 2 (7) with an M2x6mm sheet metal screw. Then, with the **servo positioned at 90°**, screw the link onto the servo so that it is in the **vertical position**, as shown in the figure. If you don't know how to position the servo at 90°, please see the **Servo Control** section.

Materials

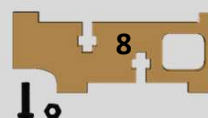


Step 5

Now, screw the support for link 2 (8) to the right link 2 (7), as shown in the figure. Use an M3x10mm screw and M3 nut.



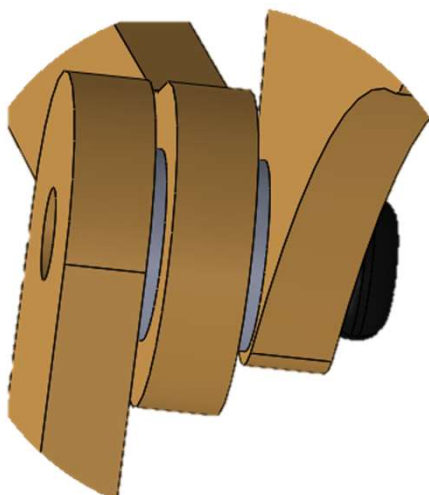
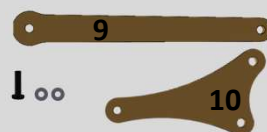
Materials



Step 6

Screw the lower right beam of link 3 (9) and the triangle to the right link 2 (10) from the outside. When screwing, use two washers between the pieces to reduce friction (see the figure with the enlarged view). The screw is screwed against the wood itself. The M3x10mm screw must be tight enough so that the robot does not have too much slack, but at the same time, the parts must rotate easily.

Materials

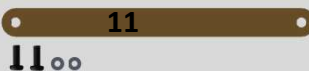


Assembly Instructions

Step 7



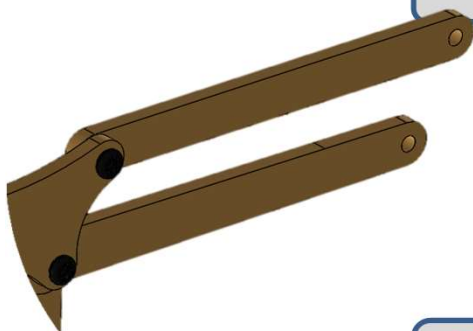
Materials



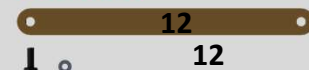
Screw in a simple beam (11) to form the four bar mechanism of the rear right side that joins link 2 (7) to link 1 (3). The screws are M3x8mm (screwed against the wood itself). Use washers so that is tight enough so that the robot does not have too much slack, but at the same time, the parts should rotate easily. Make sure you use one of the beam with two holes, one of them slightly larger that allows the screw to pass easily (there are two of them identical).

Step 8

Similarly, now screw in the beam that has a slightly smaller hole (12) and a larger one (link 3 above). Screw a M3x8mm screw against the wood itself and use a washer to obtain a low friction movement and avoid excess of slack.



Materials

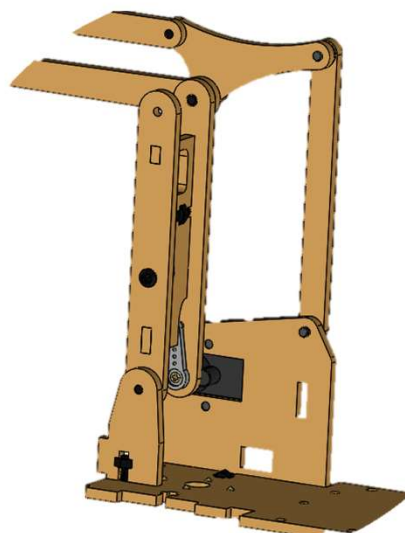


Step 9

Screw the left link 2 (13) and the vertical support (14) that joins the link 2 (7) with the link 1 (3). Use a small M3x6mm screw with a washer against the wood itself. Subsequently, screw, using M3x10mm screws and M3 nuts, the two pieces on one side to link 1 (3) and on the other side to the support of link 2 (7). Make sure that when screwing the M3x6mm screw that it has no slack, but allows movement.



Materials

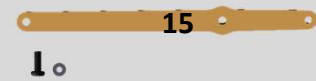


Assembly Instructions

Step 10



Materials

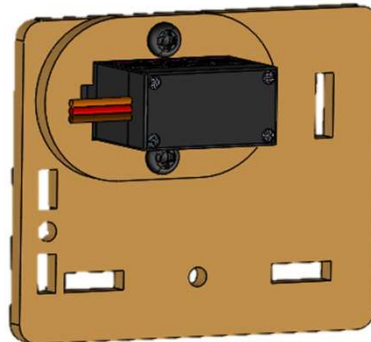


Screw in link 3 left (15), the longest beam of all. Use a M3x8mm screw and washer and screw it against the wood itself. Make sure it has no slack, but with the least possible friction.

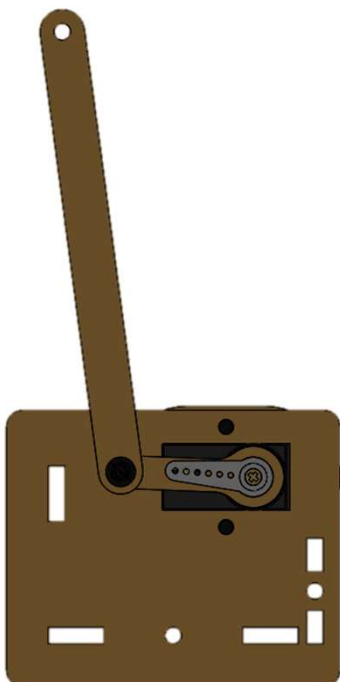
Step 11

Now assemble the left side of link 1 (16), along with a servo bracket (17) and a servo (SG90 or MG90S). Observe the direction of the servo cables for the correct assembly. Screw two M3x8mm screws against the wood itself.

