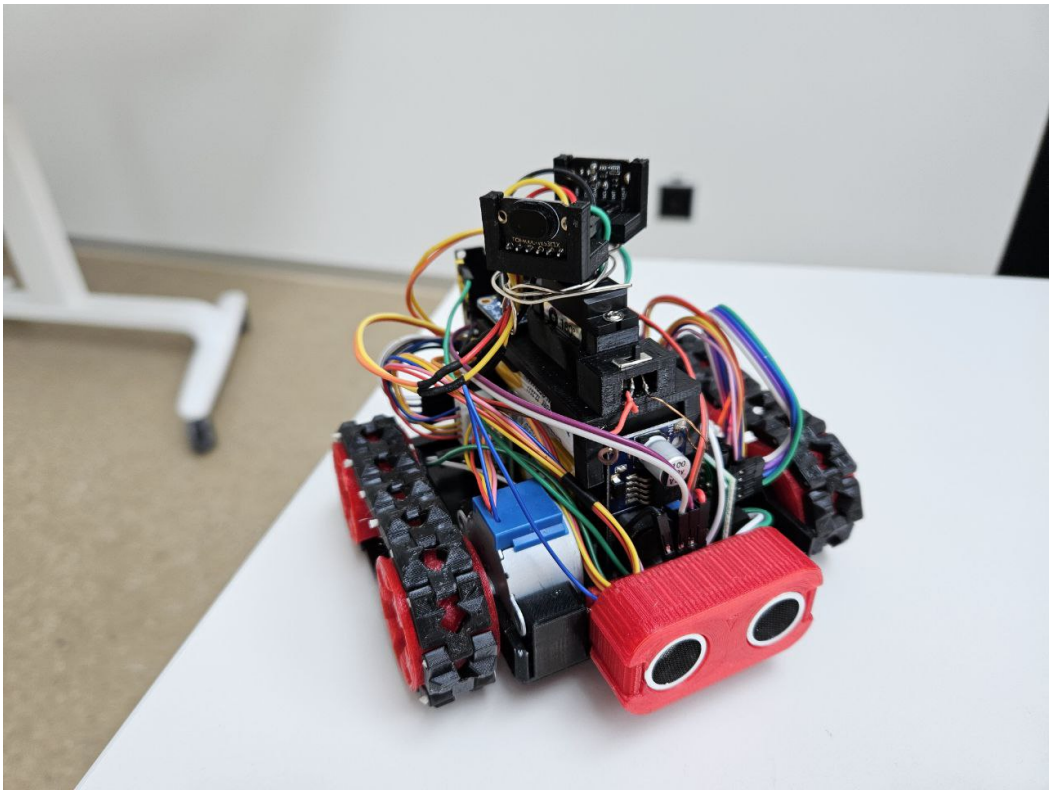
Wall-SLAM - Simultaneous Localization And Mapping

Contents

Contents
- Description
- Overview
- Components and supplies
- 3D Design
  - Sketches
  - Obstacle Detection
- Robot localization
- Algorithm & Data
- Arduino -Explanation of the different arduino files:
- References

Timofey Kreslo, Sylvain Pichot, Finn MacNamara, Alonso Coaguila, Florian Dejean.

June 2023



Description

Wall-SLAM is a project that we have developed within the course CS-358 Making Intelligent Things, under the direction of Prof. Koch and under the supervision of Federico Stello and Anirudhh Ramesh. This report outlines the key features, methodology, challenges faced, and a detailed “How To Build” guide.

Our project proposal, can be useful to understand some deviations in our project from our expectations.

The primary objective of the project was to build a cost-effective and compact SLAM robot capable of mapping unknown and dynamic environments.

Overview

[small video and demo of all functionalities]

Main idea:

We utilize the data from the distance sensors (Ultrasonic and LIDAR), IMU (Inertial Measurement Unit) sensor, and Stepper motors to estimate the robot's pose (position and orientation) and simultaneously construct a map of the environment. All this data will be processed and then

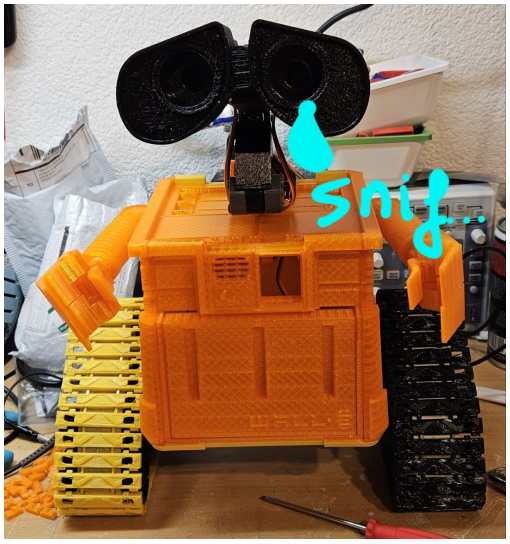
displayed on on a website that This involved integrating data from different sensors using techniques such as sensor calibration, fusion and Kalman filtering algorithms.

Components and supplies

- ESP32
- NXP Precision 9DoF IMU
- VL53L1X Time of Flight Distance Sensor (2x)
- HC-SR04 Ultrasonic Sensor
- 180-Degree Servo
- Stepper motor 28byj-48 (2x)
- ULN2003 motor driver (2x)
- XL6009E1 DC-DC Voltage Converter - 5V
- 9V Battery supply
- Power Switch
- 3D Prints (STLs)

3D Design

As zealous devotees of the enchanting Disney’s figure Wall-E, we couldn’t resist the temptation to transform our prototype into his spitting image. But, alas, our professor promptly intervened, bursting our bubble of whimsy with a witty remark. He reminded us that while Wall-E excelled in garbage collection and exuded undeniable cuteness, our project’s aspirations extended beyond those realms. Thus, we bid farewell to our beloved Wall-E robot, sparing it from a destiny of cuteness overload and instead refocusing our efforts on more practical endeavors.

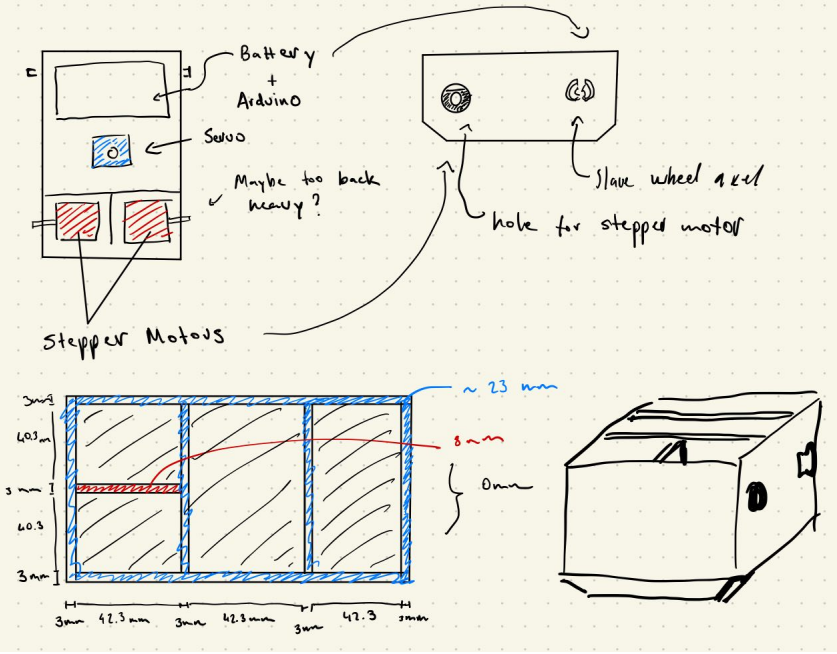


We decided to shift our focus to functionality that would enable mapping of the surrounding environment, which is covered by SLAM. Additionally, we chose to retain the concept of tracks with wheels, as it offers enhanced precision in movement and rotation (especially the ability to turn on the spot), while opting for rubber tracks to minimize slippage during maneuvering.

Sketches

We started by sketching our ideas and figuring out how to arrange electronic components in the best way possible. Our main goal was to create a compact design that would make the assembly process easier and speed up 3D printing. This approach aimed to make the construction phase more efficient overall.

