Universidade do Porto

Faculdade de Engenharia

FEUP

Four-Legged Walking Robot

Bruno Silva

Sara Carvalho

Report of the Practical Work executed in the scope of the curricular unit
Embedded Computing Architectures, of the 1st year of the
Master of Science in Electrical and Computer Engineering

February 4, 2023

We declare that this work/report is our authorship and has not been previously used in another course or curricular unit, from this or another institution. References to other authors (statements, ideas, thoughts) scrupulously respect the rules of attribution, and are duly indicated in the text and in the bibliographic references, according to the referencing rules. We are aware that the practice of plagiarism and self-plagiarism constitutes an academic offense.

In "Ethical Code of Academic Conduct", art.14, University of Porto, 2017.

Table of Contents

1 Introduction

During the course of this report, the steps taken to build and program a Four-Legged Walking Robot which is able to walk, change direction when presented with an obstacle, and compensate for its inclination, will be presented. For identity reasons, it was decided that the robot was going to be called "Albino, The Crab" due to its resemblance to the animal and its white color.

2 Material

- **Raspberry Pi Pico**

  The microcontroller used in the project was the Rasberry Pi Pico. Its chip is the RP2040, which is known for its low cost and high performance - ideal for this project.

  RP2040 stands for:

  - RP - Raspberry
  - 2 - Number os process cores
  - 0 - Type of core
  - 4 - Floor(log2(ram / 16k))
  - 0 - Non-volatile memory on board (0 means none)

  Regarding the GPIO, it has 26 multi-function pins presented in fig.1.

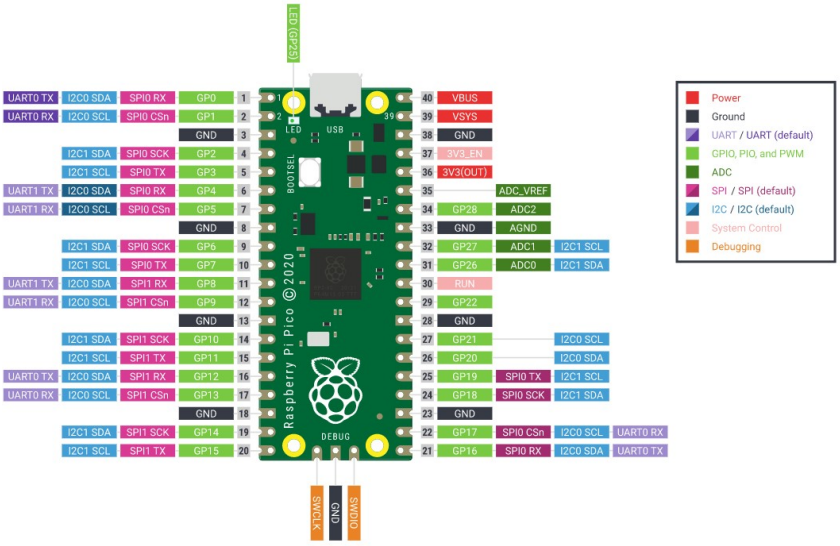


Figure 1: Pinout of the Pi Pico

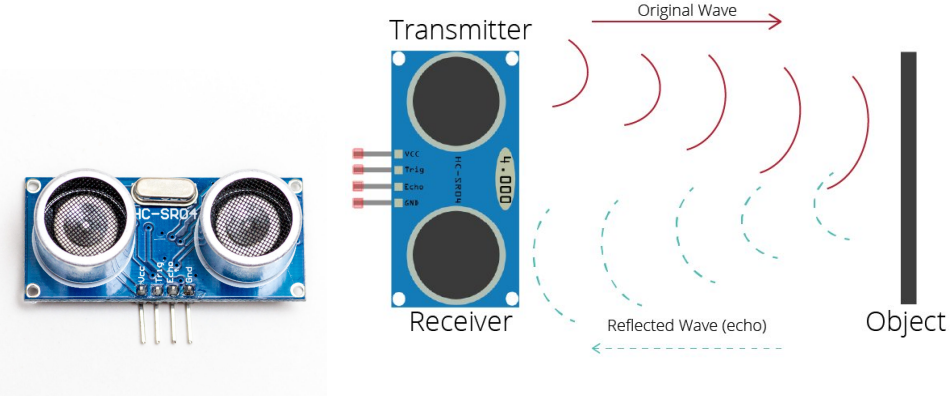
- **Servo Motor SG90 9g (x8)**

  In the project, the eight micro servo motors were essential since they allowed leg movement of the robot. In terms of hardware, each servo has 3 wires: brown (ground), red (Vcc) and yellow (PWM control signal) - all presented in fig.2. Thus, it was possible to send a PWM control signal through the Raspberry Pi Pico, in order to get the desired angle rotation. To achieve this, it was used a library "$RP2040\_PWM - 1.4.0$" by khoih-prog available in the PaltformIO libraries. Since the angle desired was relative to the position the servo was at that moment, a calibration of each servo needed to be done in order to define fixed PWM values for 0, 90 and 180 degrees.

Figure 2: Servo Motor SG90 9g

- **Ultrasonic ranging module HC-SR04**

  The HC-SR04 was used to measure the distance to the nearest object/obstacle in front of the robot. The ultrasonic sensor is a sensor that measures distances through ultrasound which travels through the air. An HC-SR04 ultrasonic distance sensor consists of two ultrasonic transducers. The transmitter converts the electrical signal into 40 KHz ultrasonic sound pulses and the receiver waits for these pulses. If the ultrasound wave hits an object or obstacle in its path, it will then be reflected back toward the sensor. Using the time between the sent (trig pin) and the received (echo pin) signals and the air velocity, the distance is obtained. The HC-SR04, in particular, has a ranging distance of $2cm$ to $400cm/4m$ and a resolution of $0.3cm$. Furthermore, the sensor works in $5V$, both in the Vcc and Echo pins. The Pico, however, works in $3V3$, which raised the need to make a simple voltage divider, using two resistances, explained deeper in the report.



