Kinematic planning under Nonholonomic Constraints

Shambhuraj Anil Mane

October 30, 2023

Abstract

This report presents the results of implementing a kinematic path planner to park three vehicles of increasing complexity into a tight space. The vehicles - a delivery robot, car, and truck with trailer - each have different steering constraints. A simulated 2D environment was created with obstacles, and configurations space was modeled accounting for nonholonomic constraints. Custom path planning algorithms generated feasible paths satisfying kinodynamic constraints to maneuver each vehicle from a start position into the target parking spot. The planned paths are presented along with snapshots visualizing the parking maneuvers. Though simplistic, this project demonstrates automated parking for nonholonomic vehicles in cluttered environments.

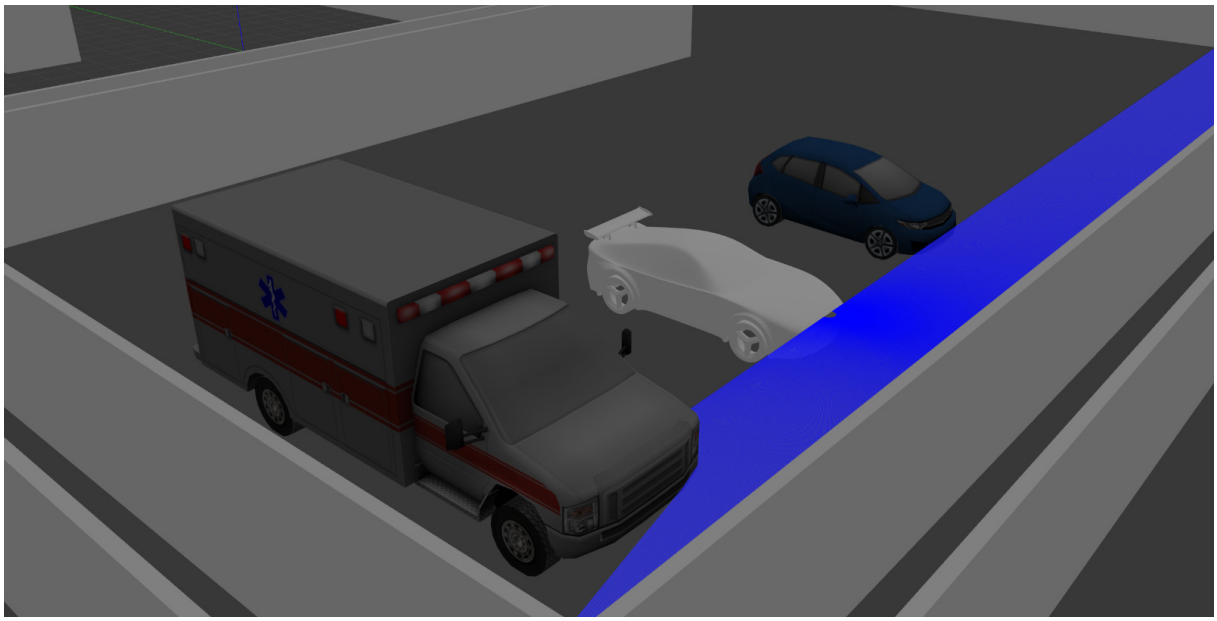


Figure 1: Autonomous Parking Simulation

1 Introduction

This project involves developing path planning algorithms to park vehicles in a cluttered environment. The objective is to maneuver three vehicles - each with different steering constraints - from a start position to a tight parking spot, while avoiding collisions. This is a common challenge faced by autonomous vehicles operating in urban environments.

The simulated 2D world consists of a parking lot with obstacles flanking the target parking space on two sides. Additionally, there are multiple obstruction in the middle of the lot. The vehicles must navigate around these obstacles and utilize their full range of steering motion to efficiently park in the compact spot.

The three vehicles of increasing complexity are:

- Delivery robot: Differential drive/skid steering
- Car: Ackermann steering
- Truck with trailer: Ackermann steering plus trailer kinematics

Each vehicle has kinodynamic constraints in the form of nonholonomic constraints dictated by its steering mechanism. The path planner utilizes a hybrid A* algorithm which takes into account these constraints to generate feasible trajectories.

This project demonstrates automated parking for vehicles with different steering configurations in a constrained space using a custom motion planning algorithm.

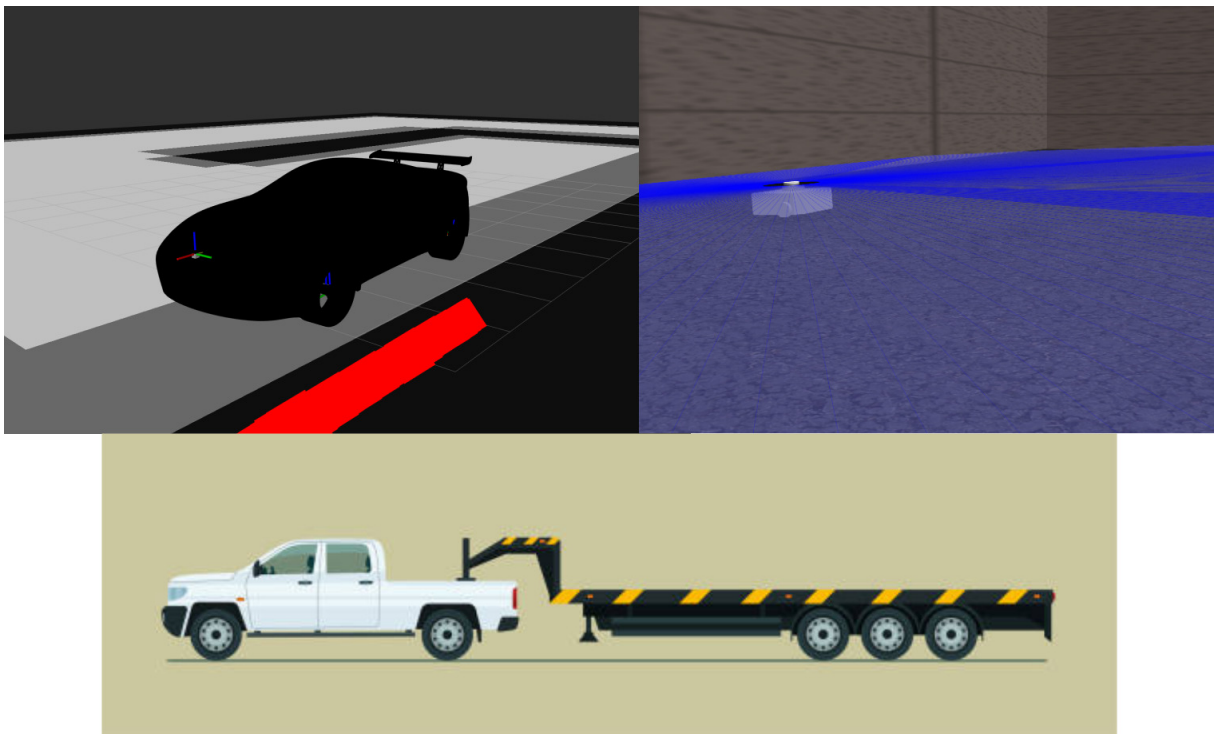


Figure 2: Three types of Vehicles

2 Methods

The path planning approach taken involves using a hybrid A* search algorithm to generate optimal trajectories for parking each vehicle. This extends the traditional A* graph search by incorporating kinodynamic constraints in the cost function.

