Ministry of Higher Education and Scientific Research
****
Carthage University
****
National Institute of Applied Sciences and Technology

Mandatory Internship report

Branch: Industrial Computing and Automation (IIA)
Lavel : 4th Year

Topic :
Development of a Trajectory planning algorithm for the P-Solar robot

Work of: Ilef Mghirbi

Company: **Enova Robotics**

Ministry of Higher Education and Scientific Research
****
Carthage University
****
National Institute of Applied Sciences and Technology

Mandatory Internship report

Branch: Industrial Computing and Automation
Level: 4th Year

Topic :
Development of a Trajectory planning algorithm for the P-Solar robot

Work of: Ilef Mghirbi

Company: **Enova Robotics**

Supervisor at the company : **Mr. Amine Bouabid**	**Opinion of the internship committee:**

Table of content :

Tables of Illustrations :

1. *General Introduction:*

This report aims to present and explain the work done in order to fulfill the specifications set by the company. It contains the company's problematic, the suggested solutions, the outcomes, and the perspectives. Additionally, this report aims to showcase my growth and learning throughout the internship period, highlighting the practical application of algorithmic concepts in real-world scenarios.

2. *Company Presentation:*

2.1 Introduction:

Enova Robotics is a Tunisian Sousse based company founded in 2014 by **Dr. Anis Sahbeni**, a robotics researcher that launched the company to serve the goal of making both simple daily tasks and crucial missions easier and safer for humans.

2.2 Company's Structure:

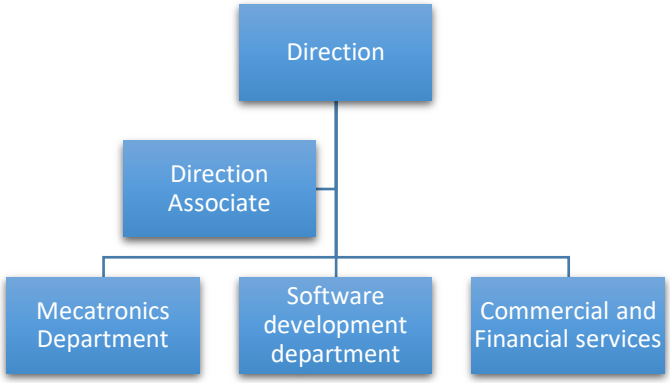
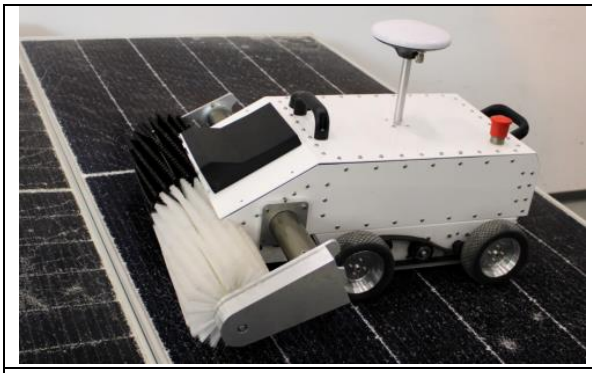


Figure 1: Company structure

2.3 Products:

Robot	Description
	**PGuard:** The autonomous patroller is the company's most advanced robot, it has until now 4 versions. It is used for security missions in rough conditions or when human intervention is hard/dangerous.

	**PSolar**: The autonomous robot is subject to my mandatory internship, it is the newest product and a solution developed for a solar panels dealer aiming to enlargen their products range.
	**Minilab** is a laboratory oriented robot, designed to help students and professors apply robotics related applications and learn through experiments.
	**Veasense** was designed and developed during the Covid-19 pandemic times to cover the shortage in caregivers and help doctors with their daily missions while protecting them by to interact and also delivering medicines to the sick.
	**AGV** is an autonomous mobile cart designed to carry a payload from warehouses to production lines in a wide range of industries. The AGV is used by manufacturers to automate their in-house transportation and logistics.

2.4 Problematic and Objectives:

P-Solar is an autonomous solar panel cleaning robot. Its mission is to optimize the panels' performance by doing regular cleanings without resort to an operator to manipulate it. Solar panels are most performing when they are 100% clean allowing it to utilize the light efficiently. The robot's importance becomes evident in deserted or windy/rainy areas where it is hard to hire someone to clean and where we need panel cleaning more often.

Given that the company just finished developing the first version of the robot, there are many potential features to be developed, tested, and implemented on the next versions:

- Developing the trajectory-generating algorithm according to a pattern chosen by the company and simulating it using the panel's corners coordinates as input.
- Developing the trajectory-generating algorithm in the presence of obstacles with known coordinates.
- Trajectory tracking to have traceability of the swept (cleaned) areas and the non-swept areas.
- Refining the panel's GPS coordinates for seamless integration into the trajectory algorithm.

