POLITECNICO
MILANO 1863

Autonomous Vehicles - Projects Report

a.y. 2024-2025

Edoardo Barutta
edoardo.barutta@mail.polimi.it
250790

Lorenzo Vignoli
lorenzo.vignoli@asp-poli.it
252085

Nicola Visentin
nicola.visentin@mail.polimi.it
252081

Contents

1 Project I

The project consists in implementing the RRT algorithm for solving a path searching problem from a start to a goal position in some provided maps. Moreover, other two variations of this method are developed, RRT-connect and RRT*, and all the results are compared along with A* algorithm in terms of computational load, time and quality of the solution.

1.1 RRT

Rapidly exploring Random Tree (RRT) is a sample-based searching algorithm that explores the space, trying to reach a goal position by building a tree from a start point, randomly sampling the space itself. This method, compared to other path-planning algorithms such as A* or other grid-based approaches, is in general much faster and less resolution-dependent, and for these reasons it is particularly suitable for exploration, dynamical environments, high-dimensional spaces or when the knowledge of the map is not complete. On the other hand, even if RRT is proven to eventually find a solution, if one exists, this solution is often non-optimal. Moreover, due to its non-deterministic nature, we can't know in advance how long will it take or how many samples will be needed to find a solution. Finally, if a solution does not exisist, RRT is not able to recognize it. Some of these drawbacks are mitigated by considering modified versions of the base algorithm: one of the most important ones is RRT*, which will be presented in section 1.3. Another relevant aspect is that the algorithm needs to be run from scratch any time the initial condition changes.

Implementation Drawing inspiration from LaValle's studies [1], RRT algorithm was implemented in MATLAB. Without entering too much into detail, the basic idea is to grow a tree starting from an initial node through an iterative procedure:

1. sample a random [1] point x_{rand} on the map and find the closest node [2] x_{near} in the tree;
2. try to extend the tree from x_{near} to x_{rand} by a certain *step* (edge validation algorithms are needed for this purpose).

The procedure is stopped when the tree is close enough to the goal and the two can be connected. At this point, starting from the end, it is possible to backward reconstruct the unique parenting chain that leads to the starting point. Pseudocode is presented in Algorithm 1.

Testing The code was tested on four provided maps, that had a resolution of 300x300 pixels and were rescaled to a dimension of 10000x10000 in order to implement the algorithm. However, calculations such that the length of the solution path were performed assuming an area of 300×300 meters.
Results are visible in *figures 1,2,3,4*: the parameters considered for performances evaluation are final path length, computation time, number of nodes in the tree and number of iterations required. Further considerations and comments on these data will be discussed in section 1.4.

[1] An heuristic correction can be imposed so that x_{rand} is more likely to be taken towards the goal; this improved the speed in finding a solution, at cost of a less uniform exploration of the space.

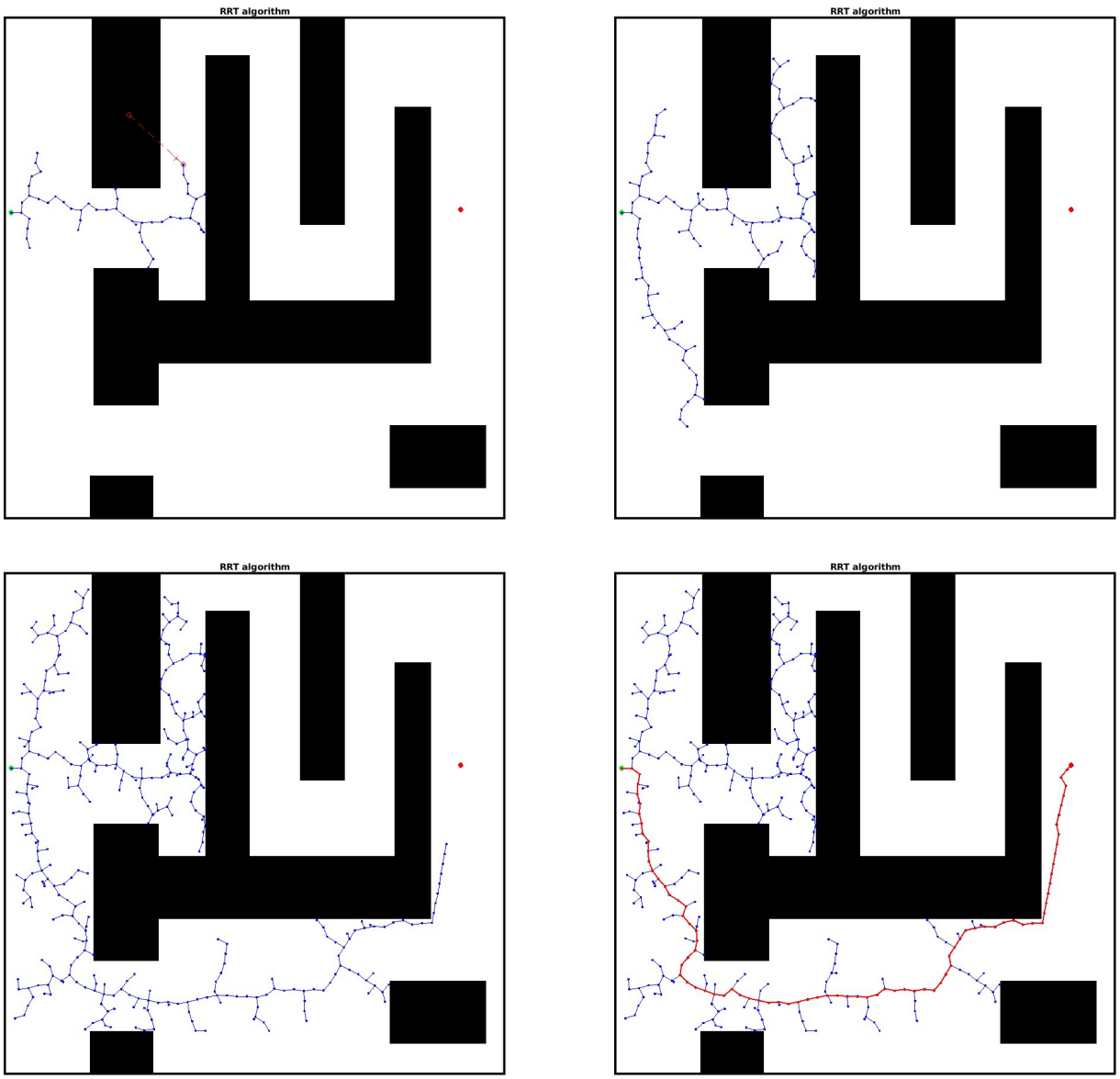
[2] Many different NEAREST_NEIGHBOR algorithms exists for this purpose.

Algorithm 1 RRT

Input: N, C, C_{free}, s, g, $bias$, δx {max iter, space, free-space, start, goal, heuristic, step}
Output: τ
τ.init(s)
for $n \leftarrow 1$ to N **do**
  $x_{rand} \leftarrow$ random point in C ($bias$ probability of being g)
  $x_{near} \leftarrow$ NEAREST_NEIGHBOR(x_{rand}, τ)
  $x_{new} \leftarrow$ NEW_STATE($x_{near}, x_{rand}, \delta x$)
  if edge(x_{new}, x_{near}) $\in C_{free}$ **then**
    τ.add_vertex(x_{new})
    τ.add_edge(x_{near}, x_{new})
    if DIST(x_{new}, g) $< \delta x$ **and** edge(x_{new}, g) $\in C_{free}$ **then**
      τ.add_vertex(g)
      τ.add_edge(x_{new}, g)
      break
    end if
  end if
end for

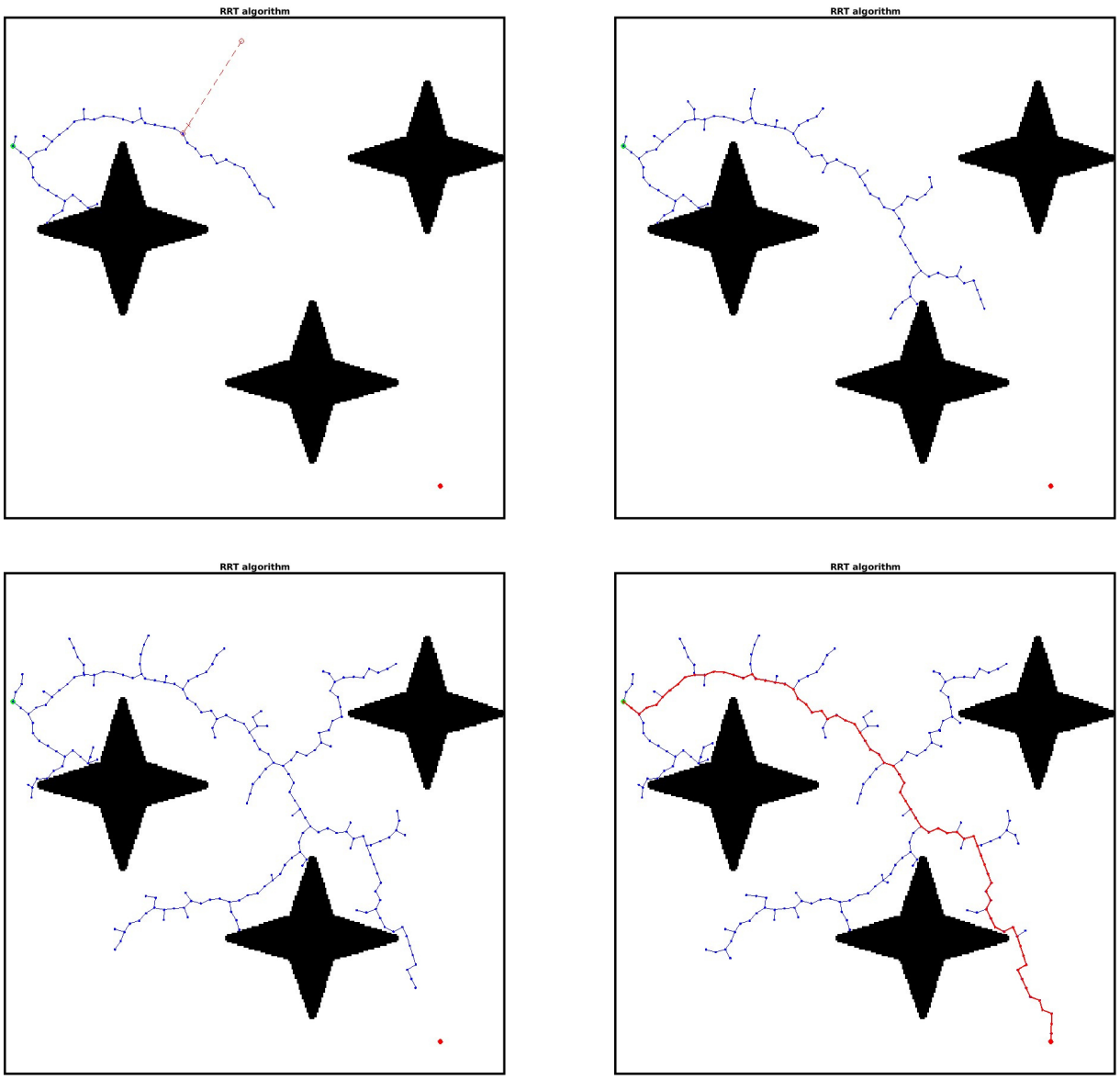
Algorithm 2 NEW_STATE

Input: x_{near}, x_{rand}, δx
Output: x_{new}
if $DIST(x_{rand}, x_{near}) < \delta x$ **then**
  $x_{new} \leftarrow x_{rand}$
else
  $x_{new} \leftarrow$ DIRECTION(x_{rand}, x_{near}) $\cdot \delta x$
end if



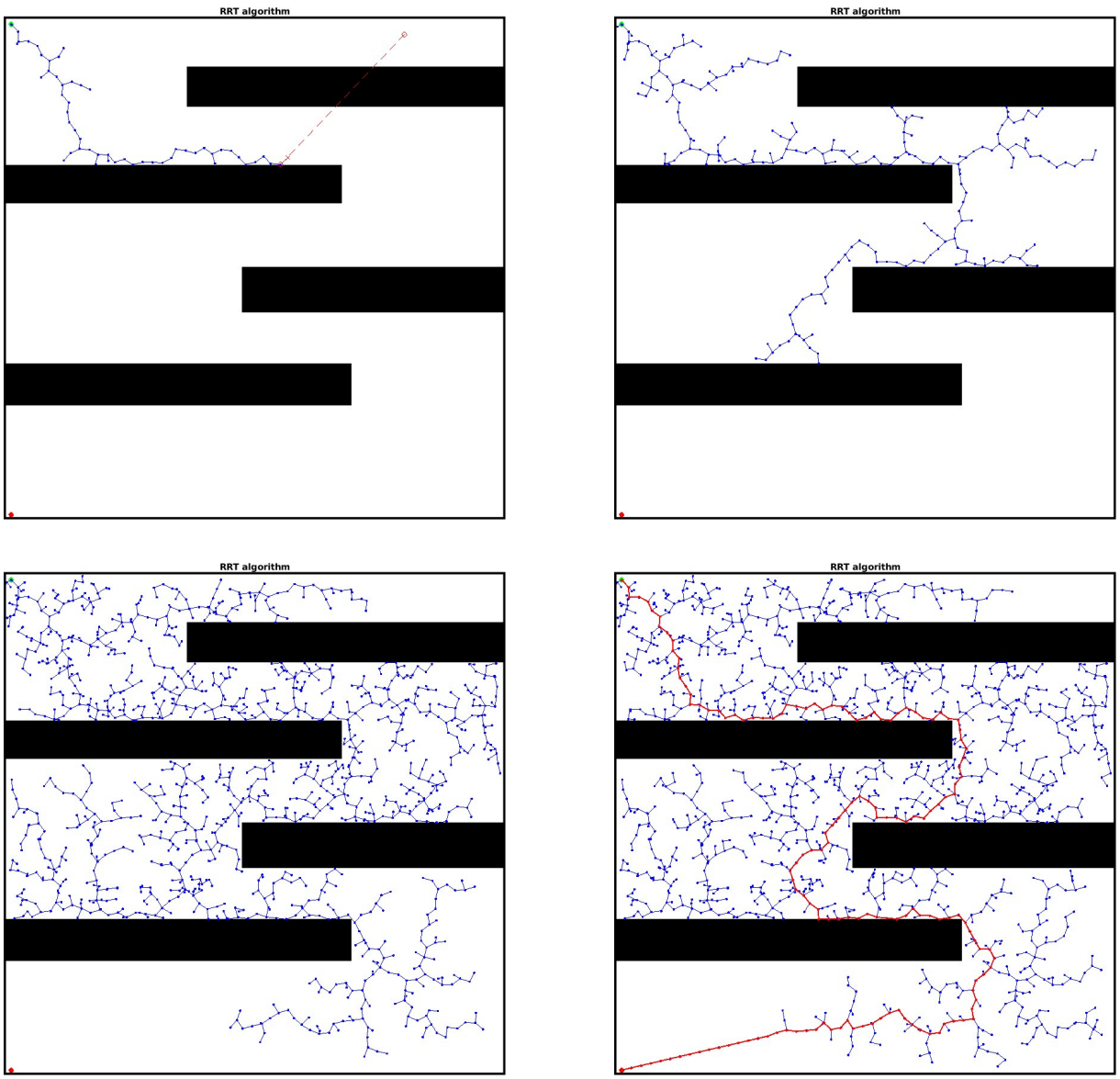
Length[m]	501.5
Time[s]	0.12
Iterations	891
Nodes	324

