MT_RRT, the general purpose multi threading library for RRT.

1.0

Generated by Doxygen 1.8.17

Chapter 1

Foundamental concepts

1.1 What is an RRT algorithm?

Rapidly Random exploring Tree(s), aka RRT(s), is one of the most popular technique adopted for solving planning path problems in robotics. In essence, a planning problem consists of finding a feasible trajectory or path that leads a manipulator, or more in general a dynamical system, from a starting configuration/state to an ending desired one, consistently with a series of constraints. RRTs were firstly proposed in [5]. They are able to explore a state space in an incremental way, building a search tree, even if they may require lots of iterations before terminating. They were proved be capable of always finding at least one solution to a planning problem, if a solution exists, i.e. they are probabilistic complete. RRT were also proved to perform well as kinodynamic planners, designing optimal LQR controllers driving a generic dynamical system to a desired final state, see [9] and [8].
The typical disadvantage of RRTs is that even for medium-complex problems, they require thousands of iterations to get the solution. For this reason, the aim of this library is to provide multi-threaded planners implementing parallel version of RRTs, in order to speed up the planning process.
It is possible to use this library for solving each possible problem tackled by an RRT algorithm. The only necessary thing to do when facing a new class of problem is to derive a specific object describing the problem itself.
Then it is clear that one of the most common problem one may solve with RRT is a standard path planning problem for an articulated arm. What matters in such cases is to have a collision checker, which is not provided by this library. Anyway, the interfaces Tunneled_check_collision and Bubbles_free_ configuration allows you to integrate the collision checker you prefer for solving standard path planning problems (see also Section 2.2.3).

The next Section briefly reviews the basic mechanism of the RRT. The notations and formalisms introduced in the next Section will be also adopted by the other Sections. Therefore, the reader is strongly encouraged to read before the next Section.
Section 1.3 will describe the typical pipeline to consider when using MT_RRT, while some examples of planning problems are reported in Chapter 2. Chapter 3 will describe the possible parallelization strategy that MT_RRT offers you. [1].

1.2 Background on RRT

[1] A similar guide, but in a html format, is also available at http://www.andreacasalino.altervista.org/__MT_RRT_doxy_guide/index.html.

1.2.1 Standard RRT

RRTs are able to find a series of states connecting two particular ones: a starting state x_o and an ending one x_f. This is done by building a search tree $T(x_o)$ having x_o as root. Each node $x_i \in T$ is connected to its unique father $x_{fi} = Fath(x_f)$ by a trajectory $\tau_{fi \rightarrow i}$. The root x_o is the only node not having a father ($Fath(x_o) = \emptyset$). The set $\mathcal{X} \subseteq \mathbb{R}^d$ will contain all the possible states x of the system whose motion must be controlled, while $\underline{\mathcal{X}} \subseteq \mathcal{X}$ is a subset describing the admissible region induced by a series of constraints. The solution we are interested in, consists clearly of a sequence of trajectories τ entirely contained in $\underline{\mathcal{X}}$. If we consider classical path planning problems, the constraints are represented by the obstacles populating the scene, which must be avoided. However, according to the nature of the problem considered, different kind of constraints might need to be accounted. The basic version of an RRT algorithm is described by Algorithm 1, whose steps are visually represented by Figure 1.2. Essentially, the tree is randomly grown by performing several steering operations. Sometimes, the extension of the tree toward the target state x_f is tried in order to find an edge leading to that state.

Data: x_o, x_f
$T = \{x_o\}$;
for $k = 1 : MAX\_ITERATIONS$ **do**
  sample $r \sim U(0, 1)$;
  if $r < \sigma$ **then**
    $x_{steered}$ =Extend(T, x_f);
    if $x_{steered}$ is $VALID$ **then**
      if $\|x_{steered} - x_f\| \leq \epsilon$ **then**
        Return Path_to_root($x_{steered}$)$\cup x_f$;
      end
    end
  end
  else
    sample a $x_R \in \mathcal{X}$;
    Extend(T, x_R);
  end
end

Algorithm 1: Standard RRT. A deterministic bias is introduced for connecting the tree toward the specific target state x_f. The probability σ regulates the frequency adopted for trying the deterministic extension. The Extension procedure is described in algorithm 2.

Data: T, x_R
$x_{Nearest}$ =Nearest_Neighbour(T, x_R);
$x_{steered}$ = Steer($x_{Nearest}, x_R$);
if $x_{steered} \notin \underline{\mathcal{X}}$ **then**
  Mark $x_{steered}$ as $INVALID$;
end
if $x_{steered}$ is $VALID$ **then**
  $T = T \cup x_{steered}$;
end
Return $x_{steered}$;

Algorithm 2: The Extend procedure.

Data: T, x_R
Return $\underset{x_i \in T}{\mathrm{argmin}}(C(\tau_{i \rightarrow R}))$;

Algorithm 3: The Nearest_Neighbour procedure: the node in T closest to the given state x_R is searched.

The Steer function in algorithm 2 must be problem dependent. Basically, It has the aim to extend a certain state x_i already inserted in the tree, toward another one x_R. To this purpose, an optimal trajectory $\tau_{i \rightarrow R}$, agnostic of the constraints, going from x_i to x_R, must be taken into account. Ideally, the steering procedure

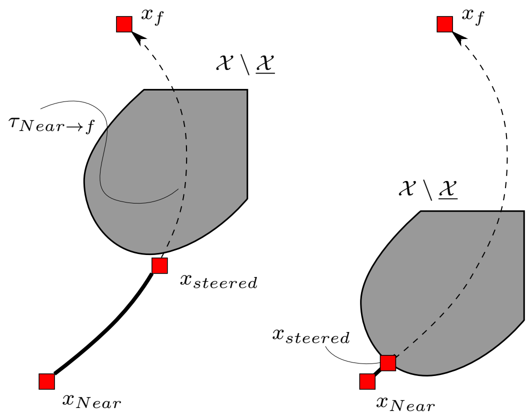


Figure 1.1 The dashed curves in both pictures are the optimal trajectories, agnostic of the constraints, connecting the pair of states x_{Near} and x_f, while the filled areas are regions of X not allowed by constraints. The steering procedure is ideally in charge of searching the furthest state to x_{Near} along $\tau_{Near \to f}$. For the example on the right, the steering is not possible: the furthest state along $\tau_{Near \to f}$ is too much closer to x_{Near}.

should find the furthest state from x_i that lies on $\tau_{i \to R}$ and for which the portion of $\tau_{i \to R}$ leading to that state is entirely contained in $\underline{\mathcal{X}}$. However, in real implementations the steered state returned might be not the possible farthest from x_i. Indeed, the aim is just to extend the tree toward x_R. At the same time, in case such the steered state results too closer to x_i, the steering should fails [2].

The Nearest_Neighbour procedure relies on the definition of a cost function $C(\tau)$. Therefore, the closeness of states does not take into account the shape of $\underline{\mathcal{X}}$. Indeed $C(\tau)$ it's just an estimate agnostic of the constraints. Then, the constraints are taken into account when steering the tree. The algorithm terminates when a steered configuration x_s sufficiently close to x_f is found.
The steps involved in the standard RRT are summarized by Figure 1.2.

1.2.2 Bidirectional version of the RRT

The behaviour of the RRT can be modified leading to a bidirectional strategy [6], which expands simultaneously two different trees. Indeed, at every iteration one of the two trees is extended toward a random state. Then, the other tree is extended toward the steered state previously obtained. At the next iteration, the roles of the trees are inverted. The algorithm stops, when the two trees meet each other. The detailed pseudocode is reported in Algorithm 4.

This solution offers several advantages. For instance, the computational times absorbed by the Nearest Neighbour search is reduced since this operation is done separately for the two trees and each tree contains at an average half of the states computed. The steps involved in the bidirectional strategy are depicted in Figure 1.3.

1.2.3 Compute the optimal solution: the RRT*

For any planning problem there are infinite $\tau_{o \to f} \subset \underline{\mathcal{X}}$, i.e. infinite trajectories starting from x_o and terminating in x_f which are entirely contained in the admissible region $\underline{\mathcal{X}}$. Among the aforementioned set, we might be interested in finding the trajectory minimizing the cost $C(\tau_{o \to f})$, refer to Figure 1.4. The basic version of the RRT algorithm is proved to find with a probability equal to 1, a suboptimal solution [4]. The optimality is addressed by a variant of the RRT, called RRT* [4], whose pseudocode is contained in Algorithm 5. Essentially, the RRT* after inserting in a tree a steered state, tries to undertake local improvements to the connectivity of the tree, in order to minimize the cost-to-go of the states in the $Near$ set. This approach is proved to converge to the optimal solution after performing an infinite number of iterations [3]. There are no precise stopping criteria for the RRT*: the more iterations are performed, the more the solution found get closer to the optimal one.

[2] This is done to avoid inserting less informative nodes in the tree, reducing the tree size.
[3] In real cases, after a sufficient big number of iterations an optimizing effect can be yet appreciated.

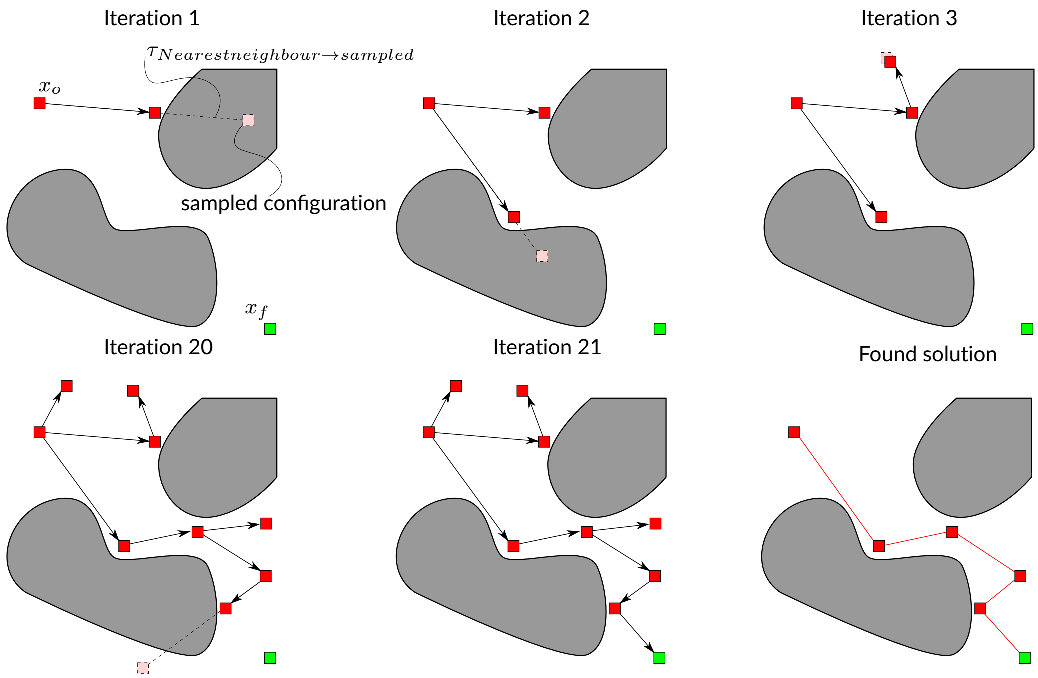


Figure 1.2 Examples of iterations done by an RRT algorithm. The solution found is the one connecting the state in the tree that reached x_f, with the root x_o.

Data: x_o, x_f
$T_A = \{x_o\}$;
$T_B = \{x_f\}$;
$x_{target} =$ root of T_A;
$x_2 =$ root of T_B;
$T_{master} = T_A$;
$T_{slave} = T_B$;
for $k = 1 : MAX\_ITERATIONS$ **do**
  sample $r \sim U(0,1)$;
  if $r < \sigma$ **then**
    $x_{steered} =$Extend(T_{master}, x_{target});
  end
  else
    sample a $x_R \in \mathcal{X}$;
    $x_{steered} =$Extend(T_{master}, x_R);
  end
  if $x_{steered}$ is $VALID$ **then**
    $x_{steered2} =$Extend$(T_{slave}, x_{steered})$;
    if $x_{steered2}$ is $VALID$ **then**
      if $\|x_{steered} - x_{steered2}\| \leq \epsilon$ **then**
        Return Path_to_root($x_{steered}$) $\cup$ Revert $\Big($ Path_to_root($x_{steered2}$) $\Big)$;
      end
    end
  end
  Swap T_{target} and T_2 ;
  Swap T_{master} and T_{slave} ;
end

Algorithm 4: Bidirectional RRT. A deterministic bias is introduced for accelerating the steps. The probability σ regulates the frequency adopted for trying the deterministic extension. The Revert procedure behaves as exposed in Figure 1.3.

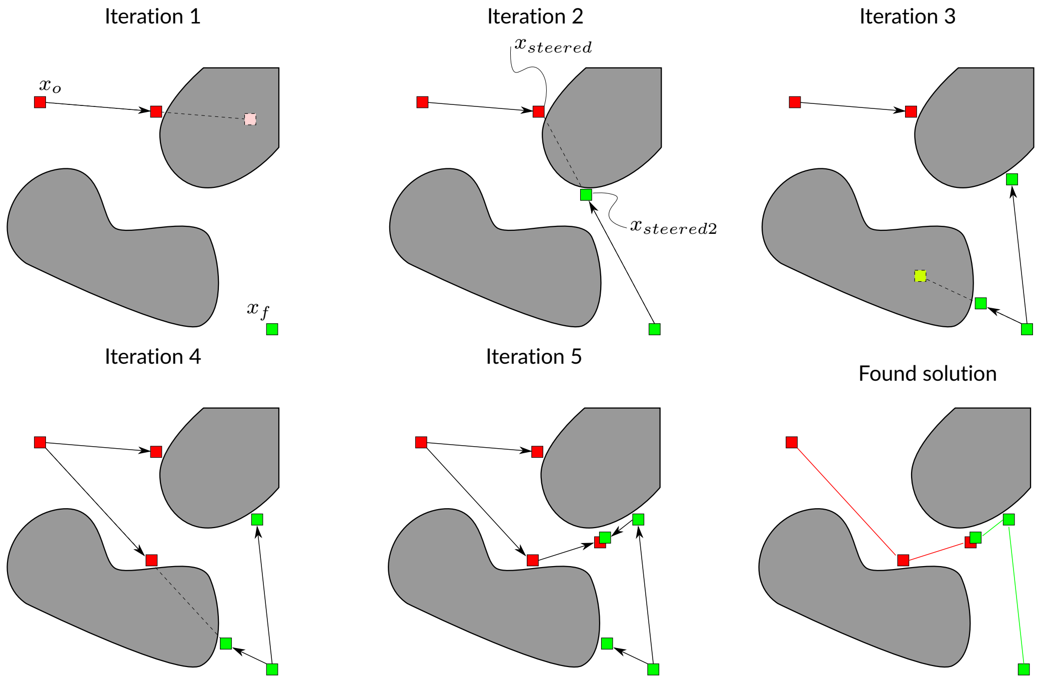


Figure 1.3 Examples of iterations done by the bidirectional version of the RRT. The path in the tree rooted at x_f is reverted to get the second part of the solution.

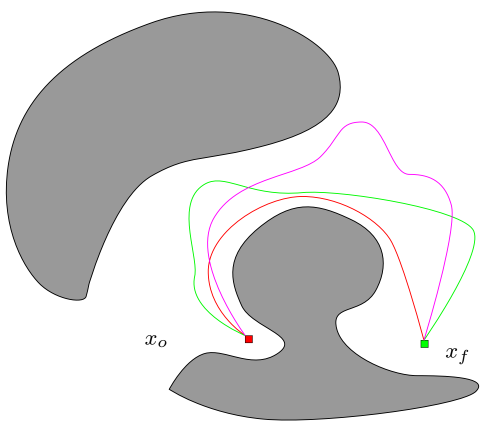


Figure 1.4 Different trajectories connecting x_o with x_f, entirely contained in $\underline{\mathcal{X}}$. If we assume ad cost the length of a path, the red solution is the optimal one.

Data: x_o, x_f
$T = \{x_o\}$;
$Solutions = \emptyset$;
for $k = 1 : MAX\_ITERATIONS$ **do**
    sample $r \sim U(0,1)$;
    if $r < \sigma$ **then**
        $x_{steered}$ =Extend_Star(T, x_f);
        if $x_{steered}$ *is* $VALID$ **then**
            if $\|x_{steered} - x_f\| \leq \epsilon$ **then**
                $Solutions = Solutions \cup x_{steered}$;
            end
        end
    end
    else
        sample a $x_R \in \mathcal{X}$;
        Extend_Star(T, x_R);
    end
end
$x_{best} = \underset{x_S \in Solutions}{\operatorname{argmin}}$ (Cost_to_root(x_S));
Return Path_to_root$(x_{best}) \cup x_f$;
Algorithm 5: RRT*. The Extend_Star, Rewird and Cost_to_root procedures are explained in, respectively, algorithm 6, 7 and 8.

Data: T, x_R
$x_{steered}$ = Extend(T, x_R);
if $x_{steered}$ *is* $VALID$ **then**
    $Near = \left\{ x_i \in T \middle| C(\tau_{i \to steered}) \leq \gamma(\frac{log(|T|)}{|T|})^{\frac{1}{d}} \right\}$;
    Rewird($Near, x_{steered}$);
end
Return $x_{steered}$;
Algorithm 6: The Extend_Star procedure. d is the cardinality of $\mathcal{X}$.

Data: $Near, x_s$
$x_{bestfather} = Fath(x_s)$;
$C_{min} = C(\tau_{bestfather \to s})$;
for $x_n \in Near$ **do**
    if $\tau_{n \to s} \subset \underline{\mathcal{X}}$ *AND* $C(\tau_{n \to s}) < C_{min}$ **then**
        $C_{min} = C(\tau_{n \to s})$;
        $x_{bestfath} = x_n$;
    end
end
$Fath(x_s) = x_{bestfath}$;
C_s =Cost_to_root(x_s);
$Near = Near \setminus x_{bestfath}$;
for $x_n \in Near$ **do**
    if $\tau_{s \to n} \subset \underline{\mathcal{X}}$ **then**
        $C_n = C(\tau_{s \to n}) + C_s$;
        if $C_n <$ *Cost_to_root*(x_n) **then**
            $Fath(x_n) = x_s$;
        end
    end
end
Algorithm 7: The Rewird procedure.

Data: x_n
if $Fath(x_n) = \emptyset$ **then**
  Return 0;
end
else
  Return $C(\tau_{Fath(n) \rightarrow n})+$ Cost_to_root($Fath(x_n)$) ;
end
Algorithm 8: The Cost_to_root procedure computing the cost spent to go from the root of the tree to the passed node.

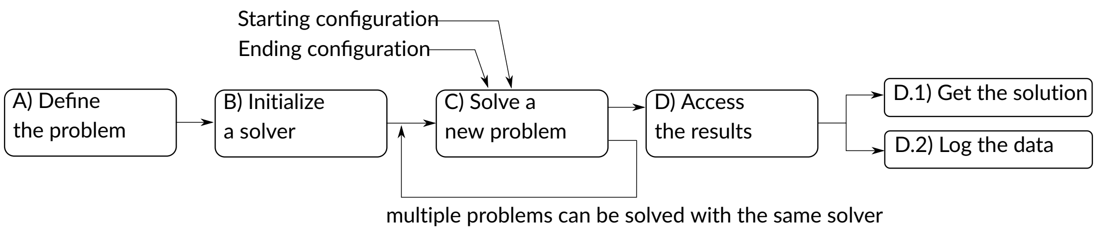


Figure 1.5 Pipeline to follow for using MT_RRT.

1.3 MT_RRT pipeline

When solving a planning problem with MT_RRT, the pipeline of Figure 1.5 should be followed.