Materials



Step 12



Materials



Screw a servo horn to the rocker of the four-bar mechanism of link 3 (19) with an M2x6mm sheet metal screw. Then, **with the servo positioned at 90°**, screw the rocker to the servo so that it is in a **horizontal position**, as shown in the figure with another M2x6mm sheet metal screw. If you don't know how to position the servo at 90°, please see the **Servo Control** section. Finally, screw the link 3 four-bar mechanism coupler (18) with an M3x8mm screw and washer. It leaves little slack, but enough that it turns freely.

Assembly Instructions

Step 13

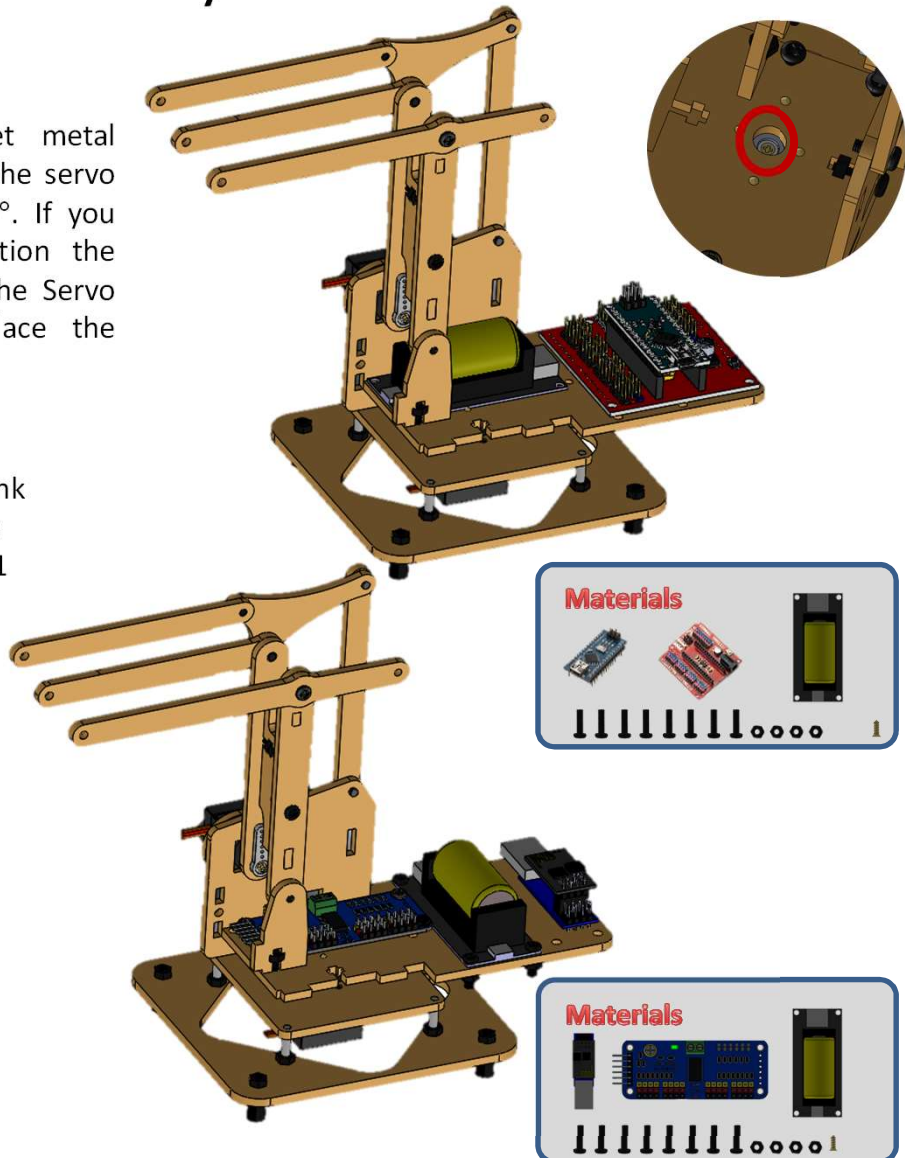
Using an M2x6mm sheet metal screw, screw link 1 (5) to the servo of link 1 positioned at 90°. If you don't know how to position the servo at 90°, please see the **Servo Control section**. Then place the electronics on link 1

With Arduino

The Arduino Nano Powerbank and expansion board can be screwed to the base of link 1 with M3x8mm screws and nuts.

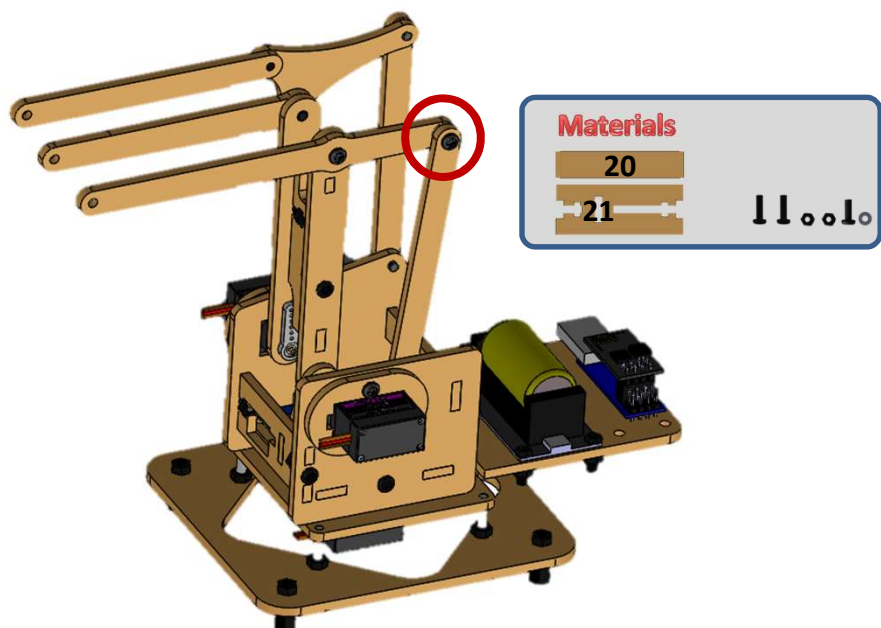
With ESP-01S

PCA9685 servo controller and 16340 powerbank can be screwed with M3x8mm screws. The USB programmer for ESP-01S cannot be screwed, so you will need to use glue (with the hot glue gun).