Figure 1: RRT algorithm on map 1



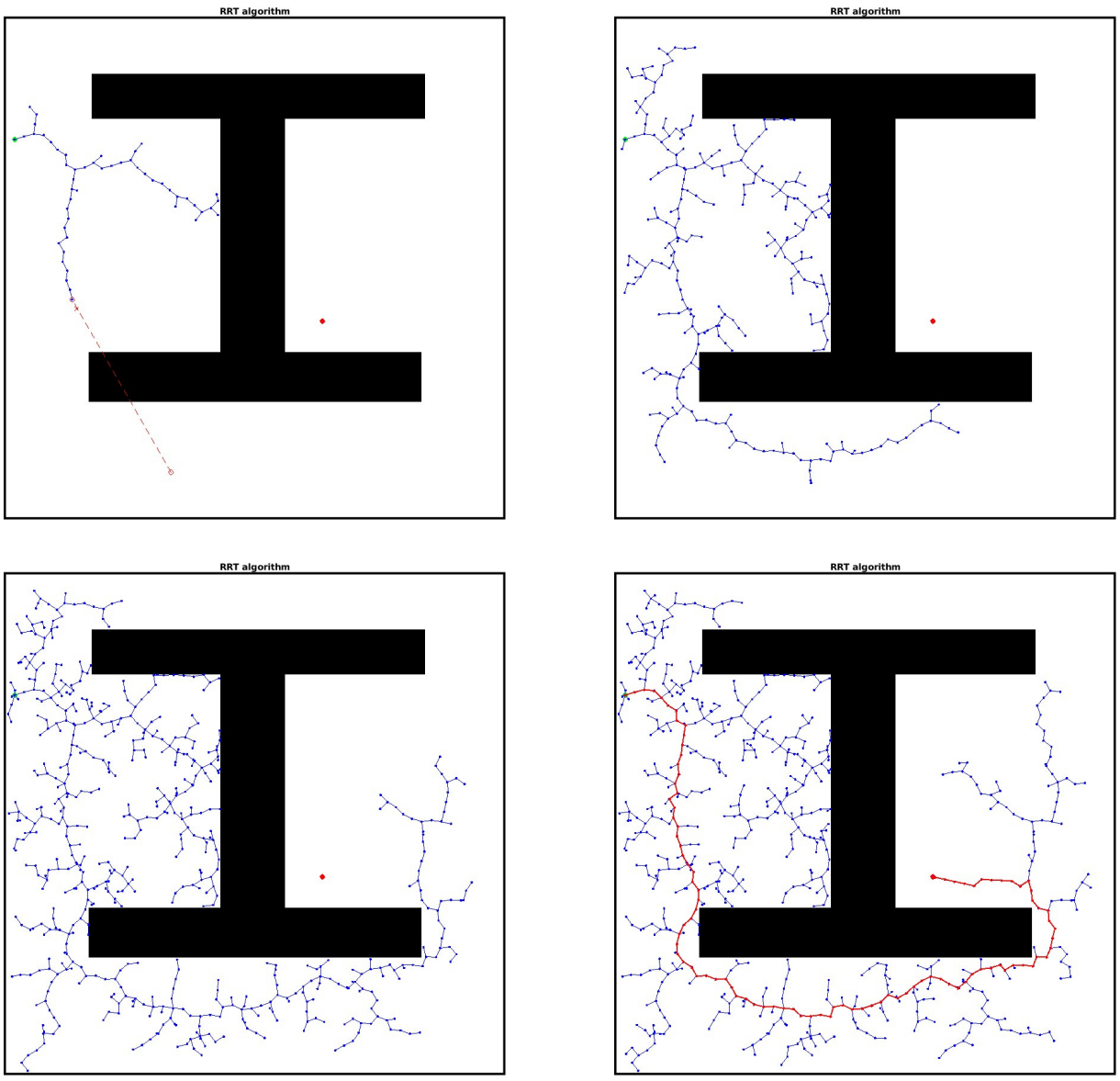
Length[m]	434.4
Time[s]	0.11
Iterations	324
Nodes	207

Figure 2: RRT algorithm on map 2



Length[m]	906.2
Time[s]	0.39
Iterations	3021
Nodes	1288

Figure 3: RRT algorithm on map 3



Length[m]	575.6
Time[s]	0.20
Iterations	1408
Nodes	630

Figure 4: RRT algorithm on map 4

1.2 RRT-connect

RRT-connect is a variation of RRT in which two trees are generated, one growing from the start position and the other from the goal. This should reduce the time required to find a feasible solution, but precludes the possibility to take into account differential constraints in the planner.

Implementation Also in this case many possible implementations are possible. In our case, RRT-connected was developed in MATLAB by drawing inspiration from [2], where an additional

heuristic correction is suggested: it consists in trying to extend the tree for a longer distance, when a new branch needs to be added. Once again, a bias was imposed on the random generation of x_{rand}, making so that the trees are "pushed" to grow towards each other. According to [2], all these strategies are shown to improve the convergence of the algorithm.
Pseudocode is presented in Algorithm 3: the only difference with RRT is that each tree is attempted to be extended every two cycles, since now we have two trees. A check is made at each iteration to see if the two tress can be connected together.

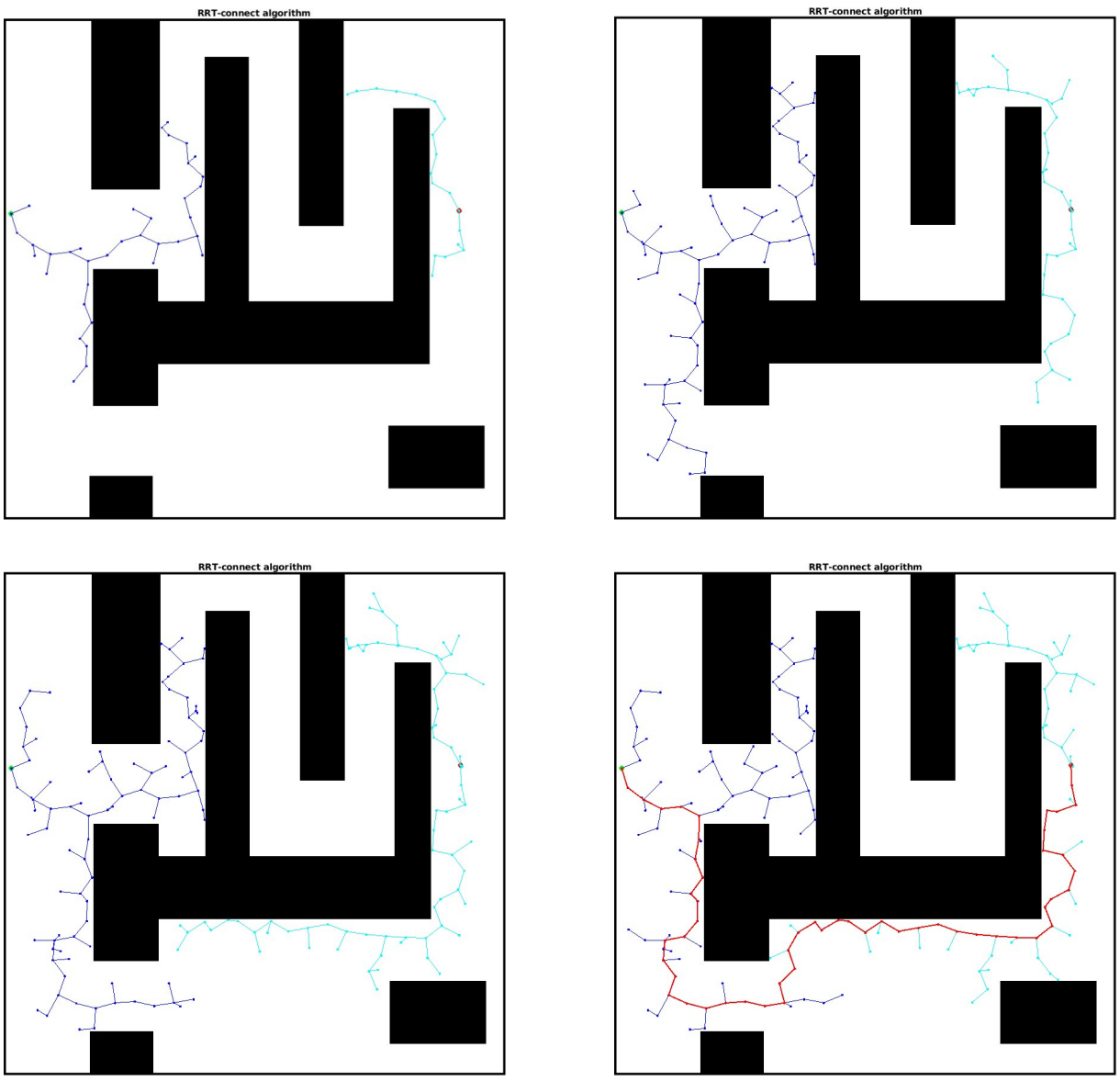
Testing RRT-connect was tested on the same maps of section 1.1. Results are visible in *figures 5,6,7,8*: the parameters considered for performances evaluation are final path length, computation time, number of nodes in the tree and number of iterations required. Further considerations and comments on these data will be discussed in section 1.4.

Algorithm 3 RRT-connect

Input: N, C, C_{free}, s, g, $bias$, δx, $greedy$
Output: τ
τ_s.init(s)
τ_g.init(g)
for $n \leftarrow 1$ to N **do**
  if n is odd **then**
    $i \leftarrow s, j \leftarrow g$
  else
    $i \leftarrow g, j \leftarrow s$
  end if
  $x_{rand} \leftarrow$ random point in C ($bias$ probability of being j)
  $x_{near} \leftarrow$ NEAREST_NEIGHBOR(x_{rand}, τ)
  $x_{new} \leftarrow$ NEW_STATE_GREEDY($x_{near}, x_{rand}, \delta x, greedy$)
  if edge(x_{new}, x_{near}) $\in C_{free}$ **then**
    τ_i.add_vertex(x_{new})
    τ_i.add_edge(x_{near}, x_{new})
    $dist_{max} \leftarrow step$ **if** $greedy = 0$ **else** $greedy \cdot step$
    if DIST(x_{new}, τ_j) $< dist_{max}$ **and** edge(x_{new}, τ_j) $\in C_{free}$ **then**
      $\tau \leftarrow$ CONNECT_TREES(τ_i, τ_j)
      break
    end if
  end if
end for

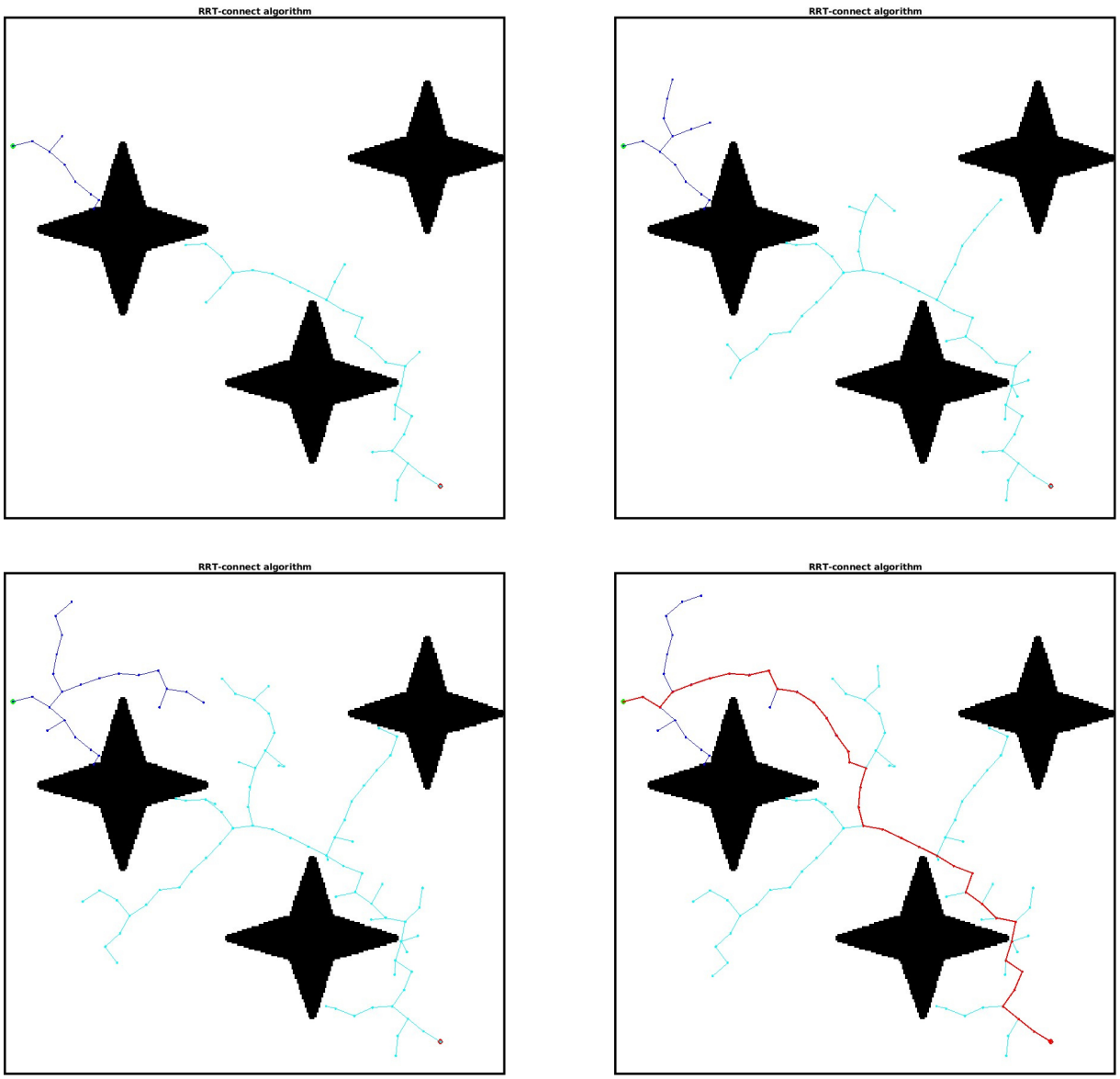
Algorithm 4 NEW_STATE_GREEDY

Input: x_{near}, x_{rand}, δx, $greedy$
Output: x_{new}
if $greedy > 0$ **then**
  if DIST$(x_{rand}, x_{near}) < greedy \cdot \delta x$ **then**
    $x_{new} \leftarrow x_{rand}$
  else
    $x_{new} \leftarrow$ farthest point in DIRECTION(x_{rand}, x_{near})
    such that:
      $DIST(x_{new}, x_{near}) < greedy \cdot \delta x$
      edge$(x_{new}, x_{near}) \in C_{free}$
  end if
else
  $x_{new} \leftarrow$ NEW_STATE$(x_{near}, x_{rand}, \delta x)$
end if