1.3.0.1 A) Define the problem

First of all, you need to derive a class that tells MT_RRT how your problem is made, i.e. you need to defined how to compute the optimal trajectories τ, the shape of the constraints admitted region $\underline{\mathcal{X}}$, how to perform the steering of a node in the tree, etc.. This must be done by deriving a specific object from the interface called Node::I_Node_factory. The object derived is able to describe the kind of problem, without the need to know the particular starting and ending configurations that you need to join with RRT. Therefore, you can recycle this object for resolving different RRT planning, addressing the same kind of problem. Chapter 2 reports some examples of planning problems, describing how to derive the corresponding Node::I_Node_factory object.

1.3.0.2 B) Initialize a solver

After having defined the problem, you need to build a solver, to be used to solve later a planning problem. The solver can be: a serial standard solver or a multi-threaded solver. In the first case, you must use Planner_canonical and you are basically using the standard RRT versions described at the beginning in Section 1.2.1. In the second case you can chose one of the approach described in Chapter 3, exploiting multi-threading to reduce the computation times.

1.3.0.3 C) Solve the problem with a certain strategy

One single solver can be used to solve multiple problems, using one of the strategies discussed in Sections 1, 1.2.2 and 5. This can be done by calling RRT_basic, RRT_bidirectional or RRT_star on the built solver. Results are internally stored into the solver and can be later accessed as explained in the next paragraph. Clearly, only the results concerning the last found solution are saved, i.e. data are overwritten when calling multiple times RRT_basic (and the other two methods) for solving different problems of the same category, but with different starting and ending configurations.

json version

Solution: $\{x_1, \dots, x_j, \dots, x_M\} \Longrightarrow$ $" \left[[x_1^1, \dots, x_1^n], \dots, [x_M^1, \dots, x_M^n] \right] "$

$$x_j \in \mathbb{R}^n \rightarrow \begin{bmatrix} x_j^1 \\ \vdots \\ x_j^n \end{bmatrix}$$

json version

Trees: $\left[Tree_1 \right]$, or $\left[Tree_1, Tree_2 \right]$

$$Tree = \{x_{t1}, x_{t2}, \dots, x_{tT}\} \Longrightarrow " \Big[\{"E" : [x_{t1}^1, \dots, x_{t1}^n], "S" : [x_{Fath(t1)}^1, \dots, x_{Fath(t1)}^n]\}$$

$$, \{"E" : [x_{t2}^1, \dots, x_{t2}^n], "S" : [x_{Fath(t2)}^1, \dots, x_{Fath(t2)}^n]\}$$

$$\vdots$$

$$, \{"E" : [x_{tT}^1, \dots, x_{tT}^n], "S" : [x_{Fath(tT)}^1, \dots, x_{Fath(tT)}^n]\} \Big] "$$

Figure 1.6 Format of the json file returning the results.

1.3.0.4 D) Access the results

To access the results computed by a solver there are two possible ways.

D.1. The first way is to get the waypoints representing the solution, i.e. a series of states $x_{1,2,3,\dots,M}$ that must be visited to get from the starting configuration to the ending one, by traversing the trajectories $\tau_{1\rightarrow 2}, \tau_{2\rightarrow 3}, \dots, \tau_{M-1\rightarrow M} \subset \underline{\mathcal{X}}$. Such series of waypoints can be obtained by calling I_Planner::Get_solution, which returns a list of configurations. In case a solution was not found, an empty list is externally returned.

D.2. The second way to get the results is to log them into a json string. This can be done by using two possible methods. I_Planner::Get_Solution_as_JSON is similar to I_Planner::Get_solution, but returns the waypoints representing the solution as a json, i.e. an array of arrays refer to top of Figure 1.6. On the opposite, I_Planner::Get_Trees_as_JSON returns a json structure that describes the tree(s) [4] computed by the solver in order to get the solution [5]. The bottom part of Figure 1.6 shows the structure of the json representing a single searching tree.

[4] A single tree is addressed when using RRT_basic or RRT_star, while two trees are computed when considering RRT_bidirectional.
[5] In case of the solver described in Sections 3.0.3 and 3.0.4 the tree owned by the main thread is returned.

Chapter 2

Customize your own planning problem

MT_RRT can be deployed to solve each possible problems for which RRT can be used. The only thing to do is to derive a specific object from the interface called Node::I_Node_factory to have an object describing your particular problem. The methods contained in this object are in charge of sampling new random states in $\mathcal{X}$ or computing the optimal trajectories τ, which is the pre-requisite for performing steering operations (Figure 1.1). Such functions are problem-specific and for this reason they must be implemented every time a new kind of problem must be solved.
In order to help the user in understanding how to implement a derivation to Node::I_Node_factory, three main kind of examples are part of the library. In the following Sections, they will be briefly reviewed.

2.1 Planar maze problem

The state space characterizing this problem is a two dimensional one, having $x_{1,2}$ as coordinates. The aim is to connect two 2D coordinates while avoiding the rectangular obstacles depicted in Figure 2.1. The state space is bounded by two corners describing the maximum and minimum possible x_1 and x_2, see Figure 2.1.

2.1.1 Sampling

A sampled state x_R lies in the square delimited by the spatial bounds, i.e.:

$$x_R = \begin{bmatrix} x_{R1} \sim U(x_{1min}, x_{1max}) \\ x_{R2} \sim U(x_{2min}, x_{2max}) \end{bmatrix} \tag{2.1}$$

2.1.2 Optimal trajectory and constraints

The optimal trajectory $\tau_{i \rightarrow k}$ between two states in $\mathcal{X}$ is simply the segment connecting that states. The cost $C(\tau_{i \rightarrow k})$ is assumed to be the length of such segment:

$$C(\tau_{i \rightarrow k}) = \|x_i - x_k\| \tag{2.2}$$

The admissible region $\underline{\mathcal{X}}$ is obtained subtracting the points pertaining to the obstacles. In other words, the segment connecting the states in the tree should not traverse any rectangular obstacle, refer to the right part of Figure 2.1.

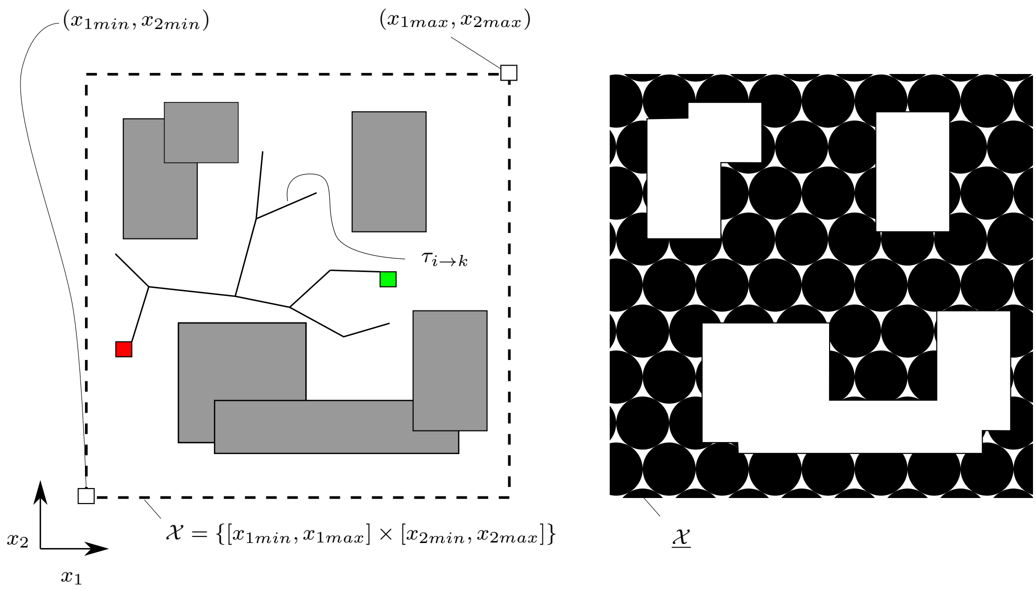
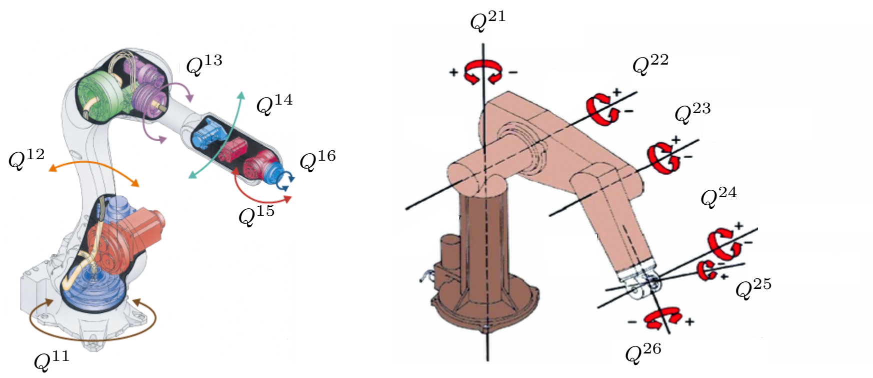


Figure 2.1 Example of maze problem.



$$Q^1 = \begin{bmatrix} Q^{11} & Q^{12} & Q^{13} & Q^{14} & Q^{15} & Q^{16} \end{bmatrix}^T \qquad Q^2 = \begin{bmatrix} Q^{21} & Q^{22} & Q^{23} & Q^{24} & Q^{25} & Q^{26} \end{bmatrix}^T$$

Figure 2.2 Rotating joints of two articulated manipulators.

2.1.3 Steer procedure

The steering procedure is done as similarly described in Section 2.2.3, checking a close state $x_{steered}$ that lies on the segment departing from the state to steer.

2.2 Articulated arm problem

This is for sure one of the most common problem that can be solved using MT_RRT. Consider a cell having a group of articulated serial robots. Q^i will denote the vector describing the configuration of the i^{th} robot, i.e. the positional values assumed by each of its joint. A generic state x_i is characterized by the series of poses assumed by all the robots in the cell:

$$x_i = Q_i = \begin{bmatrix} (Q_i^1)^T & \dots & (Q_i^n)^T \end{bmatrix}^T \tag{2.3}$$

refer also to Figure 2.2.

These kind of problems consist in finding a path in the configurational space that leads the set of robots from an initial state Q_o to and ending one Q_f, while avoiding the obstacles populating the scene, i.e. avoid collisions between any object in the cell and any part of the robots as well as cross-collision between all the robot parts. Here the term path, refer to a series of intermediate waypoints $Q_{1,\dots,m}$ to traverse to lead the robot from Q_o to Q_f.

2.2.1 Sampling

The i^{th} joint of the k^{th} robot, denoted as Q^{ki}, is subjected to some kinematic limitations prescribing that its positional value must remain always within a compact interval $Q^{ki} \in [Q^{ki}_{min}, Q^{ki}_{max}]$. Therefore, the sampling of a random configuration Q_R is done as follows:

$$Q_R = \begin{bmatrix} Q_R^{11} \sim U(Q^{11}_{min}, Q^{11}_{max}) \\ Q_R^{12} \sim U(Q^{11}_{min}, Q^{11}_{max}) \\ \vdots \\ Q_R^{21} \sim U(Q^{21}_{min}, Q^{21}_{max}) \\ Q_R^{22} \sim U(Q^{21}_{min}, Q^{21}_{max}) \\ \vdots \\ Q_R^{n1} \sim U(Q^{n1}_{min}, Q^{n1}_{max}) \\ Q_R^{n2} \sim U(Q^{n1}_{min}, Q^{n1}_{max}) \\ \vdots \end{bmatrix} \tag{2.4}$$

2.2.2 Optimal trajectory and constraints

Similarly to the problem described in Section 2.1.2, $\tau_{i \rightarrow k}$ is assumed to be a segment in the configurational space and the cost C is the Euclidean distance of a pair of states. The admissible region $\underline{X}$ is made by all the configurations Q for which a collision is not present.

2.2.3 Steer procedure

The trajectory going from Q_i to Q_k can be parametrized in order to characterize all the possible configurations pertaining to $\tau_{i \rightarrow k}$:

$$Q(s) = \tau_{i \rightarrow k}(s) = Q_i + s\Big(Q_k - Q_i\Big) \tag{2.5}$$

s is a parameter spanning $\tau_{i \rightarrow k}$ and can assume a value inside $[0, 1]$. Ideally, the steer process has the aim of determine that state $Q(s_{steered})$ that is furthest from Q_i and at the same time contained in $\underline{X}$ (Figure 1.1). Anyway, determine the exact value of $s_{steered}$ would be too much computationally demanding. Therefore, in real situations, two main approaches are adopted: a tunneled check collision or the bubble of free configuration.

2.2.3.1 Tunneled check collision

This approach consider as steered state $Q_{steered}$ the following quantity:

$$Q_{steered} = \begin{cases} \text{if}(\|Q_k - Q_i\| \leq \epsilon) \Rightarrow Q_k \\ \textit{else} \Rightarrow Q_i + s_{\Delta}(Q_k - Q_i) \text{ s.t. } s_{\Delta} \|Q_k - Q_i\| = \epsilon \end{cases} \tag{2.6}$$

with ϵ in the order of few degrees. $Q_{steered}$ is checked to be or not in $\underline{X}$ and is consequently marked as $VALID$ or $INVALID$. The class Tunneled_check_collision is in charge of implementing such an extension strategy. It absorbs an object of type I_Collision_checker to check whether for a certain state are present or not collisions. I_Collision_checker is just an interface: you can integrate your own collision checker (using for example [1] or [2]) by deriving an object from this interface.
Clearly, multiple tunneled check, starting from Q_i, can be done in order to get as close as possible to Q_k. This process can be arrested when reaching Q_k or an intermediate state for which a collision check is not passed. This behaviour can be obtained by using Node_factory_multiple_steer, absorbing a Tunneled_check_collision object.
Figure 2.3 summarizes the above considerations.

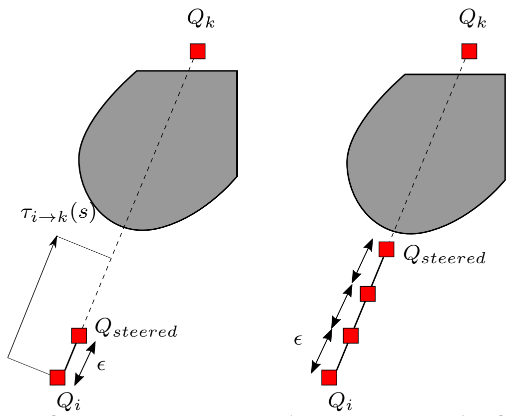


Figure 2.3 Steer extension along the segment connecting two states in the configuration space. On the left a single steer approach, on the right a multiple one.

2.2.3.2 Bubble of free configuration

This approach was first proposed in [7] and is based on the definition of a so called bubble of free configuration $\mathcal{B}$. Such a bubble is a region of the configurational space that is built around a state Q_i. More formally, $\mathcal{B}(Q_i)$ is defined as follows:

$$\mathcal{B}(\bar{Q} = \left[\bar{Q}^{1T} \quad \dots \quad \bar{Q}^{nT}\right]^T) = \left\{ \bar{Q} \middle| \quad \forall \quad j \in \{1, \dots, n\} \sum_i r^{ji} |Q^{ji} - \bar{Q}^{ji}| \leq d^j_{min} \text{ and} \right.$$
$$\left. \forall \quad j,k \in \{1, \dots, n\} \sum_i r^{ji} |Q^{ji} - \bar{Q}^{ji}| + \sum_i r^{ki} |Q^{ki} - \bar{Q}^{ki}| \leq d^{jk}_{min} \right\} \tag{2.7}$$

where d^j_{min} is the minimum distance between the i^{th} robot and all the obstacles in the scene, while d^{jk}_{min} is the minimum distance between the i^{th} and the k^{th} robot. r^{ki} is the distance of the furthest point of the shape of the k^{th} robot to its i^{th} axis of rotation. Refer also to Figure 2.4.
Each configuration $Q \in \mathcal{B}$ is guaranteed to be inside the admitted region $\underline{X}$. This fact can be exploited for performing steering operation. Indeed, we can take as $Q_{steered}$ the pose at the border of $\mathcal{B}(Q_i)$ along the segment connecting Q_i to Q_k. It is not difficult to prove that such a state is equal to:

$$\begin{aligned} Q_{steered} &= \left[Q^{1T}_{steered} \quad \dots \quad Q^{nT}_{steered}\right]^T = Q_i + s_{steered}(Q_k - Q_i) \\ s_{steered} &= min\left\{ s_A, s_B \right\} \\ s_A &= min_{j \in \{1,\dots,n\},q} \left\{ \frac{d^j_{min}}{r^{jq}|Q^{jq}_i - Q^{jq}_k|} \right\} \\ s_B &= min_{j,k \in \{1,\dots,n\},q,q_2} \left\{ \frac{d^{jk}_{min}}{r^{jq}|Q^{jq}_i - Q^{jq}_k| + r^{kq2}|Q^{kq_2}_i - Q^{kq_2}_k|} \right\} \end{aligned} \tag{2.8}$$

Also in this case a multiple steer approach is possible for this strategy, by using Node_factory_multiple_steer, absorbing a Bubbles_free_configuration object, refer also to Figure 2.5.
Bubbles_free_configuration contains the functionalities for performing steering operations using the bubble of free configuration. Then, you have to deploy your own geometric engine in order to compute the distances d^j_{min}, d^{jk}_{min} as well as the radii r^{ki}, deriving an object from I_Proximity_calculator. Robots_info stored in these kind of objects is a structure containing the distance d^j_{min}, as well as the radii r^{ki} (with an order that goes from the base to the end effector), while Robot_distance_pairs is a buffer of distances storing all the possible d^{jk}_{min}, with the order indicated in Figure 2.6.

2.3 Navigation problem

This problem is typical when considering autonomous vehicle. We have a 2D map in which a cart must move. In order to simplify the collision check task, a bounding box $\mathcal{L}$ is assumed to contain the entire

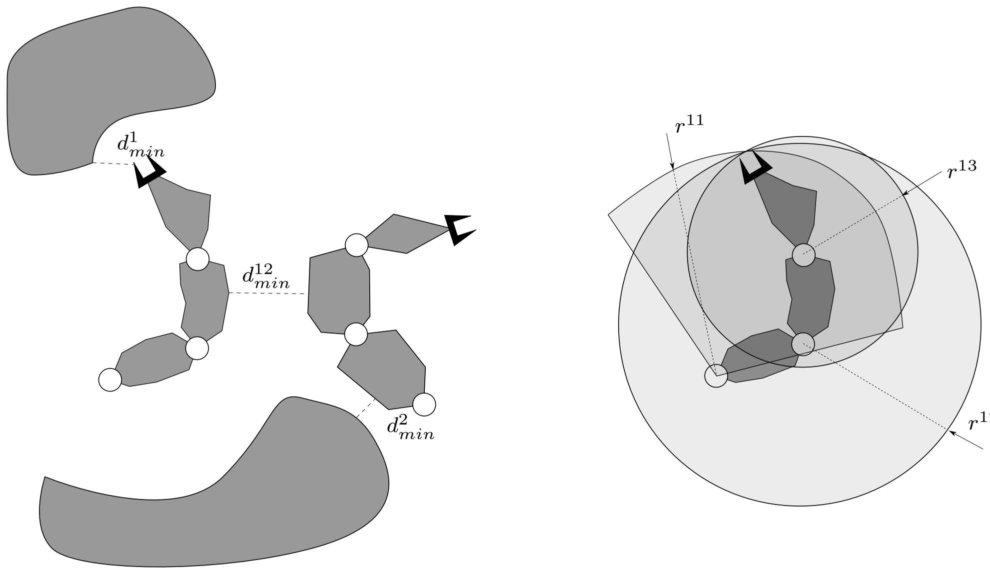


Figure 2.4 The quantities involved in the computation of the bubble $\mathcal{B}$.