- Simulation

  The complete parking scenario was simulated using pyplot as well as ROS2 and Gazebo. Pyplot provided ease in visualizing and working on algorithms in 2D environment and ROS2 provided the infrastructure for controlling the vehicles and visualizing the environment. Gazebo handled the physics simulation and rendering.

- Collision Checking

  Continuous spaces are mathematically challenging to work with, especially when planning paths and performing collision checks. Discretizing the space simplifies the problem by dividing it into smaller, manageable states. It allows the algorithm to check for collisions with obstacles by examining a finite number of states, rather than continuously evaluating the entire continuous space.

- Discrete Motion Planning with Nonholonomic Constraints

  Kinodynamic planning is crucial for vehicles and robots with non-holonomic constraints, such as cars, or robots with differential drive systems. These systems cannot instantly change their velocity or direction and have specific dynamics that must be considered to plan feasible trajectories. The diagram and explanation on this topic is given in the appendix section.

- Hybrid A-star algorithm

  The path planning approach taken involves use of a hybrid A* search algorithm to generate optimal trajectories for parking each vehicle. This extends the traditional A* graph search by incorporating kinodynamic constraints in the cost function.

  At each iteration, the algorithm considers reachable configurations based on the vehicle's kinematics. It selects the lowest cost node based on the A* evaluation function: f(n) = g(n) + h(n). The function g(n) represents the path cost from start to node n, while h(n) estimates the cost to reach the goal. The kinodynamic constraints are encoded in g(n) to prune infeasible motions.

  This process repeats, expanding nodes until the goal is reached. The optimal path minimizing traversal cost while satisfying steering constraints is then extracted. This path is converted to a time-parameterized trajectory using the kinematic model.

The complete scenario is simulated in matplotlib and ROS2 with Gazebo managing the physics and visualizing the environment.

3 Results

The hybrid A* path planner successfully generated collision-free trajectories to park all three vehicles in the allotted space.

For the delivery robot, a smooth path was produced utilizing the differential drive constraints to neatly maneuver around the obstacles to the goal position. The planned path is shown in Figure overlaid on the environment map.

The sedan also navigated the cluttered parking lot efficiently as seen in Figure. The Ackermann steering constraints resulted in wider turns compared to the delivery robot.

Parking the truck with trailer was most challenging due to the trailer kinematics. The planner accounted for these constraints and planned a feasible path to the goal as shown in Figure. Executing the maneuver required careful tuning of velocities and accelerations.

All vehicles managed to avoid collisions and navigate within their steering limitations.

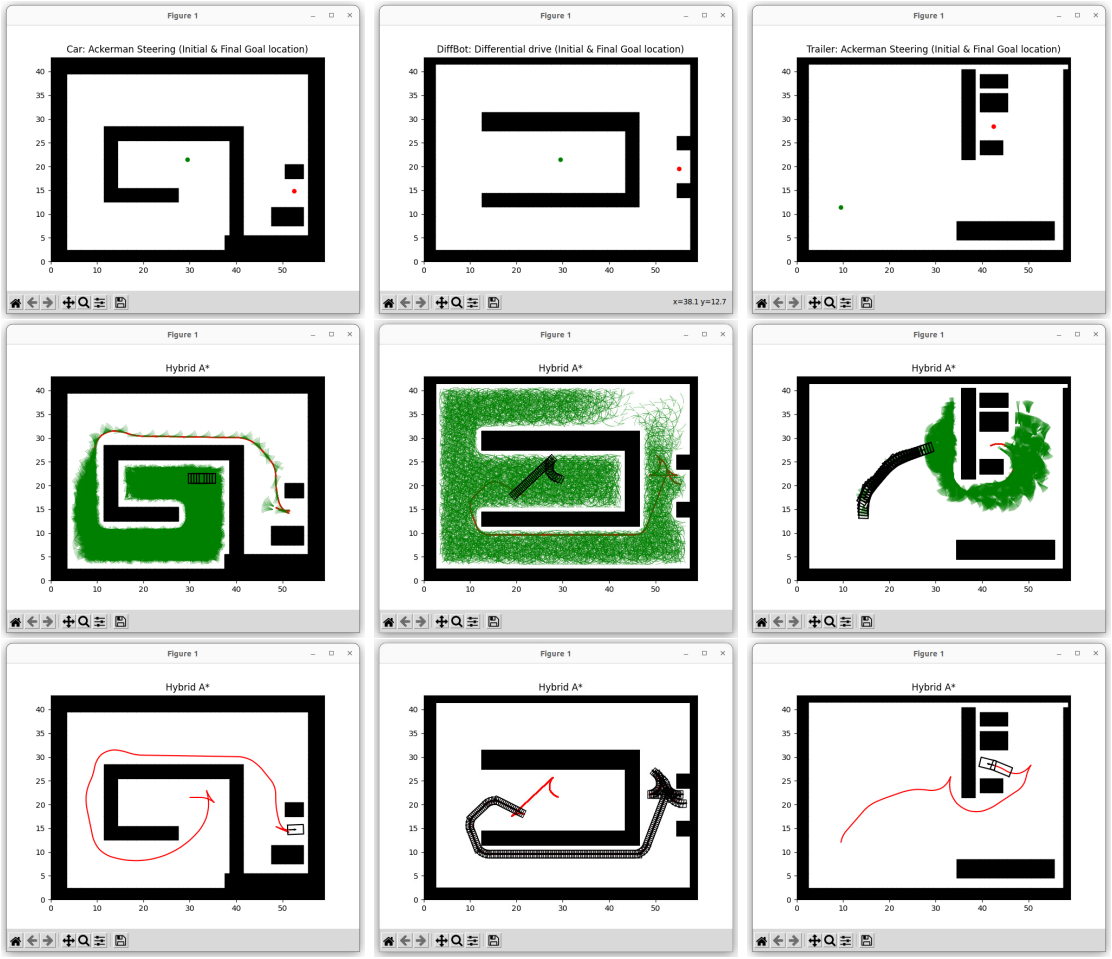


Figure 3: Planned path in all three scenarios

Cost functions have been instrumental in the path planning process. Even minor alterations in their values have had a significant impact on factors such as path length, exploration duration, and the number of directional changes.