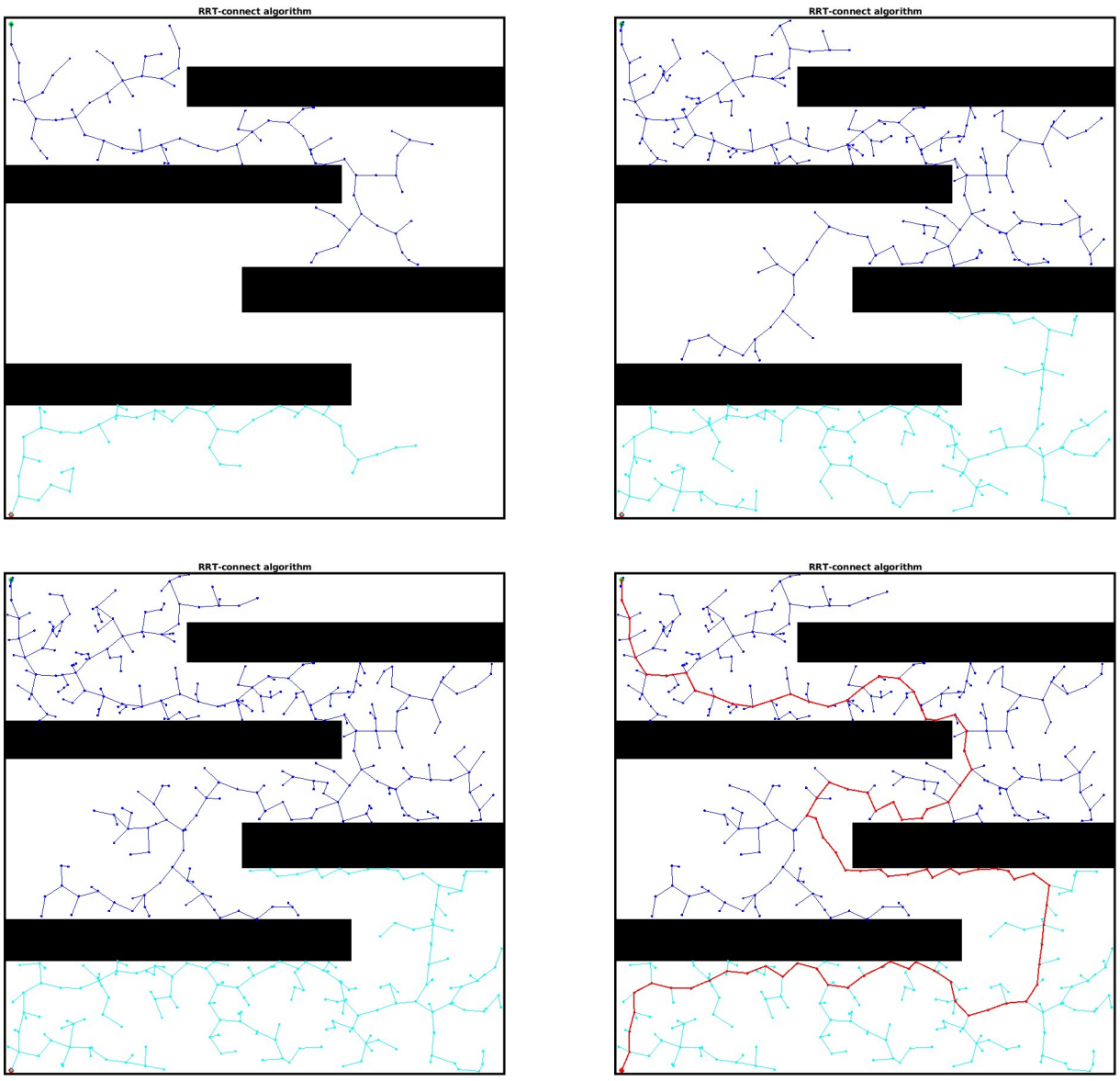
Length[m]	609.1
Time[s]	0.11
Iterations	594
Nodes	175

Figure 5: RRT-connect algorithm on map 1 (with greedy behavior)



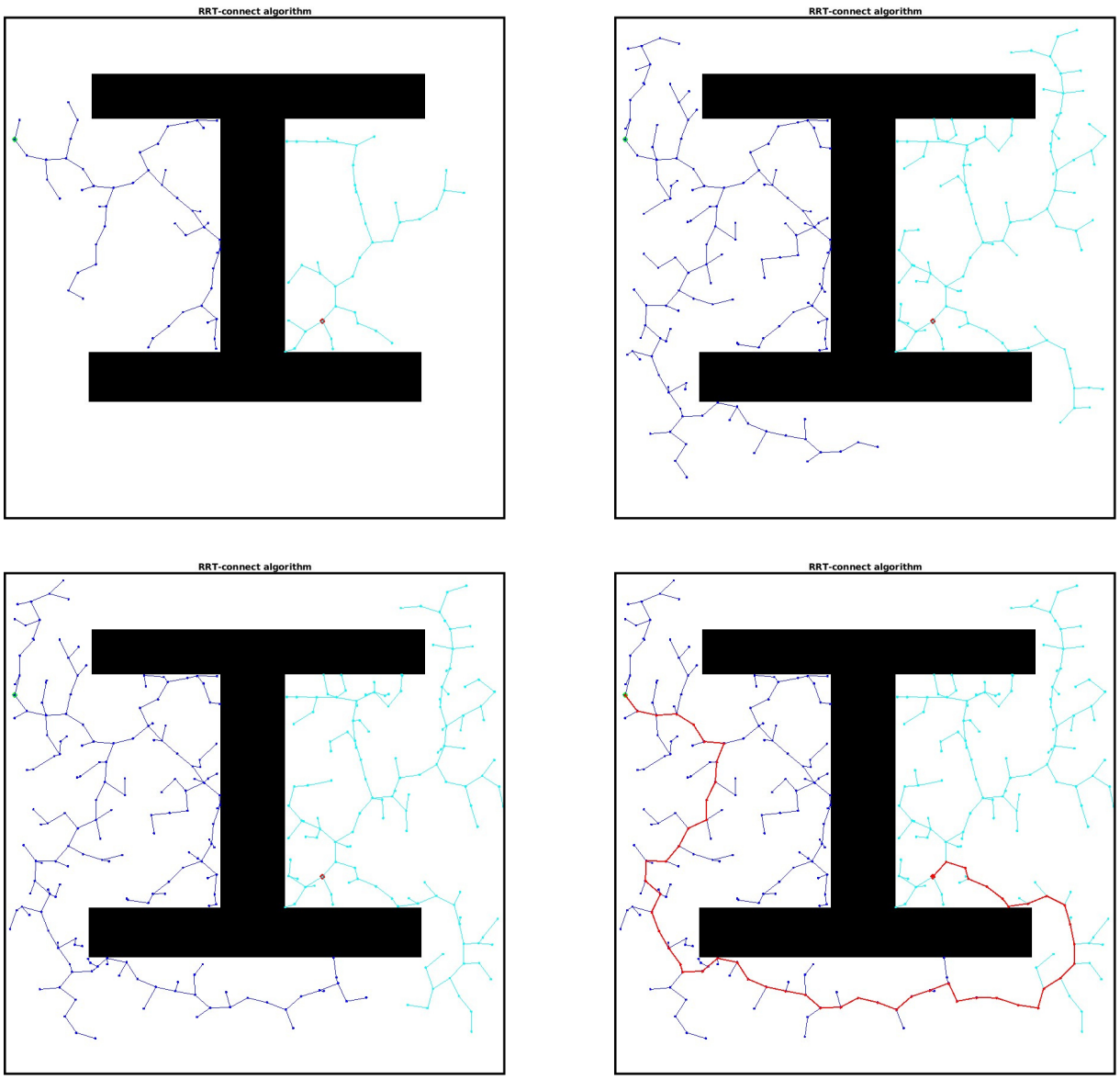
Length[m]	436.5
Time[s]	0.07
Iterations	199
Nodes	105

Figure 6: RRT-connect algorithm on map 2 (with greedy behavior)



Length[m]	1024.6
Time[s]	0.35
Iterations	1658
Nodes	512

Figure 7: RRT-connect algorithm on map 3 (with greedy behavior)



Length[m]	654.8
Time[s]	0.22
Iterations	931
Nodes	305

Figure 8: RRT-connect algorithm on map 4 (with greedy behavior)

1.3 Informed RRT*

As we already said, one of the problems of RRT is that the solution is often non-optimal and also very jagged and sharp. RRT* is a variation of the algorithm that improves this aspect by trying to "re-organize" the tree each time a new node x_{new} is added (even after the solution has been found). The outcome is a smoother path, that is also proven to be asymptotically optimal [3], at cost of an increased computational load and time.
To reduce, at least in part, the computational intensity of this procedure, an "informed" behavior

can be enabled [4]. When the goal has been reached, in fact, it is useless to rewire the tree in areas that are far from the solution: informed RRT* defines an ellipse that limits the sampling zone in the map. The resulting algorithm combines efficient local optimization through rewiring and targeted global refinement via informed sampling.

Implementation The algorithm was implemented in MATLAB basing on [3] and [4]. Pseudocode is shown in Algorithm 5, and the main difference with RRT consists in the *rewiring* and *informed sampling* procedures, presented in Algorithms 6 and 7.

Algorithm 5 Informed RRT*

Input: N, C, C_{free}, s, g, $bias$, δx, $patience$
Output: τ
τ.init(s)
for $n \leftarrow 1$ to N **do**
  if goal *not* reached **then**
    $x_{rand} \leftarrow$ random point in C ($bias$ probability of being j)
  else
    $x_{rand} \leftarrow$ INFORMED_SAMPLE(s, g, c_{best})
  end if
  $x_{near} \leftarrow$ NEAREST_NEIGHBOR(x_{rand}, τ)
  $x_{new} \leftarrow$ NEW_STATE($x_{near}, x_{rand}, \delta x$)
  if edge(x_{new}, x_{near}) $\in C_{free}$ **then**
    τ.add_vertex(x_{new})
    τ.add_edge(x_{near}, x_{new})
    update costs
    τ, costs $\leftarrow$ REWIRE($\tau, n, costs, x_{new}$)
    if DIST(x_{new}, g) $< \delta x$ **and** edge(x_{new}, g) $\in C_{free}$ **and** goal *not* reached **then**
      τ.add_vertex(g)
      τ.add_edge(x_{new}, g)
      goal reached, compute c_{best}
    end if
  end if
  if goal reached **then**
    compute c_{goal}, maybe lower due to *rewiring*
    $c_{best} \leftarrow c_{goal}$ **if** $c_{goal} < c_{best}$
    break if have not updated for more than $patience$ iterations
  end if
end for

rewiring focuses on optimizing the tree locally. After adding a new node, neighboring nodes within a dynamic radius are evaluated. If a shorter path to any neighbor is possible through the new node, their parent and cost are updated. This step ensures that the tree progressively minimizes path costs while maintaining connectivity. In Algorithm 6:

- **Rewire** x_{new}: optimizes the parent of the newly added node (x_{new}) by selecting the neighbor within a dynamic radius that minimizes the cost-to-come.

- **Rewire neighbors:** Optimizes neighboring nodes by checking if rerouting them through x_{new} reduces their cost-to-come, updating parent and cost accordingly.