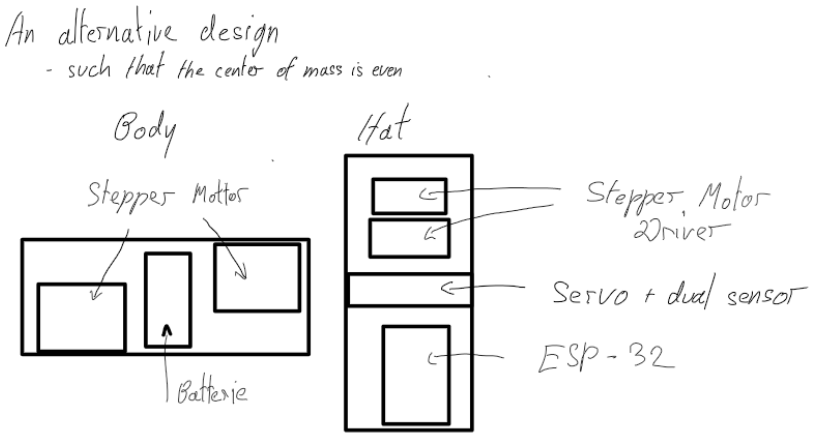
We initiated the development process by focusing on the chassis design. Subsequently, we proceeded to devise an arrangement strategy. The robot incorporated two stepper motors along with their corresponding drivers, a servo, a battery, and a micro-controller. For the initial prototype, we positioned both stepper motors at the rear section of the chassis. In order to counterbalance the weight distribution, the batteries and the micro-controller were placed at the front section. Lastly, the servo, responsible for holding the sensors, was centrally positioned within the chassis.



This approach was promptly dismissed as it became apparent that in order to facilitate turning, both motors needed to rotate in opposite directions, necessitating the rotation of the entire car around its vertical axis.

The rotation of the machine necessitates the alignment of the servo and the primary axis of rotation. This alignment is crucial during turning maneuvers as the distance sensors are required to scan in close proximity to the adjacent wall. By maintaining the sensors in close proximity to the central axis of rotation, the objective is to minimize noise and optimize the accuracy of the scanning process.

Consequently, we proposed an alternative design to address this limitation.



This design greatly appealed to us, and we made the decision to proceed in this particular direction. During our exploration on the web, we chanced upon a robot named

SMARS

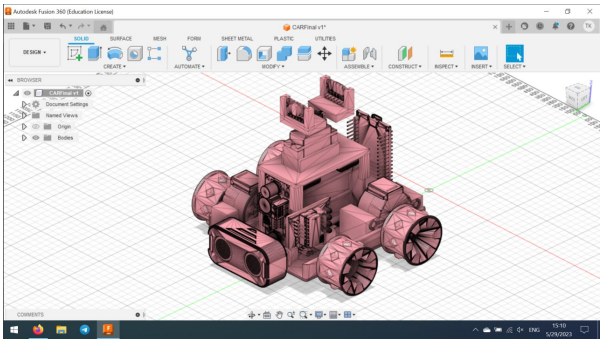
(Screwed/Screwless Modular Assembleable Robotic System), which seemed to align closely with our envisioned requirements. We found inspiration in its design and opted to incorporate a chassis that closely resembled to SMARS. To assess the feasibility of the track concept, we even went as far as designing the entire SMARS idea to evaluate its effectiveness.

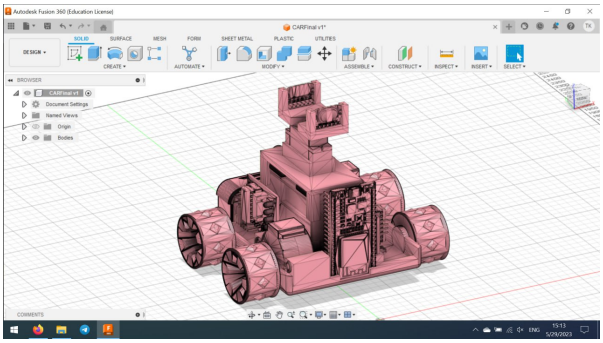
Having appreciated the remarkable ingenuity behind the notion of using the printing filament as a means to securely bind the tracks together, as well as the exceptional functionality of this approach in practical implementation, we wholeheartedly adopted this design for our project. Consequently, this decision entailed the utilization of the same wheels, as the tracks had been meticulously tailored to ensure precise compatibility with these specific wheel components. This deliberate alignment not only guaranteed optimal performance but also facilitated seamless integration between the tracks and wheels, thereby fortifying the overall effectiveness of our system.

The subsequent focus of our attention revolved around the development of the chassis, and the incorporation of the battery, the motors and the sensors.

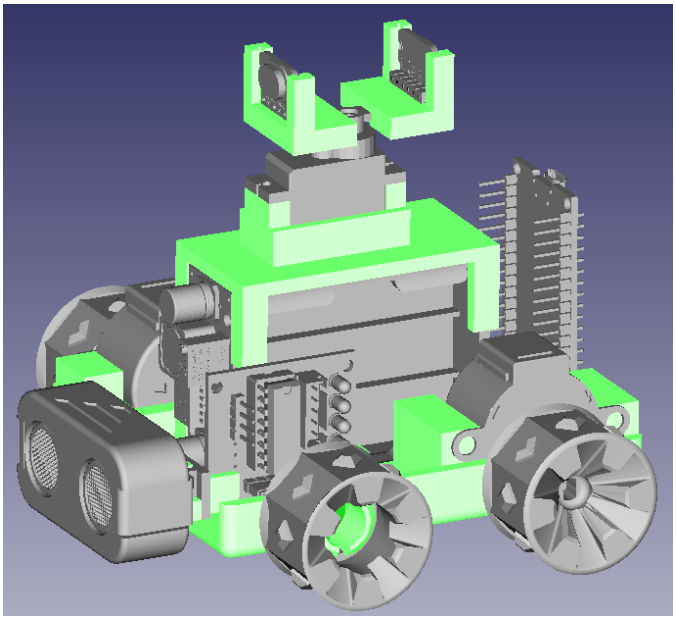
As soon as we agreed on the placement of all the components, we started designing in 3D. In order to leverage the collaborative design features, we opted to use Fusion 360, enabling effective teamwork during the design process. Minor design changes have been made in Freecad, in order to save time.

Here’s what we’ve come up with.

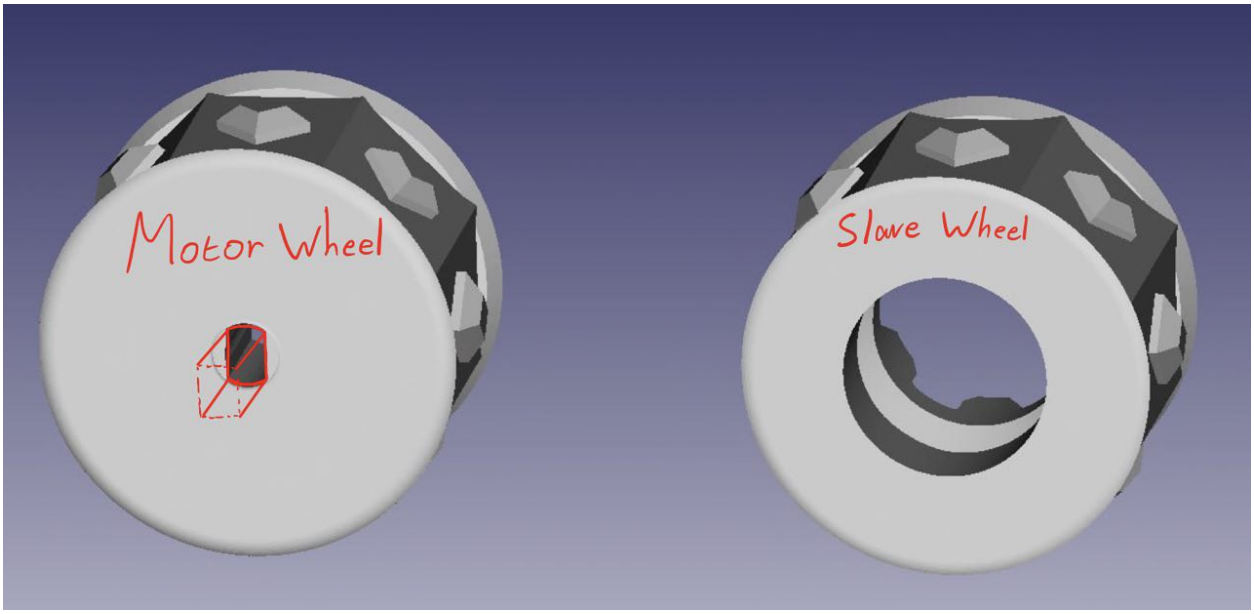




Now we will explain the choice we’ve made for positioning the electronic pieces. The parts that are highlighted in green are those that we designed ourselves. The Ultrasonic sensor’s frame as well as the wheels and the tracks were borrowed from SMARS project, because they fit our requirements well.



