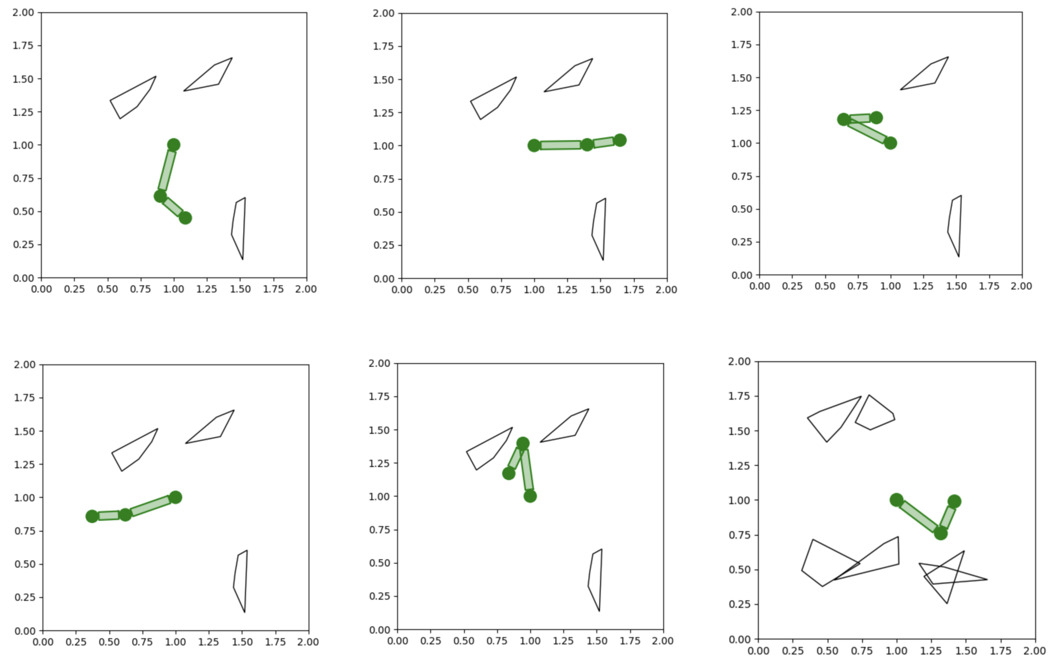
CS 460: INTRO TO COMPUTATIONAL ROBOTICS

Assignment 2

Lorenzo Nieto
len44@rutgers.edu
Tasha Pais
tdp74@rutgers.edu
Rutgers University, Department of Computer Science
October 8, 2023

1 Motion Planning for a 2-link Planar arm

1.1 Sampling random collision-free configurations



The functions in the provided code are part of a system for simulating and checking collisions for a robotic arm with multiple links.

Function: `check_collision_with_obstacle`

The `check_collision_with_obstacle` function is responsible for determining whether any part of the robotic arm intersects with a given obstacle. The function iterates through each link of the arm and each joint, treating them as separate entities for collision detection. For each link, it creates a rectangle representing the link's position and orientation, then checks if this rectangle collides with the obstacle using the `collision_checking.collides` function. The function does the same for each joint, representing them as circles (approximated to polygons) and checking for collisions. If any link or joint intersects with the obstacle, the function returns `True`, indicating a collision.

Function: `update_points`

The `update_points` function calculates the position of each joint (and the end effector) of the robotic arm based on the current joint angles. It iteratively calculates the position of each joint by adding the length of each arm link multiplied by the cosine and sine of the sum of the joint angles up to that point. This approach assumes that the arm's links are rigid and that each joint rotates only in the plane. The calculation is geometric, based on the principles of trigonometry and the kinematic chain of the arm.