Algorithm 6 REWIRE

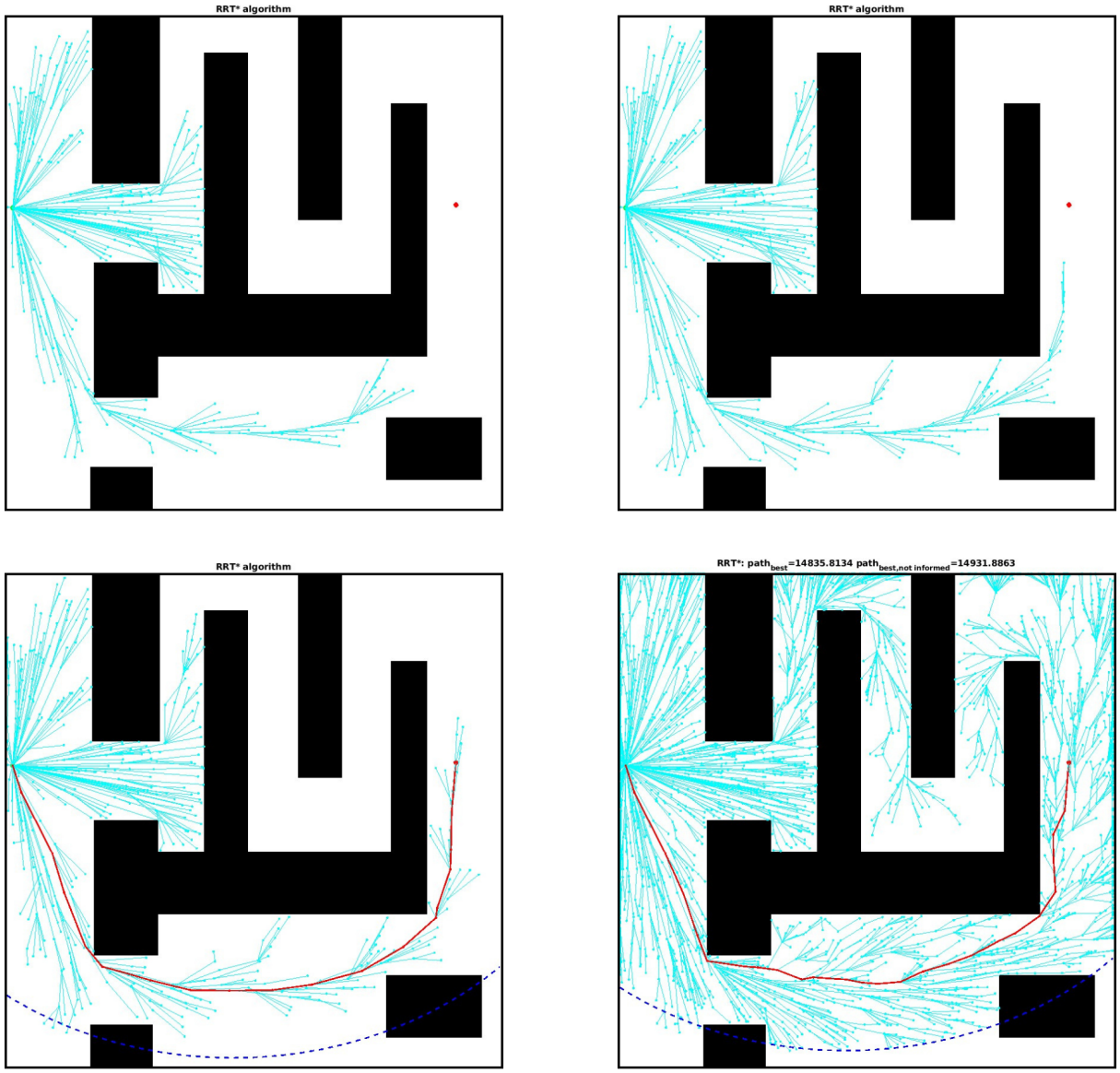
Input: τ, n, $costs$, x_{new}
Output: τ, $costs$
$d \leftarrow 2$, $\zeta_d \leftarrow \pi$, $\mu_{free} \leftarrow$ free space measure
$\gamma \leftarrow 2 \cdot \sqrt{1 + 1/d} \cdot \sqrt{\mu_{free}/\zeta_d}$
$r \leftarrow \max(\gamma(\log(n)/n)^{1/d}, step)$
$\mathcal{N} \leftarrow \{x \in \tau : \|x - x_{new}\| \leq r\}$
(i) Rewire x_{new}:
$c_{min} \leftarrow \min_{x \in \mathcal{N}}(cost(x) + \|x - x_{new}\|)$
Update parent and cost of x_{new} to minimize c_{min}
(ii) Rewire neighbors:
for $x \in \mathcal{N}$ **do**
  if edge$(x_{new}, x) \in C_{free}$ **and** $cost(x_{new}) + \|x_{new} - x\| < cost(x)$ **then**
    Update parent and cost of x
  end if
end for

Once an initial path to the goal is found, the algorithm switches to *informed sampling*. Instead of sampling uniformly in the space, new points are drawn within an ellipse defined by the start and goal points as foci and the current best-path cost as the ellipse's major axis. This strategy restricts exploration to the most promising region, significantly accelerating convergence toward the optimal path. In Algorithm 7, the INFORMED_SAMPLE function generates a random sample x_{rand} constrained within an ellipse. Sampling is first performed in the unit circle, as it allows uniform and efficient random sampling in a normalized space. The sampled point is then mapped to the ellipse, defined by the start point (s) and the goal point (g) as foci, with the sum of distances from these foci to any point on the ellipse equal to the current best path cost (c_{best}).

Algorithm 7 INFORMED_SAMPLE

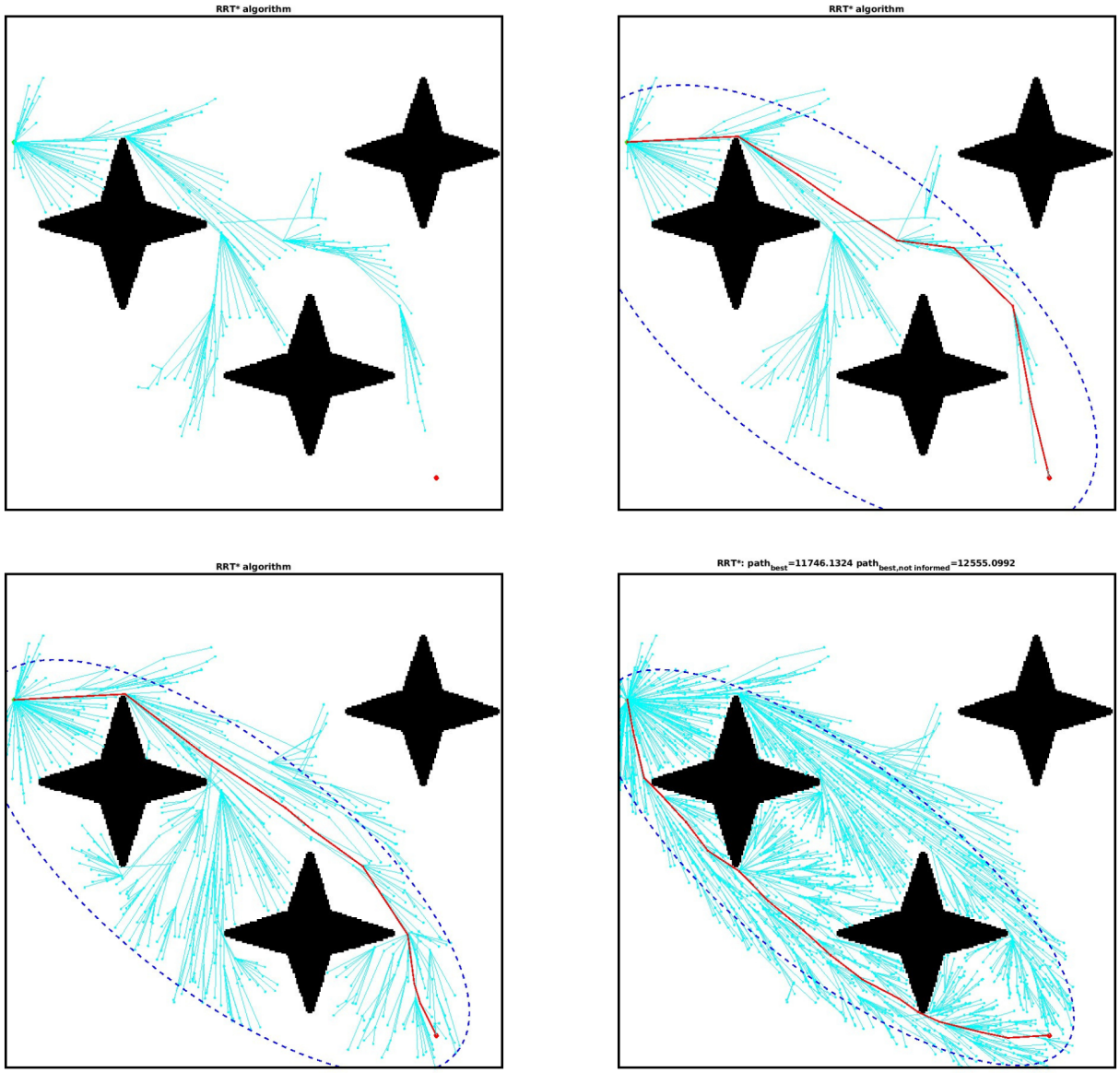
Input: s, g, c_{best}
Output: x_{rand}
sample P (in polar coordinates) in the unit circle
$x_{rand} \leftarrow$ map_to_ellipse(P), with ellipse:
  Foci: s, g
  Sum of distances: c_{best}

Testing Informed RRT* was tested on the same maps of section 1.1. Results are visible in *figures 9,10,11,12*: the parameters considered for performances evaluation are final path length, improvement with respect to the initial solution, computation time, number of nodes in the tree and number of iterations required. Further considerations and comments on these data will be discussed in section 1.4.



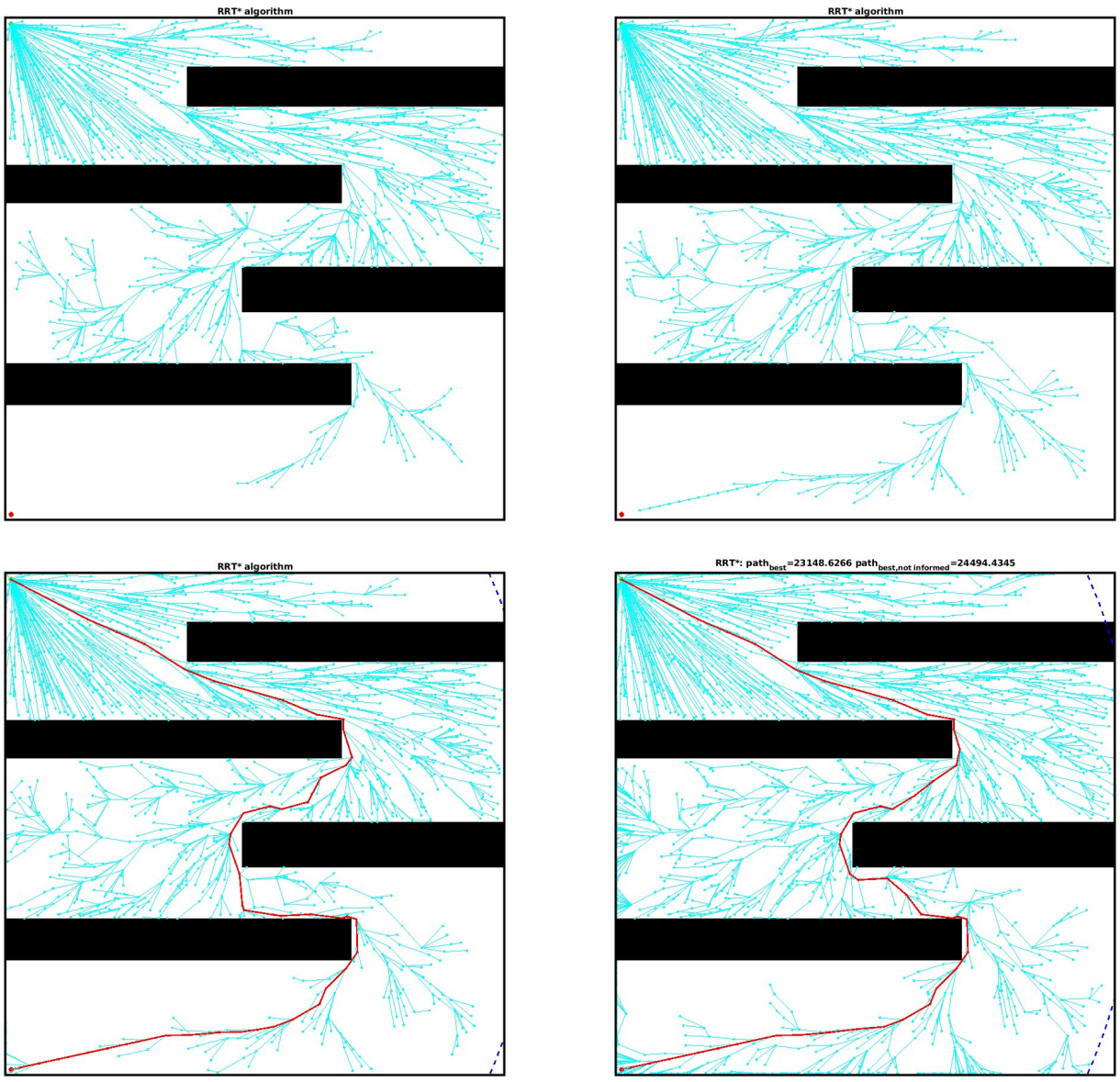
Length[m]	445.1
Time[s]	2.70
Iterations	4102
$Length_{notinformed}$[m]	448.0
Nodes	2532

Figure 9: Informed RRT algorithm on map 1*



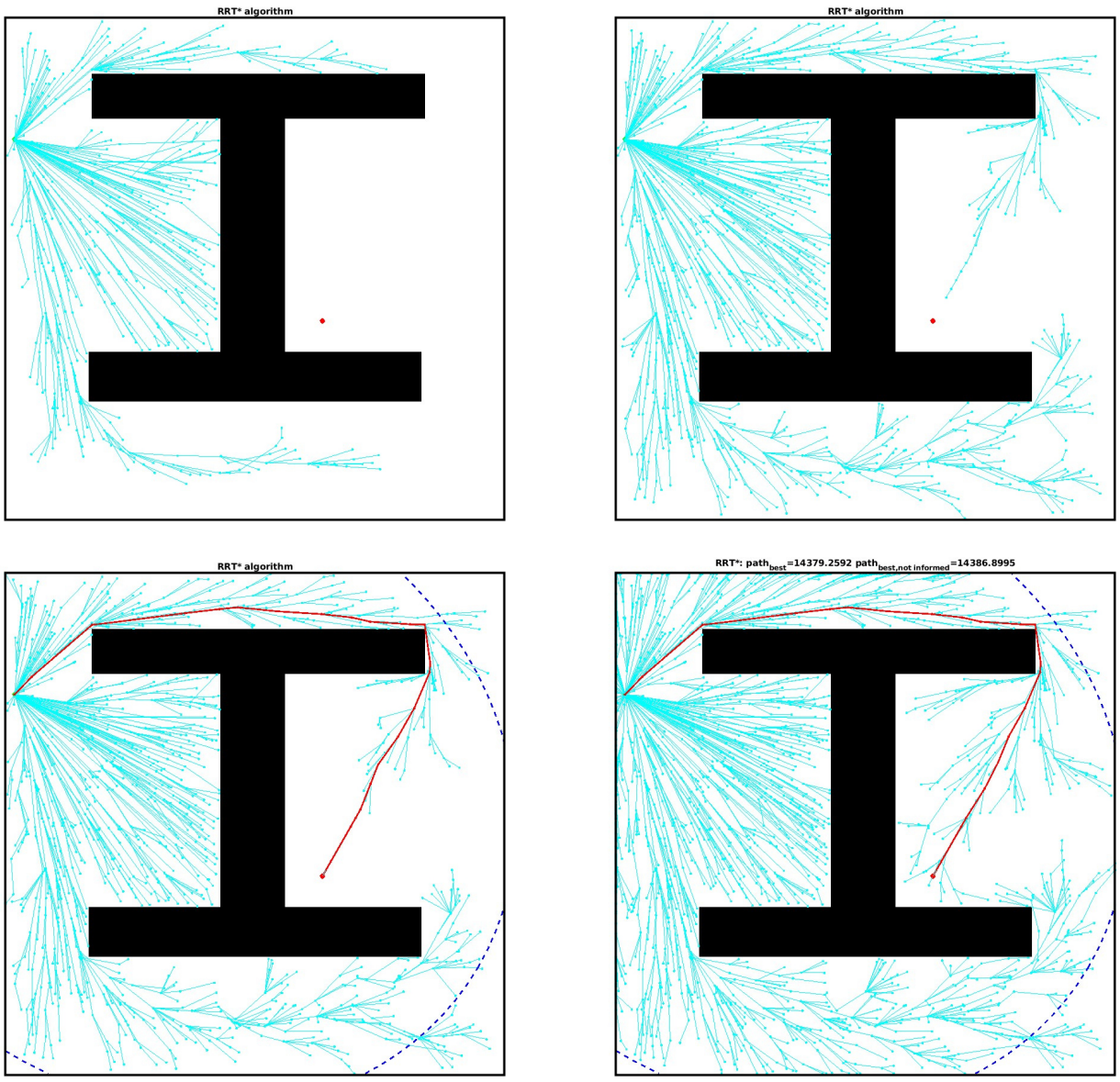
Length[m]	352.4
Time[s]	3.35
Iterations	2278
$Length_{notinformed}$[m]	376.7
Nodes	1851