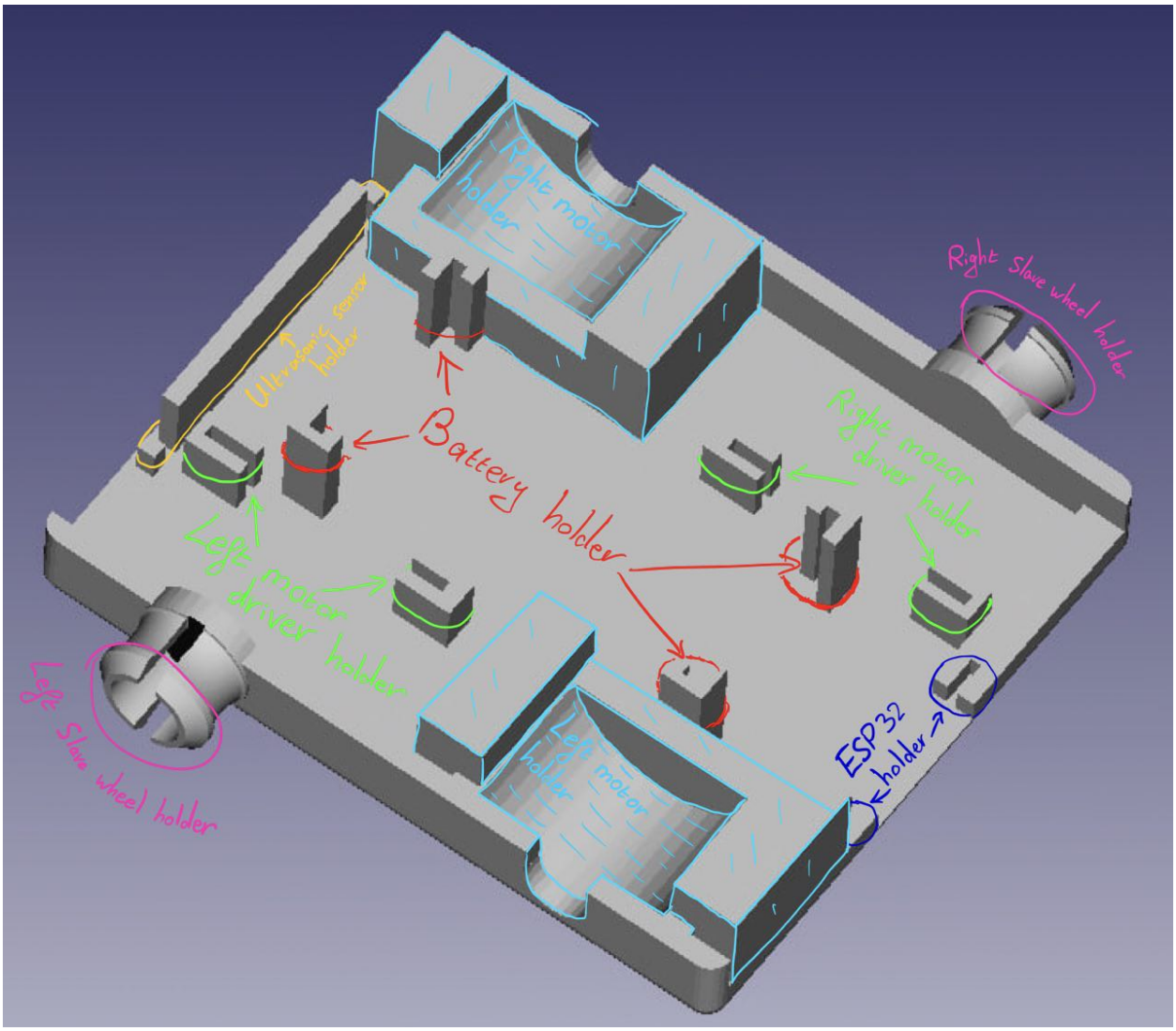
While we adopted the wheel design from SMARS (link for STLs), we had to adapt it's axle mount to fit the rectangular axle of the motor we are using. Furthermore, it is essential to align the axle of the slave wheel with the motor's axle to ensure that both wheels are at the same level.



In order to optimize the efficiency of the printing and assembly process, we strategically positioned all the components in close proximity to one another, ensuring minimal wasted space on the chassis.

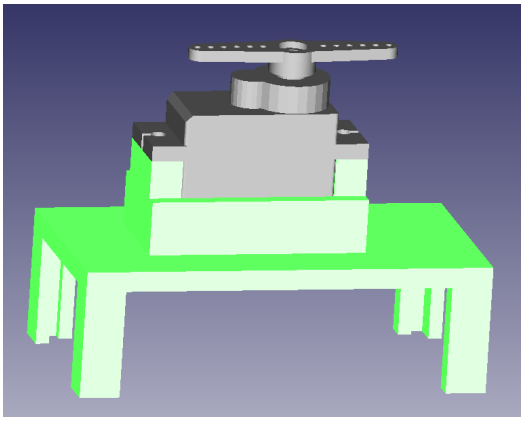
As the heaviest electronic component of the entire construction, the battery serves as a central anchor within the overall design. By positioning it at the center, we establish a stable foundation upon

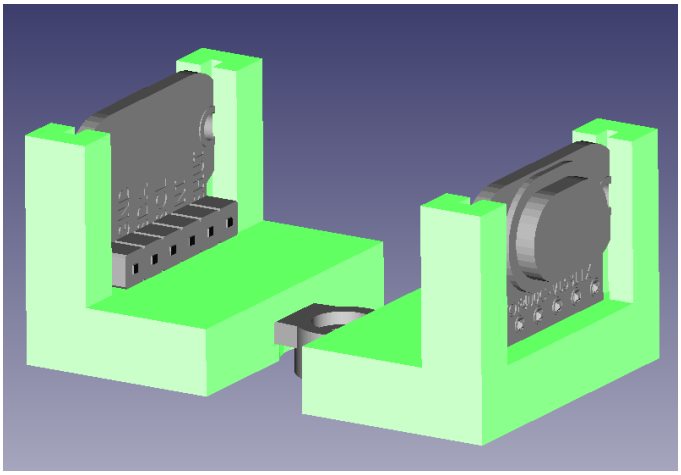
which the remaining components, i.e. the servo and the sensors are securely mounted and connected.



