Kinodynamic Motion Planning

June 10, 2022

# 1 Abstract

Autonomous navigation systems usually include a path planner and a local planner. The first one provides a somewhat kinematically reasonable path, which often cannot be passed by a robot, while the second one tries to adapt the path to the dynamic environment and other constraints that were not checked by the path planner. This work is an attempt to create an optimized RRT-based path planner that takes into account nonholonomic constraints. This is achieved through an analysis and combination of already existing solutions, usage of R-Trees, techniques like goal-biased sampling and more. As a result, open source reference implementation in Python constructs trees that require far less local (reactive) planning than typically.

## 1.1 Keywords

Path planning, Collision avoidance, Nonholonomic constraints, Rapidly exploring Random Tree, R-Tree, Reeds-Shepp Car.

# 2 Introduction

Motion planning and autonomous vehicles have been developing for quite long time already and in recent 20 years those areas have grown from enterprise-only solutions (i.e. robotic arms and other manufacturing devices) to consumer-friendly ones (robot vacuums and ”autopilots” in cars). If initially the problem was to ”get from point A to point B”, by this time it has been expanded with operation in dynamic environments, collision avoidance, computation efficiency and others.

All of the mentioned problems have one thing in common - Configuration Spaces framework that is used to describe the state of the robot in the environment. In order to state the problem one should define C-Space - $\mathcal{C}$, which includes configurations of the robot $q$, world $\mathcal{W}$ (including obstacles) and other constraints.

This work is aimed to solve the problem of planar path-planning, thus $\mathcal{W} = \mathbb{R}^2$. Some parts of the world are occupied by the obstacles $\mathcal{O} \subset \mathcal{W}$ defined by polygons,

which cannot be self-intersecting. And finally the robot - a rigid body $\mathcal{A}$, which can translate in the world, making $\mathcal{C}$ a manifold $M_1 = \mathbb{R}^2$, and rotate around its bounding box center, which adds $M_2 = \mathbb{S}^1$, thus without obstacles:

$$\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1 = SE(2) \tag{2.1}$$

The robot's configuration in the space is defined as a vector $q = (x, y, \theta)$ and the space that it is occupying as $\mathcal{A}(q) \subset \mathcal{W}$. Thus, the configuration space for obstacles is:

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\} \tag{2.2}$$

$$\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs} \tag{2.3}$$

The algorithm is created with holonomic - the robot can only travel in $\mathcal{C}_{free}$ - and nonholonomic constraints in mind - it uses Reeds-Shepp Car model and steering function as the only way to produce a path from $q_1 \in \mathcal{C}_{free}$ to $q_2 \in \mathcal{C}_{free}$, making some configurations in $C_{free}$ unreachable. The Reeds-Shepp Car has only two gears - forward and reverse ($u \in \{-1, 1\}$); and angular velocity that depends on maximum turning radius with unit-velocity $r_{turn}$ - $w = \frac{v=1}{r_{turn}}$. In general, the path would be a non-linear function of time, as a turn is a curve. However, in order for a robot to be able to follow the path, each turn is approximated with a given level of discretization $\delta$ to a set of straight lines:

$$\begin{cases} \dot{x} = x_{prev} + u \cdot \delta \cdot \cos\theta \\ \dot{y} = y_{prev} + u \cdot \delta \cdot \sin\theta \\ \dot{\theta} = \theta_{prev} + u \cdot \frac{\delta}{r_{turn}} \end{cases} \tag{2.4}$$

Finally, the starting points are defined as $q_i \in \mathcal{C}_{free}$ and the goal point is $q_g \in \mathcal{C}_{free}$. The local planner should produce a path, which is a set of connected linearized curves that were provided by the steering function.

Generally, industry solves such problem with either a "sampling" algorithm (RRT and variants, PRM) or a graph traversal algorithm like A*. Right now RRT-like algorithms are used more as with a proper set of improvements they can converge quite fast. Classical implementation comes with two parameters - iterations limit $K$ and maximum edge length $\Delta t$. The data on obstacles is often stored as points in KD-Tree and the collision detection is performed by querying it. Nonholonomic constraints are usually left to the reactive planner. There are several drawbacks related to those approaches. Firstly, the use of KD-Tree implies a complexity increase on tree expansion, as KD-Tree is not meant to be updated. Secondly, collision detection in such cases is often done by checking the area of some radius around the robot, which makes it impossible for a planner to create a path through narrow corridors, or by checking its bounding box, which is inefficient for any shape except for rectangles that are aligned to the axes (which is not true for any rotated rectangle). And finally, lack of nonholonomic constraints might make the whole path unreachable. The aim of the presented algorithm is not to suffer from those issues with as little overhead compared to others as possible. This is achieved with

the help of R-Tree, special polygon approximation technique, goal-biased sampling and nearest neighbor randomization.

