Rapidly-exploring Random Trees

Team: JSqaured

Jaidev Shriram (2018101012)
Jyoti Sunkara (2018101044)

February 20, 2021

Contents

1 Objective **3**

2 General Algorithm **3**

3 Obstacle Map and Configuration Space **3**

3.1 Obstacles 3

3.1.1 Implementation 4

3.2 Collision 4

4 Holonomic Robot **4**

4.1 Algorithm 5

4.2 Implementation 5

5 Non-Holonomic Robot **5**

5.1 Robot 6

5.2 Kinematics 7

5.3 Algorithm 7

5.4 CSC Trajectories 7

5.5 CCC Trajectories 8

6 Tangent Lines Geometry **9**

6.1 Inner tangents 9

6.2 Outer tangents 13

7 Results **13**

1 Objective

The goal of RRT is to compute a kinematically feasible trajectory for a robot from a start to final position, on a given map filled with obstacles.

2 General Algorithm

There are two phases to RRT - exploration and motion planning. In the exploration phase, we iteratively build our exploration tree on the map. Once the tree reaches the endposition, we halt. Using the computed graph, we plan our trajectory for the robot.

The algorithm is essentially:

1. for k=1 to K
   (a) Get Random state X
   (b) Find nearest neighbour in tree, to state X
   (c) Create a new node in the tree, that moves closer to X (expand nearest neighbour)
   (d) If the above operation, doesn't result in a collision, create the node.
2. Once the algorithm is done, use a path planning algorithm to compute the shortest path (if it exists).

3 Obstacle Map and Configuration Space

The RRT algorithm is run on a pre-computed map. This map has a list of obstacles, which can take the shape of arbitrary polygons. The robots task is to navigate this map without colliding with any of the obstacles - in the physical space.

For ease, we consider path planning in the configuration space. The configuration (or parametric) space is the parametric space where the point robot can move, or the points in this space correspond to the degrees of freedom of the robot. Calling the grow method for all obstacles sets up the configuration space, allowing us to shrink the robot to a point.

3.1 Obstacles

Obstacles represent non-navigable areas on the map. We support both regular polygons and complex shapes. This is done via a geometry module that helps with collision calculation.

3.1.1 Implementation

Obstacles are defined by a specific obstacle class which has definitions for - its shape (such as radius, center point) and other essential functions. Some obstacle classes are ‘CircleObstacle’, ‘RectObstacle’, ‘RandomPolygon’ - all of which inherit from the ‘Obstacle’ class. Hence, these obstacles have identical interfaces but different implementations.

Obstacles are instantiated using a dictionary. This dictionary specifies various parameters of the obstacle such as colour and shape.

The variable ‘baseshape’ stores the pure obstacle. On calling the ‘grow’ method, we compute the obstacle configuration space for the current shape, which is stored in the ‘shape‘ variable. This is done by computing the Minkowski sum of the robot and the obstacle. The robot shape is also specially defined as a shape - circle or a robot in our case.

3.2 Collision

The obstacle classes have definitions to check if a line passes through an obstacle. This helps ensure that no line in the RRT passes through the obstacle. There is also a function to check if a given point lies within an obstacle - used to ensure that the random points chosen for RRT aren’t part of any obstacle.

4 Holonomic Robot

A holonomic robot has the degree of control as the same as degree of freedom of the configuration space. This is a simple case for path planning, since the kinematics of the robot are simple. While there are many holonomic robots, we chose to implement a **synchro** drive robot.

A synchro drive robot is one which can move in any direction, due to wheels that can change its orientation.

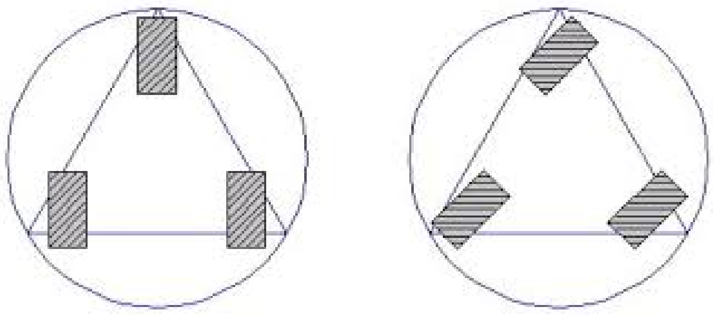


Figure 1: Sample Synchro Robot

The above image shows a sample synchro robot, which can move its wheels in any direction. We use a four wheel variant of this, though there are no differences practically with this three wheel system.