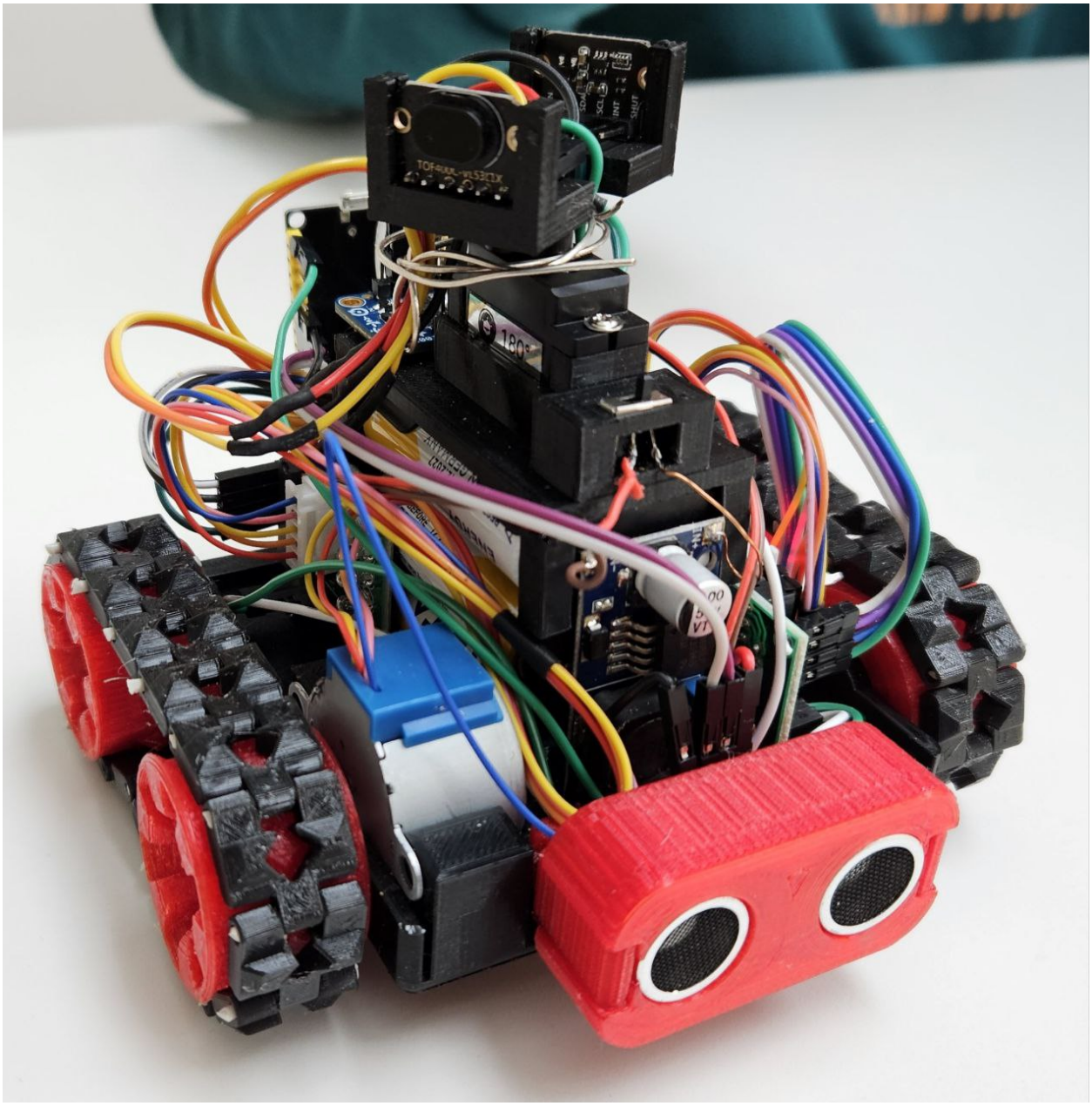
The cover slides over the battery and encases the servo on top.

The servo, in return, holds both lidars (distance sensors) that are sneaked on the servo’s arm.

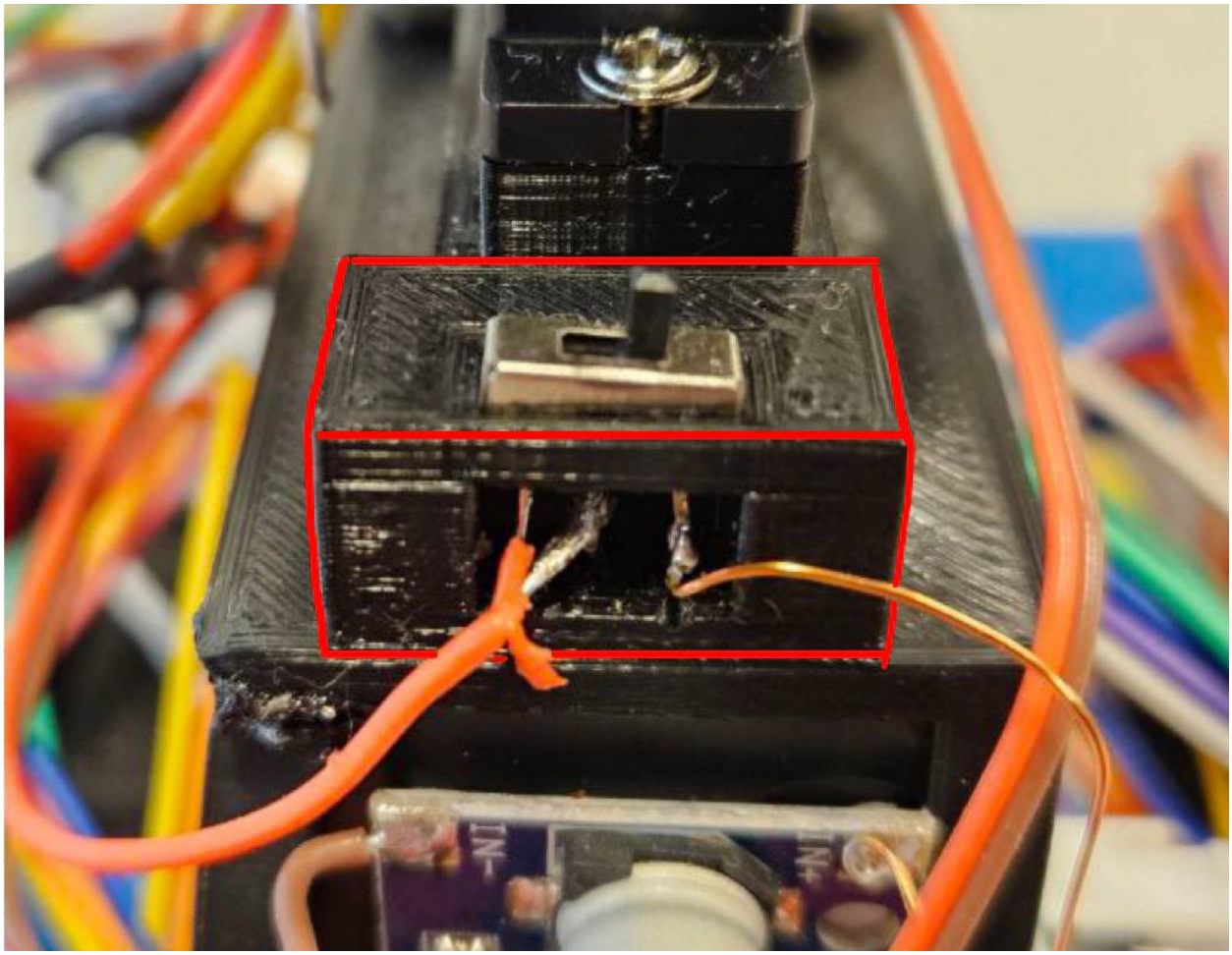




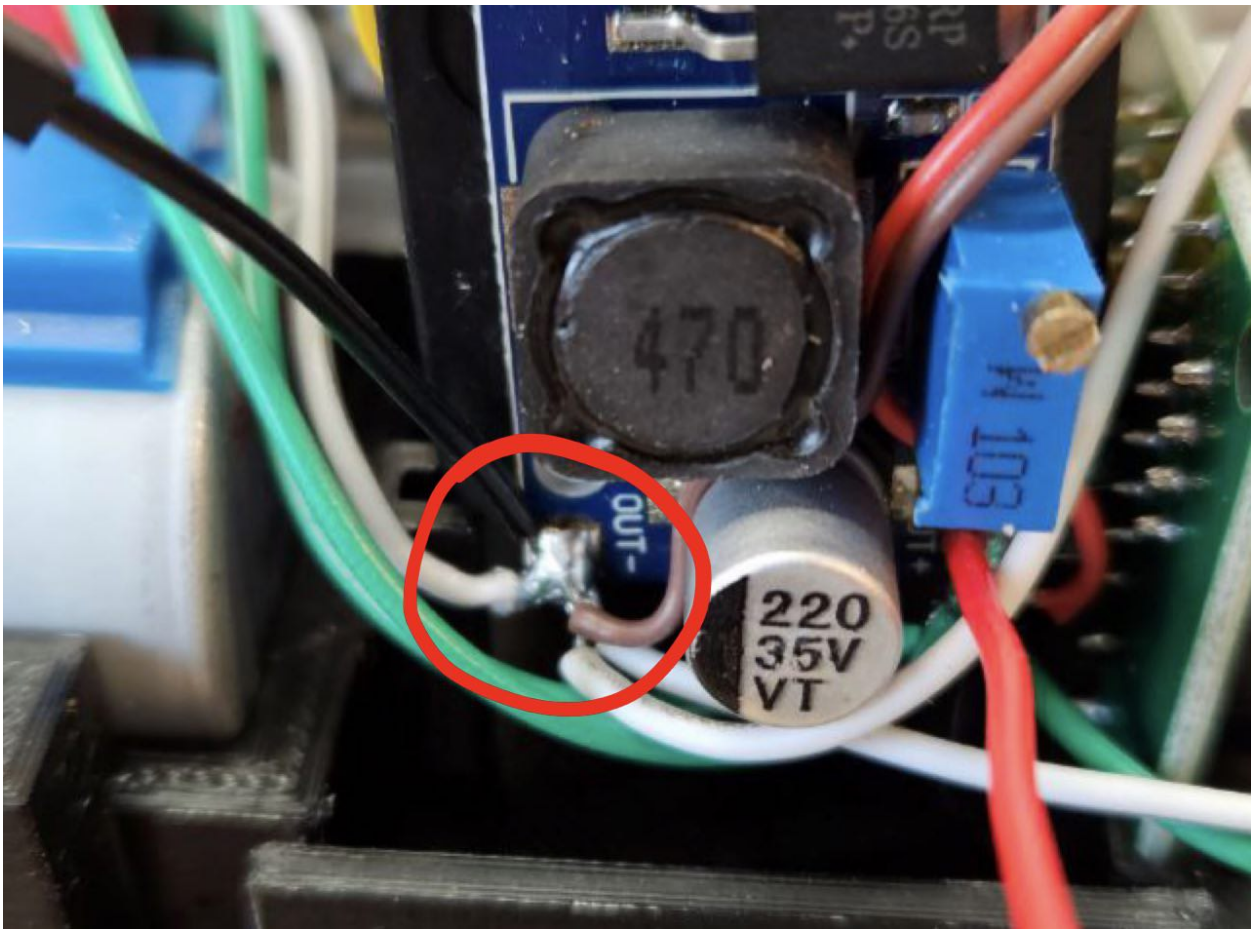
AssemblyOnce the design was ready, it was printed at EPFL in DLLEL, and partly at home. Mostly PETG material was used, except for the tracks which were printed using rubber filament, under the consent of Sébastien Martinerie - 3D printing Coach at the SPOT. Caterpillars are connected by means of an ordinary filament in its initial form, as mentioned above.