Function: `rotate_joint`

The `rotate_joint` function allows for the rotation of a specified joint of the robotic arm. It first stores the current joint angles in case a rollback is needed. Then, it increments the angle of the specified joint by a fixed amount (5 degrees, converted to radians). After updating the joint angles, it recalculates the positions of all points in the arm using `update_points`. It then checks for collisions with each provided obstacle. If a collision is detected, the function reverts the joint angles to their original state and updates the arm's points again to reflect this reversion, effectively undoing the rotation.

Function: `collides`

The `collides` function checks for collisions between two polygons. It uses a geometric approach by testing if any edges of one polygon intersect with edges of the other. The function `edges_intersect` is used for this purpose, which checks whether two line segments intersect. The function also checks if any vertex of one polygon is inside the other, which is another condition for collision. The overall approach combines edge-edge intersection tests and point-in-polygon tests to comprehensively determine if the two polygons overlap. If any of these tests are positive, `collides` returns `True`, indicating a collision.

1.2 Nearest neighbors with linear search approach

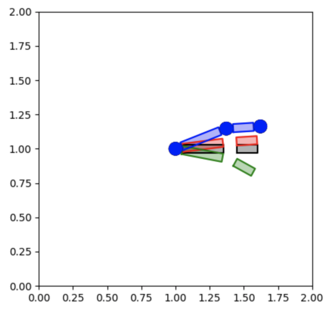


Figure 1: target=[0,0], k = 3

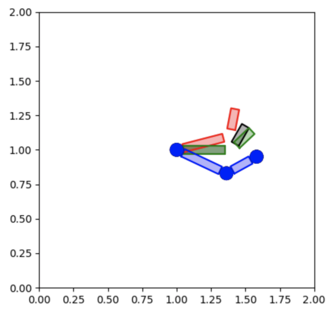


Figure 2: target=[0, 1.047], k = 3

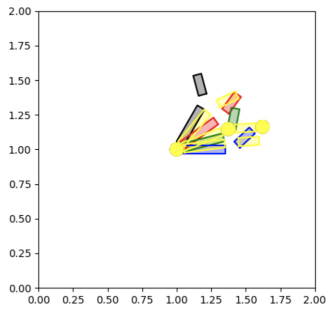


Figure 3: target=[1.047, 0.785], k = 6

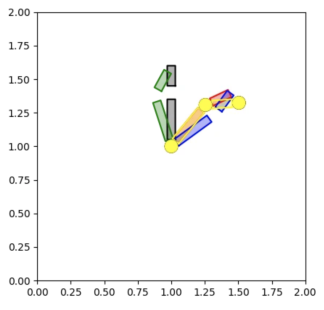


Figure 4: target=[1.571, 0], k = 4

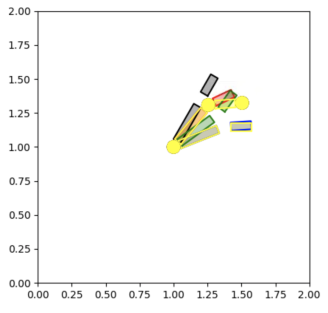


Figure 5: target=[1.047, 0], k = 5

The implementation for the naïve linear search approach to finding the three nearest neighbors to a target configuration involves the following steps: In the context of the robot's configuration space, the Euclidean distance ($d_{\text{Euclidean}}$) is used as the distance metric. Euclidean distance measures the straight-line distance between two points in a space. For each pair of configurations (represented as points in a multi-dimensional space), the Euclidean distance is calculated as follows:

$$d_{\text{Euclidean}}(\mathbf{config}_1, \mathbf{config}_2) = \sqrt{\sum_{i=1}^{n} (\mathbf{config}_1[i] - \mathbf{config}_2[i])^2}$$

Here, $\mathbf{config}_1$ and $\mathbf{config}_2$ are two configurations, each represented as a vector of joint angles in the robot's configuration space, and n is the number of dimensions (joint angles).

