Assignment 2: Sampling Based Motion Planning: RRT and RRT*

Agam Modasiya, Section 1, NetID - ajm432, RUID - 185009911, CS - 460

November 7th, 2021

The Python libraries used in this assignment were:

1. numpy - To use it with matplotlib and do matrix calculations.
2. random - To obtain random points for the sample function.
3. matplotlib - To visualize the configuration space and the tree from both algorithms. It was also used to animate the robot over the found path.
4. math - To do trig math for rotating the robot.

1 2D Geometric Motion Planning

1.1 Parser for the input file and problem visualization

For the first part of the assignment, we were tasked with writing two functions. The first one was to parse two text files that hold the data for the robot, obstacles and start and goal locations for the problem. The second function was to visualize the robot, obstacles and the start and goal locations of the given problem. Both the functions are given below with explanation for each.

```
def parse_problem(world_file, problem_file):

    world_data = open(world_file, "r")
    problem_locations = open(problem_file, "r")
    robot = []
    obstacles = []
    problems = []

    split_data = world_data.readline().split(" ") # split the first line by one space

    for j in range(0, len(split_data)):
        if (j + 1) % 2 == 0:
            robot.append((float(split_data[j - 1]), float(split_data[j]))) # populate robot points

    for i in world_data: # populate the obstacles list, starting from line 2
        split_data = i.split(" ")
        obstacle_single = []
        for j in range(0, len(split_data)):
            if (j + 1) % 2 == 0:
                obstacle_single.append((float(split_data[j - 1]), float(split_data[j])))

        obstacles.append(obstacle_single)

    for i in problem_locations: # populate the start and goal in pairs to a list
        split_data = i.split(" ")
        start_goal = []
        for j in range(0, len(split_data)):
            if (j + 1) % 2 == 0:
                start_goal.append((float(split_data[j - 1]), float(split_data[j])))

        problems.append(start_goal)

    rtrn_val = (robot, obstacles, problems)
    return rtrn_val # return the robot obstacles and problems list
```

The function above is for parsing the two given files and extracting the robot points, obstacles points and the start and goal locations. The robot points are put in the robot list. There are multiple obstacles so there is a nested list that holds the points for each given obstacle. Lastly, there are multiple start and goal locations in a pair, so they are also put in a nested list. The function returns a tuple, where the first index holds the list of robot points, the second index holds a list that holds the list of individual obstacles points, and the third index holds a list that has lists that holds start and goal location for problems.

```
def visualize_problem(robot, obstacles, start, goal):

    plt.xlim(0, 10)
    plt.ylim(0, 10)
    robot_start = translate(robot, start) # translates the robot to the given point
    robot_strt_pts = np.array(robot_start)
    robot_p_st = Polygon(robot_strt_pts, color=[0, 1, 0]) # add the robot patch at start point,
        with green color
    robot_goal = translate(robot, goal)
    robot_goal_pts = np.array(robot_goal)
    robot_p_gl = Polygon(robot_goal_pts, color=[1, 0, 0]) # add the robot patch at end point, with
        red color
    display = plt.gca()
    display.add_patch(robot_p_st)
    display.add_patch(robot_p_gl)

    for i in obstacles: # add patches for the obstacles
        obs = Polygon(i, color=[1, 0, 1])
        display.add_patch(obs)

    plt.show()
```

The function above uses the matplotlib library to create a visual of the given problem. It first translates the robot to the start location using the translate function, given below. The input to the translate function are the robot points and the point we want the robot to be translated to. Since the robot is going to have a point on (0, 0) we can easily add the point's x and y values to the x and y values of the robot's points to get the robot's translated location. Once we get the robot's translated points, they can be converted to numpy arrays and using the Polygon function of matplotlib it can be added to the graph a patch. The same procedure is used for the goal node and the obstacles. The robot at start location is colored green, at the goal location it is colored red, and the obstacles are colored purple.

```
def translate(robot, goal):

    location = []

    for i in robot:
        location.append((i[0] + goal[0], i[1] + goal[1])) # since the robot starts at (0, 0), the
            goal points can be
        # added directs to robot points individually

    return location
```

1.2 Sampler to sample a random point in the configuration space, and visualize the robot at that point in the world

For this part of the assignment, we were tasked with writing a sampler function and a function for visualize robot at given points.

```
def sample():

    rtrn_val = (random.randrange(0, 100)/10, random.randrange(0, 100)/10) # get a random number
        between 0 and 100 then divide by 10

    return rtrn_val
```

The sample funciton given above takes no input, but it returns a random coordinate where x and y can be between 0 and 10, inclusive, in .1 steps. The random number is generated using the random library.

```
def visualize_points(points, robot, obstacles, start, goal):

    plt.xlim(0, 10)
    plt.ylim(0, 10)
    robot_start = translate(robot, start) # robot points at the start position
    robot_strt_pts = np.array(robot_start)
    robot_p_st = Polygon(robot_strt_pts, color=[0, 1, 0])
    robot_goal = translate(robot, goal) # robot points at the goal position
    robot_goal_pts = np.array(robot_goal)
    robot_p_gl = Polygon(robot_goal_pts, color=[1, 0, 0])
    display = plt.gca()

    if points: # for all the given points, move the robot there and add a patch
        for j in points:
            temp_pnt = Polygon(np.array(translate(robot, j)), color=[0, 0, 0]) # translate the robot
                to given point
            display.add_patch(temp_pnt)

    display.add_patch(robot_p_st)
    display.add_patch(robot_p_gl)

    for i in obstacles: # add patches for the obstacles
        obs = Polygon(i, color=[1, 0, 1])
        display.add_patch(obs)

    plt.show()
```

The input for visualize_points function is a list of points, list of robot points, obstacles, and start and goal point. Similar to the visualize_problem function, matplotlib Polygon function is used to plot the robot and obstacles. For each of the given point in the points list, the robot is translated there and it is graphed on the final plot.

1.3 A collision checker for the robot and obstacles

For this part of the assignment, a collision detection function was implemented. It was to check if the robot was colliding with any given obstacles or outsie the bounds of the given world space.

```
def isCollisionFree(robot, point, obstacles):

    robot_points = translate(robot, point)

    robot_lines_list = [] # holds the list of lines that can be made from the edges of the robot

    for i in range(-1, len(robot_points) - 1):

        robot_lines_list.append([robot_points[i], robot_points[i+1]])

    for obs in obstacles:

        temp_obstacles = []

        for i in range(-1, len(obs) - 1):

            temp_obstacles.append([obs[i], obs[i + 1]])

        for robot_line in robot_lines_list:

            for obs_line in temp_obstacles:
                intersection = isIntersecting(robot_line, obs_line)
                rob_x = sorted([robot_line[0][0], robot_line[1][0]])
                obs_x = sorted([obs_line[0][0], obs_line[1][0]])
                rob_y = sorted([robot_line[0][1], robot_line[1][1]])
                obs_y = sorted([obs_line[0][1], obs_line[1][1]])

                if rob_x[0] <= intersection[0] <= rob_x[1] and obs_x[0] <= intersection[0] <= obs_x
                    [1]: # check
                    # if the intersection is in range of the robot path line
                    if rob_y[0] <= intersection[0] <= rob_y[1] and obs_y[0] <= intersection[0] <=
                        obs_y[1]:
                        # print(intersection)
                        return False

    for i in obstacles: # check if the robot is inside the polygon
        current_obs_path = plpath.Path(i)
        for j in robot_points:
            if current_obs_path.contains_point(j):
                return False
            elif j[0] > 10 or j[1] > 10 or j[0] < 0 or j[1] < 0:
                return False

    return True
```