3. Work done:

3.1 Tools:

- **VS code: code editor**
- **Python and Python libraries: Matplotlib, NumPy, SymPy**
  a) NumPy: NumPy is a Python library for efficient numerical operations on arrays and matrices, commonly used in data analysis and scientific computing.
  b) Matplotlib: Matplotlib is a versatile Python library for creating a wide range of 2D visualizations, ideal for data plotting and scientific graphics.
  c) SymPy: SymPy is a Python library for symbolic mathematics, allowing for algebraic manipulation, calculus, and equation solving in a symbolic form.

- **Mathematical tools**
- **Json:** JavaScript Object Notation is a lightweight data interchange format. It is easy for humans to read and write and easy for machines to parse and generate. JSON is often used to transmit data between a server and a web application or between different parts of a software system.

3.2 Algorithm1: Simple Trajectory planner

This algorithm aims to generate the robot's trajectory through multiple checkpoints that draw the pattern in figure 2. This pattern was chosen after trials of the actual robot that proved that the pattern in figure 1 could not be implemented due to the risks of errors leading to the robot failing to find the correct checkpoint or falling off the solar panel.

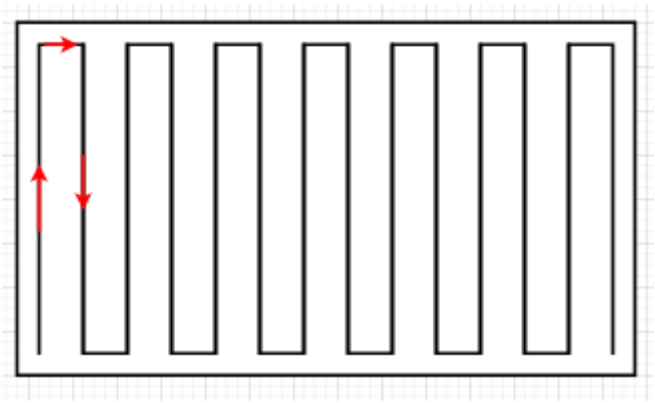


Figure 2: Old Trajectory Pattern

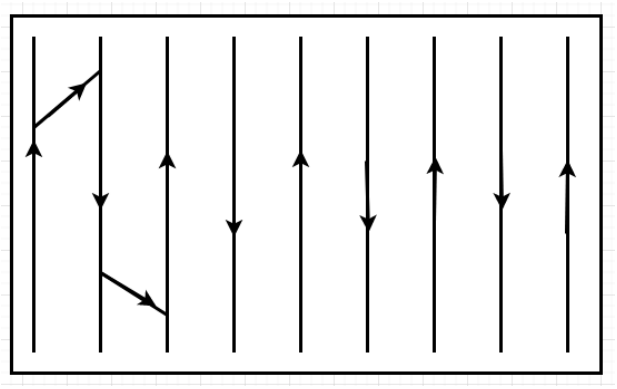


Figure 3: The considered Trajectory Pattern

- **Pattern:** After numerous tests of different navigation algorithms, and given the inclination and error factors, the engineering team has decided to consider the pattern in figure 2. The idea is to save the forward-facing position of the robot and to execute the transition from one section to the other in a **lateral** motion as shown in the same figure. Checkpoints are used to make sure the robot is not deviating with an interpolation step of 2 meters. Sections are divided into even and odd sections using their left to right order where even sections are executed in a **forward** motion and odd section are executed in a **backward** motion. The checkpoints are saved in Json format to be delivered to the navigation part.

- **Code sections:**

  Generating the checkpoints:

```
def IntermPoints_Generator_even(X,Nbr_Checkpoints, Interpolation_Step,Y_start ):
    Interm_Points_Array = [[0 for x in range(Nbr_Checkpoints)] for y in range(2)]
    for i in range (Nbr_Checkpoints):
        Interm_Points_Array[0][i] = X
        Interm_Points_Array[1][i] = Interpolation_Step * (i+1) + Y_start

    return  Interm_Points_Array[0],  Interm_Points_Array[1]

def IntermPoints_Generator_odd(X,Nbr_Checkpoints, Interpolation_Step, Y_end): #nbr of checkpoints
    Interm_Points_Array = [[0 for x in range(Nbr_Checkpoints)] for y in range(2)]
    for i in  range (Nbr_Checkpoints):
        Interm_Points_Array[0][i] = X
        Interm_Points_Array[1][i] = Y_end  - (Interpolation_Step * (i+1))

    return  Interm_Points_Array[0],  Interm_Points_Array[1]

def section_divider(Panel_Width, Robot_Width ):
    Eucl = divmod(Panel_Width,Robot_Width)
    reste = Eucl[1]
    while (reste != 0):
        Robot_Width -=1
        Eucl = divmod(Panel_Width,Robot_Width)
        reste = Eucl[1]
    result = int(Panel_Width/Robot_Width)
    return result, Robot_Width
```