For each configuration in the dataset, the Euclidean distance between that configuration and the target configuration is computed. These distances are stored in a list:

$$\text{distances} = [d_{\text{Euclidean}}(\text{target}, \text{config}) \text{ for config in configs}]$$

Here, target is the target configuration, and configs is the list of all configurations in the dataset.

After computing the distances, the np.argsort function is used to obtain the indices of the configurations with the smallest distances (i.e., the nearest neighbors). The first three indices are selected to obtain the three nearest neighbors:

$$\text{neighbor\_indices} = \text{np.argsort}(\text{distances})[: k]$$

Here, k is the number of nearest neighbors to be found (in this case, 3).

Discussion on Topology of Configuration Space:

The configuration space of the robot is the space of all possible joint configurations. It is typically a multi-dimensional space, with each dimension representing a joint angle. The Euclidean distance metric is suitable for this space because it calculates distances in a way that corresponds to the physical space traversed by the robot arm as it moves from one configuration to another. In other words, it measures how much the joint angles need to change to go from one configuration to another.

The naïve linear search approach works well for small datasets but can be computationally expensive for large datasets since it computes the distances between the target and all configurations. If efficiency is a concern, other data structures like kd-trees or ball trees may be explored to accelerate nearest neighbor search.

In summary, the implementation computes the Euclidean distances between the target configuration and all configurations in the dataset and selects the three configurations with the shortest distances as the three nearest neighbors. This approach is suitable for the topology of the robot's configuration space and provides a reasonable measure of similarity between configurations.

1.3 Interpolation along the straight line in the C-space

The `interpolate_configs` function is responsible for generating a sequence of configurations (joint angles) that smoothly interpolate between a given start configuration and a goal configuration. This interpolation is achieved by linearly dividing the path between the start and goal into a specified number of steps (`num_steps`). In each step, the function calculates an intermediate configuration by taking a weighted average of the start and goal configurations. Specifically, for each step, it computes the new configuration by adding a fraction of the difference between the goal and start configurations to the start configuration. This process continues for all steps, ensuring a continuous and linear transition between the two endpoint configurations. The result is a list of configurations that represent the robot arm's path from the start to the goal, with the number of configurations determined by the `num_steps` parameter. This smooth interpolation is crucial for creating visually appealing animations of the robot arm's motion between different joint configurations.

1.4 RRT (Rapidly-Exploring Random Tree) Implementation

The provided code demonstrates the implementation of the RRT algorithm for path planning in robotics. The goal of the RRT algorithm is to find a collision-free path from a start configuration to a goal configuration in the presence of obstacles.

The implementation utilizes the following components:

1. **RRTNode Class:** The RRTNode class represents a node in the RRT tree. Each node contains a configuration (joint angles) and a reference to its parent node. The configuration is stored as a NumPy array.

2. **rrt Function:** The main RRT algorithm is implemented in the rrt function. It takes the following parameters:

   - `start_config` and `goal_config`: The starting and goal configurations, respectively.
   - `obstacles`: A description of the obstacles in the environment.
   - `max_nodes`: The maximum number of nodes to generate in the RRT tree (a termination condition).

3. **RRT Initialization:** The algorithm starts with a single node representing the start configuration. This node is stored in a list called `nodes`.

4. **Random Sampling:** The algorithm repeatedly generates random configurations. With a probability defined by `GOAL_SAMPLE_RATE`, it samples the goal configuration; otherwise, it generates a random configuration using `generate_random_configuration()`.

5. **Finding the Nearest Node:** The algorithm identifies the nearest node in the current RRT tree to the randomly sampled configuration. This is done using the `find_nearest_node()` function.

6. **Steering Towards Random Configuration:** The algorithm then attempts to "steer" from the nearest node towards the random configuration, resulting in a new node. This is done using the `steer_towards()` function.

7. **Collision Checking:** Before adding the new node to the tree, the algorithm checks whether the path from the nearest node to the new node is collision-free by calling the `check_collision()` function. If there is a collision, the new node is discarded.