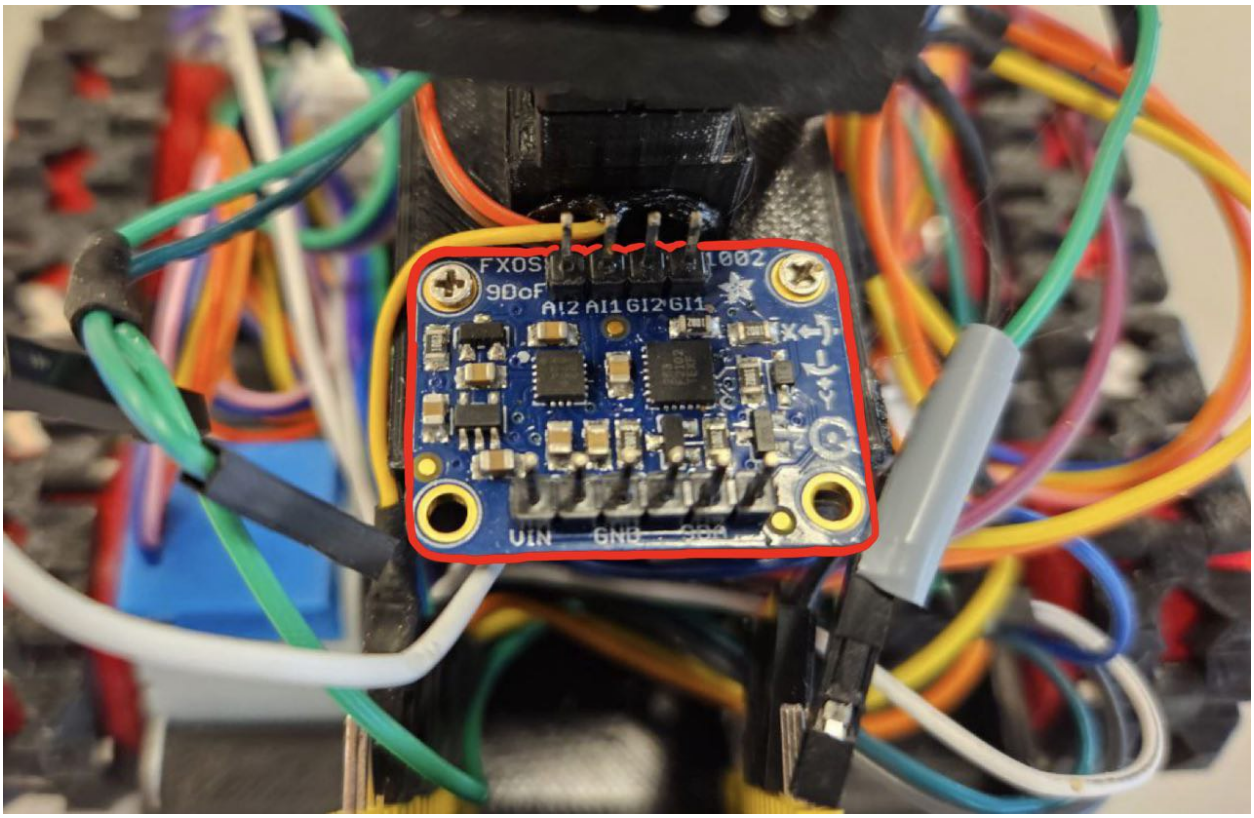
To make power control easier, a switch has been added on top of the cover that holds the servo.



It is directly connected to the voltage regulator underneath, which outputs exactly 5 volts, since multiple components, including the microcontroller operates at this power. Because they also share a common ground, the wires are soldered together in a centralized manner resembling the arrangement of an octopus.

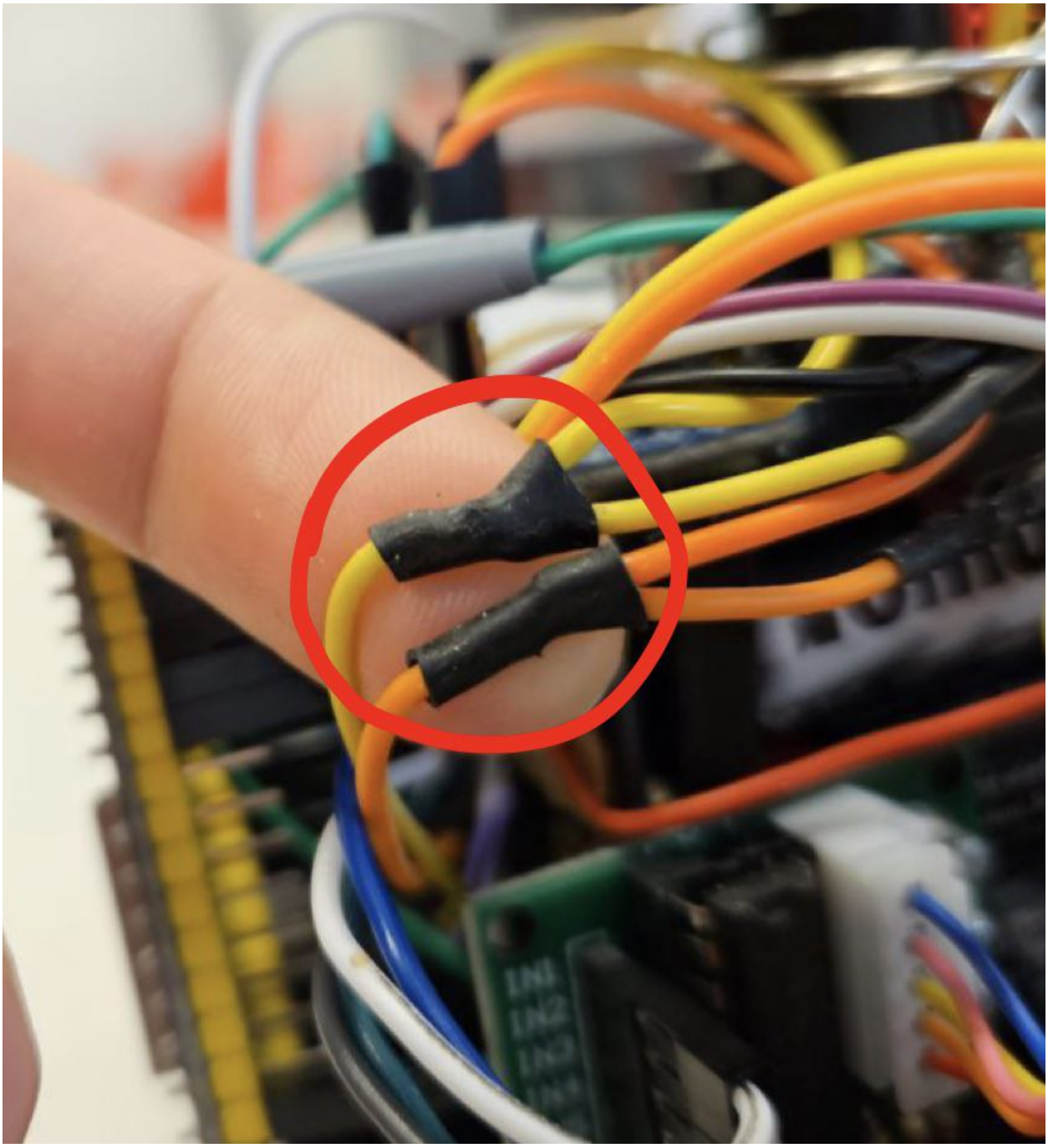


The opposite side of the switch features the motion sensor (IMU) securely attached to the cover that holds the servo, using screws.

