INF421 PI: PATH PLANNING ALGORITHMS

Merouane Benadda, Antoine Fèvre

February 15, 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. See Algorithm 1 for the pseudocode.

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. 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.

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 $f(g_{best})$

Initialize g_{best} with the first particle's waypoints
for $iter = 1$ **to** I **do**
    `// Evaluate Fitness and Update Memory`
    foreach *particle p in swarm* **do**
        $cost \leftarrow \text{fitness}(p.waypoints, P)$
        if $cost < p.best\_cost$ **then**
            $p.best\_cost \leftarrow cost$
            $p.best\_waypoints \leftarrow p.waypoints$
        if $cost < g_{best}.cost$ **then**
            $g_{best}.cost \leftarrow cost$
            $g_{best}.waypoints \leftarrow p.waypoints$
    `// Update Kinematics`
    foreach *particle p in swarm* **do**
        for *each waypoint i in path* **do**
            $r_1, r_2 \leftarrow \text{random}(0, 1)$
            `// Velocity Update`
            $v_i \leftarrow w \cdot v_i + c_1 \cdot r_1 \cdot (p.best\_waypoints_i - p.waypoints_i) + c_2 \cdot r_2 \cdot (g_{best}.waypoints_i - p.waypoints_i)$
            `// Position Update`
            $p.waypoints_i \leftarrow p.waypoints_i + v_i$
            `// Boundary Constraint`
            $p.waypoints_i \leftarrow \text{clamp}(p.waypoints_i, P.\min, P.\max)$
return g_{best}

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

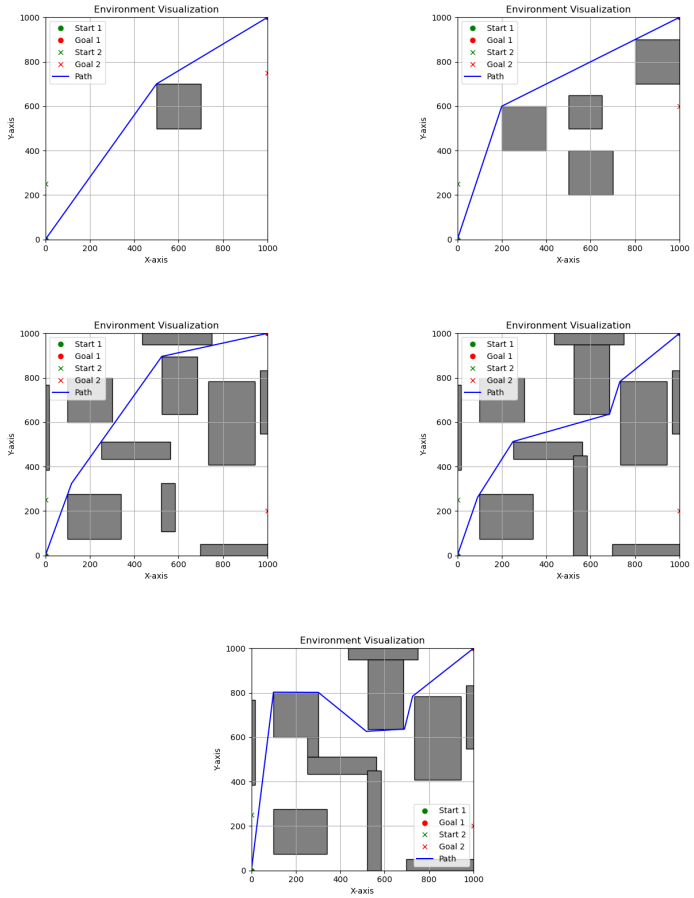

Figure 1: Test scenarios 0-4

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 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/utils.cpp`. To avoid any situation where the alogorithm 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.

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, exluding the extremities. See the implementation in the files `src/RRT.hpp` and `src/RRT.cpp`.