8. **Node Addition:** If the path is collision-free, the new node is added to the RRT tree, and its parent is set to the nearest node.

9. **Goal Reached Check:** After adding a new node, the algorithm checks if the new node is close to the goal configuration (using a specified threshold). If the condition is met, a valid path from the start to the goal has been found, and the algorithm terminates, returning the RRT nodes representing the path.

10. **Termination:** If the algorithm reaches the maximum allowed number of nodes (`max_nodes`) without finding a valid path, it terminates and returns the RRT nodes generated up to that point without a path.

In summary, the RRT algorithm incrementally builds a tree of configurations by randomly sampling and connecting nodes. Edges are added to the tree if the path between nodes is collision-free. The algorithm continues until it either finds a valid path from the start to the goal or exhausts the maximum number of nodes allowed.

1.5 PRM (Probabilistic Roadmap) Algorithm Implementation

In this implementation, the PRM algorithm is utilized to find a collision-free path from a given start configuration to a goal configuration while avoiding obstacles indicated in the map file. The key components of the PRM implementation are as follows:

1. **PRM Data Structures:** The PRM algorithm maintains several data structures, including `prm_nodes` to store sampled configurations, `prm_edges` to store valid edges between configurations, and `parents` to maintain the parent-child relationships between configurations. The `parents` dictionary is initialized with the start configuration as having no parent.

2. **Sampling Configurations:** The algorithm begins by adding the start configuration to `prm_nodes`. It then repeatedly samples random configurations. With a probability defined by `GOAL_SAMPLE_RATE`, the algorithm samples the goal configuration; otherwise, it generates a random configuration using `generate_random_configuration()`.

3. **Collision Checking:** For each sampled configuration, the algorithm performs collision checking using `check_collision()`. If the sampled configuration is collision-free, it is added to `prm_nodes`.

4. **Finding Neighbors:** The algorithm uses a k-d tree (`KDTree`) to efficiently find the k-nearest neighbors of the sampled configuration among existing nodes in `prm_nodes`. The neighbors are identified based on Euclidean distance.

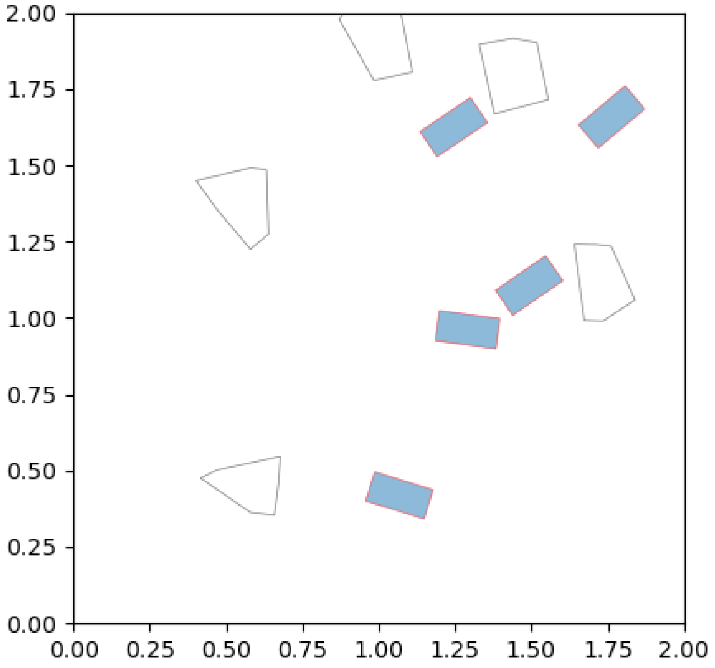
5. **Edge Collision Checking:** For each neighbor found, the algorithm checks if the edge (connection) between the neighbor and the sampled configuration is collision-free using `check_edge_collision()`. If the edge is collision-free, it is added to `prm_edges`, and the parent of the sampled configuration is updated to the neighbor, establishing a parent-child relationship.