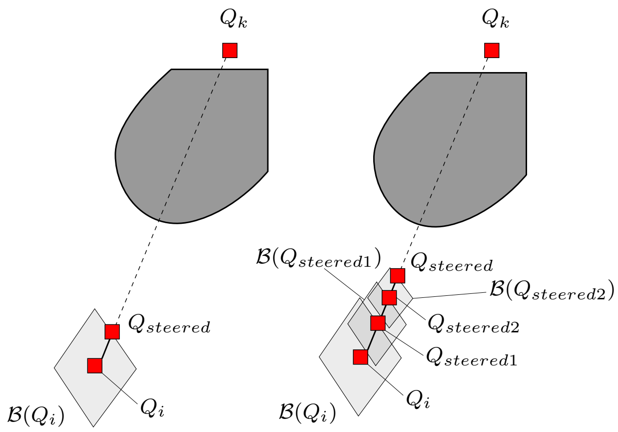


Figure 2.5 Single (left) and multiple (right) steer using the bubbles of free configurations.

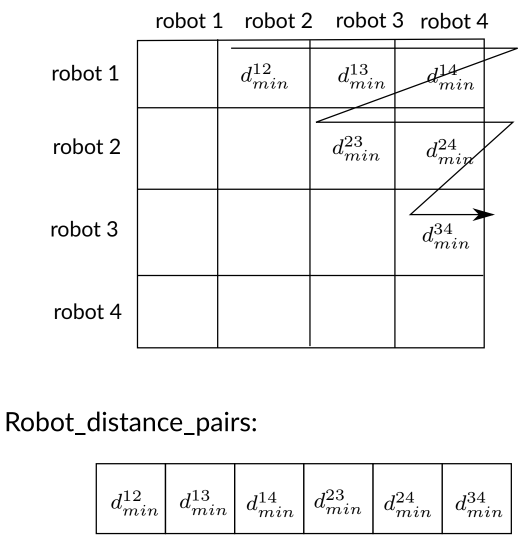


Figure 2.6 Values stored in Robot_distance_pairs.

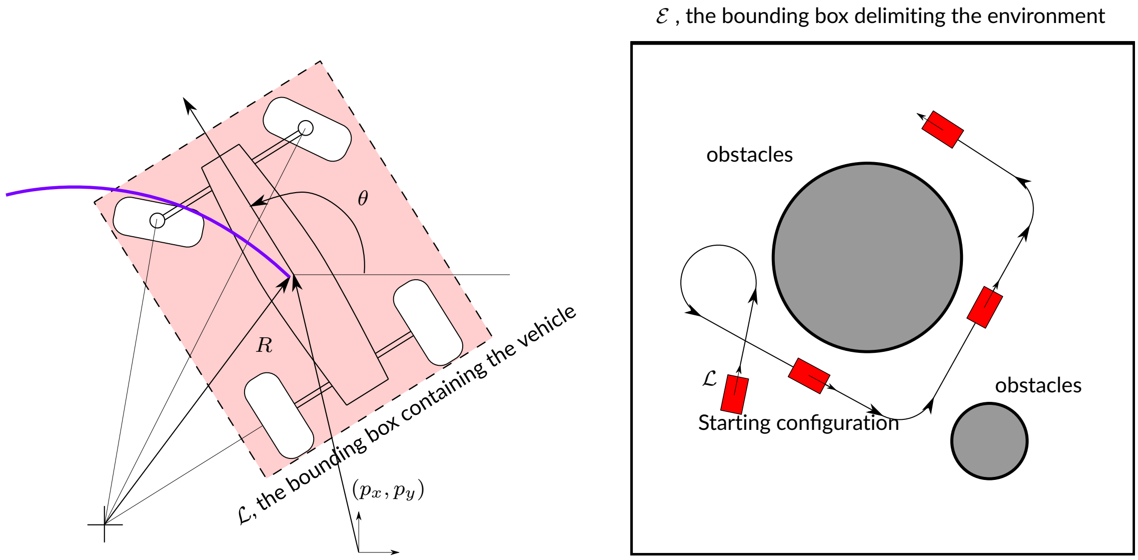


Figure 2.7 Vehicle motion in a planar environment.

shape of the vehicle, Figure 2.7. The cart moves at a constant velocity when advancing on a straight line and cannot change instantaneously its cruise direction. Indeed, the cart has a steer, which allows to do a change direction by moving on a portion of a circle, refer to Figure 2.7. In order to simplify the problem, we assume that the steering radius must be constant and equal to a certain value R and the velocity of the cart while performing the steering maneuver is constant too.
Since the cart is a rigid body, its position and orientation in the plane can be completely described using three quantities: the coordinates p_x, p_y of its center of gravity and the absolute angle θ. Therefore, a configuration $x_i \in \mathcal{X}$ is a vector defined as follows: $x_j = \begin{bmatrix} p_{xi} & p_{yi} & \theta_i \end{bmatrix}$. The admitted region $\underline{\mathcal{X}}$ is made by all the configurations x for which the vehicle results to be not in collision with any obstacles populating the scene.

2.3.1 Sampling

The environment where the vehicle can move is assumed to be finite and equal to a bounding box $\mathcal{E}$ with certain sizes, right portion of Figure 2.7. The sampling of a random configuration for the vehicle is done in this way:

$$x_R = \begin{bmatrix} (p_{xi}, p_{yi}) \sim \mathcal{E} \\ \theta \sim U(-\pi, \pi) \end{bmatrix} \tag{2.9}$$

2.3.2 Optimal trajectory and constraints

The optimal trajectory connecting two configurations x_i, x_j is made of three parts (refer to the examples in the right part of Figure 2.7 and the top part of Figure 2.8):

- a straight line starting from x_i
- a circular portion motion used to get from θ_i to θ_j
- a straight line ending in x_j

The cost $C(\tau)$ is assumed to be the total length of τ. It is worthy to remark that not for every pair of configurations exists a trajectory connecting them, refer to Figure 2.8. Therefore, in case the trajectory $\tau_{i\rightarrow j}$ does not exists, $C(\tau_{i\rightarrow j})$ is assumed equal to $+\infty$.

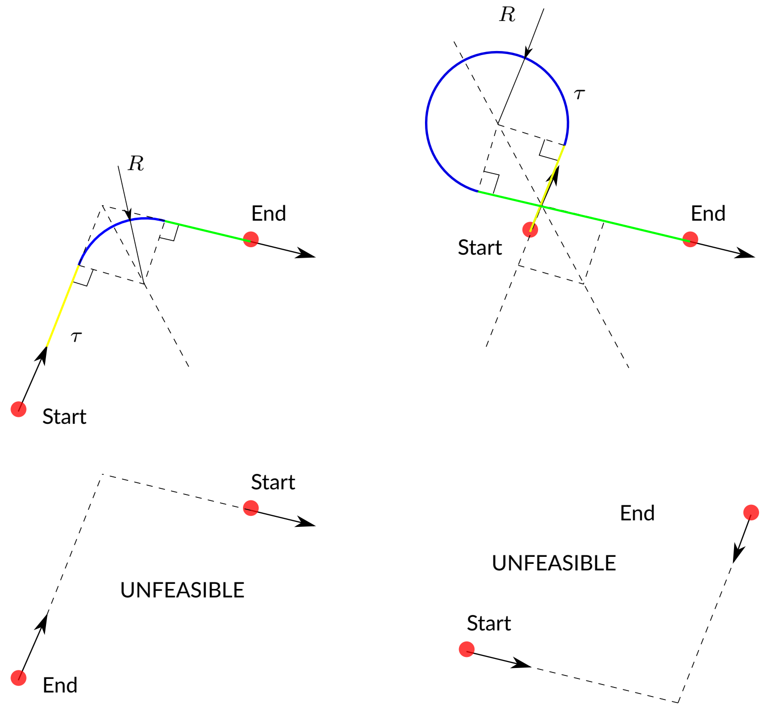


Figure 2.8 Examples of feasible, top, and non feasible trajectories, bottom. The different parts of the feasible trajectories are highlighted with different colors.

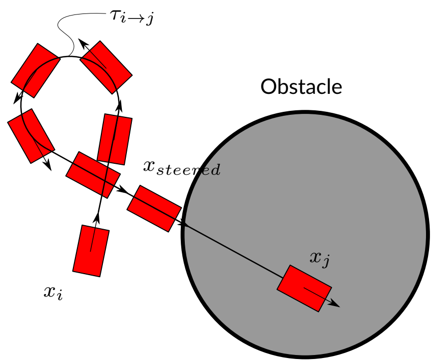


Figure 2.9 Steering procedure for a planar navigation problem.

2.3.3 Steer procedure

The steering from a state x_i toward another x_j is done by moving along the trajectory $\tau_{i \to j}$, advancing every time of a little quantity of space (also when traversing the circular part of the trajectory). The procedure is arrested when a configuration not lying in $\underline{\mathcal{X}}$ is found or x_j is reached. Figure 2.9 summarizes the steering procedure.

Chapter 3

Parallel RRT

This Chapter will provide details about the strategies adopted for parallelizing RRT that MT_RRT contains. Further details are contained also in [3], which is the publication were for the first time MT_RRT was presented. In [3] you can find also a comparison in terms of computational times.
Each strategy described in the following Sections is able to parallelize the three RRT versions exposed in Sections 1.2.1, 1.2.2 and 1.2.3. The only exception must be made only for the strategy exposed in Section 3.0.4, for which a bidirectional RRT (Section 1.2.2) cannot be applied.

3.0.1 Parallelization of the query activities

All the RRT versions spend a significant time in performing query operations on the tree, i.e. operations that require to traverse all the tree. Such operations are mainly the nearest neighbour search, algorithm 3, and the determination of the near set, algorithm 6.
The key idea is to perform the above query operations with a parallel for, where at an average all the threads process the same amount of nodes in the tree, computing their distances for determine the nearest neighbour or the near set. The parallel regions are not re-opened and closed every time, but a thread pooling strategy is adopted: all the threads are spawn when a new planning problem must be solved and remain active and ready to perform the parallel for described before. All the operations of the RRT (regardless the version considered) are done by the main thread, which notifies at the proper time when a new query operation must be process collectively by all the threads. Figure 3.1.a summarizes the approach.
The class implementing this approach is Planner_query_parall.

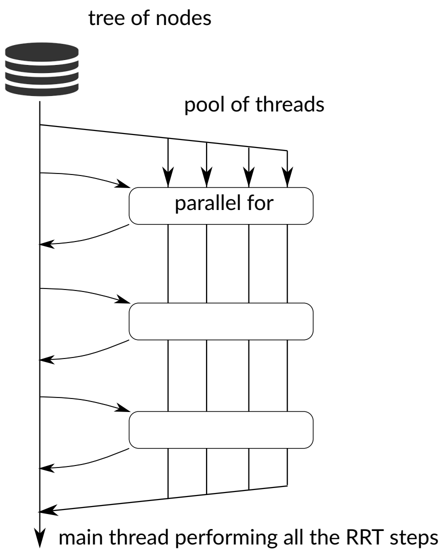
3.0.2 Shared tree critical regions

Another way to obtain a parallelization is to actually do simultaneously, every single step of the RRT versions. Therefore, we can imagine having threads sharing a common tree (or two trees in the case of a bidirectional strategy), executing in parallel every step of the expansion process. Some critical sections must be designed to allow the threads executing the maintenance of the shared tree(s) (inserting new nodes or executing new rewirds) one at a time. More precisely, the steer is done outside and only the insertion of the steered configuration in the tree is performed inside a critical region. Similarly, the extending procedure of the RRT*, algorithm 6, is modified by shifting the determination of the near set and the Rewird procedure in a critical section. Figure 3.1.b summarizes the approach.
The class implementing this approach is Planner_shared_parall.

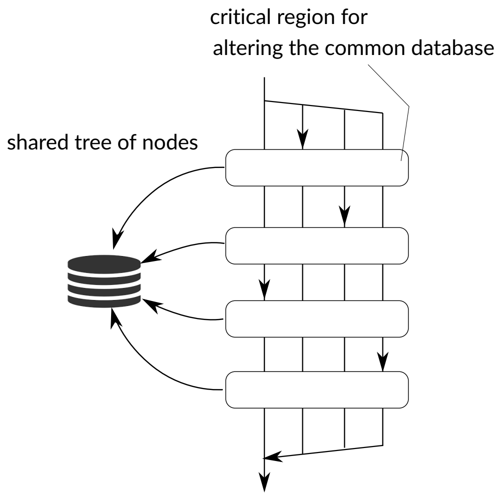
3.0.3 Parallel expansions of copied trees

To limit as much as possible the overheads induced by the presence of critical sections, we can consider a version similar to the one proposed in the previous Section, but for which every thread has a private copy of the search tree. After a new node is added by a thread to its own tree, $P-1$ copies are computed and dispatched [1] to the other threads, were P is the number of working threads. Sporadically, all the threads

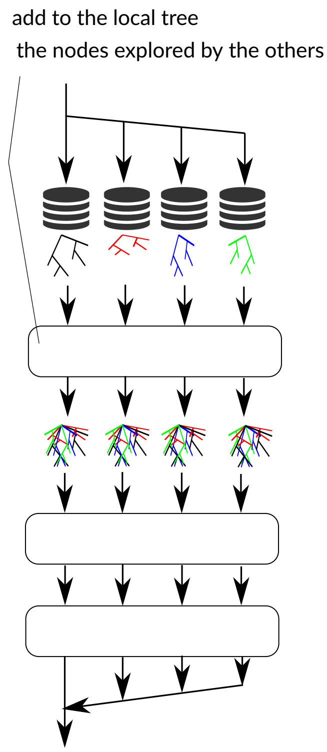
[1] They are dispatched into proper buffer, but not directly inserted in the private copies of the other trees.



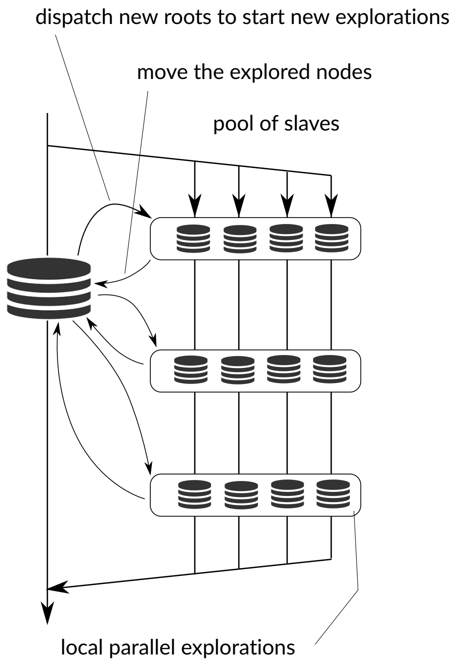
(a) Schematic representation of the parallelization of the query activities approach.



(b) Schematic representation of the parallel extensions of a common tree approach.



(c) Schematic representation of the parallel expansions of copied trees approach.



(d) Schematic representation of the multi agent approach.

Figure 3.1 Approaches adopted for parallelize RRT.

take into account the list of nodes received from the others and insert them into their private trees. This mechanism is able to avoid the simultaneous modification of a tree by two different threads, avoiding the use of critical sections.
When considering the bidirectional RRT, the mechanism is analogous but introducing for every thread a private copy of both the involved trees.
Instead, the RRT* version is slightly modified. Indeed, the rewirds done by a thread on its own tree are not dispatched to the others. At the same time, each thread consider all the nodes produced and added to its own tree when doing their own rewirds. When searching the best solution at the end of all the iterations, the best connections among all the trees in every threads are taken into account. Indeed, the predecessor of a node is assumed to be the parent with the lowest cost to go among the ones associated to each clones. Figure 3.1.c summarizes the approach.
Clearly, the amount memory required by this approach is significantly high, since multiple copies of a node must live in the different threads. This can be a problem to account for.
The class implementing this approach is Planner_copied_parall.

3.0.4 Multi agents approach

The strategy described in this Section aims at exploiting a significant number of threads, with both a reduced synchronizing need and allocation memory requirements. To this purpose, a variant of the RRT was developed for which every exploring thread has not the entire knowledge of the tree, but it is conscious of a small portion of it. Therefore, we can deploy many threads to simultaneously explore the state space $\mathcal{X}$ (ignoring the results found by the other agents) for a certain amount of iterations. After completing this sub-exploration task, all data incoming from the agents are collected and stored in a centralized data base while the agents wait to begin a new explorative batch, completely forgetting the nodes found at the previous iteration. The described behaviour resembles one of many exploring ants, which reports the exploring data to a unique anthill.
Notice that there is no need to physically copy the states computed by the agents when inserting them into the central database, since threads share a common memory: the handler of the node is simply moved.
When considering this approach a bidirectional search is not implementable, while the RRT* can be extended as reported in the following. Essentially, the agents perform a standard non-optimal exploration, implementing the steps of a canonical RRT, Section 1.2.1. Then, at the time of inserting the nodes into the common database, the rewirds are done by the main thread.
The described multi agent approach is clearly a modification of the canonical RRT versions, since the agents start exploring every time from some new roots, ignoring all the previously computed nodes. However, it was empirically found that the global behaviour of the path search is not deteriorated and the optimality properties of the RRT* seems to be preserved.
Before concluding this Section it is worthy to notice that the mean time spent for the querying operations is considerably lower, since such operations are performed by agents considering only their own local reduced size trees.
Figure 3.1.d summarizes the approach. The class implementing this approach is Planner_multi_agents.

Chapter 4

Namespace Index

4.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 5

Hierarchical Index

5.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 6

Class Index

6.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 7

Namespace Documentation

7.1 MT_RTT Namespace Reference

Classes

- struct bidir_solution
- class Bidirectional_Extension_job

  Handles the bidirectional extension strategy (Section 1.2.2 of the documentation).
- class Bubbles_free_configuration

  In this object, the steering is done considering bubbles of free configuration, Section 2.2.3 of the documentation.
- class I_Extension_job

  Interface for handling the extension steps involved in each RRT strategy.
- class I_Node_factory_decorator

  Interface for the generic I_Node_factory decorator.
- class I_Planner

  Interface for a planner.
- class I_Planner_MT

  This class contains common functionalities that every multi-threaded planner may use.
- class I_Tree

  Interface for handling a tree involved in a RRT strategy.
- class I_Tree_decorator

  A decorator may contain another decorator or a concrete Tree.
- class json_parser

  This class can be used to decode or encode a simple category of json structures.
- class Manipulator_path_handler

  Interface for handling classical path planning problem of a single or a group of articulated fixed robots (Section 2.2 of the documentation).
- class Node

  Used internally by a tree (see Tree.h) for representing a state x \in \underline{\mathcal{X}}, Section 1.2.1 of the documentation.
- class Node_factory_concrete

  Each handler describing a real planning problem must be derived from this class.
- class Node_factory_multiple_steer

  Used for performing each steer operation multiple times, trying to reach faster the target node.
- struct Node_State

  Used for externally represent a state x \in \mathcal{X}, Section 1.2.1 of the documentation.

- class Planner_canonical
- class Planner_copied_parall
- class Planner_multi_agents
- class Planner_query_parall
- class Planner_shared_parall
- class Single_Extension_job

    Handles the single tree extension strategy.
- struct single_solution
- class Tree_concrete

    Interface for a concrete case of the decorator pattern modelling the Tree class.
- class Tree_star

    This decorator perform the RRT∗ extension steps (near set computation, rewirds, etc. refer to Section 1.2.3 of the documentation).
- class Tunneled_check_collision

    In this object, the collision along a certain segment in the configurational space (i.e. a trajectory connecting two nodes) is checked considering a discrete set of equispaced samples, Section 2.2.3 of the documentation.

