

# INF421 PI: PATH PLANNING ALGORITHMS

Merouane Benadda, Antoine Fèvre

February 13, 2026

**Question 1.** We chose to implement the algorithm in C++, as it allows better performance. For visualization, we used Python with Matplotlib, as it provides a more convenient way to create visual representations of the results.

**Question 2.** See the implementation in the file `scripts/visualize.py`.

**Question 3.** See the implementation in the file `src/pso.cpp`.

**Question 4.** See the implementation in the file `src/pso.cpp`.

**Question 5.**

---

**Algorithm 1:** Particle Swarm Optimization for Path Planning

---

**Input :** Problem scenario  $P$ , iterations  $I$ , constants  $c_1, c_2, w$

**Output:** Global best path  $g_{best}$  and its cost  $f(g_{best})$

```
1 Initialize  $g_{best}$  with the first particle's waypoints
2 for  $iter = 1$  to  $I$  do
    // Evaluate Fitness and Update Memory
3   foreach particle  $p$  in swarm do
4      $cost \leftarrow \text{fitness}(p.\text{waypoints}, P)$ 
5     if  $cost < p.\text{best\_cost}$  then
6        $p.\text{best\_cost} \leftarrow cost$ 
7        $p.\text{best\_waypoints} \leftarrow p.\text{waypoints}$ 
8     if  $cost < g_{best}.cost$  then
9        $g_{best}.cost \leftarrow cost$ 
10       $g_{best}.waypoints \leftarrow p.\text{waypoints}$ 
    // Update Kinematics
11   foreach particle  $p$  in swarm do
12     for each waypoint  $i$  in path do
13        $r_1, r_2 \leftarrow \text{random}(0, 1)$ 
14       // Velocity Update
15        $v_i \leftarrow w \cdot v_i + c_1 \cdot r_1 \cdot (p.\text{best\_waypoints}_i - p.\text{waypoints}_i) + c_2 \cdot$ 
16        $r_2 \cdot (g_{best}.waypoints_i - p.\text{waypoints}_i)$ 
17       // Position Update
18        $p.\text{waypoints}_i \leftarrow p.\text{waypoints}_i + v_i$ 
19       // Boundary Constraint
20        $p.\text{waypoints}_i \leftarrow \text{clamp}(p.\text{waypoints}_i, P.\text{min}, P.\text{max})$ 
17 return  $g_{best}$ 
```

---

**Question 6.** The complexity of the algorithm is  $O(I \cdot N \cdot M)$ , where  $I$  is the number of iterations,  $N$  is the number of particles in the swarm, and  $M$  is the number of waypoints in each particle's path.

Indeed, for each iteration, we evaluate the fitness of each particle, which takes  $O(N \cdot M)$  time, since we compute the fitness by iterating over each waypoint in each particle's path. Then, we update the velocity and position of each particle, which also takes  $O(N \cdot M)$  time, as we need to update each waypoint for each particle. Therefore, the overall complexity is  $O(I \cdot N \cdot M)$ .

**Question 7.** See the implementation in the file `src/pso.cpp`.

**Question 8.**

**Scenario 0**

**Scenario 1**

**Scenario 2**

**Scenario 3**

#### **Scenario 4**

**Question 9.**

**Question 10.**

**Question 11.**

**Question 12.** To improve the performance of the algorithm, we could refine the fitness function. Instead of simply being the euclidian distance plus infinity if the path crosses an obstacle, we can make it output the euclidian distance plus a penalty proportional to the distance crossed in the obstacles.

This way, the algorithm would converge faster, especially in cases with lots of obstacles, as it would be able to differentiate between paths that are close to the optimal one but cross an obstacle and paths that are far from the optimal one but do not cross any obstacle.

To do so, we compute the intersection of the path with the obstacles in `src/utls.cpp`. To avoid any situation where the algorithm would prefer crossing an obstacle to reduce the distance, we can set the penalty to be a large constant multiplied by the distance crossed in the obstacle.

**Question 13.**

**Question 14.**

**Question 15.**

**Question 16.**

**Question 17.**

**Question 18.**

**Question 19.**

**Question 20.**

**Question 21.**

**Question 22.**

**Question 23.**

**Question 24.**

**Question 25.** We use Github for version control and to share our code. It is a very convenient tool for collaboration, as it allows us to easily track changes, manage branches, and review each other's code.

We also used GitHub Copilot to help us with some of the redundant code, such as the verification of the input or the geometry functions. It is a very useful tool for increasing productivity, as it can generate code snippets based on the context, which saves us time and allows us to focus on the more complex parts of the implementation. However, we used it very carefully, as it can sometimes generate incorrect code, so we always reviewed the generated code line by line to ensure its correctness.