The inputs to the collision checker function are the points of the robot, the point at which we have to check if the robot is collision free and the obstacles. The algorithm creates a list that contains another lists containing the pairs of points that make the lines of the robot. The same thing is done for the obstacles, one by one in the for loop. The pairs of points can be used to create a line. Every line from robot_lines_list is used with the lines of the obstacles one by one to find out if they intersect, using the isIntersecting function. The isintersection function inputs are two lists that hold the pair of points that makes the lines we need to find the intersection between. If they don't intersect, the line is collision free, but if they do intersect, we have to make sure the intersection point in on the obstacle and the robot. That is done by checking if the

x and y values of the intersection point are in the range of the x and y values of both the pairs of points used. For example, if the robot was defined by [(0, 0), (0, 1), (1, 1), (1, 0)] and an obstacle is defined by [(0, 8), (1, 9), (2, 8)] and the two lines being compared are defined by point pair [(1, 1), (1, 0)] (for robot) and [(2, 8), (0, 8)] (for obstacle) the intersection point would be (1, 8). Since the y value of the intersection point is not between the y values of the line of the robot, it is not considered a collision. The algorithms also uses a function from matplotlib for an edge case if the robot is fully inside the obstacle. The matplotlib function used is contains_point(point), that checks if a given point is inside a given polygon. If there are no collisions found the function returns True. As soon as a collision is found by the algorithm it returns False.

```
def isIntersecting(line1, line2): # check if the lines intersect, if they do return the point of
    intersection

    dx = (line1[0][0] - line1[1][0], line2[0][0] - line2[1][0])
    dy = (line1[0][1] - line1[1][1], line2[0][1] - line2[1][1])

    div = determinant(dx, dy)
    if div == 0:
        return tuple((-1, -1))

    d = (determinant(*line1), determinant(*line2))
    x = round(determinant(d, dx) / div, 4)
    y = round(determinant(d, dy) / div, 4)
    rtrn_tuple = (x, y)

    return rtrn_tuple
```

1.4 A data structure to represent the search tree of the planning process

For the assignment, a Tree class was also made to keep track of the data. The input for initializing it were the robot points, list of obstacles, and start and goal points.

```
    def __init__(self, robot, obstacles, start, goal):

        self.robot = robot
        self.obstacles = obstacles
        self.start = start
        self.goal = goal
        self.node_data = {start: [start, None]} # name of the node is the key and the list is it's
            parent,
        # and distance from parent
```

The self.node_data dictionary hold the data for each node. The key is the point itself and it holds a list whose first index is it's parent and the second index is the distance to the parent. The first input to the function is the parent and the second input is it's child node.

```
    def add(self, point1, point2):

        dx = point1[0] - point2[0]
        dy = point1[1] - point2[1]
        distance = math.sqrt(pow(dx, 2) + pow(dy, 2)) # euclidean distance between point1 and point2
        self.node_data[point2] = [point1, distance] # add point2 and have point1 as it's parent
```

The exists function checks if a given points exists in the tree, which can be simply be checked by checking is the point is in the self.node_data dictionary. The input to the function is the point that needs to be checked if it is inside the tree.

```
    def exists(self, point):

        return point in self.node_data.keys() # if the point is in the dictionary as a tree, return
            True
```

The parent function returns the parent of the input point.

```
    def parent(self, point):

        return self.node_data[point][0] # the first index of the list holds the parent of the node
```

The nearest function searches through all the nodes of the tree and return the one that has the smallest euclidean distance to the input point.

```
    def nearest(self, point):

        distance = math.inf # keep track of the shortest distance
        rtrn_tuple = None # point that has the lowest distance from the point
        for key in self.node_data:
            dx = key[0] - point[0]
            dy = key[1] - point[1]
            temp_distance = math.sqrt(pow(dx, 2) + pow(dy, 2)) # find the distance from the point to
                 key

            if temp_distance < distance and key != point: # if the distance from current point is
                 lower than the
                # lowest found before and the key is not the point
                distance = temp_distance # replace the distance with lowest
                rtrn_tuple = key # replace the key with the shortest distance

        return rtrn_tuple
```

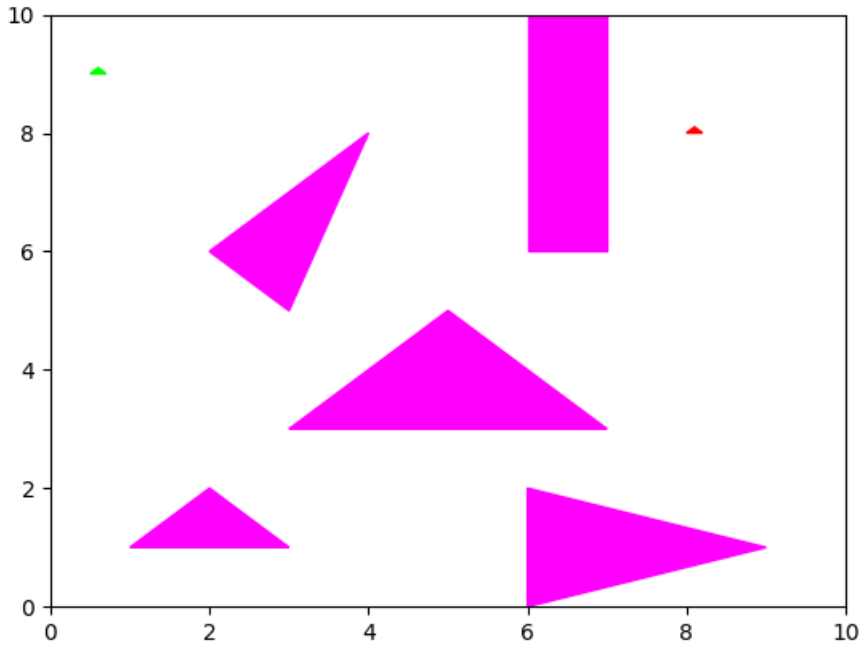
The extend function takes in two points, point1 and point2. To extend from point1 to point2, we check if the path from point1 to point2 is collision free. This collision checking is complete, meaning we can get the exact intersection points. The collision detection function is the same one as explained earlier except is takes point1 and point2 as inputs. For each point of the robot at point1, is paired with the same point of the robot at point2. For example, if we have a square robot, we pair the top right corner point of the robot when the robot is at point1 with the top right corner point when the robot is at point2. These points can be used to make line segment and it is checked if the line segments intersect with any of the obstacles.

```
    def extend(self, point1, point2):

        if self.isCollisionFree(point1, point2): # if the path from poit1 to poit2 is collision free
             , extend
            self.add(point1, point2) # point2 is added with point1 being the parent node
            return True

        return False
```

