MT-RRT, the general purpose multi threading library for RRT.

1.0

Generated by Doxygen 1.8.17

Chapter 1

Foundamental concepts

1.1 What is an RRT algorithm?

Rapidly Random exploring Tree(s), aka RRT(s), is one of the most popular technique adopted for solving planning path problems in robotics. In essence, a planning problem consists of finding a feasible trajectory or path that leads a manipulator, or more in general a dynamical system, from a starting configuration/state to an ending desired one, consistently with a series of constraints. RRTs were firstly proposed in [5]. They are able to explore a state space in an incremental way, building a search tree, even if they may require lots of iterations before terminating. They were proved be capable of always finding at least one solution to a planning problem, if a solution exists, i.e. they are probabilistic complete. RRT were also proved to perform well as kinodynamic planners, designing optimal LQR controllers driving a generic dynamical system to a desired final state, see [9] and [8].
The typical disadvantage of RRTs is that 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 any problem handled by an RRT algorithm. The only necessary thing to do when facing a new class of problem is to derive some specific objects describing the problem itself as detailed in Section 2.
At the same time, one of the most common problem to solve with RRT is a standard path planning 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 explore the state space of a particular problem, in order to find a series of states connecting a starting x_o and an ending one x_f, at the same time accounting for the presence of constraints. More precisely, this is done by building at least 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 to solve, 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 $\tau_{i \rightarrow R}$,

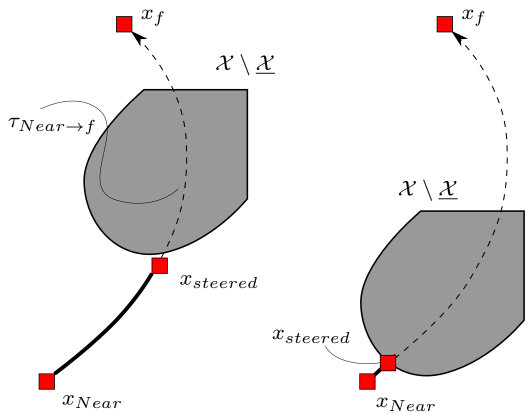


Figure 1.1 The dashed curves in both pictures are the optimal trajectories, agnostic of the constraints, connecting the pair of states x_{Near} and x_f, while the filled areas are regions of X not allowed by constraints. The steering procedure is ideally in charge of searching the furthest state to x_{Near}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

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

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

1.2.2 Bidirectional version of the RRT

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

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

1.2.3 Compute the optimal solution: the RRT*

For any planning problem there are infinite $\tau_{o \to f} \subset \underline{\mathcal{X}}$, i.e. infinite trajectories starting from x_o and terminating in x_f which are entirely contained in the admissible region $\underline{\mathcal{X}}$. Among the aforementioned set, we might be interested in finding the trajectory minimizing the cost $C(\tau_{o \to f})$, refer to Figure 1.4. The basic version of the RRT algorithm is proved to find with a probability equal to 1, a suboptimal solution [4]. The optimality is addressed by a variant of the RRT, called RRT* [4], whose pseudocode is contained

[2] This is done to avoid inserting less informative nodes in the tree, reducing the tree size.

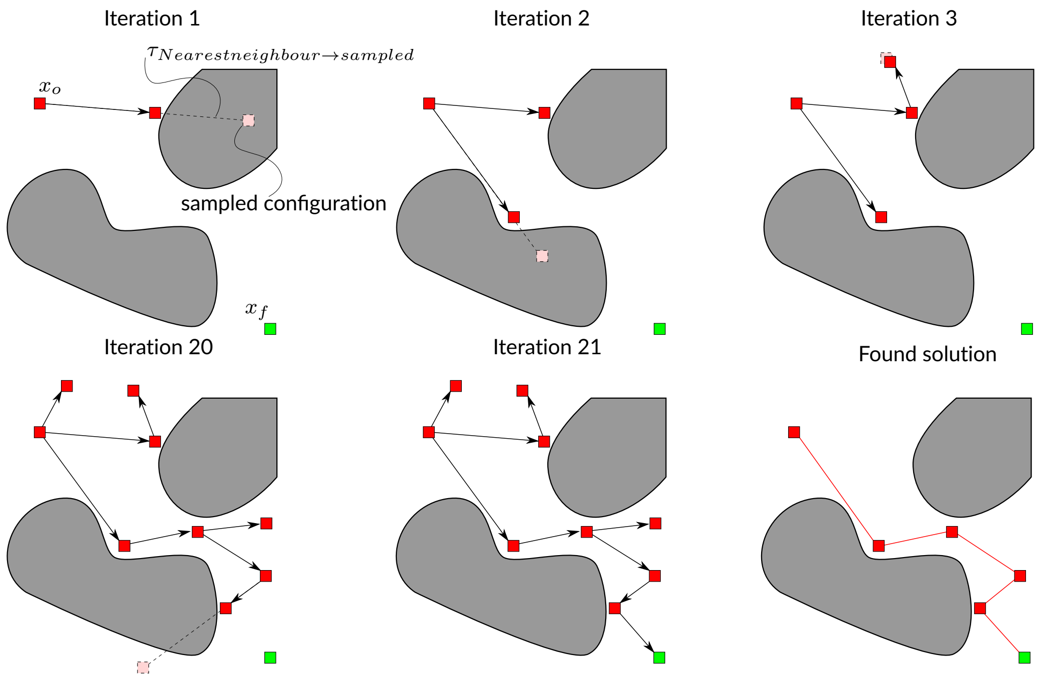


Figure 1.2 Examples of iterations done by an RRT algorithm. The solution found is the one connecting the state in the tree that reached x_f, with the root x_o.

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.