The incorporation of heuristic costs effectively reduced exploration time, enabling the path to converge to the goal point more swiftly. However, it's worth noting that the

resulting path, while expedited, may not always be optimal in terms of both length and directional adjustments. On the other hand, when the heuristic cost is disregarded (hybridCost=0), it leads to more extensive space exploration, resulting in prolonged exploration times. Nevertheless, this approach tends to yield an optimized path in both length and directional adjustments. The following figures serve as conclusive evidence of the outcomes described above.

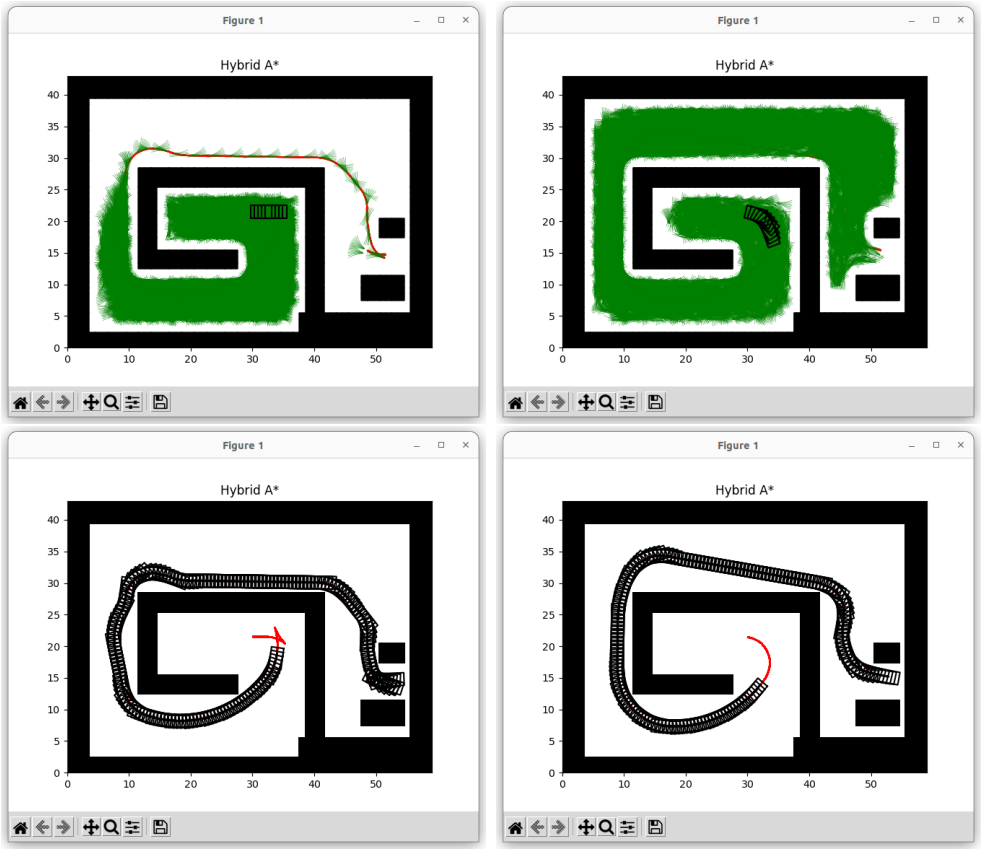


Figure 4: Planned path in all three scenarios

4 Conclusion

This project demonstrated successful path planning and control of various steered vehicles for parking in tight spaces. Optimal, collision-free trajectories were generated taking into account kinodynamic constraints of each vehicle model.

The hybrid A* motion planning algorithm was effective at producing smooth paths to the goal in reasonable time. On average, parking was completed within 20-35 seconds for the vehicles. The planner balances exploration of the configuration space with exploitation of lowest cost paths. Further tuning of heuristic functions could improve planning times.

While the planned paths were optimal in terms of length, the trajectories could be further optimized for time. Velocity profiles could be smoothed to minimize accelerations and jerks. Dynamic constraints could also be incorporated within the kinodynamic framework to generate time-optimal trajectories.

The system could be extended to more complex vehicles, tighter environments, and dynamic obstacles. Overall, this project demonstrated automated parking for nonholonomic vehicles, achieving the objective efficiently using motion planning methods.

References

5 Appendix

All the points from method section are explained below in detail.

5.1 Simulation

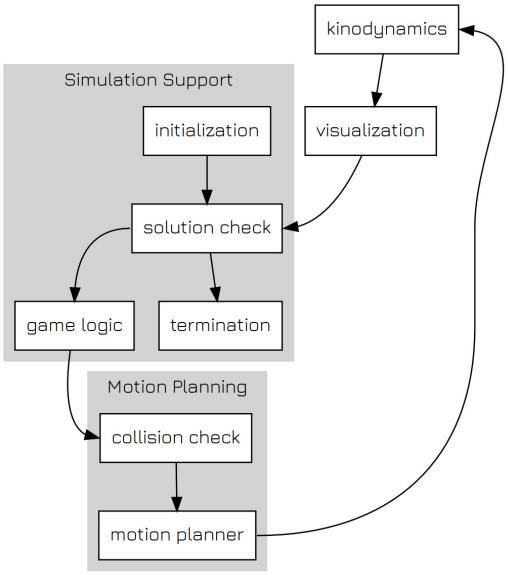


Figure 5: Simulation structure

As shown in Fig. following points are considered in this project.

Initialization: Initial robot state along with the Initial simulation states ie. obstacle positions were set.

Solution Check: Robot position at goal location and termination condition such as reaching to goal location or path not found were checked.

Game logic: Is not used as only statis obstacles are considered in this project.

Collision Checking: Collisions given the current robot configuration and world state were considered for collision free path planning.

Motion Planning: Motion planner is implemented using current robot state as input and trajectory fro robot controller was calculated.

Kinodynamics: Considering the the different steering mechanisms which commands for specific kinematics constraints and dynamic equations were considered in deciding the algorithm.

Visualization: Animation frames for each iteration were created using simplified graphics in Matplotlib as well as 3D visualization in Gazebo.