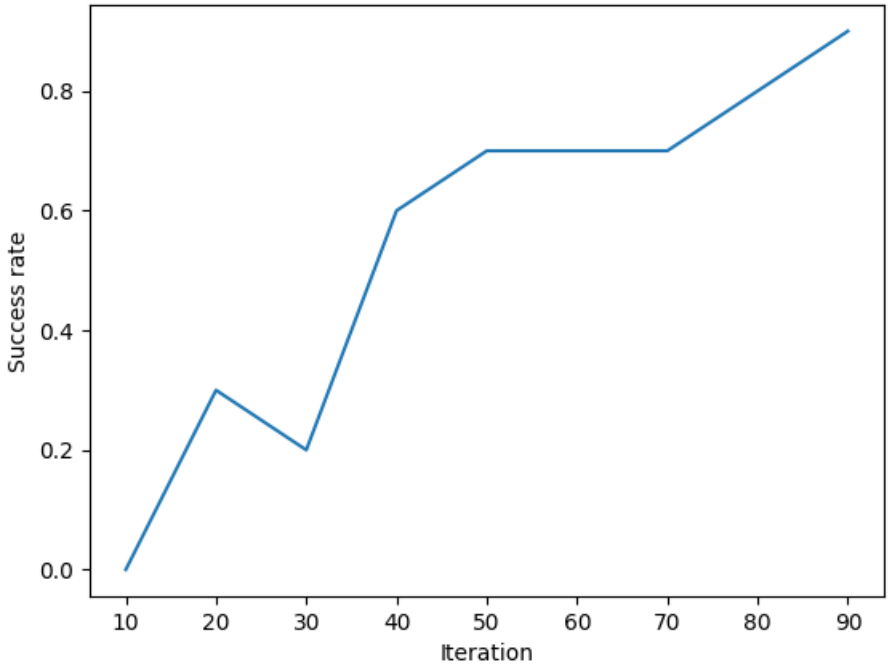
1.5 Implement the RRT algorithm

For this part of the assignment, we were tasked with implementing the RRT algorithm. The function for the RRT algorithm is given below. The inputs to the function are the vertices of the robot, vertices of the obstacles, start and goal point, and n iterations for sampling. First the Tree structure is initialized with the given values. Next, a for loop is run for n is iterations. Inside the for loop, the sample function is run to get a random point. Next, the nearest point to the sample point is found and the the sample point is added to the tree using the extend function. After the for loop, it is checked if the goal node is in the tree. If it's not inside the tree then the nearest point that is collision free is found using the nearest_collision_free function. This function takes the goal node as the input. If the nearest_collision_free function returns None, there is no path that connects the goal node to the start node and None is returned. If a point is found, we extend from that point to the goal node. Next, the goal node is added to the path list and a while loop is run. In the while loop we get the parent of the last node added to the path list and add it to the end of the path list. This is done until the last node in the path list is the start goal. Lastly, the path list is reversed and it is returned.

```
def rrt(robot, obstacles, start, goal, itern_n):

    tree_struct = Tree(robot, obstacles, start, goal) # initialize the tree structure with given
        data

    for i in range(itern_n): # run the loop for given n iterations
        sample_point = sample()
        if not tree_struct.exists(sample_point): # if the sample point is already in the tree don't
            run
            if len(tree_struct.node_data) == 1: # if it's the first point add it to the start node
                tree_struct.extend(start, sample_point)
            else:
                nearest_point = tree_struct.nearest(sample_point) # find the nearest point to the
                    sample point
                tree_struct.extend(nearest_point, sample_point) # extend from the nearest point to
                    the sampel point

    if not tree_struct.exists(goal): # if the goal not is not in the tree
        nearest_to_goal = tree_struct.nearest_collision_free(goal) # find a node that is nearest to
            the goal and is
        # collision free
        if nearest_to_goal is None: # if there is no point that is collision free close to goal,
            return None
            return None
        path_to_goal = tree_struct.extend(nearest_to_goal, goal) # if there is a node, extend from
            node to goal
    else:
        path_to_goal = True

    path = [goal] # add goal to the path list
    if path_to_goal:
        curr_node = goal
        while curr_node != start: # while we don't reach the start node

            path.append(tree_struct.parent(curr_node)) # keep adding the parent of the current node
            curr_node = path[-1] # update the current node to be the last node in the path list

        path.reverse() # reverse the path
        return path # return the path list

    return None
```



The problem visualization above is used for comparing the iteration vs. success rate for RRT. The data was collected by running the algorithm 10 times for iterations from 10 to 100 in steps of 10. The collected data is graphed below:



It can be observed that as the number of iteration increase, the success rate also increases. This makes sense because as we sample more, we learn more about the space and the probability of finding a path increases.

1.6 Transform the world into configuration space for visualization

The visualize_path function uses similar code to the visualize_problem function to graph the obstacles. In addition, it takes the list of point that make the path from start to goal position. The robot is animated over the path using the matplotlib animation function.

```
def visualize_path(robot, obstacles, path):

    if path is None:
        return None

    robot_start = translate(robot, path[0]) # robot points at the start position
    robot_strt_pts = np.array(robot_start)
    robot_p_st = Polygon(robot_strt_pts, color=[0, 1, 0])
    robot_goal = translate(robot, path[-1])
    robot_goal_pts = np.array(robot_goal) # robot points at the goal position
    robot_p_gl = Polygon(robot_goal_pts, color=[1, 0, 0])
    disp = plt.figure()
    display = disp.gca()
    plt.xlim(0, 10)
    plt.ylim(0, 10)
    robot_path_patch = display.add_patch(robot_p_st)
    for i in obstacles:
        obs = Polygon(i, color=[1, 0, 1])
        display.add_patch(obs)
    step = len(path)

    discritized_path = []
    for z in range(step - 1):
        dx = path[z + 1][0] - path[z][0]
        dy = path[z + 1][1] - path[z][1]
        x_step = dx / 40
        y_step = dy / 40
        for j in range(41):
            node = (path[z][0] + (x_step * j), path[z][1] + (y_step * j)) # add the small steps
                between the 2 points
            discritized_path.append(node)

    step = len(discritized_path)

    def animate(step_num):
        step_size = discritized_path[step_num]
        robot_path_patch.set_xy(translate(robot, step_size)) # translate the robot to given point
        return robot_p_st

    ani = animation.FuncAnimation(disp, animate, frames=step, interval=10, repeat=False)

    x_vals = []
    y_vals = []

    for j in path: # add circles to the points for better visuals and animate the path line
        temp_pnt = plt.Circle(j, .05)
        display.add_patch(temp_pnt)
        x_vals.append(j[0])
        y_vals.append((j[1]))

    plt.plot(x_vals, y_vals)
    plt.show()
```

Next, the configuration space is calculated where the robot becomes a point. This is done using discrete Minkowski difference. The robot is flipped over the line y = -x and the flipped robot is plotted over all the lines of the obstacles. The lines are discretized and the flipped robot is translated to each of those points and graphed. This gives us a rough map of the configuration space.

```
def visualize_configuration(robot, obstacles, start, goal):

    plt.xlim(0, 10)
    plt.ylim(0, 10)
    robot_start = translate(robot, start) # translates the robot to the given point
    robot_strt_pts = np.array(robot_start)
    robot_p_st = Polygon(robot_strt_pts, color=[0, 1, 0]) # add the robot patch at start point,
        with green color
    robot_goal = translate(robot, goal)
    robot_goal_pts = np.array(robot_goal)
    robot_p_gl = Polygon(robot_goal_pts, color=[1, 0, 0]) # add the robot patch at end point, with
        red color
    display = plt.gca()
    display.add_patch(robot_p_st)
    display.add_patch(robot_p_gl)
    flipped_robot = []
    for point in robot: # flip the robot over the y = -x line

        flipped_robot.append((-point[0], -point[1]))

    boundary_list = [(0, 0), (0, 10), (10, 10), (10, 0)]
    obstacles.append(boundary_list)
    for obs in obstacles:

        for i in range(len(obs)): # for every line of the obstacles, travrese the flipped robot over
             the line

            start_point = obs[i]
            end_point = obs[i - 1]
            dx = start_point[0] - end_point[0]
            dy = start_point[1] - end_point[1]
            x_step = dx / 400
            y_step = dy / 400
            for j in range(401): # discritize the line and add a patch of the robot at every point
                node = (end_point[0] + (x_step * j), end_point[1] + (y_step * j))
                points_list = translate(flipped_robot, node)
                plgn = Polygon(np.array(points_list), color=[.5, .5, .5])
                display.add_patch(plgn)

    obstacles.remove(boundary_list)

    for i in obstacles: # add patches for the obstacles
        obs = Polygon(i, color=[1, 0, 1])
        display.add_patch(obs)

    plt.show()
```