# 3 Literature Overview

All previous conclusions about ”the industry” were obtained by analysing recent researches published at ICRA, IROS, IEEE and other robotics-related conferences.

# 4 The Proposed Method: BR-RRT

Our method (BR-RRT - Boxes and R-Trees-RRT)is based on the RRT algorithm and essentially is an extension for it. The difference from other works lies in data structures, point sampling approach and collision detection.

## 4.1 Tree and Obstacles in R-Tree

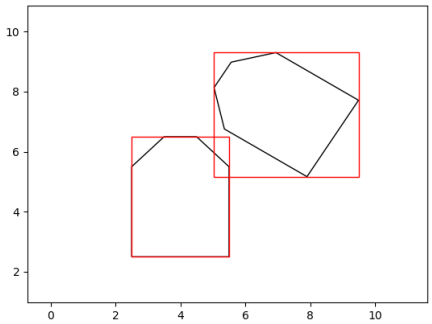
As the data structure for the tree points we propose R-Tree, which does not always require full tree rebuild for an insertion. While R-Tree does not drastically differ in complexity terms from a KD-Tree, it has an advantage of being able to store not just vertices, but spatial data in forms of rectangles.

Some works proposed the use of R-Trees as a structure for obstacles as well. However, in shaped that is not a rectangle and not perfectly aligned to the axes requires significantly more $\mathcal{C}_{free}$ space than it actually should (Fig. 4.1.a). Since it can drastically decrease performance of the algorithm at turns, we are proposing an algorithm for approximation of polygons to a set of axes-aligned rectangles. This kind of approximation of both obstacles and the car allows to store everything in the R-Tree. However, it would be more beneficial to store it in a differently configured R-Tree, as density and amount of data points is incomparable to those in the RRT nodes tree.

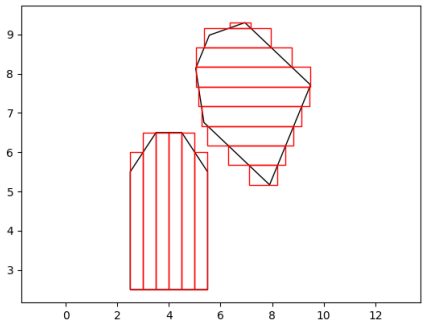
**Algorithm 4.1** Boxify

Input $O$: Obstacle, $p$: detalization factor
$x_{min}, y_{min}, x_{max}, y_{max}$ = BoundingBox($O$)
**if** $y_{max} - y_{min} < x_{max} - x_{min}$ **then**
    **while** $y_{min} < y_{max}$ **do**
        $y_{up} = \min(y_{max}, y_{min} + p)$
        $O_{cut}$ = RectangleClip($O, x_{min}, y_{min}, x_{max}, y_{max}$)
        $y_{min} = y_{min} + p$
        yield $O_{cut}$
    **end while**
**else**
    **while** $x_{min} < x_{max}$ **do**
        $x_{up} = \min(x_{max}, x_{min} + p)$
        $O_{cut}$ = RectangleClip($O, x_{min}, y_{min}, x_{max}, y_{max}$)
        $x_{min} = x_{min} + p$
        yield $O_{cut}$
    **end while**
**end if**

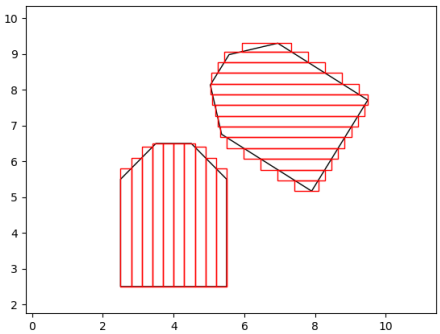
The function RectangleClip tries to find an intersection between the obstacle and given rectangle. Such function is defined and well optimized in various geometry engines (e.g. GEOS).