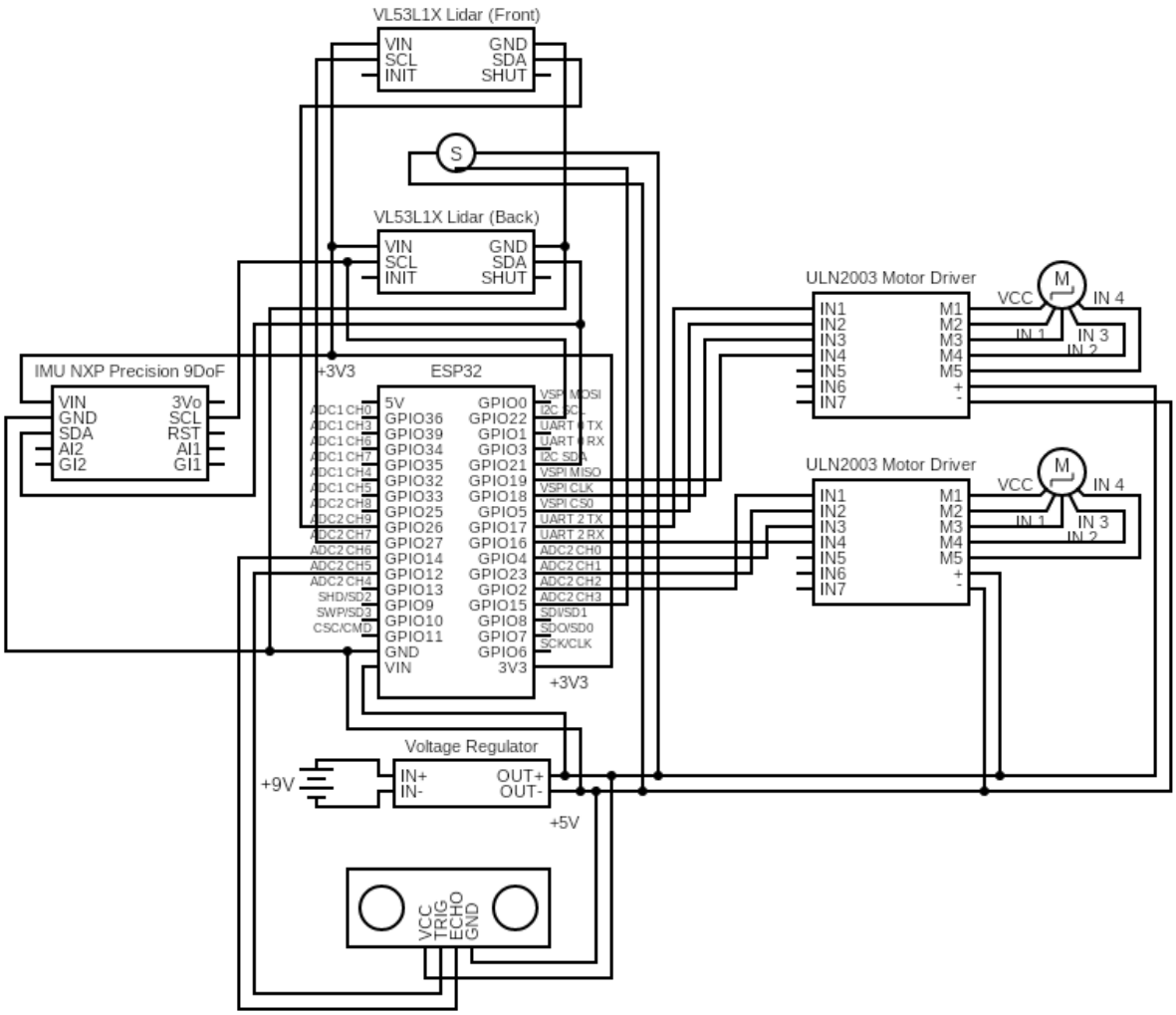


Here the difficulty turned out to be to establish connections between numerous devices that rely on the I2C (Inter-Integrated Circuit) serial communication protocol. In particular, there was an issue with two lidars having the same address, preventing them from being connected on the same I2C bus. One possible solution would have been to use a multiplexer, but since one was not available, an alternative approach was devised through code.

To overcome this challenge, some of the ESP GPIOs needed to be reconfigured to function as I2C pins, to which we connected the front lidar. Subsequently, the motion sensor was connected to the standard I2C pins, and the back lidar was also connected to these same pins as well. This was possible because the motion sensor and the lidar had different addresses assigned to them, ensuring that there was no conflict in their communication over the shared I2C bus. Finally, to ensure proper insulation and protection, we applied heat shrink tubing at the junction point where the three wires met.

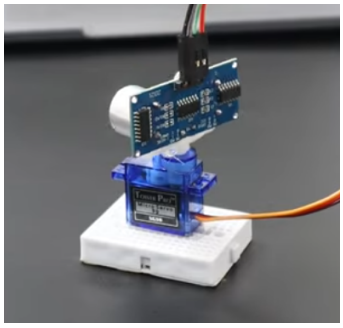


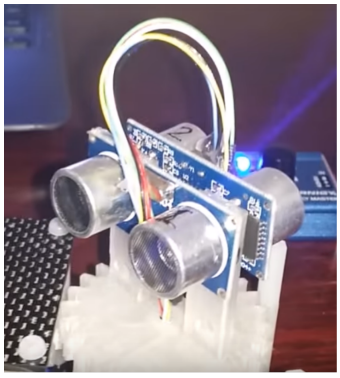
A detailed circuit diagram of all connections is shown below:

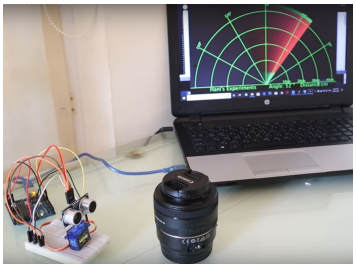


Obstacle Detection

For our obstacle detection and mapping we rely on on two different types of sensors: one ultrasonic sensor who`s purpose is purely to avoid bumping into an obstacle, placed on the front of the car. The sensors that are responsible for mapping are two VL53L1X Lidar sensors, mounted back to back. Keep in mind that these two lidars can only plot single points, so we need to spin them around to maximize.







Robot localization

To determine where the robot is on the map we will use two stepper motors, the inputs given to the motors allow us to calculate the position of the robot based on its starting point. (https://www.youtube.com/watch?v=5CmjB4WF5XA). Regarding the software, we need to create an algorithm that creates a link between the current position and the new targeted position without going through obstacle. To limit inaccuracy, we will use an IMU to determine the orientation of the robot (ex: https://www.youtube.com/watch?v=KMhbV1p3MWk).

Algorithm & Data

Unfortunately, the SLAM algorithm is too memory-bound and computationally slow for a microcontroller. To address this issue, we have the microcontroller connect via Wi-Fi to a capable external computer called the Command Computer (CC). This is why the microcontroller used has Wi-

Fi capability. The Wi-Fi connection also allows the robot to move freely. Therefore, the microcontroller only receives commands movement and returns sensor data to the CC.

