

# INF421 PI: PATH PLANNING ALGORITHMS

Merouane Benadda, Antoine Fèvre

February 15, 2026

In this project, we implemented two path planning algorithms: Particle Swarm Optimization (PSO) and Rapidly-exploring Random Tree (RRT). Path planning is a fundamental problem in robotics and artificial intelligence, where the goal is to find a path from a start point to a goal point while avoiding obstacles. In our case, the environment is a rectangular area with randomly placed rectangular obstacles.

**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.

The corresponding method can be found in the file `src/Problem.cpp`.

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

**Question 3.** See the implementation in the file `src/pso.cpp`. Our fitness function is defined as the euclidian distance of the path, plus infinity if the path crosses an obstacle. This way, the algorithm will try to find a valid path that minimizes the distance to the goal point.

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

**Question 5.** See Algorithm 1 for the pseudocode in the appendix.

**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`. We implemented the algorithm following the pseudocode given for question 5. We initialize the particles with random waypoints, and then we iteratively evaluate their fitness, update their memory of the best position, and update their velocity and position according to the PSO equations. We also ensure that the waypoints

stay within the bounds of the environment by clamping their coordinates after each update.

**Question 8.** For the basic PSO algorithm, we had to pick an important number of particles because, since our fitness function is quite simple (the euclidian distance plus infinity if the path crosses an obstacle), finding a first valid path is a matter of luck, and the more particles we have, the more likely we are to find a valid path that we can optimize.

As we can see in Figure 1, the solutions are not always optimal.

```
const int NUM_PARTICLES = 100000;
const int NUM_WAYPOINTS = 5;
const int NUM_ITERATIONS = 1000;
const double C1 = 2.0; // cognitive coefficient
const double C2 = 2.0; // social coefficient
const double W = 0.75; // inertia weight
```

#### Question 9.

The random restart method allows us to reduce the number of particles needed to find a valid path, as it gives the algorithm multiple chances to find a valid path by restarting the particles' positions after a certain number of iterations. This way, even if we start with a small number of particles, we can still find a valid path by giving the algorithm multiple attempts. The restart interval is sufficiently large to allow the particles to explore the search space and converge towards a good solution before being restarted.

```
const int NUM_PARTICLES = 1000;
const int NUM_WAYPOINTS = 5;
const int NUM_ITERATIONS = 50000;
const int RESTART_INTERVAL = 2500;
```

**Question 10.** The cooling rate makes the temperature stay high for a longer time, which allows the algorithm to explore the search space more thoroughly and avoid getting stuck in local minima. We chose the initial temperature empirically, by testing different values and observing the convergence of the algorithm.

```
const int NUM_PARTICLES = 1000;
const int NUM_WAYPOINTS = 5;
const int NUM_ITERATIONS = 30000;
const int RESTART_INTERVAL = 3000;
double initial_temperature = 100.0;
double cooling_rate = 0.99;
```

#### Question 11.

We chose a small stagnation threshold so the particles can quickly benefit from the dimensional learning when they are stuck in a local minimum. Since

our fitness function is quite simple, it is easy for the particles to get stuck in local minima, especially in scenarios with many obstacles. By applying dimensional learning after a small number of iterations without improvement, we allow the particles to escape these local minima and explore new areas of the search space, which can lead to better solutions.

```
const int NUM_PARTICLES = 500;
const int NUM_WAYPOINTS = 5;
const int NUM_ITERATIONS = 30000;
const int RESTART_INTERVAL = 5000;
double initial_temperature = 100.0;
double cooling_rate = 0.99;
int stagnation_threshold = 15;
```

**Question 12.** To improve the performance of the algorithm, we can 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/utils.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.

```
const int NUM_PARTICLES = 200;
const int NUM_WAYPOINTS = 5;
const int NUM_ITERATIONS = 30000;
const int RESTART_INTERVAL = 2500;
double initial_temperature = 100.0;
double cooling_rate = 0.999;
int stagnation_threshold = 15;
```