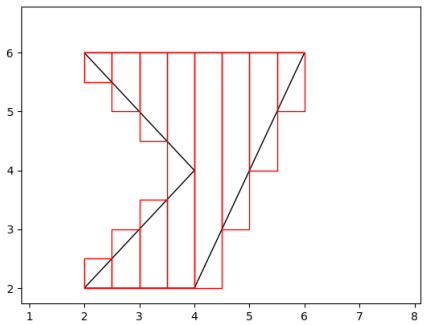
(a) Bounding box false collision



(b) Boxify p=0.5



(c) Boxify p=0.3



(d) Boxify p=0.5

Figure 4.1: An example of inefficient collision check with bounding boxes compared to the proposed function with different parameter.

## 4.2 Sampling

By default, RRT algorithm picks a random point $q \in \mathcal{C}_{free}$ and tries to build a trajectory from the nearest point in the tree to it (at the first iteration the nearest point is set to be $q_i$). The way to determine that the point is reachable is usually constructing a straight line between the sample point and the existing one. Then, if trajectory is built, the point is added the tree.

A common enhancement to this process is goal-biased sampling. In this case, with a parameterized probability $g_r$ instead of choosing a random point in $C_{free}$. This approach, however, has a flaw. If the tree has grown close to the $q_g$ with only one branch with last node $q_1$ and due to nonholomonic constraints cannot reach the $q_g$, all later samples that were changed on $q_g$ will get trapped by $q_1$, which has minimal distance to the $q_g$, until a new point $q_2 : dist(q_2, q_g) < dist(q_1, q_g)$ is added to the tree. In order to increase the chances of "untrapping" the $q_g$, we propose the following: since after sampling a point or choosing the $q_g$ as the sample point the algorithm tries to find the nearest nodes already added to the tree, one could select a random node from the set of $n$-closest nodes without any computational drawback.

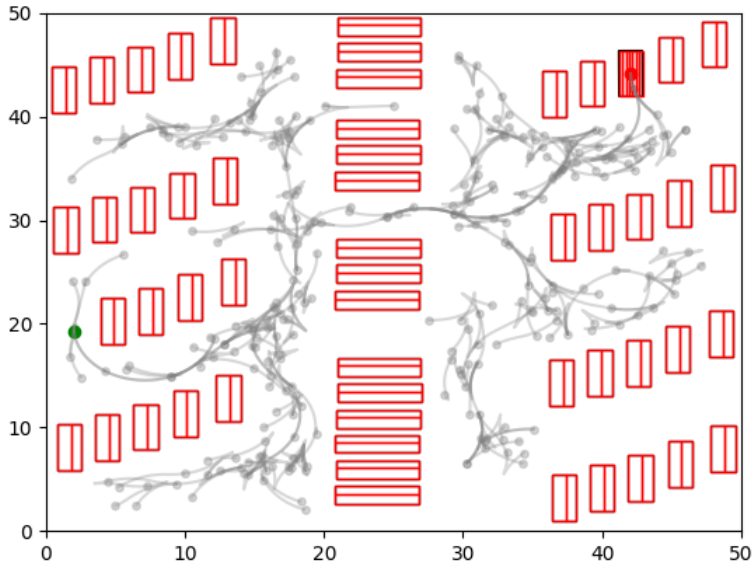



Figure 4.2: An example of a vehicle nonholonomically trapped due to angular precision. It stands at the finish point in the wrong direction. RRT with goal sampling will fail to find a trajectory at each $q_g$ check.

Additionally, instead of computing Euclidean distances to all nodes, our algorithm queries the R-Tree on the closest neighbors and computes the Reeds-Shepp distance to them. The final step of the sampling process involves checking feasibility of the provided trajectory. For each point $q$ provided by the trajectory generator (steering function) the algorithm performs $Boxify(\mathcal{A}(q), p)$. Other researches proposed converting $\mathcal{O}$

into TP-Space (Trajectory Parameter-Space), which would represent an intersection of $\mathcal{C}_{free}$ and kinodynamically feasible trajectories available in $\mathcal{A}(q)$. However, this would strongly increase the dimension of the $\mathcal{C}_{free}$ and would require storing nodes in the same dimension, which then would lead to higher complexity in the nearest neighbor queries on the RRT tree data structure. In our implementation $\mathcal{C}_{free}$ stays in $SE(2)$ and the maximum number of collision checks $\mathcal{O}_{checks}$ depends on the amount of iterations $k$, RRT maximum edge length $\Delta t$, chosen shape approximation discretization factor $p$ and trajectory discretization factor $\delta$.