The different tools we developed for this are:

- **TCP Connect ESP32 - Python**
  - In order to process our data, we utilize a python code that run intensive computation and take avoid the heavy computation from the ESP32. To do that we need to have great connectivity between the two : esp32.py and espComm.ino are the two files responsible of communication.
    - EspComm.ino : Sets up a WIFI access point to allow the CC to connect. It opens two ports, a sending port and a receiving port. This prevents us from mixing the data up.
    - Esp32.py : This automatically connect to the WIFI generated by the EspComm.ino.

```
import pywifi
def connect_to_wifi(self):
        ssid = self.ssid
        password = self.password
        wifi = pywifi.PyWiFi()  # Create a PyWiFi object
        iface = wifi.interfaces()[0]  # Get the first available network interface

        iface.disconnect()  # Disconnect from any existing Wi-Fi connection
        time.sleep(1)

        profile = pywifi.Profile()  # Create a new Wi-Fi profile
        profile.ssid = ssid  # Set the SSID (Wi-Fi network name)
        profile.auth = pywifi.const.AUTH_ALG_OPEN  # Set the authentication algorithm

        # Set the encryption type and password (comment out if the network is not password-protected)
        profile.akm.append(pywifi.const.AKM_TYPE_WPA2PSK)
        profile.cipher = pywifi.const.CIPHER_TYPE_CCMP
        profile.key = password

        iface.remove_all_network_profiles()  # Remove all existing profiles
        temp_profile = iface.add_network_profile(profile)  # Add the new profile

        iface.connect(temp_profile)  # Connect to the network
        time.sleep(5)  # Wait for the connection to establish

        return iface.status() == pywifi.const.IFACE_CONNECTED
```

Once connected we will listen for any socket communication and if the Esp sent a message we will be able to get it’s IP.

```
while self.espIP is None:
            try:
                self.get_info= socket.socket()
                self.get_info.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                self.get_info.bind(('0.0.0.0', self.recv_port))  # bind to a local address and port
                self.get_info.settimeout(3.0)  # set a timeout of 3 seconds
                self.get_info.listen(0)  # start listening for incoming connections
```

```
                self.recv_socket, client_address = self.get_info.accept() # Wait to receive a transmission
                self.get_info.close() # Close the socket
                self.espIP = client_address[0] # From the answer, get the IP assign to this device
            except Exception as e:
                print(f"No information was sent by ESP, retrying... {e}")
```

Next we will create another socket to be able to send command to the Esp too

```
self.send_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.send_socket.settimeout(5)  # set a timeout of 2 seconds
self.send_socket.connect((self.espIP, self.send_port))
```

Then to read or sent data, we can simple run those command. We will pin a thread to constantly run __listen().

```
def __listen(self):
        while self.running:
            if self.connected:
                try:
                    # Receive data from the client socket
                    data = self.recv_socket.recv(48)
                    if data:
                        data_decoded = struct.unpack('ffffffffffff', data)
                        # Use the data
                          ....

                except Exception as e:
                         print("Connection error :", e)
                         self.connected = False
```

Here, this code is responsible for sending an action number over a socket and waiting for a response from an ESP32 device. It handles connection errors and provides success/failure status codes.

```
def __send_actionNumber(self, actionNumber):
        if self.connected:
            try:
                # Send the packed angles over the socket
                data = struct.pack('f', actionNumber)
                self.send_socket.sendall(data)

                # Wait for a response from the ESP32
                response = 0

                while (not response == "200") :
                    response = self.send_socket.recv(1024).decode()
                    print("Received response from ESP32:", response)

            except Exception as e:
                print("Connection error :", e)
```

```
                self.connected =False
                return 400   # Connection failed
            return 200 # Sucess
        return 400 # Connection failed
```

In case of failure, a thread will be tasked to restart the connection procedure.

- **Web Interface**
- To see in real time the advancement of the mapping and for debugging purposea, we made a complete interface that display every step of the data flow
    - *Communication frequency*: shows the number of data packages sent and received, as well as the number of obstacles detected.
    - Orientation Line Chart: shows the sensor fusion of the accelerometer and magnetometer and its variation. The absolute orientation of the car is also displayed.
    - *Noisy Obstacle Detection*: This is a map without any filter, displaying every data point no matter how weird it is. This allows us to clearly see how much data we filter out.
    - *Redundancy Check*: This is a map that has undergone a first cleanup.
    - *K-Mean Graph*: This map is a visual representation of obstacles. With some basic machine learning that we have optimized, we can map and understand the environment with greater ease.
    - *Dijkstra Interactive*: This interactive map allows you to move the car to a specific location on the map without hitting any obstacles. It also shows the obstacles found by K-means.

The different algorithm used are

- Pathfinding Algorithm:

By default, the robot assumes it will map an area of 4 square meters, which is a 2 by 2 meter square. We have decided to sample the mappable area into a 20 by 20 cell grid. Each obstacle point that we measure will correspond to a cell in the grid.

Once a cell has accumulated a certain number of obstacles and reached a certain threshold, it will be set to an occupied obstacle state. This process creates a grid map that is constantly updated when new obstacles are detected. It is then possible to select a cell for the robot to navigate autonomously by choosing the shortest path through the obstacle cells. A Breadth First Search algorithm is used for this purpose. Here is a step-by-step description of the Dijkstra algorithm.

One challenge was to translate global instructions into instructions that the car can understand, which are limited to goForward(), turnLeft(), and turnRight(). For example, if the first instruction in the

path tells the car to start moving west, but the car is facing north, this needs to be adjusted for. The same problem arises when there is a change in direction, as it requires two commands for the car: orientateLeft() / orientateRight() followed by a forward() command.

- **Map - kmean.py and map.py**

  In order to determine if there is a real obstacle, we will kmean to determine their position and center. Finding those center are useful to move the less dense zone, the zone that is inexplored or empty. Due to our Dikjstra algorithm, we added high weight to already explore area, hence resulting in a path that will always try to explore. If a path cannot explore anymore, this mean we have map the whole area.

  - Kmean try to assign K cluster to a data set and optimize the distance between the point from a center.
    - To determine the best K, so our number of obstacle, we run multiple time the algorithm and look for the minimal distance of the cluster related to their center. ***(Elbow method)***
    - To initiate our center, we already approximate where the center should be ***(kmean ++).*** Then we move again the center to approach the center of mass better till we converge.
  - In order to reduce noise, which can be seen as the outlier of our dataset, we split our data before training.
  - To represent an obstacle, we decided to turn them into rectangle taking the minimum and maximum. This lead to more strict threshold since our obstacle are hollow shape. Leading to create multiple rectangle for one obstacle depending on its size.

  kmean.py handle all the necessary function to run the kmean++ algorithm and the Elbow method

  mapk.py handle how to interpret this data in your main app.

- **Noise Correction - Kalman**

  In order to combine our sensor, we utilize known variance of them to correct sudden noise.

  - Used notably in calculating
    - The distance travelled from the IMU and the stepper motors\
    - Using the acceleration from the IMU, the actual speed of the robot can be determined for a short period. A better estimate can be determined with a Kalman filter by combing the IMU measurements and the speed inputted in the steppers. What the Kalman filter does is an estimate by combining our prediction of speed and the correlated measurement of acceleration through the covariance of our prediction and our measurements. The covariance was gathered from the documentation from the physical components, being the IMU and the stepper motors. Unfortunately, we could not accurately implement a

Kalman filter due to multiple reasons. First of all, the stepper motors lacked proper and clear documentation regarding their covariance regarding their position and speed so this led to estimating a covariance. Second of all, we lacked experience with developing Kalman filters. Third of all, the stepper motors are too precise and the IMU is too imprecise, rendering fine tuning very difficult. Each reason exacerbated the next one. The final straw was when it was found that multiplying a scalar to the steps done by the stepper motor was more precise and requires less computation. Nevertheless, we believe that a Kalman filter is possible to implement for the simultaneous localization but given that a simpler and more effecient solution provided with better results than the prototype. If a Kalman filter were to be implemented, we would recommend replacing the stepper motors with an alternative.

    - The orientation of the magneto and gyro of the IMU **[Librariry used]**

- **Data Validation - Redundancy**

  In order to check if a data point was a validate obstacle in the real life, we will often perform a redundancy check. Since our design allow us to move freely the orientation of our sensor, we can track that obstacle as wanted.

Arduino -Explanation of the different arduino files:

1: accessPoint.ino
This code sets up a web server on an ESP32 or ESP8266 microcontroller board using the Arduino framework. It allows us to control/view/animate the data from the robot remotely through a web interface.