(a) Sonar (b) Detection Scheme

Figure 3: Detection Unit HC-SR04

- **MPU-6050**

  The MPU-6050 is a Motion Processing Unity that can be used as an accelerometer and/or a gyroscope, each with three axes: x, y, and z. It has high precision due to its 16-bit ADC. The MPU can measure the 3 axes simultaneously. The direction of the axes can be checked directly in the MPU. It has 8 pins, however, in the project only four were used: Vcc, Ground, Serial Clock, SCL and Serial Data, SDA (upper four in fig.4). Since it was needed to obtain the tilt of the robot through the angle deviation, a way of obtaining this angle, both in the x and y axis, using the acceleration and rotation speed, was explored later in subsection 4.2.5. It uses the communication protocol I2C and its supply voltage must be between 3 and $5V$.

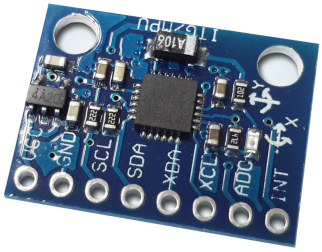


Figure 4: MPU-6050

- **Board with servo headers**

  In order to supply power to the servos used, it was used the servo driver module presented in fig.5. It has channels for 16 servos, it was used only the channels 0-7. In terms of power, it has an onboard $5V$ regulator and must be supplied with a voltage between 6 and $12V$.

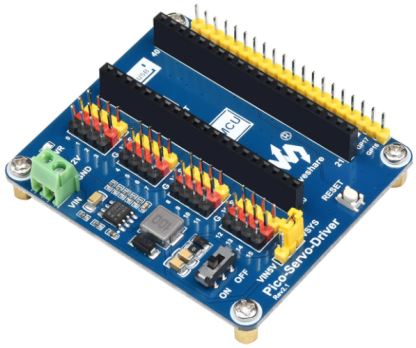


Figure 5: Motherboard used in the project

- **Rechargeable Batteries** $3.7V$ **(2x)**

  Regarding the power supply, two batteries of $3.7V$ were used with a total of $7.4V$, meeting the requirements of the board mentioned earlier. Since the testing phase would be long, getting rechargeable batteries would be favorable.

Figure 6: Batteries used in the project

- **Battery Support**

  So as to hold the batteries in place and ease the connection with the motherboard, the battery support, presented below, was used.

Figure 7: Battery Case used in the project

- **3D Printed Parts of the Robot**

  The carcass of the robot was kindly printed by the teachers at FEUP.

- **Female/Female Jumper Wires**

  It was also needed some female/female jumper wires to connect the Sonar and MPU-6050 pins to the motherboard pins.

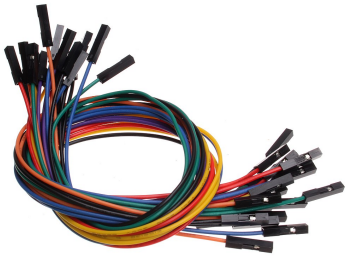


Figure 8: Female/Female jumper wires

3 The Assembling of the Robot

The robot's assembling was divided into two parts: one in which all the parts of the robot were assembled, mechanically and electrically, and another one in which all the required movements were coded and calibrated.

3.1 Hardware

This project started with the acquisition of all the components necessary.

After this, some pins were welded to the Raspberry Pi, so that it could fit in the motherboard. This unit was then screwed to the robot's body (previously printed in a 3D printer). This unit (motherboard and pico) was fundamental since it allowed the control of each servo motor separately.

Then, the leg assembling started: each leg needed two servos (previously calibrated), one for up/down movements, and one the side-to-side movements. This is presented in fig.10.

Afterward, the legs were screwed to the body and all 8 servos were connected to the motherboard. The power supply unit (batteries + batteries support) was screwed to the lower part of the body of the robot, allowing its two wires direct connection to the motherboard. In the 9, it is possible to check, in the left-down corner, the power entry (green component with screws).

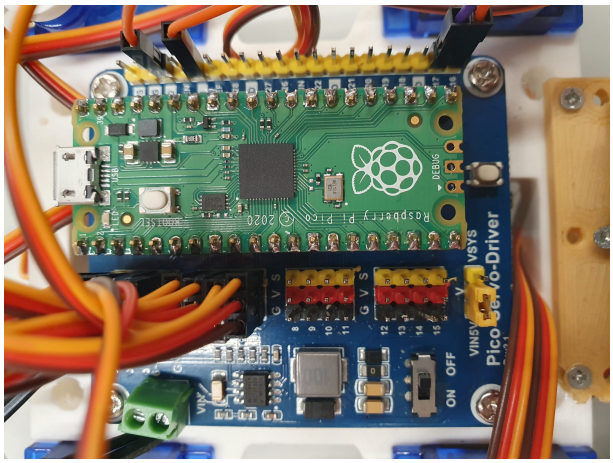
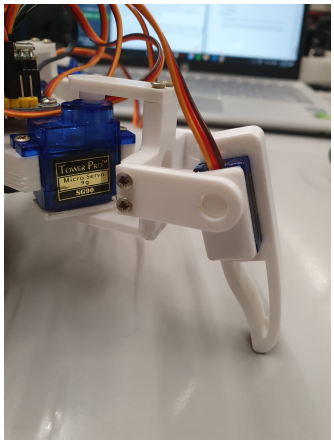
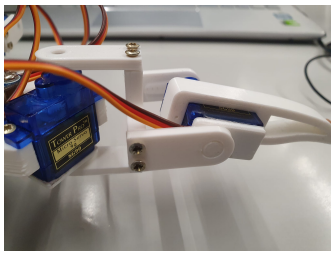


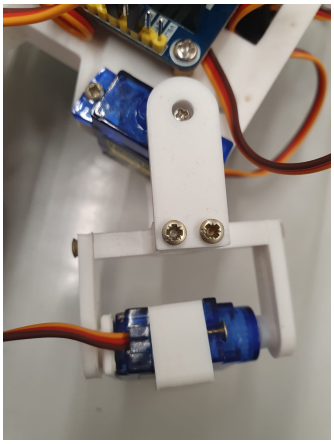
Figure 9: Motherboard connected with Pi Pico