Figure 10: Informed RRT algorithm on map 2*



Length[m]	694.5
Time[s]	2.17
Iterations	4381
$Length_{notinformed}$[m]	734.8
Nodes	2215

Figure 11: Informed RRT algorithm on map 3*



Length[m]	431.4
Time[s]	1.76
Iterations	2927
$Length_{notinformed}$[m]	431.6
Nodes	1804

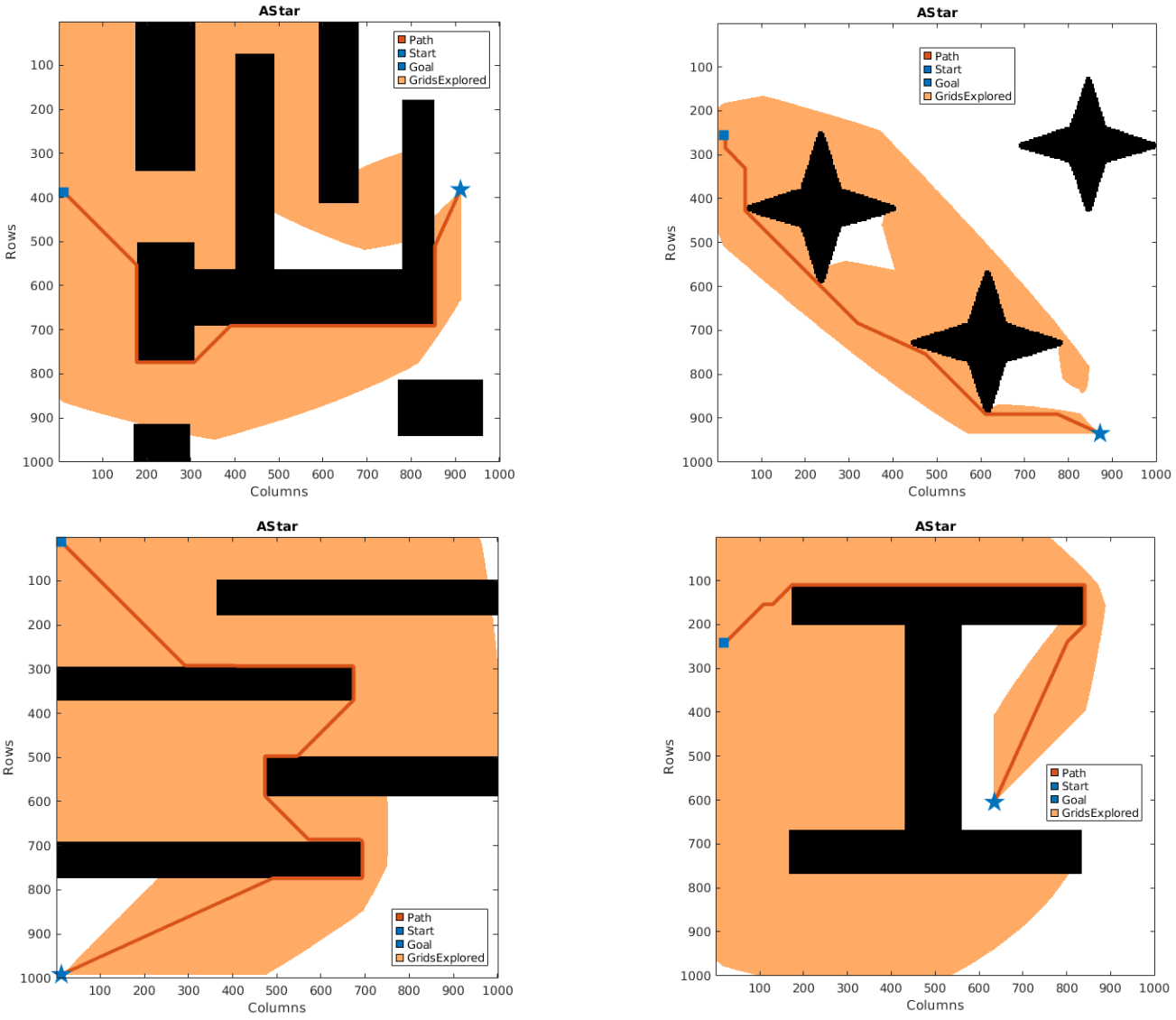
Figure 12: Informed RRT algorithm on map 4*

1.4 Results and comparison

As anticipated, RRT, RRT-connected and informed RRT* were compared, also considering the results obtained with A* (*figure 13*), which is a well-known grid-based planner algorithm. In particular, tests were performed by rescaling the provided maps to a resolution of 10000x10000 pixels for the sample-based approaches, while for implementing A* algorithm the built-in MATLAB planner included in the Navigation Toolbox was exploited, with maps rescaled to a resolution of

1000x1000 pixels. Finally, to compute path lengths a "real world" map size of 300x300 meters was assumed, and these parameters were set:

- *step* (default edge length for all RRT-based algorithms): 200 px;
- *bias* (heuristic on moving towards the goal): 20%;
- *greedy* (greedy behavior for RRT-connect): 2.5 times *step*;
- *patience* (tolerance on the refinement of the solution for informed RRT*): 500 iterations;



	Map 1	Map 2	Map 3	Map 4
Length [m]	449.27	363.39	694.99	437.23
Time [s]	0.29	0.24	0.35	0.33
Explored nodes	440605 (63%)	263357 (29%)	627238 (79%)	539740 (66%)

Figure 13: A algorithm*

Notice that due to the random nature of RRT algorithms, the results are different each time the codes are run: to give a comprehensive analysis, many tests should be done, averaging the results and performing a statistical study on them.

Path length and optimality The lengths of the resulting paths are listed in Table 1. Significant differences are present among the various algorithms. RRT produces relatively long paths, that also have a jagged shape (see *figure 3*, for example), indicating that the random exploratory approach is not suitable to find optimal solutions. RRT-connect generates even longer paths, for the same reason. Although the dual-tree approach reduces computation time, the connection strategy can lead to less refined paths.

Algorithm	**Map 1**	**Map 2**	**Map 3**	**Map 4**
RRT	501.5	434.4	906.2	575.6
RRT-Connect	609.1	436.5	1024.6	654.8
Informed RRT*	445.1	352.4	694.5	431.4
A*	449.27	363.39	694.99	437.23

Table 1: Path lengths for RRT, RRT-connect, informed RRT and A*. Results are in meters.*

Consistent with the theory, informed RRT* achieves the best result due to the rewiring strategy that continuously smooths and optimizes the solution, while A* shows slightly worse performance. In fact, despite its guarantee of optimality, the grid-based algorithm is affected by the discretization of the map and thus is optimal only *within* the given grid. Notice that when comparing RRT* and A* it's important to take into account for the chosen grid resolution (for A*) and *patience* parameter (for RRT*): different combinations could lead to make one algorithm better than the other. For example, as we can see from *figure 11* and Table 1, for very "dense" maps (in this case Map 3) informed RRT* tends to give worse paths (in fact A* gives almost the same length), since it would require a bigger *patience* to further shorten the solution.

Computing time The computation times (Table 2) show clear differences between the evaluated algorithms. RRT and RRT-connect exhibit the shortest average times among all the maps and, in general, RRT-connect is slightly better than RRT due to its dual-tree approach and "greedy" heuristic, which accelerates convergence despite producing less refined paths. It was also observed that increasing this heuristic behavior helps reducing time even more, as expected.
Informed RRT* demonstrates significantly higher computation times. This is primarily due to the computational overhead introduced by additional sampling and rewiring, which prioritizes path optimality over speed. Of course, decreasing *patience* parameter helps reducing time at cost of a less optimal solution, while disabling the informed sampling heuristic increases computational time bringing no advantages. In particular, ellipsoidal informed sampling is observed to remarkably improve the performances in "clean" maps such as Map 2, where the ellipse is able to force a very restricted sampling region (*figure 10*). Counterintuitively, computational time is higher for these maps (Table 2): this is related to the fact that the path is more likely to be updated during the iterations, so the *patience* stop criterion will be triggered later.
A* achieves intermediate results between RRT* and RRT/RRT-connect. It's important to notice that this algorithm performs a "brute force" deterministic analysis on the available grid, so the computational time is highly dependent on the number of nodes. For this reason we can say that A* is very sensitive to the specific problem we have, leading soon to unacceptable computational

times if the dimensionality or the resolution are too high.
In general, RRT and RRT-connect are the fastest approaches, A* shows good time-performances along with optimality properties but it's limited by the type of problem we are facing, and finally informed RRT* appears to be a good trade-off that allows to find a good solution in a reasonable amount of time, not being limited too much by the specific case of interest due to its random sampling nature. Moreover, differently from A*, RRT* presents a larger flexibility, allowing to tune the *patience* level and rewiring parameters in order to obtain a fastest, less optimal solution or a slower, more optimal one.

Algorithm	**Map 1**	**Map 2**	**Map 3**	**Map 4**
RRT	0.12	0.11	0.39	0.20
RRT-Connect	0.11	0.07	0.35	0.22
Informed RRT*	2.70	3.35	2.17	1.76
A*	0.29	0.24	0.35	0.33

Table 2: Computation time for RRT, RRT-connect, informed RRT and A*. Results are in seconds.*

Some important observation need to be done on the computational times. First of all, the obtained values depend on the machine used for running the algorithms (ex.: other active processes, type of processor, etc...), and also on how the codes were implemented: for example, A* was run by exploiting the built-in MATLAB function, so it is most likely better optimized with respect to our codes for RRTs, which makes the comparison hard. Moreover, random nature of RRTs algorithms makes so that output time changes every time (random seed are set to change each time the codes run). However, despite these limitations, all the observations in this paragraph are still generally valid.

Iterations and nodes in the tree (for RRT algorithms) RRT* is the algorithm that leads to the highest number of iterations (Table 3), as expected, due to the fact that sampling does not stop immediately as soon as a solution is found. RRT-connect, instead, reduces significantly the number of cycles, thanks to its bidirectional tree expansion which enables faster connection between start and goal regions. Obviously, is easy to understand that the less obstacles we have on the map, the higher the probability of generating a new node at each iteration is: that's why when planning on Map 2, for example, the final number of nodes is closer to the number of iterations compared to more "dense" maps. Also the position of start and goal influences the number of iterations and nodes.

Algorithm	**Map 1**	**Map 2**	**Map 3**	**Map 4**
RRT	891 (324)	324 (207)	3021 (1288)	1408 (630)
RRT-Connect	594 (175)	199 (105)	1658 (512)	931 (305)
Informed RRT*	4102 (2532)	2278 (1851)	4381 (2215)	2927 (1804)

Table 3: Iterations and nodes in the final tree (inside parenthesis) for RRT, RRT-Connect, and Informed RRT.*

2 Project II

The aim of this second project is to map a specific environment in Gazebo (`turtlebot3_house`) using the `turtlebot3 waffle pi`, then compute the optimal trajectory between a starting position and a final position, and finally develop a feedback control logic to make the robot follow that path. Simulink and ROS were used along with Gazebo to perform the task.