2. espComm.ino

WiFi: The code sets up an Access Point using the ESP32's built-in WiFi capabilities. It creates a server to receive data on one port and send data on another port.

```
void readData() {
  if (!recieveClient.connected()) {
    Serial.println("No client connected (Recieve)");
    recieveClient = recieveServer.available();
  } else {
    Serial.println("New client connected (Recieve)");
    if (recieveClient.connected() && recieveClient.available()) {
      // Read the packed dataRecieved from the socket
      float dataRecieved[1];
      recieveClient.read((byte*)dataRecieved, sizeof(dataRecieved));

      // Unpack the dataRecieved
      actionNumber = dataRecieved[0];
```

```
      // Do something with the decoded angles
      Serial.print("Action recieved: ");
      Serial.print(actionNumber);

      //action(actionNumber);
      // Send "200" back to the client
      recieveClient.write("200");
    }
  }
}

void sendData() {
  packData();

  if (!sendClient.connected()){
    if(sendClient.connect(user_IP, SEND_DATA_PORT)) {
      Serial.println("Connected to server");
    } else{
      Serial.println("Non connection");
    }
  } else {
    sendClient.flush();
    sendClient.write((byte*)dataSend, sizeof(dataSend));
    Serial.println("Data Sent");
  }
}

void packData() {
  updateSensors();
  dataSend[0] = servo.read();
  dataSend[1] = distanceSonarFront;
  dataSend[2] = lidarDistanceFront;
  dataSend[3] = lidarDistanceBack;
  dataSend[4] = servo.read() + goToOrientation;
  dataSend[5] = curr_x;
  dataSend[6] = curr_y;
  dataSend[7] = elapsedTime;
  dataSend[8] = orientationMag; // Mag or
  dataSend[9] = yaw;  // Gyro Or
  dataSend[10] = heading;
  dataSend[11] = goToOrientation;
  Serial.print("Orientation :");
  Serial.print(goToOrientation);

  Serial.println("magOrientation: "+ String(orientationMag));
  Serial.println("gyroOrientation: "+ String(yaw));
  Serial.println("Kalman heading: "+ String(heading));

}
```

Sensors: The code initializes and reads data from various sensors, including ultrasonic sensor, VL53L1X Lidar sensors (front and back), and the FXAS21002C gyro and FXOS8700 accelmag sensors.

```
void updateSensors() {

  //Update Timer
  elapsedTime = (float)(millis() - startTime);

  // Ultrasonic Update
  distanceSonarFront = frontUltrasonic.ping_cm() * 10;

  // Lidar Update
  rotateServo();
  readLidar();

  getHeading();

  // IMU Update
  gyro.getEvent(&event);
  accelmag.getEvent(&aevent, &mevent);

  // Orientation Update
  orientationMag = atan2(-mevent.magnetic.y, mevent.magnetic.x) * 180 / PI - initialMagHeading;

  if (orientationMag < 0) {
      orientationMag += 360;
    } else if (orientationMag >= 360) {
      orientationMag -= 360;
    }

  unsigned long currentTimestamp = millis();
  float deltaTime = (currentTimestamp - prevTimestamp) / 1000.0; // Convert milliseconds to seconds
  prevTimestamp = currentTimestamp;

  // Update the yaw
  yaw -= (gyroZ * 180/PI) * deltaTime;

   //Keep yaw within the range of 0 to 360 degrees
  if (yaw < 0) {
    yaw += 360;
  } else if (yaw >= 360) {
    yaw -= 360;
  }

}
```

Motors: The code controls the movement of stepper motors connected to the robot. It includes functions to move the robot forward, backward, right, left, and stop the motors.

```
void moveForward() {
  stepperLeft.setSpeed(-CONST_SPEED_STEPPER);
  stepperRight.setSpeed(CONST_SPEED_STEPPER);
  if(stepperLeft.runSpeed()){
    if (goToOrientation == 0) {
        curr_y += DIST_PER_STEP;  // move north
    } else if (goToOrientation == 90) {
        curr_x += DIST_PER_STEP;  // move west
    } else if (goToOrientation == 180) {
```

```
        curr_y -= DIST_PER_STEP;  // move south
    } else if (goToOrientation == 270) {
        curr_x -= DIST_PER_STEP;  // move east
    }
  }
  stepperRight.runSpeed();
}

void action(float actionNumber) {

  switch ((int)actionNumber) {
    case 0:
      stopMotors();
      break;
    case 1:
      if (distanceSonarFront < MIN_SONAR_DISTANCE && actionNumber == 1) {
        stopMotors();
      }
      else moveForward();
      break;

    case 2:
      moveBackward();
      break;

    case 3:
      moveRight();
      actionNumber = 0;
      break;

    case 4:
      moveLeft();
      actionNumber = 0;

      break;

    default:
      stepperLeft.run();
      stepperRight.run();
  }

  actionNumber = 0;
}

void Task1code(void* pvParameters) {
  for (;;) {
    action(actionNumber);
  }
}
```

Servo: The code controls a servo motor connected to the robot. It rotates the servo to a specific angle, which is controlled by the "servoAngle" variable.

```
void rotateServo() {
  if (servoAngle >= 180 || servoAngle <= 0) direction = -direction;
  servoAngle += direction;
  servo.write(servoAngle);
```

```
}
```

Data Transmission: The code includes functions to send and receive data over WiFi. It packs sensor data into an array and sends it to a client connected to the robot's Access Point.

```
void readData() {
  if (!recieveClient.connected()) {
    Serial.println("No client connected (Recieve)");
    recieveClient = recieveServer.available();
  } else {
    Serial.println("New client connected (Recieve)");
    if (recieveClient.connected() && recieveClient.available()) {
      // Read the packed dataRecieved from the socket
      float dataRecieved[1];
      recieveClient.read((byte*)dataRecieved, sizeof(dataRecieved));

      // Unpack the dataRecieved
      actionNumber = dataRecieved[0];

      // Do something with the decoded angles
      Serial.print("Action recieved: ");
      Serial.print(actionNumber);

      //action(actionNumber);
      // Send "200" back to the client
      recieveClient.write("200");
    }
  }
}

void sendData() {
  packData();

  if (!sendClient.connected()){
    if(sendClient.connect(user_IP, SEND_DATA_PORT)) {
      Serial.println("Connected to server");
    } else{
      Serial.println("Non connection");
    }
  } else {
    sendClient.flush();
    sendClient.write((byte*)dataSend, sizeof(dataSend));
    Serial.println("Data Sent");
  }
}

void packData() {
  updateSensors();
  dataSend[0] = servo.read();
  dataSend[1] = distanceSonarFront;
  dataSend[2] = lidarDistanceFront;
  dataSend[3] = lidarDistanceBack;
  dataSend[4] = servo.read() + goToOrientation;
  dataSend[5] = curr_x;
  dataSend[6] = curr_y;
```

```
  dataSend[7] = elapsedTime;
  dataSend[8] = orientationMag; // Mag or
  dataSend[9] = yaw;  // Gyro Or
  dataSend[10] = heading;
  dataSend[11] = goToOrientation;
  Serial.print("Orientation :");
  Serial.print(goToOrientation);

  Serial.println("magOrientation: "+ String(orientationMag));
  Serial.println("gyroOrientation: "+ String(yaw));
  Serial.println("Kalman heading: "+ String(heading));

}
```

Overall, this code provides a framework for controlling a robot's movement, reading sensor data, and transmitting the data over WiFi. It can be further customized to implement specific behaviors and functionalities for the robot.

References

- [1] Figure 1 : Alain Godot. Algorithmes SLAM (Simultaneous Localization and Mapping). May 2019. URL: https://www.innowtech.com/2019/05/16/les-algorithmes-slam-simultaneous-localization-and-mapping/
- [2] Figure 2 : @CodersCafeTech. DIY Radar With Ultrasonic Sensor And Chat-GPT Generated Arduino Code — Coders Cafe. Youtube. 2023. URL: https://youtube.com/shorts/o7DMHJKhpws?feature=share
- [3] Figure 3 : Channel Everything. Two ultrasonic sensors on a servo for angle and distance data. Youtube. 2016. URL: https://www.youtube.com/watch?v=dHZB0WhLl8g&t=12s
- [4] Figure 4 : Hani’s Experiments. How to make an ultrasonic Radar. Youtube. 2021. URL: https://www.youtube.com/watch?v=xngpwyQKnRw
- [5] Figure 5 : Kevin McAleer. SMARS Fan. 2018. URL: https://www.smarsfan.com