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 [**?**]. 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 [**?**] and [**?**].
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

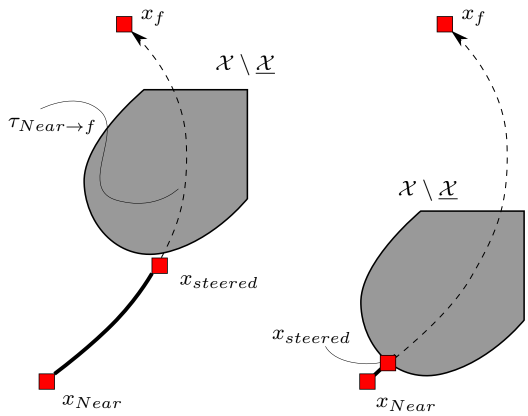


Figure 1.1 The dashed curves in both pictures are the optimal trajectories, agnostic of the constraints, connecting the pair of states x_{Near} and x_f, while the filled areas are regions of X not allowed by constraints. The steering procedure is ideally in charge of searching the furthest state to x_{Near} 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 [**?**], 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 [**?**]. The optimality is addressed by a variant of the RRT, called RRT* [**?**], 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.

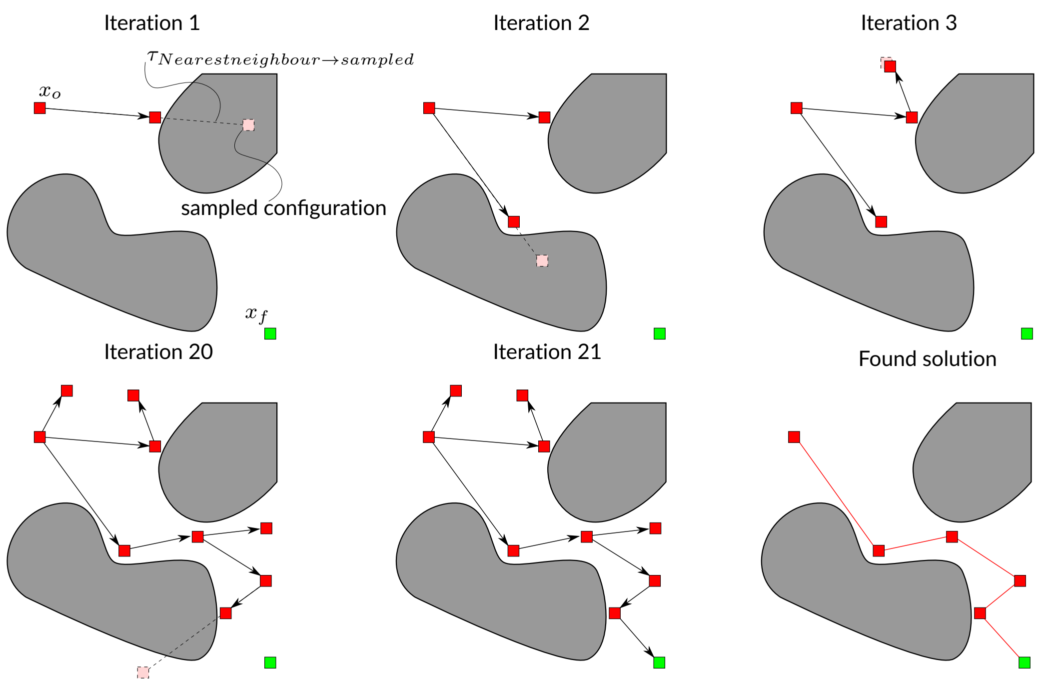


Figure 1.2 Examples of iterations done by an RRT algorithm. The solution found is the one connecting the state in the tree that reached x_f, with the root x_o.

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.

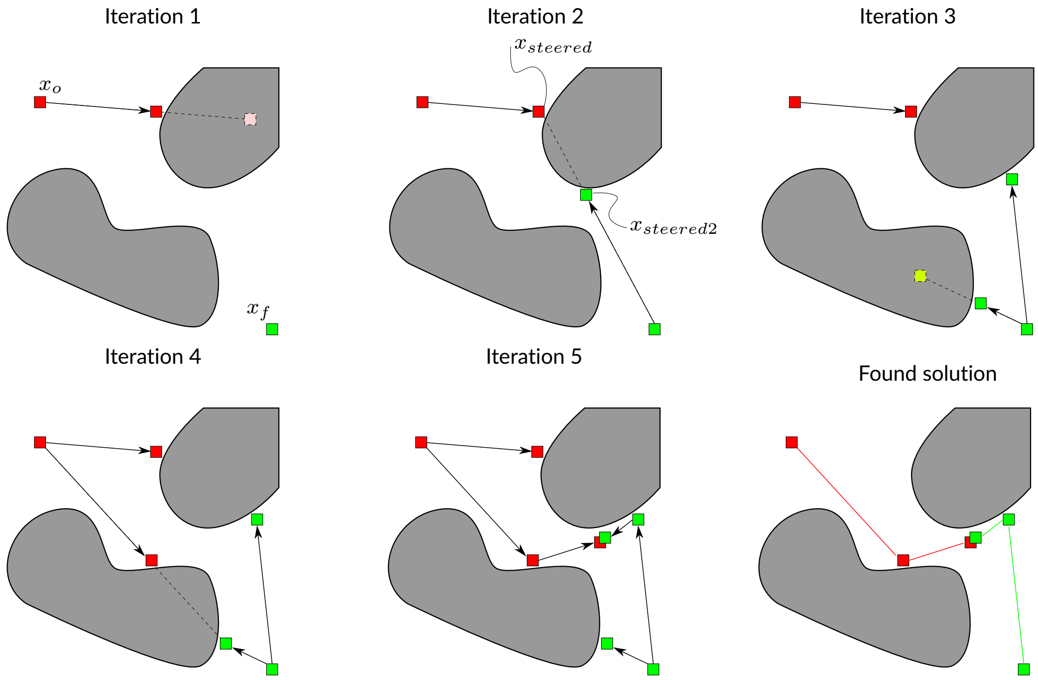


Figure 1.3 Examples of iterations done by the bidirectional version of the RRT. The path in the tree rooted at x_f is reverted to get the second part of the solution.

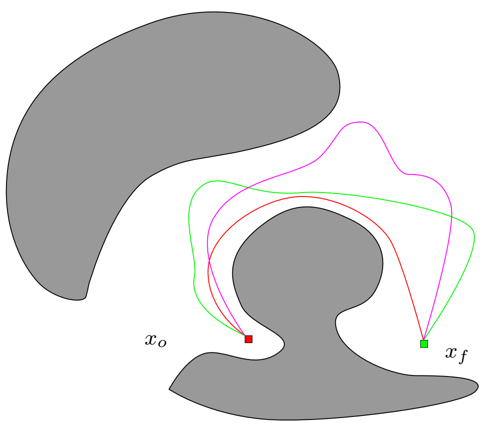


Figure 1.4 Different trajectories connecting x_o with x_f, entirely contained in $\underline{\mathcal{X}}$. If we assume ad cost the length of a path, the red solution is the optimal one.

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.

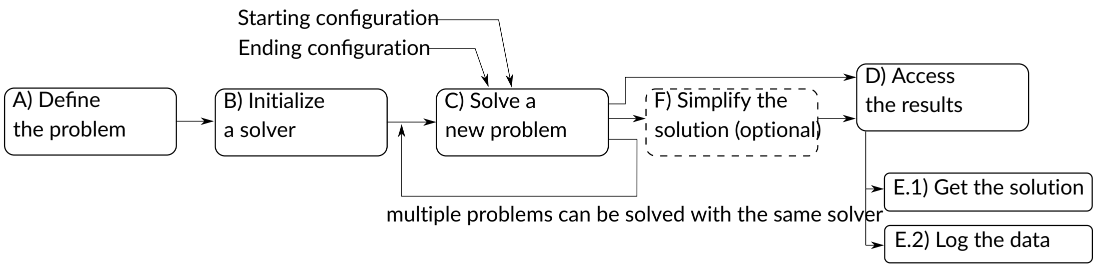


Figure 1.5 Pipeline to follow for using MT_RRT.

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

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

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

json version

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

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

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

$$\vdots$$

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

Figure 1.6 Format of the json file returning the results.

1.3.0.4 D) Access the results

There two possible way to access the results computed by a solver.

E.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.

E.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.

1.3.0.5 F) Post processing of the solution

It is possible to post process the solution gained by the solver in order to try to improve it. This is particularly relevant for the approaches presented in 1.2.1 and 1.2.2, while may be omitted for the one discussed in 1.2.3 (although it is possible to either post process the solution in order to improve it a little bit more). In order to post process a solution, an object deriving from I_Simplifier must be used. To be precise, the solution is internally improved by a solver, building an instance of a particular simplifier. Then, when accessing the solution, step E.1, you are actually accessing the improved solution. By default no simplifications are done and you can instruct the solver about the processor to use by calling I_Planner::Set_post_processer.

[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.

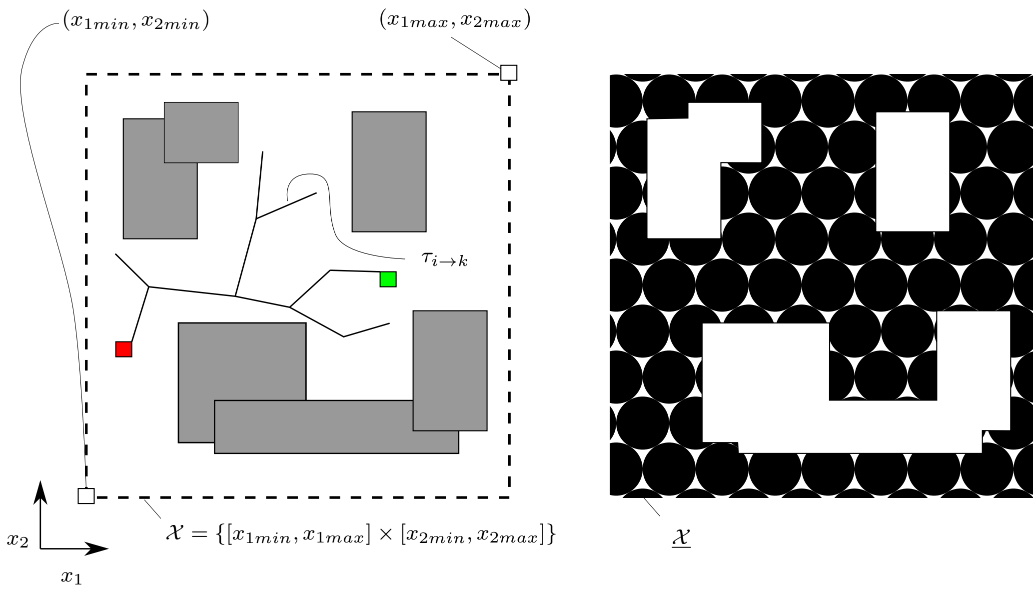


Figure 2.1 Example of maze problem.

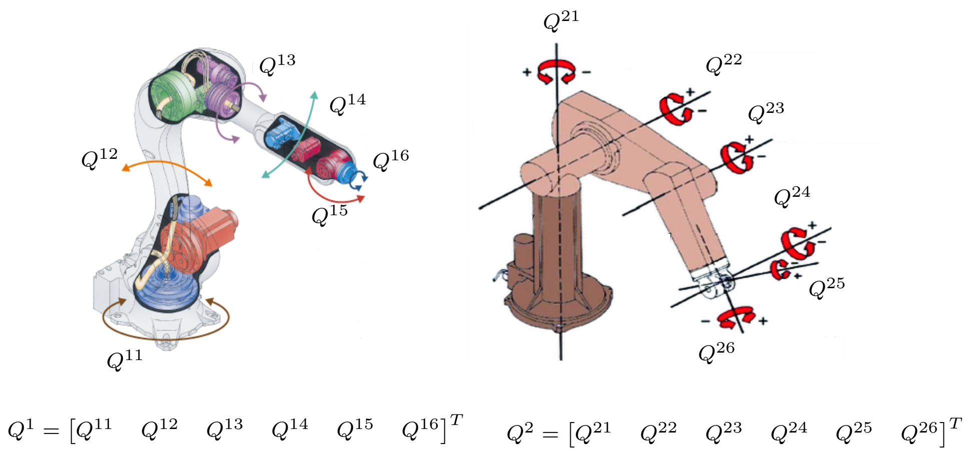


Figure 2.2 Rotating joints of two articulated manipulators.

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:

$$x_i = Q_i = \left[(Q_i^1)^T \quad \dots \quad (Q_i^n)^T\right]^T \tag{2.3}$$

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 $\tau_{i \rightarrow k}$:

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

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 [**?**] or [**?**]) 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 I_Node_factory::Set_Steer_iterations.
Figure 2.3 summarizes the above considerations.

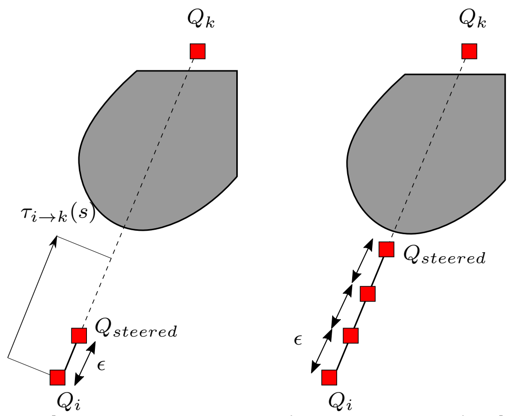


Figure 2.3 Steer extension along the segment connecting two states in the configuration space. On the left a single steer approach, on the right a multiple one.