Next, RRT algorithm is visualized. This function is a combination of the RRT and visualize_configuration function. First the RRT algorithms is run and then the configuration space is calculated. Next, the configuration space is plotted with the robots at the start and goal points. Lastly, the tree structure is animated as it was grew during the RRT algorithm.

```
def visualize_rrt(robot, obstacles, start, goal, itern_n):

    tree_struct = Tree(robot, obstacles, start, goal) # initialize the tree structure with given
        data

    for i in range(itern_n): # run the loop for given n iterations

        sample_point = sample()
        while not isCollisionFree(robot, sample_point, obstacles) or tree_struct.exists(sample_point
            ):
            # get asample that is collision free and a point that is not in the tree
            sample_point = sample()

        if len(tree_struct.node_data) == 1: # if it's the first point add it to the start node

            tree_struct.extend(start, sample_point)

        else:
            nearest_point = tree_struct.nearest(sample_point) # find the nearest point to the sample
                 point
            tree_struct.extend(nearest_point, sample_point) # extend from the nearest point to the
                sampel point

    if not tree_struct.exists(goal): # if the goal not is not in the tree
        nearest_to_goal = tree_struct.nearest_collision_free(goal) # find a node that is nearest to
            the goal and is
        # collision free
        if nearest_to_goal is None: # if there is no point that is collision free close to goal,
            return None
            visualize_points(tree_struct.node_data.keys(), robot, obstacles, start, goal)
            return None
        path_to_goal = tree_struct.extend(nearest_to_goal, goal) # if there is a node, extend from
            node to goal
    else:
        path_to_goal = True

    path = [goal] # add goal to the path list
    if path_to_goal:
        curr_node = goal
        while curr_node != start: # while we don't reach the start node
            path.append(tree_struct.parent(curr_node)) # keep adding the parent of the current node
            curr_node = path[-1] # update the current node to be the last node in the path list

        path.reverse() # reverse the path

    plt.xlim(0, 10)
    plt.ylim(0, 10)
    robot_start = translate(robot, start) # robot points at the start position
    robot_strt_pts = np.array(robot_start)
    robot_p_st = Polygon(robot_strt_pts, color=[0, 1, 0])
    robot_goal = translate(robot, goal) # robot points at the goal position
    robot_goal_pts = np.array(robot_goal)
    robot_p_gl = Polygon(robot_goal_pts, color=[1, 0, 0])
```

```
display = plt.gca()
display.add_patch(robot_p_st)
display.add_patch(robot_p_gl)
flipped_robot = []
for point in robot: # flip the robot over the y = -x line

    flipped_robot.append((-point[0], -point[1]))
boundary_list = [(0, 0), (0, 10), (10, 10), (10, 0)]
obstacles.append(boundary_list)
for obs in obstacles:

    for i in range(len(obs)): # for every line of the obstacles, travrese the flipped robot over
         the line

        start_point = obs[i]
        end_point = obs[i - 1]
        dx = start_point[0] - end_point[0]
        dy = start_point[1] - end_point[1]
        x_step = dx / 400
        y_step = dy / 400
        for j in range(401): # discritize the line and add a patch of the robot at every point
            node = (end_point[0] + (x_step * j), end_point[1] + (y_step * j))
            points_list = translate(flipped_robot, node)
            plgn = Polygon(np.array(points_list), color=[.5, .5, .5])
            display.add_patch(plgn)

obstacles.remove(boundary_list)

for i in obstacles: # add patches for the obstacles
    obs = Polygon(i, color=[1, 0, 1])
    display.add_patch(obs)

for j in tree_struct.node_data.keys():
    temp_pnt = plt.Circle(j, .05) # for all the given points, add a small circle patch
    display.add_patch(temp_pnt)
    x_vals = [j[0], tree_struct.parent(j)[0]] # add the tree stucture lines
    y_vals = [j[1], tree_struct.parent(j)[1]]
    plt.plot(x_vals, y_vals, color=[0, 0, 0])
    plt.draw()
    plt.pause(.01) # animate the tree growth

plt.show()
```

1.7 Cost and rewire

For the RRT* algorithm, a new function get_cost is added to the Tree class. This function takes a point inside the tree as an input and returns the cost from it to the start node.

```
def get_cost(self, point):

    rtrn_val = 0
    curr_node = point
    while curr_node != self.start: # while we don't reach the start goal

        rtrn_val += self.node_data[curr_node][1] # add the cost from current node to parent
        curr_node = self.node_data[curr_node][0] # replace the current node to it's parent

    return rtrn_val
```

Next function required for RRT* is the rewire function that takes a point and radius as an input. First, the neighbors in the radius r of the point are found and put in the neighbors_list. Next, for each point(node) in the neighbors_list, the cost to get there is compared to the cost to get there from other points(temp_node) in the list. If a shorter path to the node point is found than the current path, the parent of the node point is changed to the temp_node. This is run inside a while loop until no more changes are made to the tree.

```
def rewire(self, point, r):

    neighbors_list = [] # list of neighbors in the radius r

    for key in self.node_data:

        dx = key[0] - point[0]
        dy = key[1] - point[1]
        distance = math.sqrt(pow(dx, 2) + pow(dy, 2)) # find the distance

        if distance <= r: # if the distance is less than r, add it to the neighbors list
            neighbors_list.append(key)

    if neighbors_list is None or len(neighbors_list) <= 0: # if there are no neighbors, return
        None
        return

    replacement_node = self.start

    while replacement_node: # to check if a new change was made, if there was then the loop is
        run again

        for node in neighbors_list: # for each node in the neighbors list

            if node == self.start: # if node == start node, we don't run the loop
                continue
            current_cost = self.get_cost(node)
            shrtst_dst_frm_temp_node = math.inf
            replacement_node = None
            for temp_node in neighbors_list:
                if temp_node == node or temp_node == self.start: # if node is start or temp, we
                    don't run the loop
                    continue
                if self.isCollisionFree(node, temp_node): # only do the calculations if the
                    points have a path
                    temp_node_cost = self.get_cost(temp_node)
```

```
                dx = node[0] - temp_node[0]
                dy = node[1] - temp_node[1]
                temp_node_to_node = math.sqrt(pow(dx, 2) + pow(dy, 2))
                if current_cost > (temp_node_to_node + temp_node_cost): # shorter than the
                    current path

                    if (temp_node_to_node + temp_node_cost) < shrtst_dst_frm_temp_node: #
                        shorter than the
                        # shortest path found before

                        shrtst_dst_frm_temp_node = temp_node_to_node + temp_node_cost #
                            replace the shortest dist
                        replacement_node = temp_node # replace the node to new parent of node

            if replacement_node: # if there is a replacement node, replace it
                dx = node[0] - replacement_node[0]
                dy = node[1] - replacement_node[1]
                temp_dist = math.sqrt(pow(dx, 2) + pow(dy, 2)) # update the distance
                self.node_data[node][0] = replacement_node # update the parent node
                self.node_data[node][1] = temp_dist
                replacement_node = None
```