Functions

- template< typename T >
  bool **not_in_L** (const T &to_find, const std::list< T > &L)
- string **extract** (const string &s, const size_t &po, const size_t &pf)
- string **get_compact_JSON** (const string &JSON_raw)
- list< size_t > **find** (const string &JSON, const char &c)
- vector< float > **parse_array** (const string &to_parse)
- vector< vector< float > > **parse_arrays** (const string &to_parse)
- vector< vector< float > > **parse_field** (const string &JSON)
- string **stringify_field** (const json_parser::field &f)
- void **copy_solution** (list< Node_State > &receiving, const list< Node_State > &sending)
- void **round_robin_rewird_gather** (std::vector< std::list< Tree_star::Node2Node_Traj >> &Rewirds)
- float ∗ **alloc_state_copy** (const float ∗state, const size_t &Size)
- vector< float > **Equal** (const float &val, const size_t &Size)

7.1.1 Detailed Description

Author: Andrea Casalino Created: 16.05.2019

report any bug to andrecasa91@gmail.com.

Chapter 8

Class Documentation

8.1 MT_RTT::I_Planner::__last_solution_info Struct Reference

Public Attributes

- size_t **Iteration_done**
- std::list< Node_State > **Solution**
- std::list< I_Tree * > **Trees**

The documentation for this struct was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Planner.h

8.2 MT_RTT::I_Tree::_extend_info Struct Reference

Public Attributes

- bool **random_or_deter**
- bool **target_reached**

The documentation for this struct was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Tree.h

8.3 MT_RTT::bidir_solution Struct Reference

Public Member Functions

- **bidir_solution** (const float &c, const Node *A, const Node *B)
- **bidir_solution** (const bidir_solution &o)
- bool **operator==** (const bidir_solution &o) const
- bool **operator<** (const bidir_solution &o) const

Public Attributes

- float **cost**
- const Node ∗ **peer_A**
- const Node ∗ **peer_B**

The documentation for this struct was generated from the following file:

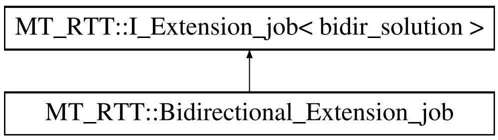
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Extensions.h

8.4 MT_RTT::Bidirectional_Extension_job Class Reference

Handles the bidirectional extension strategy (Section 1.2.2 of the documentation).

```
#include <Extensions.h>
```

Inheritance diagram for MT_RTT::Bidirectional_Extension_job:



Public Member Functions

- Bidirectional_Extension_job (I_Tree ∗to_extend_A, I_Tree ∗to_extend_B, const float ∗det_coeff, const bool ∗cumul_sol)

    Constructor.
- virtual void Extend_within_iterations (const size_t &Iterations)

    Perform the specified number of estensions on a wrapped tree(s).
- virtual std::list< I_Tree ∗ > Remove_Trees ()

    Move outside of this object all the trees it contains.

Protected Member Functions

- void **compute_sol** (bidir_solution &sol, const Node ∗N1, const Node ∗N2, const bool &caso)
- void **compute_cost** (bidir_solution &sol)
- virtual void **__Get_best_solution** (std::list< Node_State > ∗solution, const std::list< bidir_solution > &solutions)

Protected Attributes

- I_Tree ∗ **T_a**
- I_Tree ∗ **T_b**

Additional Inherited Members

8.4.1 Detailed Description

Handles the bidirectional extension strategy (Section 1.2.2 of the documentation).

Two trees, A and B, are simultaneously extended within the given iterations, trying to establish a connection between them. A deterministic extension is performed sometimes, which tries to connect the nearest neighbour in tree A, to the root of tree B (or vice-versa). Here the solution consists in the pair of pointers to the nodes in tree A and B that touches themself, establishing a connection between the two trees.

8.4.2 Constructor & Destructor Documentation

8.4.2.1 Bidirectional_Extension_job()

```
MT_RTT::Bidirectional_Extension_job::Bidirectional_Extension_job (
             I_Tree * to_extend_A,
             I_Tree * to_extend_B,
             const float * det_coeff,
             const bool * cumul_sol )  [inline]
```

Constructor.

Parameters

in	*to_extend_A*	the tree A to absorb and extend toward B
in	*to_extend_B*	the tree B to absorb and extend toward A
in	*det_coeff*	same as in I_Extension_job::I_Extension_job
in	*cumul_sol*	same as in I_Extension_job::I_Extension_job

8.4.3 Member Function Documentation

8.4.3.1 Extend_within_iterations()

```
void MT_RTT::Bidirectional_Extension_job::Extend_within_iterations (
             const size_t & Iterations )  [virtual]
```

Perform the specified number of estensions on a wrapped tree(s).

This function may be called multiple times, for performing batch of extensions. If the cumulation of the solution was not enabled, calling this method when a solution was already found raise an exception.

Parameters

in	*Iterations*	the number of extensions to perform

Implements MT_RTT::I_Extension_job< bidir_solution >.

8.4.3.2 Remove_Trees()

```
list< I_Tree * > MT_RTT::Bidirectional_Extension_job::Remove_Trees ( )  [virtual]
```

Move outside of this object all the trees it contains.

If the trees are not removed, they are all destroyed when destroying this object

Parameters

out	*return*	the trees that are removed from this object

Implements MT_RTT::I_Extension_job< bidir_solution >.

The documentation for this class was generated from the following files:

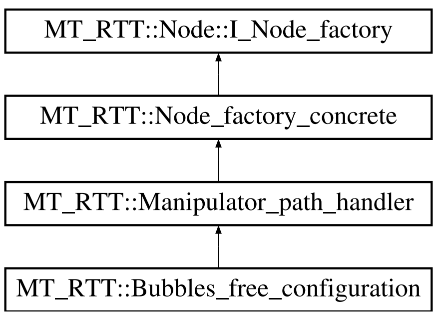
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Extensions.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Extensions.cpp

8.5 MT_RTT::Bubbles_free_configuration Class Reference

In this object, the steering is done considering bubbles of free configuration, Section 2.2.3 of the documentation.

```
#include <Problem_path_basic.h>
```

Inheritance diagram for MT_RTT::Bubbles_free_configuration:



Classes

- class I_Proximity_calculator

    The object in charge of computing the distances between the robots and the obstacles, which are used for computing the bubbles.

Public Member Functions

- I_Proximity_calculator ∗ **Get_proxier** ()
- Bubbles_free_configuration (const float &Gamma, const Node_State &Q_max, const Node_State &Q_min, std::unique_ptr< I_Proximity_calculator > &prox_calc)

    Constructor. The passed prox_calc is absorbed and destroyed when destroying this object.
- Bubbles_free_configuration (const float &Gamma, const float &q_max, const float &q_min, const size_t &dof, std::unique_ptr< I_Proximity_calculator > &prox_calc)

    Constructor.
- virtual std::unique_ptr< I_Node_factory > copy ()

    used for cloning this object: a deep copy must be implemented.
- virtual void Steer (float ∗cost_steered, float ∗steered_state, const float ∗start_state, const float ∗target_state, bool ∗trg_reached)

    The steering procedure is done considering the bubble of free configurations, Section 2.2.3 of the documentation.
- virtual void Cost_to_go_constraints (float ∗result, const float ∗start_state, const float ∗ending_state)

    The collisions along an entire segment are checked by building the bubble of free configuration centered at the start_state. In case the segment connecting the start_state and the ending_state is completely contained in the bubble, no collisions are present.
- void Set_dist_for_accept_steer (const float &value)

    Set the threshold for accepting a steering operation.

Additional Inherited Members

8.5.1 Detailed Description

In this object, the steering is done considering bubbles of free configuration, Section 2.2.3 of the documentation.

An external object, in charge of computing the distance between the robot(s) links and the obstacles is passed and absorbed.

8.5.2 Constructor & Destructor Documentation

8.5.2.1 Bubbles_free_configuration() [1/2]

```
MT_RTT::Bubbles_free_configuration::Bubbles_free_configuration (
            const float & Gamma,
            const Node_State & Q_max,
            const Node_State & Q_min,
            std::unique_ptr< I_Proximity_calculator > & prox_calc )
```

Constructor. The passed prox_calc is absorbed and destroyed when destroying this object.

Parameters

in	*prox_calc*	the object in charge of computing the distances w.r.t to the obstacles and the reciprocal distances of the robots
in	*Gamma*	same meaning as in Manipulator_path_handler::Manipulator_path_handler
in	*Q_max*	same meaning as in Manipulator_path_handler::Manipulator_path_handler
in	*Q_min*	same meaning as in Manipulator_path_handler::Manipulator_path_handler

8.5.2.2 Bubbles_free_configuration() [2/2]

```
MT_RTT::Bubbles_free_configuration::Bubbles_free_configuration (
            const float & Gamma,
            const float & q_max,
            const float & q_min,
            const size_t & dof,
            std::unique_ptr< I_Proximity_calculator > & prox_calc )
```

Constructor.

Similar to Bubbles_free_configuration(const float& Gamma, const Node_State& Q_max, const Node_State& Q_min, std::u but assuming that Q_max (and Q_min) have all the same values equal to q_max and has a size equal to dof.

8.5.3 Member Function Documentation

8.5.3.1 copy()

```
virtual std::unique_ptr<I_Node_factory> MT_RTT::Bubbles_free_configuration::copy ( )  [inline],
[virtual]
```

used for cloning this object: a deep copy must be implemented.

This function is invoked by parallel planners for dispatching copies of this class to the other working threads. In this way, the threads must not be forced to synchronize for accessing the methods of an I_Node_factory. Therefore, when deriving your own factory describing your own problem, be carefull to avoid shallow copies and implement deep copies.

Parameters

out	*return*	a clone of this object

Implements MT_RTT::Node::I_Node_factory.

8.5.3.2 Set_dist_for_accept_steer()

```
void MT_RTT::Bubbles_free_configuration::Set_dist_for_accept_steer (
            const float & value )
```

Set the threshold for accepting a steering operation.

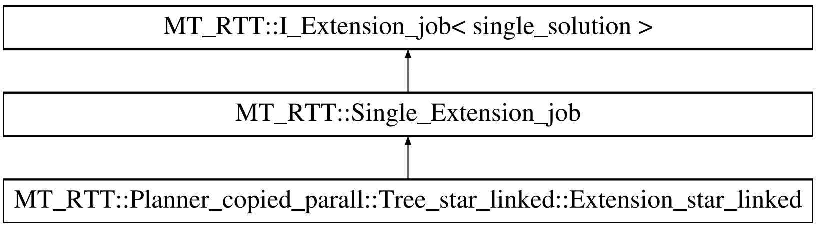
When considering the bubble of free configuration, the steering is always possible, but can lead to obtain a configuration very close to the nearest neighbour (when the robots are close to the obstacles). This function regulates the threashold that is used to decide or not to accept a new steered configuration.

The documentation for this class was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_path_basic.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Problem_path_basic.cpp

8.6 MT_RTT::Planner_copied_parall::Tree_star_linked::Extension_star_linked Class Reference

Inheritance diagram for MT_RTT::Planner_copied_parall::Tree_star_linked::Extension_star_linked:



Public Member Functions

- **Extension_star_linked** (I_Tree *to_extend, const Node_State &target, const float *det_coeff, const bool *cumul_sol)

Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_copied_parall.cpp

8.7 MT_RTT::json_parser::field Struct Reference

The structure describing each node in the array to encode.

```
#include <json.h>
```

Public Attributes

- std::string **name**
- std::vector< std::vector< float > > **values**

8.7.1 Detailed Description

The structure describing each node in the array to encode.

The documentation for this struct was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/json.h

8.8 MT_RTT::Tunneled_check_collision::I_Collision_checker Class Reference

The object in charge of performing the collision check for a single generic configuration.

```
#include <Problem_path_basic.h>
```

Public Member Functions

- virtual std::unique_ptr< I_Collision_checker > copy_checker ()=0

  It is mainly used when copying the Tunneled_check_collision object that contains this checker.
- virtual bool Collision_present (const float *Q_state)=0

  Returns true when a collision is detected for the configuration passed as input.

8.8.1 Detailed Description

The object in charge of performing the collision check for a single generic configuration.

8.8.2 Member Function Documentation

8.8.2.1 Collision_present()

```
virtual bool MT_RTT::Tunneled_check_collision::I_Collision_checker::Collision_present (
            const float * Q_state )  [pure virtual]
```

Returns true when a collision is detected for the configuration passed as input.

Parameters

out	*return*	the result of the collision check detection
in	*Q_state*	the pose to check

8.8.2.2 copy_checker()

```
virtual std::unique_ptr<I_Collision_checker> MT_RTT::Tunneled_check_collision::I_Collision_↩
checker::copy_checker ( )  [pure virtual]
```

It is mainly used when copying the Tunneled_check_collision object that contains this checker.

All the parameters inside the object must be copied, since the copied object will be used by a different thread.

Parameters

out	*return*	a copy of this object inside a smart pointer

The documentation for this class was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_path_basic.h

8.9 MT_RTT::I_Extension_job< Sol_found > Class Template Reference

Interface for handling the extension steps involved in each RRT strategy.

```
#include <Extensions.h>
```

Public Member Functions

- virtual void Extend_within_iterations (const size_t &Iterations)=0
  Perform the specified number of estensions on a wrapped tree(s).
- const bool & Get_solution_was_found ()
  Returns true in case at least a solution was found in the iterations so far done by this extender, false otherwise.
- const size_t & Get_Iterations ()
  Get the extensions so far done.
- virtual std::list< I_Tree ∗ > Remove_Trees ()=0
  Move outside of this object all the trees it contains.
- void Get_best_solution (std::list< Node_State > ∗solution)
  Returns the best cost solution found so far.

Static Public Member Functions

- template< typename Ext >
  static void Get_best_solution (std::list< Node_State > *solution, std::vector< Ext > &battery)

  Similar to Get_best_solution(std::list<Node_State> solution), but taking the best solution among the ones contained in the passed array of extenders.*

Protected Member Functions

- I_Extension_job (const float *det_coeff, const bool *cumul_sol)

  Constructor.
- void **Check_Extension** ()
- virtual void **__Get_best_solution** (std::list< Node_State > *solution, const std::list< Sol_found > &solutions)=0

Static Protected Member Functions

- static const Sol_found * **get_best_solution** (const std::list< Sol_found > &solutions)

Protected Attributes

- bool **A_solution_was_found**
- const bool * **Cumulate_sol**
- const float * **Deterministic_coefficient**
- size_t **Iterations_done**
- std::list< Sol_found > **Solutions_found**

8.9.1 Detailed Description

template<typename Sol_found>
class MT_RTT::I_Extension_job< Sol_found >

Interface for handling the extension steps involved in each RRT strategy.

Solutions eventually found while extending the trees, are saved and stored inside this kind of objects

8.9.2 Constructor & Destructor Documentation

8.9.2.1 I_Extension_job()

```
template<typename Sol_found >
MT_RTT::I_Extension_job< Sol_found >::I_Extension_job (
            const float * det_coeff,
            const bool * cumul_sol ) [inline], [protected]
```

Constructor.

Parameters

in	*det_coeff*	a pointer to the value regulating the probability of a deterministic extension (refer to the parameter \sigma of the algorithms exposed in Sections 1.2.1, 1.2.2 and 1.2.3 of the documentation)
in	*cumul_sol*	a pointer to the boolean that explaining whether to accumulate or not feasible found solutions

8.9.3 Member Function Documentation

8.9.3.1 Extend_within_iterations()

```
template<typename Sol_found >
virtual void MT_RTT::I_Extension_job< Sol_found >::Extend_within_iterations (
            const size_t & Iterations )  [pure virtual]
```

Perform the specified number of estensions on a wrapped tree(s).

This function may be called multiple times, for performing batch of extensions. If the cumulation of the solution was not enabled, calling this method when a solution was already found raise an exception.

Parameters

in	*Iterations*	the number of extensions to perform

Implemented in MT_RTT::Bidirectional_Extension_job, and MT_RTT::Single_Extension_job.

8.9.3.2 Get_best_solution() [1/2]

```
template<typename Sol_found >
void MT_RTT::I_Extension_job< Sol_found >::Get_best_solution (
            std::list< Node_State > * solution )  [inline]
```

Returns the best cost solution found so far.

Parameters

out	*solution*	the list of states representing the solution. Values are copied from the correspoding node states.

8.9.3.3 Get_best_solution() [2/2]

```
template<typename Sol_found >
template<typename Ext >
static void MT_RTT::I_Extension_job< Sol_found >::Get_best_solution (
            std::list< Node_State > * solution,
            std::vector< Ext > & battery ) [inline], [static]
```

Similar to Get_best_solution(std::list<Node_State>∗ solution), but taking the best solution among the ones contained in the passed array of extenders.

Ext should be something deriving from I_Extension_job<Sol_found>.

Parameters

in	*battery*	the array of extender to consider
out	*solution*	the list of states representing the solution. Values are copied from the correspoding node states.

8.9.3.4 Remove_Trees()

```
template<typename Sol_found >
virtual std::list<I_Tree*> MT_RTT::I_Extension_job< Sol_found >::Remove_Trees ( ) [pure
virtual]
```

Move outside of this object all the trees it contains.

If the trees are not removed, they are all destroyed when destroying this object

Parameters

out	*return*	the trees that are removed from this object

Implemented in MT_RTT::Bidirectional_Extension_job, and MT_RTT::Single_Extension_job.

The documentation for this class was generated from the following file:

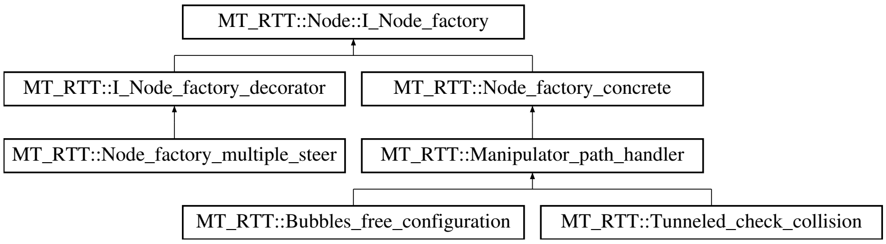
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Extensions.h

8.10 MT_RTT::Node::I_Node_factory Class Reference

Interface for the class describing the particular planning problem to solve.

```
#include <Problem_description.h>
```

Inheritance diagram for MT_RTT::Node::I_Node_factory:



Public Member Functions

- **I_Node_factory** (const I_Node_factory &o)=delete
- I_Node_factory & **operator=** (const I_Node_factory &o)=delete
- Node Random_node ()

  Returns a node having a state randomly sampled in the \mathcal{X} space, Section 1.2.1 of the documentation.
- void Cost_to_go (float ∗result, const Node ∗start, const Node ∗ending_node)

  Evaluates the cost C(\tau), Section 1.2.3 of the documentation, of the trajectory \tau going from the starting node to the ending one, for two nodes not already connected.
- void Cost_to_go_constraints (float ∗result, const Node ∗start, const Node ∗trg)

  Evaluates the constrained cost of the trajectory going from the starting node to the ending one, for two nodes not already connected.
- Node Steer (Node ∗start, const Node ∗trg, bool ∗trg_reached)

  Performs a steering operation, Section 1.2.1 of the documentation, from a staring node to a target one.
- Node Clone_Node (const Node &o)

  Generates a node with the same state and father of the node passed as input.
- virtual const size_t & Get_State_size () const =0

  Returns the cardinality of \mathcal{X}, Section 1.2.1 of the documentation, of the plannig problem handled by this object.
- virtual const float & Get_Gamma () const =0

  Returns the \gamma parameter, Section 1.2.3 of the documentation, regulating the near set size, that RRT∗ versions must compute.
- virtual const bool & Get_symm_flag () const =0

  Returns true in case the planning problem handled by this object is symmetric, i.e. the cost to go from a node A to B is the same of the cost to go from B to A.
- Node New_root (const Node_State &state)

  Builds a new root for a tree.
- virtual std::unique_ptr< I_Node_factory > copy ()=0

  used for cloning this object: a deep copy must be implemented.
- virtual void Random_node (float ∗random_state)=0

  internally called by I_Node_factory::Random_node().
- virtual void Cost_to_go (float ∗result, const float ∗start_state, const float ∗ending_state)=0

  internally called by I_Node_factory::Cost_to_go(float∗ result, const Node∗ start, const Node∗ trg).
- virtual void Cost_to_go_constraints (float ∗result, const float ∗start_state, const float ∗ending_state)=0

  internally called by I_Node_factory::Cost_to_go_constraints(float∗ result, const Node∗ start, const Node∗ trg).
- virtual void Steer (float ∗cost_steered, float ∗steered_state, const float ∗start_state, const float ∗target_state, bool ∗trg_reached)=0

  internally called by I_Node_factory::Steer(Node∗ start, const Node∗ trg, bool∗ trg_reached).