(a) Contracted Leg



(b) Extended Leg



(c) Top Leg View

Figure 10: Leg Components

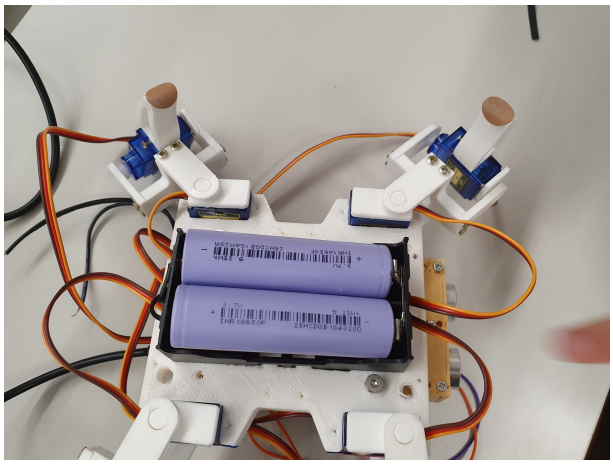
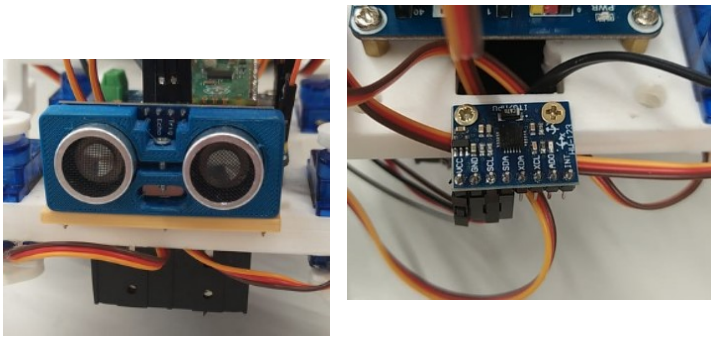


Figure 11: Supply Unit

In order to facilitate its movement, sponges were added to the robot's leg tips, visible in fig.11. The sponges increased the grip on the surfaces, avoiding a glide effect, which negatively impacted the movement.

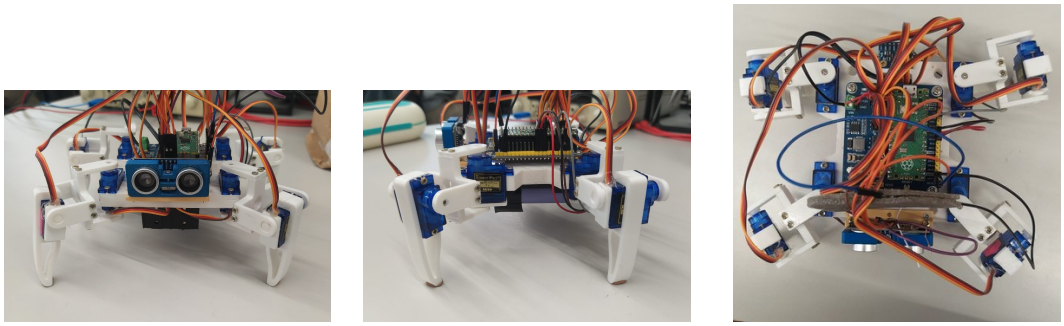
At last, the sonar and the MPU were also screwed to the body, the first one (fig.12a), at the front using a specific support, 3D-printed in advance, and the second one (fig.12b), on the back part of the main body.



(a) Sonar and respective support (b) MPU in the main body

Figure 12: Sonar and MPU

After all the assembly steps, the robot's hardware part was finished. The final build is presented below.



(a) Front View (b) Side View (c) Top View

Figure 13: Final Build

3.2 Software

For this project's code implementation, it was used State Machines for a simpler and easier approach, which fulfilled the requirements. The software used was Visual Studio Code with the PlatformIO extension that allows an easy connection between the C++ code in VSC and the Pi Pico.

3.2.1 Servo Calibration

As mentioned before, the servos needed to be calibrated, i.e. associate a duty cycle PWM value to the limit angles: 0, 90, and 180 degrees and make sure all servos had these angles in the same relative position. Thus, it was created a separate PlatformIO project only for this process.

The PWM approach used was set to have a period of $20ms$. This square wave's duty cycle was then adjusted to meet the angle desired.

It was possible to build the table 1, with all the duty cycle values (in ms) of each servo corresponding the angles. For 0 and 180 degrees, the duty cycle was calibrated directly. However, for the 90 degrees angle, linearity was assumed, i.e. its value was the average value of the other two. This method seemed adequate enough.

Servo	0	90	180
0	3.2	7.9	12.6
1	3.1	7.8	12.5
2	2.5	7.25	12
3	3.2	7.85	12.5
4	2.5	7.3	12.1
5	2.1	7.05	12
6	2.3	7.1	11.9
7	2.8	7.7	12.6

Table 1: Servo calibration duty cycles, in milliseconds.

In order to use these PWM values more easily, a code function was made. This function, named *setPWM* receives the servo index and the desired angle, in degrees, and returns the corresponding PWM value through *pwmRet*. Its code was as follows:

```
float setPWM(int servoNum, float angle){
  float pwmRet = 0;
  float cal[8][2] = {{7.9, 3.2},
                     {7.8, 12.5},
                     {7.25, 2.5},
                     {7.85, 12.5},
                     {12.1, 2.5},
                     {12, 2.1},
                     {2.3, 11.9},
                     {2.8, 12.6}};
  if(servoNum <= 3){
    if(servoNum == 0 || servoNum == 2)
      pwmRet =  cal[servoNum][0] - abs((angle/90) *
      (cal[servoNum][0] - cal[servoNum][1]));
    else if(servoNum == 1 || servoNum == 3)
      pwmRet =  cal[servoNum][0] + abs((angle/90) *
      (cal[servoNum][0] - cal[servoNum][1]));
  }
  else if(servoNum >= 4){
    if(servoNum == 4 || servoNum == 5)
      pwmRet =  cal[servoNum][0] - abs((angle/180) *
      (cal[servoNum][0]-cal[servoNum][1]));
    else if(servoNum == 6 || servoNum == 7)
      pwmRet =  cal[servoNum][0] + abs((angle/180) *
      (cal[servoNum][0]-cal[servoNum][1]));
  }
  return pwmRet;
}
```

Obs.: servos 0, 1, 2, and 3 have a 90 degrees range, whilst servos 4, 5, 6, and 7 have a 180 degrees range.