Figure 4: Generating the checkpoints

Combining the checkpoints:

```
for i in range (Nbr_sections):
    Parity = divmod(i,2)
    if (Parity[1]==0): #even
        #start
        Trajectory_Points_Array[0][i*Nbr_Targets]= Start_Points_Array[0][i]
        Trajectory_Points_Array[1][i*Nbr_Targets]= Start_Points_Array[1][i]
        #checkpints
        Interm_X, Interm_Y = IntermPoints_Generator_even(Start_Points_Array[0][i], Nbr_Checkpoints,
                                                         Interpolation_Step, Y_start)

        Trajectory_Points_Array[0][(i*Nbr_Targets)+1: (i*Nbr_Targets)+Nbr_Checkpoints+1]= Interm_X
        Trajectory_Points_Array[1][(i*Nbr_Targets)+1: (i*Nbr_Targets)+Nbr_Checkpoints+1]= Interm_Y
        #end
        Trajectory_Points_Array[0][(i*Nbr_Targets)+Nbr_Checkpoints+1]= End_Points_Array[0][i]
        Trajectory_Points_Array[1][(i*Nbr_Targets)+Nbr_Checkpoints+1]= End_Points_Array[1][i]

        #commuting points
        StartCommuting_point[0][i]= End_Points_Array[0][i]
        StartCommuting_point[1][i]= End_Points_Array[1][i] - U1 #6 meters backward
        EndCommuting_point[0][i]  = End_Points_Array[0][i] + Robot_Width
        EndCommuting_point[1][i]  = End_Points_Array[1][i] - U2

        Trajectory_Points_Array[0][(i*Nbr_Targets)+Nbr_Checkpoints+2] = StartCommuting_point[0][i]
        Trajectory_Points_Array[0][(i*Nbr_Targets)+Nbr_Checkpoints+3] = EndCommuting_point[0][i]
        Trajectory_Points_Array[1][(i*Nbr_Targets)+Nbr_Checkpoints+2] = StartCommuting_point[1][i]
        Trajectory_Points_Array[1][(i*Nbr_Targets)+Nbr_Checkpoints+3] = EndCommuting_point[1][i]
```

Figure 5: Creating the checkpoints' array

- **Simulation Results :**

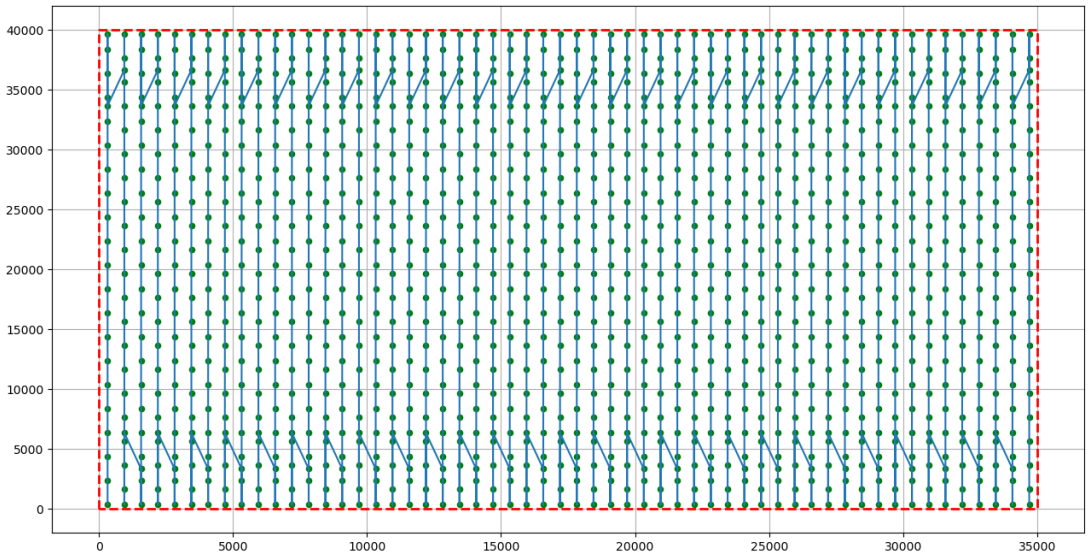


Figure 6: Simulation Results

Comments : the red frame represents the panel's limits and the green dots are the checkpoints delivered to the robot to navigate to.

3.3 Algorithm2: Trajectory Planner with Obstacles

This algorithm is dedicated to solar panels that present obstacles interrupting their continuity (figure 6)

Figure 7: Solar panel on a warehouse's rooftop presenting obstacles

I developed the first version of the algorithm that considers panels with aligned obstacles having the same width. During the configuration and setup of the environment, an operator is responsible for saving the panel and the obstacles coordinates. This data is saved in a Json file and delivered to the algorithm which processes them. The algorithm sorts the obstacles based on ascendant 'x', then, it "scans" the panel and simultaneously divides the panel into small sections limited by the obstacles. For each section, the planned trajectory is same as in algorithm 1.