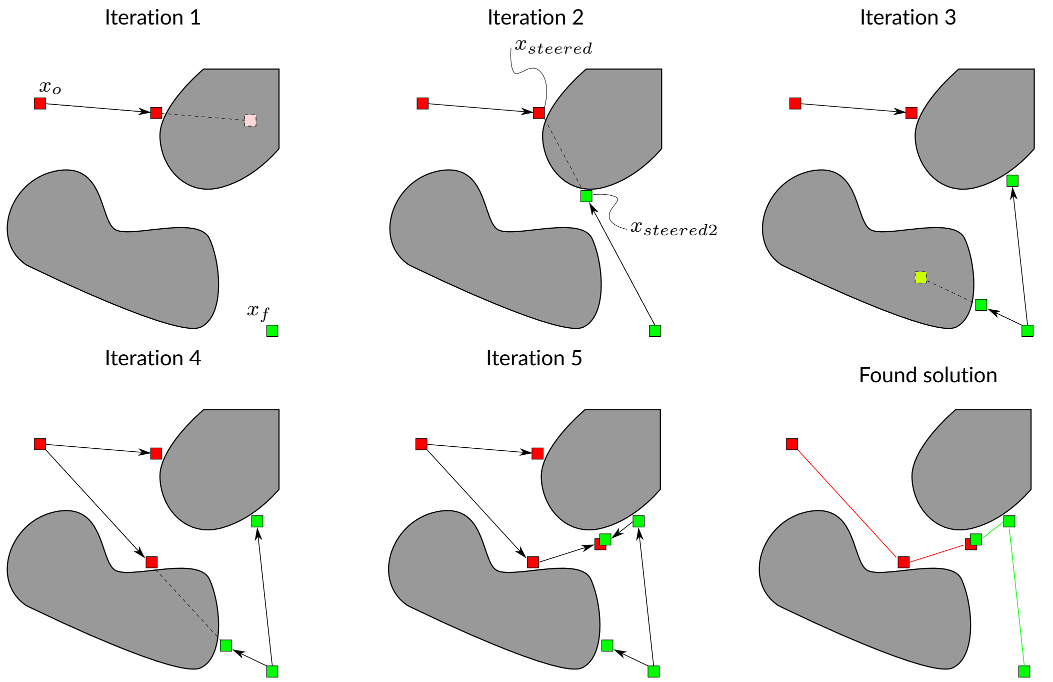


Figure 1.3 Examples of iterations done by the bidirectional version of the RRT. The path in the tree rooted at x_f is reverted to get the second part of the solution.

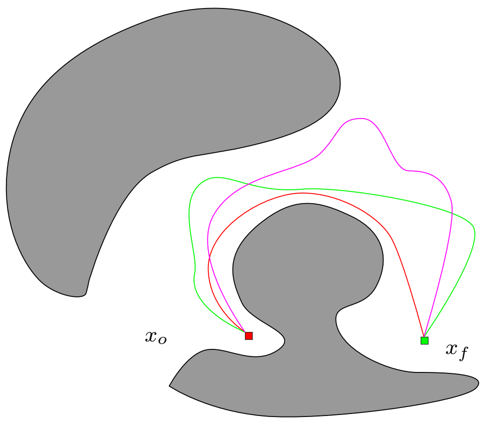


Figure 1.4 Different trajectories connecting x_o with x_f, entirely contained in $\underline{\mathcal{X}}$. If we assume as cost the length of a path, the red solution is the optimal one.

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.

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 class

Whenever you need to solve a new class of planning problem, you should define:

[3] In real cases, after a sufficient big number of iterations an optimizing effect can be yet appreciated.

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
$x_{best} = \underset{x_S \in Solutions}{\text{argmin}}$ (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** $C(\tau_{Fath(n) \to n})$+ 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.

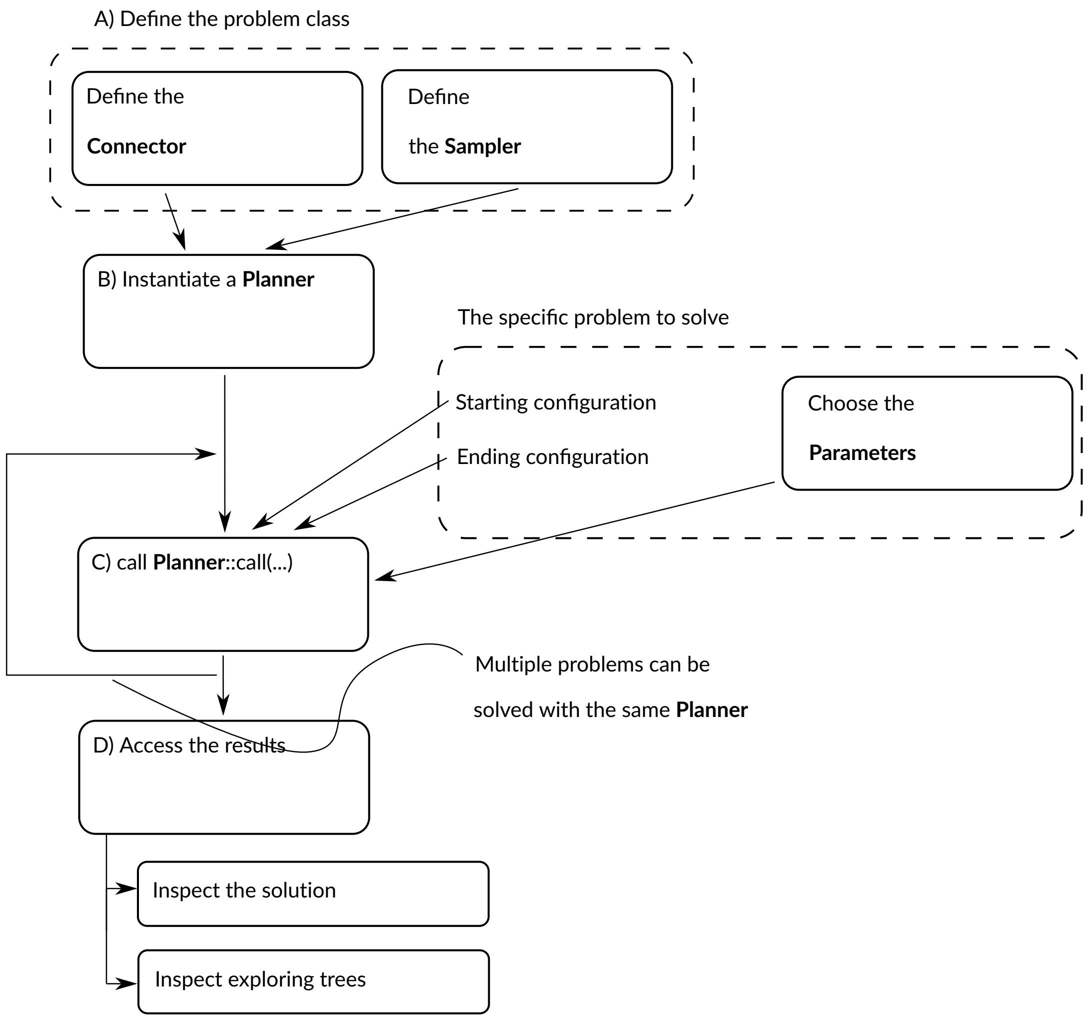


Figure 1.5 Steps to follow for consuming the MT-RRT library.

- a **Sampler**, whose responsability is to draw random states to allow the tree(s) growth.
- a **Connector**, whose responsability is generate optimal trajectories τ connecting pairs of states.

Chapter 2 reports some examples of planning problems, describing froma theoretical point of view how to derive the corresponding **Sampler** and **Connector**. The concrete implementations are instead contained under the samples folder.

1.3.0.2 B) Instantiate a Planner

After having defined the problem, you need to build a **Planner**. This can be a classicla mon threaded StandardPlanner or a multi threaded one. The population of such planners offered by this library is (refer also to Sections 3.0.1, 3.0.2, 3.0.3 and 3.0.4):

- EmbarassinglyParallel
- ParallelizedQueriesPlanner
- SharedTreePlanner
- LinkedTreesPlanner
- MultiAgentPlanner

Each planner support all the rrt algorithms described in Sections 1, 1.2.2 and 5, with the only exception that the MultiAgentPlanner does not support the bidirectional algorithm.

1.3.0.3 C) Solve the problem calling Planner::solve

You can use the generated **Planner** to solve a specific problem, trying to connect a starting and an ending configuration. You need also to specify additional **Parameters** like for instance the kind of strategy (Sections 1, 1.2.2 and 5).