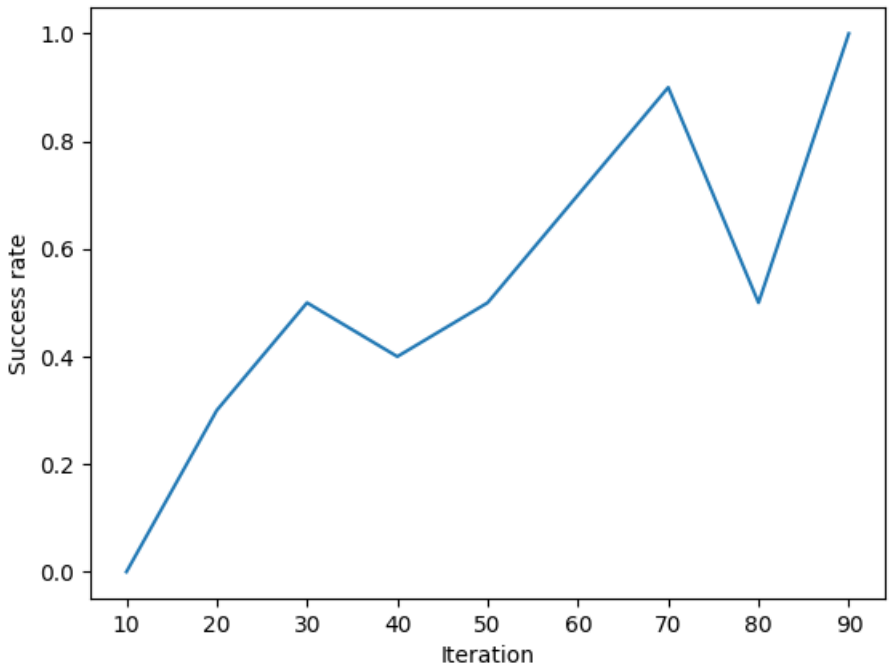
1.8 RRT*

The RRT* function is very similar to the RRT function. For RRT*, if the sample point was added to the tree, the rewire function is run. The rewire function re-configures the parents of the node in the radius r of the point sample point if there is a shorter path available. The radius for the rewire function is set to 1.5, as it was observed to give reliable rewiring.

```
def rrt_star(robot, obstacles, start, goal, itern_n):

    tree_struct = Tree(robot, obstacles, start, goal) # initialize the tree structure with given
        data

    for i in range(itern_n): # run the loop for given n iterations

        sample_point = sample()
        while not isCollisionFree(robot, sample_point, obstacles) or tree_struct.exists(sample_point
            ):
            # get asample that is collision free and a point that is not in the tree
            sample_point = sample()

        if len(tree_struct.node_data) == 1: # if it's the first point add it to the start node

            tree_struct.extend(start, sample_point)

        else:
            nearest_point = tree_struct.nearest(sample_point) # find the nearest point to the sample
                point
            if tree_struct.extend(nearest_point, sample_point): # if the sample point was added,
                rewire
                tree_struct.rewire(sample_point, 1.5)

    if not tree_struct.exists(goal): # if the goal not is not in the tree
        nearest_to_goal = tree_struct.nearest_collision_free(goal) # find a node that is nearest to
            the goal and is
```

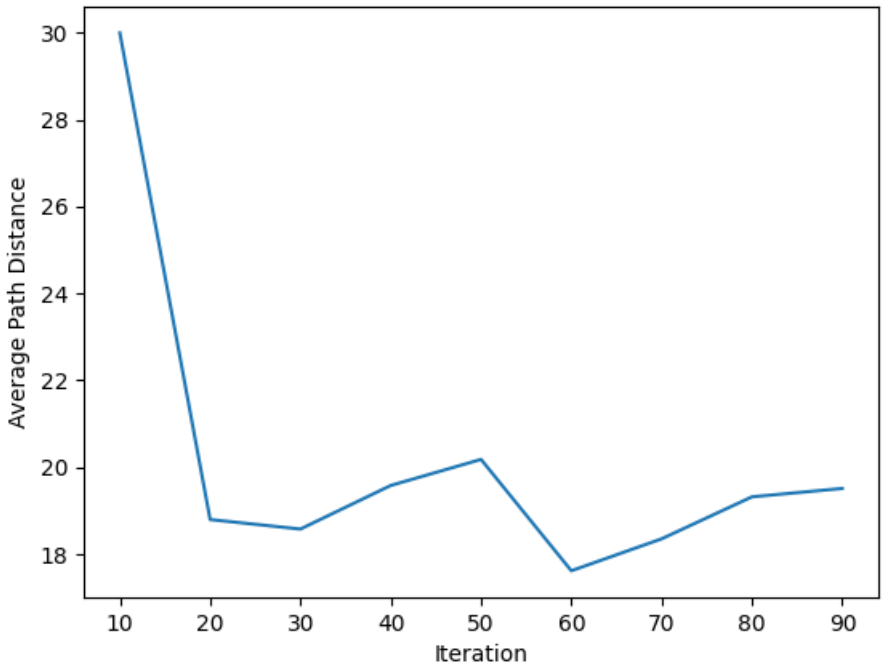
```
            # collision free
            if nearest_to_goal is None: # if there is no point that is collision free close to goal,
                return None
                return None
            path_to_goal = tree_struct.extend(nearest_to_goal, goal) # if there is a node, extend from
                node to goal
        else:
            path_to_goal = True

        path = [goal] # add goal to the path list
        if path_to_goal:
            curr_node = goal
            while curr_node != start: # while we don't reach the start node
                path.append(tree_struct.parent(curr_node)) # keep adding the parent of the current node
                curr_node = path[-1] # update the current node to be the last node in the path list

            path.reverse() # reverse the path
            return path

        return None
```

The iteration vs success rate test was done in the same way as it was done for RRT and the collected data is graphed below:

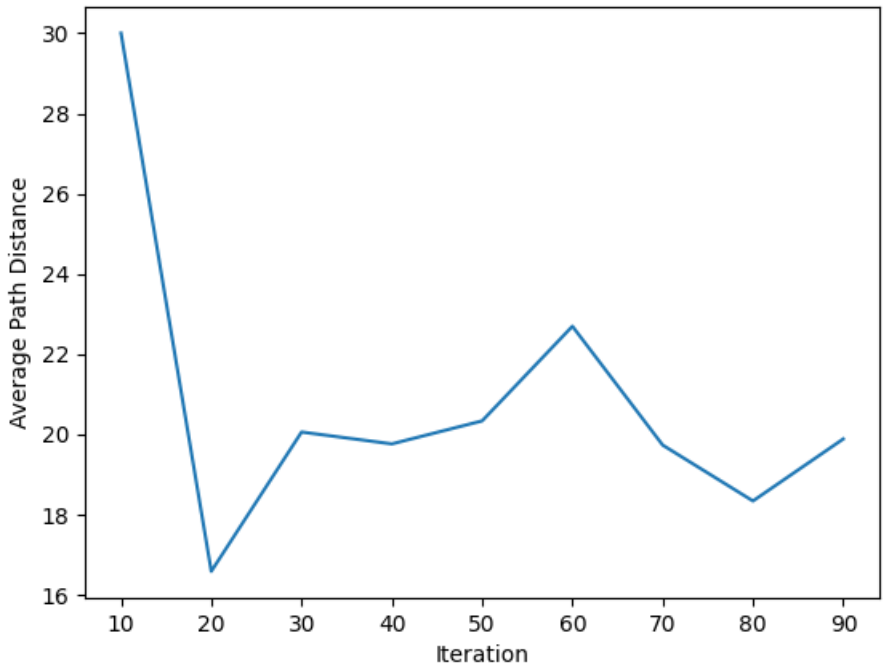


It can be noticed that as the number of iteration grows, the success rate also grows with it. The path length was compared between RRT and RRT*. Each algorithm was run 10 times from 10 to 100 iterations and the average path was calculated using the tests that resulted in a path from start to goal.

The graph below is for the average path distance vs iterations for RRT*.

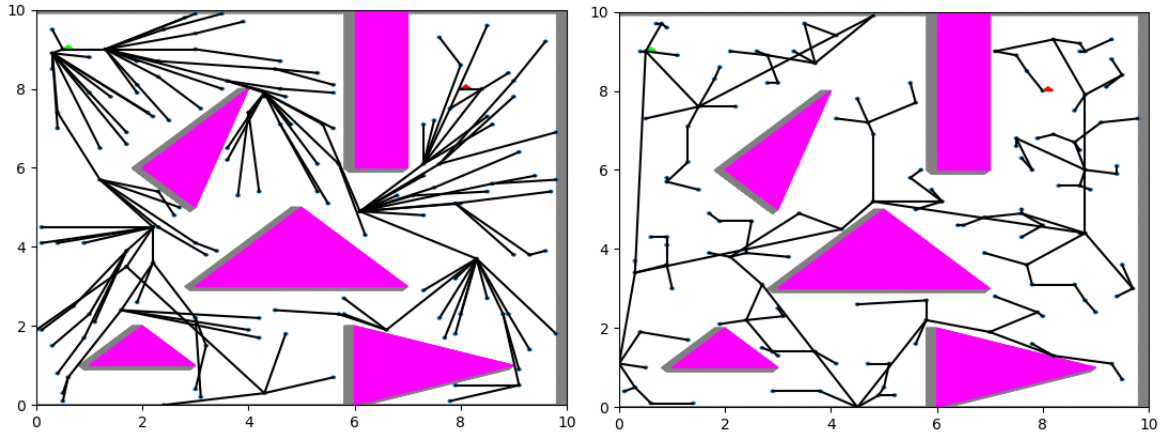


The graph below is for the average path distance vs iterations for RRT.