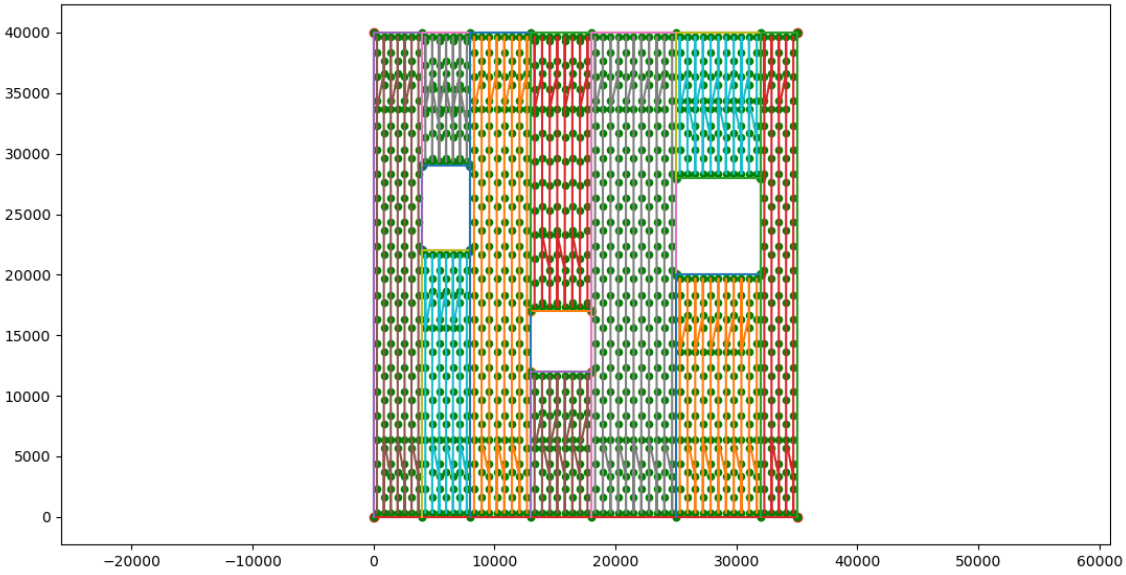


Figure 8: simulation results of algorithm2

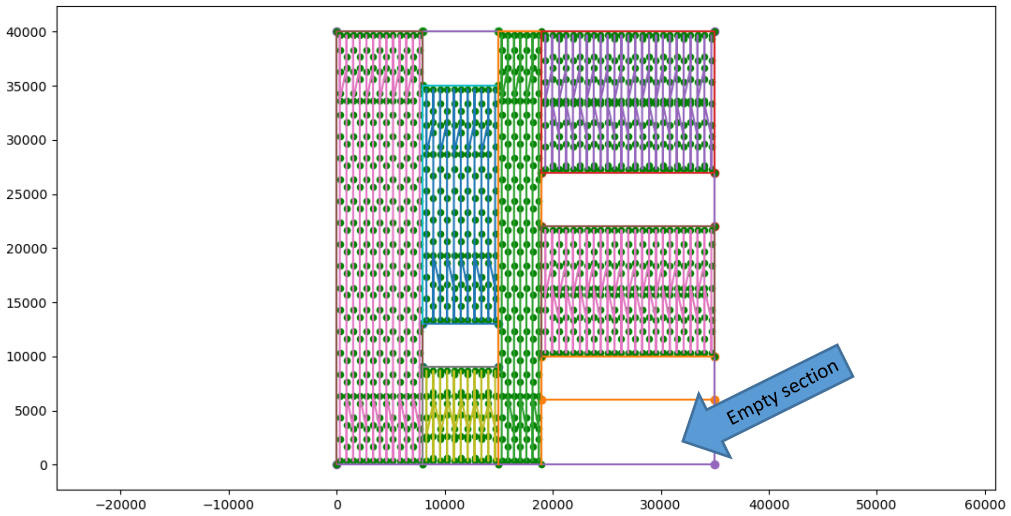


Figure 9: Aligned obstacles and empty sections

Comments : the figure above shows the section division in the case of aligned obstacles. It also shows an empty section : that is due to the fact that the lateral motion transition requires a 6-meter range in order to be executed, if the section's height is less than 6 meters, the section will not be considered in this algorithm.