Step 14

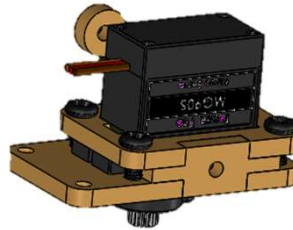
Place the front support of link 1 (21) and screw it with M3x8mm screws and M3 nuts, joining the assembled parts of the right side with the left side. The rear link 1 part (20) does not need screws. Finally, screw with a M3x8mm screw, to the wood itself, and a washer the left link 3 (15) with link 2 (7).



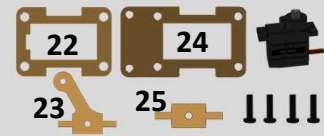
Assembly Instructions

Step 15

Insert a servo (SG90 or MG90S) into the gripper base (24) and, using four M3x8mm screws, screw the base together with the gripper servo bracket (22), the gripper small beam (23) and the small adapter with a hole (25). Pay attention to the servo cable, must be in the front of the gripper.

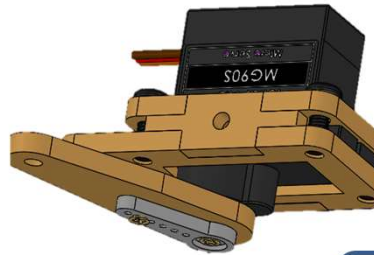


Materials



Step 16

Now screw a simple servo horn to the gripper crank (26) and then attach it to the servo with M2x6mm screws. The **servo must be positioned at 90°**. If you don't know how to position the servo at 90°, please see the section **Servo Control**.

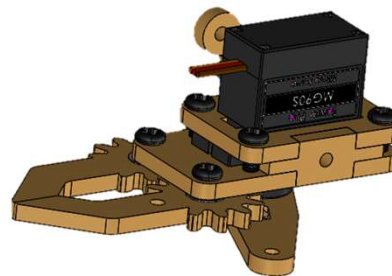


Materials



Step 17

Now screw the fingers of the gripper (27 and 28) with M3x8mm screws and washers. Position the fingers as shown in the figure, with the fingertips practically touching at the 90° position.



Materials

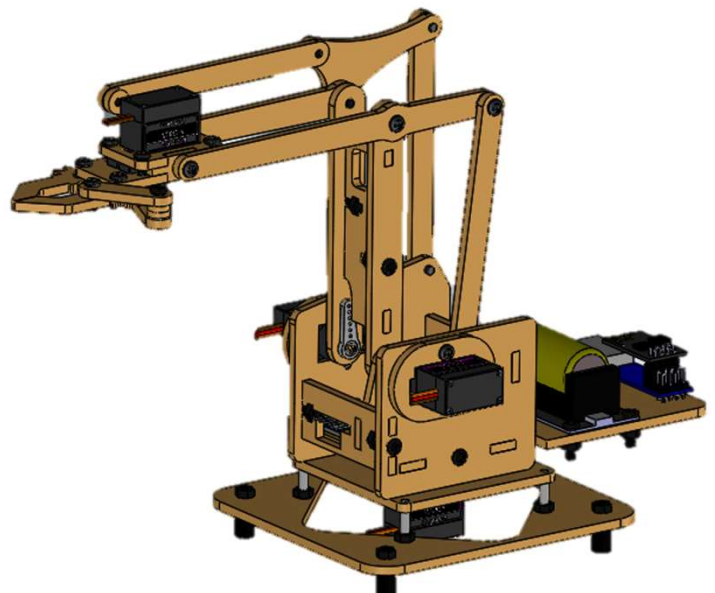


Step 18

Screw the gripper coupler (29) between the crank (26) and the left finger (27). Use a M3x8mm screw to screw in the finger and a M3x10mm screw to screw in the crank. Use washers between the wooden pieces to have less friction and the round spacer (30) as a spacer. Then, screw the gripper to the rest of the robot with M3x8mm screws.

Congrats! You're done!

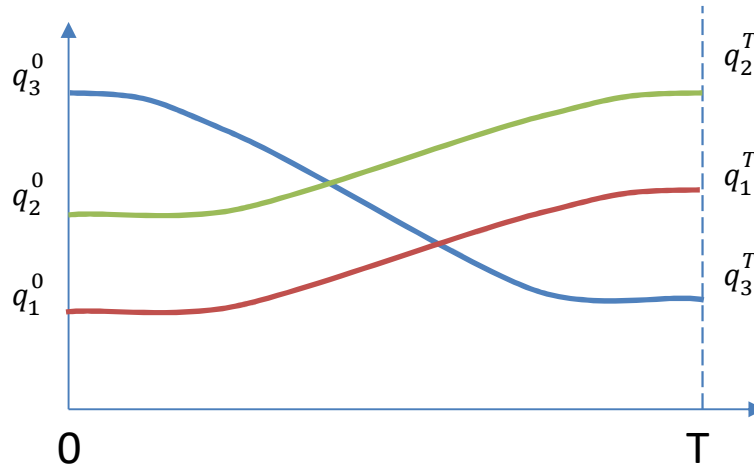
Materials



Axis Control

Polynomial Trajectory Axis Control

In axis control mode, the robot must move to a joint configuration determined by the target values in a given time. In this mode, the joints are coordinated, in the sense that each of them tries to reach a target position in the configuration space, but all at the same time.



We define a smooth trajectory, $q_i(t)$, in the configuration space with a polynomial expression (the derivative of the trajectory is $\dot{q}_i(t)$) and we set the contour conditions, $q_i(0)$ and $q_i(T)$ and their derivatives, $\dot{q}_i(0)$ y $\dot{q}_i(T)$. In particular, for a cubic trajectory, we can set as contour conditions for the trajectory the initial and final configurations q_i^0 and q_i^T , respectively; and the initial and final speeds will be set to zero. Thus, the cubic expression is defined as follows:

Trajectory (position) $q_i(t) = a_i t^3 + b_i t^2 + c_i t + d_i$

Trajectory (speed): $\dot{q}_i(t) = 3a_i t^2 + 2b_i t + c_i$

Contour conditions: $q_i(0) = q_i^0 \quad q_i(T) = q_i^T$
 $\dot{q}_i(0) = 0 \quad \dot{q}_i(T) = 0$

From contour conditions, we can get the values of the cubic trajectory parameters:

$$a_i = -\frac{2(q_i^T - q_i^0)}{T^3} \quad b_i = \frac{3(q_i^T - q_i^0)}{T^2} \quad c_i = 0 \quad d_i = q_i^0$$

The parameters of the path are different for each servo and must be recalculated each time the start, end position, speeds or the time of the path is modified.