2.2.3.2 Bubble of free configuration

This approach was first proposed in [?] 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 [1]

$$\mathcal{B}(\bar{Q} = \begin{bmatrix}\bar{Q}^{1T} & \dots & \bar{Q}^{nT}\end{bmatrix}^T) = \mathcal{B}_O(\bar{Q}) \cap \mathcal{B}_C(\bar{Q}) \tag{2.7}$$

where $\mathcal{B}_O$ contains describes the region containing the poses guaranteed to not manifest collisions with the fixed obstacles, while $\mathcal{B}_C$ describes the poses for which the robots do not collide with each others. They are defined as follows:

$$\mathcal{B}_O(\bar{Q}) = \left\{ Q \middle| \forall j \in \{1, \dots, n\} \sum_i R^{ji} |Q^{ji} - \bar{Q}^{ji}| \leq d^j_{min} \right\} \tag{2.8}$$

$$\mathcal{B}_C(\bar{Q}) = \left\{ Q \middle| \forall j,k \in \{1, \dots, n\} \sum_i R^{ji} |Q^{ji} - \bar{Q}^{ji}| + \sum_i R^{ki} |Q^{ki} - \bar{Q}^{ki}| \leq d^{jk}_{min} \right\} \tag{2.9}$$

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:

$$\begin{aligned}
Q_{steered} &= \begin{bmatrix}Q^{1T}_{steered} & \dots & Q^{nT}_{steered}\end{bmatrix}^T = Q_i + s_{steered}(Q_k - Q_i) \\
s_{steered} &= min\left\{ s_A, s_B \right\} \\
s_A &= min_{j \in \{1,\dots,n\},q} \left\{ \frac{d^j_{min}}{\sum_q R^{jq} |Q^{jq}_i - Q^{jq}_k|} \right\} \\
s_B &= min_{j,k \in \{1,\dots,n\},q,q_2} \left\{ \frac{d^{jk}_{min}}{\sum_q R^{jq} |Q^{jq}_i - Q^{jq}_k| + \sum_{q_2} R^{kq_2} |Q^{kq_2}_i - Q^{kq_2}_k|} \right\}
\end{aligned} \tag{2.10}$$

Also in this case a multiple steer approach is possible for this strategy, 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.

[1] where $\bar{Q}^{jT}$ refers to the pose of the j^{th} robot, see equation (2.5).

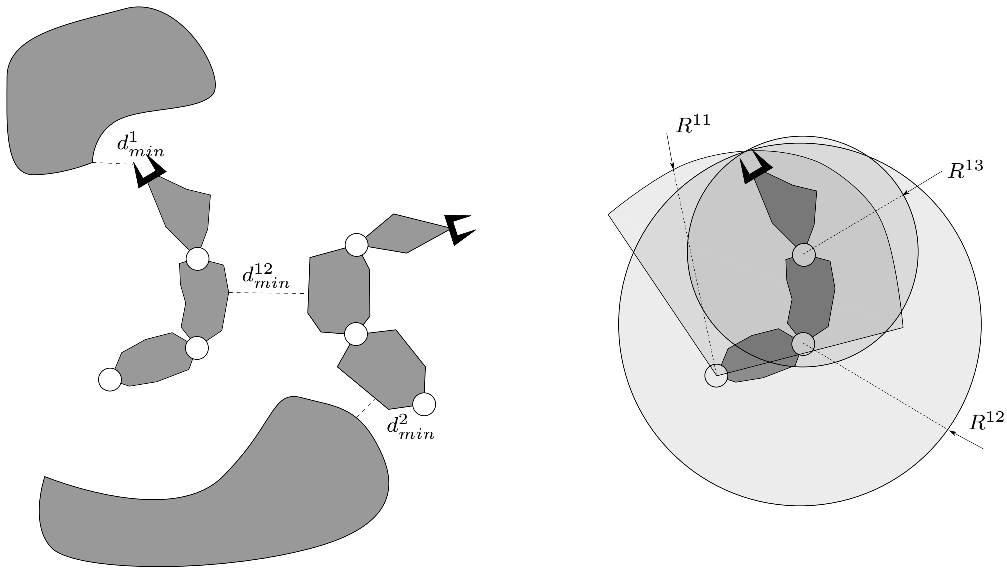


Figure 2.4 The quantities involved in the computation of the bubble $\mathcal{B}$.

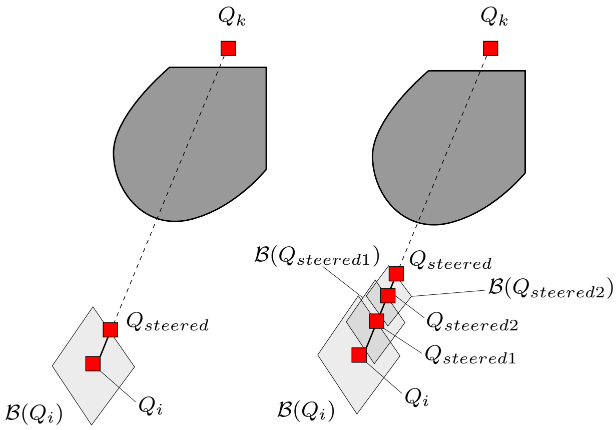


Figure 2.5 Single (left) and multiple (right) steer using the bubbles of free configurations.

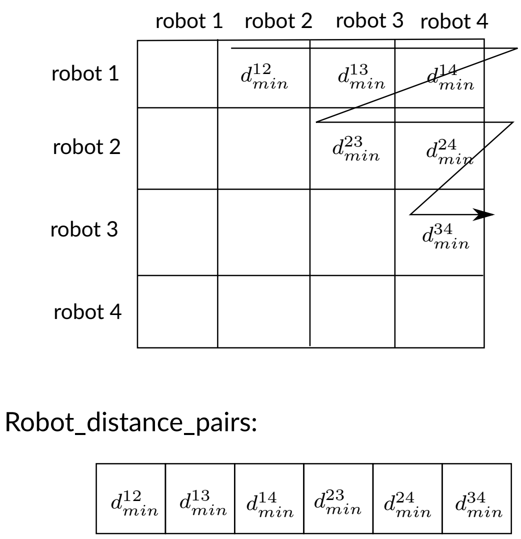


Figure 2.6 Values stored in Robot_distance_pairs.

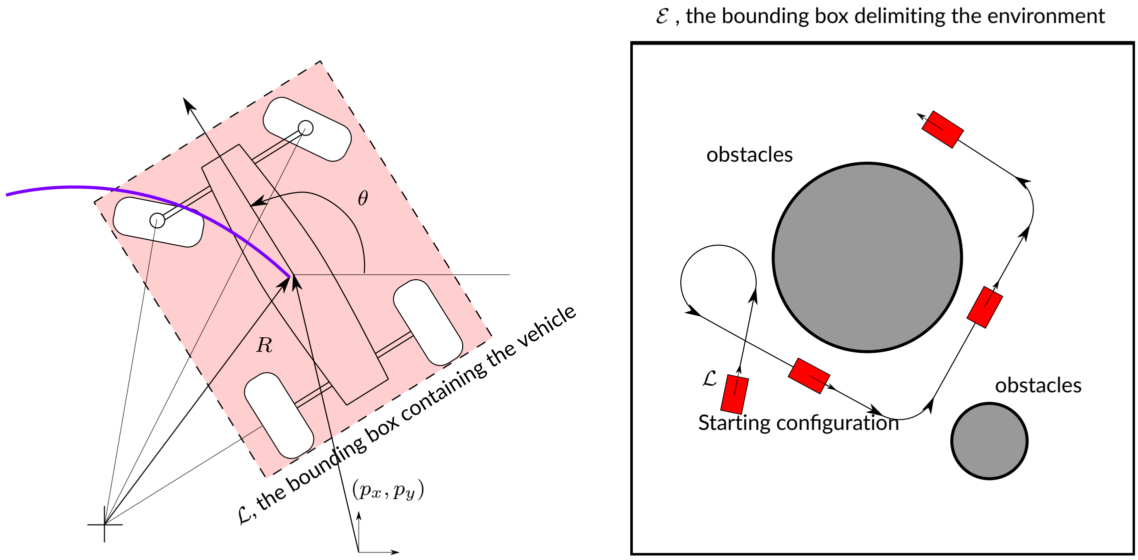


Figure 2.7 Vehicle motion in a planar environment.

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 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:

$$x_R = \begin{bmatrix} (p_{xi}, p_{yi}) \sim \mathcal{E} \\ \theta \sim U(-\pi, \pi) \end{bmatrix} \tag{2.11}$$

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 $\tau_{i \to j}$ does not exists, $C(\tau_{i \to j})$ is assumed equal to $+\infty$.

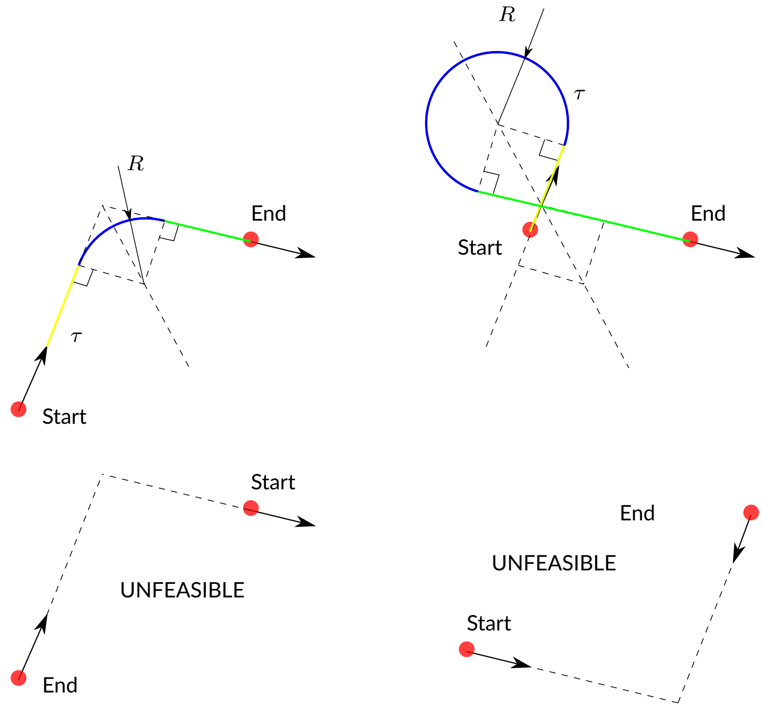


Figure 2.8 Examples of feasible, top, and non feasible trajectories, bottom. The different parts of the feasible trajectories are highlighted with different colors.

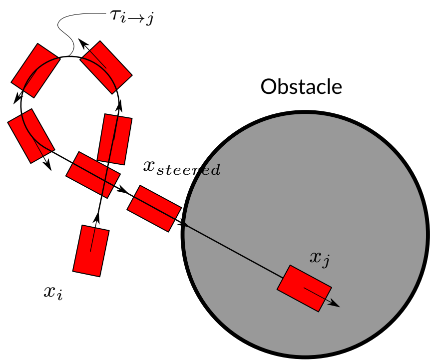


Figure 2.9 Steering procedure for a planar navigation problem.

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 [?], which is the publication were for the first time MT_RRT was presented. In [?] 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.

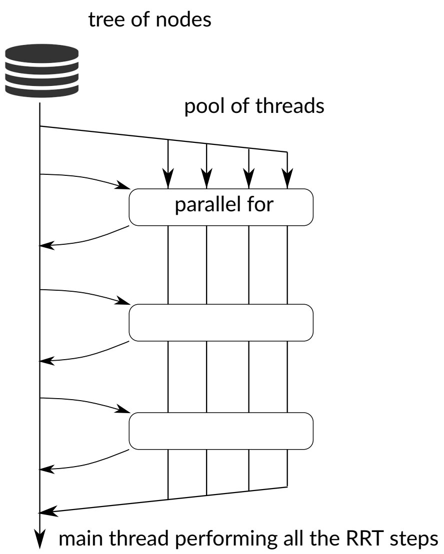
3.0.2 Shared tree critical regions

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.

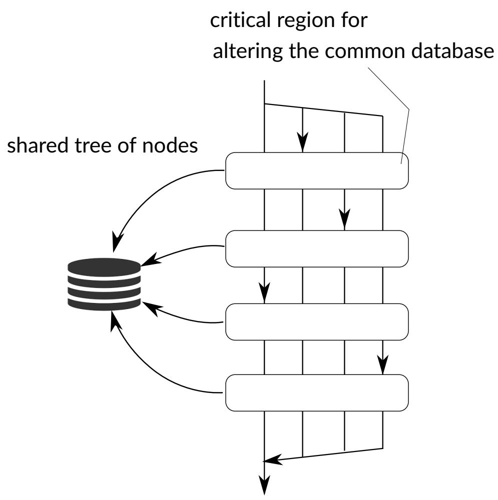
3.0.3 Parallel expansions of copied trees

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

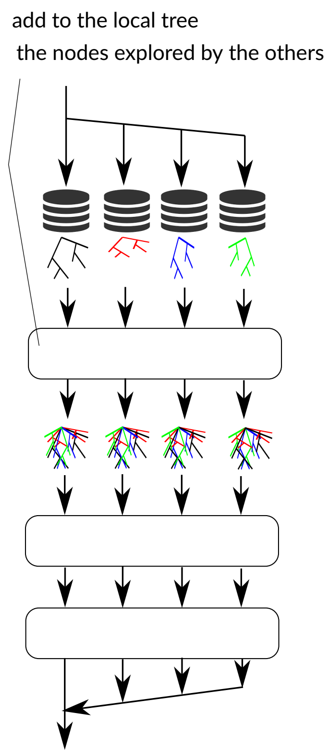
[1]They are dispatched into proper buffer, but not directly inserted in the private copies of the other trees.



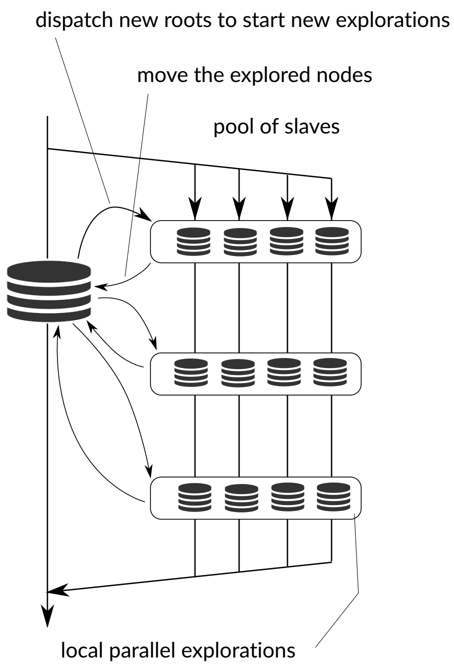
(a) Schematic representation of the parallelization of the query activities approach.