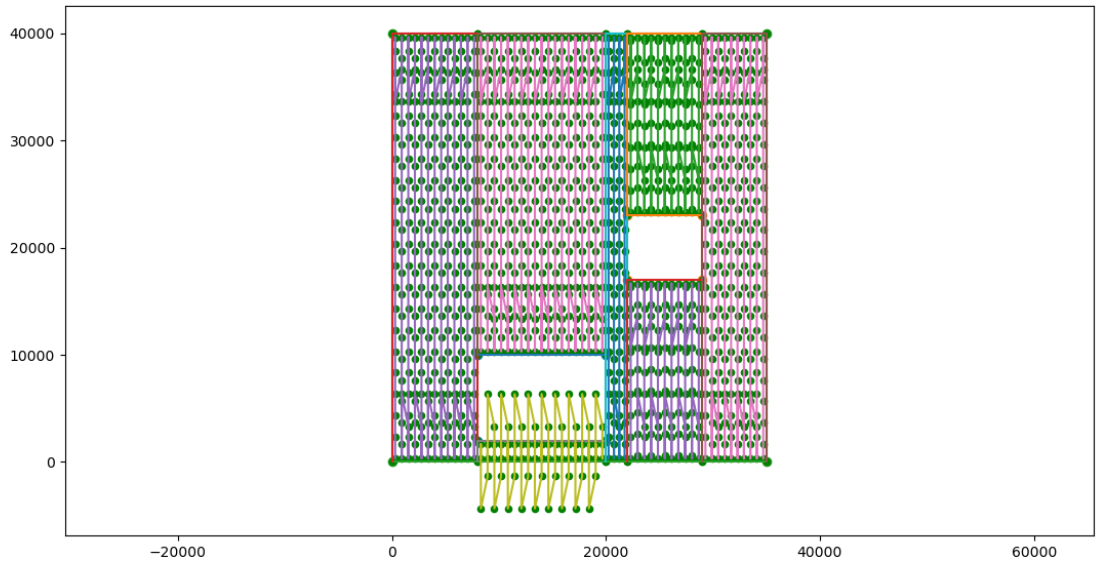


Figure 10: problem in simulation

- **Json obstacles input:**

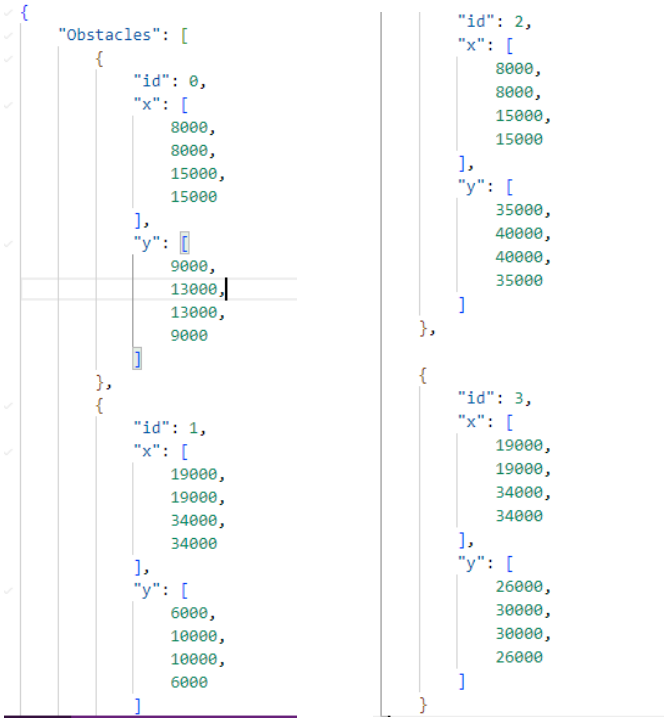


Figure 11: Json obstacles input

- **Code sections:**

```
def occurence (i,array):
    global occurence
    for j in range (i+1,len(array)-1):
        if (array[i].x_coordinates[0] == array[j].x_coordinates[0]):

            occurence = Obstacle (array[i].id, array[i].x_coordinates, array[i].y_coordinates)
            occurence.append(array[j])

            for i in range(len(occurence)):
                for j in range(len(occurence)-1):
                    if occurence[j].y_coordinates[0] > occurence[j+1].y_coordinates[0] :
                        occurence[j], occurence[j+1] = occurence[j+1], occurence[j]

    return occurence
```