4.1 Algorithm

The algorithm for the holonomic case is exactly as mentioned earlier. Since there are no kinematic constraints, it is straightforward.

4.2 Implementation

The holonomic code is largely broken down into five classes - for low coupling and greater modularity:

- Map
- Graph
- Obstacle (and ObstacleList)
- Synchro
- RRT

The RRT class interfaces with the Map class to create new nodes and generate the tree. The map in turns create a graph to store this information through the graph class. Since the map also holds the obstacles, it maintains a list of obstacles. To check for collisions, it simply calls a defined functions in the obstacles. The map object also has a function to draw the obstacles.

The obstacles are coloured in dark shades, while their configuration space, or the result of minkowsky sum is in a lighter transulcent shade.

The obstacles store the shape, position, and obstacle configuration space. Since shapes can vary, we have varying implementations of collision detection here.

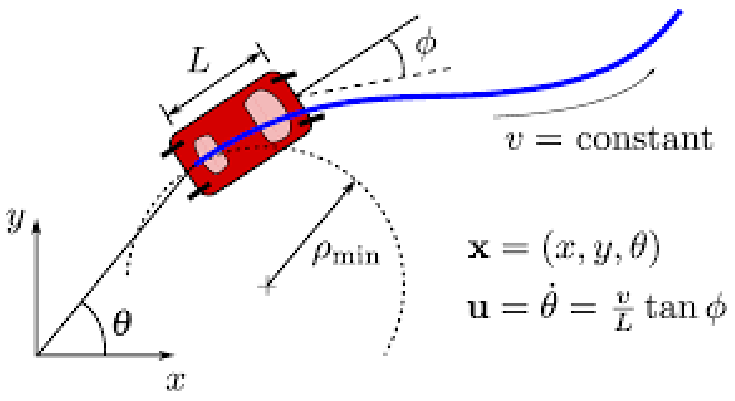
The Synchro class stores the object shape, and encodes the vehicles kinematics. Passing the robot to the map helps build the configuration space using Minkowsky sum with obstacles. To visualise the wheel trajectories, we simply pass the obtained path from RRT to the kinematic function.

5 Non-Holonomic Robot

A non-holonomic robot is one where the controllable degree of freedom is less than the total degrees of freedom. A car has three degrees of freedom which are its position in two axes and its orientation. However, there are only two controllable degrees of freedom which are acceleration and turning angle of steering wheel. We have chosen to implement a **Dubins car**.

A Dubins path refers to the shortest curve that connects two points in the two-dimensional Euclidean plan with a constraint on the curvature of the path and with prescribed initial and terminal tangents to the path, and an assumption that the vehicle traveling the path can only travel forward.

5.1 Robot



L
ϕ
$v =$ constant
y
$\rho_{\min}$
$\mathbf{x} = (x, y, \theta)$
$\mathbf{u} = \dot{\theta} = \frac{v}{L} \tan\phi$
θ
x

Figure 2: Dynamics for the Dubins car model

A Car's configuration can be describe by the triplet $\langle x, y, \theta \rangle$. Define the car's velocity (actually speed, because it will be a scalar quantity) as v. When a car is moving at velocity v about a circle of radius r_{turn} , it will have angular velocity

will have angular velocity

$$\omega = \frac{v}{r_{turn}}$$

Now we define how our system evolves over time. When we want to describe how our system evolves over time, we use notation like $\dot{x} totalkabouthowourxcoordinatechangesovertime$.

In basic vector math, if our car is at position $A = (x_1, y_1)$ with $\theta = \theta_1$, and we move straight forward for a single timestep, then our new configuration is $B = (x_1 + vcos\left(\theta_1\right), y_1 + vsin\left(\theta_1\right), \theta_1)$ Note how our x coordinate changes as a function of $cos\left(\theta\right)$. Therefore, we'd say that $\dot{x} = cos\left(\theta\right)$. A full system description would read like this:

$$\dot{x} = cos\left(\theta\right)$$

$$\dot{y} = sin\left(\theta\right)$$

$$\dot{\theta} = \omega = \frac{v}{r_{turn}}$$

5.2 Kinematics

A Dubins car essentially has only 3 controls:

- Turn left at maximum
- Turn right at maximum
- Go straight

All the paths traced out by the Dubin's car are combinations of these three controls. The controls are formally named in the following manner: Turn left at maximum is L, Turn right at maximum is R, and Go straight is S. Things are made even more general: left and right turns both describe curves, so they are grouped under a single category that we'll call C (”curve”) and the remaining control is grouped as S(”straight”). Lester Dubins proved in his paper that there are only 6 combinations of these controls that describe all the shortest paths, and they are: RSR, LSL, RSL, LSR, RLR, and LRL.Or using the more general terminology, there are only two classes: CSC and CCC.