1.3.0.4 D) Inspect the results after solving the problem

Planner::solve returns a **PlannerSolution** structure which contains all the information about the found solution, among which you can find:

- the found 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 $\tau_{1\rightarrow 2}, \tau_{2\rightarrow 3}, \dots, \tau_{M-1\rightarrow M} \subset \underline{\mathcal{X}}$. In case a solution was not found, an empty option is returned.
- the tree(s) computed for finding the solution.

Additional information (the iterations spent, the computation time) regarding the solution can be accessed too.

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 specific concretions of Sampler and Connector containing all the problem-specific information, Section 1.3.0.1.
In order to help the user in understanding how to implement such derivations, 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 two dimensional, having $x_{1,2}$ as coordinates. The aim is to connect two 2D coordinates while avoiding the rectangular obstacles depicted in Figure 2.1. The state space is bounded by two corners describing the maximum and minimum possible x_1 and x_2, see Figure 2.1.

2.1.1 Sampling

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

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

2.1.2 Optimal trajectory and constraints

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

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

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

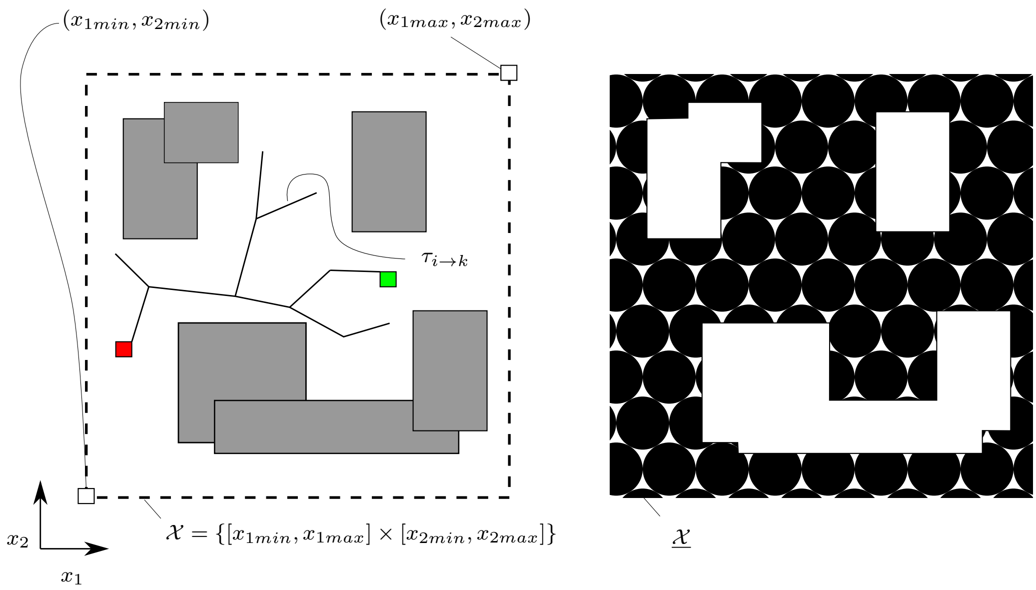
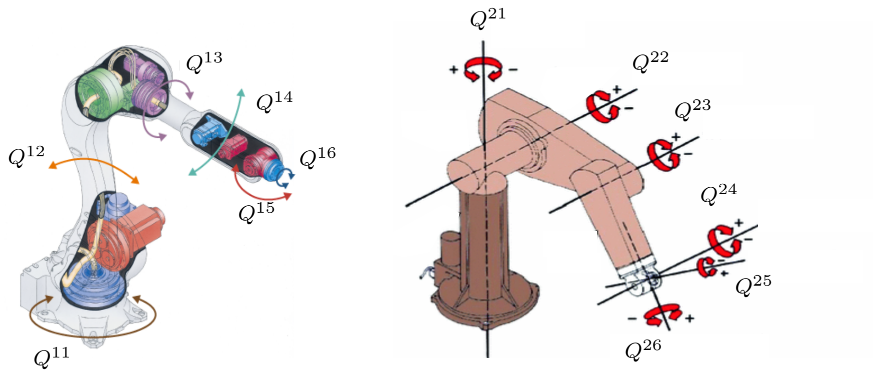


Figure 2.1 Example of maze problem.



$Q^1 = [Q^{11} \quad Q^{12} \quad Q^{13} \quad Q^{14} \quad Q^{15} \quad Q^{16}]^T \qquad Q^2 = [Q^{21} \quad Q^{22} \quad Q^{23} \quad Q^{24} \quad Q^{25} \quad Q^{26}]^T$

Figure 2.2 Rotating joints of two articulated manipulators.

2.1.3 Advancement along the optimal trajectory

The steering procedure is done as similarly described in Section 2.2.3.1, advancing at every steer trial of a quantum of space along τ and checking every time that the segment connecting the previously steered state and the reached one entirely lies in the admitted space.

2.2 Articulated arm problem

This is for sure one of the most common problem that can be solved using rrt algorithms. 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 $Q_{1,...,m}$ 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:

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

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 Advancement along the optimal trajectory

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

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

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

2.2.3.1 Tunneled check collision

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

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

with ϵ in the order of few degrees. $Q_{steered}$ is checked to be or not in $\underline{X}$, by checking the presence of collisions, and is consequently marked as $VALID$ or $INVALID$. This library does not provide a general collision checker (some very basics geometrical functions were implemented in order to run the samples). Anyway when implementing such an approach, you can easily integrate your favourite collision checker (like for example [1] or [2]) to embed in your own Connector.
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 setting a value grater than 1 with Solver::setSteerTrials(...).
Figure 2.3 summarizes the above considerations.

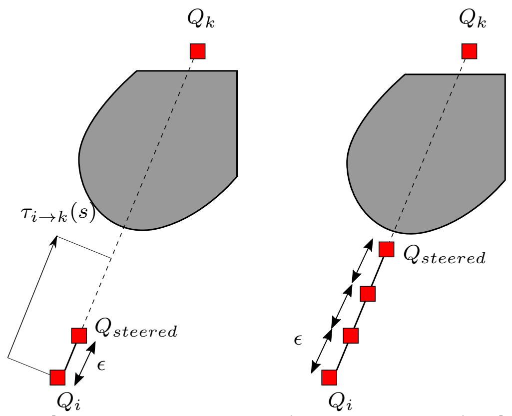


Figure 2.3 Steer extension along the segment connecting two states in the configuration space. On the left a single steer trial approach, on the right a multiple one.