Protected Member Functions

- float ∗ **Alloc_state** ()

8.10.1 Detailed Description

Interface for the class describing the particular planning problem to solve.

It is crucial for addressing step A of the pipeline presented in Section 1.3 of the documentation.

8.10.2 Member Function Documentation

8.10.2.1 Clone_Node()

```
Node MT_RTT::Node::I_Node_factory::Clone_Node (
            const Node & o )
```

Generates a node with the same state and father of the node passed as input.

Parameters

out	*return*	the cloned node (the result is returned by internally using move: it is not copied).
in	*o*	the node to clone

8.10.2.2 copy()

```
virtual std::unique_ptr<I_Node_factory> MT_RTT::Node::I_Node_factory::copy ( ) [pure virtual]
```

used for cloning this object: a deep copy must be implemented.

This function is invoked by parallel planners for dispatching copies of this class to the other working threads. In this way, the threads must not be forced to synchronize for accessing the methods of an I_Node_factory. Therefore, when deriving your own factory describing your own problem, be carefull to avoid shallow copies and implement deep copies.

Parameters

out	*return*	a clone of this object

Implemented in MT_RTT::Node_factory_multiple_steer, MT_RTT::Bubbles_free_configuration, and MT_RTT::Tunneled_check_collision.

8.10.2.3 Cost_to_go() [1/2]

```
virtual void MT_RTT::Node::I_Node_factory::Cost_to_go (
            float * result,
```

```
            const float * start_state,
            const float * ending_state )  [pure virtual]
```

internally called by I_Node_factory::Cost_to_go(float* result, const Node* start, const Node* trg).

This is function is actually in charge of computing C(\tau_{start -> ending_node}), considering the passed start_state and ending_state, which are array of values describing the staring and the ending state to consider.

Implemented in MT_RTT::I_Node_factory_decorator, and MT_RTT::Manipulator_path_handler.

8.10.2.4 Cost_to_go() [2/2]

```
void MT_RTT::Node::I_Node_factory::Cost_to_go (
            float * result,
            const Node * start,
            const Node * ending_node )  [inline]
```

Evaluates the cost C(\tau), Section 1.2.3 of the documentation, of the trajectory \tau going from the starting node to the ending one, for two nodes not already connected.

This cost doesn't account for constraints, but considers only the optimal unconstrained trajectory \tau leading from the starting to the ending node.

Parameters

out	*result*	the computed cost
in	*start*	the starting node in the trajectory whose cost is to evaluate
in	*ending_node*	the ending node in the trajectory whose cost is to evaluate

8.10.2.5 Cost_to_go_constraints() [1/2]

```
virtual void MT_RTT::Node::I_Node_factory::Cost_to_go_constraints (
            float * result,
            const float * start_state,
            const float * ending_state )  [pure virtual]
```

internally called by I_Node_factory::Cost_to_go_constraints(float* result, const Node* start, const Node* trg).

This function is actually in charge of computing max{ C(\tau_{start -> ending_node}) , C_adm}, considering the passed start_state and ending_state, which are array of values describing the staring and the ending state to consider.

Implemented in MT_RTT::I_Node_factory_decorator, MT_RTT::Bubbles_free_configuration, and MT_RTT::Tunneled_check_collision.

8.10.2.6 Cost_to_go_constraints() [2/2]

```
void MT_RTT::Node::I_Node_factory::Cost_to_go_constraints (
            float * result,
            const Node * start,
            const Node * trg ) [inline]
```

Evaluates the constrained cost of the trajectory going from the starting node to the ending one, for two nodes not already connected.

This cost accounts for constraints. In case the constraints are violated along the nominal trajectory going from the starting node to the ending one, a FLT_MAX is returned. Otherwise, the cost returned is the one of the nominal trajectory, i.e. the one computed with I_Node_factory::Cost_to_go.

Parameters

out	*result*	the computed cost
in	*start*	the starting node in the trajectory whose cost is to evaluate
in	*ending_node*	the ending node in the trajectory whose cost is to evaluate

8.10.2.7 New_root()

```
Node MT_RTT::Node::I_Node_factory::New_root (
            const Node_State & state )
```

Builds a new root for a tree.

The root is a node having a NULL father.

Parameters

in	*state*	the state that will be contained in the root to create.
out	*return*	the created root (the result is returned by internally using move: it is not copied).

8.10.2.8 Random_node() [1/2]

```
Node MT_RTT::Node::I_Node_factory::Random_node ( )
```

Returns a node having a state randomly sampled in the \mathcal{X} space, Section 1.2.1 of the documentation.

This function is invoked for randomly growing a searching tree.

Parameters

out	*return*	the random node computed (the result is returned by internally using move: it is not copied).

8.10.2.9 Random_node() [2/2]

```
virtual void MT_RTT::Node::I_Node_factory::Random_node (
            float * random_state )  [pure virtual]
```

internally called by I_Node_factory::Random_node().

The passed random_state is an array of values already with the cardinality of \mathcal{X}, which must be set to random values by this function.

Implemented in MT_RTT::I_Node_factory_decorator, and MT_RTT::Manipulator_path_handler.

8.10.2.10 Steer() [1/2]

```
virtual void MT_RTT::Node::I_Node_factory::Steer (
            float * cost_steered,
            float * steered_state,
            const float * start_state,
            const float * target_state,
            bool * trg_reached )  [pure virtual]
```

internally called by I_Node_factory::Steer(Node∗ start, const Node∗ trg, bool∗ trg_reached).

This function is actually in charge of performing the steering operation, considering the passed start↩_state and target_state, which are array of values describing the starting and target state to consider. cost_steered must be returned equal to NULL in case the steering was not possible.

Implemented in MT_RTT::Node_factory_multiple_steer, MT_RTT::I_Node_factory_decorator, MT_RTT::Bubbles_free_conf and MT_RTT::Tunneled_check_collision.

8.10.2.11 Steer() [2/2]

```
Node MT_RTT::Node::I_Node_factory::Steer (
            Node * start,
            const Node * trg,
            bool * trg_reached )
```

Performs a steering operation, Section 1.2.1 of the documentation, from a staring node to a target one.

The node returned contains the steered state. In case a steering operation is not possible, a Node with a NULL State is returned.

Parameters

out	*return*	the node with the steered configuration (the result is returned by internally using move: it is not copied).
in	*start*	the starting node from which the steer operation must be tried
in	*trg*	the target node to which the steer operation must be tried
out	*trg_reached*	returns true in case the steering was possible and led to reach the target node. Otherwise false is returned.

The documentation for this class was generated from the following files:

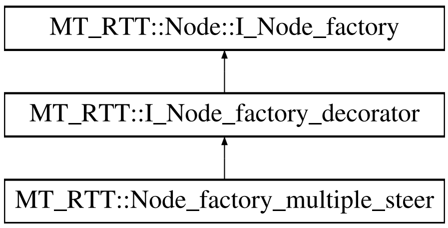
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_description.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Problem_description.cpp

8.11 MT_RTT::I_Node_factory_decorator Class Reference

Interface for the generic I_Node_factory decorator.

```
#include <Problem_description.h>
```

Inheritance diagram for MT_RTT::I_Node_factory_decorator:



Public Member Functions

- I_Node_factory_decorator (std::unique_ptr< I_Node_factory > &to_wrap)

    The I_Node_factory is absorbed and destroyed by ∼I_Node_factory_decorator.
- virtual void Random_node (float ∗random_state)

    internally called by I_Node_factory::Random_node().
- virtual void Cost_to_go (float ∗result, const float ∗start_state, const float ∗ending_state)

    internally called by I_Node_factory::Cost_to_go(float∗ result, const Node∗ start, const Node∗ trg).
- virtual void Cost_to_go_constraints (float ∗result, const float ∗start_state, const float ∗ending_state)

    internally called by I_Node_factory::Cost_to_go_constraints(float∗ result, const Node∗ start, const Node∗ trg).
- virtual void Steer (float ∗cost_steered, float ∗steered_state, const float ∗start_state, const float ∗target_state, bool ∗trg_reached)

    internally called by I_Node_factory::Steer(Node∗ start, const Node∗ trg, bool∗ trg_reached).
- virtual const size_t & Get_State_size () const

    Returns the cardinality of \mathcal{X}, Section 1.2.1 of the documentation, of the plannig problem handled by this object.
- virtual const float & Get_Gamma () const

    Returns the \gamma parameter, Section 1.2.3 of the documentation, regulating the near set size, that RRT∗ versions must compute.
- virtual const bool & Get_symm_flag () const

    Returns true in case the planning problem handled by this object is symmetric, i.e. the cost to go from a node A to B is the same of the cost to go from B to A.
- std::unique_ptr< I_Node_factory > & Get_Wrapped ()

Additional Inherited Members

8.11.1 Detailed Description

Interface for the generic I_Node_factory decorator.

8.11.2 Constructor & Destructor Documentation

8.11.2.1 I_Node_factory_decorator()

```
MT_RTT::I_Node_factory_decorator::I_Node_factory_decorator (
            std::unique_ptr< I_Node_factory > & to_wrap )  [inline]
```

The I_Node_factory is absorbed and destroyed by ∼I_Node_factory_decorator.

Not familiar with the concept of decorator? Check `https://en.wikipedia.org/wiki/↩ Decorator_pattern`.

8.11.3 Member Function Documentation

8.11.3.1 Cost_to_go()

```
virtual void MT_RTT::I_Node_factory_decorator::Cost_to_go (
            float * result,
            const float * start_state,
            const float * ending_state )  [inline], [virtual]
```

internally called by I_Node_factory::Cost_to_go(float∗ result, const Node∗ start, const Node∗ trg).

This is function is actually in charge of computing C(\tau_{start -> ending_node}), considering the passed start_state and ending_state, which are array of values describing the staring and the ending state to consider.

Implements MT_RTT::Node::I_Node_factory.

8.11.3.2 Cost_to_go_constraints()

```
virtual void MT_RTT::I_Node_factory_decorator::Cost_to_go_constraints (
            float * result,
            const float * start_state,
            const float * ending_state )  [inline], [virtual]
```

internally called by I_Node_factory::Cost_to_go_constraints(float* result, const Node* start, const Node* trg).

This function is actually in charge of computing max{ C(\tau_{start -> ending_node}) , C_adm}, considering the passed start_state and ending_state, which are array of values describing the staring and the ending state to consider.

Implements MT_RTT::Node::I_Node_factory.

8.11.3.3 Get_Wrapped()

```
std::unique_ptr<I_Node_factory>& MT_RTT::I_Node_factory_decorator::Get_Wrapped ( )  [inline]
```

Parameters

out	*return*	the contained Node_factory .

8.11.3.4 Random_node()

```
virtual void MT_RTT::I_Node_factory_decorator::Random_node (
            float * random_state )  [inline], [virtual]
```

internally called by I_Node_factory::Random_node().

The passed random_state is an array of values already with the cardinality of \mathcal{X}, which must be set to random values by this function.

Implements MT_RTT::Node::I_Node_factory.

8.11.3.5 Steer()

```
virtual void MT_RTT::I_Node_factory_decorator::Steer (
            float * cost_steered,
            float * steered_state,
            const float * start_state,
            const float * target_state,
            bool * trg_reached )  [inline], [virtual]
```

internally called by I_Node_factory::Steer(Node∗ start, const Node∗ trg, bool∗ trg_reached).

This function is actually in charge of performing the steering operation, considering the passed start↩
_state and target_state, which are array of values describing the starting and target state to consider. cost_steered must be returned equal to NULL in case the steering was not possible.

Implements MT_RTT::Node::I_Node_factory.

Reimplemented in MT_RTT::Node_factory_multiple_steer.

The documentation for this class was generated from the following file:

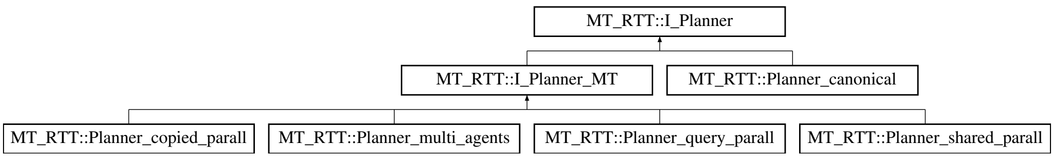
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_description.h

8.12 MT_RTT::I_Planner Class Reference

Interface for a planner.

`#include <Planner.h>`

Inheritance diagram for MT_RTT::I_Planner:



Classes

- struct __last_solution_info

Public Member Functions

- **I_Planner** (const I_Planner &)=delete
- void **operator=** (const I_Planner &)=delete
- void Cumulate_solutions ()

  Use this method to enable the cumulation of the solution also for non RRT∗ versions of the planner.
- void RRT_basic (const Node_State &start, const Node_State &end)

  Tries to solve the problem by executing the basic single tree RRT version (Section 1.2.1 and the Sections contained in Chapter 3 of the documentation) of the solver represented by this object, step C of the pipeline presented in Section 1.3 of the documentation.
- void RRT_bidirectional (const Node_State &start, const Node_State &end)

  Tries to solve the problem by executing the bidirectional RRT version (Section 1.2.2 and the Sections contained in Chapter 3 of the documentation) of the solver represented by this object, step C of the pipeline presented in Section 1.3 of the documentation.
- void RRT_star (const Node_State &start, const Node_State &end)

  Tries to solve the problem by executing the RRT∗ version (Section 1.2.3 and the Sections contained in Chapter 3 of the documentation) of the solver represented by this object, step C of the pipeline presented in Section 1.3 of the documentation.

- size_t Get_Iteration_done ()

    Access the number of iterations performed by the solver for trying to solve the last specified planning problem.
- void Get_solution (std::list< Node_State > *result)

    Access the number of iterations performed by the solver for trying to solve the last specified planning problem, step D.1 of the pipeline presented in Section 1.3 of the documentation.
- std::string Get_Trees_as_JSON ()

    Returns a json structure describing the searching trees computed when solving the last specified planning problem, step D.2 of the pipeline presented in Section 1.3 of the documentation.
- std::string Get_Solution_as_JSON ()

    Append to the passed string a json structure describing the sequence of states representing the last solution found, step D.2 of the pipeline presented in Section 1.3 of the documentation.

Static Public Member Functions

- static std::unique_ptr< I_Planner > Get_canonical (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory *handler)

    Get a canonical solver implementing the standard non parallel versions of the RRT algorith (Sections 1.2.1, 1.2.2 and 1.2.3 of the documentation).
- static std::unique_ptr< I_Planner > Get_query___parall (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory *handler, const size_t &N_threads=0)

    Get the solver for which the query operations are parallelized (Section 3.0.1 of the documentation).
- static std::unique_ptr< I_Planner > Get_shared__parall (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory *handler, const size_t &N_threads=0)

    Get the solver for which many threads synchronize to expand a common tree (Section 3.0.2 of the documentation).
- static std::unique_ptr< I_Planner > Get_copied__parall (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory *handler, const size_t &N_threads=0, const float &reallignement_percentage=0.1f)

    Get the solver for which many threads expands its own copy of the searching tree (Section 3.0.3 of the documentation).
- static std::unique_ptr< I_Planner > Get_multi_ag_parall (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory *handler, const size_t &N_threads=0, const float &reallignement_percentage=0.1f)

    Get the solver implementing the multi agents version of the RRT algorithm (Section 3.0.4 of the documentation).

Protected Member Functions

- I_Planner (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory *handler)

    Constructor.
- virtual void _RRT_basic (const Node_State &start, const Node_State &end)=0

    the method overrided by all the derived planner for performing an RRT search
- virtual void _RRT_bidirectional (const Node_State &start, const Node_State &end)=0

    the method overrided by all the derived planner for performing a bidirectionl search
- virtual void _RRT_star (const Node_State &start, const Node_State &end)=0

    the method overrided by all the derived planner for performing an RRT search*
- void **Set_Solution** (const __last_solution_info &last_sol)

Protected Attributes

- Node::I_Node_factory * **Handler**
- float **Deterministic_coefficient**
- size_t **Iterations_Max**
- bool **Cumulate_sol**

8.12.1 Detailed Description

Interface for a planner.

For solving a planning problem you must first build the solver, using I_Planner::Get_canonical, I_Planner::Get_query___parall, I_Planner::Get_shared__parall, I_Planner::Get_copied__parall or I_Planner::Get_multi_ag_pa Then you can call I_Planner::RRT_basic, I_Planner::RRT_bidirectional or I_Planner::RRT_star of the created solver, passing the starting and ending states of the problem that you want to solve. Finally, you can use I_Planner::Get_solution to get the obtained solution, i.e. the chain of states connecting the staring and the ending nodes, compliant with the constraints. In case a solution was not found, an empty path is returned. Another planning process can be invoked using the same object. In this case, the info about the last planning are detroyed and replaced with the new one.

8.12.2 Constructor & Destructor Documentation

8.12.2.1 I_Planner()

```
MT_RTT::I_Planner::I_Planner (
            const float & det_coeff,
            const size_t & max_iter,
            Node::I_Node_factory * handler ) [protected]
```

Constructor.

Parameters

in	*det_coeff*	same meaning as in I_Planner::Get_canonical
in	*max_iter*	same meaning as in I_Planner::Get_canonical
in	*handler*	same meaning as in I_Planner::Get_canonical

8.12.3 Member Function Documentation

8.12.3.1 Cumulate_solutions()

```
void MT_RTT::I_Planner::Cumulate_solutions ( ) [inline]
```

Use this method to enable the cumulation of the solution also for non RRT∗ versions of the planner.

In case the cumulation is enabled, the search is not arrested when a first solution is found, but is kept active till the maximum number of iterations is reached. All the solutions found are stored and the best one is selected at the end. This solution will be the one externally accesible.

8.12.3.2 Get_canonical()

```
unique_ptr< I_Planner > MT_RTT::I_Planner::Get_canonical (
            const float & det_coeff,
            const size_t & max_iter,
            Node::I_Node_factory * handler )  [static]
```

Get a canonical solver implementing the standard non parallel versions of the RRT algorith (Sections 1.2.1, 1.2.2 and 1.2.3 of the documentation).

The creation of this kind of solver addresses step B of the pipeline presented in Section 1.3 of the documentation.

Parameters

out	*return*	the solver to use later
in	*det_coeff*	the percentage of times for which a deterministic extension must be tried to get a solution (the parameter \sigma of the algorithms exposed in Sections 1.2.1, 1.2.2 and 1.2.3 of the documentation)
in	*max_iter*	the maximum number of iterations to consider when soving the planning problems
in	*handler*	the object describing the planning problem to solve

8.12.3.3 Get_copied__parall()

```
unique_ptr< I_Planner > MT_RTT::I_Planner::Get_copied__parall (
            const float & det_coeff,
            const size_t & max_iter,
            Node::I_Node_factory * handler,
            const size_t & N_threads = 0,
            const float & reallignement_percentage = 0.1f )  [static]
```

Get the solver for which many threads expands its own copy of the searching tree (Section 3.0.3 of the documentation).

The creation of this kind of solver addresses step B of the pipeline presented in Section 1.3 of the documentation.