3.2.2 Walking

To fully understand the walking algorithm that supported the code written it was crucial to numerate and understand the angle range of each servo. The scheme present in fig.14 shows the servo indexing for

each leg and the fig.15a and fig.15b the angle references.

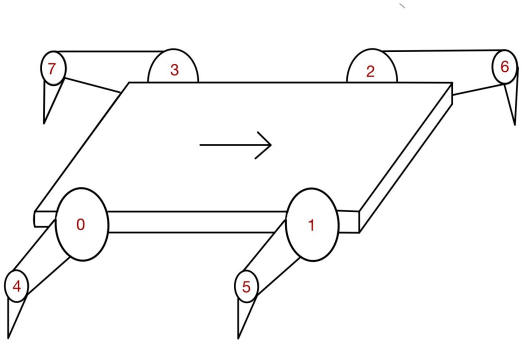
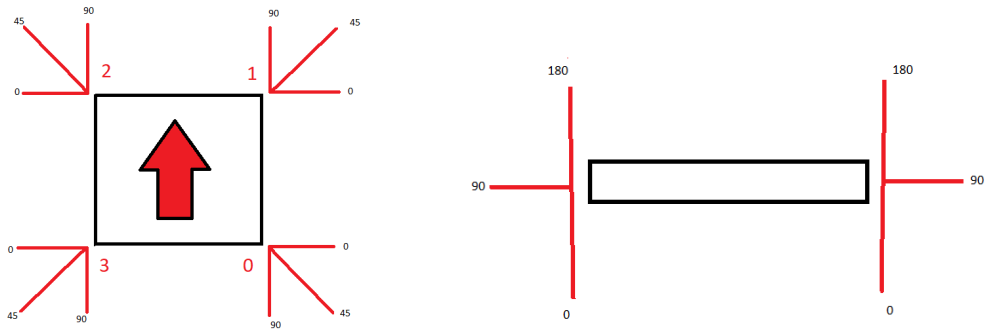


Figure 14: Numbered servos



(a) Top View Angles (b) Side View Knee Angles

Figure 15: Servos Angle Limits

The algorithm used was inspired by the steps depicted in the figure below.

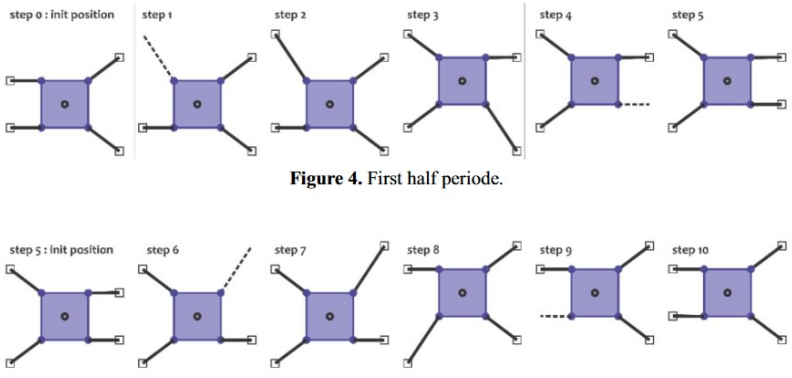


Figure 4. First half periode.

Figure 5. Second half periode.

Figure 16: Walking Algorithm

Starting with step 0, which was the initial position:

1. Servos #2 and #3 with 0 degrees and #0 and #1 with 45 degrees.
2. Leg with servo #2 is raised and positioned at 45 degrees.

3. Servo #0 goes to 60 degrees, #1 and #2 to 0 degrees and #3 to 45 degrees.
4. To end the first half of the period, servo #0 is raised and positioned at 45 degrees, and step 5 is reached.

The second period is a mirrored version of the first period:

5. Leg with servo #1 is raised and positioned at 65 degrees.
6. Servo #0 goes to 45 degrees, #1 and #2 to 0 degrees and #3 to 60 degrees.
7. To end the second half of the period, servo #0 is raised and positioned at 45 degrees, and step 5 is reached.
8. The loop is, then, concluded and step 0 was reached again.

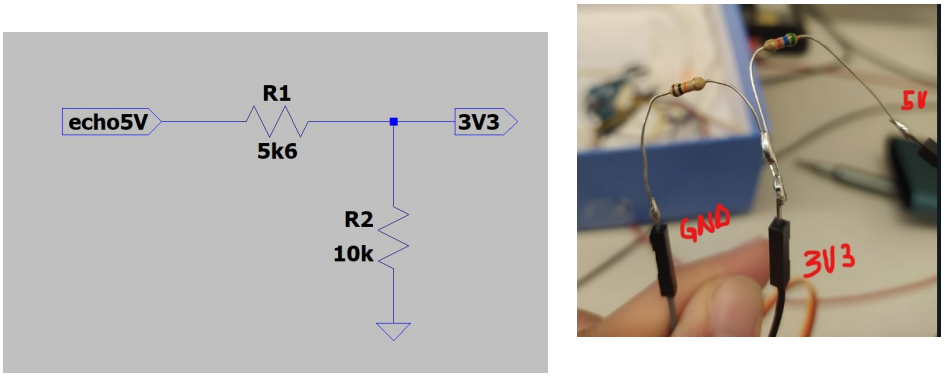
It is important to note that raising a leg means that the servo further away from the body (#4, #5, #6 and #7) must be positioned at 30 degrees - vertical movement, then the respective servo closer to the body can move - horizontal movement. A short period of time after, the servo further from the body can return to 0 degrees - vertical movement again. These vertical movement servos were called "knees" to ease the project.

In steps 2. and 5., after the raise of the front legs, they were not positioned at the same angle. This occurs because the legs behave differently, and these angle values had to be adjusted through calibration to ensure the robot walked forward with the least deviation. The values: 45 degrees for #2 and 65 degrees for #1 were the result of that calibration.

3.2.3 Object Detection

After understating the walking algorithm and constructing its code, the first sensor studied and implemented was the HC-SR04. It would be used for object/obstacle detection - the robot should stop when the sensor detects an object within a distance below a fixed value.

Using the "HCSR04.h" library, pins 16 and 17 were used as the trigger and echo pins, respectively. However, the echo pin in the sensor sends a signal with $5V$ amplitude and the Pico pins must not receive voltage values higher than $3V3$, therefore it was built a voltage divider, capable of converting the $5V$ into $3V3$. The scheme and real voltage divider are in fig.17.