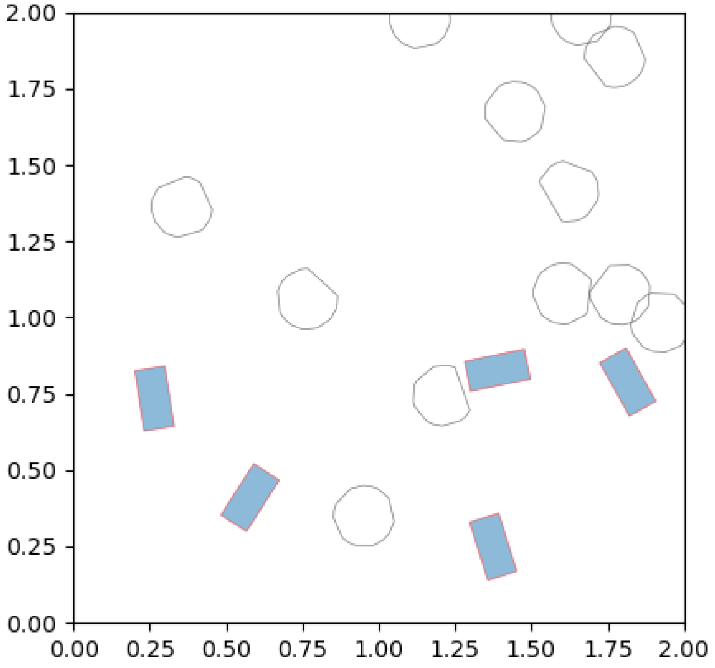
6. **Goal Configuration Reachability:** The algorithm also checks if the goal configuration is reachable by finding its k-nearest neighbors. If a collision-free edge can be established between a neighbor and the goal configuration, the algorithm adds this edge to `prm_edges`, and the parent of the goal configuration is updated.

7. **Termination:** The algorithm continues to sample configurations, add valid edges, and update parent-child relationships until a specified maximum number of nodes, `MAX_NODES`, is reached. This termination condition ensures computational efficiency and control over the size of the PRM.

Upon completion of the PRM construction, the algorithm provides a roadmap in the form of `prm_nodes` and `prm_edges`. A path from the start to the goal configuration can be obtained by traversing the parent-child relationships established in the `parents` dictionary. The PRM implementation is a key step in path planning for robots operating in complex environments, and it provides a way to find safe and efficient paths while considering obstacle avoidance.