(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 Namespace Reference

Namespaces

- sampling
- solver
- traj

Classes

- class Copiable

  Interface for a copiable object.
- class Error

  A runtime error that can be raised by any object inside mt::
- class ExtBidir
- class Extender

  Used to extend one or two connected search trees.
- class ExtSingle
- class Limited

  A numeric quantity whose value should always remain between defined bounds.
- class LowerLimited

  A @Limited quantity, having +infinite as upper bound.
- class Node

  Used for representing a state x \in \underline{\mathcal{X}}, Section METTERE of the documentation.
- class Positive

  A @LowerLimited quantity, having 0.0 as lower bound.
- class Problem

  Object storing the information needed to extend exploring trees.
- struct Rewire

  newFather should be set as father node for involved, with a cost to go equal to newCostFromFather
- class Tree

  Interface for a Nodes container. Minimal functionalties to iterate the container should be implemented in descendants.
- class TreeBase

Base class Tree, storing a problem pointer.
- class TreeCore

  Tree with the minimal functionalities required to implement an rrt algorithm.
- class TreeExtendable

  Base class for an extendable tree, i.e. a tree whose nodes can be incremented over time.
- class TreeIterable

  Base Tree class physically storing the nodes.
- class TreeRewirer

  Base Tree class with the capability of performing rewires, refer to METTERE.
- class TreeStar

  A tree that always compute and applies the rewires when adding a new node in the tree.
- class UpperLimited

  A @Limited quantity, having -infinite as lower bound.

Typedefs

- typedef std::vector< float > **NodeState**
- typedef std::unique_ptr< Node > **NodePtr**
- typedef std::unique_ptr< Problem > **ProblemPtr**
- typedef std::list< NodePtr > **Nodes**
- typedef std::unique_ptr< Tree > **TreePtr**
- typedef std::unique_ptr< const Tree > **TreePtrConst**
- typedef std::tuple< const Node *, const Node *, float > **BidirSolution**
- typedef std::pair< const Node *, float > **SingleSolution**

Functions

- std::vector< NodeState > **convert** (const std::list< const NodeState * > nodes)
- TreeCore * **convert** (Tree *t)
- template<typename Extender >
  std::size_t getIterationsDone (const std::vector< Extender > &battery)
- bool **operator**< (const BidirSolution &a, const BidirSolution &b)
- std::vector< ExtBidir > **make_battery** (const bool &cumulateSolutions, const double &deterministic↩ Coefficient, const std::vector< TreePtr > &treesA, const std::vector< TreePtr > &treesB)
- bool **operator**< (const SingleSolution &a, const SingleSolution &b)
- std::vector< ExtSingle > **make_battery** (const bool &cumulateSolutions, const double &deterministic↩ Coefficient, const std::vector< TreePtr > &trees, const NodeState &target)

7.1.1 Detailed Description

Author: Andrea Casalino Created: 16.02.2021

report any bug to `andrecasa91@gmail.com`.

Author: Andrea Casalino Created: 16.05.2019

report any bug to `andrecasa91@gmail.com`.

7.1.2 Function Documentation

7.1.2.1 getIterationsDone()

```
template<typename Extender >
std::size_t mt::getIterationsDone (
            const std::vector< Extender > & battery )
```

Returns

the sum of extensions done by all the passed extenders

7.2 mt::sampling Namespace Reference

Classes

- class HyperBox
  A sampler drawing a sample inside an hypercube of n-dimensions, described by 2 corners. For example, corners [l1, l2, l3, l4] and [u1, u2, u3, u4], describe an hyperbox whose points [x1,x2,x3,x4] are all such that: li <= xi <= ui
- class Sampler
  Interface for a sampler of states.

Typedefs

- typedef std::unique_ptr< Sampler > **SamplerPtr**

7.2.1 Detailed Description

Author: Andrea Casalino Created: 16.05.2019

report any bug to andrecasa91@gmail.com.

7.3 mt::solver Namespace Reference

Namespaces

- linked
- multiag
- qpar
- shared

Classes

- class LinkedTreesStrategy
  strategy described in METTERE
- class MultiAgentStrategy
  strategy described in METTERE
- struct Parameters
  Parameters used to solve a planning problem.
- class QueryParallStrategy
  strategy described in METTERE
- class SerialStrategy
  The standard serial strategy described in METTERE.
- class SharedTreeStrategy
  strategy described in METTERE
- struct SolutionInfo
  Is produced internally to @Solver, every time a new planning problem is solved. The various quantity can be then accessed using the getters of @Solver.
- class Solver
  Solver storing results of planning problem. Every time solve(...) is called, a new problem is solved and the results can be accessed using the getters provided in this interface. When another problem is solved calling again solve(...), the information regarding the previous problem are lost.
- struct SolverData
- class Strategy
  An interface for an object in charge of solving a plannig problem.

Enumerations

- enum RRTStrategy { **Single**, **Bidir**, **Star** }
  The kind of rrt strategy to use, refer to METTERE.

7.3.1 Detailed Description

Author: Andrea Casalino Created: 16.05.2019

report any bug to `andrecasa91@gmail.com`.

7.4 mt::solver::linked Namespace Reference

Classes

- class ListLinked
- class NodeLinked
- class TreeContainer
- class TreeLinked
- class TreeStarContainer
- class TreeStarLinked

Functions

- std::vector< NodePtr > **make_copies** (Node &node)

7.4.1 Detailed Description

Author: Andrea Casalino Created: 16.05.2019

report any bug to andrecasa91@gmail.com.

7.5 mt::solver::multiag Namespace Reference

Classes

- class TreeMaster
- class TreeSlave
- class TreeStarMaster

7.5.1 Detailed Description

Author: Andrea Casalino Created: 16.05.2019

report any bug to andrecasa91@gmail.com.

7.6 mt::solver::qpar Namespace Reference

Classes

- class Pool
- class Query
- class TreeIterator
- class TreeQPar
- class TreeStarQPar

Typedefs

- typedef std::function< void(void)> **Job**

Functions

- template<typename Q >
  std::vector< Q > **make_results** (const std::vector< Problem * > &problems, const Tree &tree)

7.6.1 Detailed Description

Author: Andrea Casalino Created: 16.05.2019

report any bug to andrecasa91@gmail.com.

7.7 mt::solver::shared Namespace Reference

Classes

- class TreeShared
- class TreeStarShared

7.7.1 Detailed Description

Author: Andrea Casalino Created: 16.05.2019

report any bug to andrecasa91@gmail.com.

7.8 mt::traj Namespace Reference

Classes

- class Cost

  Describes a cost to go.
- class Line

  Advances along a segment in the state space, traversing everytime a distance not higher than steerDegree.
- class LineFactory
- class LineTrgSaved

  Internally saves the target state, in order for the const refernce stored in Line to remain meaningful.
- class TargetStorer
- class Trajectory

  Interface describing an optimal trajectory connecting 2 states, in a particular problem to solve. Refer to METTERE. A cursor internally stored the state currently reached. When avancing this object, the cursor is modified in order to traverse the trajectory.
- class TrajectoryBase

  Base class for a Trajectory. Calling advance a second time, after the first one returned blocked throw an exception.
- class TrajectoryComposite

  Base class for a Trajectory made of pieces of sub-ones.
- class TrajectoryFactory

  Creator of optimal trajectories, refer to METTERE. Each specific problem to solve need to define and use its specific TrajectoryFactory.

Typedefs

- typedef std::unique_ptr< Trajectory > **TrajectoryPtr**
- typedef std::unique_ptr< TrajectoryFactory > **TrajectoryFactoryPtr**

Enumerations

- enum AdvanceInfo { **blocked**, **advanced**, **targetReached** }

  blocked -> when the advancement is not anymore possible, i.e. last state reached is not admitted by constraints advanced -> normal advancement. Last state reached is admitted by constraints. targetReached -> when the last advancement led to the target state

Functions

- float euclideanDistance (const float *stateA, const float *stateB, const std::size_t &buffersSize)

  Computes the euclidean distance of bufferA w.r.t bufferB, both having a size equal to buffersSize.

7.8.1 Detailed Description

Author: Andrea Casalino Created: 16.05.2019

report any bug to `andrecasa91@gmail.com`.

Chapter 8

Class Documentation

8.1 mt::Copiable< T > Class Template Reference

Interface for a copiable object.

```
#include <Copiable.h>
```

Public Member Functions

- virtual std::unique_ptr< T > copy () const =0
  A deep copy need to be implemented for the descendant.

8.1.1 Detailed Description

template<typename T>
class mt::Copiable< T >

Interface for a copiable object.

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

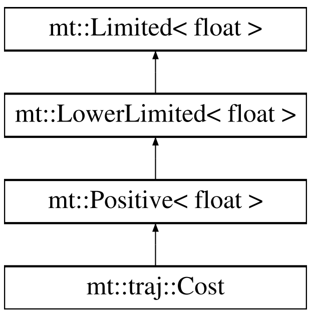
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Copiable.h

8.2 mt::traj::Cost Class Reference

Describes a cost to go.

```
#include <Cost.h>
```

Inheritance diagram for mt::traj::Cost:



Static Public Attributes

- static const float COST_MAX
  Equal to the maximum possible float and assumed as upper bound for this object.

Additional Inherited Members

8.2.1 Detailed Description

Describes a cost to go.

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

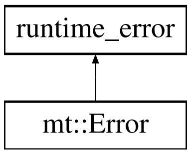
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/Cost.h

8.3 mt::Error Class Reference

A runtime error that can be raised by any object inside mt::

```
#include <Error.h>
```

Inheritance diagram for mt::Error:



Public Member Functions

- **Error** (const std::string &what)

8.3.1 Detailed Description

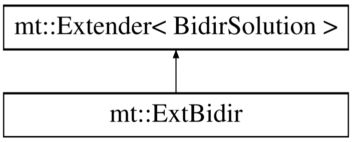
A runtime error that can be raised by any object inside mt::

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Error.h

8.4 mt::ExtBidir Class Reference

Inheritance diagram for mt::ExtBidir:



Public Member Functions

- **ExtBidir** (const bool &cumulateSolutions, const double &deterministicCoefficient, Tree &leftTree, Tree &rightTree)
- void extend (const std::size_t &Iterations) override

  Perform the specified number of estensions on the wrapped tree(s). This function may be called multiple times, for performing batch of extensions. All the solutions found while extending are saved and stored in this object.
- std::vector< NodeState > **computeSolutionSequence** (const BidirSolution &sol) const override

Additional Inherited Members

8.4.1 Member Function Documentation

8.4.1.1 extend()

```
void mt::ExtBidir::extend (
            const std::size_t & Iterations )  [override], [virtual]
```

Perform the specified number of estensions on the wrapped tree(s). This function may be called multiple times, for performing batch of extensions. All the solutions found while extending are saved and stored in this object.

Parameters

the	number of extension to perform

Implements mt::Extender< BidirSolution >.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/extn/header/ExtenderBidir.h

8.5 mt::Extender< Solution > Class Template Reference

Used to extend one or two connected search trees.

```
#include <Extender.h>
```

Public Member Functions

- virtual void extend (const std::size_t &Iterations)=0

  Perform the specified number of estensions on the wrapped tree(s). This function may be called multiple times, for performing batch of extensions. All the solutions found while extending are saved and stored in this object.
- std::size_t getIterationsDone () const

  Get the extensions done so far.
- const std::set< Solution > & getSolutions () const

  Get the collection solutions found.
- std::vector< NodeState > computeBestSolutionSequence () const
- bool **isCumulating** () const
- virtual std::vector< NodeState > **computeSolutionSequence** (const Solution &sol) const =0

Static Public Member Functions

- template< typename ExtT >
  static std::vector< NodeState > computeBestSolutionSequence (const std::vector< ExtT > extenders)

Protected Member Functions

- **Extender** (const bool &cumulateSolutions, const double &deterministicCoefficient)

Protected Attributes

- sampling::UniformEngine **randEngine**
- const bool cumulateSolutions

  when set true, the extension process is not arrested when a first solution is found
- const double **deterministicCoefficient**
- std::size_t **iterationsDone** = 0
- std::set< Solution > **solutionsFound**

8.5.1 Detailed Description

template<typename Solution>
class mt::Extender< Solution >

Used to extend one or two connected search trees.

8.5.2 Member Function Documentation

8.5.2.1 computeBestSolutionSequence() [1/2]

```
template<typename Solution >
std::vector<NodeState> mt::Extender< Solution >::computeBestSolutionSequence ( ) const [inline]
```

Returns

the sequence of states pertaining to the best solution found. In case no solution were found at all, an empty vector is returned

8.5.2.2 computeBestSolutionSequence() [2/2]

```
template<typename Solution >
template<typename ExtT >
static std::vector<NodeState> mt::Extender< Solution >::computeBestSolutionSequence (
            const std::vector< ExtT > extenders ) [inline], [static]
```

Returns

the sequence of states pertaining to the best solution found, among all the ones stored in all the passed extenders

8.5.2.3 extend()

```
template<typename Solution >
virtual void mt::Extender< Solution >::extend (
            const std::size_t & Iterations ) [pure virtual]
```

Perform the specified number of estensions on the wrapped tree(s). This function may be called multiple times, for performing batch of extensions. All the solutions found while extending are saved and stored in this object.

Parameters

the	number of extension to perform

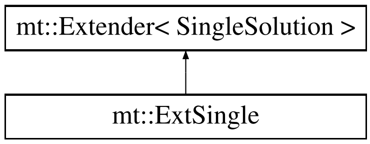
Implemented in mt::ExtBidir, and mt::ExtSingle.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/extn/header/Extender.h

8.6 mt::ExtSingle Class Reference

Inheritance diagram for mt::ExtSingle:



Public Member Functions

- **ExtSingle** (const bool &cumulateSolutions, const double &deterministicCoefficient, Tree &tree, const NodeState &target)
- void extend (const std::size_t &Iterations) override

  Perform the specified number of estensions on the wrapped tree(s). This function may be called multiple times, for performing batch of extensions. All the solutions found while extending are saved and stored in this object.
- std::vector< NodeState > **computeSolutionSequence** (const SingleSolution &sol) const override

Additional Inherited Members

8.6.1 Member Function Documentation

8.6.1.1 extend()

```
void mt::ExtSingle::extend (
            const std::size_t & Iterations )  [override], [virtual]
```

Perform the specified number of estensions on the wrapped tree(s). This function may be called multiple times, for performing batch of extensions. All the solutions found while extending are saved and stored in this object.