5.3 Algorithm

5.4 CSC Trajectories

The CSC trajectories include RSR, LSR, RSL, and LSR which is a turn followed by a straight line followed by another turn.

We pick a position and orientation for the start and goal configurations. Next, we draw circles to the left and right of the car with radius r_{min} . The circles should be tangent at the location of the car. To visualize one can draw tangent lines from the circles at the starting configuration to the circles at the goal configuration.

For each pair of circles (RR), (LL), (RL), (LR), there should be four possible tangent lines, but there is only one valid line for each pair. That is, for the RR circles, only one line extending from the agent's circle meets the goal's circle such that everything is going in the correct direction. Therefore, for any of the CSC Trajectories, there is a unique tangent line to follow. This tangent line makes up the ‘S’ portion of the trajectory. The points at which the line is tangent to the circles are the points the agent must pass through to complete its trajectory.

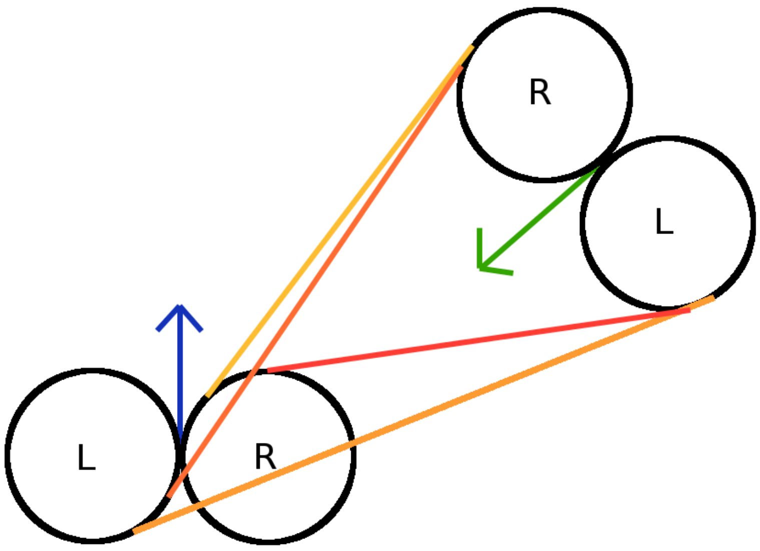


Figure 3: Valid tangent lines

5.5 CCC Trajectories

CCC trajectories consist of a turn in one direction followed by a turn in the opposite direction, and then another turn in the original direction. They are only valid when the agent and its goal are relatively close to each other, else one circle would need to have a radius larger than r_{min} which is sub-optimal. For the Dubin's Car there are only 2 CCC Trajectories: RLR and LRL. The third circle that we turn about is still tangent to the agent and goal turning circles, but these tangent points are not the same as those from tangent line calculations. Therefore, solving these trajectories boils down to correctly computing the location of this third tangent circle.