Parameters

out	*return*	the solver to use later
in	*det_coeff*	same meaning as in I_Planner::Get_canonical
in	*max_iter*	same meaning as in I_Planner::Get_canonical
in	*handler*	same meaning as in I_Planner::Get_canonical
in	*N_threads*	same meaning as in I_Planner::Get_query___parall
in	*reallignement_percentage*	the size of the explorative batches (percentage w.r.t. max_iter), see Section 3.0.3 of the documentation.

8.12.3.4 Get_Iteration_done()

```
size_t MT_RTT::I_Planner::Get_Iteration_done ( )
```

Access the number of iterations performed by the solver for trying to solve the last specified planning problem.

Result is 0 in case no problems were solved at the time of invoking this function.

Parameters

out	*return*	the number of iterations done

8.12.3.5 Get_multi_ag_parall()

```
unique_ptr< I_Planner > MT_RTT::I_Planner::Get_multi_ag_parall (
            const float & det_coeff,
            const size_t & max_iter,
            Node::I_Node_factory * handler,
            const size_t & N_threads = 0,
            const float & reallignement_percentage = 0.1f )  [static]
```

Get the solver implementing the multi agents version of the RRT algorithm (Section 3.0.4 of the documentation).

The creation of this kind of solver addresses step B of the pipeline presented in Section 1.3 of the documentation.

Parameters

out	*return*	the solver to use later
in	*det_coeff*	same meaning as in I_Planner::Get_canonical
in	*max_iter*	same meaning as in I_Planner::Get_canonical
in	*handler*	same meaning as in I_Planner::Get_canonical
in	*N_threads*	same meaning as in I_Planner::Get_query___parall
in	*reallignement_percentage*	similar meaning as in I_Planner::Get_copied__parall, see Section 3.0.4 of the documentation.

8.12.3.6 Get_query___parall()

```
unique_ptr< I_Planner > MT_RTT::I_Planner::Get_query___parall (
            const float & det_coeff,
            const size_t & max_iter,
            Node::I_Node_factory * handler,
            const size_t & N_threads = 0 )  [static]
```

Get the solver for which the query operations are parallelized (Section 3.0.1 of the documentation).

The creation of this kind of solver addresses step B of the pipeline presented in Section 1.3 of the documentation.

Parameters

out	*return*	the solver to use later
in	*det_coeff*	same meaning as in I_Planner::Get_canonical
in	*max_iter*	same meaning as in I_Planner::Get_canonical
in	*handler*	same meaning as in I_Planner::Get_canonical
in	*N_threads*	the number of threads to use. When passing 0, the maximal number admitted by this machine is used.

8.12.3.7 Get_shared__parall()

```
unique_ptr< I_Planner > MT_RTT::I_Planner::Get_shared__parall (
            const float & det_coeff,
            const size_t & max_iter,
            Node::I_Node_factory * handler,
            const size_t & N_threads = 0 )  [static]
```

Get the solver for which many threads synchronize to expand a common tree (Section 3.0.2 of the documentation).

The creation of this kind of solver addresses step B of the pipeline presented in Section 1.3 of the documentation.

Parameters

out	*return*	the solver to use later
in	*det_coeff*	same meaning as in I_Planner::Get_canonical
in	*max_iter*	same meaning as in I_Planner::Get_canonical
in	*handler*	same meaning as in I_Planner::Get_canonical
in	*N_threads*	same meaning as in I_Planner::Get_query___parall

8.12.3.8 Get_solution()

```
void MT_RTT::I_Planner::Get_solution (
            std::list< Node_State > * result )
```

Access the number of iterations performed by the solver for trying to solve the last specified planning problem, step D.1 of the pipeline presented in Section 1.3 of the documentation.

An empty list is returned in case the solution was not found.

Parameters

out	*return*	the number of iterations done

8.12.3.9 Get_Solution_as_JSON()

```
string MT_RTT::I_Planner::Get_Solution_as_JSON ( )
```

Append to the passed string a json structure describing the sequence of states representing the last solution found, step D.2 of the pipeline presented in Section 1.3 of the documentation.

An empty structure is returned in case no problems were solved at the time of invoking this function.

Parameters

out	*return*	a json structure with the waypoints representing the solution (move is internally called when returning the result).

8.12.3.10 Get_Trees_as_JSON()

```
string MT_RTT::I_Planner::Get_Trees_as_JSON ( )
```

Returns a json structure describing the searching trees computed when solving the last specified planning problem, step D.2 of the pipeline presented in Section 1.3 of the documentation.

An empty structure is returned in case no problems were solved at the time of invoking this function.

Parameters

out	*return*	a json structure describing the searching tree computed to solve the last problem (move is internally called when returning the result).

8.12.3.11 RRT_basic()

```
void MT_RTT::I_Planner::RRT_basic (
            const Node_State & start,
            const Node_State & end )
```

Tries to solve the problem by executing the basic single tree RRT version (Section 1.2.1 and the Sections contained in Chapter 3 of the documentation) of the solver represented by this object, step C of the pipeline presented in Section 1.3 of the documentation.

The solution found is internally stored, as well as the computed searching tree. The data about the solutions of any previous problem solved are deleted.

Parameters

in	*start*	the staring state of the problem to solve
in	*end*	the ending state of the problem to solve

8.12.3.12 RRT_bidirectional()

```
void MT_RTT::I_Planner::RRT_bidirectional (
            const Node_State & start,
            const Node_State & end )
```

Tries to solve the problem by executing the bidirectional RRT version (Section 1.2.2 and the Sections contained in Chapter 3 of the documentation) of the solver represented by this object, step C of the pipeline presented in Section 1.3 of the documentation.

The solution found is internally stored, as well as the computed searching trees. The data about the solutions of any previous problem solved are deleted. This planning strategy cannot be adopted for non symmetric problem, see also Node::I_Node_factory::Get_symm_flag

Parameters

in	*start*	the staring state of the problem to solve
in	*end*	the ending state of the problem to solve

8.12.3.13 RRT_star()

```
void MT_RTT::I_Planner::RRT_star (
            const Node_State & start,
            const Node_State & end )
```

Tries to solve the problem by executing the RRT$*$ version (Section 1.2.3 and the Sections contained in Chapter 3 of the documentation) of the solver represented by this object, step C of the pipeline presented in Section 1.3 of the documentation.

The solution found is internally stored, as well as the computed searching trees. The data about the solutions of any previous problem solved are deleted. When invoking this function the cumulation of the solutions is aitomatically enabled.

Parameters

in	*start*	the staring state of the problem to solve
in	*end*	the ending state of the problem to solve

The documentation for this class was generated from the following files:

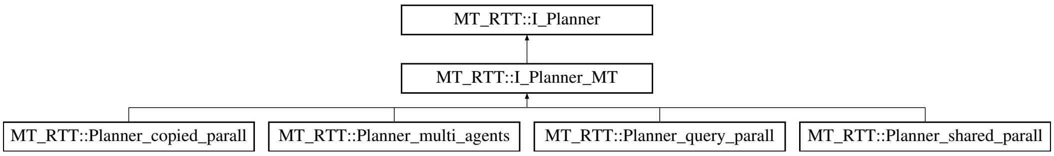
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Planner.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner.cpp
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_canonical.cpp
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_copied_parall.cpp
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_multi_agents.cpp
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_query_parall.cpp
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_shared_parall.cpp

8.13 MT_RTT::I_Planner_MT Class Reference

This class contains common functionalities that every multi-threaded planner may use.

```
#include <Planner_MT.h>
```

Inheritance diagram for MT_RTT::I_Planner_MT:



Public Member Functions

- const size_t & get_Threads ()
- void set_Threads (const size_t &Threads)

Protected Member Functions

- **I_Planner_MT** (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory ∗handler, const size_t &N_threads)
- void Init_Single_Extension_battery (std::vector< Single_Extension_job > ∗battery, const std::vector< I_Tree ∗ > &T, const Node_State &target)

  This method initializes a battery of single tree extenders (see Single_Extension_job).
- void Init_Bidirectional_Extension_battery (std::vector< Bidirectional_Extension_job > ∗battery, const std::vector< I_Tree ∗ > &A, const std::vector< I_Tree ∗ > &B)

  This method initializes a battery of bidirectional extenders (see Bidirectional_Extension_job).
- std::vector< unsigned int > random_seeds (const size_t &N_seeds)

  Get seeds for initialize rand in a different way in all the used threads.

Static Protected Member Functions

- template< typename T >
  static std::vector< I_Tree ∗ > **cast_to_I_Tree** (const std::vector< T ∗ > &to_cast)

Additional Inherited Members

8.13.1 Detailed Description

This class contains common functionalities that every multi-threaded planner may use.

8.13.2 Member Function Documentation

8.13.2.1 get_Threads()

```
const size_t& MT_RTT::I_Planner_MT::get_Threads ( )  [inline]
```

Parameters

out	*return*	the number of threads considered by this multi-threaded solver

8.13.2.2 Init_Bidirectional_Extension_battery()

```
void MT_RTT::I_Planner_MT::Init_Bidirectional_Extension_battery (
            std::vector< Bidirectional_Extension_job > * battery,
            const std::vector< I_Tree * > & A,
            const std::vector< I_Tree * > & B )  [protected]
```

This method initializes a battery of bidirectional extenders (see Bidirectional_Extension_job).

Each of the element in the battery will be manipulated by a single thread. Therefore, the elements in the battery will be concurrently extended.

8.13.2.3 Init_Single_Extension_battery()

```
void MT_RTT::I_Planner_MT::Init_Single_Extension_battery (
            std::vector< Single_Extension_job > * battery,
            const std::vector< I_Tree * > & T,
            const Node_State & target )  [protected]
```

This method initializes a battery of single tree extenders (see Single_Extension_job).

Each of the element in the battery will be manipulated by a single thread. Therefore, the elements in the battery will be concurrently extended.

8.13.2.4 random_seeds()

```
std::vector< unsigned int > MT_RTT::I_Planner_MT::random_seeds (
            const size_t & N_seeds )  [protected]
```

Get seeds for initialize rand in a different way in all the used threads.

Seeds are initialized considering the actual time.

Parameters

out	*return*	the computed seeds
in	*N_seeds*	the number of seeds to compute

8.13.2.5 set_Threads()

```
void MT_RTT::I_Planner_MT::set_Threads (
            const size_t & Threads )
```

Parameters

in	*Threads*	set the number of threads to use for solving future problems.

The documentation for this class was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Planner_MT.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_MT.cpp

8.14 MT_RTT::Bubbles_free_configuration::I_Proximity_calculator Class Reference

The object in charge of computing the distances between the robots and the obstacles, which are used for computing the bubbles.

`#include <Problem_path_basic.h>`

Classes

- struct single_robot_prox

  The distances concerning a single robot.

Public Member Functions

- virtual std::unique_ptr< I_Proximity_calculator > copy_calculator ()=0

  It is mainly used when copying the Bubbles_free_configuration object that contains this calculator.
- virtual void Recompute_Proximity_Info (const float *Q_state)=0

  The information about the distances between the robots and the obstacles are recomputed considering the new passed pose.
- const std::vector< single_robot_prox > & Get_single_info () const

  Get the last computed distances w.r.t the obstacles and robot.
- const std::vector< float > & Get_distances_pairs () const

  Get the last reciprocal computed distances between the robots.

Protected Member Functions

- **I_Proximity_calculator** (const std::vector< size_t > &Dof)

Protected Attributes

- std::vector< single_robot_prox > **Robots_info**
- std::vector< float > **Robot_distance_pairs**

8.14.1 Detailed Description

The object in charge of computing the distances between the robots and the obstacles, which are used for computing the bubbles.

8.14.2 Member Function Documentation

8.14.2.1 copy_calculator()

```
virtual std::unique_ptr<I_Proximity_calculator> MT_RTT::Bubbles_free_configuration::I_Proximity↩
_calculator::copy_calculator ( ) [pure virtual]
```

It is mainly used when copying the Bubbles_free_configuration object that contains this calculator.

All the parameters inside the object must be copied, since the copied object will be used by a different thread.

Parameters

out	*return*	a copy of this object

8.14.2.2 Recompute_Proximity_Info()

```
virtual void MT_RTT::Bubbles_free_configuration::I_Proximity_calculator::Recompute_Proximity↩
_Info (
            const float * Q_state ) [pure virtual]
```

The information about the distances between the robots and the obstacles are recomputed considering the new passed pose.

Parameters

in	*Q_state*	the new pose to consider for updating the distances values

The documentation for this class was generated from the following files:

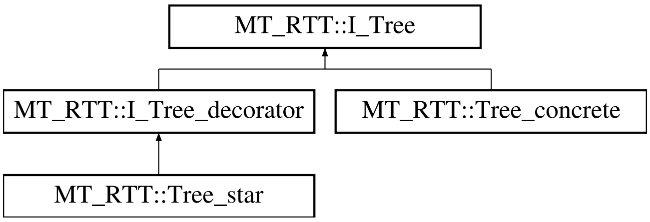
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_path_basic.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Problem_path_basic.cpp

8.15 MT_RTT::I_Tree Class Reference

Interface for handling a tree involved in a RRT strategy.

```
#include <Tree.h>
```

Inheritance diagram for MT_RTT::I_Tree:



Classes

- struct _extend_info

Public Member Functions

- **I_Tree** (const I_Tree &o)=delete
- void **operator=** (const I_Tree &o)=delete
- const Node * Extend_random ()

  Tries to extend the tree toward a random configuration that is internally sampled. \detials In case an extensio was possible, the node added to the tree is returned. Otherwise, NULL is returned.
- const Node * Extend_deterministic (const Node *target)

  An extension toward the passed target node is tried.
- std::string Get_Tree_as_JSON ()

  Create a json describing the nodes contained in this tree.
- virtual Node::I_Node_factory * Get_Problem_Handler ()=0

  Get the object describing the planning problems this tree refers to.
- const Node * Get_root ()

  Get the root of the tree.
- const bool & Get_target_reached_flag ()

  Get the reached target flag.

Protected Member Functions

- virtual Node * **Extend** (const Node *target)=0
- virtual std::list< Node * > * **Get_Nodes** ()=0
- virtual _extend_info * **Get_extend_info** ()=0
- bool **Extend_reached_determ_target** ()

Static Protected Member Functions

- static Node * **Extend_o** (I_Tree *o, const Node *target)
- static std::list< Node * > * **Get_Nodes_o** (I_Tree *tree)
- static _extend_info * **Get_extend_info_o** (I_Tree *o)

8.15.1 Detailed Description

Interface for handling a tree involved in a RRT strategy.

8.15.2 Member Function Documentation

8.15.2.1 Extend_deterministic()

```
const Node * MT_RTT::I_Tree::Extend_deterministic (
            const Node * target )
```

An extension toward the passed target node is tried.

In case the extension succeeds, a new node with the steered configuration, Section 1.2 of the documentation, is automatically inserted to the list of nodes contained in this tree and returned. On the opposite, when the extension was not possible a NULL value is returned.

Parameters

in	*target*	the target node toward which the extension must be tried
out	*return*	the node added to the tree as a consequence of the extension (NULL is returned in case the extension was not possible).

8.15.2.2 Extend_random()

```
const Node * MT_RTT::I_Tree::Extend_random ( )
```

Tries to extend the tree toward a random configuration that is internally sampled. \detials In case an extensio was possible, the node added to the tree is returned. Otherwise, NULL is returned.

Parameters

out	*return*	the node added to the tree as a consequence of the extension (NULL is returned in case the extension was not possible).

8.15.2.3 Get_Problem_Handler()

```
virtual Node::I_Node_factory* MT_RTT::I_Tree::Get_Problem_Handler ( ) [pure virtual]
```

Get the object describing the planning problems this tree refers to.

Parameters

out	*return*	the object describing the planning problem

Implemented in MT_RTT::I_Tree_decorator, and MT_RTT::Tree_concrete.

8.15.2.4 Get_target_reached_flag()

```
const bool& MT_RTT::I_Tree::Get_target_reached_flag ( ) [inline]
```

Get the reached target flag.

The reach target flag describes whether the previous extension tried (using Extend_random or Extend_↩ deterministic) suceeded in reaching the target or not. More formally, when a kind of extension is tried for the tree, this quantity is internally set equal to true only in the case that: A) the extension was possible B) the steered configuration is equal to the target one, i.e. the extension leads to reach the target It is set to false in all other cases.

Parameters

out	*return*	the reach target flag

8.15.2.5 Get_Tree_as_JSON()

```
string MT_RTT::I_Tree::Get_Tree_as_JSON ( )
```

Create a json describing the nodes contained in this tree.

The json produced is an array of node, each having an array describing the state of the i^th node and an array describing the state of its parent.

Parameters

out	*return*	the json structure describing the tree (the result is returned by internally using move: it is not copied).

The documentation for this class was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Tree.h

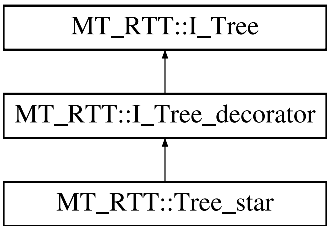
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Tree.cpp

8.16 MT_RTT::I_Tree_decorator Class Reference

A decorator may contain another decorator or a concrete Tree.

`#include <Tree.h>`

Inheritance diagram for MT_RTT::I_Tree_decorator:



Public Member Functions

- I_Tree_decorator (I_Tree *to_wrap, const bool &destroy_wrap)
  Constructor.
- virtual Node::I_Node_factory * Get_Problem_Handler ()
  Get the object describing the planning problems this tree refers to.

Protected Member Functions

- I_Tree * **Get_Wrapped** ()
- virtual Node * **Extend** (const Node *target)
- virtual std::list< Node * > * **Get_Nodes** ()
- virtual _extend_info * **Get_extend_info** ()

Additional Inherited Members

8.16.1 Detailed Description

A decorator may contain another decorator or a concrete Tree.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 I_Tree_decorator()

```
MT_RTT::I_Tree_decorator::I_Tree_decorator (
            I_Tree * to_wrap,
            const bool & destroy_wrap ) [inline]
```

Constructor.

Parameters

in	*to_wrap*	the object to wrap inside this object
in	*destroy_wrap*	when passed true, to_wrap is deleted when deleting this object

8.16.3 Member Function Documentation

8.16.3.1 Get_Problem_Handler()

```
virtual Node::I_Node_factory* MT_RTT::I_Tree_decorator::Get_Problem_Handler ( ) [inline],
[virtual]
```

Get the object describing the planning problems this tree refers to.

Parameters

out	*return*	the object describing the planning problem

Implements MT_RTT::I_Tree.

The documentation for this class was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Tree.h

8.17 MT_RTT::json_parser Class Reference

This class can be used to decode or encode a simple category of json structures.

```
#include <json.h>
```

Classes

- struct field

  The structure describing each node in the array to encode.

Static Public Member Functions

- static std::vector< field > parse_JSON (const std::string &JSON)

  Performs decoding from an input string.
- static std::string get_JSON_from_file (const std::string &file_location)

  Read the json contained in a specified file. The returned result can be later parsed using parse_JSON.
- static const std::vector< std::vector< float > > * get_field (const std::vector< field > &fields, const std::string &name_to_search)

  Get a specific field with a specific name.
- static std::string load_JSON (const std::vector< field > &fields)

  Performs encoding.
- static std::string load_JSON (const float *buffer, const size_t &Size)

  Performs encoding of a single numerical array.
- static std::string load_JSON (const std::vector< std::vector< float >> &values)

  Performs encoding of a single matrix of numbers (each row can have a different size).
- static std::string load_JSON (const float *buffer, const std::vector< size_t > &Sizes)

  Similar to json_parser::load_JSON(const std::vector<std::vector<float>>& values), but passing a single buffer, whose row subdivision is expressed by Sizes.

8.17.1 Detailed Description

This class can be used to decode or encode a simple category of json structures.

It is not possible to handle a generic json. Indeed, the structure to parse must be an array of node, each having a name and a matrix of values. The information describing each node are contained in json_parser::field.