Parameters

the	number of extension to perform

Implements mt::Extender< SingleSolution >.

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

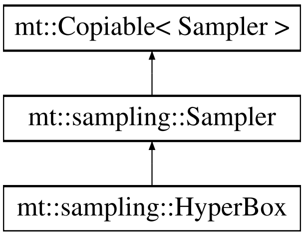
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/extn/header/ExtenderSingle.h

8.7 mt::sampling::HyperBox Class Reference

A sampler drawing a sample inside an hypercube of n-dimensions, described by 2 corners. For example, corners [l1, l2, l3, l4] and [u1, u2, u3, u4], describe an hyperbox whose points [x1,x2,x3,x4] are all such that: li <= xi <= ui

```
#include <HyperBox.h>
```

Inheritance diagram for mt::sampling::HyperBox:



Public Member Functions

- HyperBox (const NodeState lowerCorner, const NodeState upperCorner)
- std::unique_ptr< Sampler > **copy** () const override
- NodeState randomState () const override

  Returns a node having a state randomly sampled in the \mathcal{X} space, Section METTERE of the documentation. This function is invoked mainly for randomly growing a searching tree.
- const NodeState & **getLowerLimit** () const
- const NodeState & **getDeltaLimit** () const

8.7.1 Detailed Description

A sampler drawing a sample inside an hypercube of n-dimensions, described by 2 corners. For example, corners [l1, l2, l3, l4] and [u1, u2, u3, u4], describe an hyperbox whose points [x1,x2,x3,x4] are all such that: li <= xi <= ui

8.7.2 Constructor & Destructor Documentation

8.7.2.1 HyperBox()

```
mt::sampling::HyperBox::HyperBox (
            const NodeState lowerCorner,
            const NodeState upperCorner )
```

Parameters

the	lower corner of the hyperbox
the	upper corner of the hyperbox

Exceptions

if	lowerCorner and upperCorner size mismatch or some of the values inside lowerCorner are greater than ones in upperCorner

8.7.3 Member Function Documentation

8.7.3.1 randomState()

```
NodeState mt::sampling::HyperBox::randomState ( ) const  [override], [virtual]
```

Returns a node having a state randomly sampled in the \mathcal{X} space, Section METTERE of the documentation. This function is invoked mainly for randomly growing a searching tree.

Returns

a drawn random state.

Implements mt::sampling::Sampler.

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

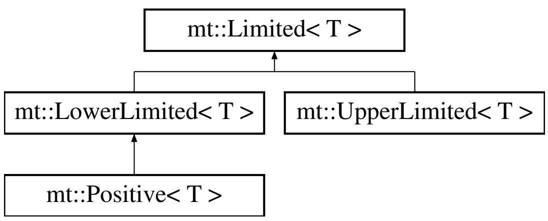
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/sampler/HyperBox.h

8.8 mt::Limited< T > Class Template Reference

A numeric quantity whose value should always remain between defined bounds.

```
#include <Limited.h>
```

Inheritance diagram for mt::Limited< T >:



Public Member Functions

- Limited (const T &lowerBound, const T &upperBound, const T &initialValue)
- Limited (const T &lowerBound, const T &upperBound)

  similar to Limited::Limited(const T& lowerBound, const T& upperBound, const T& initialValue), assuming lower↩ Bound as initial value
- **Limited** (const Limited &)=default
- Limited & **operator=** (const Limited &)=default
- const T & **getLowerBound** () const
- const T & **getUpperBound** () const
- T get () const
- void set (const T &newValue)

Protected Attributes

- T **value**
- const T **lowerBound**
- const T **upperBound**

8.8.1 Detailed Description

template<typename T>
class mt::Limited< T >

A numeric quantity whose value should always remain between defined bounds.

8.8.2 Constructor & Destructor Documentation

8.8.2.1 Limited()

```
template<typename T >
mt::Limited< T >::Limited (
            const T & lowerBound,
            const T & upperBound,
            const T & initialValue )  [inline]
```

Parameters

lower	bound for the value
upper	bound for the value
initial	value to set

8.8.3 Member Function Documentation

8.8.3.1 get()

```
template<typename T >
T mt::Limited< T >::get ( ) const  [inline]
```

Returns

the current value

8.8.3.2 set()

```
template<typename T >
void mt::Limited< T >::set (
            const T & newValue )  [inline]
```

Parameters

the	new value to assumed

Exceptions

if	the value is not consistent with the bounds

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

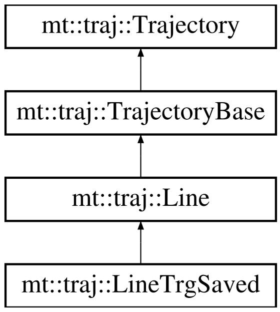
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Limited.h

8.9 mt::traj::Line Class Reference

Advances along a segment in the state space, traversing everytime a distance not higher than steerDegree.

```
#include <Line.h>
```

Inheritance diagram for mt::traj::Line:



Public Member Functions

- **Line** (const NodeState &start, const NodeState &target, const float &steerDegree)
- NodeState getCursor () const override

Protected Member Functions

- AdvanceInfo **advanceInternal** () override

Protected Attributes

- const NodeState & **target**
- const float **steerDegree**
- NodeState **cursor**
- NodeState **previousState**

8.9.1 Detailed Description

Advances along a segment in the state space, traversing everytime a distance not higher than steerDegree.

8.9.2 Member Function Documentation

8.9.2.1 getCursor()

```
NodeState mt::traj::Line::getCursor ( ) const  [inline], [override], [virtual]
```

Returns

the current state of the cursor. IMPORTANT: it is a no-sense value in case last advance() returned blocked

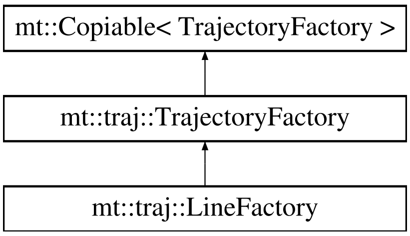
Implements mt::traj::Trajectory.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/Line.h

8.10 mt::traj::LineFactory Class Reference

Inheritance diagram for mt::traj::LineFactory:



Protected Member Functions

- **LineFactory** (const float &steerDegree)
- float **cost2GoIgnoringConstraints** (const NodeState &start, const NodeState &ending_node) const override

Protected Attributes

- const float **steerDegree**

Additional Inherited Members

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

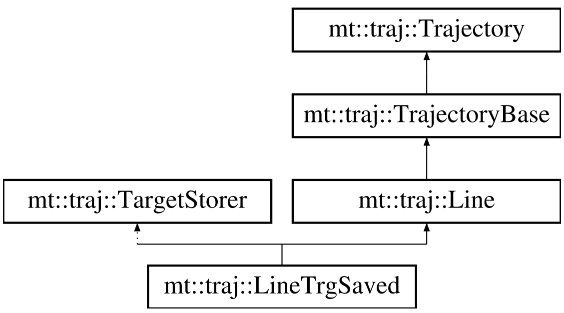
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/Line.h

8.11 mt::traj::LineTrgSaved Class Reference

Internally saves the target state, in order for the const refernce stored in Line to remain meaningful.

```
#include <LineTrgSaved.h>
```

Inheritance diagram for mt::traj::LineTrgSaved:



Public Member Functions

- **LineTrgSaved** (const NodeState &start, const NodeState &target, const float &steerDegree)

Additional Inherited Members

8.11.1 Detailed Description

Internally saves the target state, in order for the const refernce stored in Line to remain meaningful.

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

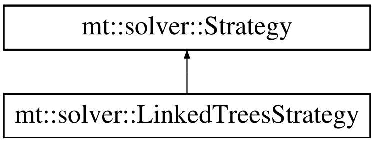
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/LineTrgSaved.h

8.12 mt::solver::LinkedTreesStrategy Class Reference

strategy described in METTERE

```
#include <LinkedTreesStrategy.h>
```

Inheritance diagram for mt::solver::LinkedTreesStrategy:



Public Member Functions

- std::unique_ptr< SolutionInfo > solve (const NodeState &start, const NodeState &end, const RRTStrategy &rrtStrategy) final
  solve a planning problem
- Limited< double > & **getIterationsMax** ()

Additional Inherited Members

8.12.1 Detailed Description

strategy described in METTERE

8.12.2 Member Function Documentation

8.12.2.1 solve()

```
std::unique_ptr<SolutionInfo> mt::solver::LinkedTreesStrategy::solve (
            const NodeState & start,
            const NodeState & end,
            const RRTStrategy & rrtStrategy )  [final], [virtual]
```

solve a planning problem

Parameters

the	starting state
the	ending state
the	rrt strategy to use

Implements mt::solver::Strategy.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/strategies/LinkedTreesStrategy.h

8.13 mt::solver::linked::ListLinked< T > Class Template Reference

Public Member Functions

- **ListLinked** (const ListLinked< T > &)=delete
- ListLinked & **operator=** (const ListLinked< T > &)=delete

Static Public Member Functions

- static void **link** (std::vector< ListLinked< T > ∗ > &group)

Protected Types

- typedef std::shared_ptr< std::list< T > > **shared_buffer**

Protected Member Functions

- template<typename Action >
  void **gatherResult** (const Action &action)

Protected Attributes

- std::vector< shared_buffer > **incomings**
- std::vector< shared_buffer > **outgoings**

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

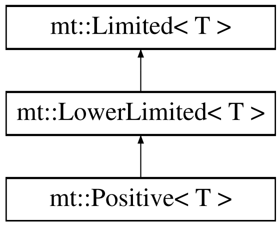
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/linkedTrees/header/ListLinked.h

8.14 mt::LowerLimited< T > Class Template Reference

A @Limited quantity, having +infinite as upper bound.

```
#include <Limited.h>
```

Inheritance diagram for mt::LowerLimited< T >:



Public Member Functions

- **LowerLimited** (const T &lowerBound, const T &initialValue)
- **LowerLimited** (const T &lowerBound)

Additional Inherited Members

8.14.1 Detailed Description

template<typename T>
class mt::LowerLimited< T >

A @Limited quantity, having +infinite as upper bound.

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

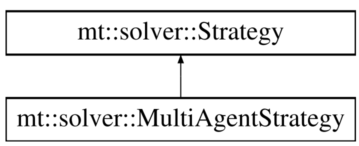
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Limited.h

8.15 mt::solver::MultiAgentStrategy Class Reference

strategy described in METTERE

```
#include <MultiAgentStrategy.h>
```

Inheritance diagram for mt::solver::MultiAgentStrategy:



Public Member Functions

- std::unique_ptr< SolutionInfo > solve (const NodeState &start, const NodeState &end, const RRTStrategy &rrtStrategy) final
- Limited< double > & **getIterationsMax** ()

Additional Inherited Members

8.15.1 Detailed Description

strategy described in METTERE

8.15.2 Member Function Documentation

8.15.2.1 solve()

```
std::unique_ptr<SolutionInfo> mt::solver::MultiAgentStrategy::solve (
            const NodeState & start,
            const NodeState & end,
            const RRTStrategy & rrtStrategy )  [final], [virtual]
```

Exceptions

passing	Bidir as rrtStrategy

Implements mt::solver::Strategy.

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

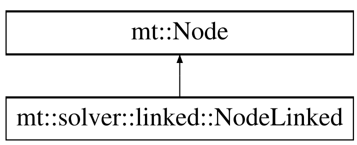
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/strategies/MultiAgentStrategy.h

8.16 mt::Node Class Reference

Used for representing a state x \in \underline{\mathcal{X}}, Section METTERE of the documentation.

```
#include <Node.h>
```

Inheritance diagram for mt::Node:



Public Member Functions

- Node (const NodeState &state)
- **Node** (const Node &)=delete
- Node & **operator=** (const Node &)=delete
- **Node** (Node &&)=delete
- Node & **operator=** (Node &&)=delete
- float cost2Root () const
- const float & getCostFromFather () const
- const NodeState & getState () const
- Node ∗ getFather () const
- void setFather (Node ∗new_father, const float &cost_from_father)

    Connect this node to the new one passed as input.

8.16.1 Detailed Description

Used for representing a state x \in \underline{\mathcal{X}}, Section METTERE of the documentation.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 Node()

```
mt::Node::Node (
            const NodeState & state )
```

Parameters

the	values inside the vector respresenting this state

Exceptions

when	passing an empty state

8.16.3 Member Function Documentation

8.16.3.1 cost2Root()

```
float mt::Node::cost2Root ( ) const
```

Returns

Computes the cost to get from the root to this node, see METTERE.

Exceptions

when	the root is not reached, cause loopy connections were made

8.16.3.2 getCostFromFather()

```
const float& mt::Node::getCostFromFather ( ) const [inline]
```

Returns

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

8.16.3.3 getFather()

```
Node* mt::Node::getFather ( ) const [inline]
```

Returns

the node to reach before this one, in the path connecting the root to this node. Returns nullptr for the root

8.16.3.4 getState()

```
const NodeState& mt::Node::getState ( ) const [inline]
```

Returns

the state describing this node

8.16.3.5 setFather()

```
void mt::Node::setFather (
            Node * new_father,
            const float & cost_from_father )
```

Connect this node to the new one passed as input.

Parameters

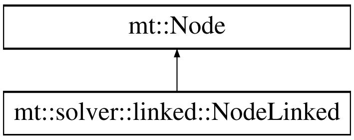
the	node to assume as new father
the	cost to go from the new father to set

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Node.h

8.17 mt::solver::linked::NodeLinked Class Reference

Inheritance diagram for mt::solver::linked::NodeLinked:



Public Member Functions

- const std::vector< NodeLinked * > & **getLinked** () const

Static Public Member Functions

- static std::vector< std::unique_ptr< NodeLinked > > **make_roots** (Node &node, const std::size_t &threadsNumber)
- static std::vector< std::unique_ptr< NodeLinked > > **make_linked** (Node &node)

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/linkedTrees/header/NodeLinked.h

8.18 mt::solver::Parameters Struct Reference

Parameters used to solve a planning problem.