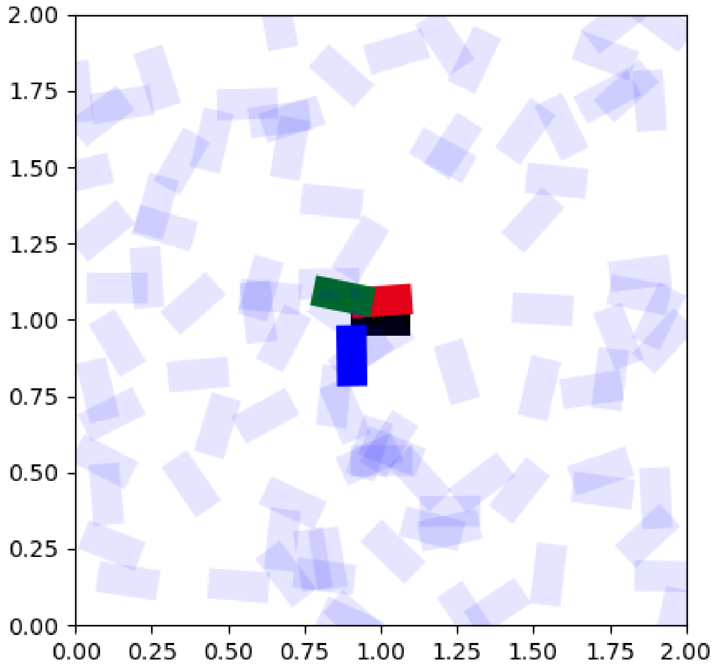
2.1



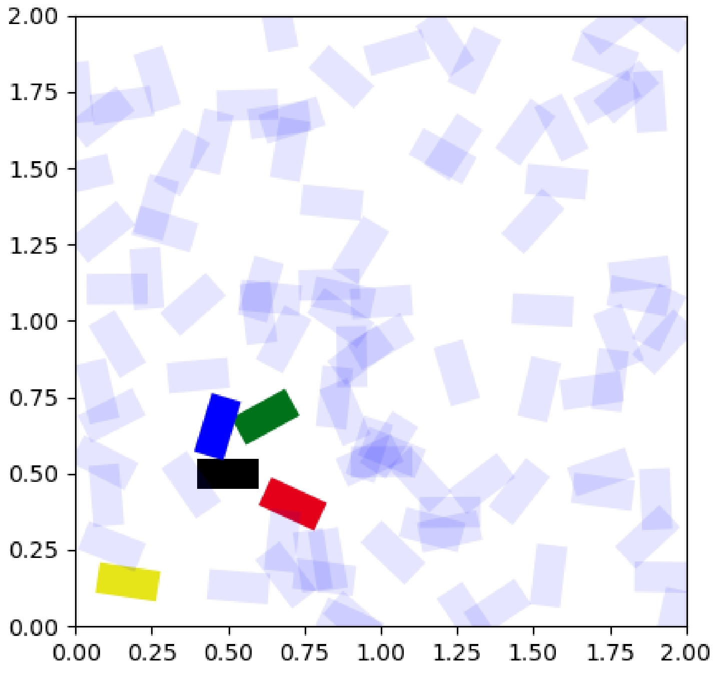


For part 2.1, configurations were randomly and uniformly sampled. This was done by randomly generating a value for each dimension of the configuration space. Then, it is checked

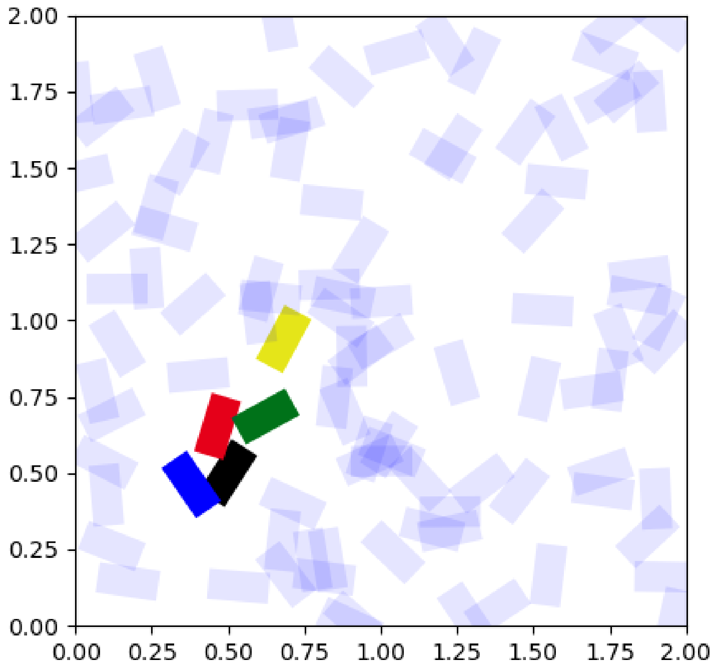
if the configuration collides with obstacles or if it goes outside the 2x2 region. This is done by mapping the configuration to the workspace. If the configuration is collision free, it is kept and its position in the workspace is shown.
2.2