Simulation initialization stage images are shown below.

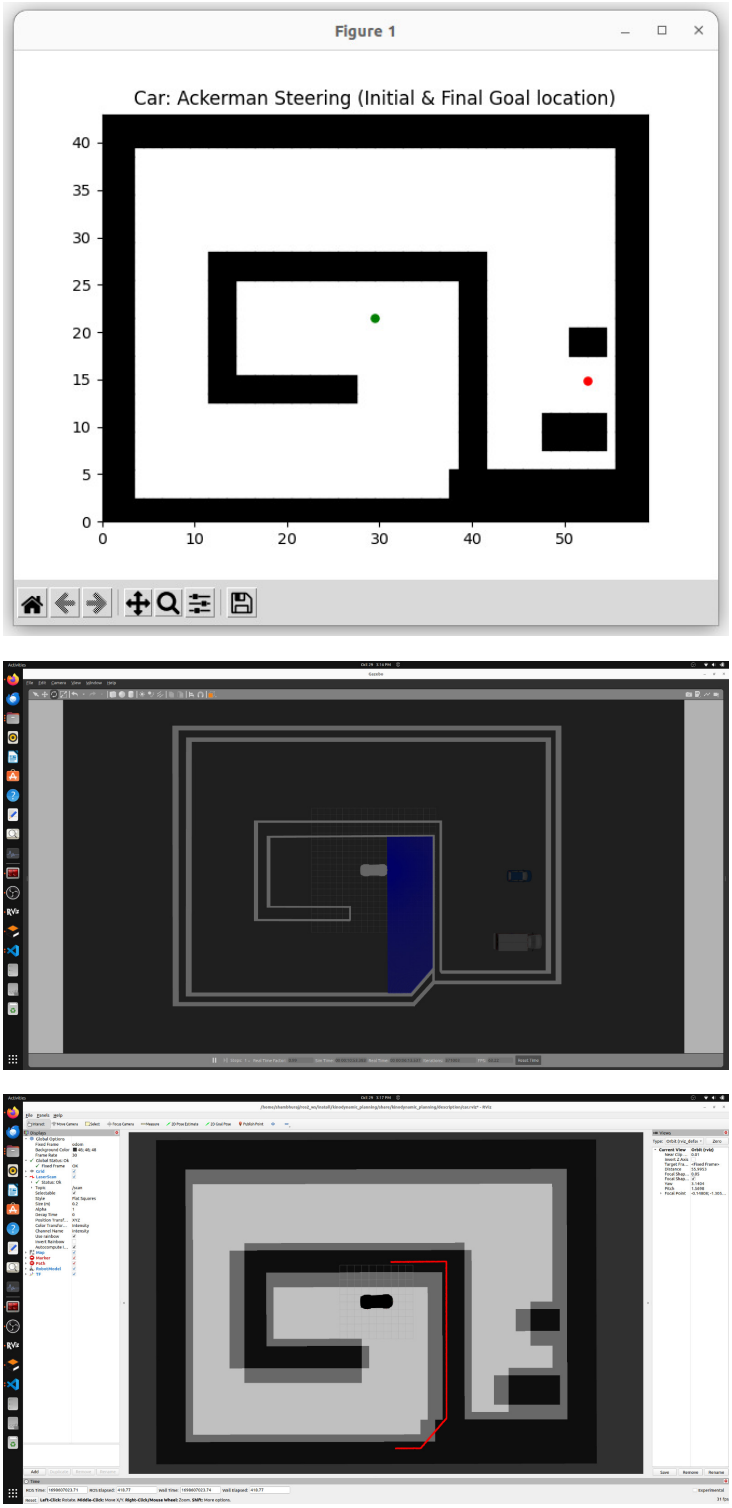


Figure 6: Simulation structure

5.2 Collision Checking

As shown in Fig. the environment is divided into Occupancy grid. Occupancy grids allow you to represent the environment as a grid where each cell indicates whether it is occupied by an obstacle or free space. This representation is particularly useful for motion planning because it provides a structured way to model the environment's obstacles and free areas. A module scipy.spatial.KDTree in Python Scipy is used in this project.

Using a KD-Tree for occupancy grid data can be beneficial for tasks such as nearest-neighbor search, collision checking, or other spatial queries. KD-Trees are a data structure that can efficiently partition and organize your occupancy grid data.

Occupancy grids have advantages like, well suited for discrete planners, low storage/memory requirements and tree can be utilized for better space and computational efficiency

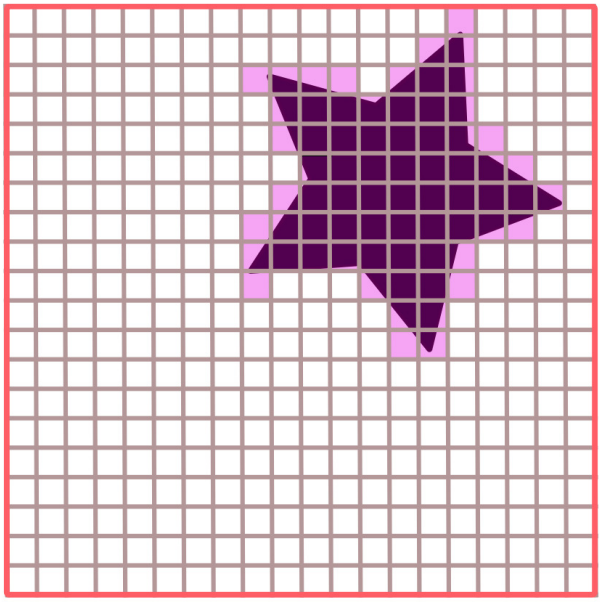


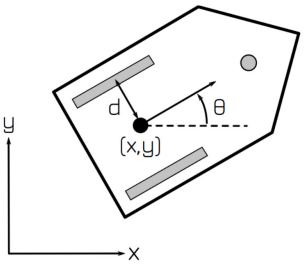
Figure 7: Occupancy grid

5.3 Discrete Motion Planning with Nonholonomic Constraints

For static planners the C-space representation is sufficient which takes into account robot's position and orientation. State space (X) incorporates the dynamic state and extend C-space by including time derivative of each dimension in the C-space. Working in state space allows planner to incorporate dynamic constraints on path doubles the dimensionality of the planning problem. Holonomic and non-holonomic constraints are concepts used in the field of mechanics and robotics to describe the constraints on the motion of objects or systems. Planners must incorporate robot dynamics and model the relationship between control inputs and state. These constraints may vary based on the specific vehicle configuration.