Algorithm 2: AddVertex

Input: Point v, index *parent_index*

Ajouter v à *tree.vertices*;
Ajouter *parent_index* à *tree.parents*;
cost $\leftarrow$ *tree.costs*[*parent_index*] + d(*tree.vertices*[*parent_index*], v);
Ajouter *cost* à *tree.costs*;

Algorithm 3: ReconstructPath

Input: Index *vertex_index*
Output: Chemin de points

path $\leftarrow \emptyset$;
current $\leftarrow$ *tree.parents*[*vertex_index*]; `// Partir du parent du goal`
while *tree.parents*[*current*] $\neq -1$ **do**
    Ajouter *tree.vertices*[*current*] à *path*;
    current $\leftarrow$ *tree.parents*[*current*];
end
Inverser *path*; `// Obtenir le chemin de start à goal`
return *path*;

Algorithm 4: BuildRRT (variante RRT*)

Input: Problème *problem*, distance δ_s, rayon δ_r, entier *max_iterations*

$iterations \leftarrow 0$;
while $iterations < max\_iterations$ **do**
  // Échantillonner un point aléatoire
  $v_r \leftarrow$ point aléatoire dans $[0, x_{max}] \times [0, y_{max}]$;
  if v_r *est dans un obstacle* **then**
    continuer;
  end
  // Trouver le sommet le plus proche
  $v_n \leftarrow \arg\min_{v \in tree.vertices} d(v, v_r)$;
  // Créer un nouveau point vers v_r à distance max δ_s
  if $d(v_n, v_r) \leq \delta_s$ **then**
    $v \leftarrow v_r$;
  else
    $\theta \leftarrow \arctan 2(v_r.y - v_n.y, v_r.x - v_n.x)$;
    $v.x \leftarrow v_n.x + \delta_s \cos(\theta)$;
    $v.y \leftarrow v_n.y + \delta_s \sin(\theta)$;
  end
  // Choisir le meilleur parent dans le rayon δ_r (RRT*)
  $parent \leftarrow$ NULL;
  if *pas de collision entre* v_n *et* v **then**
    $parent \leftarrow v_n$;
  end
  foreach $v_i \in tree.vertices$ **do**
    if $d(v_i, v) < \delta_r$ ***et*** *pas de collision entre* v_i *et* v **then**
      $cost_{new} \leftarrow tree.costs[v_i] + d(v_i, v)$;
      if $parent = NULL$ ***ou*** $cost_{new} < tree.costs[parent] + d(parent, v)$ **then**
        $parent \leftarrow v_i$;
      end
    end
  end
  if $parent = NULL$ **then**
    continuer; // Aucun parent valide trouvé
  end
  AddVertex$(v, parent)$;
  $idx_v \leftarrow |tree.vertices| - 1$;
  // Rewiring : recâbler les voisins (RRT*)
  for $i \leftarrow 0$ **to** $|tree.vertices| - 1$ **do**
    if $d(tree.vertices[i], v) < \delta_r$ ***et*** *pas de collision* **then**
      $cost_{via\_v} \leftarrow tree.costs[idx_v] + d(v, tree.vertices[i])$;
      if $tree.costs[i] > cost_{via\_v}$ **then**
        $tree.parents[i] \leftarrow idx_v$;
        $tree.costs[i] \leftarrow cost_{via\_v}$;
      end
    end
  end
  // Vérifier si le goal est atteignable
  if $d(v, goal) \leq \delta_s$ ***et*** *pas de collision entre* v *et goal* **then**
    AddVertex$(goal, idx_v)$;
    sortir; // Goal atteint
  end
  $iterations \leftarrow iterations + 1$;
end

Algorithm 5: RRTPath - Planification de chemin

Input: Problème *problem*, distance δ_s, rayon δ_r, entier *max_iterations*
Output: Chemin de points

```
1 BuildRRT(problem, δ_s, δ_r, max_iterations);
2 goal_index ← |tree.vertices| − 1; // Le goal est le dernier sommet
3 return ReconstructPath(goal_index);
```

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.