On both the graphs, 30 path length represents no path found for that iteration. We can notice that the average path length gets smaller as the iterations decrease. This can be explained by the fact that as we sample more, we find more points on the configuration space and therefore shorter paths. We can also notice that the paths for RRT* are, on average, samller than the paths for RRT. This can be explained by the rewiring function for RRT* that tries to reconfigure the tree to get the shortest the nodes. The path quality is also better with RRT* because of the rewiring that results in the path sticking closer to the edges of the

obstacles and always pointing towards empty space. An example of the RRT* (left) and RRT (right) can be seen in the graphed search trees below. Both the algorithms were run with 200 iterations. We can clearly see that the edges for RRT* are pointing to empty space and the cost to get to a node is always optimal, given the nodes in the tree. On the other hand, for RRT the edges are very random and they overlap much more. WE can also notics that RRT* seems to have discovered a lot more of the configuration space than RRT.



1.9 Visualize RRT*

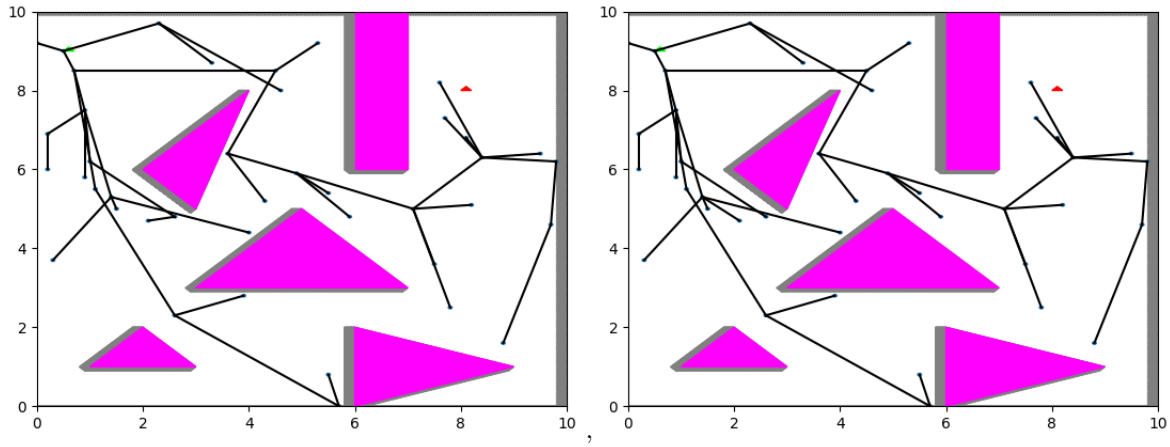
Lastly. a function to visualize the RRT* function is implemented. This function plots the configuration space, with the robot at the start and goal points. The growth of the tree is then animated.

```
def visualize_rrt_star(robot, obstacles, start, goal, itern_n):

    tree_struct = Tree(robot, obstacles, start, goal) # initialize the tree structure with given
        data

    for i in range(itern_n): # run the loop for given n iterations

        sample_point = sample()
        while not isCollisionFree(robot, sample_point, obstacles) or tree_struct.exists(sample_point
            ):
            # get asample that is collision free and a point that is not in the tree
            sample_point = sample()

        if len(tree_struct.node_data) == 1: # if it's the first point add it to the start node

            tree_struct.extend(start, sample_point)

        else:
            nearest_point = tree_struct.nearest(sample_point) # find the nearest point to the sample
                point
            if tree_struct.extend(nearest_point, sample_point): # if the sample point was added,
                rewire
                tree_struct.rewire(sample_point, 1.5)

    if not tree_struct.exists(goal): # if the goal not is not in the tree
        nearest_to_goal = tree_struct.nearest_collision_free(goal) # find a node that is nearest to
            the goal and is
        # collision free
        if nearest_to_goal is None: # if there is no point that is collision free close to goal,
```

```
        return None
        visualize_points(tree_struct.node_data.keys(), robot, obstacles, start, goal)
        return None
    path_to_goal = tree_struct.extend(nearest_to_goal, goal) # if there is a node, extend from
        node to goal
else:
    path_to_goal = True

path = [goal] # add goal to the path list
if path_to_goal:
    curr_node = goal
    while curr_node != start: # while we don't reach the start node
        path.append(tree_struct.parent(curr_node)) # keep adding the parent of the current node
        curr_node = path[-1] # update the current node to be the last node in the path list

    path.reverse() # reverse the path

plt.xlim(0, 10)
plt.ylim(0, 10)
robot_start = translate(robot, start) # robot points at the start position
robot_strt_pts = np.array(robot_start)
robot_p_st = Polygon(robot_strt_pts, color=[0, 1, 0])
robot_goal = translate(robot, goal) # robot points at the goal position
robot_goal_pts = np.array(robot_goal)
robot_p_gl = Polygon(robot_goal_pts, color=[1, 0, 0])
display = plt.gca()
display.add_patch(robot_p_st)
display.add_patch(robot_p_gl)
flipped_robot = []
for point in robot: # flip the robot over the y = -x line
    flipped_robot.append((-point[0], -point[1]))
boundary_list = [(0, 0), (0, 10), (10, 10), (10, 0)]
obstacles.append(boundary_list)
for obs in obstacles:

    for i in range(len(obs)): # for every line of the obstacles, traverse the flipped robot over
        the line

        start_point = obs[i]
        end_point = obs[i - 1]
        dx = start_point[0] - end_point[0]
        dy = start_point[1] - end_point[1]
        x_step = dx / 400
        y_step = dy / 400
        for j in range(401): # discritize the line and add a patch of the robot at every point
            node = (end_point[0] + (x_step * j), end_point[1] + (y_step * j))
            points_list = translate(flipped_robot, node)
            plgn = Polygon(np.array(points_list), color=[.5, .5, .5])
            display.add_patch(plgn)

obstacles.remove(boundary_list)

for i in obstacles: # add patches for the obstacles
    obs = Polygon(i, color=[1, 0, 1])
    display.add_patch(obs)

for j in tree_struct.node_data.keys():
    temp_pnt = plt.Circle(j, .05) # for all the given points, add a small circle patch
    display.add_patch(temp_pnt)
```

```
            x_vals = [j[0], tree_struct.parent(j)[0]] # add the tree stucture lines
            y_vals = [j[1], tree_struct.parent(j)[1]]
            plt.plot(x_vals, y_vals, color=[0, 0, 0])
            plt.draw()
            plt.pause(.01) # animate the tree growth

    plt.show()
```

Below is an example of before and after rewire function was run:



On the left is the graph before rewiring and on the right after rewiring. There was only one node that was rewired, it was around point (2, 4.5) on the left image and a shorter path was found from another node so the point was assigned a new parent. The rewiring helps make sure the path to a point is the shortest possible using the points in the tree. The rewiring can be made more optimal as we increase the radius r. The larger the r value, the more neighbors will be examined and more optimal path would be found. But increasing r comes with more computations since we not have to run collision detection for many more points.

2 Geometric Motion Planning for a Car

This part of the assignment involves a rectangular object that represents a car that can translate in 2D and rotate.