$$\mathcal{O}_{checks} \leq k \cdot p \cdot \frac{\Delta t}{\delta} \tag{4.1}$$

**Algorithm 4.2** Complete BR-RRT

Constants: R-Tree with obstacles $rt_{obs}$
Environment and Car Inputs: $\mathcal{C}$, $q_i$, $q_g$, $precision$, $\mathcal{A}$
Parameters Inputs: $\Delta t$, $K$, $g_r$, $p$, $\delta$, $n$
$rt_{tree}$ = R-Tree($g_i$)
**for** $k$ in $1..K$ **do**
    **if** $random(0,1) > (1 - g_)$ **then**
        $q_s = q_g$
    **else**
        $q_s = q_{rand} \in \mathcal{C}_{free}$
    **end if**
    $nodes$ = NNQuery($rt_{tree}, q_s, n$)
    $q_n$ = RandomChoice($nodes$)
    $path$ = Reeds-Shepp($q_n$, $q_s$)
    **for** $q$ in $path$ **do** $boxes$ = Boxify($\mathcal{A}(q)$)
        **for** $box$ in $boxes$ **do**
            **if** IntersectionQuery($rt_{obs}, box$) $> 0$ **then**
                continue
            **end if**
        **end for**
    **end for**
    **if** InGoalRegion($q_n, q_g, precision$) **then**
        return Node$_s$
    **end if**
    Node$_n$ = GetNode($q_n$)
    Node$_s$ = Node$\{position = q_s, root =$ Node$_n, path = path\}$
    Add($rt_{tree}$, Node$_s$)
**end for**
return "Not enough iterations"

## 4.3 Testing methodology

### 4.3.1 Car Model

Since the research is conducted for car-like robots autopilots, for testing purposes we have conducted a brief analysis of local car market and chose parameters of one of the

best selling cars.

The polygonal model of the car is simplified to a rectangle, but the implementation can accept any polygonal object as a car, and the algorithm was designed for complex-shaped cars. The model has the following parameters: $length = 4.42m$, $width = 1.7m$ and turning radius of $r_{turn} = 5.12m$.

For the car model discretization factor $p$ of the Boxify algorithm was set to $0.25$, which means approximation to rectangles with smallest edge of at most $0.25cm$. The position $q$ is defined by coordinates of the centroid of the car.

### 4.3.2 Map Model

For the map we have chosen an arbitrary parking problem that was designed to include complex scenarios. The parking ($50x50m$) is filled with same cars as our testing car. Obstacle discretization factor $p$ is set to $p = 1$ ($1m$).