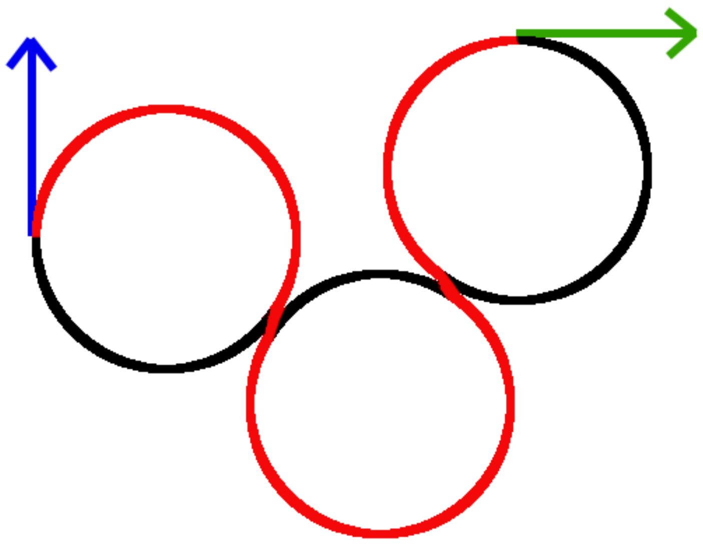


Figure 4: Computing a RLR Trajectory

6 Tangent Lines Geometry

6.1 Inner tangents

1. First draw a vector $\vec{V_1}$ from p_1 to p_2. $\vec{V_1} = (x_2 - x_1, y_2 - y_1)$. This vector has a magnitude:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

2. Construct a circle C_3 centered at the midpoint of $\vec{V_1}$ with radius $r_3 = \frac{D}{2}$ That is,

$$p_3 = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

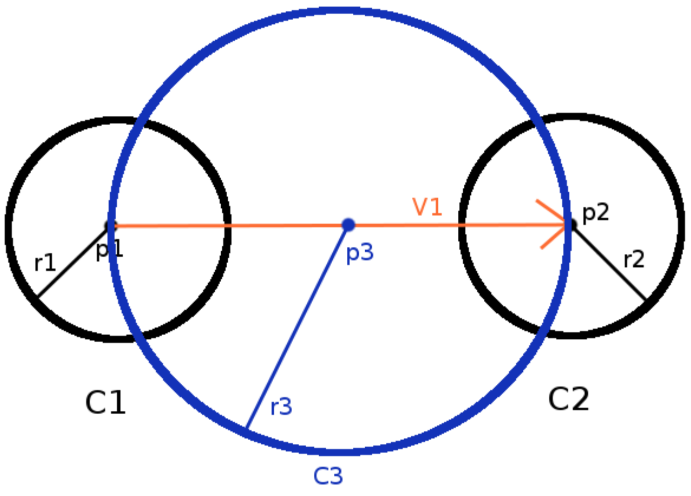


Figure 5: Inner Tangents Steps One And Two

3. Construct a circle C_4 centered at C_1's center, with radius $latexr_4 = r_1 + r_2$

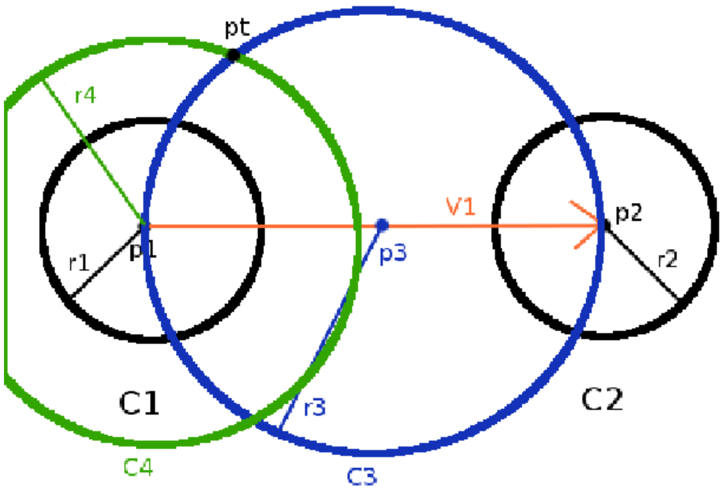


Figure 6: Inner Tangents Step Three

4. Construct a vector $\vec{V_2}$ from p_1 to the "top" point, $p_t = (x_t, y_t)$ of intersection between C_3 and C_4 . If we can compute this vector, then we can get the first tangent point because $\vec{V_2}$ is pointing at it in addition to p_t .