(a) Voltage Divider Schematic (b) Real Voltage Divider

Figure 17: Voltage Divider

3.2.4 Turning

The robot's response to object detection implemented was as follows:

1. Detects obstacle and stops.
2. Turns 90 degrees to the right.
3. Walks for a preset amount of time.
4. Turns 90 degrees to the left, returning to the initial direction, and checks if there is still an obstacle.
5. If there is an obstacle, it goes to point 2. again. If not, it walks forward completing the object avoidance process.

The turning left or right movement was obtained by raising one leg at a time and placing it in the direction desired. After the four legs' placement, each leg returns to the 45 degrees position, forcing the main body to rotate. The figs. 18 and 19 depict this logic. This was done eight times to obtain a full 90 degrees right turn and five times for the same effect to the left. Once again these values were not the same since they are a result of calibration.

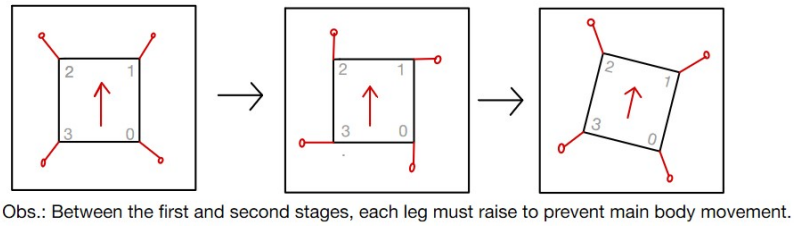


Figure 18: Algorithm for turning right

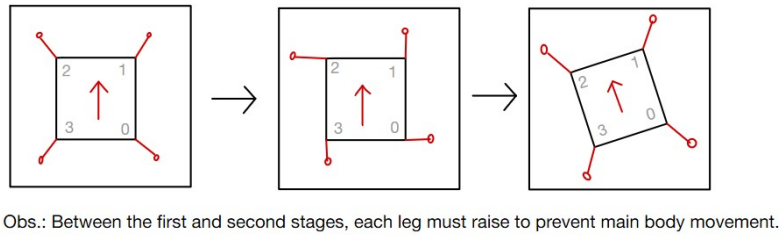


Figure 19: Algorithm for turning left

A video was recorded and posted on YouTube, displaying the robot's behavior described. Available at `https://youtu.be/Br6DCTsEOTg`.

3.2.5 Static Balance

Another functionality implemented was the ability to compensate inclination with the knee servos.

Using the MPU, which gives values of acceleration and rotational speed for the x, y, and z axes, it was possible to obtain the current tilt angle in both the x and y domains with the following formulas:

$$x = atan\left(\frac{X}{\sqrt{Y^2+Z^2}}\right)57.296 \tag{1}$$

$$y = atan\left(\frac{Y}{\sqrt{X^2+Z^2}}\right)57.296 \tag{2}$$

X, Y, and Z are the respective accelerations. The formulas calculate the tilt in radians, however, the servos used degrees, hence the multiplication for the constant 57.296, which allows passing from RAD to DEG.

Using the values obtained to directly control the servos compensation seemed to bring a lot of oscillations in the servos due to the small cycle time used. Consequently, a filter was used to prevent abrupt changes in tilt. The filter ensures that the current value of the x and y tilt depends not only on the current MPU reads but also on the previous cycle value. It was used a ratio of 9/1 ($\alpha = 0.9$), resulting in the equations:

$$angle_x = \alpha \cdot (angle_x + gyro_x \cdot dtime \cdot 57.296) + (1 - \alpha) \cdot x \tag{3}$$

$$angle_y = \alpha \cdot (angle_y + gyro_y \cdot dtime \cdot 57.296) + (1 - \alpha) \cdot y \tag{4}$$

Now that the tilt values were obtainable, the compensation strategy was thought of. Compensation with offsets was the method used.

The next figure shows the direction of the x and y rotation in relation to the robot servos.

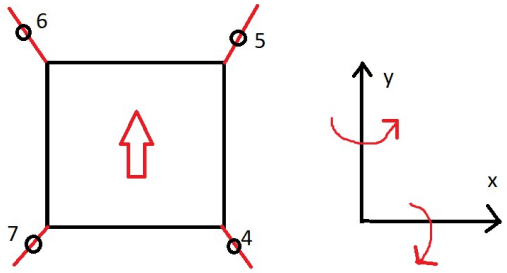


Figure 20: X and Y Tilts related to the Knees

When a tilt was detected in the direction of a pair os legs, the knees in those legs would assume that same tilt angle:

- Positive x tilt detected - knees 7 and 4.
- Negative x tilt detected - knees 6 and 5.
- Positive y tilt detected - knees 5 and 4.
- Negative y tilt detected - knees 6 and 7.

This strategy seemed to be adequate and resulted in the ability to balance when the robot was static, i.e. not walking nor turning. A video demonstrating this behavior was filmed, available on YouTube at `https://youtu.be/gY_QBBDyOj4`

3.2.6 State Machines

To end the software section, the state machines used to model all the robot's movements: walking, turning right, and turning left, are presented.

- M1: Models the walking pattern of the robot.
- M2: Controls all the other three state machines, receiving the distance from the HC-SRO4.
- M3: Models the steps involved in turning right.

- M4: Models the steps involved in turning left.

In the state machines diagram presented, there are a few details worth mentioning. A representation servo:angle was followed (X:Y means that the servo with the index X goes to the angle Y), which helped make the diagram compact yet understandable.

The yellow highlighted text means that the servo suffered an update from the previous state to the current state.

A few flags and auxiliary variables were created and used to facilitate the communications between machines:

- stop: used to stop the walking machine, is true when the current distance is under $15cm$ - the safety distance.
- flagdec: used to ensure the counter dec is decremented once per waking cycle.
- dec: counter used to define the remaining number of walking cycles performed after the turning right and before the turning left.
- distance: receives the current distance from the HC-SR04.
- set1 and set2: used to check if the right and left number of cycles are over, respectively.
- direita: used to start M3.
- esquerda: used to start M4.
- countEsq: used to start the walking between the turning right and left.
- count1 and count2: counters used to define the remaining number of turning cycles for both the right and left movement, respectively.