2.2.3.2 Bubble of free configuration

This approach was first proposed in [7] and is based on the definition of a so called bubble of free configuration $\mathcal{B}$. Such a bubble is a region of the configurational space that is built around a state Q_i. More formally, $\mathcal{B}(Q_i)$ is defined as follows [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 j^{th} robot and all the obstacles in the scene, while d^{jk}_{min} is the minimum distance between the j^{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:

$$\begin{aligned} Q_{steered} &= \begin{bmatrix} Q^{1T}_{steered} & \dots & Q^{nT}_{steered} \end{bmatrix}^T = Q_i + s_{steered}(Q_k - Q_i) \\ s_{steered} &= min \left\{ s_A, s_B \right\} \\ s_A &= min_{j \in \{1,\dots,n\},q} \left\{ \frac{d^j_{min}}{\sum_q R^{jq} |Q^{jq}_i - Q^{jq}_k|} \right\} \\ s_B &= min_{j,k \in \{1,\dots,n\},q,q_2} \left\{ \frac{d^{jk}_{min}}{\sum_q R^{jq} |Q^{jq}_i - Q^{jq}_k| + \sum_{q_2} R^{kq2} |Q^{kq_2}_i - Q^{kq_2}_k|} \right\} \end{aligned} \tag{2.10}$$

Also in this case a multiple steer approach is possible for this strategy, refer also to Figure 2.5.
Again, you can deploy your own geometric engine in order to compute the distances d^j_{min}, d^{jk}_{min} as well as the radii R^{ki} and define your custom Connector implementing the approach described in this Section.

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

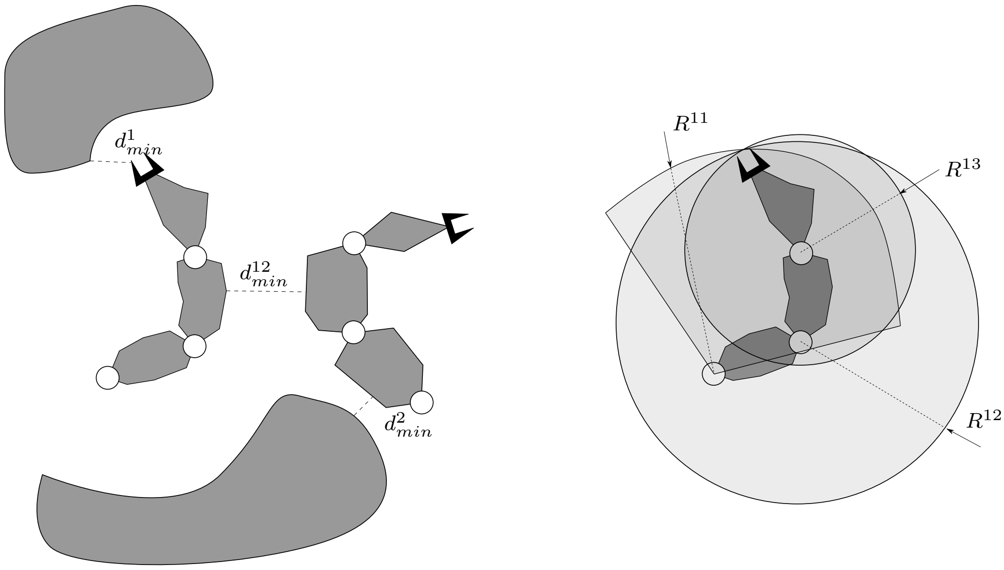


Figure 2.4 The quantities involved in the computation of the bubble $\mathcal{B}$.

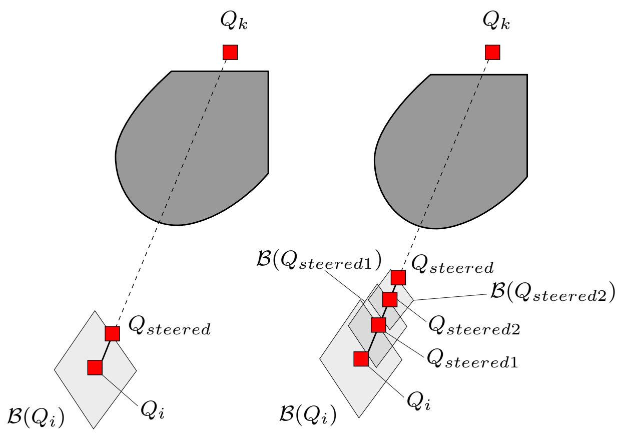


Figure 2.5 Single (left) and multiple (right) steer using the bubbles of free configurations.

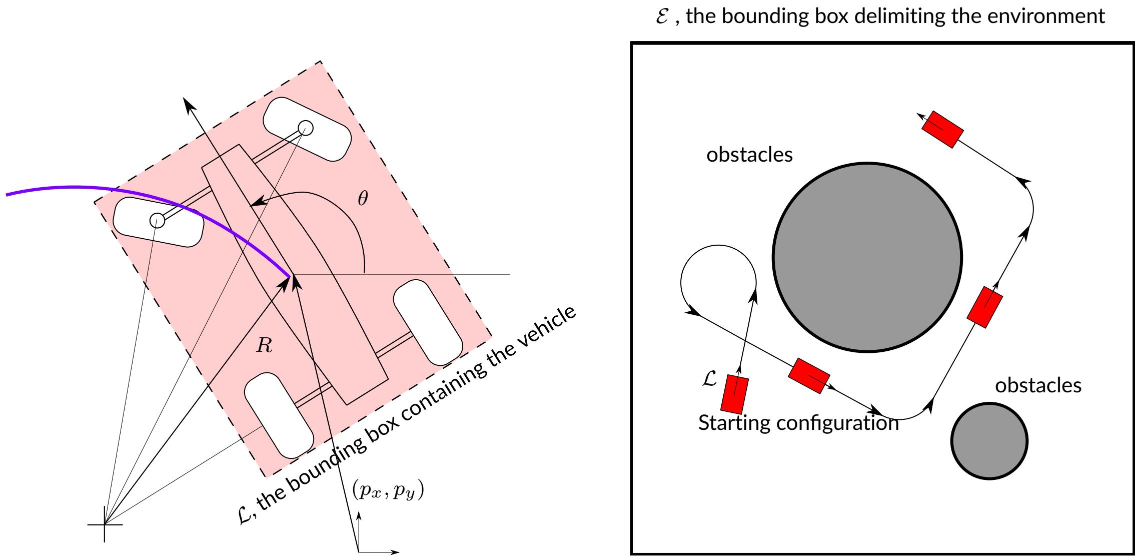


Figure 2.6 Vehicle motion in a planar environment.

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 2.6. 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 2.6. We assume that possible the steering radius R in a compact interval $[R_{min}, R_{max}]$.
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 2.6. The sampling of a random configuration for the vehicle is done in this way:

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

2.3.2 Optimal trajectory and constraints

The optimal trajectory connecting two configurations x_i, x_j is made of three parts [2] (refer to the examples in the right part of Figure 2.6 and the top part of Figure 2.7):

- a straight line starting from x_i

[2]Except when the starting and ending configuration have the same orientation and lies on the same line. In that case the trejectory is a simple segment connecting the 2 states.