Table 1: Performance Comparison across All Test Scenarios and Methods

Scenario	Basic PSO		Random Restart		Annealing PSO		Dim. Learning		DL + Refined Fitness	
	Cost	Time (s)	Cost	Time (s)	Cost	Time (s)	Cost	Time (s)	Cost	Time (s)
0	1443.33	22.72	1443.33	10.65	1443.33	7.82	1443.33	5.00	1443.33	1.29
1	1449.06	31.06	1449.06	15.17	1449.06	10.70	1449.06	7.38	1449.12	1.45
2	1522.56	37.66	1505.19	18.20	1505.19	12.99	1505.65	7.32	1505.19	1.91
3	1524.98	34.25	1522.54	16.23	1522.90	10.43	1522.52	6.36	1522.86	2.23
4	1961.92	35.89	1946.06	16.81	1948.74	10.55	1946.45	6.33	1941.36	2.30

As we can see in Table 1, adding the different improvements to the basic PSO algorithm allows us to significantly reduce the cost of the solution and the

time needed to find it. The final algorithm outputs a path that is very close to the optimal one, and does so in a reasonable amount of time, even in the most complex scenarios.

#### **Question 13.**

To represent the tree, we use three vectors. For each vertex index, we have the corresponding point in the `vertices` vector, the index of its parent vertex in the `parents` vector, and the distance of the path from the root to the vertex in the `costs` vector. This way, we can easily reconstruct the path from any vertex to the root by following the parent indices, and efficiently compute the length of any path in the tree when updating parents in the rewiring process. See the implementation in the file `src/RRT.hpp`.

#### **Question 14.**

To correspond to the conventions taken before, the method gives the vertices coordinates, from the root to the goal, excluding the extremities. See the implementation in the files `src/RRT.hpp` and `src/RRT.cpp`.

#### **Question 15.**

Most of the work is done by `BuildRRT` (Algorithm 3), which builds the tree. `RRTPath` (Algorithm 4) only calls `BuildRRT`, and then reconstructs the path with `reconstructPath` implemented in Question 14.

The method `addVertex` (Algorithm 2) is used to update the tree, given a point and the parent vertex.

#### **Question 16.**

For one iteration of the while loop of the algorithm `BuildRRT`, the time complexity is a  $O(NM)$ , where  $N$  is the number of nodes in the tree and  $M$  the number of obstacles. Indeed, the two loops at lines 16-20 and 25-30 take at most that amount of time, as in the worst case all vertices are at a distance to  $v$  lower than  $\delta_s$ , and all obstacles have to be tried. All other instructions in an iteration have a strictly lower complexity than that.

If a tree is built before reaching the maximum number of iterations  $I_{max}$ , at the iteration  $i$ , there are less than  $i$  vertices in the tree, as there is at most one node created by iteration. That is why, overall, the time complexity is  $O(MI^2)$ , where  $I$  iterations are necessary to build a tree reaching the goal point. Indeed `reconstructTree` takes at most  $O(I)$  time in execution.

The hyperparameter  $\delta_s$  has an impact on real time execution. Increasing it might reduce the number of iterations needed, as each one makes a more significant advance. However, this increases the complexity at each iteration because more vertices will be at a lower distance from  $v$  than  $\delta_s$ .

#### **Question 17.**

See the implementation in the file `RRT.cpp`.

#### **Question 18.**

I initially chose, for the hyperparameters  $\delta_s = 100$  and  $\delta_r = 100$ . Considering the size of the zone (1000\*1000), it seemed appropriate. After trying on different

scenarios, it appears that increasing  $\delta_s$  gives best results in areas with few obstacles.

Here are the results for  $\delta_s = 100$  and  $\delta_r = 100$ :

Table 2: RRT results for different scenarios

Scenario	Path length	Iterations	CPU Time (s)
0	1742.09	224	0
1	1718.12	533	0.002
2	1739.89	242	0.001
3	1757.14	410	0.004
4	2186.43	721	0.009

#### Question 19.

The path built are made of segments of length at most  $\delta_s$ . However, there is sometimes a free space longer than  $\delta_s$  to cross, which is made by assembling multiple segments. As these segments do not have the same direction, the path is not optimal, according to the triangle inequality. Indeed, it is more efficient to suppress intermediate nodes in the path, if the lines created do not cross any obstacle.

To do so, we keep the same path building method, but we improve the returned path by suppressing intermediate nodes.