- Modeling of Delivery robot: Differential drive/skid steering

  Equations for modeling differential drive robot are shown in Fig.



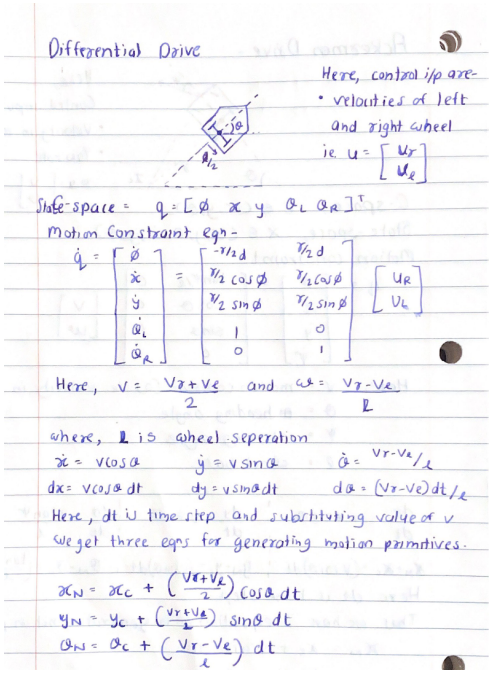


Figure 8: Modeling of Delivery robot: Differential drive/skid steering

Car class was used to define the dimension of the car in algorithm for given car model simulated in Gazebo environment.

class DiffBot:

speedPrecision = 4

wheelSeperation=0.287 * 5

botDiameter=0.32 * 5

wheelDiameter=0.066 *5

maxVelocity=2.0

minVelocity=1.0

stepSize=0.5

Calculations using these equations are shown below:

- Modeling of Car: Ackermann steering Equations for modeling differential drive robot are shown in Fig.

  Car class was used to define the dimension of the car in algorithm for given car model simulated in Gazebo environment.

  class Car:

  maxSteerAngle = 0.6

  steerPresion = 5

  wheelBase = 2.58

  axleToFront = 3.0

  axleToBack = 0.4

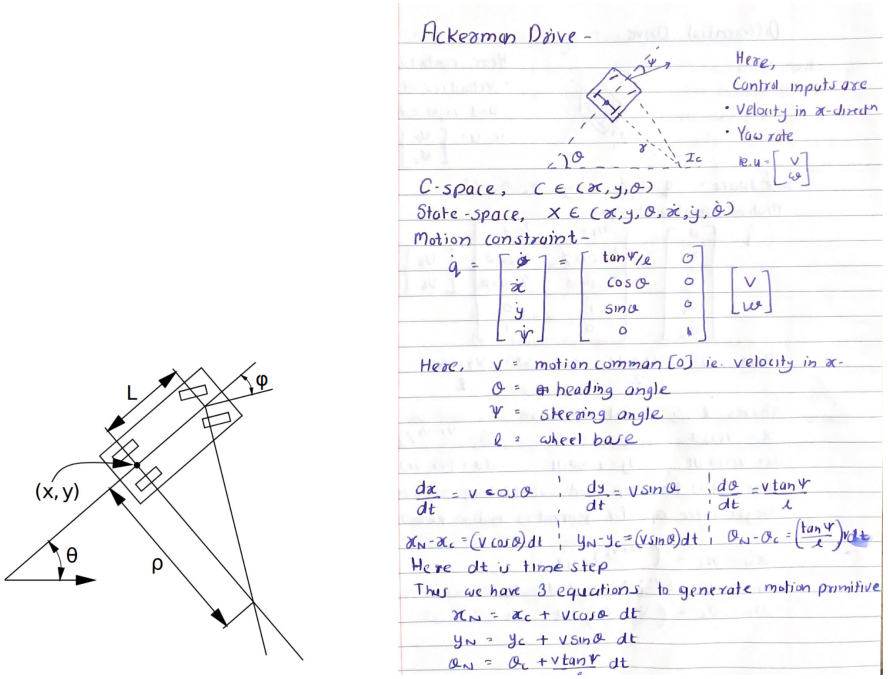


Figure 9: Modeling of Car: Ackermann steering

width = 2.0

Calculations using these equations are shown below:

- Modeling of Truck with trailer:

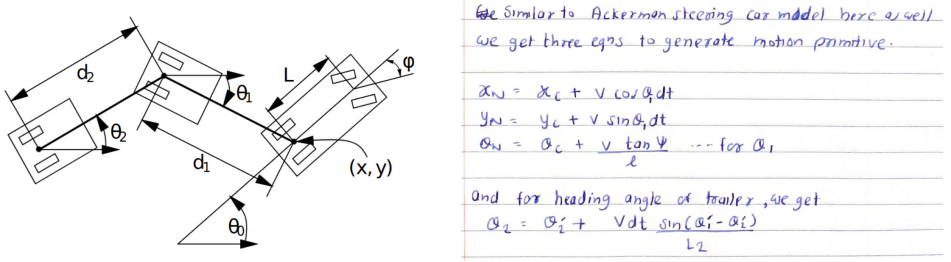


Figure 10: Modeling of Truck with trailer

Ackermann steering plus trailer kinematics Equations for modeling differential drive

robot are shown in Fig.

Car class was used to define the dimension of the car in algorithm for given car model simulated in Gazebo environment.

class CarWithTrailer:

maxSteerAngle = 0.6

steerPresion = 10

wheelBase = 2.58 [m] wheel base: rear to front steer

axleToFront = 3.0 [m] distance from rear to vehicle front end of vehicle

axleToHitch = 0.4

hitchToTrailer = 2.0

axleToBack = 0.4 [m] distance from rear to vehicle back end of vehicle

width = 2.0 [m] width of vehicle

RTF = 0.4 [m] distance from rear to vehicle front end of trailer

RTB = 4.0 [m] distance from rear to vehicle back end of trailer

Calculations using these equations are shown below:

5.4 Hybrid A-star algorithm

The Hybrid A* algorithm is a motion planning algorithm used in robotics to find collision-free paths for vehicles or robots with non-holonomic constraints, such as cars. It combines elements of both continuous and discrete state spaces, providing a compromise between computational efficiency and precision in path planning.

The algorithm is as follows:

Following is the code for hybrid A star algorithm used for this project.