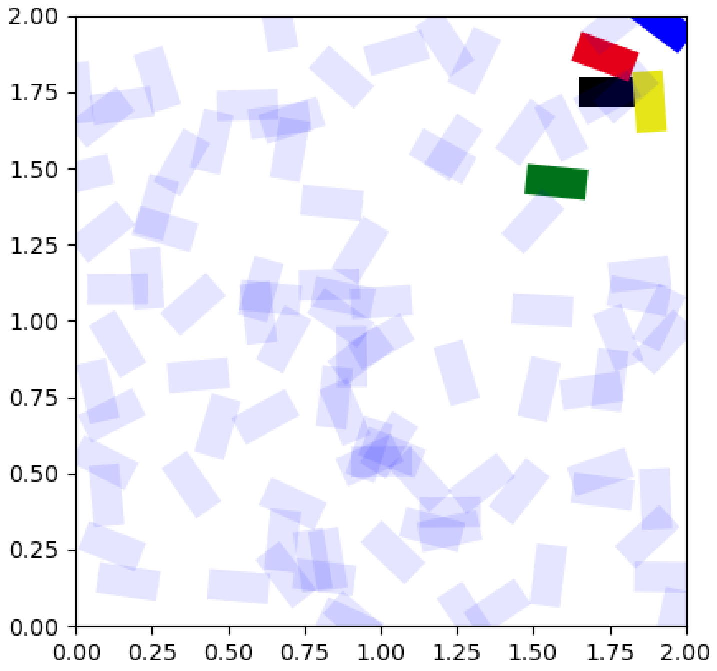
–target 1 1 0 –k 3



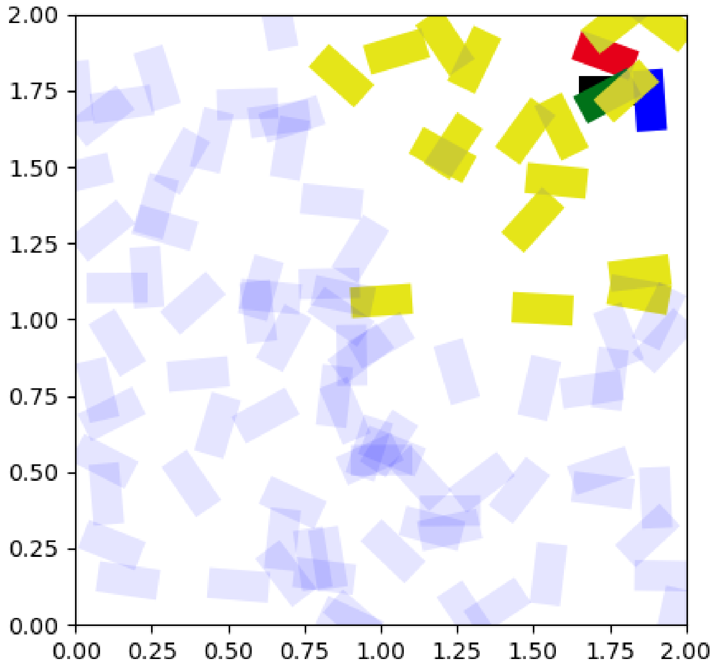
–target .5 .5 0 –k 4



–target .5 .5 1 –k 4



–target 1.75 1.75 0 –k 4



–target 1.75 1.75 0 –k 20

For 2.2, the distances of all the file's configurations to the goal configuration were stored