2.1 Mapping the environment

When an autonomous vehicle is navigating an unfamiliar environment, it requires a tool that can determine its own position while simultaneously mapping its surroundings. This process is known as SLAM, which stands for Simultaneous Localization And Mapping. The idea behind it is to set an initial known position for the robot, then, using sensors like LIDAR, lasers, and cameras, the environment is mapped and its location is tracked as it moves. However, both the robot's position and the map features are affected by noise if they estimated only once during the robot's movement. To reduce this problem, further estimation of the same points is done during the motion of the robot, so the mean value of the noise is significantly reduced.
In our specific case, the environment was mapped exploiting the gmapping SLAM algorithm of ROS, combined with the ROS Navigation stack to save the generated data. To set this up, the following commands are used:

```
$ export TURTLEBOT3_MODEL=waffle_pi                          ▶ load robot model
$ roslaunch tutlebot3_gazebo turtlebot3_house.launch         ▶ load environment
$ roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch  ▶ Rviz for monitoring
```

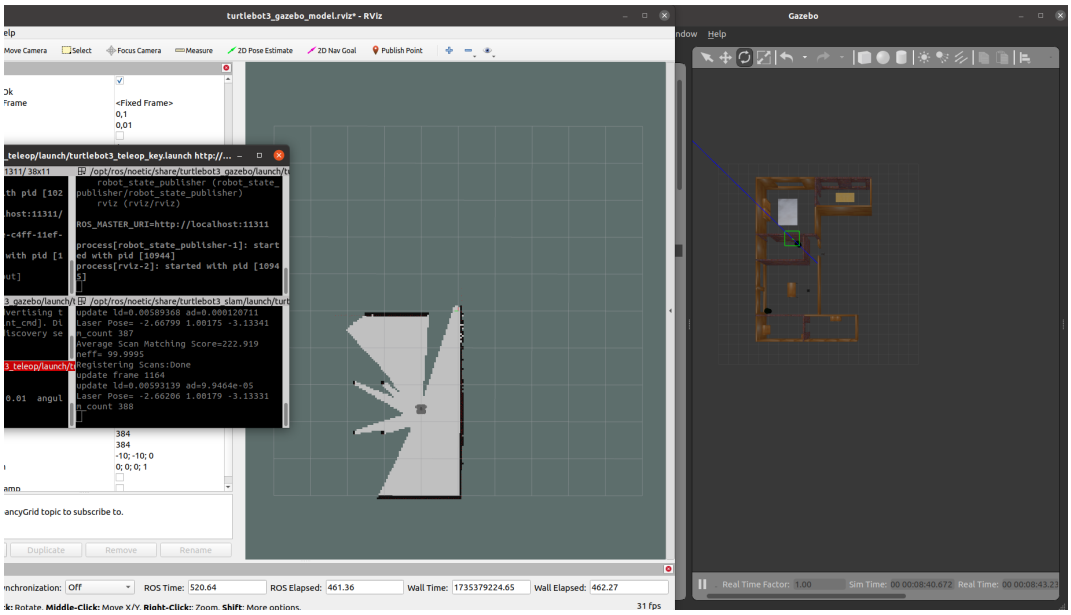


Figure 14: Rviz interface

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch   ▶ to move the robot
$ roslaunch turtlebot3_slam turtlebot3_gmapping.launch       ▶ launch gmapping node
```

Finally, the map is saved using the following command:

```
$ rosrun map_server map_saver -f house_map
```

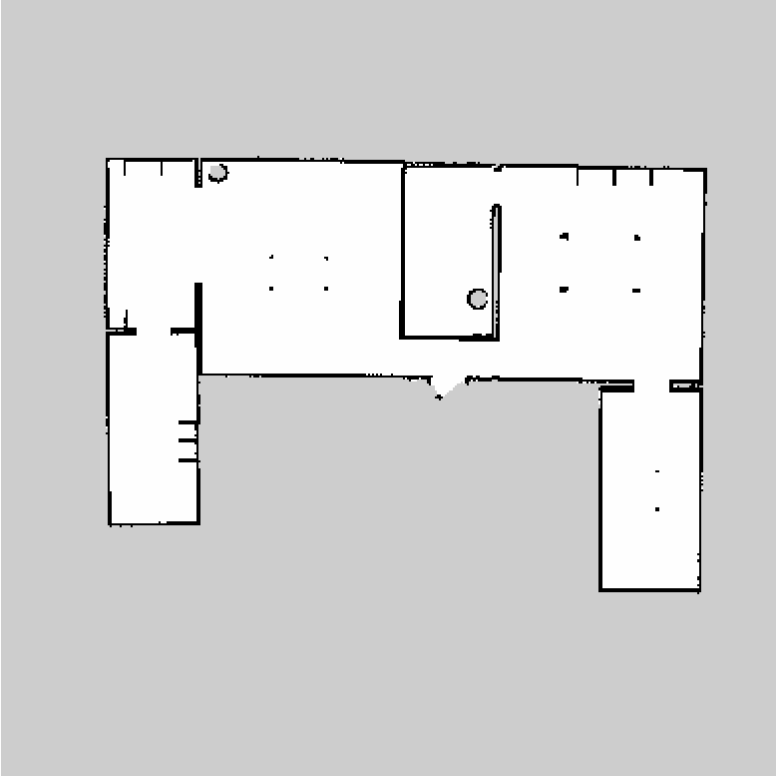


Figure 15: Final reconstructed map

2.2 Path planning

Two different path-searching algorithms were employed in this project to define the trajectory to follow: A* and RRT*.
The A* algorithm is part of the grid-based approaches used to find the optimal path between a starting position and a goal. Grid-based methods require discretizing the continuous world into a grid. This grid consists of free and forbidden cells, and the vehicle is assumed to move only between adjacent free cells. Mathematically, this is equivalent to pathfinding in a discrete graph $G = (V, E)$, where V represents the set of free cells (vertices) and E represents the set of connections between adjacent grid cells (edges). In particular, A* is proven to find the optimal (shortest) solution within G, but it is highly sensitive to the grid resolution and, in general, computationally intensive.
RRT* was instead implemented in its "informed" version, as presented in Project I (section 1). Refer to section 1 for the properties of A* and RRT* and a comparison between the two algorithms.

2.2.1 Planning with A*

Grid creation To run A* algorithm we first need to process the available map in order to generate a proper grid G. The 384x384 gray scale map obtained with gmapping was rescaled to a resolution of 768x768 pixels, that was observed to offer a good trade off between accuracy and computational time. It was then converted to binary form, distinguishing between free vertices

and obstacles (uncertain nodes were treated as obstacles) and a clearance was added (taking into account the robot's radius and a reasonable safety margin) to prevent issues with the turtlebot's movement near objects. This process is shown in *figures 16,17*.

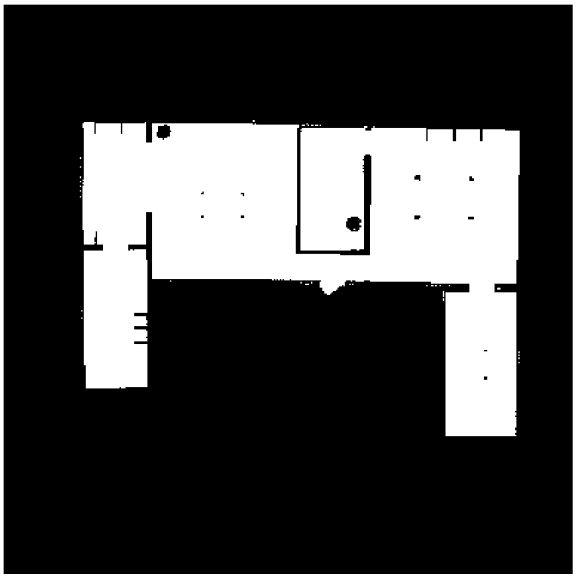


Figure 16: Rescaled and binarized map, before adding clearance

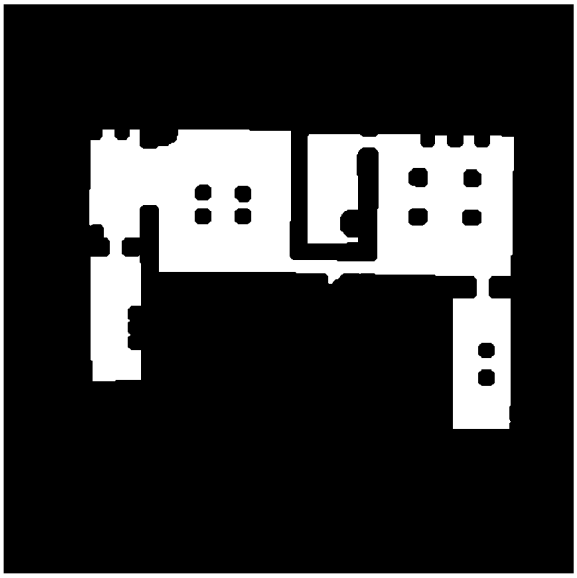


Figure 17: Rescaled and binarized map, after adding clearance

Once vertices V have been created, edges E must be defined as well. This task is a little bit more complicated, since memory issues could rise due to the high amount of information needed for describing the connections between nodes. The strategy that was developed was the following:

1. if possible, binary vertices matrix V is transferred to the GPU, which is much faster for big matrix operations;
2. V is divided into blocks, each one of them as big as possible (compatibly with memory dimension);
3. these blocks are taken two at the time, by two for loops (parallelization can be used to further increase the speed);
4. a distance matrix is computed between the two blocks: it contains the distances between all nodes in the considered blocks; `pdist2` function of Machine Learning Toolbox is exploited to efficiently do that;
5. all distances higher than $\sqrt{2}$ are set to zero, so that sparse matrices can be exploited;
6. all non-null elements of distance matrix are checked to store the connected nodes in the output matrix E.

This procedure returns the list of edges E that can then be fed, together with the vertices V, to Navigation Toolbox `navGraph` function. This function creates a graph object G which is then used

to run A* algorithm directly implemented in `plannerAStar` [3]. The use of GPU, parallelization, sparse matrices and built-in functions improves the performances by a big margin, allowing in particular to build arbitrarily big maps.

A* algorithm After having built the decomposition, the optimal path can be obtained through the A* algorithm. A cost function $f(q)$ is defined by adding two contributions: the "cost-to-arrival" $g(q)$, which represents the cost of reaching the current node from the start, and the "cost-to-go" $h(q)$, which is the estimated cost to reach the goal. The second component, $h(q)$, is what distinguishes A* from other algorithms and must be a heuristic function that underestimates the cost from the current node to the goal (Euclidean distance was used in our case). Notice that $f(q)$ is used just to determine the order in which nodes are extracted from the alive set (or "open list"), while the "real" cost of a vertex is just $g(q)$. With this idea in mind, A* progressively explores the grid up to the goal point, continuously updating the cost and the parenting of each vertex as soon as a shorter path is found to reach it, and moving the checked nodes to the "closed list". The pseudocode is presented in Algorithm 8.

Algorithm 8 A* search algorithm