See the implementation in the method `optimizePath`, in file `src/RRT.cpp`.

Thus the complexity of the algorithm remains the same, as the optimization step is done in  $O(I)$  time, where  $I$  is the number of iterations needed to build the tree, which is negligible compared to the complexity of building the tree. This is confirmed by the results, the computation time of optimization is equal to 0s in all scenarios.

#### Question 20.

Results are shown in Table 3. For the RRT, as the path optimization complexity is negligible compared to the tree building, results shown come from the same tree building. Moreover, this allows to compare the path length before and after optimization on one single tree, making the comparison more relevant. The metrics for PSO are taken from the DL + Refined fitness version.

#### Question 21.

#### Question 22.

#### Question 23.

#### Question 24.

**Question 25.** We used 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.

Table 3: Performance comparison between PSO and RRT (without and with optimized path)

Scenario	Path length			CPU time (s)	
	PSO	RRT	RRT (optimized path)	PSO	RRT
0	1443.33	1785.36	1437.43	1.29	0
1	1449.12	1810.40	1433.18	1.45	0.002
2	1505.19	1975.92	1525.40	1.91	0
3	1522.86	1643.85	1515.43	2.23	0
4	1941.36	2319.06	2006.87	2.30	0.016

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.

## A Pseudo-code

---

**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  $\leftarrow$  fitness( $p.waypoints, P$ )
        5   if  $cost < p.best\_cost$  then
            6      $p.best\_cost \leftarrow cost$ 
            7      $p.best\_waypoints \leftarrow p.waypoints$ 
        8   if  $cost < g_{best}.cost$  then
            9      $g_{best}.cost \leftarrow cost$ 
            10     $g_{best}.waypoints \leftarrow p.waypoints$ 
    // Update Kinematics
    11  foreach particle  $p$  in swarm do
        12    for each waypoint  $i$  in path do
            13       $r_1, r_2 \leftarrow \text{random}(0, 1)$ 
            // Velocity Update
            14       $v_i \leftarrow w \cdot v_i + c_1 \cdot r_1 \cdot (p.best\_waypoints_i - p.waypoints_i) + c_2 \cdot$ 
                   $r_2 \cdot (g_{best}.waypoints_i - p.waypoints_i)$ 
            // Position Update
            15       $p.waypoints_i \leftarrow p.waypoints_i + v_i$ 
            // Boundary Constraint
            16       $p.waypoints_i \leftarrow \text{clamp}(p.waypoints_i, P.\min, P.\max)$ 
    17 return  $g_{best}$ 

```

---

**Algorithm 2:** AddVertex - Add a vertex to the tree

---

```

1 Function ADDVERTEX( $v, parent\_index$ ):
2   Append  $v$  to  $tree.vertices$ 
3   Append  $parent\_index$  to  $tree.parents$ 
4    $cost \leftarrow tree.costs[parent\_index] + d(tree.vertices[parent\_index], v)$ 
5   Append  $cost$  to  $tree.costs$ 

```

---

---

**Algorithm 3:** BuildRRT

---

**Input** : Problem  $P$ , step size  $\delta_s$ , rewiring radius  $\delta_r$ , max iterations  $I_{max}$

**Output:** Tree connecting start to goal