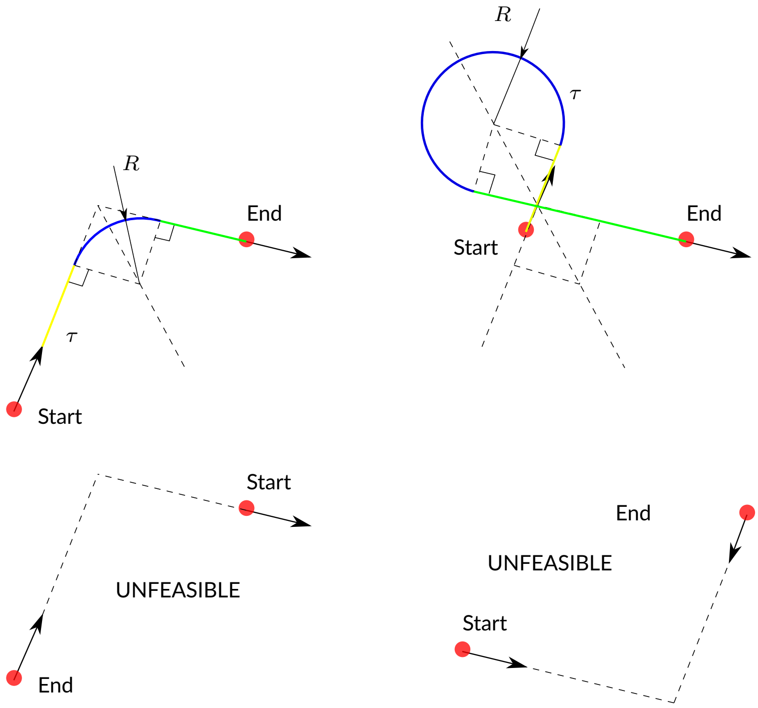


Figure 2.7 Examples of feasible, top, and non feasible trajectories, bottom. The different parts of the feasible trajectories are highlighted with different colors.

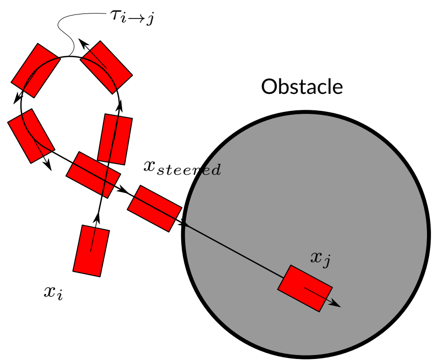


Figure 2.8 Steering procedure for a planar navigation problem.

- 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 2.7. Therefore, in case the trajectory $\tau_{i \to j}$ does not exists, $C(\tau_{i \to j})$ is assumed equal to $+\infty$.

2.3.3 Advancement along the optimal trajectory

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 2.8 summarizes the steering procedure.

Chapter 3

Parallel RRT

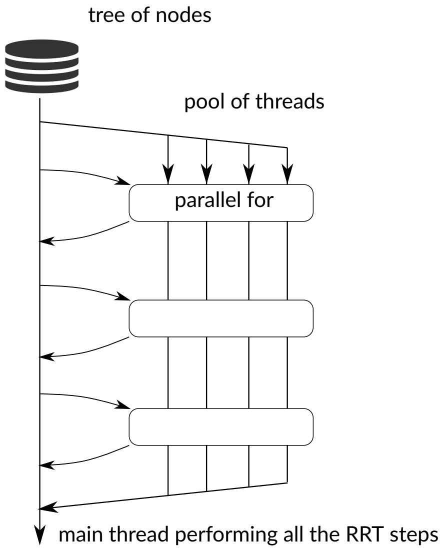
This Chapter will provide details about the multi-threaded strategies provided by MT-RRT. Further details are contained also in [3], which is the publication were for the first time MT-RRT was presented. In [3] you can find also a comparison in terms of computational times.

3.0.1 Parallelization of the query activities

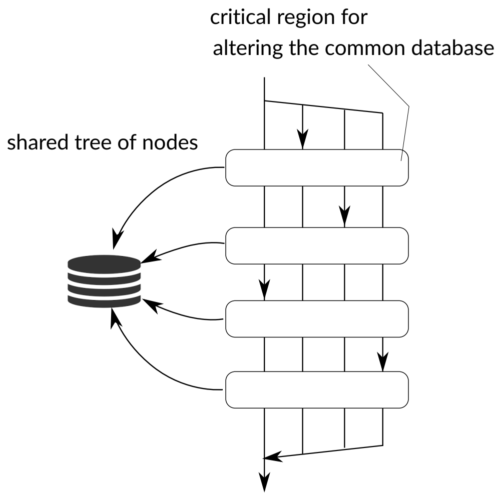
All the RRT versions spend a significant time in performing query operations on the tree, i.e. operations that require to traverse all the tree. Such operations are mainly the nearest neighbour search, algorithm 3, and the determination of the near set, algorithm 6.
The key idea is to perform the above query operations by making use of a thread pool implementing parallel for regions, 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. All the threads in the pool 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 object implementing this approach is QueryParallStrategy.

3.0.2 Shared tree critical regions

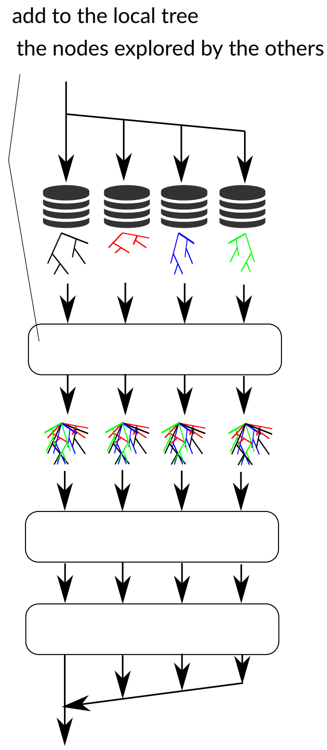
Another way to obtain a parallelization is to actually do simultaneously, every single step of the RRT versions. Therefore, we can imagine having threads sharing a common tree (or two trees in the case of a bidirectional strategy), executing in parallel every step of the expansion process. Some critical 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 object implementing this approach is SharedTreeStrategy.



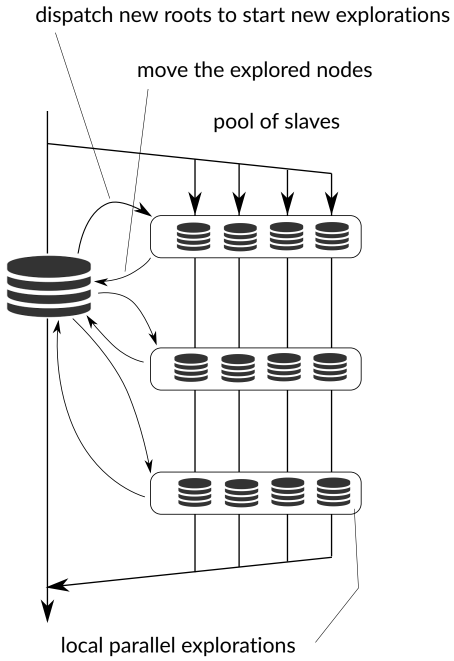
(a) Schematic representation of the parallelization of the query activities approach.



(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.

3.0.3 Parallel expansions on linked 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 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 object implementing this approach is LinkedTreesStrategy.