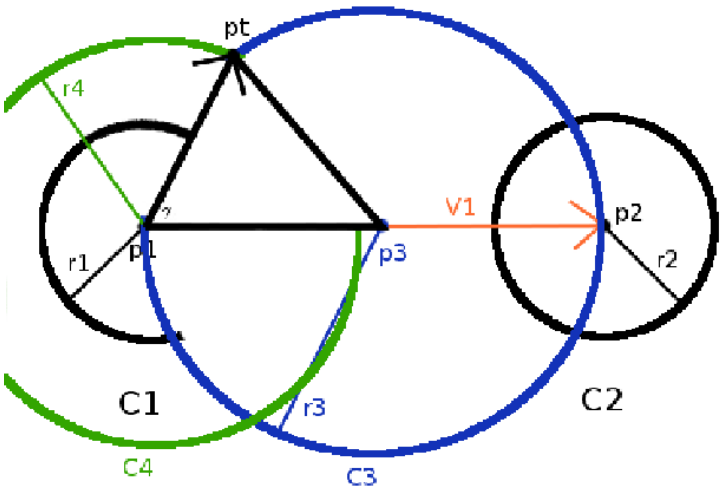


Figure 7: Inner Tangents Step Four

5. This is accomplished by first drawing a triangle from p_1 to p_3 to pt. The segments $\overline{p_1 p_3}$ and $\overline{p_3 p_t}$ have magnitude $r_4 = \frac{D}{2}$. The segment $\overline{p_1 p_t}$ has magnitude $r_3 = r_1 + r_2$. We are interested in the angle $\gamma = \angle p_t p_1 p_3$. γ will give us the angle that vector $\vec{V_1}$ that would need to rotate through to point in the same direction as vector $\vec{V_2} = (p_t - p_1)$. We obtain the full amount of rotation about the x axis, θ for $\vec{V_2}$, by the equation $\theta = \gamma + atan2\left(\vec{V_1}\right)$

p_t is therefore obtained by traversing $\vec{V_2}$ for a distance of r_4 . That is,

$$x_t = x_1 + (r_1 + r_2) * cos\,(\theta)$$

$$y_t = y_1 + (r_1 + r_2) * sin\,(\theta)$$

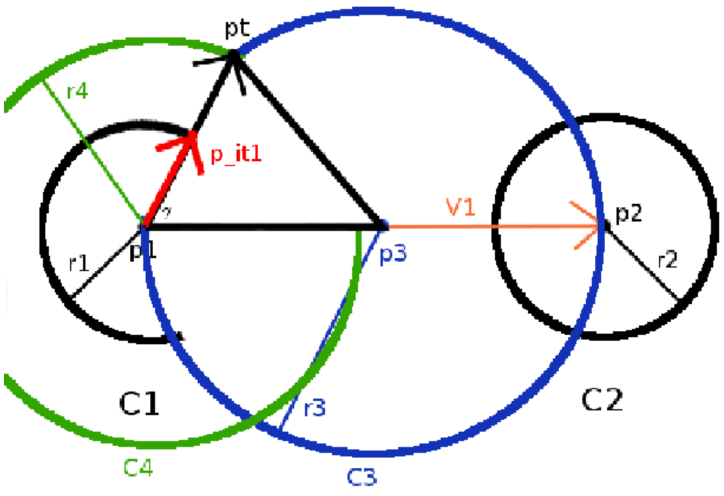


Figure 8: Inner Tangents Step Five

6. To find the first inner tangent point pit1 on C1 , we follow a similar procedure to how we obtained pt — we travel along $\vec{V_2}$ from p_1 , but we only go a distance of r_1 . Because we know p_t , we are now able to actually compute $\vec{V_2} = (p_t - p_1)$. Now, we need to normalize $\vec{V_2}$ and then multiply it by r_1 to achieve a vector $latex \vec{V_3}$ to $latex p_{it1}$ from p_1. (Remember that, to normalize a vector, you divide each element by the vector's magnitude $\rightarrow \vec{V_3} = \frac{\vec{V_2}}{\|\vec{V_2}\|} * r_1 . p_{it1}$ follows simply:

$$p_{it1} = p_1 + \vec{V_3}$$

7. Now that we have p_t , we can draw a vector $\vec{V_4}$ from p_t to p_2. Note that this vector is parallel to an inner tangent between C_1 and C_2

$$\vec{V_4} = (p_2 - p_t)$$

We can take advantage of its magnitude and direction to find an inner tangent point on C_2 . Given that we've already calculated p_{it1} , getting its associated tangent point on C_2 , p_{it2} is as easy as:

$$p_{it2} = p_{it1} + \vec{V_4}$$

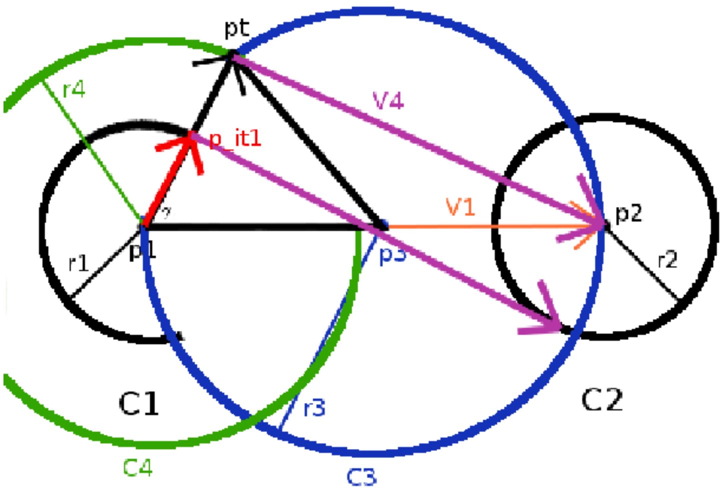


Figure 9: Inner Tangents Step Seven

6.2 Outer tangents

Constructing outer tangents is very similar to constructing the inner tangents. Given the same two circles C_1 and C_2 as before, and assuming $r_1 \geq r_2$ Remember how we started off by making C_4 centered at p_1 with radius $r_4 = r_1 + r_2$? Well this time around we'll construct it a bit differently. C_4 is centered at p_1 as before, but this time it has radius $r_4 = r_1 - r_2$.

Following the steps we performed for the interior tangent points, constructing C_3 on the midpoint of $\vec{V_1}$ exactly the same as before. Find the intersection between C_3 and C_4 to get p_t just as before as well. After we've gone through all the steps as before up to the point where we've obtained $\vec{V_2}$, we can get the first outer tangent point p_{ot1} by following $\vec{V_2}$ a distance of r_1 just as before. I just wanted to note that the magnitude of $\vec{V_2}$ before normalization is $r_4 > r_1$ instead of $r_4 > r_1$. To get p_{ot2} , the accompanying tangent point on C_2 , we perform addition:

$$p_{ot2} = p_{ot1} + \vec{V_4}$$

This is exactly the same as before. In essence, the only step that changes between calculating outer tangents as opposed to inner tangents is how C_4 is constructed. All other steps remains exactly the same.