8.17.2 Member Function Documentation

8.17.2.1 get_field()

```
const std::vector< std::vector< float > > * MT_RTT::json_parser::get_field (
            const std::vector< field > & fields,
            const std::string & name_to_search )  [static]
```

Get a specific field with a specific name.

In case the specified field does not exists, NULL is returned.

Parameters

in	*fields*	the array of nodes that contain the field to search
in	*name_to_search*	the name of the node to return
out	*return*	the values associated to the node having the specified name.

8.17.2.2 get_JSON_from_file()

```
std::string MT_RTT::json_parser::get_JSON_from_file (
            const std::string & file_location )  [static]
```

Read the json contained in a specified file. The returned result can be later parsed using parse_JSON.

Parameters

in	*file_location*	the file to read
out	*return*	the content of the file to read (the result is returned by internally using move: they are not copied).

8.17.2.3 load_JSON() [1/4]

```
std::string MT_RTT::json_parser::load_JSON (
            const float * buffer,
            const size_t & Size )  [static]
```

Performs encoding of a single numerical array.

Parameters

in	*buffer*	the buffer of numbers to encode
in	*Size*	the size of the buffer of numbers to encode
out	*return*	the encoded json (the result is returned by internally using move: they are not copied).

8.17.2.4 load_JSON() [2/4]

```
std::string MT_RTT::json_parser::load_JSON (
            const float * buffer,
            const std::vector< size_t > & Sizes )  [static]
```

Similar to json_parser::load_JSON(const std::vector<std::vector<float>>& values), but passing a single buffer, whose row subdivision is expressed by Sizes.

Parameters

in	*buffer*	the matrix of numbers to encode
in	*Sizes*	the size of each row
out	*return*	the encoded json (the result is returned by internally using move: it is not copied).

8.17.2.5 load_JSON() [3/4]

```
string MT_RTT::json_parser::load_JSON (
            const std::vector< field > & fields )  [static]
```

Performs encoding.

Parameters

in	*fields*	the data to encode
out	*return*	the encoded json as a string (the result is returned by internally using move: it is not copied).

8.17.2.6 load_JSON() [4/4]

```
std::string MT_RTT::json_parser::load_JSON (
            const std::vector< std::vector< float >> & values )  [static]
```

Performs encoding of a single matrix of numbers (each row can have a different size).

Parameters

in	*values*	the matrix of numbers to encode
out	*return*	the encoded json (the result is returned by internally using move: it is not copied).

8.17.2.7 parse_JSON()

```
vector< json_parser::field > MT_RTT::json_parser::parse_JSON (
            const std::string & JSON )  [static]
```

Performs decoding from an input string.

Parameters

in	*JSON*	the string to decode
out	*return*	the parsed array of nodes (the result is returned by internally using move: it is not copied).

The documentation for this class was generated from the following files:

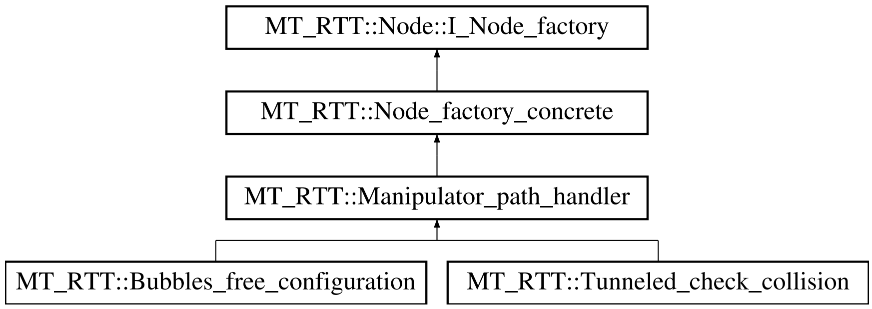
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/json.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/json.cpp

8.18 MT_RTT::Manipulator_path_handler Class Reference

Interface for handling classical path planning problem of a single or a group of articulated fixed robots (Section 2.2 of the documentation).

```
#include <Problem_path_basic.h>
```

Inheritance diagram for MT_RTT::Manipulator_path_handler:



Public Member Functions

- virtual void Random_node (float ∗random_state)
  The hypercube delimited by the maximal and minimum possible joint excursions is sampled.
- virtual void Cost_to_go (float ∗result, const float ∗start_state, const float ∗ending_state)
  The cost to go unconstrained is simply the euclidean distance (in the configurational space) between the two poses.

Protected Member Functions

- Manipulator_path_handler (const float &Gamma, const float ∗Q_max, const float ∗Q_min, const size_t &Q_size)
  Constructor.
- Manipulator_path_handler (const float &Gamma, const std::vector< float > &Q_max, const std↩ ::vector< float > &Q_min)
  Constructor.
- const float ∗ **Get_max** () const
- const float ∗ **Get_min** () const

8.18.1 Detailed Description

Interface for handling classical path planning problem of a single or a group of articulated fixed robots (Section 2.2 of the documentation).

The steering configurations lie always along a segment in the configurational space connecting the nearest neighbour to the target pose. When considering multi-robot problems, the state Q of a single node in a tree is a buffer containing the poses of all the robots in a single array as follows: Q = [Q1^T, Q2^T, ..., Qn^T], where Qi is the pose of the i-th robot.

8.18.2 Constructor & Destructor Documentation

8.18.2.1 Manipulator_path_handler() [1/2]

```
MT_RTT::Manipulator_path_handler::Manipulator_path_handler (
            const float & Gamma,
            const float * Q_max,
            const float * Q_min,
            const size_t & Q_size )  [protected]
```

Constructor.

Q_min and Q_max are the joint limits, i.e. any pose must be Q_min < Q < Q_max (for each joint). A consistency check is internally done to ensure that for each joint i: Q_max[i] >= Q_min[i]. In case multiple robots are part of the cell, each Q_max and Q_min are buffers composed as follows: Q_max = [Q_max1^T, Q_max2^T, ..., Q_maxn^T], Q_min = [Q_min1^T, Q_min2^T, ..., Q_minn^T]

Parameters

in	*Gamma*	the parameter regulating the near set size (see Node::I_Node_factory::Get_Gamma)
in	*Q_max*	the maximal values allowed for each joint of the robot(s)
in	*Q_min*	the minimum values allowed for each joint of the robot(s)

8.18.2.2 Manipulator_path_handler() [2/2]

```
MT_RTT::Manipulator_path_handler::Manipulator_path_handler (
            const float & Gamma,
            const std::vector< float > & Q_max,
            const std::vector< float > & Q_min )  [inline], [protected]
```

Constructor.

Similar to Manipulator_path_handler(const float& Gamma, const float* Q_max, const float* Q_min, const size_t& Q_size) but passing some vectors insted of row buffers

The documentation for this class was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_path_basic.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Problem_path_basic.cpp

8.19 MT_RTT::Planner_query_parall::Tree_master::Nearest_Neighbour_Query Class Reference

Inherits MT_RTT::Planner_query_parall::I_Query.

Public Member Functions

- **Nearest_Neighbour_Query** (Tree_concrete ∗tree, Node::I_Node_factory ∗problem)

Public Attributes

- const Node ∗ **target**
- Node ∗ **nearest**
- float **nearest_cost**

The documentation for this class was generated from the following file:

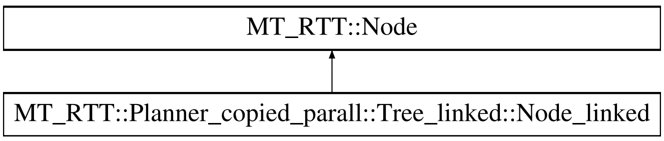
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_query_parall.cpp

8.20 MT_RTT::Node Class Reference

Used internally by a tree (see Tree.h) for representing a state x \in \underline{\mathcal{X}}, Section 1.2.1 of the documentation.

`#include <Problem_description.h>`

Inheritance diagram for MT_RTT::Node:



Classes

- class I_Node_factory

  Interface for the class describing the particular planning problem to solve.

Public Member Functions

- Node (Node &&o)

  Moving constructor.
- **Node** (const Node &)=delete
- Node & **operator=** (const Node &)=delete
- Node & **operator=** (Node &&o)=delete
- void Cost_to_root (float ∗result) const

  Computes the cost to go from the root to this node.
- void Cost_to_root (float ∗result, const size_t &l_max) const

  Similar to Node::Cost_to_root(float∗ result), but throwing an exception when the length of the path that must be followed to reach the root is higher than l_max.
- const float & Get_Cost_from_father () const

  Computes the cost to go from the father of this node to this node.
- const float ∗ Get_State () const

  Returns the state represented by this node.
- Node ∗ Get_Father () const

  Returns the father of this node.
- void Set_Father (Node ∗new_father, const float &cost_from_father)

  Connect this node to the new one passed as input.

Protected Member Functions

- **Node** (Node *father, const float &cost, float *state)
- **Node** (float *state)

8.20.1 Detailed Description

Used internally by a tree (see Tree.h) for representing a state x \in \underline{\mathcal{X}}, Section 1.2.1 of the documentation.

8.20.2 Constructor & Destructor Documentation

8.20.2.1 Node()

```
MT_RTT::Node::Node (
            Node && o )
```

Moving constructor.

State of o is put to NULL, before trasferring the pointer to the Node to build.

8.20.3 Member Function Documentation

8.20.3.1 Cost_to_root() [1/2]

```
void MT_RTT::Node::Cost_to_root (
            float * result ) const
```

Computes the cost to go from the root to this node.

Parameters

out	*result*	the cost to go to compute.

8.20.3.2 Cost_to_root() [2/2]

```
void MT_RTT::Node::Cost_to_root (
            float * result,
            const size_t & I_max ) const
```

Similar to Node::Cost_to_root(float* result), but throwing an exception when the length of the path that must be followed to reach the root is higher than l_max.

This method is called by Tree_star to detect the existance of loop in the tree. It is never used when _R↩
EW_DEBUG (check Tree.h) is not activated

8.20.3.3 Get_Cost_from_father()

```
const float& MT_RTT::Node::Get_Cost_from_father ( ) const [inline]
```

Computes the cost to go from the father of this node to this node.

Parameters

out	*return*	the cost to go to return.

8.20.3.4 Get_Father()

```
Node* MT_RTT::Node::Get_Father ( ) const [inline]
```

Returns the father of this node.

Each node has a single father and can be the father of many nodes.

Parameters

out	*return*	the father node of this node.

8.20.3.5 Get_State()

```
const float* MT_RTT::Node::Get_State ( ) const [inline]
```

Returns the state represented by this node.

Parameters

out	*return*	the address of the first value of the array representing the state.

8.20.3.6 Set_Father()

```
void MT_RTT::Node::Set_Father (
```

```
Node * new_father,
const float & cost_from_father )  [inline]
```

Connect this node to the new one passed as input.

Each node has a single father and can be the father of many nodes.

Parameters

in	*new_father*	the node to assume as new father
in	*cost_from_father*	the cost to go from the new father

The documentation for this class was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_description.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Problem_description.cpp

8.21 MT_RTT::Tree_star::Node2Node_Traj Struct Reference

This structure store the information regarding a possible rewird, i.e. a trajectory going from a starting node to an ending one, which are not already connected. cost is the cost to go from start to end.

```
#include <Tree.h>
```

Public Attributes

- Node * **start**
- Node * **end**
- float **cost**

8.21.1 Detailed Description

This structure store the information regarding a possible rewird, i.e. a trajectory going from a starting node to an ending one, which are not already connected. cost is the cost to go from start to end.

The documentation for this struct was generated from the following file:

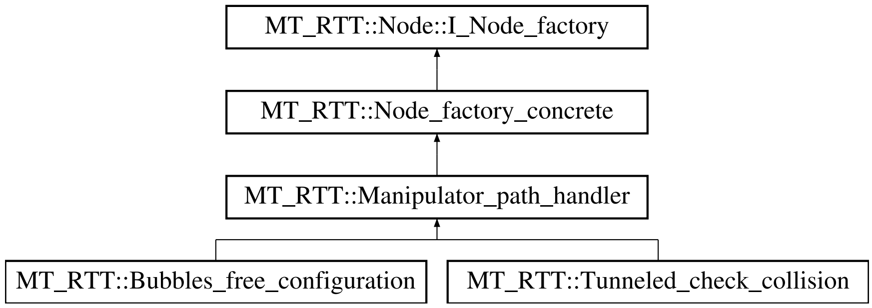
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Tree.h

8.22 MT_RTT::Node_factory_concrete Class Reference

Each handler describing a real planning problem must be derived from this class.

```
#include <Problem_description.h>
```

Inheritance diagram for MT_RTT::Node_factory_concrete:



Public Member Functions

- const size_t & Get_State_size () const

  Returns the cardinality of \mathcal{X}, Section 1.2.1 of the documentation, of the plannig problem handled by this object.
- const float & Get_Gamma () const

  Returns the \gamma parameter, Section 1.2.3 of the documentation, regulating the near set size, that RRT∗ versions must compute.
- const bool & Get_symm_flag () const

  Returns true in case the planning problem handled by this object is symmetric, i.e. the cost to go from a node A to B is the same of the cost to go from B to A.

Protected Member Functions

- Node_factory_concrete (const size_t &X_size, const float &gamma, const bool &traj_symm_flag)

  Constructor of a new concrete problem.

8.22.1 Detailed Description

Each handler describing a real planning problem must be derived from this class.

8.22.2 Constructor & Destructor Documentation

8.22.2.1 Node_factory_concrete()

```
MT_RTT::Node_factory_concrete::Node_factory_concrete (
            const size_t & X_size,
            const float & gamma,
            const bool & traj_symm_flag )  [inline], [protected]
```

Constructor of a new concrete problem.

Parameters

in	*X_size*	the cardinality of the space where the planning problem lives
in	*gamma*	the \gamma (see Node::I_Node_factory::Get_Gamma()) parameter used by RRT∗
in	*traj_symm_flag*	a flag explaining whether the problem is symmetric or not, see Node::I_Node_factory::Get_symm_flag()

The documentation for this class was generated from the following file:

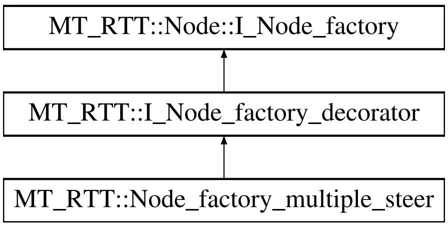
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_description.h

8.23 MT_RTT::Node_factory_multiple_steer Class Reference

Used for performing each steer operation multiple times, trying to reach faster the target node.

```
#include <Problem_description.h>
```

Inheritance diagram for MT_RTT::Node_factory_multiple_steer:



Public Member Functions

- Node_factory_multiple_steer (std::unique_ptr< I_Node_factory > &to_wrap, const size_t &max_numb_trials)
- virtual std::unique_ptr< I_Node_factory > copy ()

  used for cloning this object: a deep copy must be implemented.
- virtual void Steer (float ∗cost_steered, float ∗steered_state, const float ∗start_state, const float ∗target_state, bool ∗trg_reached)

  The Steer function of the wrapped I_Node_factory is called for a maximum number of times equal to Maximum_trial in order to reach target_state.
- virtual const float & **Get_Gamma** ()

Additional Inherited Members

8.23.1 Detailed Description

Used for performing each steer operation multiple times, trying to reach faster the target node.

The \gamma parameter (see Node::I_Node_factory::Get_Gamma) is set equal to the one of the wrapped I_Node_factory times the number of maximal steering triala.

8.23.2 Constructor & Destructor Documentation

8.23.2.1 Node_factory_multiple_steer()

```
MT_RTT::Node_factory_multiple_steer::Node_factory_multiple_steer (
            std::unique_ptr< I_Node_factory > & to_wrap,
            const size_t & max_numb_trials )
```

Parameters

in	*to_wrap*	the I_Node_factory to absorb
in	*max_numb_trials*	the maximum number of times for which the Steer must be tried

8.23.3 Member Function Documentation

8.23.3.1 copy()

```
std::unique_ptr< Node::I_Node_factory > MT_RTT::Node_factory_multiple_steer::copy ( )  [virtual]
```

used for cloning this object: a deep copy must be implemented.

This function is invoked by parallel planners for dispatching copies of this class to the other working threads. In this way, the threads must not be forced to synchronize for accessing the methods of an I↩_Node_factory. Therefore, when deriving your own factory describing your own problem, be carefull to avoid shallow copies and implement deep copies.

Parameters

out	*return*	a clone of this object

Implements MT_RTT::Node::I_Node_factory.

8.23.3.2 Steer()

```
void MT_RTT::Node_factory_multiple_steer::Steer (
            float * cost_steered,
            float * steered_state,
            const float * start_state,
            const float * target_state,
            bool * trg_reached )  [virtual]
```

The Steer function of the wrapped I_Node_factory is called for a maximum number of times equal to Maximum_trial in order to reach target_state.

The procedure is terminated before reaching the maximum trials, when a state not compliant with the constraints is reached.

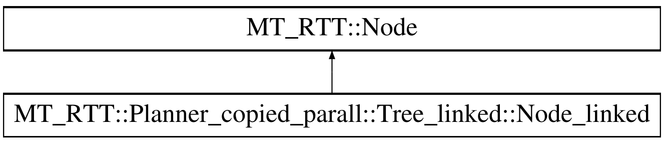
Reimplemented from MT_RTT::I_Node_factory_decorator.

The documentation for this class was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_description.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Problem_description.cpp

8.24 MT_RTT::Planner_copied_parall::Tree_linked::Node_linked Class Reference

Inheritance diagram for MT_RTT::Planner_copied_parall::Tree_linked::Node_linked:



Public Member Functions

- const std::vector< Node_linked ∗ > ∗ **Get_Copies** () const

Static Public Member Functions

- static void **create_linked_roots** (const Node_State &root_state, const vector< Tree_linked ∗ > &trees)
- static void **dispatch_last_added** (Tree_linked ∗tree)

Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_copied_parall.cpp

8.25 MT_RTT::Node_State Struct Reference

Used for externally represent a state x \in \mathcal{X}, Section 1.2.1 of the documentation.

```
#include <Problem_description.h>
```

Public Member Functions

- Node_State (const float *vals, const size_t &size)

    The values in vals are copied and stored in Node_State::Vals.
- Node_State (const Node_State &to_copy)

    Copy constructor.
- Node_State & **operator=** (const Node_State &)=delete
- const float & operator[ ] (const size_t &pos) const

    Access the value at position equal to pos.
- const size_t & size () const

    Returns the size of the represented state.

8.25.1 Detailed Description

Used for externally represent a state x \in \mathcal{X}, Section 1.2.1 of the documentation.

It cannot be empty.

8.25.2 Constructor & Destructor Documentation

8.25.2.1 Node_State()

```
MT_RTT::Node_State::Node_State (
             const float * vals,
             const size_t & size )
```

The values in vals are copied and stored in Node_State::Vals.

Parameters

in	*vals*	a pointer to the array of numbers describing the state to represent
in	*size*	the number of values in vals

8.25.3 Member Function Documentation

8.25.3.1 operator[]()

```
const float & MT_RTT::Node_State::operator[] (
             const size_t & pos ) const
```

Access the value at position equal to pos.