Due to the size of the state machines, their respective figures were placed in the Attachments - section 7.

4 Work Setbacks

Throughout the course of this project, there were many hardware and software complications. The harder to solve was the faulty material. There was a constant need to repair or change broken/obsolete components, which increased the time needed to develop the project. A section mentioning these setbacks is relevant since they were opportunities for learning even if by making mistakes.

1. Motor Servos

   The motor servos are a good example since they either broke due to the strength needed for walking/climbing or started shaking when trying to reach a position. This happened because the turning speed of the servos was too high, and it made the servo surpass the needed position, so they started shaking trying to correct the position to the one that was required.

2. Batteries

   The batteries were also a source of problems. It was challenging to find such specific batteries because they were mostly sold out, so the lab technician kindly lent them to the project.

   Since the batteries used were second-handed, the surrounding plastic started wearing out and both the batteries poles were exposed without us noticing, which caused a short circuit and consequently a small burn spot in one of the batteries, which was fixed later using an improvised plastic case.

   Also, the battery's case constantly needed to be welded due to the wires breaking.

3. Raspberry Pi Pico

   The Raspberry Pi Pico was also a source of distress as a result of the constant difficulty connecting it to the computer.

   At first, the Pico would disconnect when trying to flash the code due to overloading it with both the USB and motherboard's voltages ($3.3V$ plus $5V$). So, following that, the code was always flashed with the batteries disconnected from the motherboard.

   Still, there were still occasional issues when trying to flash the code due to problems with the Pico's drivers.

4. Sonar

   When the sonar configuration and code were complete, there would be occasional code breaks, i.e., the robot would stop in the middle of the movement process. That happened because the Pico was configured to receive $3.3V$ of signal from the sonar, but was, in fact, receiving $5V$, since that was the voltage the echo pin was sending.

   To fix that, the project needed a voltage divider, as mentioned in section 3.2.3, which solved this issue.

5 Conclusions

Initially, the difficulty of this project was underestimated. This assignment was filled with many hardware and software complications that allowed a deeper technical knowledge and understanding of how to manufacture and program a robot, from software programming to hardware construction and all the intermediate steps.

To conclude, even with all the issues that constantly delayed the process, this project was, surely, one of the most enriching projects this course has proposed.

6 Final Regards

Lastly, this project would have never been possible without the lab, technician, Carlos Grafe, who guided and helped us with our setbacks, with kindness and helpfulness. We are also thankful for the guidance of professors André Baltazar and Paulo da Costa.

References

[1] Raspberry Pi Pico: https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html

[2] RP 2040: https://www.raspberrypi.com/documentation/microcontrollers/rp2040.html#welcome-to-rp2040

[3] HC-SR04: https://www.seeedstudio.com/blog/2019/11/04/hc-sr04-features-arduino-raspberrypi-guide/

[4] HC-SR04: https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/

[5] MPU-6050: https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/

[6] MPU-6050: https://www.hobbytronics.co.uk/accelerometer-info

[7] Motherboard: https://www.waveshare.com/pico-servo-driver.htm

[8] Walking Algorithm: https://iopscience.iop.org/article/10.1088/1742-6596/1201/1/012010/pdf