7 Results

RRT Holonomic

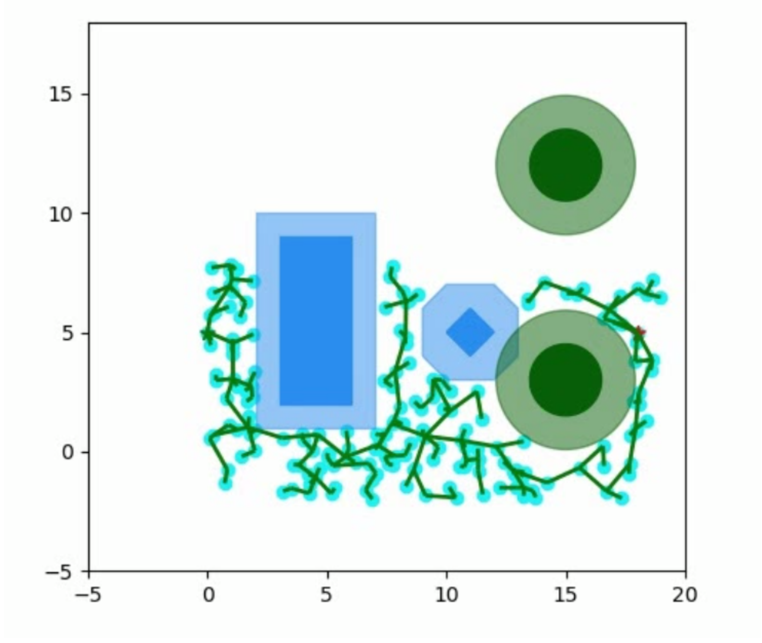


Figure 10: RRT Holonomic In Configuration Space

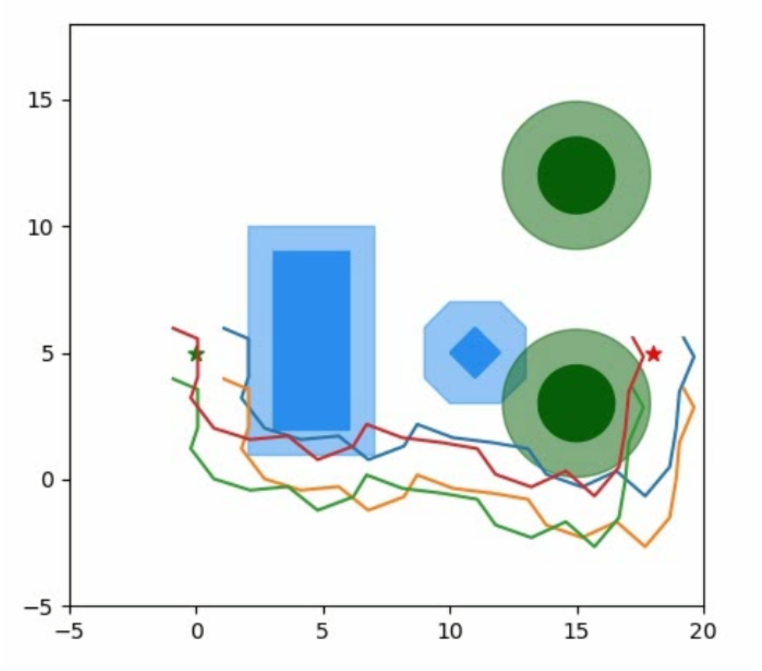


Figure 11: RRT Holonomic Wheel Trajectories

RRT Non-Holonomic

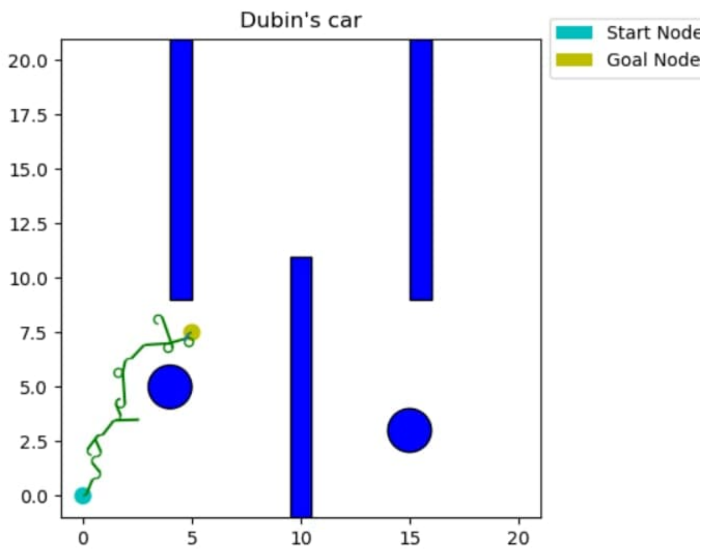


Figure 12: RRT Non-Holomic Dubins Car

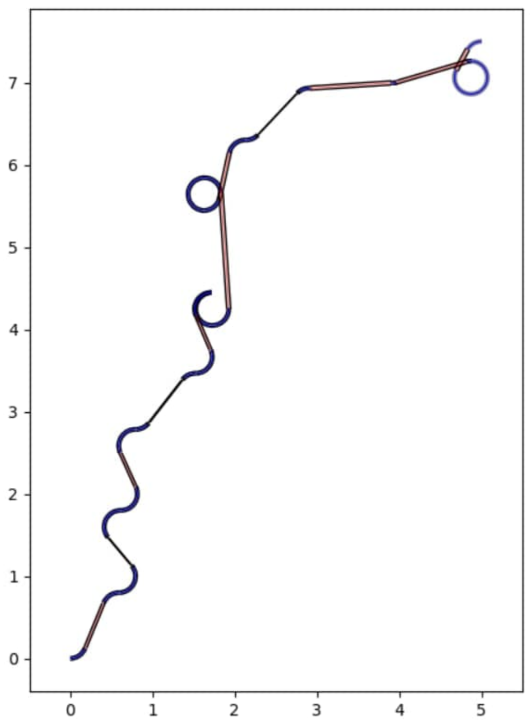


Figure 13: RRT Non-Holonomic Wheel Trajectories

Video Link
https://tinyurl.com/RRTVideos

Work Division
Jaidev Shriram

- Configuration Space
- Minkowski Sum
- Holonomic Wheel Trajectories
- Non-Holonomic Wheel Trajectories

Jyoti Sunkara

- Base RRT Algorithm
- Holonomic Path Planning
- Non-Holonomic Path Planning and Kinematics