2.1 Model the robot by separating the geometry and pose

First, we have to implement a new class called Robot that takes the width and height of the robot as input. The class also holds two functions that are responsible for getting the vertexes of a robot at given point and a rotation angle.

```
    def __init__(self, width, height):

        self.width = width
        self.height = height
        self.translation = []
        self.rotation = 0

    def set_pose(self, pose):

        self.translation = (pose[0], pose[1])
        self.rotation = pose[2]

    def transform(self):

        rtrn_list = []
        row_1 = [math.cos(self.rotation), - math.sin(self.rotation)]
        row_2 = [math.sin(self.rotation), math.cos(self.rotation)]
        rotation_matrix = np.array([row_1, row_2])
        point = np.array([self.translation[0], self.translation[1]])
        l_b = (-self.width / 2, -self.height / 2) # left bottom point
        l_t = (-self.width / 2, self.height / 2) # left top point
        r_t = (self.width / 2, self.height / 2) # right top point
        r_b = (self.width / 2, -self.height / 2) # right bottom point

        start_loc = [l_b, l_t, r_t, r_b]

        for i in start_loc:
            i_array = np.array([i[0], i[1]])
            temp_array = np.dot(rotation_matrix, i_array) + point
            rtrn_list.append([temp_array[0], temp_array[1]])

        return rtrn_list
```

2.2 Parser for the input file, and visualize the obstacles and the planning problem

The parser function and visualize_problem for this problem are very similar to the parser and visualize_problem mentioned earlier in the report. For this problem each point of the robot holds three values - x, y, and rotation angle. For the visualize_problem function, the robot at start and goal locations are now rotated to their respective start and goal angles.

2.3 Extend the previous functions

For getting the random points suitable for this problem, the sample function was modified to return a tuple that holds random x, y and rotation angle. The rotation angle is between -π and +π.

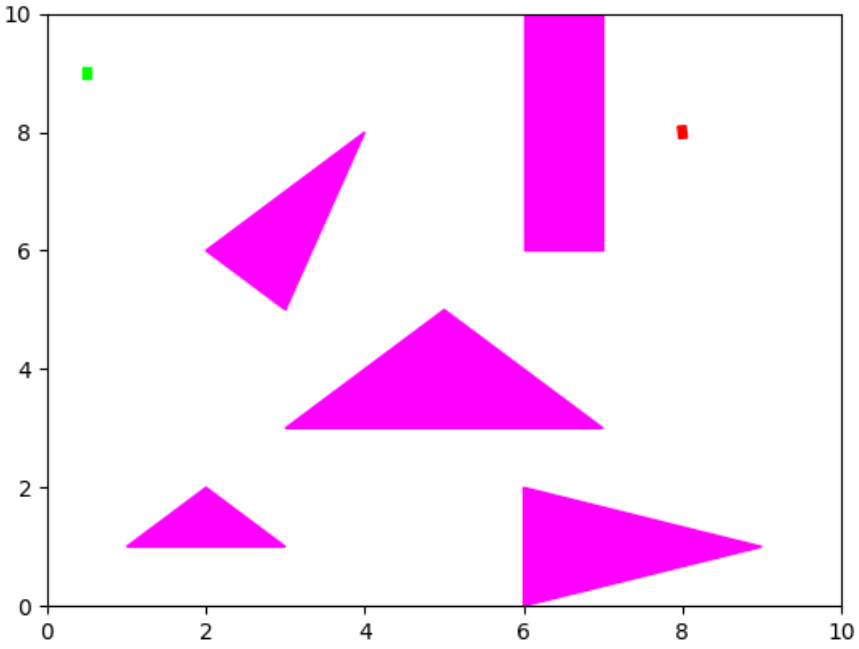
```
def sample():

    rtrn_val = (random.randrange(0, 100)/10, random.randrange(0, 100)/10, random.randrange(-314,
        314)/100)

    return rtrn_val
```

Next, the Tree class is modified so the distance used in the add and nearest function take the angle difference into account. The new equation used for distance is:

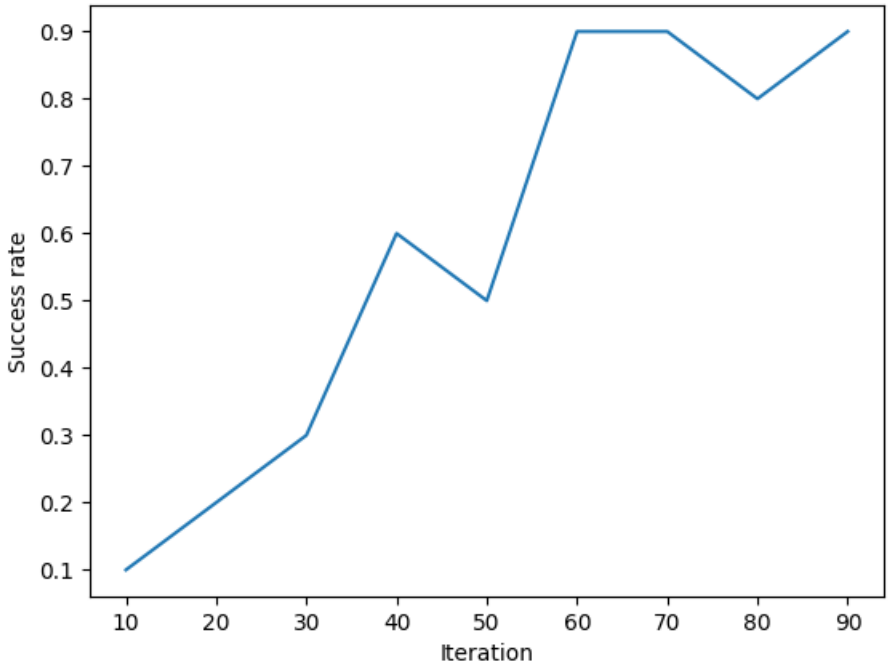
$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + d(\theta_1, \theta_2)^2}$$

2.4 Implement the complete RRT algorithm

The RRT algorithm function for this part is very similar to the RRT implemented earlier in the report. One of the difference is how the points for the transformed robot are obtained. Instead of the translate function, we now use the pose and transform function of the Robot class. Another difference is that the input to the function is a Robot class variable, obstacles, and stat and goal locations. The start and goal locations now have the start and goal rotation angles. The problem visualized below is used for comparing the iterations vs. success.

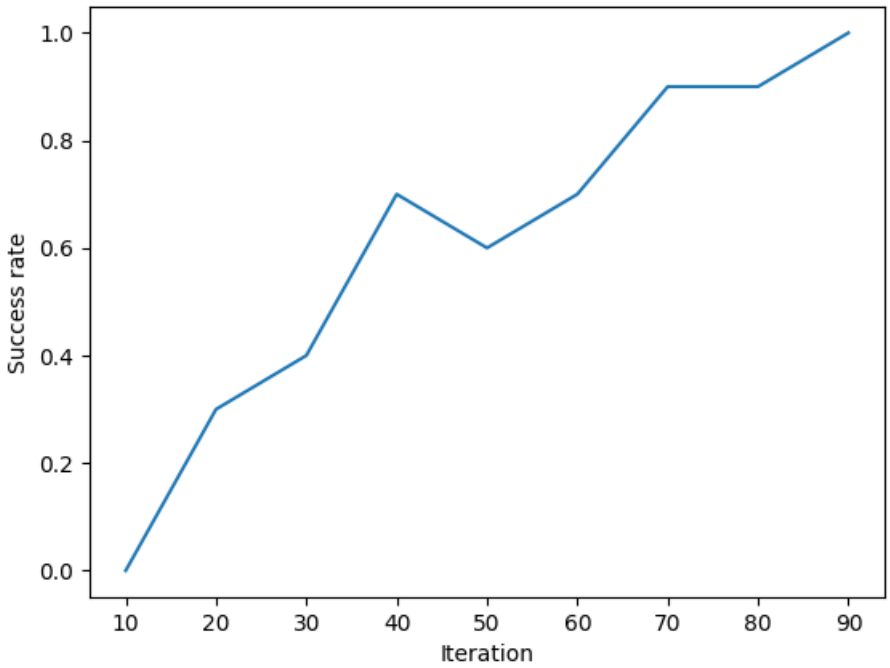


The iterations vs. success is given below. The data was collected in the same way as before. We can notice that as the number of iteration increase, the success rate also increases.