```
1 iterations  $\leftarrow 0$ 
2 while iterations  $< I_{max}$  do
    // Sample Random Point
    3  $v_r \leftarrow$  random point in  $[0, P.x_{max}] \times [0, P.y_{max}]$ 
    4 if  $v_r$  is inside an obstacle then
        | continue
    // Find Nearest Vertex
    6  $v_n \leftarrow \arg \min_{v \in tree.vertices} d(v, v_r)$ 
    // Steer Toward Random Point
    7 if  $d(v_n, v_r) \leq \delta_s$  then
        |  $v \leftarrow v_r$ 
    else
        |  $\theta \leftarrow \arctan 2(v_r.y - v_n.y, v_r.x - v_n.x)$ 
        |  $v.x \leftarrow v_n.x + \delta_s \cos(\theta)$ 
        |  $v.y \leftarrow v_n.y + \delta_s \sin(\theta)$ 
    // Choose Best Parent
    13  $parent \leftarrow -1$ 
    14 if no collision between  $v_n$  and  $v$  then
        | parent  $\leftarrow v_n$ 
    16 foreach vertex  $v_i$  in  $tree.vertices$  do
        | if  $d(v_i, v) < \delta_r$  and no collision between  $v_i$  and  $v$  then
            |  $cost_{new} \leftarrow tree.costs[v_i] + d(v_i, v)$ 
            | if  $parent = -1$  or
                |  $cost_{new} < tree.costs[parent] + d(parent, v)$  then
                    | | parent  $\leftarrow v_i$ 
    21 if  $parent = -1$  then
        | continue                                // No valid parent found
    23 ADDVERTEX( $v, parent$ )
    24  $idx_v \leftarrow |tree.vertices| - 1$ 
    // Rewire Neighbors
    25 foreach vertex  $v_i$  in  $tree.vertices$  do
        | if  $d(v_i, v) < \delta_r$  and no collision between  $v_i$  and  $v$  then
            |  $cost_{via\_v} \leftarrow tree.costs[idx_v] + d(v, v_i)$ 
            | if  $tree.costs[i] > cost_{via\_v}$  then
                | |  $tree.parents[i] \leftarrow idx_v$ 
                | |  $tree.costs[i] \leftarrow cost_{via\_v}$ 
    // Check Goal Reachability
    31 if  $d(v, P.goal) \leq \delta_s$  and no collision between  $v$  and  $P.goal$  then
        | ADDVERTEX( $P.goal, idx_v$ )
        | break                                // Goal reached
    34  $iterations \leftarrow iterations + 1$ 
```

---

---

**Algorithm 4: RRTPath - Main Path Planning Function**


---

```

1 Function RRTPATH( $P, \delta_s, \delta_r, I_{max}$ ):
    Input : Problem  $P$ , step size  $\delta_s$ , radius  $\delta_r$ , iterations  $I_{max}$ 
    Output: Path from start to goal
2   BUILDRRT( $P, \delta_s, \delta_r, I_{max}$ )
3    $goal\_index \leftarrow |tree.vertices| - 1$            // Goal is last vertex
4   return RECONSTRUCTPATH( $goal\_index$ )

```

---

## B Visualization

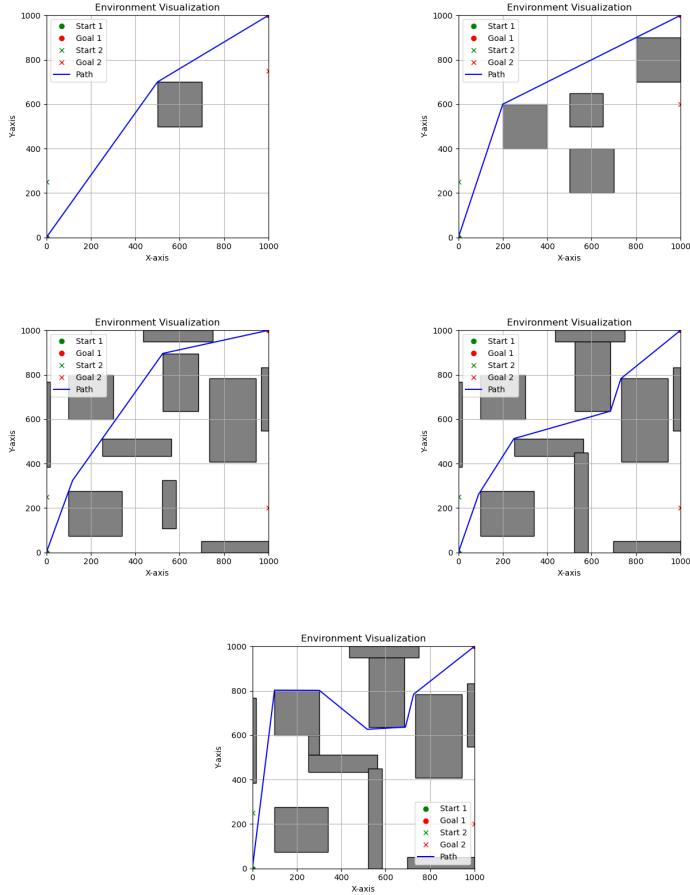


Figure 1: Test scenarios 0-4