`#include <Strategy.h>`

Public Attributes

- bool Cumulate_sol = false

  Don't stop exploring process after a solution is found.
- Limited< double > Deterministic_coefficient = Limited<double>(0.01, 0.99, 0.2)

  Regulates the determinism used to get a solution, refer to METTERE.
- LowerLimited< std::size_t > Iterations_Max = LowerLimited<std::size_t>(10, 1000)

  the maximal number of iterations to use trying to find a solution

8.18.1 Detailed Description

Parameters used to solve a planning problem.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/solver/Strategy.h

8.19 mt::solver::qpar::Pool Class Reference

Public Member Functions

- void **open** (const std::size_t &size)
- void **close** ()
- void **addJob** (const std::vector< Job > &jobs)
- void **wait** ()

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

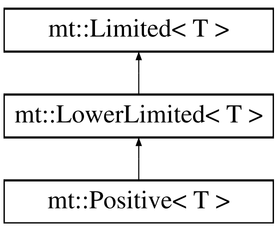
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/queryParall/header/Pool.h

8.20 mt::Positive< T > Class Template Reference

A @LowerLimited quantity, having 0.0 as lower bound.

```
#include <Limited.h>
```

Inheritance diagram for mt::Positive< T >:



Public Member Functions

- **Positive** (const T &initialValue=static_cast< T >(0))
- Positive & **operator=** (const Positive &o)

Additional Inherited Members

8.20.1 Detailed Description

template<typename T>
class mt::Positive< T >

A @LowerLimited quantity, having 0.0 as lower bound.

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

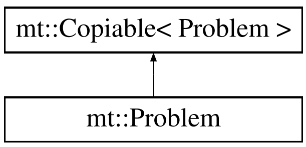
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Limited.h

8.21 mt::Problem Class Reference

Object storing the information needed to extend exploring trees.

```
#include <Problem.h>
```

Inheritance diagram for mt::Problem:



Public Member Functions

- Problem (sampling::SamplerPtr sampler, traj::TrajectoryFactoryPtr manager, const std::size_t &stateSpaceSize, const float &gamma, const bool &simmetry=true)
- Problem & **operator=** (const Problem &)=delete
- std::unique_ptr< Problem > copy () const final

  Used by @Solver: each working thread use its private Problem copy.
- NodePtr steer (Node &start, const NodeState &trg, bool &trg_reached)

  Performs a steering operation, Section METTERE of the documentation, from a staring node to a target one.
- void setSteerTrials (const std::size_t &trials)

  Sets the steering trials used when extending searching trees, refer to METTERE.
- std::size_t getProblemSize () const

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

  Returns the \gamma parameter, Section METTERE of the documentation, regulating the near set size, that RRT∗ versions must compute.
- bool isProblemSimmetric () 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.
- sampling::Sampler ∗ getSampler () const
- traj::TrajectoryFactory ∗ getTrajManager () const

Protected Member Functions

- **Problem** (const Problem &o)

Protected Attributes

- const LowerLimited< std::size_t > **stateSpaceSize**
- const Positive< float > **gamma**
- const bool **simmetry**
- LowerLimited< std::size_t > **steerTrials** = LowerLimited<std::size_t>(1,1)
- sampling::SamplerPtr **sampler**
- traj::TrajectoryFactoryPtr **trajManager**

8.21.1 Detailed Description

Object storing the information needed to extend exploring trees.

8.21.2 Constructor & Destructor Documentation

8.21.2.1 Problem()

```
mt::Problem::Problem (
            sampling::SamplerPtr sampler,
            traj::TrajectoryFactoryPtr manager,
            const std::size_t & stateSpaceSize,
            const float & gamma,
            const bool & simmetry = true )
```

Parameters

the	sampler to steal
the	trajectory factory to steal
the	dimension of the state space of the problem to solve. Refer to METTERE
the	parameters described in METTERE
true	when the problem is simmetric. Refer to METTERE

Exceptions

if	sampler or manager are nullptr

8.21.3 Member Function Documentation

8.21.3.1 getSampler()

```
sampling::Sampler* mt::Problem::getSampler ( ) const [inline]
```

Returns

the stored sampler

8.21.3.2 getTrajManager()

```
traj::TrajectoryFactory* mt::Problem::getTrajManager ( ) const [inline]
```

Returns

the stored trajectory factory

8.21.3.3 steer()

```
NodePtr mt::Problem::steer (
            Node & start,
            const NodeState & trg,
            bool & trg_reached )
```

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

Parameters

starting	configuration
target	configuration to reach
set	true from the inside of this function, when the target was reached after steering

Returns

the steered configuration. Is a nullptr when the steering was not possible at all

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Problem.h

8.22 mt::solver::qpar::Query Class Reference

Protected Member Functions

- **Query** (Problem &problem, const TreeIterator &iterator)

Protected Attributes

- Problem & **problem**
- TreeIterator **iterator**

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

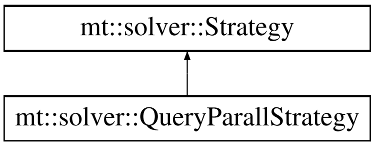
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/queryParall/header/Query.h

8.23 mt::solver::QueryParallStrategy Class Reference

strategy described in METTERE

```
#include <QueryParallStrategy.h>
```

Inheritance diagram for mt::solver::QueryParallStrategy:



Public Member Functions

- std::unique_ptr< SolutionInfo > solve (const NodeState &start, const NodeState &end, const RRTStrategy &rrtStrategy) final

    solve a planning problem

Additional Inherited Members

8.23.1 Detailed Description

strategy described in METTERE

8.23.2 Member Function Documentation

8.23.2.1 solve()

```
std::unique_ptr<SolutionInfo> mt::solver::QueryParallStrategy::solve (
            const NodeState & start,
            const NodeState & end,
            const RRTStrategy & rrtStrategy )  [final], [virtual]
```

solve a planning problem

Parameters

the	starting state
the	ending state
the	rrt strategy to use

Implements mt::solver::Strategy.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/strategies/QueryParallStrategy.h

8.24 mt::Rewire Struct Reference

newFather should be set as father node for involved, with a cost to go equal to newCostFromFather

```
#include <TreeRewirer.h>
```

Public Member Functions

- **Rewire** (Node &involved, Node &newFather, const float &newCostFromFather)

Public Attributes

- Node & **involved**
- Node & **newFather**
- float **newCostFromFather**

8.24.1 Detailed Description

newFather should be set as father node for involved, with a cost to go equal to newCostFromFather

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

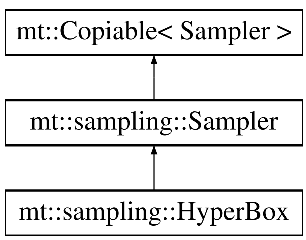
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/tree/header/TreeRewirer.h

8.25 mt::sampling::Sampler Class Reference

Interface for a sampler of states.

```
#include <Sampler.h>
```

Inheritance diagram for mt::sampling::Sampler:



Public Member Functions

- virtual NodeState randomState () const =0
  Returns a node having a state randomly sampled in the \mathcal{X} space, Section METTERE of the documentation. This function is invoked mainly for randomly growing a searching tree.

8.25.1 Detailed Description

Interface for a sampler of states.

8.25.2 Member Function Documentation

8.25.2.1 randomState()

```
virtual NodeState mt::sampling::Sampler::randomState ( ) const  [pure virtual]
```

Returns a node having a state randomly sampled in the \mathcal{X} space, Section METTERE of the documentation. This function is invoked mainly for randomly growing a searching tree.

Returns
  a drawn random state.

Implemented in mt::sampling::HyperBox.

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

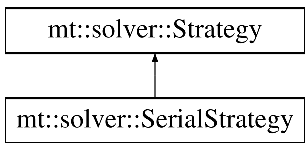
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/sampler/Sampler.h

8.26 mt::solver::SerialStrategy Class Reference

The standard serial strategy described in METTERE.

```
#include <SerialStrategy.h>
```

Inheritance diagram for mt::solver::SerialStrategy:



Public Member Functions

- std::unique_ptr< SolutionInfo > solve (const NodeState &start, const NodeState &end, const RRTStrategy &rrtStrategy) final

  solve a planning problem

Additional Inherited Members

8.26.1 Detailed Description

The standard serial strategy described in METTERE.

8.26.2 Member Function Documentation

8.26.2.1 solve()

```
std::unique_ptr<SolutionInfo> mt::solver::SerialStrategy::solve (
            const NodeState & start,
            const NodeState & end,
            const RRTStrategy & rrtStrategy )  [final], [virtual]
```

solve a planning problem

Parameters

the	starting state
the	ending state
the	rrt strategy to use

Implements mt::solver::Strategy.

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

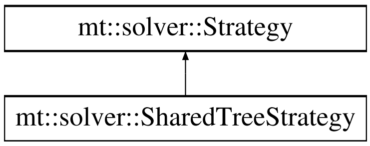
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/strategies/SerialStrategy.h

8.27 mt::solver::SharedTreeStrategy Class Reference

strategy described in METTERE

`#include <SharedTreeStrategy.h>`

Inheritance diagram for mt::solver::SharedTreeStrategy:



Public Member Functions

- std::unique_ptr< SolutionInfo > solve (const NodeState &start, const NodeState &end, const RRTStrategy &rrtStrategy) final

    solve a planning problem

Additional Inherited Members

8.27.1 Detailed Description

strategy described in METTERE

8.27.2 Member Function Documentation

8.27.2.1 solve()

```
std::unique_ptr<SolutionInfo> mt::solver::SharedTreeStrategy::solve (
            const NodeState & start,
            const NodeState & end,
            const RRTStrategy & rrtStrategy )  [final], [virtual]
```

solve a planning problem

Parameters

the	starting state
the	ending state
the	rrt strategy to use

Implements mt::solver::Strategy.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/strategies/SharedTreeStrategy.h

8.28 mt::solver::SolutionInfo Struct Reference

Is produced internally to @Solver, every time a new planning problem is solved. The various quantity can be then accessed using the getters of @Solver.

```
#include <Solver.h>
```

Public Member Functions

- **SolutionInfo** (const NodeState &start, const NodeState &target)

Public Attributes

- const NodeState **start**
- const NodeState **target**
- std::chrono::milliseconds time = std::chrono::milliseconds(0)

  elapsed time
- std::size_t iterations = 0

  iterations spent
- std::vector< NodeState > solution

  The sequence of states forming the solution to the planning problem. Is an empty vector in case a solution was not found.
- std::vector< TreePtr > trees

  The trees extended and used in order to solve the problem.

8.28.1 Detailed Description

Is produced internally to @Solver, every time a new planning problem is solved. The various quantity can be then accessed using the getters of @Solver.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/solver/Solver.h

8.29 mt::solver::Solver Class Reference

Solver storing results of planning problem. Every time solve(...) is called, a new problem is solved and the results can be accessed using the getters provided in this interface. When another problem is solved calling again solve(...), the information regarding the previous problem are lost.

```
#include <Solver.h>
```

Public Member Functions

- **Solver** (const Solver &)=delete
- Solver & **operator=** (const Solver &)=delete
- Solver (ProblemPtr problemDescription)
- Solver (ProblemPtr problemDescription, std::unique_ptr< Strategy > solverStrategy)
- void setStrategy (std::unique_ptr< Strategy > solverStrategy)
- std::unique_ptr< Strategy > extractStrategy ()
- void solve (const NodeState &start, const NodeState &end, const RRTStrategy &rrtStrategy)

  Tries to solve a new plannig problem, using the previously set strategy. Information regarding the solution(s) found are stored inside this object.
- void setSteerTrials (const std::size_t &trials)
- void setThreadAvailability (const std::size_t &threads=0)
- std::size_t getLastIterations () const
- std::chrono::milliseconds getLastElapsedTime () const
- NodeState getLastStart () const
- NodeState getLastTarget () const
- std::vector< NodeState > copyLastSolution () const
- std::vector< TreePtrConst > extractLastTrees ()
- void **saveTreesAfterSolve** ()
- void **discardTreesAfterSolve** ()
- std::size_t getThreadAvailability () const
- template<typename User >
  void useProblem (const User &user)

  Use the problem stored inside this solver for some external purpose.

8.29.1 Detailed Description

Solver storing results of planning problem. Every time solve(...) is called, a new problem is solved and the results can be accessed using the getters provided in this interface. When another problem is solved calling again solve(...), the information regarding the previous problem are lost.

8.29.2 Constructor & Destructor Documentation

8.29.2.1 Solver() [1/2]

```
mt::solver::Solver::Solver (
            ProblemPtr problemDescription )
```

Parameters

the	problem description, i.e. the obect consumed by the solver to extend exploring trees and find solution(s)

Exceptions

passing	nullptr as problemDescription

8.29.2.2 Solver() [2/2]

```
mt::solver::Solver::Solver (
            ProblemPtr problemDescription,
            std::unique_ptr< Strategy > solverStrategy )
```

Parameters

the	problem description, i.e. the obect consumed by the solver to extend exploring trees and find solution(s)
the	solving strategy to use for the following planning problems to solve (same as building the object with no strategy and then call setStrategy(...))

Exceptions

passing	nullptr as problemDescription

8.29.3 Member Function Documentation

8.29.3.1 copyLastSolution()

```
std::vector<NodeState> mt::solver::Solver::copyLastSolution ( ) const
```

Returns

a copy of the solution, i.e. sequence of states, to the last solved planning problem.

8.29.3.2 extractLastTrees()

```
std::vector<TreePtrConst> mt::solver::Solver::extractLastTrees ( )
```

Returns

extracts the trees obtained for solving the last problem. The trees are moved out and therefore a subsequent call to this method would return an empty vector. By default, the trees are NOT saved in order to save memory space. However you can specify to the solver to save the trees by calling saveTreesAfterSolve(). You can disable trees saving calling discardTreesAfterSolve()

8.29.3.3 extractStrategy()

```
std::unique_ptr<Strategy> mt::solver::Solver::extractStrategy ( )
```

Parameters

remove	the solving strategy stored in the solver. Useful to externally manipulate the strategy object (setting parameters for example) and then re-assing it using setStrategy(...)