- Model and cost initialization for each vehicle.
- Initialization of hybrid A star algorithm
- While loop for Hybrid A star
- Calculating holonomic cost considering obstacles
- Holonomic node validity check (map bounds and collision)
- Motion commands for Differential robot
- Simulating kinematic motion (Motion primitive)
- Collision check of simulated node
- Simulated path cost
- Backtrack the lowest cost path

```
Procedure EXPAND(q)
    foreach motion primitive neighbor n of q do
    g_new ← g(q) + getCost(q, n)
    if g_new < g(n) then
        g(n) ← g_new
        parent(n) ← q
    end
    f(n) ← g(n) + getHeuristic(n, q_goal) ;
Procedure A*SEARCH(q_start, q_goal)
    parent(q_start) ← 0
    g(.) ← ∞
    f(.) ← ∞
    OPEN ← q_start
    CLOSED ← 0
    while OPEN not empty do
    q ← OPEN.pop()
    CLOSED.insert(q)
    if q == q_goal then
        break
    end
    EXPAND(q) ;
```

Figure 11: Hybrid A-star algorithm

```python
class  DiffBot:
    speedPrecision = 4
    wheelSeperation=0.287 * 5           #lets take it as grid dimension
    botDiameter=0.32 * 5
    wheelDiameter=0.066 *5
    maxVelocity=2.0
    minVelocity=1.0
    step_size=0.5


class Cost:
    reverse = 30
    directionChange = 300
    steerAngle = 10
    hybridCost = 200
    lowSpeedCost=0
    steerAngleChange = 10
```

Figure 12: Model and cost initialization for Differential robot

```python
class Car:
    maxSteerAngle = 0.6
    steerPresion = 5
    wheelBase = 2.58
    axleToFront = 3.0
    axleToBack = 0.4
    width = 2.0

class Cost:
    reverse = 10
    directionChange = 150   #forward to reverse or revesre to forward
    steerAngle = 1
    steerAngleChange = 5
    hybridCost = 200
```

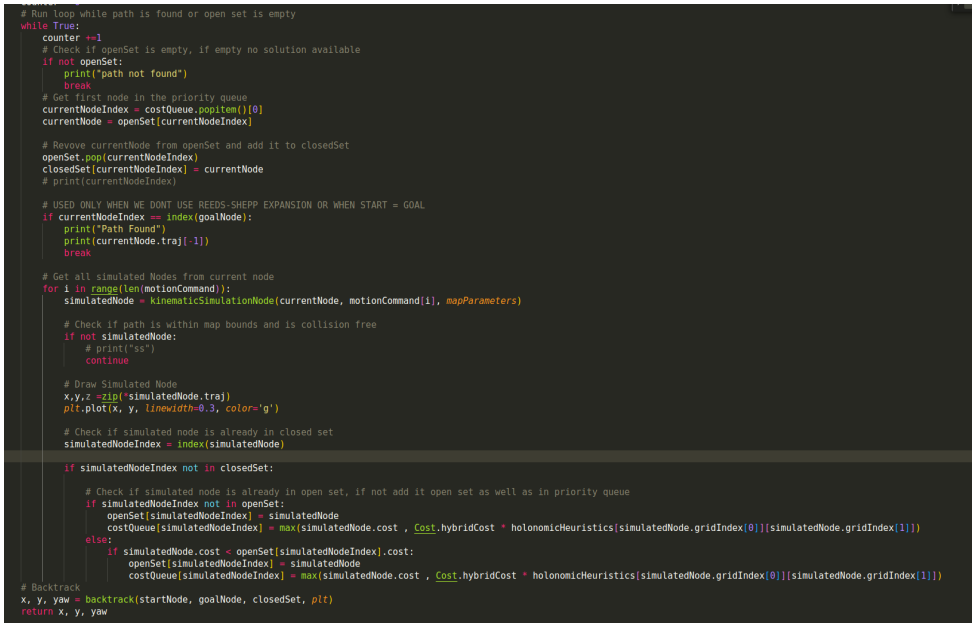
Figure 13: Model and cost initialization for car

```python
class CarWithTrailer:
    maxSteerAngle = 0.6
    steerPresion = 10
    wheelBase = 2.58    # [m] wheel base: rear to front steer
    axleToFront = 3.0   # [m] distance from rear to vehicle front end of vehicle
    axleToHitch = 0.4
    hitchToTrailer = 2.0
    axleToBack = 0.4    # [m] distance from rear to vehicle back end of vehicle
    width = 2.0   # [m] width of vehicle
    RTF = 0.4  # [m] distance from rear to vehicle front end of trailer
    RTB = 4.0  # [m] distance from rear to vehicle back end of trailer

    wheelSeperation = 0.7 * width  # [m] distance between left-right wheels

    RTR = 8.0  # [m] rear to trailer wheel

class Cost:
    reverse = 8
    directionChange = 100
    steerAngle = 1
    steerAngleChange = 5
    hybridCost = 50

class Node:
    def __init__(self, gridIndex, traj, steeringAngle, direction, cost, parentIndex):
        self.gridIndex = gridIndex         # grid block x, y, yaw index
        self.traj = traj                   # trajectory x, y  of a simulated node
        self.steeringAngle = steeringAngle # steering angle throughout the trajectory
        self.direction = direction         # direction throughout the trajectory
        self.cost = cost                   # node cost
        self.parentIndex = parentIndex     # parent node index
```