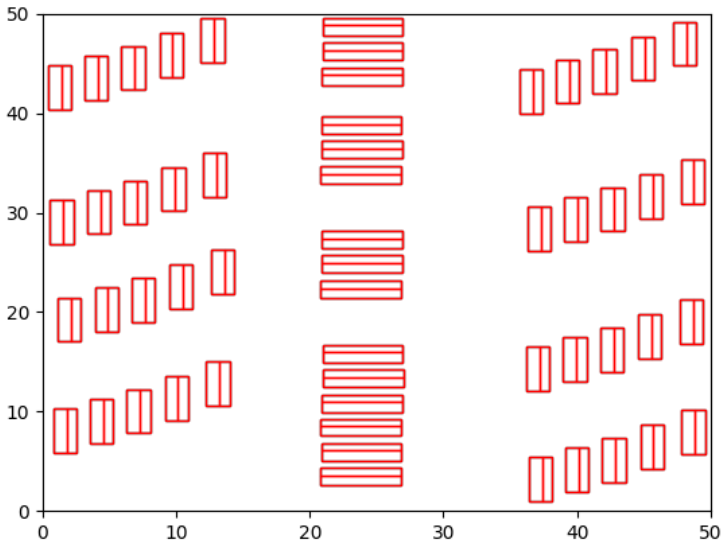


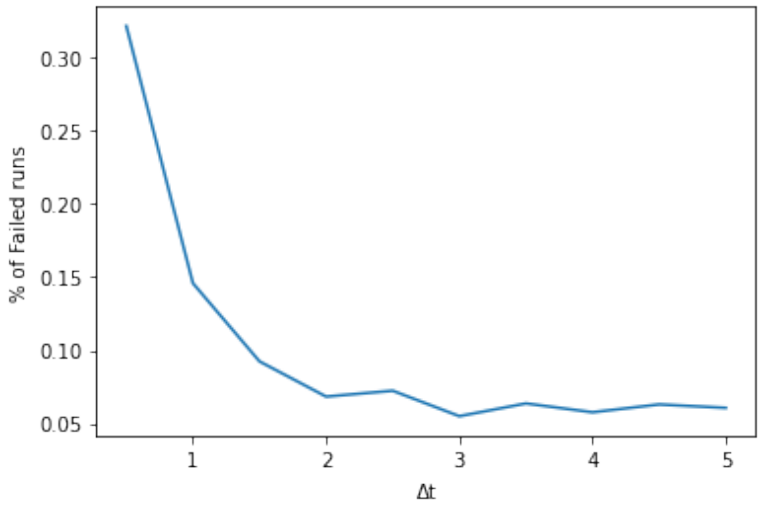

Figure 4.3: Large parking model for conducting tests

### 4.3.3 Testing System

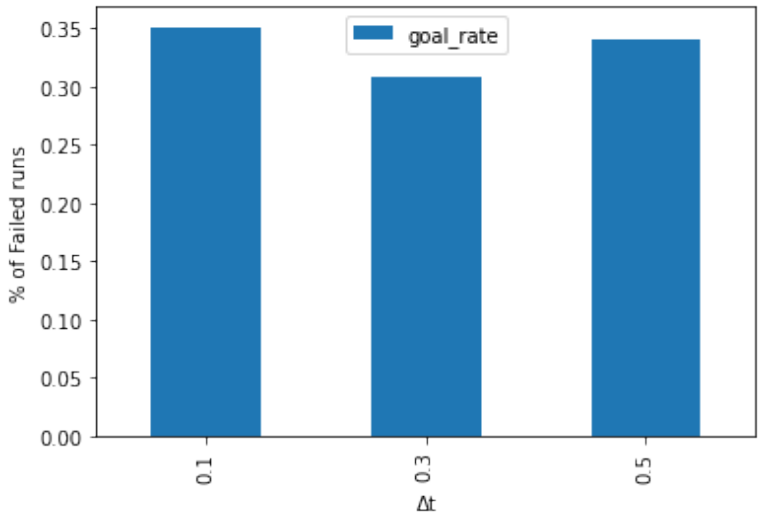
Each algorithm was tested on multiple different values of all parameters. For each set of parameters the testing system chose two random cars-obstacles from the map, removed them and inserted $q_i$ and $q_g$ instead. The testing car was then moved to $q_i$ and rotated accordingly. This environment setup was then passed to BR-RRT and the path was calculated along side with additional statistical data. This process was repeated 100 times for each set of parameters in parallel. Due to parallel execution, in this particular testing there will not be any analysis of absolute computation time. In general, since the reference implementation is written in Python and executed on a powerful machine that might not be a good representation of robot's hardware, absolute values of time are not very useful.

## 4.4 Baseline RRT Results

For the baseline algorithm we have chose not a classical RRT, but an improved one. This algorithm uses goal-biased sampling and the proposed Boxify method for the collision detection. The latter condition does not affect testing as it was enabled in all tests.




(a) Distribution grouped by $\Delta t$




(b) Distribution grouped by $g_r$

Figure 4.4: Distribution of failed runs in baseline RRT for different parameters

In the baseline RRT the maximum edge length, that can be added to the RRT tree, plays the most significant role in its completion rate [Fig. 4.4a]. From the graph we observe that $\Delta t = 0.5$ accounts for more than 30% of fails. $\Delta t >= 3.0$ shows the best results and can be considered as the best. It is important to understand why it is so. RRT has a parameter of iterations limit. On a large map, like the one that is shown at Fig. 4.3, a path from a random $q_i$ to a random $q_g$ can be long even in Euclidean terms. If the $\Delta t$ is too small, the distance between $q_i$ and $q_g$ can be just greater than $K \cdot \Delta t$. However, it still shouldn't be considered an artificial disadvantage of small $\Delta t$, because from Equation 4.1 we know that amount of iterations directly affect the amount of performed collision checks.

**4.5**