7 Attachments

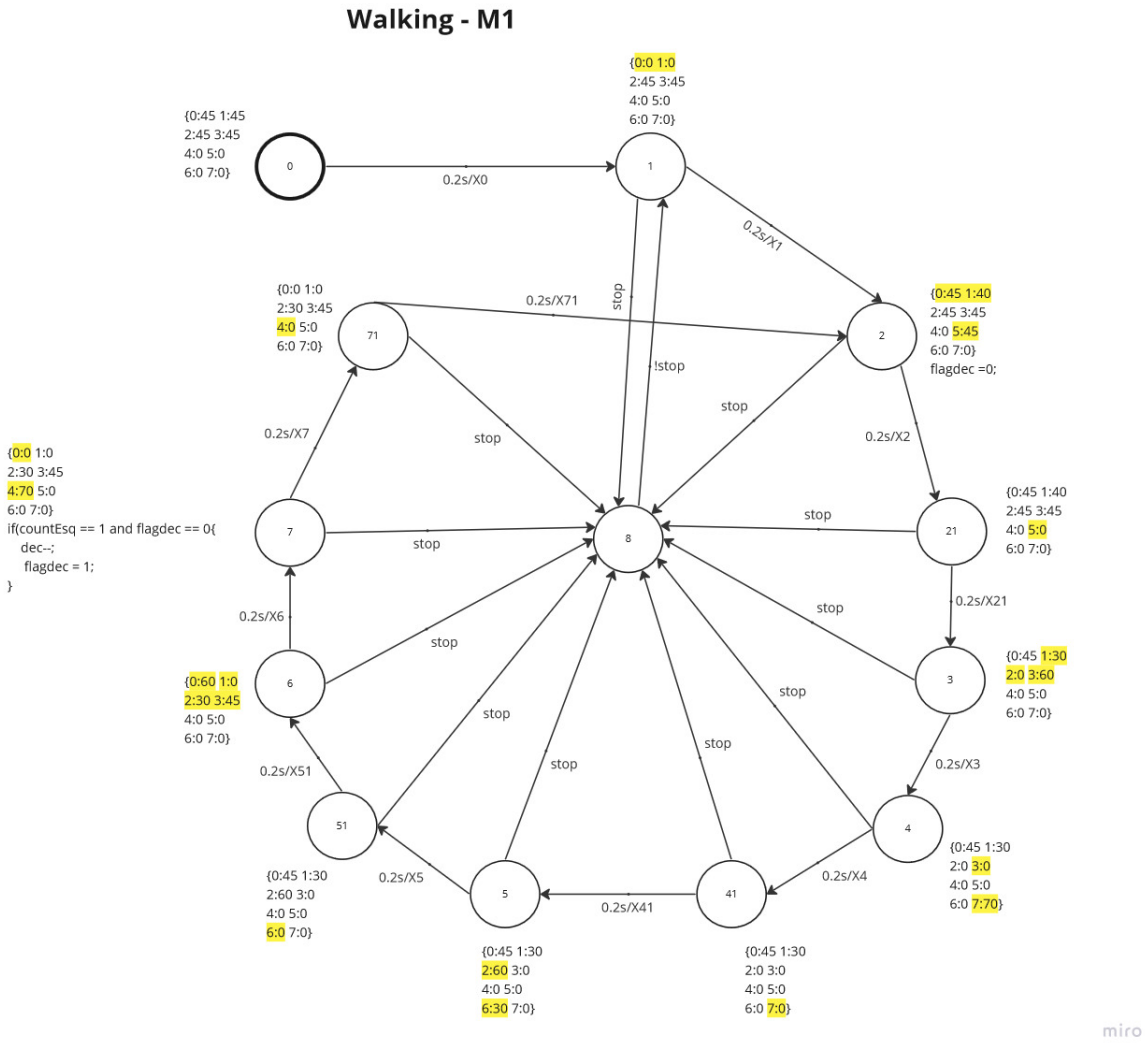


Figure 21: M1

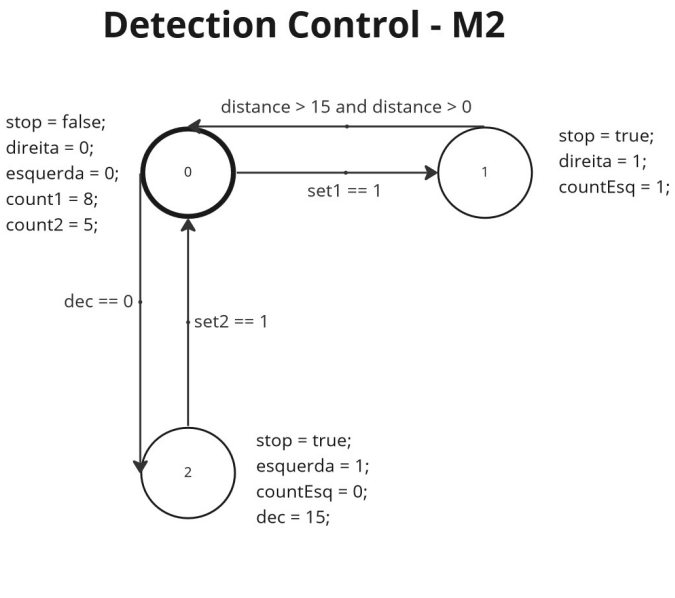

Detection Control - M2
distance > 15 and distance > 0
stop = false;
direita = 0;
esquerda = 0;
count1 = 8;
count2 = 5;
0
1
stop = true;
direita = 1;
countEsq = 1;
set1 == 1
dec == 0
set2 == 1
2
stop = true;
esquerda = 1;
countEsq = 0;
dec = 15;
miro