8.29.3.4 getLastElapsedTime()

```
std::chrono::milliseconds mt::solver::Solver::getLastElapsedTime ( ) const
```

Returns

the time required by the last planification

8.29.3.5 getLastIterations()

```
std::size_t mt::solver::Solver::getLastIterations ( ) const
```

Returns

the number of iterations required by the last planification

8.29.3.6 getLastStart()

```
NodeState mt::solver::Solver::getLastStart ( ) const
```

Returns

the starting configuration of the last solved problem

8.29.3.7 getLastTarget()

```
NodeState mt::solver::Solver::getLastTarget ( ) const
```

Returns

the target configuration of the last solved problem

8.29.3.8 getThreadAvailability()

```
std::size_t mt::solver::Solver::getThreadAvailability ( ) const
```

Returns

The number of threads that will be used when adopting a multi-threaded Strategy

8.29.3.9 setSteerTrials()

```
void mt::solver::Solver::setSteerTrials (
            const std::size_t & trials )
```

Parameters

regulates	the number of steering trials, see METTERE, to use for following plans

8.29.3.10 setStrategy()

```
void mt::solver::Solver::setStrategy (
            std::unique_ptr< Strategy > solverStrategy )
```

Parameters

the	set the solving strategy to use for the following planning problems

8.29.3.11 setThreadAvailability()

```
void mt::solver::Solver::setThreadAvailability (
            const std::size_t & threads = 0 )
```

Parameters

the	threads to use for solving future problems, when using a multi-threading Strategy. pass 0 to assume the number of cores.

8.29.3.12 solve()

```
void mt::solver::Solver::solve (
            const NodeState & start,
            const NodeState & end,
            const RRTStrategy & rrtStrategy )
```

Tries to solve a new plannig problem, using the previously set strategy. Information regarding the solution(s) found are stored inside this object.

Parameters

the	staring state of the problem to solve
the	ending state of the problem to solve
the	rrt strategy to use

Exceptions

if	no @Strategy was set before calling this method.
if	size of start is inconsistent
if	size of end is inconsistent
passing	Bidir for rrtStrategy for a problem that is not symmetric, see METTERE

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/solver/Solver.h

8.30 mt::solver::SolverData Struct Reference

Public Attributes

- std::mutex **solverMutex**
- std::vector< ProblemPtr > problemsBattery

  Each working thread should use one of the element in this battery.
- bool saveComputedTrees = false

  The obtained tree(s) are saved after solving a planning problem, in case this parameter is set true. Otherwise they are deleted in order to save memory space.

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

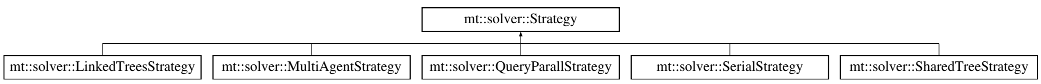
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/solver/Solver.h

8.31 mt::solver::Strategy Class Reference

An interface for an object in charge of solving a plannig problem.

```
#include <Strategy.h>
```

Inheritance diagram for mt::solver::Strategy:



Public Member Functions

- **Strategy** (const Strategy &)=delete
- Strategy & **operator=** (const Strategy &)=delete
- virtual std::unique_ptr< SolutionInfo > solve (const NodeState &start, const NodeState &end, const RRTStrategy &rrtStrategy)=0

  solve a planning problem
- void **shareSolverData** (std::shared_ptr< SolverData > solverData)
- void **forgetSolverData** ()
- bool **getCumulateFlag** () const
- void **setCumulateFlag** (bool flag)
- Limited< double > & **getDeterministicCoefficient** ()
- LowerLimited< std::size_t > & **getIterationsMax** ()

Protected Attributes

- Parameters **parameters**
- std::shared_ptr< SolverData > **solverData**

8.31.1 Detailed Description

An interface for an object in charge of solving a plannig problem.

8.31.2 Member Function Documentation

8.31.2.1 solve()

```
virtual std::unique_ptr<SolutionInfo> mt::solver::Strategy::solve (
            const NodeState & start,
            const NodeState & end,
            const RRTStrategy & rrtStrategy )  [pure virtual]
```

solve a planning problem

Parameters

the	starting state
the	ending state
the	rrt strategy to use

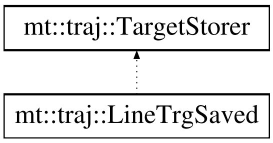
Implemented in mt::solver::MultiAgentStrategy, mt::solver::LinkedTreesStrategy, mt::solver::QueryParallStrategy, mt::solver::SerialStrategy, and mt::solver::SharedTreeStrategy.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/solver/Strategy.h

8.32 mt::traj::TargetStorer Class Reference

Inheritance diagram for mt::traj::TargetStorer:



Protected Member Functions

- **TargetStorer** (const NodeState &target)

Protected Attributes

- const NodeState **targetStored**

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

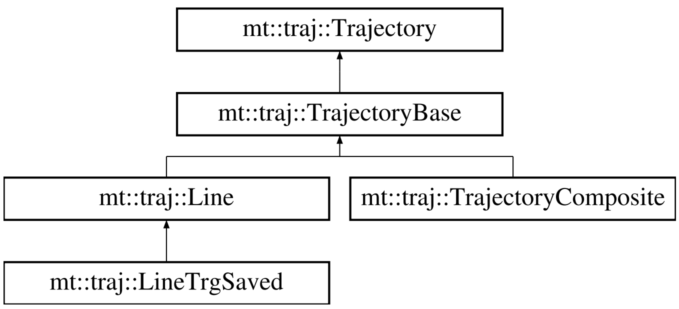
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/LineTrgSaved.h

8.33 mt::traj::Trajectory Class Reference

Interface describing an optimal trajectory connecting 2 states, in a particular problem to solve. Refer to METTERE. A cursor internally stored the state currently reached. When avancing this object, the cursor is modified in order to traverse the trajectory.

`#include <Trajectory.h>`

Inheritance diagram for mt::traj::Trajectory:



Public Member Functions

- **Trajectory** (const Trajectory &)=delete
- Trajectory & **operator=** (const Trajectory &)=delete
- virtual AdvanceInfo advance ()=0

  Move the internal cursor along the trajectory.
- virtual NodeState getCursor () const =0
- virtual float getCumulatedCost () const =0

8.33.1 Detailed Description

Interface describing an optimal trajectory connecting 2 states, in a particular problem to solve. Refer to METTERE. A cursor internally stored the state currently reached. When avancing this object, the cursor is modified in order to traverse the trajectory.

8.33.2 Member Function Documentation

8.33.2.1 getCumulatedCost()

```
virtual float mt::traj::Trajectory::getCumulatedCost ( ) const  [pure virtual]
```

Returns

the cost to go from the beginning of the trajectory to the current cursor. IMPORTANT: it is a no-sense value in case last advance() returned blocked

Implemented in mt::traj::TrajectoryBase.

8.33.2.2 getCursor()

```
virtual NodeState mt::traj::Trajectory::getCursor ( ) const  [pure virtual]
```

Returns

the current state of the cursor. IMPORTANT: it is a no-sense value in case last advance() returned blocked

Implemented in mt::traj::Line, and mt::traj::TrajectoryComposite.

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

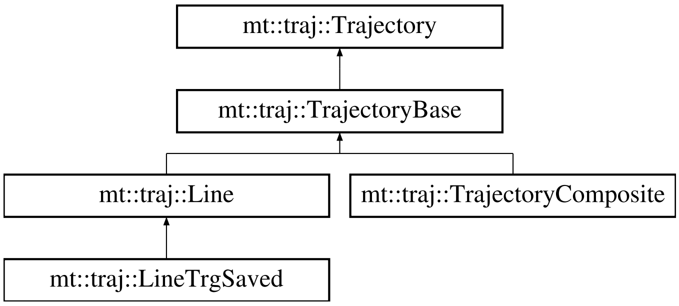
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/Trajectory.h

8.34 mt::traj::TrajectoryBase Class Reference

Base class for a Trajectory. Calling advance a second time, after the first one returned blocked throw an exception.

```
#include <TrajectoryBase.h>
```

Inheritance diagram for mt::traj::TrajectoryBase:



Public Member Functions

- AdvanceInfo advance () final
- float getCumulatedCost () const final

Protected Member Functions

- virtual AdvanceInfo **advanceInternal** ()=0

Protected Attributes

- Cost **cumulatedCost**

8.34.1 Detailed Description

Base class for a Trajectory. Calling advance a second time, after the first one returned blocked throw an exception.

8.34.2 Member Function Documentation

8.34.2.1 advance()

```
AdvanceInfo mt::traj::TrajectoryBase::advance ( )  [final], [virtual]
```

Exceptions

if	a previous call to advance returned blocked

Implements mt::traj::Trajectory.

8.34.2.2 getCumulatedCost()

```
float mt::traj::TrajectoryBase::getCumulatedCost ( ) const  [inline], [final], [virtual]
```

Returns

the cost to go from the beginning of the trajectory to the current cursor. IMPORTANT: it is a no-sense value in case last advance() returned blocked

Implements mt::traj::Trajectory.

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

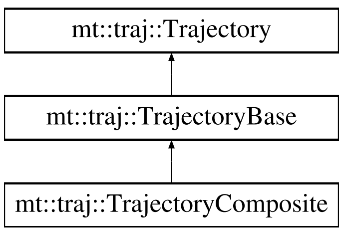
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/TrajectoryBase.h

8.35 mt::traj::TrajectoryComposite Class Reference

Base class for a Trajectory made of pieces of sub-ones.

```
#include <TrajectoryComposite.h>
```

Inheritance diagram for mt::traj::TrajectoryComposite:



Public Member Functions

- NodeState getCursor () const override

Protected Member Functions

- template<typename ... Pieces>
  TrajectoryComposite (Pieces &&... piecess)
- AdvanceInfo **advanceInternal** () override

Protected Attributes

- std::list< TrajectoryPtr > **pieces**
- std::list< TrajectoryPtr >::iterator **piecesCursor**
- Cost **cumulatedCostPrevPieces**

8.35.1 Detailed Description

Base class for a Trajectory made of pieces of sub-ones.

8.35.2 Member Function Documentation

8.35.2.1 getCursor()

```
NodeState mt::traj::TrajectoryComposite::getCursor ( ) const  [inline], [override], [virtual]
```

Returns

the current state of the cursor. IMPORTANT: it is a no-sense value in case last advance() returned blocked

Implements mt::traj::Trajectory.

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

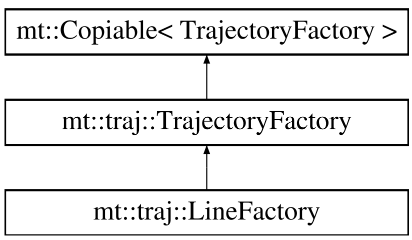
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/TrajectoryComposite.h

8.36 mt::traj::TrajectoryFactory Class Reference

Creator of optimal trajectories, refer to METTERE. Each specific problem to solve need to define and use its specific TrajectoryFactory.

```
#include <TrajectoryFactory.h>
```

Inheritance diagram for mt::traj::TrajectoryFactory:



Public Member Functions

- **TrajectoryFactory** (const TrajectoryFactory &)=delete
- TrajectoryFactory & **operator=** (const TrajectoryFactory &)=delete
- float cost2Go (const NodeState &start, const NodeState &ending_node, const bool &ignore↩ Constraints) const

  Evaluates the cost C(\tau), Section METTERE of the documentation, of the trajectory \tau going from the starting node to the ending one, for two nodes not already connected.
- virtual TrajectoryPtr getTrajectory (const NodeState &start, const NodeState &ending_node) const =0

Protected Member Functions

- virtual float **cost2GoIgnoringConstraints** (const NodeState &start, const NodeState &ending_node) const =0

8.36.1 Detailed Description

Creator of optimal trajectories, refer to METTERE. Each specific problem to solve need to define and use its specific TrajectoryFactory.

8.36.2 Member Function Documentation

8.36.2.1 cost2Go()

```
float mt::traj::TrajectoryFactory::cost2Go (
            const NodeState & start,
            const NodeState & ending_node,
            const bool & ignoreConstraints ) const
```

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

Parameters

the	starting node in the trajectory whose cost is to evaluate
the	ending node in the trajectory whose cost is to evaluate
true	when the constraints, see METTERE, need to be accounted. Cost::COST_MAX is in this case returned, when a feasible trajectory exists, but is not entirely contained in the admitted set METTERE

Returns

the cost to go of the trajectory connecting the states. Cost::COST_MAX is returned when a feasible trajectory does not exist.

8.36.2.2 getTrajectory()

```
virtual TrajectoryPtr mt::traj::TrajectoryFactory::getTrajectory (
            const NodeState & start,
            const NodeState & ending_node ) const [pure virtual]
```

Parameters

the	starting state
the	ending state

Returns

the optimal trajectory connecting the passed states. nullptr is returned in case a feasible trajectory does not exist

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

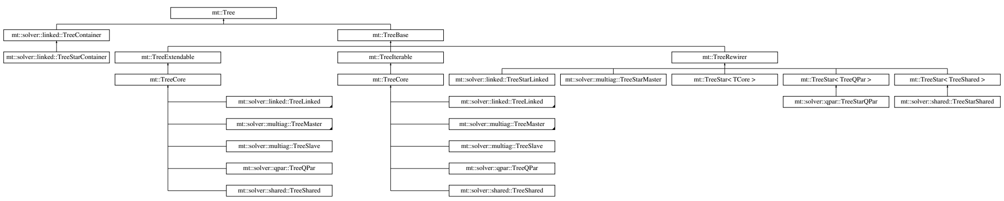
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/trajectory/TrajectoryFactory.h

8.37 mt::Tree Class Reference

Interface for a Nodes container. Minimal functionalties to iterate the container should be implemented in descendants.

`#include <Tree.h>`

Inheritance diagram for mt::Tree:



Public Member Functions

- **Tree** (const Tree &)=delete
- Tree & **operator=** (const Tree &)=delete
- virtual Nodes::const_reverse_iterator rend () const =0
- virtual Nodes::const_reverse_iterator rbegin () const =0
- const Node ∗ front () const

8.37.1 Detailed Description

Interface for a Nodes container. Minimal functionalties to iterate the container should be implemented in descendants.

8.37.2 Member Function Documentation

8.37.2.1 front()

```
const Node* mt::Tree::front ( ) const  [inline]
```

Returns

the first node in the container, i.e. the root of the tree

8.37.2.2 rbegin()

```
virtual Nodes::const_reverse_iterator mt::Tree::rbegin ( ) const  [pure virtual]
```

Returns

an iterator pointing to the first node in the container

Implemented in mt::solver::shared::TreeShared, mt::solver::linked::TreeContainer, and mt::TreeIterable.