Figure 11: Calculating obstacles number

```
def division (obstacles_list,panel_length, panel_width):

    global cells_list
    global obstacle_x # ARRAY OF ALL OBSTACLES WITHOUT RECURRENCE
    global obstacle_x_occurence # NUMBER OF OCCURENCES OF X
    Minimum_CellLength = 6684

    compteur = []
    inc = 0
    for i in range (len(obstacle_x_occurence)):
        inc += obstacle_x_occurence[i]
        compteur.append(inc)

    start_x = 0
    start_y = 0
    offset = 0
    for j in range (len(obstacle_x_occurence)):
        if (obstacles_list[offset].x_coordinates[0] != start_x):
            if ((panel_length - start_y) > Minimum_CellLength):
                adjacent_cell = Cell ([start_x, start_x, obstacles_list[offset].x_coordinates[0],obstacles_list[offset].x_coordinates[0]],
                                      [start_y,panel_length,panel_length, start_y] )
                cells_list.append(adjacent_cell)
                Check_overlap (obstacles_list, adjacent_cell, offset)

        if (obstacles_list[offset].y_coordinates[0] != start_y):
            if ((obstacles_list[offset].y_coordinates[0] - start_y) > Minimum_CellLength):
                below_cell = Cell([obstacles_list[offset].x_coordinates[0],obstacles_list[offset].x_coordinates[0],
                                   obstacles_list[offset].x_coordinates[2],obstacles_list[offset].x_coordinates[2]],
                                   [start_y,obstacles_list[offset].y_coordinates[0], obstacles_list[offset].y_coordinates[0],  start_y])
                cells_list.append(below_cell)
                Check_overlap (obstacles_list, below_cell, offset)
```

```
    if (obstacles_list[offset+ obstacle_x_occurence[j]-1].y_coordinates[1] != panel_length):
        if ((panel_length - (obstacles_list[offset+ obstacle_x_occurence[j]-1].y_coordinates[1])) > Minimum_CellLength):
            above_cell = Cell([obstacles_list[offset+ obstacle_x_occurence[j]-1].x_coordinates[0],obstacles_list[offset+ obstacle_x_occurence[j]-1].x_coordinates[0],
                               obstacles_list[offset+ obstacle_x_occurence[j]-1].x_coordinates[2],obstacles_list[offset+ obstacle_x_occurence[j]-1].x_coordinates[2]],
                              [obstacles_list[offset+ obstacle_x_occurence[j]-1].y_coordinates[1],panel_length,
                               panel_length,obstacles_list[offset+ obstacle_x_occurence[j]-1].y_coordinates[1]])
            cells_list.append(above_cell)
            Check_overlap (obstacles_list, above_cell, offset)

    if (obstacle_x_occurence[j]>1):
        for i in range(offset, offset + obstacle_x_occurence[j]-1):
            if ((obstacles_list[i+1].y_coordinates[0]- obstacles_list[i].y_coordinates[1]) > Minimum_CellLength):
                inter_Cell = Cell ([obstacles_list[i].x_coordinates[1],obstacles_list[i].x_coordinates[1],
                                    obstacles_list[i].x_coordinates[2],obstacles_list[i].x_coordinates[2]],
                                   [obstacles_list[i].y_coordinates[1], obstacles_list[i+1].y_coordinates[0],
                                    obstacles_list[i+1].y_coordinates[0], obstacles_list[i].y_coordinates[1]])
                cells_list.append(inter_Cell)
                Check_overlap (obstacles_list, inter_Cell, offset)

    start_x = obstacles_list[offset].x_coordinates[2]
    # last instruction in one j loop
    offset += obstacle_x_occurence[j] #Increments by the current x's number of occurences.

if (start_x < panel_width):
    final_cell = Cell ([start_x, start_x, panel_width, panel_width],
                       [start_y,panel_length, panel_length, start_y])
    cells_list.append (final_cell)

return cells_list
```

Figure 12: Trajectory planning

3.4 Algorithm3: description + figure, code, results, perspectives

This algorithm aims to track the swept area by the robot in order to keep traceability of the cleaned areas, be aware of the non-cleaned areas, and save them for a second sweep. It is also useful to keep track of the robot's performance.

- **Code** :

  In order to do so, I simulated the trajectory using a Sine function and used the orthogonal to the tangent to simulate the swept area

```
for i in range(len(xfunc)-1):
    a=(yfunc[i+1]-yfunc[i])/(xfunc[i+1]-xfunc[i])
    xc=xfunc[i]
    yc=yfunc[i]
    x1=-a*d/np.sqrt(a**2 + 1) + xc
    y1=d/np.sqrt(a**2 + 1) + yc
    x2=a*d/np.sqrt(a**2 + 1) + xc
    y2=-d/np.sqrt(a**2 + 1) + yc

    X1.append(x1)
    X2.append(x2)
    Y1.append(y1)
    Y2.append(y2)



    # plt.plot([xfunc[i]-d,xfunc[i]+d],[a*(xfunc[i]-d-xfunc[i])+yfunc[i],a*(xfunc[i]+d-xfunc[i])+yfunc[i]])
    # plt.plot([xfunc[i]-d,xfunc[i]+d],[-1/a*(xfunc[i]-d-xfunc[i])+yfunc[i],-1/a*(xfunc[i]+d-xfunc[i])+yfunc[i]],color='green') #perpendiculire

    #plt.scatter(solution[0][0], solution[0][1], )
    #plt.scatter(solution[1][0], solution[1][1])
X2=X2[::-1]
Y2=Y2[::-1]
plt.plot(X1+X2+[X1[0]],Y1+Y2+[Y1[0]])
plt.fill(X1+X2,Y1+Y2,color=(0,1,1,0.5))

plt.grid('both')
plt.axis('equal')
plt.show()
```