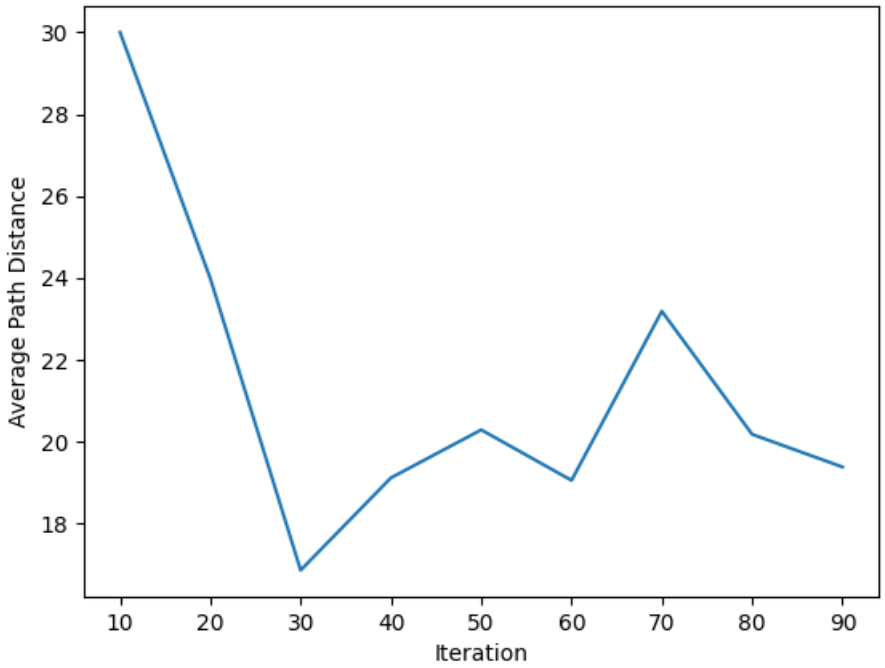


2.5 Implement the complete RRT* algorithm

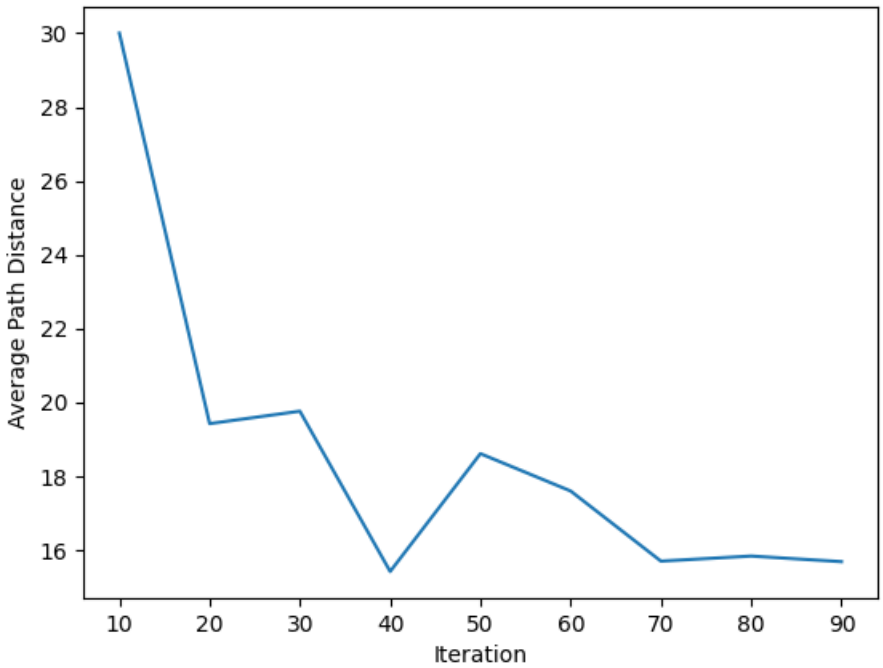
The implementation of RRT* was also similar to the RRT* implementation mentioned before. One of the only difference was the distance used to find the closest node. The success rate vs. iteration data was collected the same way as before and it is graphed below. As we have seen before, the success rate increases with iterations, since we discover more area.



The data for average path distance for RRT and RRT* were also collected. The data for RRT is graphed below:



The data for RRT* is graphed below:



Similar as before the average path distance for RRT* is smaller than RRT because of the rewiring done after every new node is added to the tree.

2.6 Visualization of the found path

The visualization of the path was also very similar to the path visualization done earlier in the report. The animation function from matplotlib was used to animate the robot over the path found. The inputs to the function are an instance of robot class, obstacles and a list of 3D tuples of the path found.

3 Kinematic Motion Planning for a Holonomic Car

3.1 Extend the planning problem to involve kinematics of the car

The kinematics function takes in the current state of the robot and the control that need to be applied to it. It returns the final velocity control has been applied to the current state.

```
def kinematics(self, state, control):

    x_new = control[0] * math.cos(state[2] + math.pi/2)
    y_new = control[0] * math.cos(state[2] + math.pi/2)
    rtrn_val = (x_new, y_new, control[1])
    return rtrn_val
```

The propagate function takes in the current state of the robot, controls that need to be applied to the robot, duration, and time step dt. It returns a list of the the state of the robot at every dt time step.

```
    def propogate(self, state, controls, durations, dt):

    if durations == 0:
        return [state]

    rtrn_list = [state]

    start = state

    for i in range(int(durations/dt)):

        temps = start
        temp_control = controls[i]

        temp_kinematics = self.kinematics(temps, temp_control)

        delta_x = temp_kinematics[0] * dt
        delta_y = temp_kinematics[1] * dt
        delta_theta = temp_kinematics[2] * dt
        correct_theta = start[2] + delta_theta

        while correct_theta > math.pi: # make sure the angle is between -pi and pi
            correct_theta -= 2 * math.pi
        while correct_theta < -math.pi:
            correct_theta += math.pi

        new_state = (start[0] + delta_x, state[1] + delta_y, correct_theta) # new state after dt

        start = new_state

        rtrn_list.append(new_state)

    return rtrn_list
```

Lastly, the extend function is modified for this problem. Now the inputs are the starting point, min and max linear velocity and the time step dt. The man and min velocities were chosen to be 10 and 20, respectively. The min and max angular velocity was chosen to be $\pm.1 * \pi$. The dt was selected to be .1.

```
    def extend(self, point, n1, n2, dt):

rand_duration = random.randint(n1, n2)

control_list = []
linear_v = random.randrange(-20, 200) / 10 # max of 2 units per second
anglular_v = .1 * random.randrange(-314, 314) / 100 # max of .5 * pi per second
for i in range(int(rand_duration/dt)):

    control_list.append((linear_v, anglular_v))

temp_path = self.robot.propogate(point, control_list, rand_duration, dt)

path = []

for temp_point in temp_path:

    self.robot.set_pose(temp_point)
    robot_points = self.robot.transform()
    if temp_point != temp_path[0]:
        temp_point_index = temp_path.index(temp_point)
        if self.isCollisionFree(temp_path[temp_point_index - 1], temp_point) and
            collision_checker(robot_points, temp_point, self.obstacles):
           path.append(temp_point)
           self.add(temp_path[temp_point_index - 1], temp_point)
        else:
           break

return path
```

3.2 Extend the RRT algorithm to work for Kinematic Motion Planning

The same RRT algorithm has been used for this part of the problem as the one motioned earlier in the report. One major difference is that now we use the updated extend function to randomly shoot in a direction and the velocity and dt are changeable.