Axis Control

Evaluating Time in a Trajectory

In order to evaluate a trajectory over time, we must know the time of the microprocessor clock. In this case, the most appropriate instruction in Arduino code is **millis()**, which provides the clock time in milliseconds. On the other hand, to ensure that we evaluate the trajectory at least every 20ms, which is the minimum time that the servo needs to generate the control signal, we must use the **delay(...)** instruction, which will generate a wait for the time indicated in milliseconds. Keep in mind that when calling the **millis()** function for the first time, the return time does not have to be zero.

```
#define JOINTS 3

RobotServo_t robotServos[JOINTS]={{...},{...},{...}};

void moveAbsJ(const RobotServo_t servos[JOINTS], const
double q0[JOINTS], const double qT[JOINTS], const double
T);

double q0[JOINTS]={...};
double qT[JOINTS]={...};

void setup()
{
  ...
  for (int i=0;i<JOINTS;i++)
    writeServo(robotServos[i],(int)q0[i]);
  delay(3000);
  moveAbsJ(...);
  delay(1000);
  for (int i=0;i<JOINTS;i++)
    detachServo(robotServos[i]);
  ...
}

void moveAbsJ(const RobotServo_t robotServos[JOINTS],
const double q0[JOINTS], const double qT[JOINTS], const
double T)
{
  //TODO
  //Compute trajectory parameters
  //Get the initial timestamp
  //Compute current joint values and move the servos those
  positions. Wait at least 20ms and repeat until the time of the
  trajectory is due.
}
```

The following code is to be completed and can be used to implement the axis-by-axis control of a polynomial path. At the beginning, define the number of axes to be controlled with the **JOINTS** constant, which in our case is equal to 3. Furthermore, the parameters for each of the robot's joints must be properly initialized in the **robotServos** variable, it is an array with the parameters of the three axes to move of the type **RobotServo_t**. Adequate values must also be provided to the variables **q0** and **qT** that represent the initial and final configuration in degrees, it is an array with the values of type **double**. The code positions the servos in the initial position, waits a while and then calls the **moveAbsJ** function, which is the one that must implement the polynomial trajectory with the information provided. The parameter **T** of the **moveAbsJ** function represents the time in seconds of the trajectory.

Task

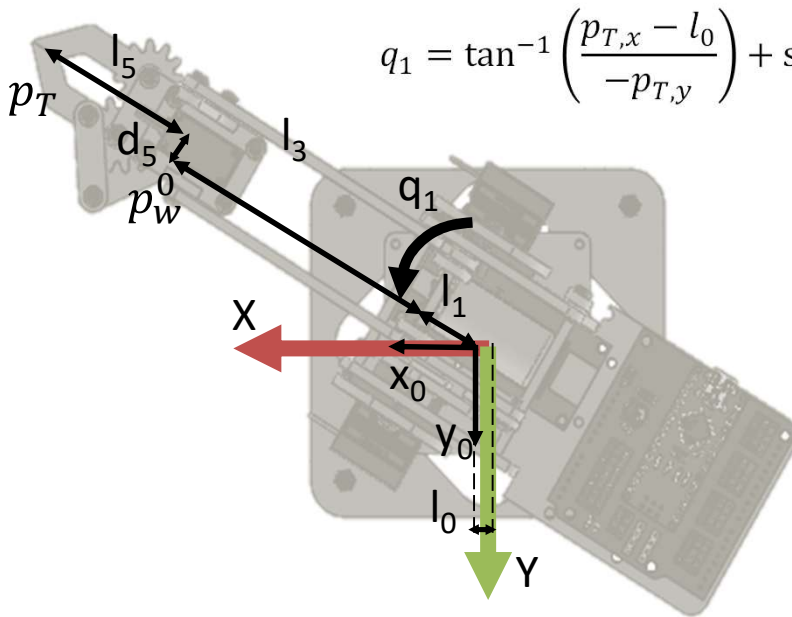
Move the robot to three consecutive configurations with a Point To Point trajectory (cubic) control starting from home configuration. Once the configurations have been reached, move the robot back to the home configuration.

Point To Point Control

Inverse Kinematics

If the input information is given in the Cartesian space (a position in the Cartesian space), then we must use the inverse kinematics to compute the joint configurations. The computation will be done in four steps:

1) Joint q_1 is computed from coordinates x and y of gripper point p_T . Due to the non-zero term d_5 , the computation of q_1 must be obtained from the sum of two angles: the first as a consequence of the angle of p_T w.r.t. the reference frame 0 (shown in the figure). Then, we need to compensate such angle due to distance d_5 from a simple trigonometric relation. On the other hand, the point of the gripper wrist with respect to the robot, p_w^0 , can be computed as:



$$q_1 = \tan^{-1} \left(\frac{p_{T,x} - l_0}{-p_{T,y}} \right) + \sin^{-1} \left(\frac{d_5}{\sqrt{(p_{T,x} - l_0)^2 + p_{T,y}^2}} \right)$$

$$p_w^0 = p_T + \begin{bmatrix} l_5 \sin q_1 - l_0 \\ -l_5 \cos q_1 \\ 0 \end{bmatrix}$$