Figure 13: Algorithm3 code section

- **Results:**

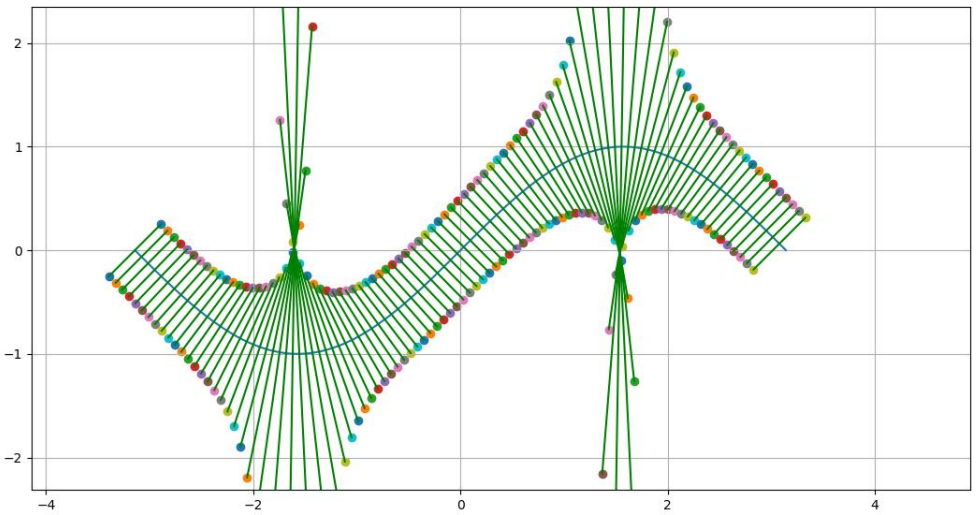


Figure 14: Result without an offset

Comment: the offset is mandatory as it limits the line simulating the robot's width.

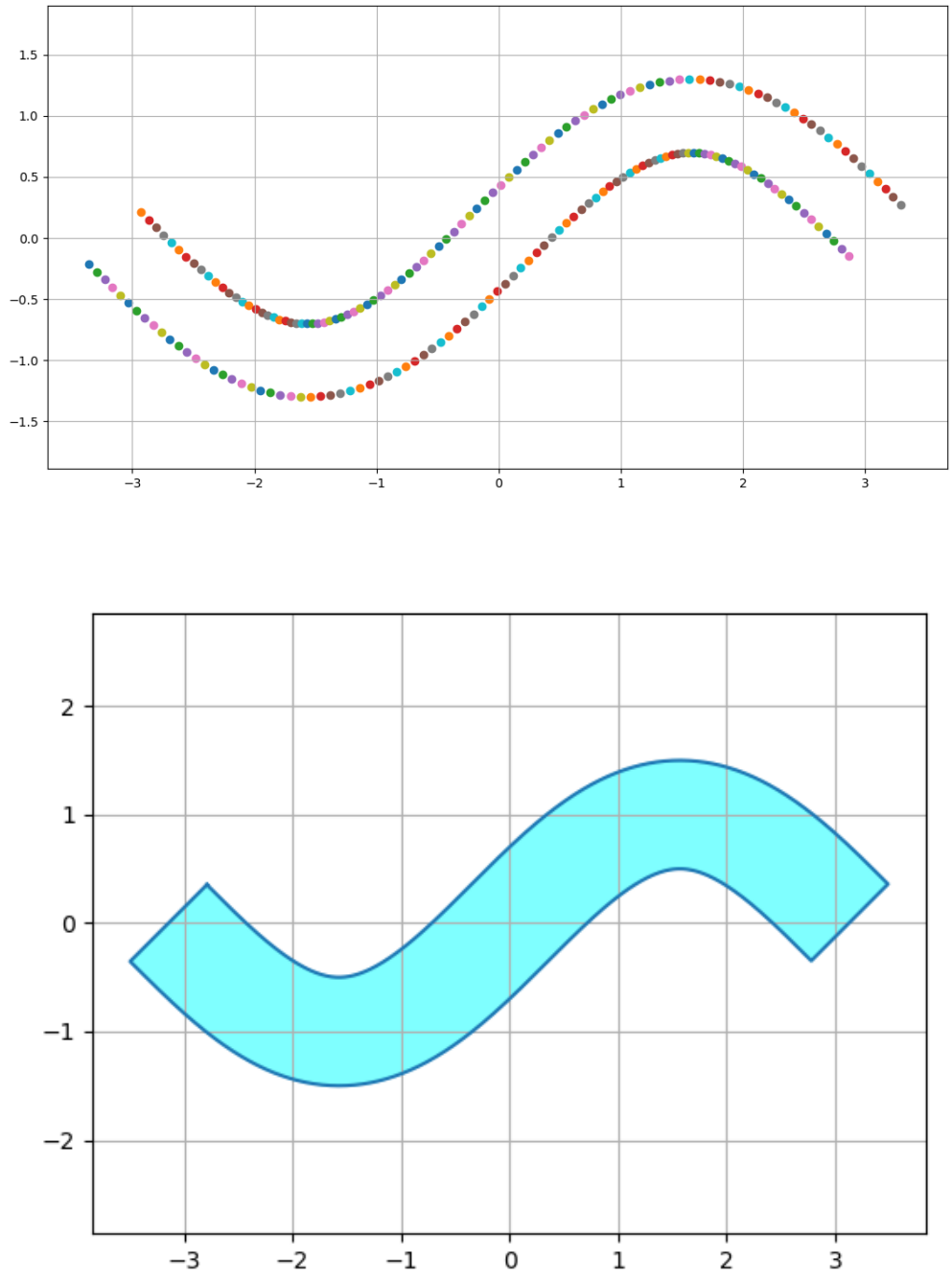


Figure 15: Results using an offset

3.5 Algorithm4: description, description base

This algorithms aims to correct the coordinates delivered by the robot's station: the base sends (x,y,z) coordinates, given the incline of the panel and the fact that the robot uses the (x,y) out of the (x,y,z) the resulting shape is not a rectangle but rather an irregular shape. We need to correct that shape to avoid undesired behaviors of the robot. In order to do so we fix one side of the resulting shape and build a rectangle out of it using orthogonals.