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.
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 object implementing this approach is MultiAgentStrategy.

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

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_rrt Namespace Reference

Classes

- class Copiable
- class EmbarassinglyParallelPlanner

    Into N different threads, N different solution are found running the classical version of the planner. The best solution among thosw found by all threads is returned.
- class Error
- class Extender

    Someone able to extend one or two connected search trees.
- class GaussianEngine

    Used to draw samples from a normal distribution.
- class Limited

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

    Approach described at Section "Parallel expansions on linked trees" of the documentation.
- class LowerLimited

    A @Limited quantity, having infinity as upper bound.
- class MultiAgentPlanner

    Approach described at Section "Multi agents approach" of the documentation.
- class MultiThreadedPlanner
- class ParallelizedQueriesPlanner

    Approach described at Section "Parallelization of the query activities" of the documentation.
- class Positive

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

    Approach described at Section "Shared tree critical regions" of the documentation.
- class SynchronizationAware
- class SynchronizationDegree

    Synchronization degree used by MultiAgentPlanner and LinkedTreesPlanner. This regulates how often the threads stop expansions and share info about explored states with other threads.
- class Threads

    Number of threads used by a MultiThreadedPlanner.
- class UniformEngine

    Used to draw sample whithin a compact inverval [l, U].
- class UpperLimited

    A @Limited quantity, having negative infinity as lower bound.

Typedefs

- using **Seed** = unsigned

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)
- Seed make_random_seed ()
- template<typename T1 , typename... Args>
  std::string merge (const T1 &first, Args... slices_to_merge)

  put all the passed slices all together into a single string, that is returned.

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_rrt::getIterationsDone (
            const std::vector< Extender > & battery )
```

Returns

the sum of extensions done by all the passed extenders

7.1.2.2 make_random_seed()

```
Seed mt_rrt::make_random_seed ( )
```

Returns

time since start of the process is used to generated the seed

Chapter 8

Class Documentation

8.1 mt_rrt::Copiable< T > Class Template Reference

Public Member Functions

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

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

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

8.2 mt_rrt::detail::Distribution< DistributionShape > Class Template Reference

Public Member Functions

- float **sample** () const
- **Distribution** (const Distribution &o)
- Distribution & **operator=** (const Distribution &o)=delete
- **Distribution** (Distribution &&o)=delete
- Distribution & **operator=** (Distribution &&o)=delete
- Seed **sampleSeed** () const

Protected Member Functions

- **Distribution** (DistributionShape &&shape, const std::optional< Seed > &seed)

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

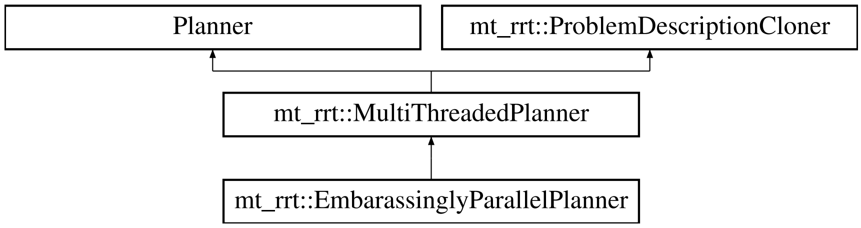
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/carpet/header/MT-RRT-carpet/Random.h

8.3 mt_rrt::EmbarassinglyParallelPlanner Class Reference

Into N different threads, N different solution are found running the classical version of the planner. The best solution among thosw found by all threads is returned.

```
#include <EmbarassinglyParallel.h>
```

Inheritance diagram for mt_rrt::EmbarassinglyParallelPlanner:



Public Member Functions

- template<typename... Args>
  MultiThreadedPlanner (Args... args)

Protected Member Functions

- void **solve_** (const State &start, const State &end, const Parameters &parameters, PlannerSolution &recipient) final

8.3.1 Detailed Description

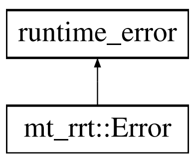
Into N different threads, N different solution are found running the classical version of the planner. The best solution among thosw found by all threads is returned.

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

- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Embarassingly↩
  Parallel.h

8.4 mt_rrt::Error Class Reference

Inheritance diagram for mt_rrt::Error:



Public Member Functions

- **Error** (const std::string &what)
- template<typename T1 , typename T2 , typename... Args>
  Error (const T1 &first, const T2 &second, Args... slices_to_merge)

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

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

8.5 mt_rrt::Extender< Solution > Class Template Reference

Someone able 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 controlled tree(s). This function may be called multiple times, for performing batch of extensions. All the solutions found while extending are saved and stored into this object.
- std::size_t getIterationsDone () const
  Get the extensions done so far (all extend() called so far are accounted).
- const std::set< Solution > & getSolutions () const
  Get the population of solutions found so far.
- std::vector< NodeState > computeBestSolutionSequence () const
- bool **isCumulating** () const

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)
- virtual std::vector< NodeState > **computeSolutionSequence** (const Solution &sol) const =0

Protected Attributes

- sampling::UniformEngine **randEngine**
- const bool cumulateSolutions
  when set true, the extension process is not arrested when a solution is found, but the extender keeps on trying to find also additional solutions.
- const double **deterministicCoefficient**
- std::size_t **iterationsDone** = 0
- std::set< Solution > **solutionsFound**

8.5.1 Detailed Description

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

Someone able 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_rrt::Extender< Solution >::computeBestSolutionSequence ( ) const
[inline]
```

Returns

the sequence pertaining to the best found solution. In case no solution was found, an empty vector is returned.

8.5.2.2 computeBestSolutionSequence() [2/2]

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

Returns

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

8.5.2.3 extend()

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

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

Parameters

the	number of extension to perform

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

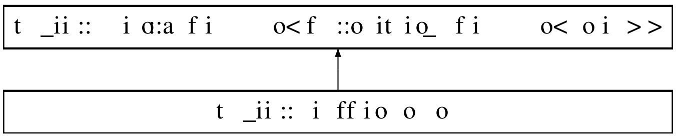
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/carpet/header/MT-RRT-carpet/Extender.h

8.6 mt_rrt::GaussianEngine Class Reference

Used to draw samples from a normal distribution.

```
#include <Random.h>
```

Inheritance diagram for mt_rrt::GaussianEngine:



Public Member Functions

- GaussianEngine (const float &mean=0, const float &stdDeviation=1.f, const std::optional< Seed > &seed=std::nullopt)

Additional Inherited Members

8.6.1 Detailed Description

Used to draw samples from a normal distribution.

8.6.2 Constructor & Destructor Documentation

8.6.2.1 GaussianEngine()

```
mt_rrt::GaussianEngine::GaussianEngine (
            const float & mean = 0,
            const float & stdDeviation = 1.f,
            const std::optional< Seed > & seed = std::nullopt )
```

Parameters

the	mean of the normal distribution
the	standard deviation of the normal distribution

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

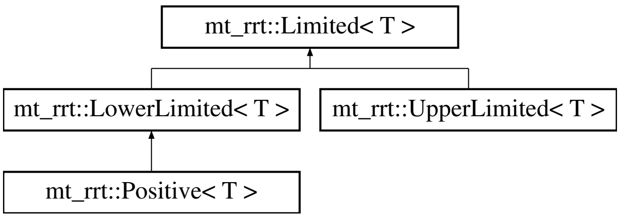
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/carpet/header/MT-RRT-carpet/Random.h

8.7 mt_rrt::Limited< T > Class Template Reference

A quantity whose value should always remain between defined bounds.