Figure 14: Model and cost initialization for truck with trailer

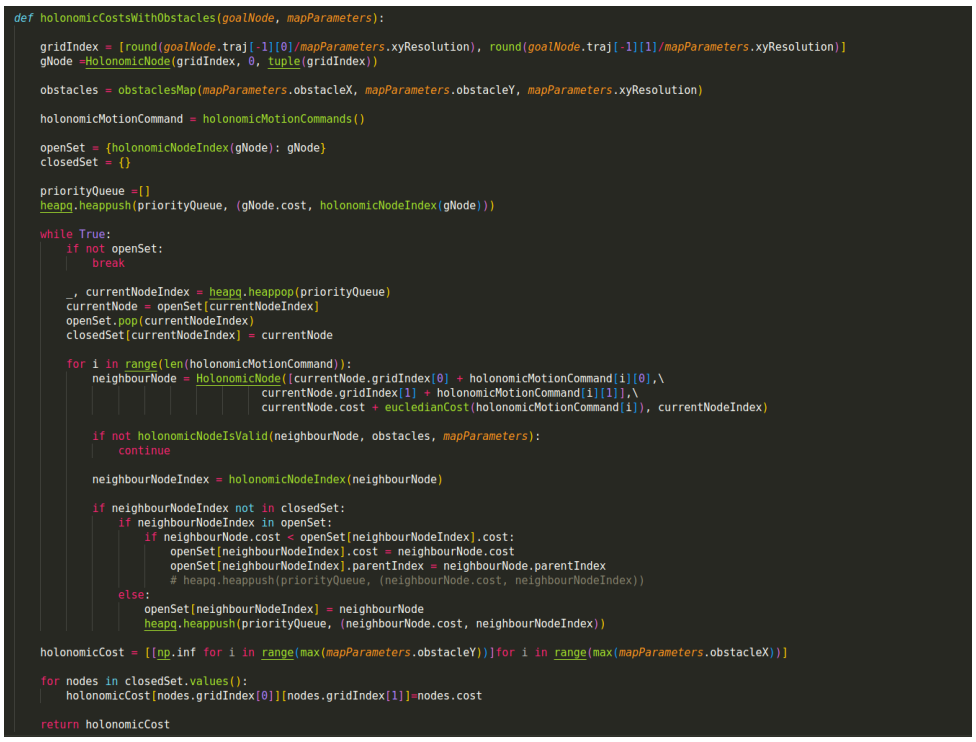
```python
def run(s, g, mapParameters, plt):

    # Compute Grid Index for start and Goal node
    sGridIndex = [round(s[0] / mapParameters.xyResolution), \
                  round(s[1] / mapParameters.xyResolution), \
                  round(s[2]/mapParameters.yawResolution)]
    gGridIndex = [round(g[0] / mapParameters.xyResolution), \
                  round(g[1] / mapParameters.xyResolution), \
                  round(g[2]/mapParameters.yawResolution)]

    # Generate all Possible motion commands to Differential drive robot
    motionCommand = motionCommands()
    # Create start and end Node
    startNode = Node(sGridIndex, [s], 0, 1, 0 , tuple(sGridIndex))
    goalNode = Node(gGridIndex, [g], 0, 1, 0, tuple(gGridIndex))
    # Find Holonomic Heuristric
    holonomicHeuristics = holonomicCostsWithObstacles(goalNode, mapParameters)
    # Add start node to open Set
    openSet = {index(startNode):startNode}
    closedSet = {}
    # Create a priority queue for acquiring nodes based on their cost's
    costQueue = heapdict()

    # Add start mode into priority queue
    costQueue[index(startNode)] = max(startNode.cost , Cost.hybridCost * holonomicHeuristics[startNode.gridIndex[0]][startNode.gridIndex[1]])
    counter = 0
```

Figure 15: Initialization of hybrid A star algorithm



```
# Run loop while path is found or open set is empty
while True:
    counter +=1
    # Check if openSet is empty, if empty no solution available
    if not openSet:
        print("path not found")
        break
    # Get first node in the priority queue
    currentNodeIndex = costQueue.popitem()[0]
    currentNode = openSet[currentNodeIndex]

    # Revove currentNode from openSet and add it to closedSet
    openSet.pop(currentNodeIndex)
    closedSet[currentNodeIndex] = currentNode
    # print(currentNodeIndex)

    # USED ONLY WHEN WE DONT USE REEDS-SHEPP EXPANSION OR WHEN START = GOAL
    if currentNodeIndex == index(goalNode):
        print("Path Found")
        print(currentNode.traj[-1])
        break

    # Get all simulated Nodes from current node
    for i in range(len(motionCommand)):
        simulatedNode = kinematicSimulationNode(currentNode, motionCommand[i], mapParameters)

        # Check if path is within map bounds and is collision free
        if not simulatedNode:
            # print("ss")
            continue

        # Draw Simulated Node
        x,y,z =zip(*simulatedNode.traj)
        plt.plot(x, y, linewidth=0.3, color='g')

        # Check if simulated node is already in closed set
        simulatedNodeIndex = index(simulatedNode)

        if simulatedNodeIndex not in closedSet:

            # Check if simulated node is already in open set, if not add it open set as well as in priority queue
            if simulatedNodeIndex not in openSet:
                openSet[simulatedNodeIndex] = simulatedNode
                costQueue[simulatedNodeIndex] = max(simulatedNode.cost , Cost.hybridCost * holonomicHeuristics[simulatedNode.gridIndex[0]][simulatedNode.gridIndex[1]])
            else:
                if simulatedNode.cost < openSet[simulatedNodeIndex].cost:
                    openSet[simulatedNodeIndex] = simulatedNode
                    costQueue[simulatedNodeIndex] = max(simulatedNode.cost , Cost.hybridCost * holonomicHeuristics[simulatedNode.gridIndex[0]][simulatedNode.gridIndex[1]])
# Backtrack
x, y, yaw = backtrack(startNode, goalNode, closedSet, plt)
return x, y, yaw
```

Figure 16: While loop for Hybrid A star



```
def holonomicCostsWithObstacles(goalNode, mapParameters):

    gridIndex = [round(goalNode.traj[-1][0]/mapParameters.xyResolution), round(goalNode.traj[-1][1]/mapParameters.xyResolution)]
    gNode =HolonomicNode(gridIndex, 0, tuple(gridIndex))

    obstacles = obstaclesMap(mapParameters.obstacleX, mapParameters.obstacleY, mapParameters.xyResolution)

    holonomicMotionCommand = holonomicMotionCommands()

    openSet = {holonomicNodeIndex(gNode): gNode}
    closedSet = {}

    priorityQueue =[]
    heapq.heappush(priorityQueue, (gNode.cost, holonomicNodeIndex(gNode)))

    while True:
        if not openSet:
            break

        _, currentNodeIndex = heapq.heappop(priorityQueue)
        currentNode = openSet[currentNodeIndex]
        openSet.pop(currentNodeIndex)
        closedSet[currentNodeIndex] = currentNode

        for i in range(len(holonomicMotionCommand)):
            neighbourNode = HolonomicNode([currentNode.gridIndex[0] + holonomicMotionCommand[i][0],\
                                           currentNode.gridIndex[1] + holonomicMotionCommand[i][1]],\
                                           currentNode.cost + eucledianCost(holonomicMotionCommand[i]), currentNodeIndex)

            if not holonomicNodeIsValid(neighbourNode, obstacles, mapParameters):
                continue

            neighbourNodeIndex = holonomicNodeIndex(neighbourNode)

            if neighbourNodeIndex not in closedSet:
                if neighbourNodeIndex in openSet:
                    if neighbourNode.cost < openSet[neighbourNodeIndex].cost:
                        openSet[neighbourNodeIndex].cost = neighbourNode.cost
                        openSet[neighbourNodeIndex].parentIndex = neighbourNode.parentIndex
                        # heapq.heappush(priorityQueue, (neighbourNode.cost, neighbourNodeIndex))
                else:
                    openSet[neighbourNodeIndex] = neighbourNode
                    heapq.heappush(priorityQueue, (neighbourNode.cost, neighbourNodeIndex))

    holonomicCost = [[np.inf for i in range(max(mapParameters.obstacleY))]for i in range(max(mapParameters.obstacleX))]

    for nodes in closedSet.values():
        holonomicCost[nodes.gridIndex[0]][nodes.gridIndex[1]]=nodes.cost

    return holonomicCost
```