8.37.2.3 rend()

```
virtual Nodes::const_reverse_iterator mt::Tree::rend ( ) const  [pure virtual]
```

Returns

an iterator pointing to the last node in the container

Implemented in mt::solver::linked::TreeContainer, and mt::TreeIterable.

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

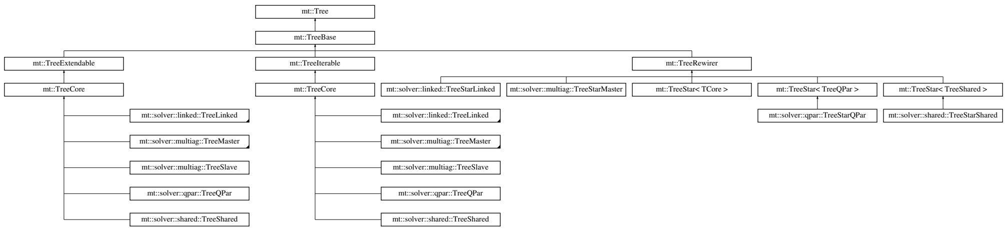
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Tree.h

8.38 mt::TreeBase Class Reference

Base class Tree, storing a problem pointer.

`#include <TreeBase.h>`

Inheritance diagram for mt::TreeBase:



Public Member Functions

- virtual Problem ∗ **getProblem** () const

Protected Attributes

- Problem ∗ problem = nullptr

  The problem description this tree should use.

8.38.1 Detailed Description

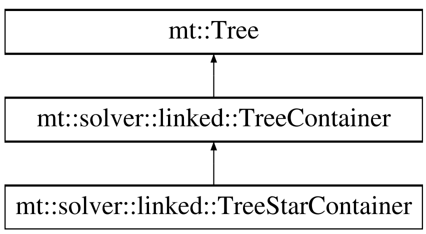
Base class Tree, storing a problem pointer.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/tree/header/TreeBase.h

8.39 mt::solver::linked::TreeContainer Class Reference

Inheritance diagram for mt::solver::linked::TreeContainer:



Public Member Functions

- **TreeContainer** (NodePtr root, const std::vector< ProblemPtr > &problems)
- Nodes::const_reverse_iterator rend () const override
- Nodes::const_reverse_iterator rbegin () const override
- void **gather** ()
- std::size_t **size** () const
- TreeLinked * **getContained** (const std::size_t &pos) const

Protected Attributes

- std::vector< std::unique_ptr< TreeLinked > > **contained**

8.39.1 Member Function Documentation

8.39.1.1 rbegin()

`Nodes::const_reverse_iterator mt::solver::linked::TreeContainer::rbegin ( ) const [inline], [override], [virtual]`

Returns

an iterator pointing to the first node in the container

Implements mt::Tree.

8.39.1.2 rend()

`Nodes::const_reverse_iterator mt::solver::linked::TreeContainer::rend ( ) const [inline], [override], [virtual]`

Returns

an iterator pointing to the last node in the container

Implements mt::Tree.

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

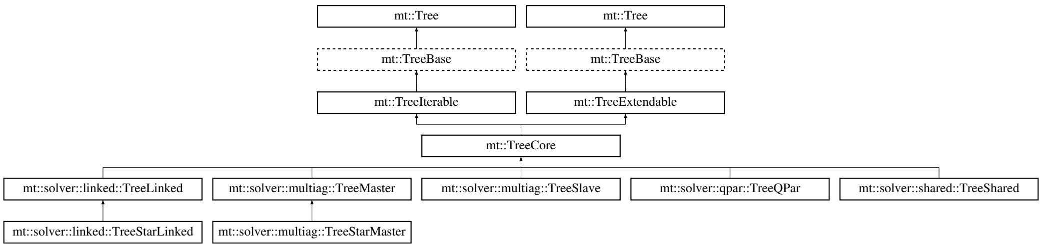
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/linkedTrees/header/TreeContainer.h

8.40 mt::TreeCore Class Reference

Tree with the minimal functionalities required to implement an rrt algorithm.

```
#include <TreeCore.h>
```

Inheritance diagram for mt::TreeCore:



Public Member Functions

- **TreeCore** (NodePtr root, Problem &problem)
- Node ∗ extendRandom ()

  sample a random state using the sampler of the problem and call extend(...) toward the sampled state. In case the extension was possible, the extended node is directly added to the tree and a pointer to it is returned.

Additional Inherited Members

8.40.1 Detailed Description

Tree with the minimal functionalities required to implement an rrt algorithm.

8.40.2 Member Function Documentation

8.40.2.1 extendRandom()

```
Node* mt::TreeCore::extendRandom ( )
```

sample a random state using the sampler of the problem and call extend(...) toward the sampled state. In case the extension was possible, the extended node is directly added to the tree and a pointer to it is returned.

Returns

the added node, in case the extension was possible.

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

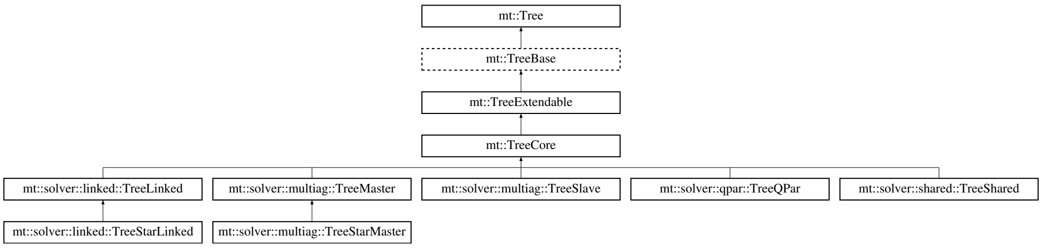
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/tree/header/TreeCore.h

8.41 mt::TreeExtendable Class Reference

Base class for an extendable tree, i.e. a tree whose nodes can be incremented over time.

```
#include <TreeExtendable.h>
```

Inheritance diagram for mt::TreeExtendable:



Public Member Functions

- std::pair< NodePtr, bool > extend (const NodeState &target)
    tried to extend the tree toward the target, see METTERE

Protected Member Functions

- virtual Node * **nearestNeighbour** (const NodeState &state) const

Additional Inherited Members

8.41.1 Detailed Description

Base class for an extendable tree, i.e. a tree whose nodes can be incremented over time.

8.41.2 Member Function Documentation

8.41.2.1 extend()

```
std::pair<NodePtr, bool> mt::TreeExtendable::extend (
            const NodeState & target )
```

tried to extend the tree toward the target, see METTERE

Returns

<a, b>: a: the node containing an extended node, having a father contained in this tree. In case the steering procedure was not possible, a nullptr is returned. b: a boolean that is true in case the extension was possible AND the target was reached. Otherwise is false.

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

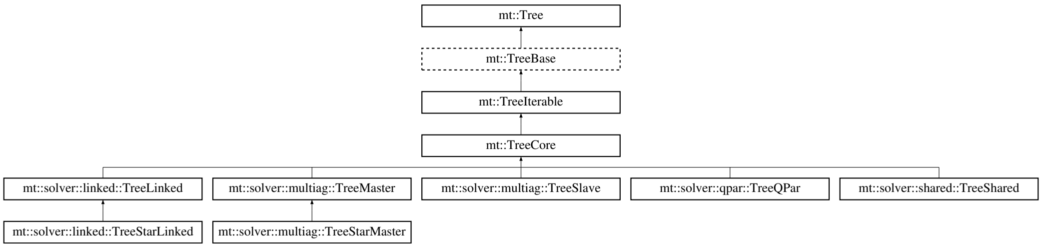
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/tree/header/TreeExtendable.h

8.42 mt::TreeIterable Class Reference

Base Tree class physically storing the nodes.

`#include <TreeIterable.h>`

Inheritance diagram for mt::TreeIterable:



Public Member Functions

- Nodes::const_reverse_iterator rend () const override
- Nodes::const_reverse_iterator rbegin () const override
- virtual Node ∗ add (NodePtr node)

  Add the passed node to the collection.

Protected Member Functions

- **TreeIterable** (NodePtr root)

Protected Attributes

- Nodes nodes

  The collection of nodes.

8.42.1 Detailed Description

Base Tree class physically storing the nodes.

8.42.2 Member Function Documentation

8.42.2.1 add()

```
virtual Node* mt::TreeIterable::add (
            NodePtr node )  [virtual]
```

Add the passed node to the collection.

Parameters

the	node to introduce in the tree

Reimplemented in mt::solver::shared::TreeShared, mt::solver::linked::TreeLinked, mt::solver::linked::TreeStarLinked, and mt::solver::multiag::TreeSlave.

8.42.2.2 rbegin()

```
Nodes::const_reverse_iterator mt::TreeIterable::rbegin ( ) const  [inline], [override], [virtual]
```

Returns

an iterator pointing to the first node in the container

Implements mt::Tree.

Reimplemented in mt::solver::shared::TreeShared.

8.42.2.3 rend()

```
Nodes::const_reverse_iterator mt::TreeIterable::rend ( ) const  [inline], [override], [virtual]
```

Returns

an iterator pointing to the last node in the container

Implements mt::Tree.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/tree/header/TreeIterable.h

8.43 mt::solver::qpar::TreeIterator Class Reference

Public Member Functions

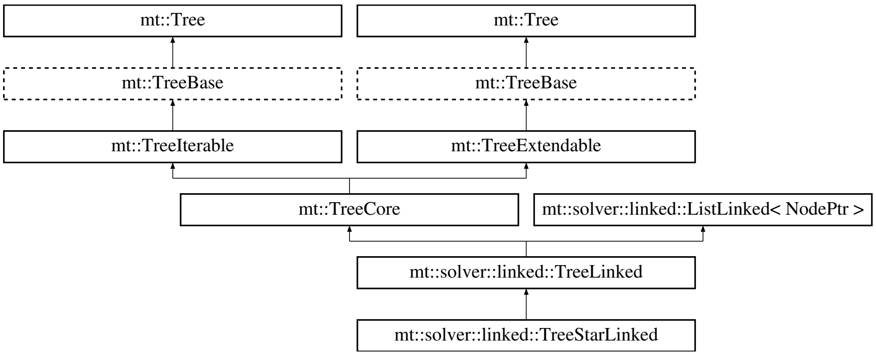
- **TreeIterator** (const mt::Tree &tree, const std::size_t &startPos, const std::size_t &delta)
- TreeIterator & **operator++** ()
- const mt::Nodes::const_reverse_iterator & **get** () const
- const mt::Nodes::const_reverse_iterator & **end** () const

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/queryParall/header/TreeIterator.h

8.44 mt::solver::linked::TreeLinked Class Reference

Inheritance diagram for mt::solver::linked::TreeLinked:



Public Member Functions

- **TreeLinked** (NodePtr root, Problem &problem)
- Node * add (NodePtr node) override

  Add the passed node to the collection.
- virtual void **gather** ()

Additional Inherited Members

8.44.1 Member Function Documentation

8.44.1.1 add()

```
Node* mt::solver::linked::TreeLinked::add (
            NodePtr node )  [override], [virtual]
```

Add the passed node to the collection.

Parameters

the	node to introduce in the tree

Reimplemented from mt::TreeIterable.

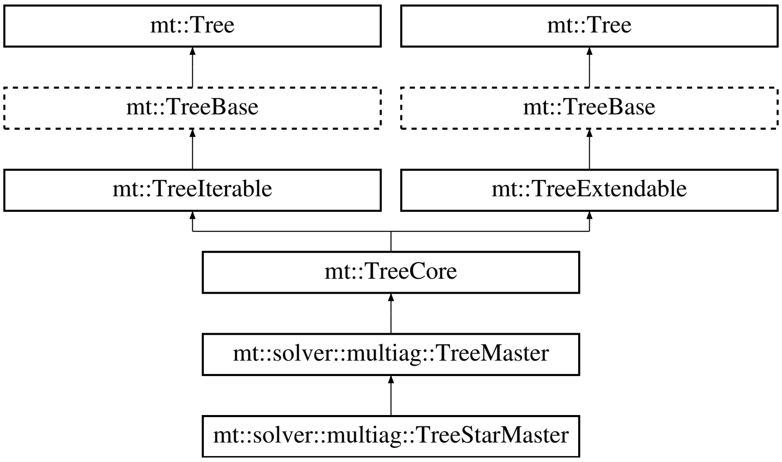
Reimplemented in mt::solver::linked::TreeStarLinked.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/linkedTrees/header/TreeLinked.h

8.45 mt::solver::multiag::TreeMaster Class Reference

Inheritance diagram for mt::solver::multiag::TreeMaster:



Public Member Functions

- **TreeMaster** (NodePtr root, const std::vector< ProblemPtr > &problems)
- void **dispatch** ()
- virtual void **gather** ()
- Tree & **getSlave** (const std::size_t &pos)

Protected Attributes

- std::vector< std::unique_ptr< TreeSlave > > **slaves**

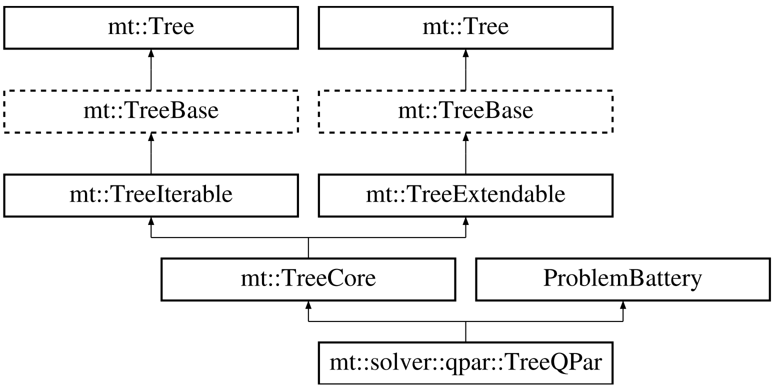
Additional Inherited Members

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/multiAgent/header/TreeMaster.h

8.46 mt::solver::qpar::TreeQPar Class Reference

Inheritance diagram for mt::solver::qpar::TreeQPar:



Public Member Functions

- **TreeQPar** (NodePtr root, const std::vector$<$ ProblemPtr $>$ &problems)
- **TreeQPar** (NodePtr root, const TreeQPar &o)
- void **open** ()
- void **close** ()

Protected Member Functions

- Node $*$ **nearestNeighbour** (const NodeState &state) const override

Protected Attributes

- std::shared_ptr$<$ Pool $>$ **pool**

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

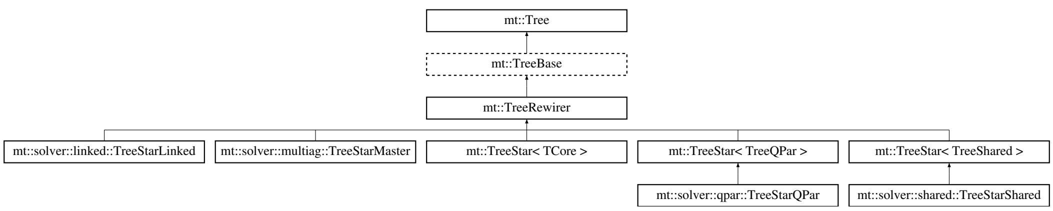
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/queryParall/header/TreeQParall.h

8.47 mt::TreeRewirer Class Reference

Base Tree class with the capability of performing rewires, refer to METTERE.

```
#include <TreeRewirer.h>
```

Inheritance diagram for mt::TreeRewirer:



Protected Member Functions

- std::list$<$ Rewire $>$ computeRewires (Node &pivot) const

  The fathe of the pivot might change to improve the connectivity, refer to METTERE, while the returned rewirds are evaluated but not applied, since might be applied later at the proper time. The pivot node is typically not already part of the tree when evaluating the rewirds.
- virtual std::set$<$ Node $*$ $>$ **nearSet** (const NodeState &state) const
- float **nearSetRay** () const

Additional Inherited Members

8.47.1 Detailed Description

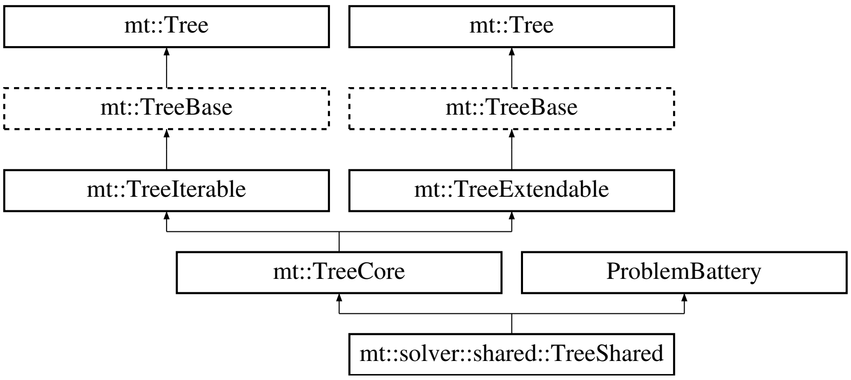
Base Tree class with the capability of performing rewires, refer to METTERE.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/tree/header/TreeRewirer.h

8.48 mt::solver::shared::TreeShared Class Reference

Inheritance diagram for mt::solver::shared::TreeShared:



Public Member Functions

- **TreeShared** (NodePtr root, const std::vector$<$ ProblemPtr $>$ &problems)
- Node $*$ add (NodePtr node) override

    Add the passed node to the collection.
- Nodes::const_reverse_iterator rbegin () const override
- Problem $*$ **getProblem** () const override

Protected Attributes

- std::mutex **mtx**

Additional Inherited Members

8.48.1 Member Function Documentation

8.48.1.1 add()

```
Node* mt::solver::shared::TreeShared::add (
            NodePtr node )  [override], [virtual]
```

Add the passed node to the collection.

Parameters

the	node to introduce in the tree

Reimplemented from mt::TreeIterable.

8.48.1.2 rbegin()

```
Nodes::const_reverse_iterator mt::solver::shared::TreeShared::rbegin ( ) const  [override],
[virtual]
```

Returns

an iterator pointing to the first node in the container

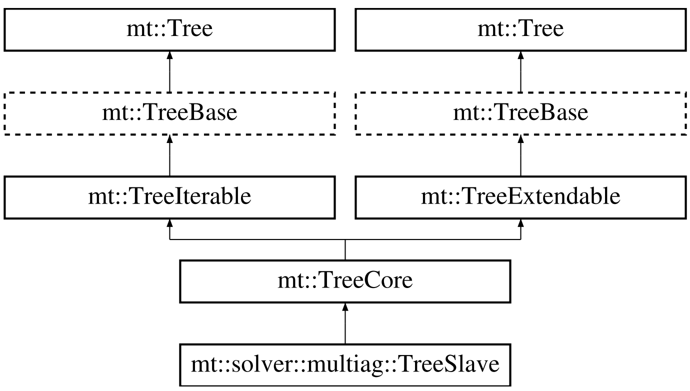
Reimplemented from mt::TreeIterable.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/sharedTree/header/TreeShared.h

8.49 mt::solver::multiag::TreeSlave Class Reference

Inheritance diagram for mt::solver::multiag::TreeSlave:



Public Member Functions

- **TreeSlave** (Problem &problem)
- Node ∗ add (NodePtr node) override

    Add the passed node to the collection.
- Nodes & **getNodes** ()

Public Attributes

- Node ∗ **originalRoot** = nullptr

Additional Inherited Members

8.49.1 Member Function Documentation

8.49.1.1 add()

```
Node* mt::solver::multiag::TreeSlave::add (
            NodePtr node ) [override], [virtual]
```

Add the passed node to the collection.

Parameters

the	node to introduce in the tree

Reimplemented from mt::TreeIterable.

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

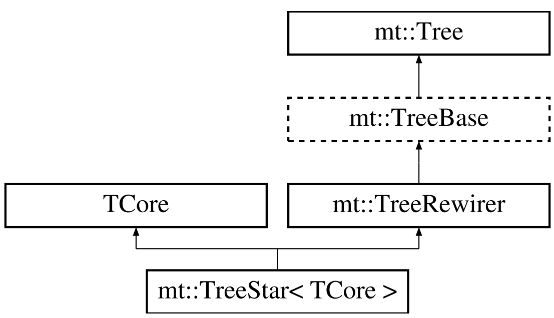
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/multiAgent/header/TreeSlave.h

8.50 mt::TreeStar< TCore > Class Template Reference

A tree that always compute and applies the rewires when adding a new node in the tree.

```
#include <TreeStar.h>
```

Inheritance diagram for mt::TreeStar< TCore >:



Public Member Functions

- template<typename ... Args>
  TreeStar (Args &&... args)
- Node ∗ **add** (NodePtr node) override

Additional Inherited Members

8.50.1 Detailed Description

template<typename TCore>
class mt::TreeStar< TCore >

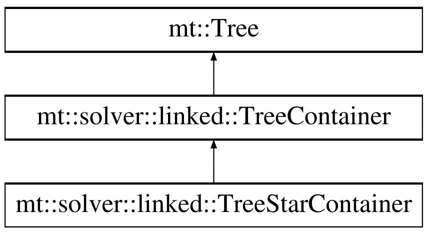
A tree that always compute and applies the rewires when adding a new node in the tree.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/tree/header/TreeStar.h

8.51 mt::solver::linked::TreeStarContainer Class Reference

Inheritance diagram for mt::solver::linked::TreeStarContainer:



Public Member Functions

- **TreeStarContainer** (NodePtr root, const std::vector< ProblemPtr > &problems)

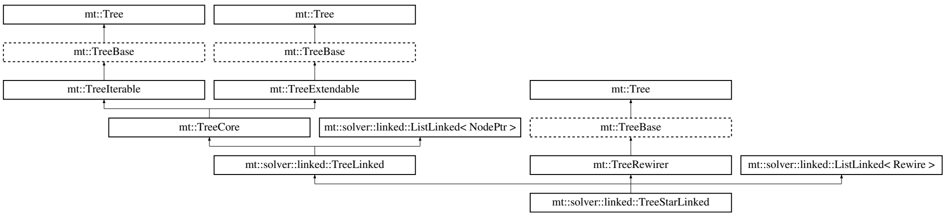
Additional Inherited Members

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/linkedTrees/header/TreeContainer.h

8.52 mt::solver::linked::TreeStarLinked Class Reference

Inheritance diagram for mt::solver::linked::TreeStarLinked:



Public Member Functions

- **TreeStarLinked** (NodePtr root, Problem &problem)
- Node ∗ add (NodePtr node) override

  Add the passed node to the collection.
- void **gather** () override

Additional Inherited Members

8.52.1 Member Function Documentation

8.52.1.1 add()

```
Node* mt::solver::linked::TreeStarLinked::add (
            NodePtr node )  [override], [virtual]
```

Add the passed node to the collection.

Parameters

the	node to introduce in the tree

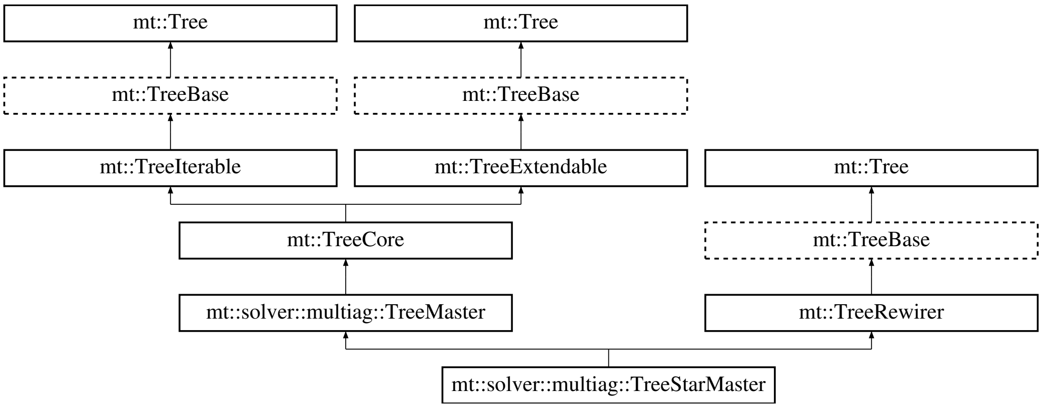
Reimplemented from mt::solver::linked::TreeLinked.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/linkedTrees/header/TreeStarLinked.h

8.53 mt::solver::multiag::TreeStarMaster Class Reference

Inheritance diagram for mt::solver::multiag::TreeStarMaster:



Public Member Functions

- **TreeStarMaster** (NodePtr root, const std::vector< ProblemPtr > &problems)
- void **gather** () override

Protected Member Functions

- std::set< Node * > **nearSet** (const NodeState &state) const override

Protected Attributes

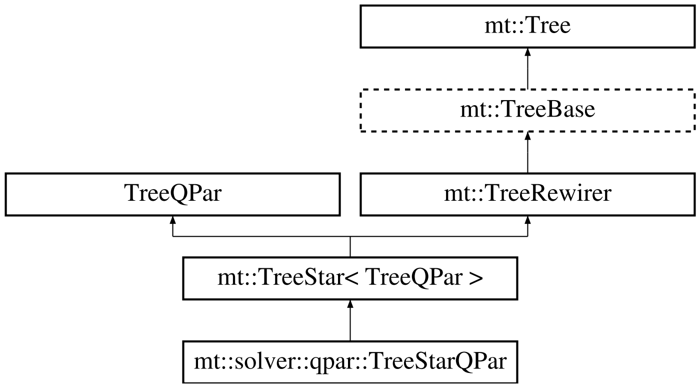
- std::list< Nodes > **temporaryBuffers**

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/multiAgent/header/TreeStarMaster.h

8.54 mt::solver::qpar::TreeStarQPar Class Reference

Inheritance diagram for mt::solver::qpar::TreeStarQPar:



Public Member Functions

- **TreeStarQPar** (NodePtr root, const std::vector< ProblemPtr > &problems)

Protected Member Functions

- std::set< Node * > **nearSet** (const NodeState &state) const override

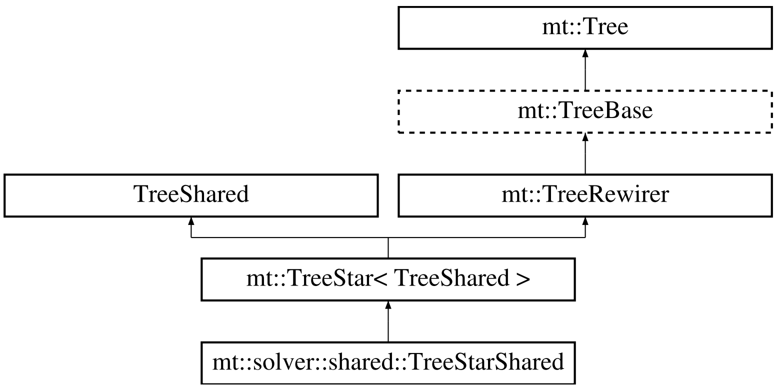
Additional Inherited Members

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/queryParall/header/TreeStarQParall.h

8.55 mt::solver::shared::TreeStarShared Class Reference

Inheritance diagram for mt::solver::shared::TreeStarShared:



Public Member Functions

- **TreeStarShared** (NodePtr root, const std::vector< ProblemPtr > &problems)
- Node * **add** (NodePtr node) override

Additional Inherited Members

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

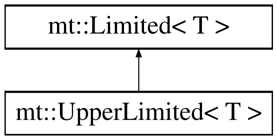
- C:/Users/andre/Desktop/MT-RRT/MT-RRT/src/strategies/sharedTree/header/TreeStarShared.h

8.56 mt::UpperLimited< T > Class Template Reference

A @Limited quantity, having -infinite as lower bound.

```
#include <Limited.h>
```

Inheritance diagram for mt::UpperLimited< T >:



Public Member Functions

- **UpperLimited** (const T &upperBound, const T &initialValue)
- **UpperLimited** (const T &upperBound)

Additional Inherited Members

8.56.1 Detailed Description

template<typename T>
class mt::UpperLimited< T >

A @Limited quantity, having -infinite as lower bound.

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

- C:/Users/andre/Desktop/MT-RRT/MT-RRT/header/Limited.h

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.