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

Inheritance diagram for mt_rrt::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**
- T **lowerBound**
- T **upperBound**

8.7.1 Detailed Description

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

A quantity whose value should always remain between defined bounds.

8.7.2 Constructor & Destructor Documentation

8.7.2.1 Limited()

```
template<typename T >
mt_rrt::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

8.7.3 Member Function Documentation

8.7.3.1 get()

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

Returns

the current value

8.7.3.2 set()

```
template<typename T >
void mt_rrt::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:

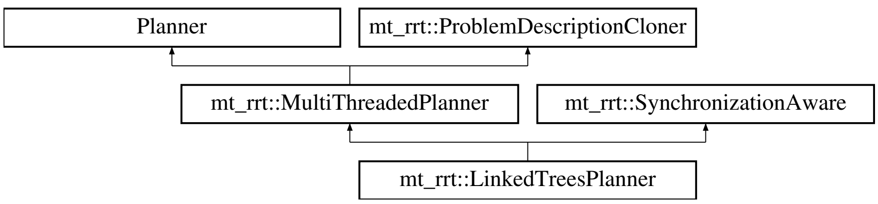
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/carpet/header/MT-RRT-carpet/Limited.h

8.8 mt_rrt::LinkedTreesPlanner Class Reference

Approach described at Section "Parallel expansions on linked trees" of the documentation.

```
#include <LinkedTreesPlanner.h>
```

Inheritance diagram for mt_rrt::LinkedTreesPlanner:



Public Member Functions

- template<typename... Args>
  MultiThreadedPlanner (Args... args)

Protected Member Functions

- void **solve_** (const State &start, const State &end, const Parameters &parameters, PlannerSolution &recipient) final

8.8.1 Detailed Description

Approach described at Section "Parallel expansions on linked trees" of the documentation.

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

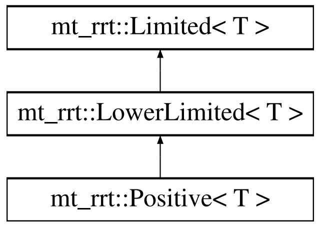
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Linked↩ TreesPlanner.h

8.9 mt_rrt::LowerLimited< T > Class Template Reference

A @Limited quantity, having infinity as upper bound.

`#include <Limited.h>`

Inheritance diagram for mt_rrt::LowerLimited< T >:



Public Member Functions

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

Additional Inherited Members

8.9.1 Detailed Description

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

A @Limited quantity, having infinity as upper bound.

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

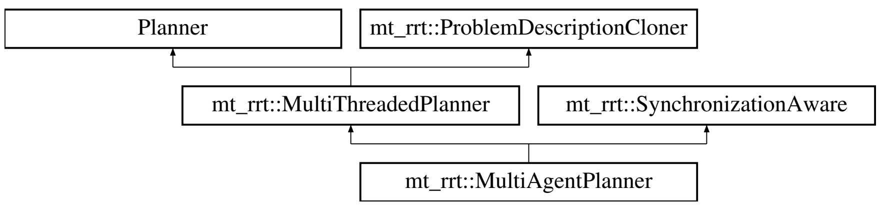
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/carpet/header/MT-RRT-carpet/Limited.h

8.10 mt_rrt::MultiAgentPlanner Class Reference

Approach described at Section "Multi agents approach" of the documentation.

`#include <MultiAgentPlanner.h>`

Inheritance diagram for mt_rrt::MultiAgentPlanner:



Public Types

- enum StarExpansionStrategyApproach { **ExploitAllThreads**, **MonoThread** }

Public Member Functions

- void **setStarApproach** (StarExpansionStrategyApproach approach)
- StarExpansionStrategyApproach **getStarApproach** () const
- template<typename... Args>
  MultiThreadedPlanner (Args... args)

Protected Member Functions

- void **solve_** (const State &start, const State &end, const Parameters &parameters, PlannerSolution &recipient) final

8.10.1 Detailed Description

Approach described at Section "Multi agents approach" of the documentation.

8.10.2 Member Enumeration Documentation

8.10.2.1 StarExpansionStrategyApproach

```
enum mt_rrt::MultiAgentPlanner::StarExpansionStrategyApproach  [strong]
```

explanation:

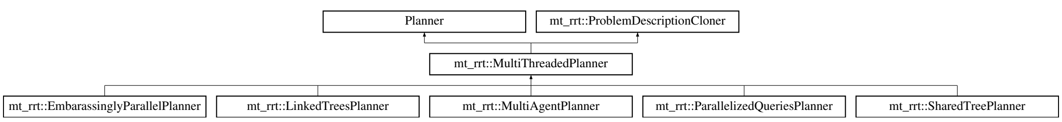
- ExploitAllThreads -> exploit all threads when gathering results from slave. This affects significantly the performance only in case of Star approach.
- MonoThread -> results from slaves are gathered one at a time.

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

- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Multi↩
  AgentPlanner.h

8.11 mt_rrt::MultiThreadedPlanner Class Reference

Inheritance diagram for mt_rrt::MultiThreadedPlanner:



Public Member Functions

- template<typename... Args>
  MultiThreadedPlanner (Args... args)
- void **setThreads** (const Threads &threads_to_use)
- void setMaxThreads ()

  the maxium possible threads available on this machine is used
- std::size_t **getThreads** () const

Additional Inherited Members

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

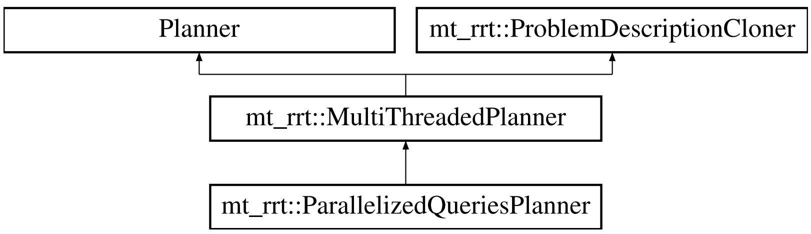
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Multi↩
  ThreadedPlanner.h

8.12 mt_rrt::ParallelizedQueriesPlanner Class Reference

Approach described at Section "Parallelization of the query activities" of the documentation.

```
#include <ParallelizedQueriesPlanner.h>
```

Inheritance diagram for mt_rrt::ParallelizedQueriesPlanner:



Public Member Functions

- template<typename... Args>
  MultiThreadedPlanner (Args... args)

Protected Member Functions

- void **solve_** (const State &start, const State &end, const Parameters &parameters, PlannerSolution &recipient) final

8.12.1 Detailed Description

Approach described at Section "Parallelization of the query activities" of the documentation.