Figure 17: Calculating holonomic cost considering obstacles

```python
def holonomicNodeIsValid(neighbourNode, obstacles, mapParameters):

    # Check if Node is out of map bounds
    if neighbourNode.gridIndex[0]<= mapParameters.mapMinX or \
       neighbourNode.gridIndex[0]>= mapParameters.mapMaxX or \
       neighbourNode.gridIndex[1]<= mapParameters.mapMinY or \
       neighbourNode.gridIndex[1]>= mapParameters.mapMaxY:
        return False

    # Check if Node on obstacle
    if obstacles[neighbourNode.gridIndex[0]][neighbourNode.gridIndex[1]]:
        return False

    return True
```

Figure 18: Holonomic node validity check (map bounds and collision)

```python
def motionCommands():


    # Motion commands for a Non-Holonomic Robot like a Differential drive robot (Trajectories using left wheel and right wheel velocity )

    angle=45
    step_length = DiffBot.maxVelocity / 2
    vr= step_length /(DiffBot.step_size* math.cos(math.radians(angle))) + ( DiffBot.wheelSeperation * math.tan(math.radians(angle)))/2
    vl= vr - (DiffBot.wheelSeperation * math.tan(math.radians(angle)))

    motionCommand = [[DiffBot.maxVelocity, DiffBot.maxVelocity],[vr,vl],[-vr,-vl]]

    return motionCommand
```

Figure 19: Motion commands for Differential robot

```python
def kinematicSimulationNode(currentNode, motionCommand, mapParameters, simulationLength=1, step = 0.2 ):

    # Simulate node using given current Node and Motion Commands
    traj = []
    angle = pi_2_pi(currentNode.traj[-1][2] + (motionCommand[0]-motionCommand[1]) * step /  DiffBot.wheelSeperation)

    traj.append([currentNode.traj[-1][0] + ((motionCommand[0] + motionCommand[1]) /2) * step * math.cos(angle),
                 currentNode.traj[-1][1] + ((motionCommand[0] + motionCommand[1]) /2) * step * math.sin(angle),
                 pi_2_pi(angle + (motionCommand[0]-motionCommand[1]) * step /  DiffBot.wheelSeperation)])
    for i in range(int((simulationLength/step))-1):
        traj.append([traj[i][0] + ((motionCommand[0] + motionCommand[1]) /2) * step * math.cos(traj[i][2]),
                     traj[i][1] + ((motionCommand[0] + motionCommand[1]) /2) * step * math.sin(traj[i][2]),
                     # pi_2_pi((motionCommand[0]-motionCommand[1]) * step /  DiffBot.wheelSeperation)])
                     pi_2_pi(traj[i][2] + (motionCommand[0]-motionCommand[1]) * step /  DiffBot.wheelSeperation)])

    # Find grid index
    # print(traj)
    gridIndex = [round(traj[-1][0]/mapParameters.xyResolution), \
                 round(traj[-1][1]/mapParameters.xyResolution), \
                 round(traj[-1][2]/mapParameters.yawResolution)]

    # Check if node is valid
    if not isValid(traj, gridIndex, mapParameters):

        return None

    # Calculate Cost of the node
    cost = simulatedPathCost(currentNode, motionCommand, simulationLength)

    return Node(gridIndex, traj, motionCommand[0], motionCommand[1], cost, index(currentNode))
```

Figure 20: Simulating kinematic motion (Motion primitive)

```python
def isValid(traj, gridIndex, mapParameters):

    # Check if Node is out of map bounds
    if gridIndex[0]<=mapParameters.mapMinX or gridIndex[0]>=mapParameters.mapMaxX or \
       gridIndex[1]<=mapParameters.mapMinY or gridIndex[1]>=mapParameters.mapMaxY:
        return False

    # Check if Node is colliding with an obstacle
    if collision(traj, mapParameters):
        # print("not valid")
        return False
    return True

def collision(traj, mapParameters):

    diffBotRadius = ( DiffBot.botDiameter)/4 + 1
    for i in traj:
        cx = i[0]
        cy = i[1]
        pointsInObstacle = mapParameters.ObstacleKDTree.query_ball_point([cx, cy], diffBotRadius)

        if not pointsInObstacle:
            continue

        for p in pointsInObstacle:
            xo = mapParameters.obstacleX[p] - cx
            yo = mapParameters.obstacleY[p] - cy
            dx = xo * math.cos(i[2]) + yo * math.sin(i[2])
            dy = -xo * math.sin(i[2]) + yo * math.cos(i[2])

            if abs(dx) < diffBotRadius and abs(dy) <  diffBotRadius:
                return True

    return False
```

Figure 21: Collision check of simulated node

```python
def simulatedPathCost(currentNode, motionCommand, simulationLength):

    # Previos Node Cost
    cost = currentNode.cost

    # Distance cost
    if motionCommand[0] > 0:
        cost += simulationLength
    elif motionCommand[0] < 0 :
        cost += simulationLength * Cost.reverse

    # Steering Angle change cost
    if currentNode != motionCommand[0] :
        cost += Cost.directionChange

    # Steering Angle Cost
    if motionCommand[0] != DiffBot.maxVelocity:
        cost += motionCommand[0] * Cost.steerAngle

    # Steering Angle change cost
    cost += abs(motionCommand[0] - currentNode.steeringAngle) * Cost.steerAngleChange
```

Figure 22: Simulated path cost

```python
def backtrack(startNode, goalNode, closedSet, plt):

    # Goal Node data
    # print(closedSet[-1])
    startNodeIndex= index(startNode)
    currentNodeIndex = list(closedSet)[-1]#goalNode.parentIndex
    currentNode = closedSet[currentNodeIndex]
    x=[]
    y=[]
    yaw=[]

    # Iterate till we reach start node from goal node
    while currentNodeIndex != startNodeIndex:
        a, b, c = zip(*currentNode.traj)
        x += a[::-1]
        y += b[::-1]
        yaw += c[::-1]
        currentNodeIndex = currentNode.parentIndex
        currentNode = closedSet[currentNodeIndex]
    return x[::-1], y[::-1], yaw[::-1]
```

Figure 23: Backtrack the lowest cost path