2) Joint coordinates q_2 and $q_{3,0}$ are computed from the triangle formed by links 2 and 3 of a conventional robot arm:

$$r = \sqrt{p_{w,x}^2 + p_{w,y}^2} - l_1$$

$$z_e = p_{w,z} - h_1$$

$$\alpha = \tan^{-1} \frac{z_e}{r}$$

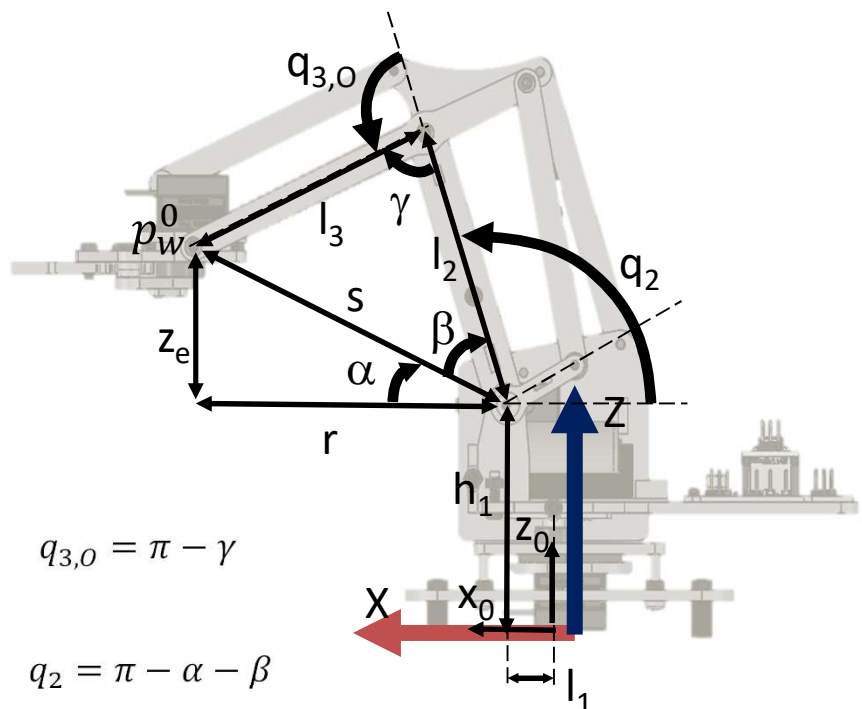
$$s = \sqrt{r^2 + z_e^2}$$

$$\gamma = \cos^{-1} \left(\frac{l_2^2 + l_3^2 - s^2}{2l_2l_3} \right)$$

$$q_{3,0} = \pi - \gamma$$

$$\beta = \cos^{-1} \left(\frac{s^2 + l_2^2 - l_3^2}{2sl_2} \right)$$

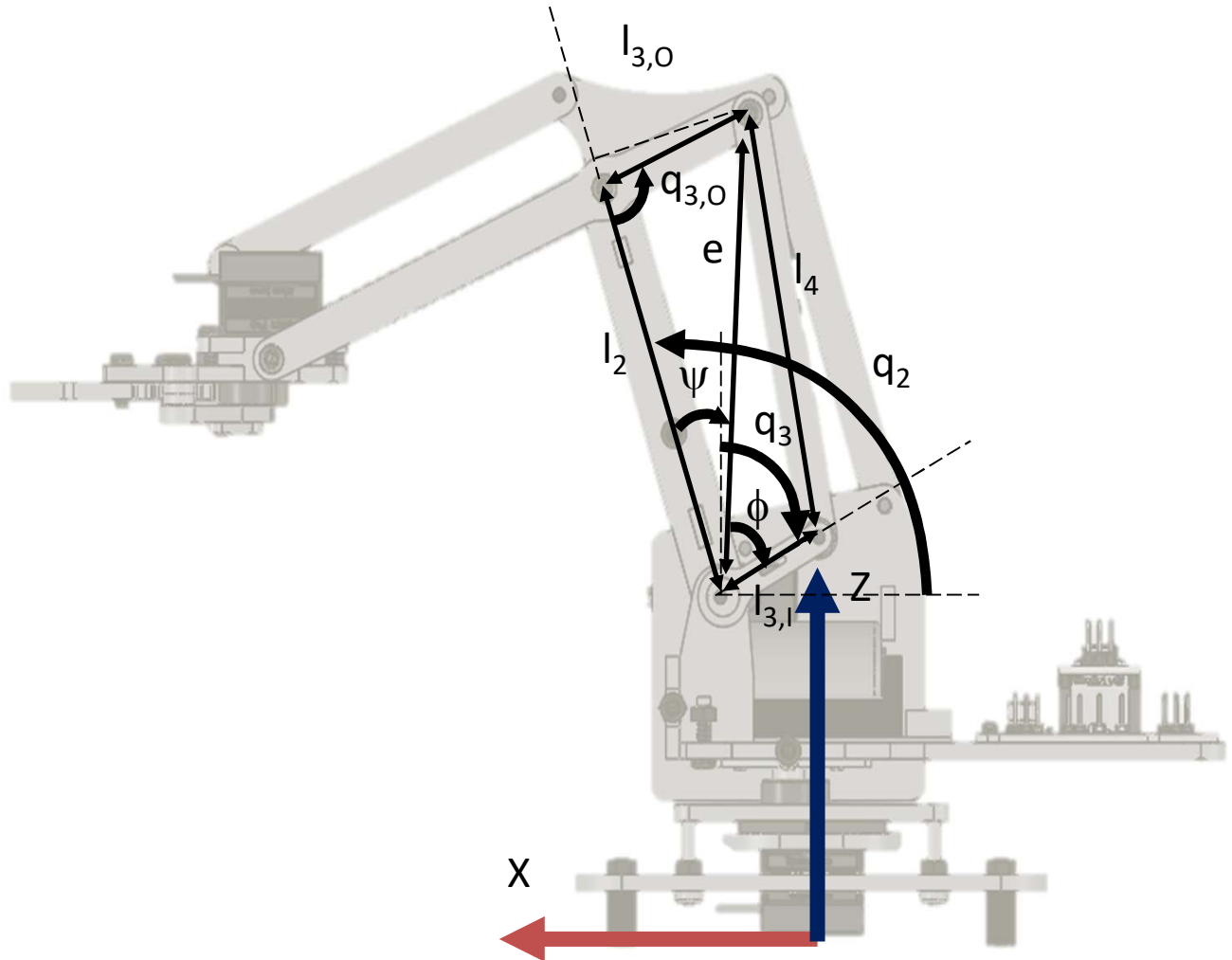
$$q_2 = \pi - \alpha - \beta$$



Point To Point Control

4 Linkage Mechanism ($q_{3,o} \rightarrow q_3$)

3) Finally, the angle of the third joint q_3 is computed from the inverse kinematics of the four-bar mechanism:



$$e = \sqrt{l_{3,o}^2 + l_2^2 - 2l_{3,o}l_2 \cos q_{3,o}}$$

$$\psi = \sin^{-1} \left(\frac{l_{3,o} \sin q_{3,o}}{e} \right)$$

$$\phi = \cos^{-1} \left(\frac{e^2 + l_{3,I}^2 - l_4^2}{2el_{3,I}} \right)$$

$$q_3 = \psi + \phi + \frac{\pi}{2} - q_2$$

Point To Point Control

Implementing the Inverse Kinematics

The following code proposes the definition of a new data type in a struct to include all required robot geometric parameters, **RobotParams_t** and it also proposes another data type to indicate a target position to reach in **RobotPosition_t**. Also, defines the function **inverseKin** so that from a target configuration and the robot parameters, returns a goal configuration to reach (in an array). Function **moveJ** combines the inverse kinematic with the point to point trajectory control.

```
...
typedef struct
{
    double l0;
    double h1;
    double l1;
    double l2;
    double l3;
    double l3l;
    double l3O;
    double l4;
    double l5;
} RobotParams_t;

typedef struct
{
    double x;
    double y;
    double z;
} RobotPosition_t;
...
RobotParams_t params={...,...,...,...};
RobotPosition_t pT={...};

void inverseKin(const RobotPosition_t &target,const RobotParams_t &params, double *q);
void moveJ(const RobotServo_t robotServos[JOINTS], const double q0[JOINTS], const RobotPosition_t target, const float T, const RobotParams_t &params);
...
void moveJ(const RobotServo_t robotServos[JOINTS], const float q0[JOINTS], const RobotPosition_t target, const float T, const RobotParams_t &params)
{
    //TODO: Call inverseKin to compute a target configuration qT, then call moveAbsJ
}
void inverseKin(const RobotPosition_t &target,const RobotParams_t &params, double *q)
{
    //TODO: Return in q the values of the configuration for the given target position.
}
```

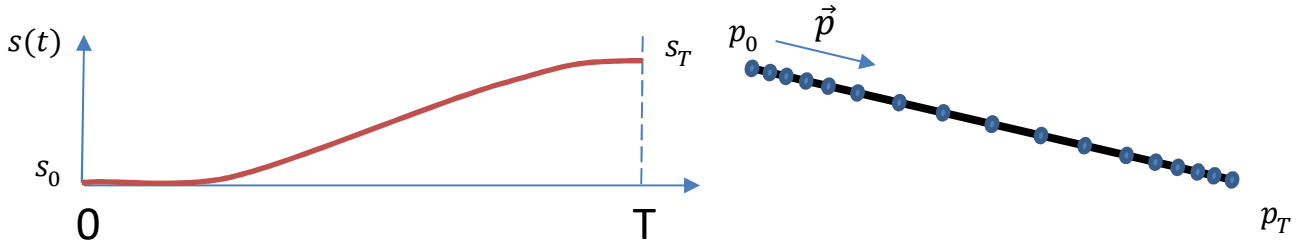
Task

Compute three robot configurations from three target positions provided in mm and move the robot to those configurations using the Point to Point trajectory control. Start from the home position, where all servos are in a 90° position.

Linear Control

Linear Control with Polynomial Trajectory

Linear motion control with a trajectory can be done with a cubic trajectory, where the trajectory displacement $s(t)$ is computed as before. The actual trajectory is computed from a unitary vector \vec{p} from the initial position, p_0 , that reaches the final position p_T . The initial speed will be zero at the beginning and the end of the trajectory, while in the intermediate points the speed will be greater, as shown in the following figure:



Therefore, the lineal trajectory is defined as: $p(t) = s(t)\vec{p} + p_0$

$$\begin{aligned} s(t) &= at^3 + bt^2 + ct + d & s(0) &= 0 & \dot{s}(0) &= 0 \\ \dot{s}(t) &= 3at^2 + 2bt + c & s(T) &= \|p_T - p_0\|_2 & \dot{s}(T) &= 0 \end{aligned}$$

with

$$a = -\frac{2\|p_T - p_0\|_2}{T^3} \quad b = \frac{3\|p_T - p_0\|_2}{T^2} \quad c = 0 \quad d = 0 \quad \vec{p} = \frac{p_T - p_0}{\|p_T - p_0\|_2}$$

Being $\|\cdot\|_2$ the Euclidean distance operator:

$$\|p_T - p_0\|_2 = \sqrt{(p_{T,x} - p_{0,x})^2 + (p_{T,y} - p_{0,y})^2 + (p_{T,z} - p_{0,z})^2}$$

Forward Kinematics

To implement the linear control you need to compute the position of the gripper from a given configuration, that is, the servo angles. As before, we explain the required computations to implement the forward kinematics of the robot.

The separation of the wrist with respect to the joint 1 axis is computed as follows (in the figure, when $q_2 = \frac{\pi}{2}$ and $q_{3,0} = \frac{\pi}{2}$, the distance value is $l_1 + l_3$):

$$r = -l_2 \cos q_2 - l_3 \cos(q_2 + q_{3,0})$$

On the other hand, the height of the gripper is computed from:

$$p_{T,z} = h_1 + l_2 \sin q_2 + l_3 \sin(q_2 + q_{3,0})$$

The x and y coordinates of the gripper are:

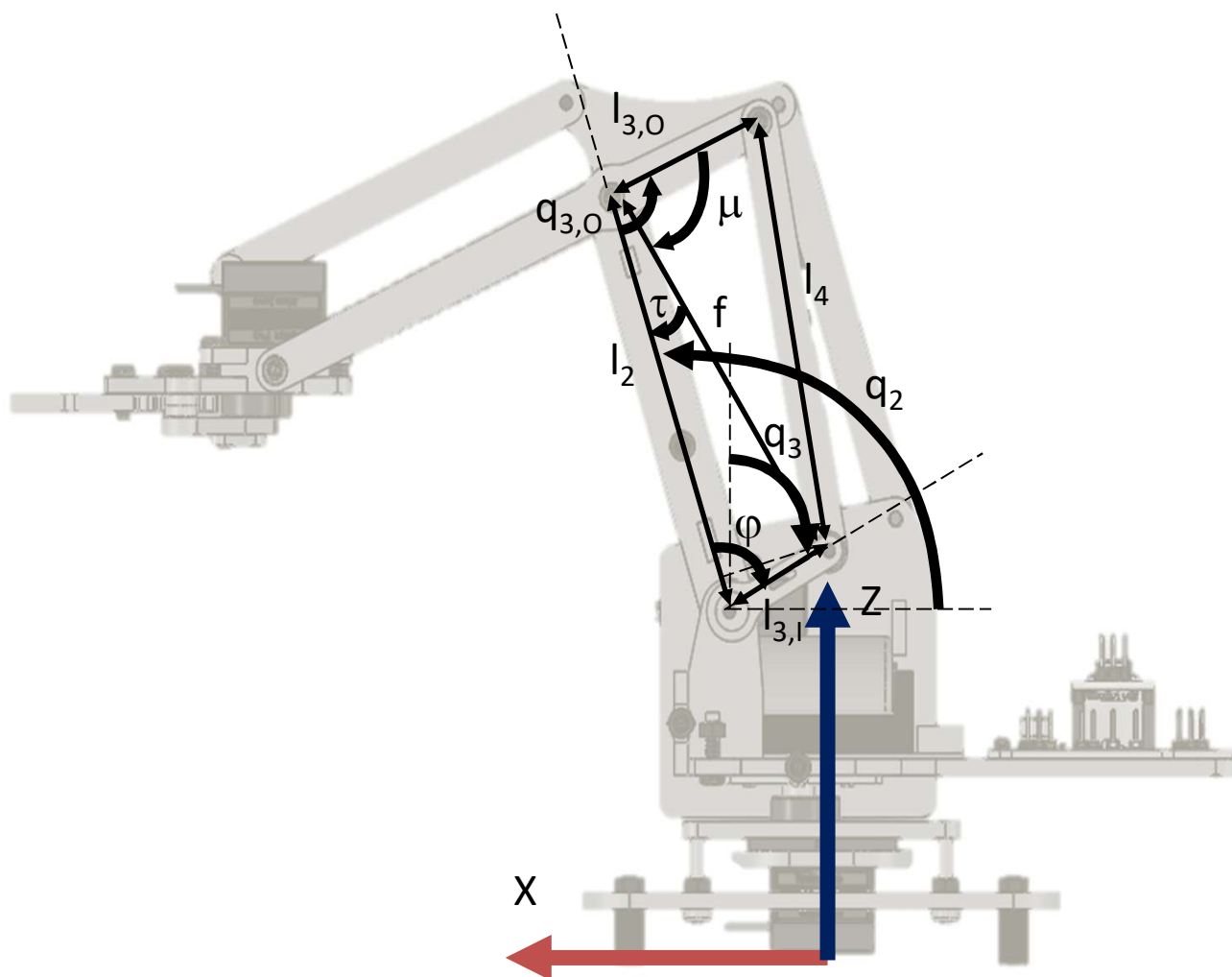
$$p_{T,x} = l_0 + (r + l_1 + l_5) \sin q_1 - d_5 \cos q_1$$

$$p_{T,y} = -(r + l_1 + l_5) \cos q_1 - d_5 \sin q_1$$

Linear Control

4 Linkage Mechanism ($q_3 \rightarrow q_{3,0}$)

Link 3 angle, q_3 , is computed from the 4 linkage mechanism:



$$\varphi = q_3 + q_2 - \frac{\pi}{2}$$

$$f = \sqrt{l_{3,I}^2 + l_2^2 - 2l_{3,I}l_2 \cos \varphi}$$

$$\mu = \cos^{-1} \left(\frac{f^2 + l_{3,0}^2 - l_4^2}{2fl_{3,0}} \right)$$

$$\tau = \sin^{-1} \left(\frac{l_{3,I} \sin \varphi}{f} \right)$$

$$q_{3,0} = \tau + \mu$$

Lineal Control

Implementing Lineal Control

The following code proposes an implementation of two functions **forwardKin** y **moveL**. Function **forwardKin** must implement the forward kinematics of the robot, while function **moveL**, must implement the linear control with a cubic trajectory.

```
...
void forwardKin(const double q[JOINTS], const RobotParams_t &params, RobotPosition_t *target);
void moveL(const RobotServo_t robotServos[JOINTS], const double q0[JOINTS], const RobotPosition_t target, const float T, const
RobotParams_t &params);
...
void moveL(const RobotServo_t robotServos[JOINTS], const float q0[JOINTS], const RobotPosition_t target, const float T, const
RobotParams_t &params)
{
    //TODO: Call forwardKin to compute the current gripper position from the given configuration. Then compute a target position and
    move the gripper along the trajectory by sending proper servos commands using the inverse kinematics.
}
void forwardKin(const double q[JOINTS], const RobotParams_t &params, RobotPosition_t *target)
{
    //TODO: Return in target the values of the position of the gripper.
}
```

Task

Implement now the three consecutive movement using a linear control. At the end, the robot must return to the initial configuration, also with a linear movement.

Circular Movement

Circular Control

In order to move the robot with a circular movement in the 3D Cartesian space, we need the position of three points. The first point, P_0 , corresponds to the actual position of robot, while the other two points, P_1 and P_2 must be given. Obviously, the points must met some conditions so that they belong to a 3D circle, such as they must not be aligned or repeated, so we will assume that these conditions are met (checking those conditions is out of the scope). The procedure is actually quite simple, we need to define a reference frame with origin P_0 and with axis X pointing towards the direction $\overrightarrow{P_{01}} = P_1 - P_0$ and axis Z perpendicular to vectors $\overrightarrow{P_{01}}$ and $\overrightarrow{P_{02}} = P_2 - P_0$. By doing this, we can express the original points in the new reference frame, where the z coordinate is zero and therefore all points will be contained in a 2D plane. If we extract the circle center \tilde{C} and undo the transformation to obtain the original 3D center, C . Once the center is obtained, we can create a trajectory starting from P_0 and finishing in P_2 passing through point P_1 . The temporal implementation for this trajectory can use a polynomic expression for the travelled distance.

The maths with all these computations are explained below. The transformation of the points is expressed with the following homogenous transformation matrix:

$$T = \begin{bmatrix} X & Y & Z & P_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

with $X = \frac{\overrightarrow{P_{01}}}{\|\overrightarrow{P_{01}}\|_2}$, $Z = \frac{\overrightarrow{P_{01}} \times \overrightarrow{P_{02}}}{\|\overrightarrow{P_{01}} \times \overrightarrow{P_{02}}\|_2}$, $Y = Z \times X$, being \times the cross product.