Figure 22: M2

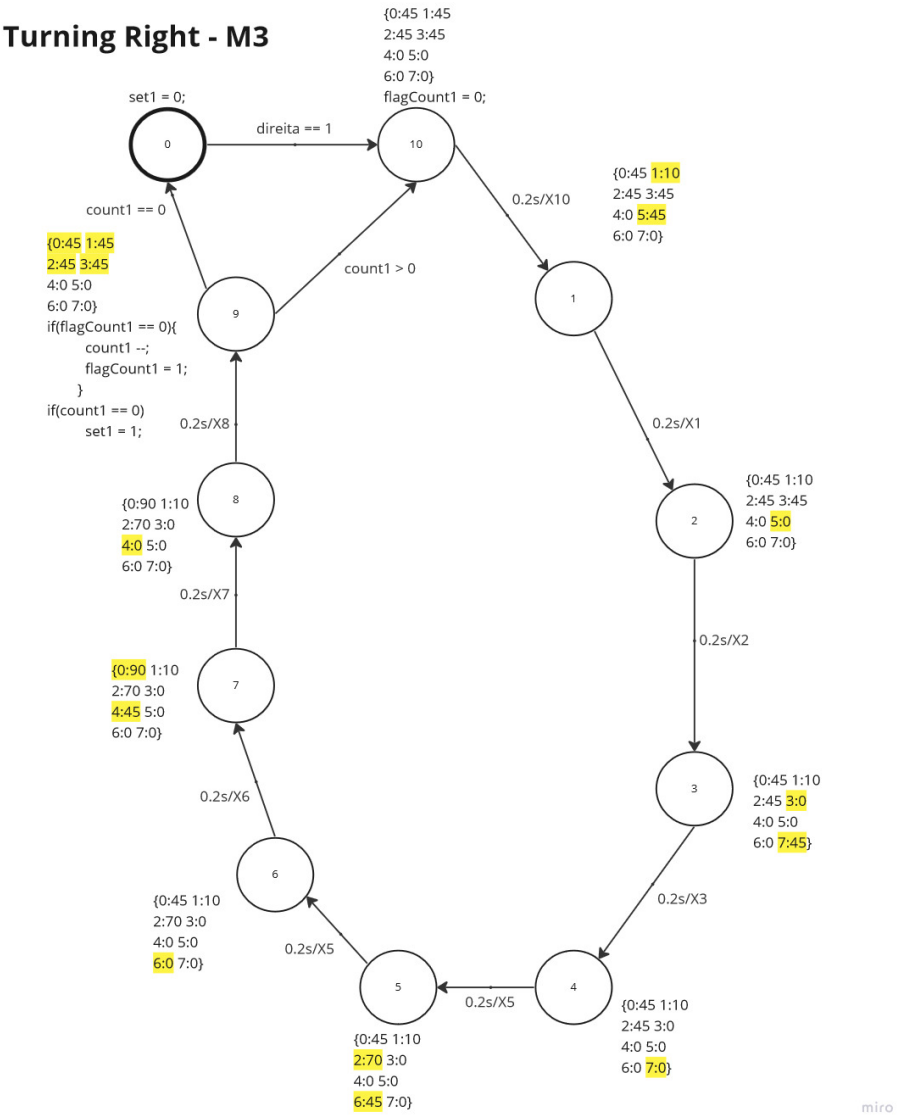

Turning Right - M3
{0:45 1:45
2:45 3:45
4:0 5:0
6:0 7:0}
flagCount1 = 0;
set1 = 0;
direita == 1
0
10
0.2s/X10
{0:45 1:10
2:45 3:45
4:0 5:45
6:0 7:0}
count1 == 0
{0:45 1:45
2:45 3:45
4:0 5:0
6:0 7:0}
if(flagCount1 == 0){
count1 --;
flagCount1 = 1;
}
if(count1 == 0)
set1 = 1;
count1 > 0
9
1
0.2s/X8
0.2s/X1
{0:90 1:10
2:70 3:0
4:0 5:0
6:0 7:0}
8
2
{0:45 1:10
2:45 3:45
4:0 5:0
6:0 7:0}
0.2s/X7
0.2s/X2
{0:90 1:10
2:70 3:0
4:45 5:0
6:0 7:0}
7
3
{0:45 1:10
2:45 3:0
4:0 5:0
6:0 7:45}
0.2s/X6
0.2s/X3
6
{0:45 1:10
2:70 3:0
4:0 5:0
6:0 7:0}
0.2s/X5
5
4
0.2s/X5
{0:45 1:10
2:45 3:0
4:0 5:0
6:0 7:0}
{0:45 1:10
2:70 3:0
4:0 5:0
6:45 7:0}
miro

Figure 23: M3

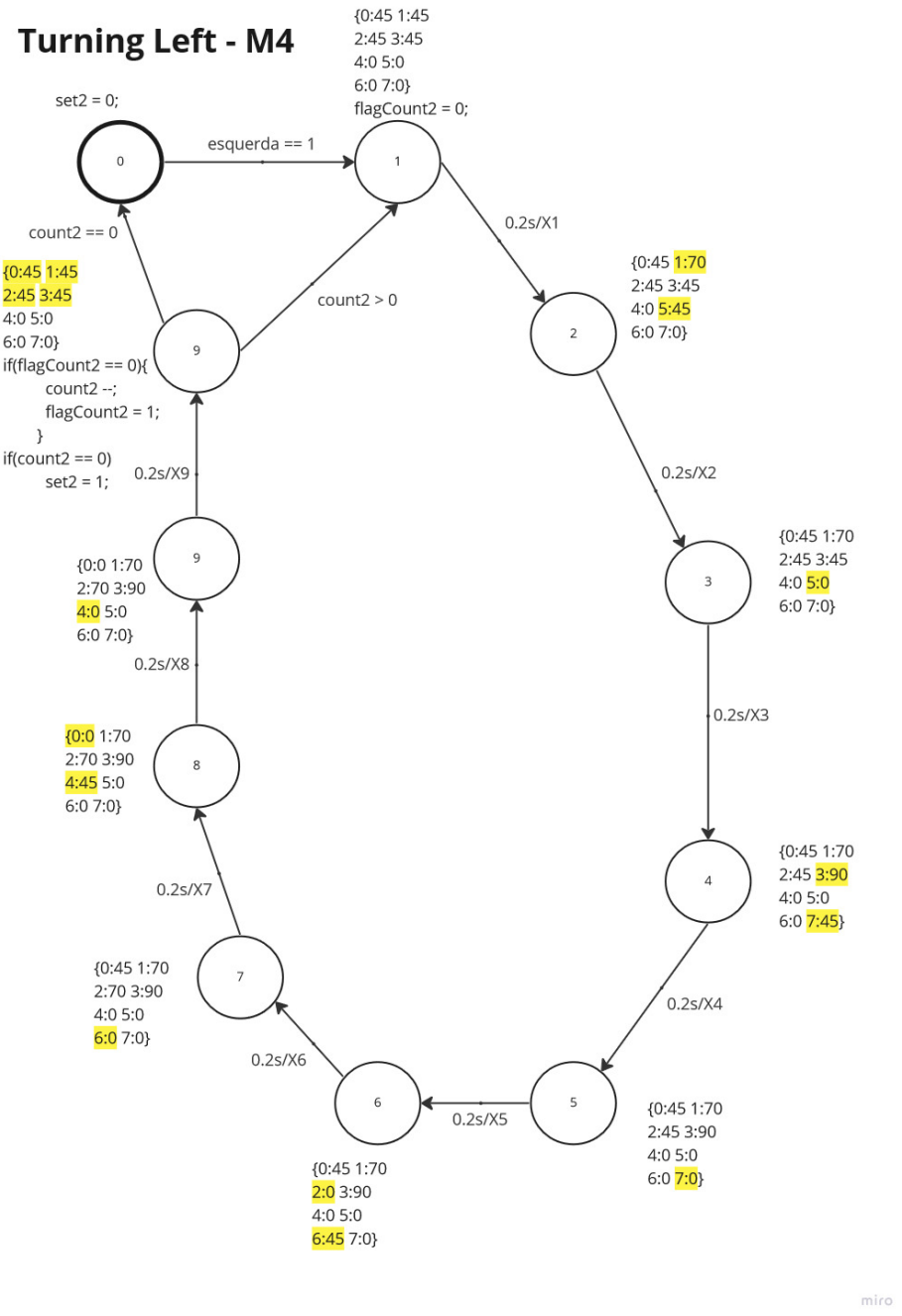

Turning Left - M4
{0:45 1:45
2:45 3:45
4:0 5:0
6:0 7:0}
flagCount2 = 0;
set2 = 0;
esquerda == 1
0
1
0.2s/X1
count2 == 0
{0:45 1:45
2:45 3:45
4:0 5:0
6:0 7:0}
if(flagCount2 == 0){
count2 --;
flagCount2 = 1;
}
if(count2 == 0)
set2 = 1;
count2 > 0
9
2
{0:45 1:70
2:45 3:45
4:0 5:45
6:0 7:0}
0.2s/X2
0.2s/X9
9
3
{0:45 1:70
2:45 3:45
4:0 5:0
6:0 7:0}
{0:0 1:70
2:70 3:90
4:0 5:0
6:0 7:0}
0.2s/X8
0.2s/X3
{0:0 1:70
2:70 3:90
4:45 5:0
6:0 7:0}
8
4
{0:45 1:70
2:45 3:90
4:0 5:0
6:0 7:45}
0.2s/X7
7
{0:45 1:70
2:70 3:90
4:0 5:0
6:0 7:0}
0.2s/X4
0.2s/X6
6
5
0.2s/X5
{0:45 1:70
2:45 3:90
4:0 5:0
6:0 7:0}
{0:45 1:70
2:0 3:90
4:0 5:0
6:45 7:0}
miro

Figure 24: M4