Parameters

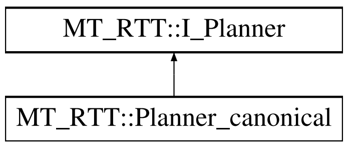
in	*pos*	the position of the value to acess in Node_State::Vals
out	*return*	the value at pos position

The documentation for this struct was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_description.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Problem_description.cpp

8.26 MT_RTT::Planner_canonical Class Reference

Inheritance diagram for MT_RTT::Planner_canonical:



Public Member Functions

- **Planner_canonical** (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory ∗handler)

Protected Member Functions

- virtual void _RRT_basic (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing an RRT search
- virtual void _RRT_bidirectional (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing a bidirectionl search
- virtual void _RRT_star (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing an RRT∗ search

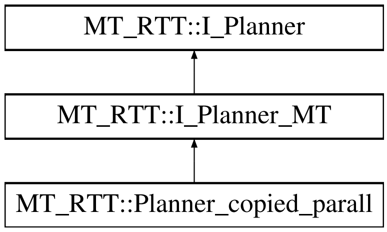
Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_canonical.cpp

8.27 MT_RTT::Planner_copied_parall Class Reference

Inheritance diagram for MT_RTT::Planner_copied_parall:



Public Member Functions

- **Planner_copied_parall** (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory ∗handler, const size_t &N_threads, const float &reallignement_percentage)

Protected Member Functions

- virtual void _RRT_basic (const Node_State &start, const Node_State &end)

    the method overrided by all the derived planner for performing an RRT search
- virtual void _RRT_bidirectional (const Node_State &start, const Node_State &end)

    the method overrided by all the derived planner for performing a bidirectionl search
- virtual void _RRT_star (const Node_State &start, const Node_State &end)

    the method overrided by all the derived planner for performing an RRT∗ search

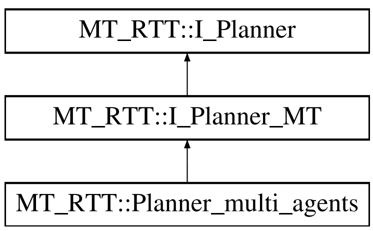
Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_copied_parall.cpp

8.28 MT_RTT::Planner_multi_agents Class Reference

Inheritance diagram for MT_RTT::Planner_multi_agents:



Public Member Functions

- **Planner_multi_agents** (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory ∗handler, const size_t &N_threads, const float &reallignement_percentage)

Protected Member Functions

- virtual void _RRT_basic (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing an RRT search
- virtual void _RRT_bidirectional (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing a bidirectionl search
- virtual void _RRT_star (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing an RRT∗ search

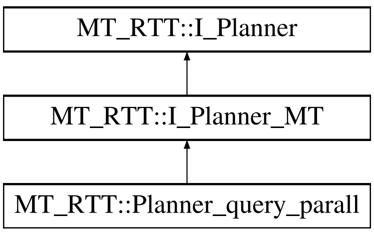
Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_multi_agents.cpp

8.29 MT_RTT::Planner_query_parall Class Reference

Inheritance diagram for MT_RTT::Planner_query_parall:



Public Member Functions

- **Planner_query_parall** (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory ∗handler, const size_t &N_threads)

Protected Member Functions

- virtual void _RRT_basic (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing an RRT search
- virtual void _RRT_bidirectional (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing a bidirectionl search
- virtual void _RRT_star (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing an RRT∗ search

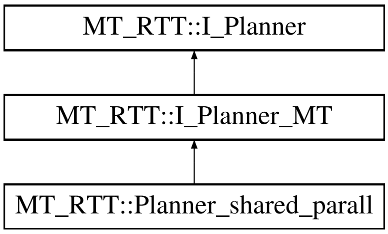
Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_query_parall.cpp

8.30 MT_RTT::Planner_shared_parall Class Reference

Inheritance diagram for MT_RTT::Planner_shared_parall:



Public Member Functions

- **Planner_shared_parall** (const float &det_coeff, const size_t &max_iter, Node::I_Node_factory ∗handler, const size_t &N_threads)

Protected Member Functions

- virtual void _RRT_basic (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing an RRT search
- virtual void _RRT_bidirectional (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing a bidirectionl search
- virtual void _RRT_star (const Node_State &start, const Node_State &end)

  the method overrided by all the derived planner for performing an RRT∗ search

Additional Inherited Members

The documentation for this class was generated from the following file:

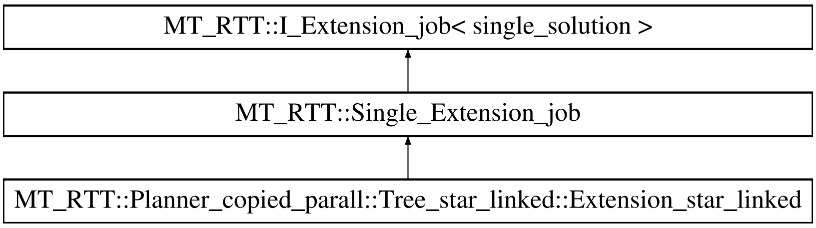
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Planner_shared_parall.cpp

8.31 MT_RTT::Single_Extension_job Class Reference

Handles the single tree extension strategy.

```
#include <Extensions.h>
```

Inheritance diagram for MT_RTT::Single_Extension_job:



Public Member Functions

- Single_Extension_job (I_Tree ∗to_extend, const Node_State &target, const float ∗det_coeff, const bool ∗cumul_sol)

    Constructor.
- virtual void Extend_within_iterations (const size_t &Iterations)

    Perform the specified number of estensions on a wrapped tree(s).
- virtual std::list< I_Tree ∗ > Remove_Trees ()

    Move outside of this object all the trees it contains.

Protected Member Functions

- virtual void **__Get_best_solution** (std::list< Node_State > ∗solution, const std::list< single_solution > &solutions)

Protected Attributes

- I_Tree ∗ **T**
- Node ∗ **Target**

Additional Inherited Members

8.31.1 Detailed Description

Handles the single tree extension strategy.

A tree is extended within the given iterations, both randomly and toward a specified target state. The deterministic extension finds the nearest neighbour of the tree to the target and try to steer it till reaching it. Here the solution consists in the pointer to the node that reached the target.

8.31.2 Constructor & Destructor Documentation

8.31.2.1 Single_Extension_job()

```
MT_RTT::Single_Extension_job::Single_Extension_job (
            I_Tree * to_extend,
            const Node_State & target,
            const float * det_coeff,
            const bool * cumul_sol )  [inline]
```

Constructor.

Parameters

in	*to_extend*	the tree to absorb and extend
in	*target*	the target node that must be connected to the tree to extend
in	*det_coeff*	same as in I_Extension_job::I_Extension_job
in	*cumul_sol*	same as in I_Extension_job::I_Extension_job
in	*del_trg*	when passed true, the absorbed target is deleted in the constructor of this object

8.31.3 Member Function Documentation

8.31.3.1 Extend_within_iterations()

```
void MT_RTT::Single_Extension_job::Extend_within_iterations (
            const size_t & Iterations )  [virtual]
```

Perform the specified number of estensions on a wrapped tree(s).

This function may be called multiple times, for performing batch of extensions. If the cumulation of the solution was not enabled, calling this method when a solution was already found raise an exception.

Parameters

in	*Iterations*	the number of extensions to perform

Implements MT_RTT::I_Extension_job< single_solution >.

8.31.3.2 Remove_Trees()

```
list< I_Tree * > MT_RTT::Single_Extension_job::Remove_Trees ( )  [virtual]
```

Move outside of this object all the trees it contains.

If the trees are not removed, they are all destroyed when destroying this object

Parameters

out	*return*	the trees that are removed from this object

Implements MT_RTT::I_Extension_job< single_solution >.

The documentation for this class was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Extensions.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Extensions.cpp

8.32 MT_RTT::Bubbles_free_configuration::I_Proximity_calculator↵ ::single_robot_prox Struct Reference

The distances concerning a single robot.

`#include <Problem_path_basic.h>`

Public Attributes

- float **Distance_to_fixed_obstacles**
- std::vector< float > **Radii**

8.32.1 Detailed Description

The distances concerning a single robot.

The documentation for this struct was generated from the following file:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_path_basic.h

8.33 MT_RTT::single_solution Struct Reference

Public Member Functions

- **single_solution** (const float &c, const Node ∗p)
- **single_solution** (const single_solution &o)
- bool **operator==** (const single_solution &o) const
- bool **operator<** (const single_solution &o) const

Public Attributes

- float **cost**
- const Node * **peer**

The documentation for this struct was generated from the following file:

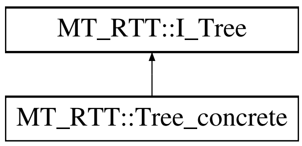
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Extensions.h

8.34 MT_RTT::Tree_concrete Class Reference

Interface for a concrete case of the decorator pattern modelling the Tree class.

```
#include <Tree.h>
```

Inheritance diagram for MT_RTT::Tree_concrete:



Public Member Functions

- Tree_concrete (const Node_State &root_state, Node::I_Node_factory *handler, const bool &clone↩_handler)

  This constructor initializes the tree with an initial root node.
- Tree_concrete (Node::I_Node_factory *handler, const bool &clone_handler)

  Similar to Tree_concrete::Tree_concrete(const Node_State& root_state, Node::I_Node_factory handler, const bool& clone_handler), but without inserting an initial root node.*
- virtual Node::I_Node_factory * Get_Problem_Handler ()

  Get the object describing the planning problems this tree refers to.

Protected Member Functions

- virtual Node * **Extend** (const Node *target)
- virtual std::list< Node * > * **Get_Nodes** ()
- virtual _extend_info * **Get_extend_info** ()
- virtual size_t **__get_Nodes_size** ()
- virtual Node * **Nearest_Neighbour** (const Node *state)

Protected Attributes

- bool **Problem_handler_was_cloned**
- Node::I_Node_factory * **Problem_handler**
- std::list< Node * > **Nodes**
- _extend_info **__ext_info**

Additional Inherited Members

8.34.1 Detailed Description

Interface for a concrete case of the decorator pattern modelling the Tree class.

This interface actually contains a list of nodes, which can be made available for all the wrapping decorators.

8.34.2 Constructor & Destructor Documentation

8.34.2.1 Tree_concrete() [1/2]

```
MT_RTT::Tree_concrete::Tree_concrete (
            const Node_State & root_state,
            Node::I_Node_factory * handler,
            const bool & clone_handler )
```

This constructor initializes the tree with an initial root node.

Parameters

in	*root_state*	the state of the root node to consider for the tree.
in	*handler*	a reference to the object describing the planning problem to solve
in	*clone_handler*	when passed true, the handler is internally cloned and the cloned handler will be the one used by this class, which is in charge of destroying it when this object is destroyed. Otherwise the passed handler is used and will be not destroyed when destroying this object.

8.34.2.2 Tree_concrete() [2/2]

```
MT_RTT::Tree_concrete::Tree_concrete (
            Node::I_Node_factory * handler,
            const bool & clone_handler )
```

Similar to Tree_concrete::Tree_concrete(const Node_State& root_state, Node::I_Node_factory∗ handler, const bool& clone
but without inserting an initial root node.

Parameters

in	*handler*	same as in Tree_concrete::Tree_concrete(const Node_State& root_state, Node::I_Node_factory∗ handler, co
in	*clone_handler*	same as in Tree_concrete::Tree_concrete(const Node_State& root_state, Node::I_Node_factory∗ handler, co

8.34.3 Member Function Documentation

8.34.3.1 Get_Problem_Handler()

`virtual Node::I_Node_factory* MT_RTT::Tree_concrete::Get_Problem_Handler ( ) [inline], [virtual]`

Get the object describing the planning problems this tree refers to.

Parameters

out	*return*	the object describing the planning problem

Implements MT_RTT::I_Tree.

The documentation for this class was generated from the following files:

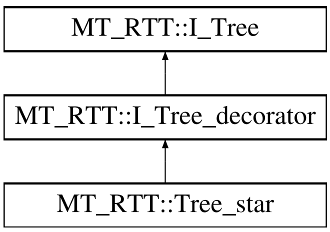
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Tree.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Tree.cpp

8.35 MT_RTT::Tree_star Class Reference

This decorator perform the RRT∗ extension steps (near set computation, rewirds, etc. refer to Section 1.2.3 of the documentation).

`#include <Tree.h>`

Inheritance diagram for MT_RTT::Tree_star:



Classes

- struct Node2Node_Traj

  This structure store the information regarding a possible rewird, i.e. a trajectory going from a starting node to an ending one, which are not already connected. cost is the cost to go from start to end.

Public Member Functions

- Tree_star (I_Tree *to_wrap, const bool &destroy_wrap)

  Constructor.
- void Connect_to_best_Father_and_eval_Rewirds (std::list< Node2Node_Traj > *possible_rewirds, Node *last_added)

  This method connect last_added node to its best father among the nodes in its near set and also evaluates the possible rewirds (Section 1.2.3 of the documentation) to do, wihtout performing it.

Protected Member Functions

- virtual Node * **Extend** (const Node *target)
- virtual void **Near_set** (std::list< Node * > *near_set, const Node *state)

Additional Inherited Members

8.35.1 Detailed Description

This decorator perform the RRT* extension steps (near set computation, rewirds, etc. refer to Section 1.2.3 of the documentation).

Such steps are automatically performed when invoking Extend_random or Extend_deterministic, when the extension was possible and a new node was actually inserted in the tree.

8.35.2 Constructor & Destructor Documentation

8.35.2.1 Tree_star()

```
MT_RTT::Tree_star::Tree_star (
            I_Tree * to_wrap,
            const bool & destroy_wrap )  [inline]
```

Constructor.

Parameters

in	*to_wrap*	the tree to decorate, for performing RRT* extensions
in	*destroy_wrap*	same meaning as in I_Tree_decorator::I_Tree_decorator

8.35.3 Member Function Documentation

8.35.3.1 Connect_to_best_Father_and_eval_Rewirds()

```
void MT_RTT::Tree_star::Connect_to_best_Father_and_eval_Rewirds (
            std::list< Node2Node_Traj > * possible_rewirds,
            Node * last_added )
```

This method connect last_added node to its best father among the nodes in its near set and also evaluates the possible rewirds (Section 1.2.3 of the documentation) to do, wihtout performing it.

Basically, this function evaluates the rewirds, i.e. possible reconnection of the tree, for performing it at a second stage.

Parameters

out	*possible_rewirds*	the possible rewirds to do for improving the connectivity
out	*last_added*	the node considered for the near set computation

The documentation for this class was generated from the following files:

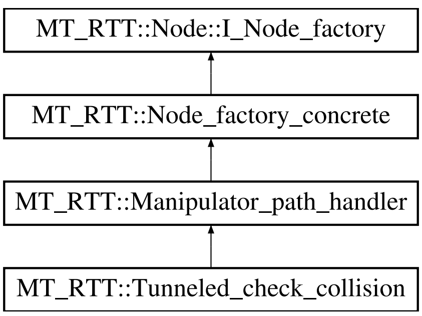
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Tree.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Tree.cpp

8.36 MT_RTT::Tunneled_check_collision Class Reference

In this object, the collision along a certain segment in the configurational space (i.e. a trajectory connecting two nodes) is checked considering a discrete set of equispaced samples, Section 2.2.3 of the documentation.

`#include <Problem_path_basic.h>`

Inheritance diagram for MT_RTT::Tunneled_check_collision:



Classes

- class I_Collision_checker

    The object in charge of performing the collision check for a single generic configuration.

Public Member Functions

- I_Collision_checker * Get_checker ()

  Returns the checker contained in this object.
- Tunneled_check_collision (const float &Gamma, const float &steer_degree, const Node_State &Q↩_max, const Node_State &Q_min, std::unique_ptr< I_Collision_checker > &coll_checker)

  Constructor.
- Tunneled_check_collision (const float &Gamma, const float &steer_degree, const float &q_max, const float &q_min, const size_t &dof, std::unique_ptr< I_Collision_checker > &coll_checker)

  Constructor.
- virtual std::unique_ptr< I_Node_factory > copy ()

  used for cloning this object: a deep copy must be implemented.
- virtual void Steer (float *cost_steered, float *steered_state, const float *start_state, const float *target_state, bool *trg_reached)

  The steered pose lies in the segment connecting the nearest neighbour to the target node, at a distance (euclidean distance in the configurational space) which at most equal to steering degree.
- virtual void Cost_to_go_constraints (float *result, const float *start_state, const float *ending_state)

  The collisions are checked only for some equispace intermediate poses lying on the segment connectin the starting state to the ending one. If a collision is detected, FLT_MAX is set as result, while in the opposite case the euclidean distance of the two poses is returned.

Additional Inherited Members

8.36.1 Detailed Description

In this object, the collision along a certain segment in the configurational space (i.e. a trajectory connecting two nodes) is checked considering a discrete set of equispaced samples, Section 2.2.3 of the documentation.

An external object, in charge of performing the collision check must be passed and absorbed.

8.36.2 Constructor & Destructor Documentation

8.36.2.1 Tunneled_check_collision() [1/2]

```
MT_RTT::Tunneled_check_collision::Tunneled_check_collision (
            const float & Gamma,
            const float & steer_degree,
            const Node_State & Q_max,
            const Node_State & Q_min,
            std::unique_ptr< I_Collision_checker > & coll_checker )
```

Constructor.

Parameters

in	*coll_checker*	the object in charge of performing the collision check of a single pose
in	*Gamma*	same meaning as in Manipulator_path_handler::Manipulator_path_handler
in	*steer_degree*	a steered pose lies in the segment connecting the nearest neighbour to the target node, at a distance (euclidean distance in the configurational space) which is at most equal to the steer_degree
in	*Q_max*	same meaning as in Manipulator_path_handler::Manipulator_path_handler
in	*Q_min*	same meaning as in Manipulator_path_handler::Manipulator_path_handler

8.36.2.2 Tunneled_check_collision() [2/2]

```
MT_RTT::Tunneled_check_collision::Tunneled_check_collision (
            const float & Gamma,
            const float & steer_degree,
            const float & q_max,
            const float & q_min,
            const size_t & dof,
            std::unique_ptr< I_Collision_checker > & coll_checker )
```

Constructor.

Similar to Tunneled_check_collision::Tunneled_check_collision(const float& Gamma, const float& steer_degree, const Nod
but assuming that Q_max (and Q_min) have all the same values equal to q_max and has a size equal to dof.

8.36.3 Member Function Documentation

8.36.3.1 copy()

```
virtual std::unique_ptr<I_Node_factory> MT_RTT::Tunneled_check_collision::copy ( )  [inline],
[virtual]
```

used for cloning this object: a deep copy must be implemented.

This function is invoked by parallel planners for dispatching copies of this class to the other working threads. In this way, the threads must not be forced to synchronize for accessing the methods of an I↩_Node_factory. Therefore, when deriving your own factory describing your own problem, be carefull to avoid shallow copies and implement deep copies.

Parameters

out	*return*	a clone of this object

Implements MT_RTT::Node::I_Node_factory.

8.36.3.2 Cost_to_go_constraints()

```
void MT_RTT::Tunneled_check_collision::Cost_to_go_constraints (
            float * result,
            const float * start_state,
            const float * ending_state )  [virtual]
```

The collisions are checked only for some equispace intermediate poses lying on the segment connectin the starting state to the ending one. If a collision is detected, FLT_MAX is set as result, while in the opposite case the euclidean distance of the two poses is returned.

The number of intermediate poses is chosen so as to realize that the intermediate poses are distant no more than the steering degree

Implements MT_RTT::Node::I_Node_factory.

The documentation for this class was generated from the following files:

- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Header/Problem_path_basic.h
- C:/Program Files/Librerie_C/My_Code/MT_RRT/MT_RRT/Source/Problem_path_basic.cpp

Index

Bibliography

[1] *Bullet3*. https://github.com/bulletphysics/bullet3.

[2] *ReactPhysics3D*. https://www.reactphysics3d.com.

[3] Andrea Casalino, Andrea Maria Zanchettin, and Paolo Rocco. Mt-rrt: a general purpose multithreading library for path planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2019)*, pages 1510–1517. IEEE, 2019.

[4] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[5] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[6] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[7] Sean Quinlan. *Real-time modification of collision-free paths*. Number 1537. Stanford University Stanford, 1994.

[8] Adnan Tahirovic and Faris Janjoš. A class of sdre-rrt based kinodynamic motion planners. In *American Control Conference (ACC)*, volume 2018, 2018.

[9] Dustin J Webb and Jur van den Berg. Kinodynamic rrt*: Optimal motion planning for systems with linear differential constraints. 2012.