The points transformed are:

$$\begin{bmatrix} \tilde{P}_1 \\ 1 \end{bmatrix} = T^{-1} \begin{bmatrix} P_1 \\ 1 \end{bmatrix}, \begin{bmatrix} \tilde{P}_2 \\ 1 \end{bmatrix} = T^{-1} \begin{bmatrix} P_2 \\ 1 \end{bmatrix}$$

In the new reference frame, the center must be located

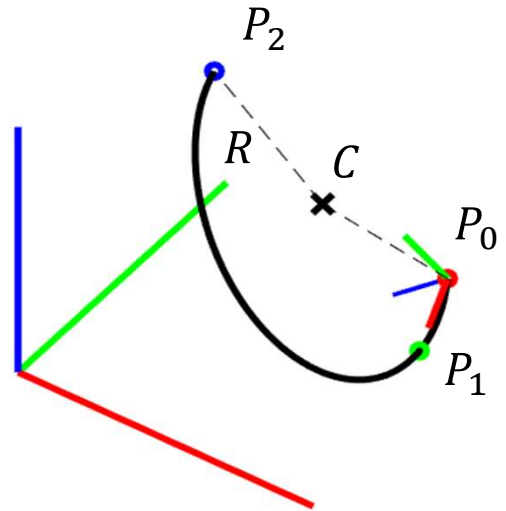
in $\tilde{C} = \begin{bmatrix} \frac{\tilde{P}_{1,x}}{2} & h & 0 \end{bmatrix}^T$, being $\tilde{P}_1 = [\tilde{P}_{1,x} \ \tilde{P}_{1,y}]^T$ the coordinates of the first point \tilde{P}_1 and h the height of the center point. From the distance relation to the center, the second point $\tilde{P}_2 = [\tilde{P}_{2,x} \ \tilde{P}_{2,y}]^T$ and the origin, we can obtain the height:

$$\left(\tilde{P}_{2,x} - \frac{\tilde{P}_{1,x}}{2} \right)^2 + (\tilde{P}_{2,y} - h)^2 = \left(\frac{\tilde{P}_{1,x}}{2} \right)^2 + h^2$$

Therefore, the position of the center in the XY plane of the new reference frame and its corresponding 3D position are computed as:

$$\tilde{C} = \begin{bmatrix} \frac{\tilde{P}_{1,x}}{2} & \frac{\left(\tilde{P}_{2,x} - \frac{\tilde{P}_{1,x}}{2} \right)^2 + \tilde{P}_{2,y}^2 - \left(\frac{\tilde{P}_{1,x}}{2} \right)^2}{2\tilde{P}_{2,y}} & 0 \end{bmatrix}^T \quad \begin{bmatrix} C \\ 1 \end{bmatrix} = T \begin{bmatrix} \tilde{C} \\ 1 \end{bmatrix} \quad R = \|\tilde{C}\|_2$$

being R , the radius of the circle, that is, the distance of the center to the origin of the new reference frame.



Circular Movement

Finally, the circular trajectory $P(t)$ is obtained from:

$$\tilde{P}(t) = R \begin{bmatrix} \cos(\theta(t)) \\ \sin(\theta(t)) \\ 0 \end{bmatrix} + \tilde{C} \quad \begin{bmatrix} P(t) \\ 1 \end{bmatrix} = T \begin{bmatrix} \tilde{P}(t) \\ 1 \end{bmatrix}$$

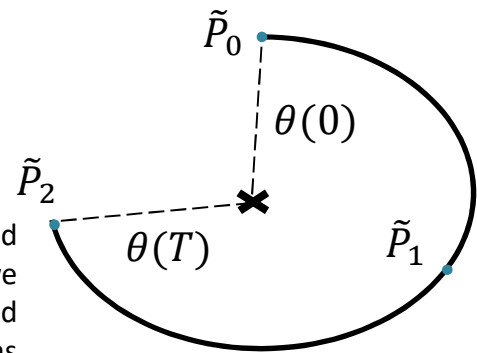
being $\theta(t)$ the angle along the circular trajectory evaluated from 0 to T. This trajectory is not necessary linear and we can demand some initial and final conditions for the speed of the trajectory using, for instance a cubic expresión as previously seen. In addition to this, we set the conditions that the trajectory must start from P_0 and end at P_2 :

$$\theta(t) = at^3 + bt^2 + ct + d \quad \dot{\theta}(t) = 3at^2 + 2bt + c$$

$$\theta(0) = \tan^{-1} \left(\frac{-\tilde{C}_y}{-\tilde{C}_x} \right) \quad \dot{\theta}(0) = 0 \quad \dot{\theta}(T) = 0$$

$$\theta(T) = \cos^{-1} \left(-\frac{(\tilde{P}_2 - \tilde{C})^T \cdot \tilde{C}}{R^2} \right) + \theta(0) \quad \text{or} \quad \theta(T) = 2\pi - \cos^{-1} \left(-\frac{(\tilde{P}_2 - \tilde{C})^T \cdot \tilde{C}}{R^2} \right) + \theta(0)$$

$$\text{with } a = -\frac{2|\theta(T) - \theta(0)|}{T^3} \quad b = \frac{3|\theta(T) - \theta(0)|}{T^2} \quad c = 0 \quad d = \theta(0)$$



Remark: One of the two solutions Will pass through point P_2 while the other condition will not. Choose the right choice for your case.

Circular Motion Implementation

Now, you are in position for implementing a function **moveC**, that generates a circular trajectory from the actual robot configuration and two target points: **target1** and **target2**, given by their Cartesian coordinates. The reminder of input arguments of the function are similar to the previous functions.

```
...
void moveC(const RobotServo_t robotServos[JOINTS], const float q0[JOINTS], const RobotPosition_t target1, const RobotPosition_t
target2, const float T, const RobotParams_t &params);
...
void moveC(const RobotServo_t robotServos[JOINTS], const float q0[JOINTS], const RobotPosition_t target1, const RobotPosition_t
target2, const float T, const RobotParams_t &params)
{
    //TODO: Compute a circular trajectory based on the current configuration, and target points 1 and 2
}
...
```

Tarea (Opcional)

Now implement a circular movement in the XY, YZ or XZ plane and with a free-choice radius, as well as the amount of rotation within the circle. If you want to make a complete circle, you must do it in two steps.