in an array. The distance was calculated by the formula Distance = .7e + .3r where e is the euclidean distance between points the center points of each configuration and r is the rotational distance, which was approximated by taking the difference of the angles of the configurations in radians and halving this number. Steps were taken to make sure that the rotational distance was always the smallest angle that it took to get from one angle to the other, so it should never exceed 180 degrees. All configurations in the file were plotted in light blue to aid in visualization of the closest k configurations
2.3
*see rigidbody2.3-1 - rigidbody2.3-3 gifs
For 2.3, the distance between the start configuration and the goal configuration for each dimension (x, y, theta) was calculated. Again, it was made sure that the distance between both angles was the shortest possible distance in the configuration space, meaning it should never exceed 180 degrees. Then a certain fraction of these distances were calculated at each point in time and the robot would be transformed with these distances to get its position at a particular point in time. The result is a smooth animation showing the robot travelling along the shortest straight line path that connects the start and goal in the configuration space. The goal configuration was plotted in red to aid in the visualization of the robot's path
2.4
*see 2.4-1 - 2.4-6 gifs
For 2.4, RRT was employed with the help of functions from previous problems. This was standard RRT so no rewiring was done and a path was returned as soon as it was found. The program was given a maximum of 2000 samples to try and find a valid path. Also, the goal configuration was sampled every 5 samples to aid in the process of discovering a path. The paths returned by the program are not of the highest quality because they were returned as soon as they were found. However, paths are returned relatively quickly if they exist and the success rate was very high for the different maps and start/goal configurations that were tested. The goal configuration was plotted in red to aid in visualization
2.5
*see 2.5-1 - 2.5-6 gifs
For 2.5, PRM was employed to build a roadmap and discover a path between two configurations. For this implementation, new nodes are attempted to be connected to the nearest 3 nodes on the existing graph. If no such connection exists, the node is discarded and a new one is generated. A total of 500 random nodes are generated and connected in this way. Then, the start and goal configuration nodes are connected to the tree. The distance metric between two configurations is the same one that was used in 2.2. After the roadmap is generated, a path between the start and goal nodes is found using the A* algorithm. The heuristic for this is just the distance between the current node and the goal node, using same distance metric as in previous parts. This method again has a very reliable success rate, but it is slower to generate the roadmap. One advantage it has over RRT however is that once it already has a roadmap, finding paths between start and goal nodes is a simple and fast process. It also has the tendency to produce higher qulity paths than RRT because A* is used to find the optimum path given the current roadmap.
3.1
*see car1.gif

The car model used for 3.1 is the same one as the differential drive model given in recitation, but with some slight modifications to account for the constraints that cars have in their motion. When running the code, press the "x" key to end the animation.
3.2
*see car2.gif
For 3.2, practically the same code is used as in part 3.1, but the control value is kept constant at whatever input was given to the program. When running the code, press the "x" key to end the animation.
3.3
*see car3 - car5 gifs
Part 3.3 implements RRT but for a robot with car-like constraints on its motion. Overall the same steps are followed as given in the instructions of the assignment. For this implementation, the size of the blossom of random controls is 30. This number was chosen to better our chances of sampling a control that takes us closer to our goal configuration, but it comes at the cost of performance, as a smaller blossom size would mean less computations. Also, the program is allowed to run for 4000 samples while it tries to find a valid path. One notable thing about this implementation is that the time that each control is allowed to run for (delta T) is variable. In the first 75 percent of samples, the end configuration of the control is 2 iterations forward in time, while in the remaining samples the end configuration of the control is only 1 iteration forward in time. This allows the robot to make more fine adjustments towards the end of the sampling phase when it might have nodes close to the goal configuration but not quite in the goal region yet. Also, the frequency of sampling the goal configuration is variable, In the first 75 percent of samples, the goal configuration is sampled once every 10 samples, while in the remaining samples it samples the goal region every once in 5 samples. This encourages the RRT tree to get closer to the goal configuration and find a path after it is assumed to have discovered a decent portion of the map. This combination of number of total samples, blossom size, goal configuration sample frequency, and delta T values was found to result in a valid path a vast majority of the time for the different maps and start/goal configurations tested. This hit rate, while high, is lower than for RRT and PRM for the rigid body in part 2. However, the solutions are often found relatively slowly. A solution can be found quickly if we get lucky, but this is not the case most of the times. In the animation, the goal configuration is plotted in red to aid in visualization