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

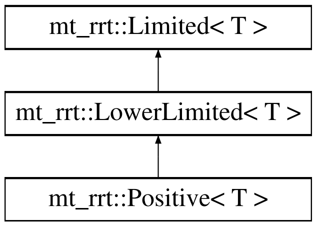
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Parallelized↩
  QueriesPlanner.h

8.13 mt_rrt::Positive< T > Class Template Reference

A @LowerLimited quantity, having 0.0 as lower bound.

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

Inheritance diagram for mt_rrt::Positive< T >:



Public Member Functions

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

Additional Inherited Members

8.13.1 Detailed Description

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

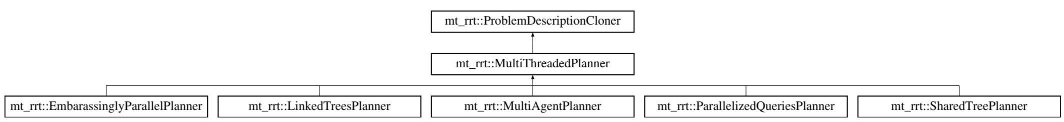
A @LowerLimited quantity, having 0.0 as lower bound.

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

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

8.14 mt_rrt::ProblemDescriptionCloner Class Reference

Inheritance diagram for mt_rrt::ProblemDescriptionCloner:



Protected Member Functions

- **ProblemDescriptionCloner** (const ProblemDescriptionPtr &problem)
- ProblemDescriptionPtr **problemAt** (const std::size_t thread_id)
- const std::vector< ProblemDescriptionPtr > & **getAllDescriptions** () const
- void **resizeDescriptions** (const std::size_t size)

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

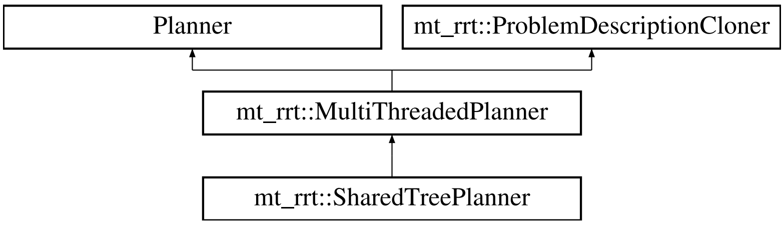
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Multi↩ThreadedPlanner.h

8.15 mt_rrt::SharedTreePlanner Class Reference

Approach described at Section "Shared tree critical regions" of the documentation.

#include <SharedTreePlanner.h>

Inheritance diagram for mt_rrt::SharedTreePlanner:



Public Member Functions

- template<typename... Args>
MultiThreadedPlanner (Args... args)

Protected Member Functions

- void **solve_** (const State &start, const State &end, const Parameters &parameters, PlannerSolution &recipient) final

8.15.1 Detailed Description

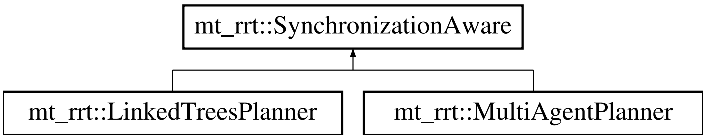
Approach described at Section "Shared tree critical regions" of the documentation.

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

- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Shared↩TreePlanner.h

8.16 mt_rrt::SynchronizationAware Class Reference

Inheritance diagram for mt_rrt::SynchronizationAware:



Public Member Functions

- SynchronizationDegree & **synchronization** ()
- const SynchronizationDegree & **synchronization** () const

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

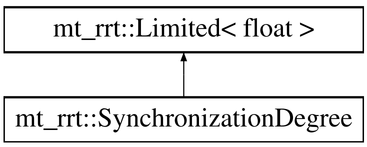
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Synchronization.↩
h

8.17 mt_rrt::SynchronizationDegree Class Reference

Synchronization degree used by MultiAgentPlanner and LinkedTreesPlanner. This regulates how often the threads stop expansions and share info about explored states with other threads.

`#include <Synchronization.h>`

Inheritance diagram for mt_rrt::SynchronizationDegree:



Public Member Functions

- **SynchronizationDegree** (const float initial_value)

Additional Inherited Members

8.17.1 Detailed Description

Synchronization degree used by MultiAgentPlanner and LinkedTreesPlanner. This regulates how often the threads stop expansions and share info about explored states with other threads.

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

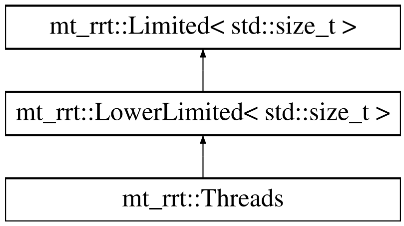
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Synchronization.↩
h

8.18 mt_rrt::Threads Class Reference

Number of threads used by a MultiThreadedPlanner.

`#include <MultiThreadedPlanner.h>`

Inheritance diagram for mt_rrt::Threads:



Public Member Functions

- **Threads** (const std::size_t threads)

Additional Inherited Members

8.18.1 Detailed Description

Number of threads used by a MultiThreadedPlanner.

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

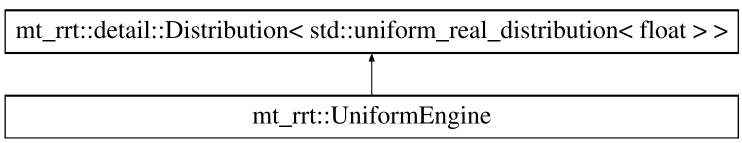
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/multi-threaded/header/MT-RRT-multi-threaded/Multi↩
ThreadedPlanner.h

8.19 mt_rrt::UniformEngine Class Reference

Used to draw sample whithin a compact inverval [l, U].

`#include <Random.h>`

Inheritance diagram for mt_rrt::UniformEngine:



Public Member Functions

- UniformEngine (const float &lowerBound=0, const float &upperBound=1.f, const std::optional< Seed > &seed=std::nullopt)

Additional Inherited Members

8.19.1 Detailed Description

Used to draw sample whithin a compact inverval [l, U].

8.19.2 Constructor & Destructor Documentation

8.19.2.1 UniformEngine()

```
mt_rrt::UniformEngine::UniformEngine (
            const float & lowerBound = 0,
            const float & upperBound = 1.f,
            const std::optional< Seed > & seed = std::nullopt )
```

Parameters

the	lower bound of the compact interval
the	upper bound of the compact interval

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

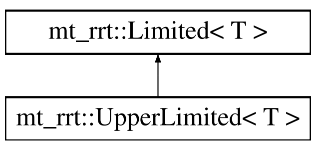
- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/carpet/header/MT-RRT-carpet/Random.h

8.20 mt_rrt::UpperLimited< T > Class Template Reference

A @Limited quantity, having negative infinity as lower bound.

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

Inheritance diagram for mt_rrt::UpperLimited< T >:



Public Member Functions

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

Additional Inherited Members

8.20.1 Detailed Description

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

A @Limited quantity, having negative infinity as lower bound.

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

- C:/Users/andre/Desktop/GitRepo/MT-RRT/src/carpet/header/MT-RRT-carpet/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.