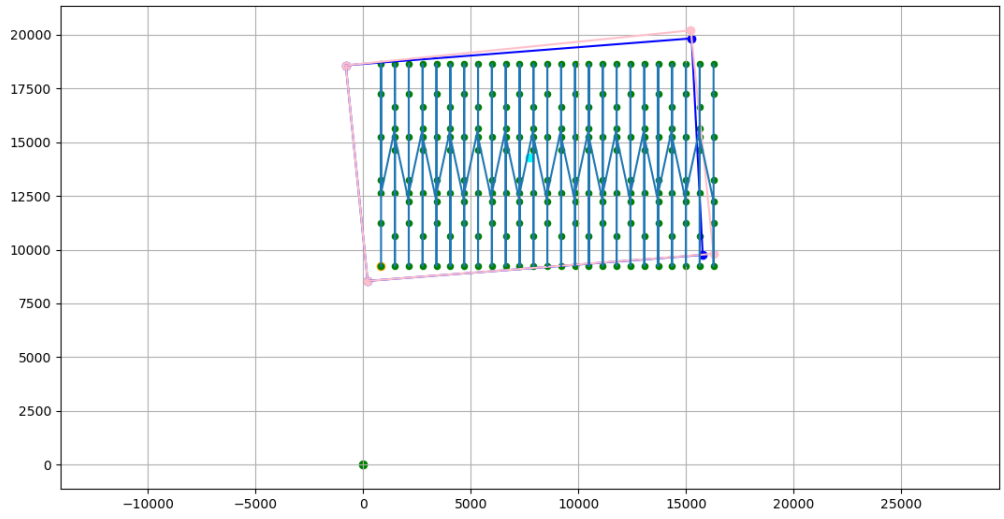


Figure 16: Corrected blue shape out of the pink shape

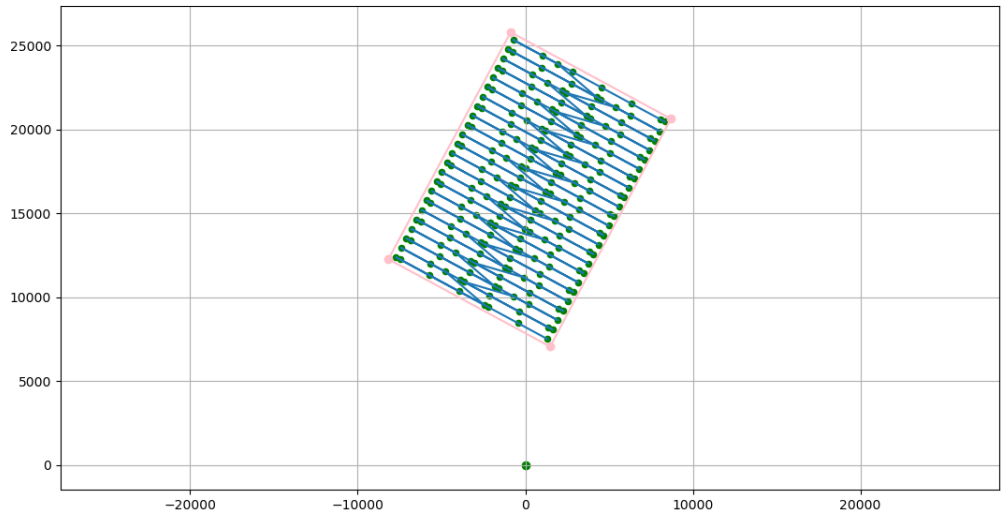


Figure 17: Checkpoints correction

3.6 Conclusion and perspectives :

Given that these re simulations of future features to be implemented on the second version of the robot, there are many areas of improvements and potential additions like:

- Refining the trajectory tracking code to localize the non-swept areas and measure the robot performance by comparing the swept area with the total one.
- Improving the trajectory generator in presence of obstacles to work with any kind and positioning of obstacles.
- Adding the links between the different sections in algorithm 2.
- Integrating water cleaning methods as they are more efficient in some cases.

4. General conclusion:

This internship was equally successful and enriching. I was honored to join an evolving team like Enova Robotics and to be involved in their dynamics. On the technical level, I learned to utilize new python libraries and to improve my code structure. I received quality supervision and mentoring which pushes me to recommend the company to my colleagues. Overall, this experience enhanced my willingness to dive deeper into the field of robotics.