```
Set the starting node as the current node.
The open list contains only the starting node with its value f(n).
The closed list is empty.
while the open list is not empty do
    Extract the node with the lowest f(n) value from the open list (this node becomes the
    current node).
    if the current node is the destination node then
        Return the path (reconstruct the path through the visited nodes).
        break
    end if
    for each neighboring node of the current node do
        Compute the values of g(n), h(n), and f(n).
        if the neighbor is not in the open list or the new f(n) value is better then
            Update the node (including the neighbor in the open list).
        end if
    end for
    Add the current node to the closed list.
end while
if the open list is empty and the destination node has not been found then
    No path exists.
end if
```

An example of output of the code for generic start and goal positions is reported in *figure 18*. Additionally, we have improved the algorithm to allow multiple goals to be added to the same path. The approach involves using the A* algorithm repeatedly, with each new starting point being the previous goal. One of the paths we obtained using this method is shown in *figure 19*

[3] It's worth to mention that there exists a built-in MATLAB planner `plannerAStarGrid` that directly works with the binarized map, creating the edge matrix on its own. This method is even more efficient but, since it was not clear to us how it worked, we decided to still use our procedure.

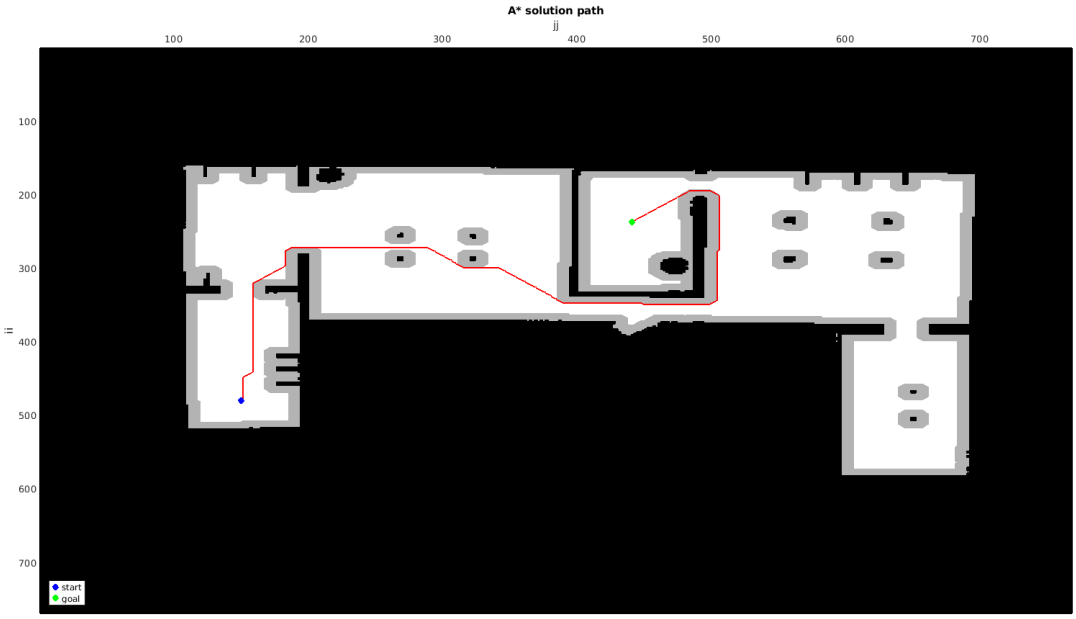


Figure 18: Example of the result of A algorithm on our map. The optimal path is enlighten in red, while gray areas represent the clearance added around obstacles, that are black*

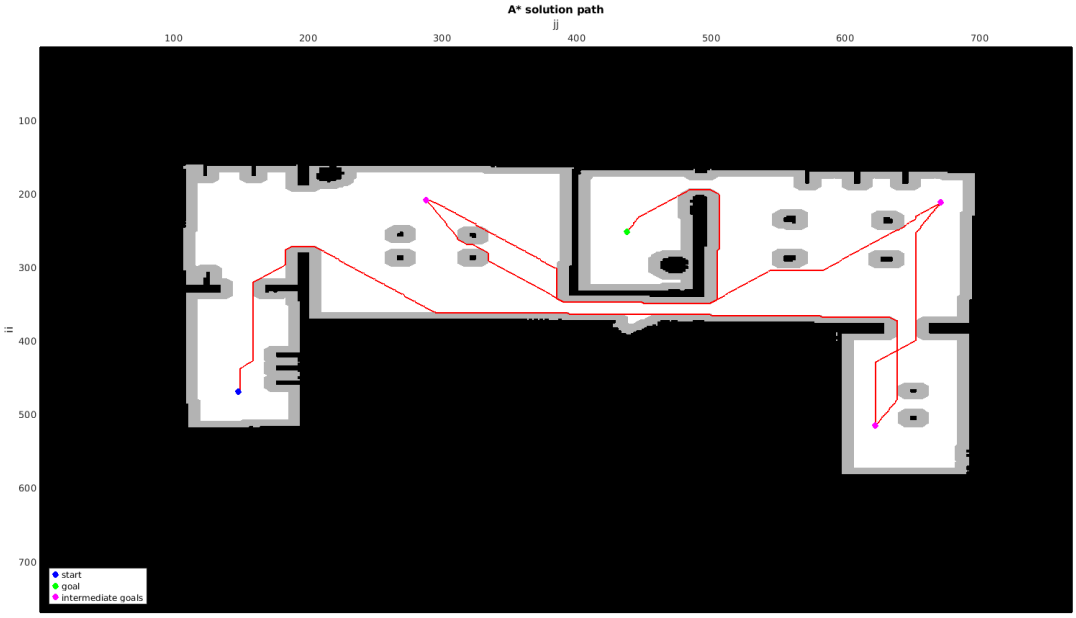


Figure 19: Planning with more intermediate goal positions

2.2.2 Planning with RRT*

As previously mentioned, RRT* was also employed to generate trajectories. Its informed version was used, as presented in Project I, section 1.3: for details and implementation refer to that. As an example, a generic trajectory obtained this way is reported in *figure 20*.

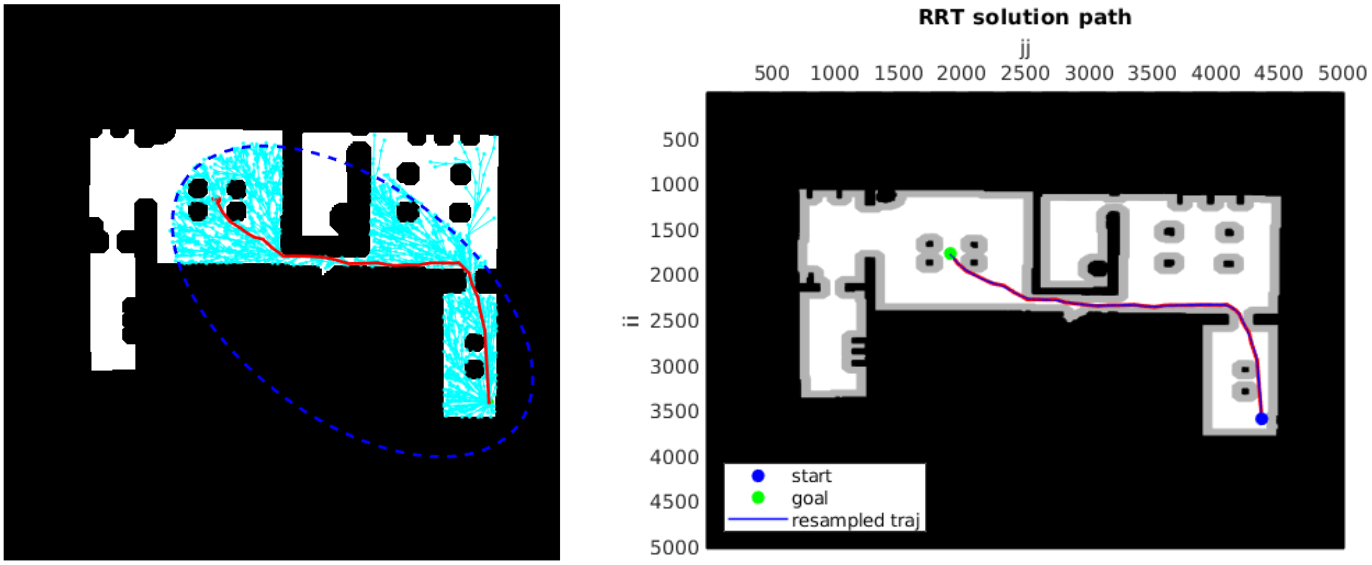


Figure 20: Example of trajectory generated with informed RRT algorithm. The informed elliptical region and the computed tree are visible on the left figure, while on the right the solution path (red) is reported, along with the smoothed trajectory (blue). Notice that for this project, the resulting RRT* trajectory has also be resampled in order to obtain a smoother curve: detail of this decision will be explained in section 2.3*

2.3 Feedback control

Once the desired path has been determined, the objective is to design a feedback controller that ensures the `turtlebot` follows the trajectory accurately. This has been implemented in Simulink using the block diagram in *figure 21*.

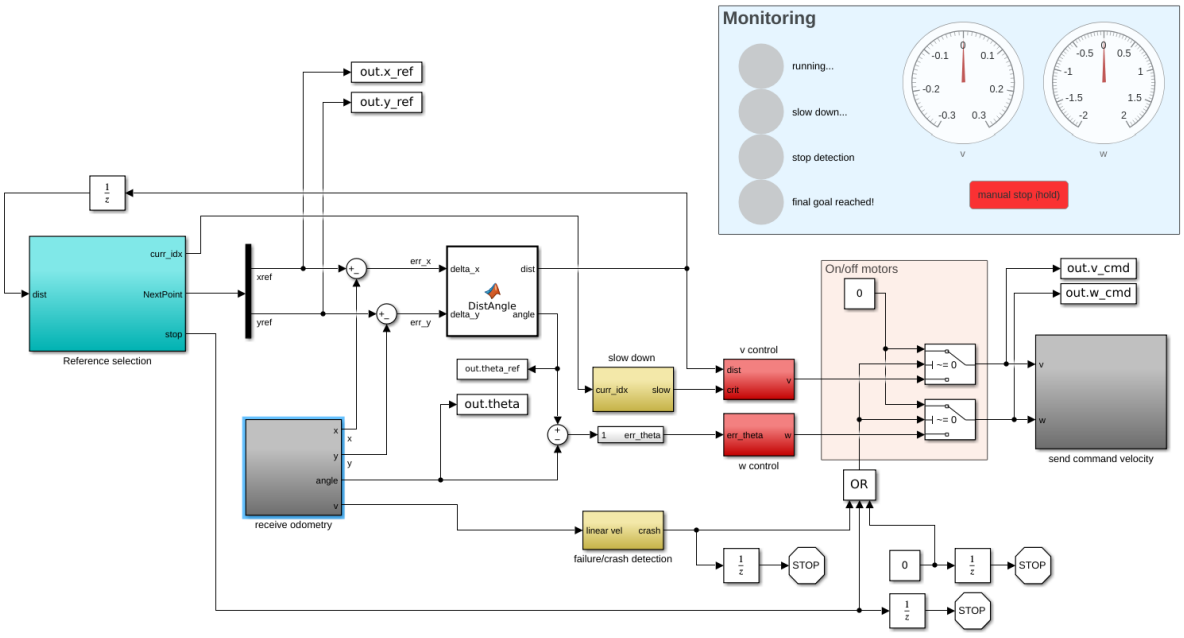


Figure 21: Simulink model for Project II

The planned solution is loaded as a sequence of points on the map in the *Reference selection*

block, that simply selects the next reference point as soon as the vehicle reaches the previous one with a certain tolerance *tol*. At this point, a very simple control logic is adopted, consisting in two separate components:

- A PD controller for robot's yaw rate ω. The feedback in this case is the angular error, which is calculated by subtracting the robot's orientation (obtained from odometry) from the angle between its current position and the next waypoint.
- A proportional controller for robot's linear velocity v, taking the distance from the next reference point as input (proportional gain and *tol* are actually such that v is always saturated at the max allowable speed, in normal travel conditions).

Moreover, three additional checks are added to regulate the linear speed: a failure detection mechanism, that stops the `turtlebot` and the simulation when it gets stuck on an obstacle, another similar logic for when the final goal is reached and finally a trigger that slows down the vehicle whenever it is approaching a sharp turn. This last control, in particular, pre-computes and saves the "critical" waypoints in the trajectory that involve a turn greater than 80 deg, which typically occurs when multiple intermediate goals are set, so that when those points are close, velocity is reduced until the turning maneuver is completed.

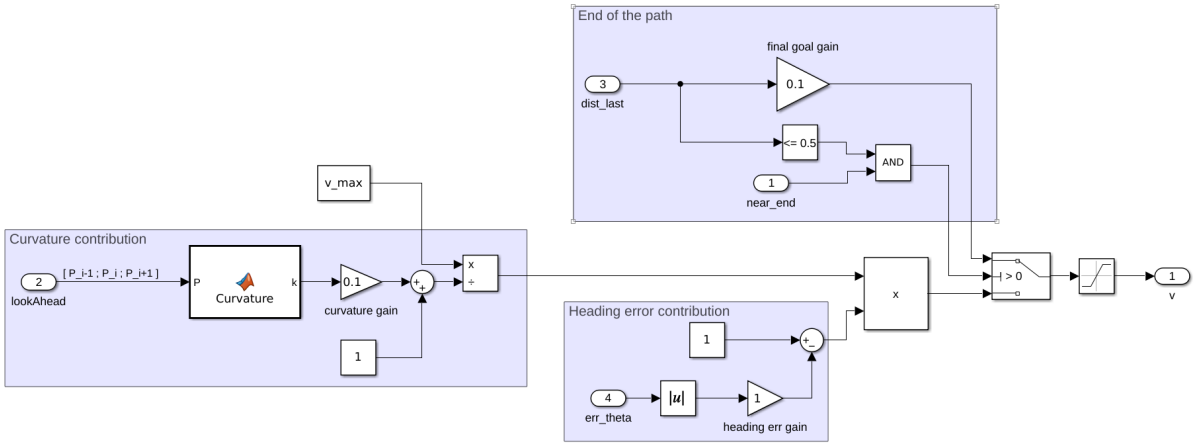


Figure 22: inside of the updated `v_control` *block for "fast" robot control Simulink model*

"Fast" robot control The previously described control logic is very simple and works if the robot is limited to a maximum linear speed of 0.26 m/s and a maximum yaw rate of ± 1.82 rad/s, that are chosen accordingly to `turtlebot3 waffle pi` specifics. However, to make things a bit more interesting, we tried to increase v_{max} to 0.6 m/s, meaning that sliding and inertial effects become more influent, requiring a more refined control on the linear velocity.
To achieve that, three main changes are introduced, as shown in *figure 22*:

- the trajectory computed with A* or RRT* is post-processed using cubic smoothing spline interpolation (`csaps` function in MATLAB): this results in a more "continuous" path, at cost of cutting same corners (clearance in the map is very important to overcome this);
- linear speed v is regulated by setting it to v_{max} and reducing it basing on:

– the curvature c of the piece of trajectory that the robot is going to face:

$$v_1 = \frac{v_{max}}{1 + K_{curv} \cdot c} \quad , \quad \textit{being } c \textit{ the curvature and } K_{curv} \textit{ a proper gain}$$

– the orientation error:

$$v = v_1 \cdot (1 - K_{or} \cdot |e_{or}|) \quad , \quad \textit{being } e_{or} \textit{ the error on the heading and } K_{or} \textit{ a proper gain}$$

- a *lookahead* distance was set, in order to select as reference waypoint not the immediate next, but a slightly more advanced one, due to the higher speed of the vehicle.

Emergency stop logic was kept as before, while the $end - simulation$ logic on the reaching of the final goal was slightly anticipated, as shown in *figure 22*, because of the higher speed.

2.4 Results

As requested, four different trajectory were generated and tested, using A* along with the "basic" control strategy (*figures 23,24,25,26*). Then two additional results are reported, obtained by simulating the "faster" robot, one with A* (*figure 27*) and the second one with RRT* (*figure 28*).

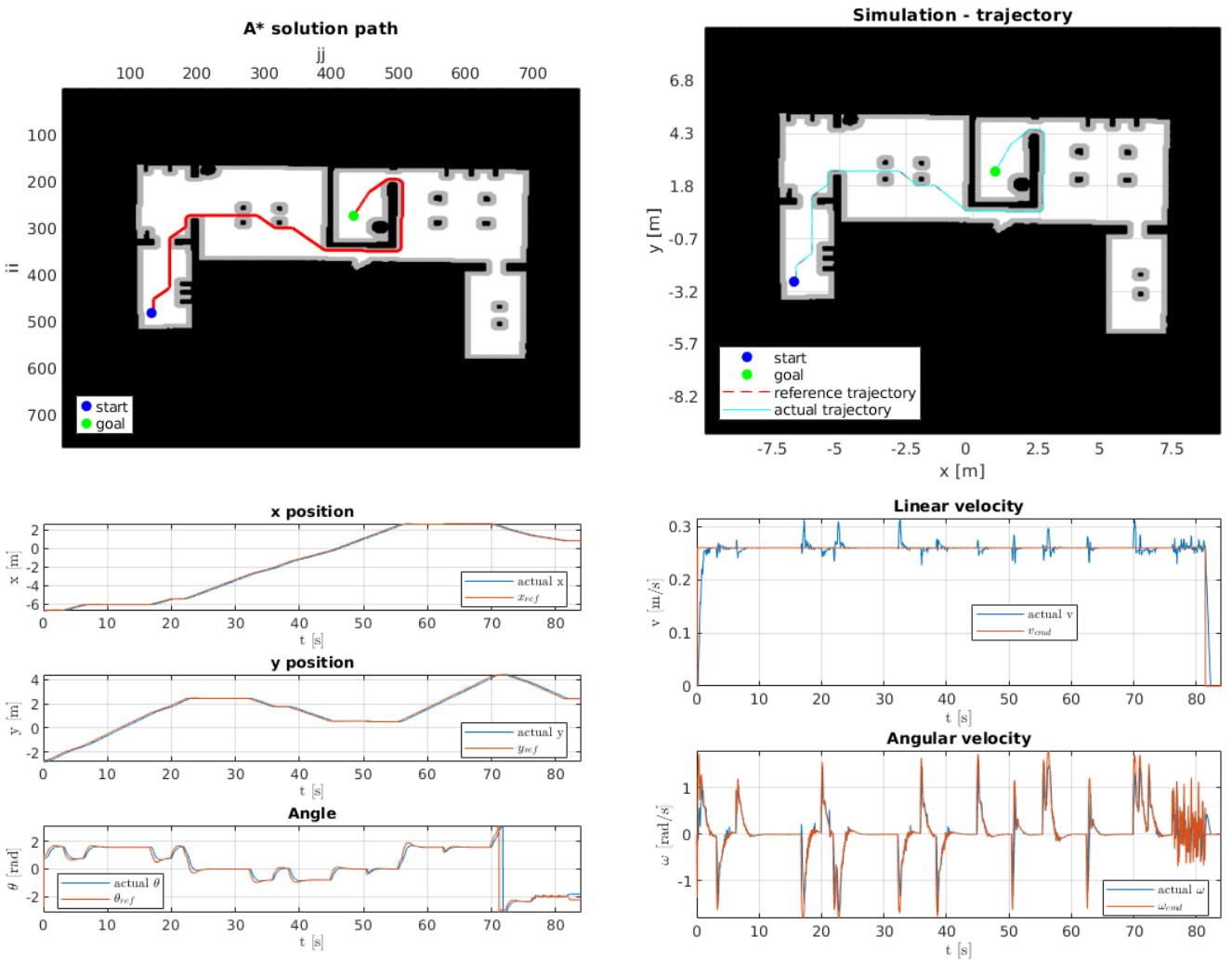


Figure 23: Test 1 (A planning)*

Figure 24: Test 2 (A planning)*

Figure 25: Test 3 (A planning)*

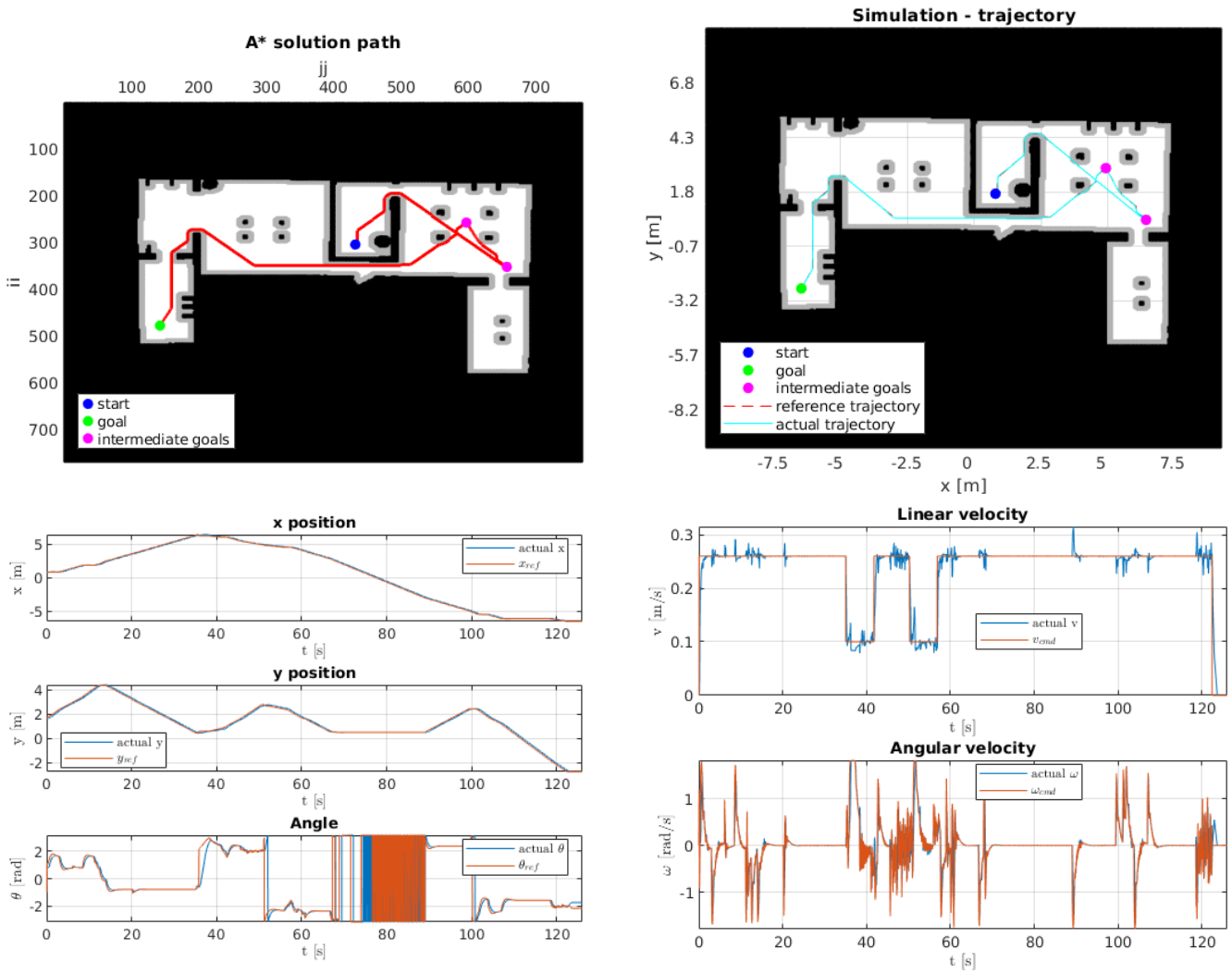


Figure 26: Test 4 (A planning with 2 intermediate goals)*

From the plots, it's clear that the `turtlebot` is able to follow the trajectory quite accurately, reaching the goal in all four tested scenarios. The linear velocity is always set to its maximum value, during the travel time, as previously anticipated, exception made in Test 4, when some intermediate goals were imposed: in those positions, in fact, the robot must perform a sharp turn, so it slows down (*figure 26*).
Notice also that sometimes the *angle* plot seems to show a "chaotic" behavior (*figure 26*): this is simply due to the fact that θ is represented between $-\pi/2$ and $\pi/2$, so when it assumes values near 180° the resulting plot is confused. Finally, we can observe that the ω_{cmd} plots are often quite scattered (*figure 25*), resulting in a non-optimal control effort: this could probably be improved by using a more performing controller, rather than a simple PD.

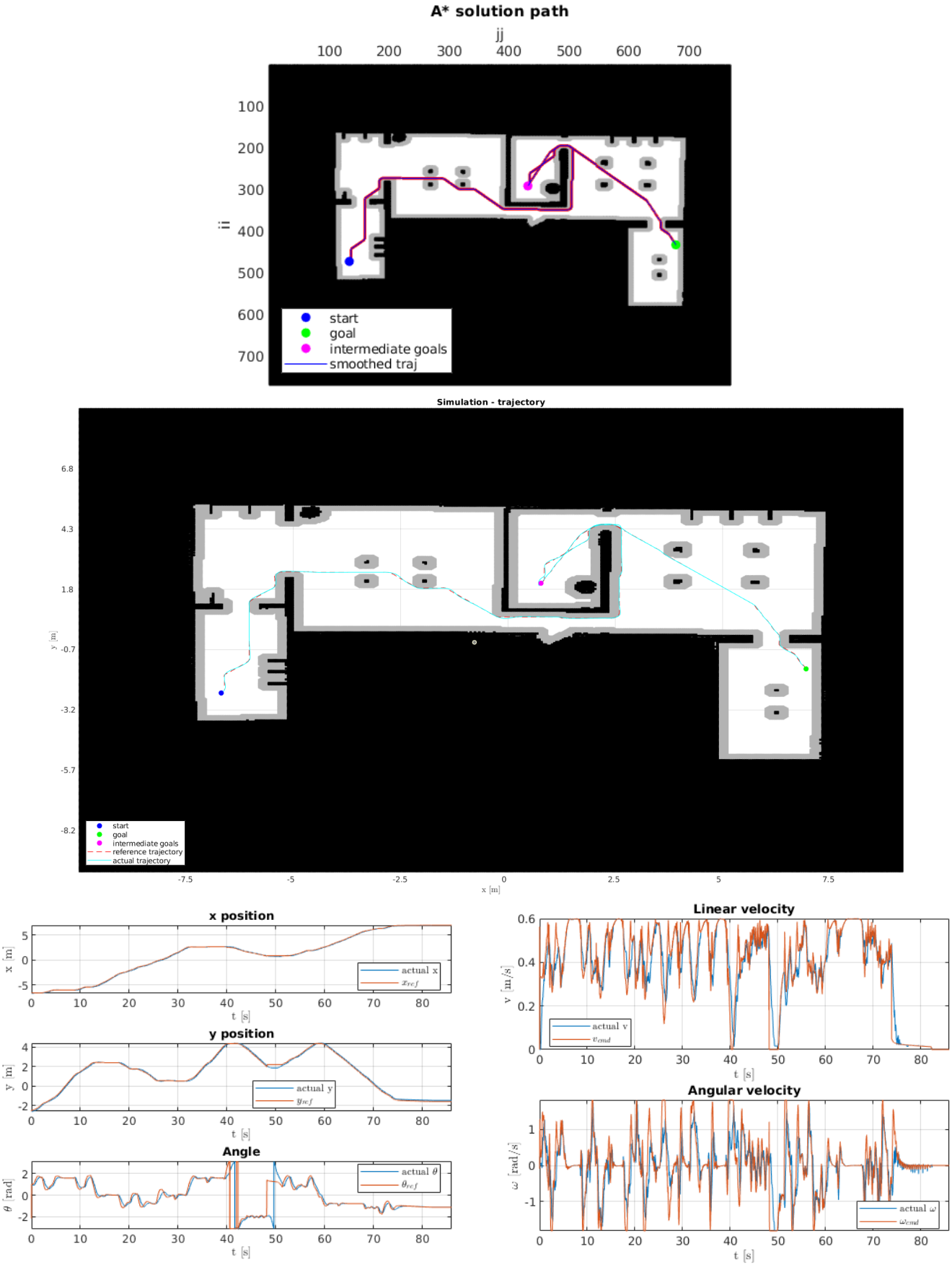


Figure 27: Test 5 (A planning with an intermediate goal and the "fast" robot)*

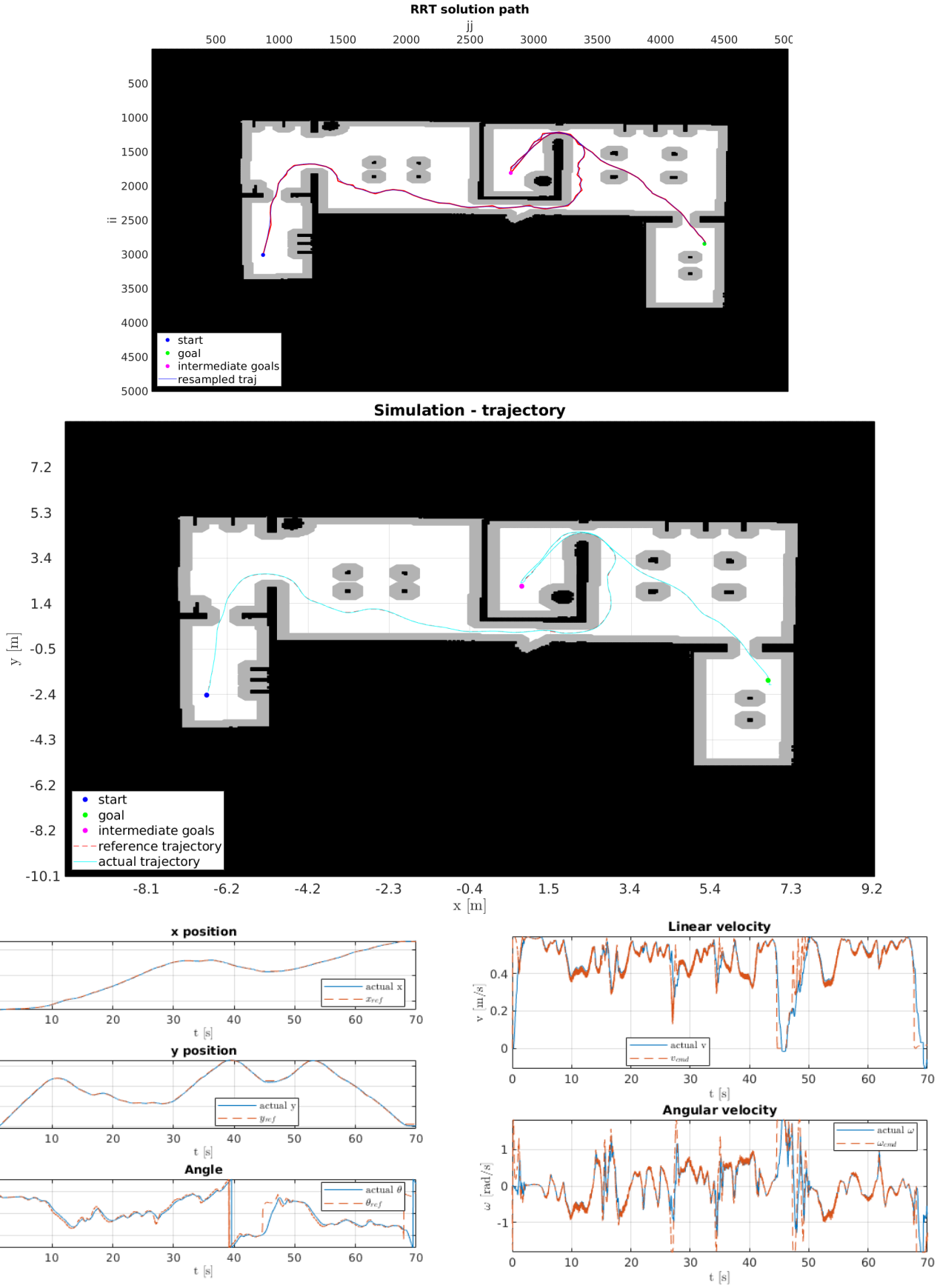


Figure 28: Test 6 (informed RRT planning with an intermediate goal and the "fast" robot)*

Concerning the two tests with $v_{max} = 0.6$ m/s (*figures 27,28*), we can see from the trajectory plots that the tracking capabilities of the robot are slightly less accurate. However, the `turtlebot` is still able to complete its task, and the travel time is much lower. Both A* and RRT* are tested with the same waypoints in this case, and the first part of the path (between the start and the intermediate goal) is the same of Test 1 (*figure 23*): in that case, 85 seconds were required to reach the end, while now only 52 seconds are needed (45 with RRT*). Moreover, comparing the velocities plots, the control in this scenario seems to be less scattered: this may be due to the introduction of the *lookahead* parameter and of the smoothing of the planned path. Another difference is that, as expected, the linear velocity is now varying during the trajectory, adapting to the curvature and not being always saturated to its maximum value.
Finally, comparing Test 5 with Test 6, even if A* algorithm is giving the shortest path, Test 6 (RRT*) lasts less time. This can be explained by looking, in particular, at the linear velocity: RRT* gives a slightly longer but smoother trajectory, so the robot needs to "slow down" less.

References

[1] Steven M. LaValle. *Rapidly-Exploring Random Trees: A New Tool for Path Planning*, 1998.

[2] Steven M.LaValle and James J. Kuffner Jr. *RRT-Connect: An Efficient Approach to Single-Query Path Planning*, 2000.

[3] Sertac Karaman and Emilio Frazzoli. *Sampling-based Algorithms for Optimal Motion Planning*, 2011.

[4] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. *Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic*, 2014.