Problem Set 2

Done by **Fatykhoph Denis** in Skoltech, Planning Algo-s, 2024

Task1. Visualization

In [ ]:
```
import pickle
import numpy as np
from environment import State, ManipulatorEnv

%matplotlib inline
import matplotlib.pyplot as plt
import angle_util as au
from icecream import ic
from typing import List
```

In [ ]:
```
a = np.array([20., 30., 40.])
b = np.array([15., 45., 78.])

ic(au.angle_linspace(a, b, n=5))
ic(au.angle_difference(a, b))
```

In [ ]:
```
# Load data from the pickle file
with open("data.pickle", "rb") as f:
    data = pickle.load(f)

start_state = State(np.array(data["start_state"]))
goal_state = State(np.array(data["goal_state"]))
obstacles = np.array(data["obstacles"])
collision_threshold = data["collision_threshold"]

# Create environment with start state
env = ManipulatorEnv(obstacles=obstacles,
                     initial_state=start_state,
                     collision_threshold=collision_threshold)

plt.figure()
plt.title("Manipulator in Start State")
env.render(plt_show=True)

env.state = goal_state

plt.figure()
plt.title("Manipulator in Goal State")
env.render(plt_show=True)
```

In the previous problem set (PS1) with *discretized orientation space*, the *state space was finite*, which allowed for simpler algorithms like *Dijkstra* or *Astar* to be applied directly. However, this comes at the cost of granularity: higher precision requires finer discretization, leading to exponential growth in states.

In contrast, this problem set uses a *continuous orientation space*. While it provides higher precision and more realistic modeling of the manipulator's motion, it complicates planning as we must use advanced techniques like sampling-based algorithms to explore the infinite state space effectively.

In [ ]:
```
for i in range(0, 4):
    i += 1
    env.state = State(np.random.randint(0, 180, 4).astype(float))

    plt.figure()
    plt.title("Manipulator in Goal State")
    condition, point = env.check_collision(env.state)
    if (condition):
        print(
            f'Attention!!! Collision detected '
            f'in link {point['link_segment']} '
            f'joint on coordinates: {np.round(point['collision_point'], 2
        )
    env.render(plt_show=True)
```

Originally, the function `ManipulatorEnv.check_collision` was proposed to detect collision for our manipulator. It was working well, and each time at least one joint or part of link was in collision with objects it returned boolean `True`. Which indicated to us that our robot in collision.

But, I modified it a little bit. So, for now, if function detects collision it returns not only the indicator for collision, but also the link which in collision and coordinates of joint, who created this situation.

Task2. Rapidly-exploring Random Trees

In [ ]:
```
def config_creation(
                    arbitrary_increment: int,
                    linspace_step: int,
                    q_init: np.ndarray = np.nan,
                    q_goal: np.ndarray = np.nan
                    ) -> List:
    """
    Generates random angle_linspase

    :param arbitrary_increment: with this variable you can control
                                scaling of goal configuration according
                                to the initial configuration
    :param linspace_step: variable which denotes the size of linspace
    :param q_init: allows to manually setup initial state of system
```

```
    :param q_goal: allows to manually setup goal state of system

    :return: List.
            a sequence of configurations connecting initial configuration
            and goal configuration
    """
    if np.isnan(q_init).any():
        q_init = np.random.randint(0, 180, 4).astype(float)

    if np.isnan(q_goal).any():
        while True:
            q_goal = np.round((2)**(np.random.random()) * q_init)
            if (np.abs(q_goal) >= 0.0).all() and (np.abs(q_goal) <= 180.0
                break

    linspace_configs = au.angle_linspace(q_init, q_goal, linspace_step)

    return [State(angles) for angles in linspace_configs]
```

In [ ]:

```
config_space = config_creation(arbitrary_increment=10,
                                linspace_step=2)
for state in config_space:
    print(state.angles)
    env.state = state
    condition, point = env.check_collision(env.state)
    if (condition):
        print(
            f'Attention!!! Collision detected '
            f'in link {point['link_segment']} '
            f'joint on coordinates: {np.round(point['collision_point'], 2
        )
    else:
        print('No collision detected')
    env.render(plt_show=False)
```

In [ ]:

```
config_space = config_creation(10, 2,
                q_init=np.array([35., 20., 68., 27.]),
                q_goal=np.array([151., 35., 118., 47.]))
for state in config_space:
    print(state.angles)
    env.state = state
    condition, point = env.check_collision(env.state)
    if (condition):
        print(
            f'Attention!!! Collision detected '
            f'in link {point['link_segment']} '
            f'joint on coordinates: {np.round(point['collision_point'], 2
        )
    else:
        print('No collision detected')
    env.render(plt_show=False)
```

Completing this part of task helped me to understand how to work with `angle_util.py`. Moreover, I chose a sequence of 3 because this is the minimum set needed to demonstrate the required state. However, the `angle_linspace` function allows us to achieve any required n discretization of the continuous space between the initial and final states.

Below I just played with creation rrt for points, not for manipulator. You can skip it

In [ ]:
```
def rrt_creation(
                    arbitrary_increment: int,
                    linspace_step: int,
                    q_init: np.ndarray = np.nan,
                    q_goal: np.ndarray = np.nan,
                    norm_type: int = 1,
                    max_rot: float = 10.
                    ) -> List:
    """
    Generates random angle_linspase

    :param arbitrary_increment: with this variable you can control
                                scaling of goal configuration according
                                to the initial configuration
    :param linspace_step: variable which denotes the size of linspace
    :param q_init: allows to manually setup initial state of system
    :param q_goal: allows to manually setup goal state of system
    :param norm_type: default is Manhattan norm
    :param mar_rot: maximum allowed rotation for each joint

    :return: List.
            a sequence of configurations connecting initial configuration
            and goal configuration
    """
    if np.isnan(q_init.all()):
        q_init = np.random.randint(0, 180, 4).astype(float)

    linspace_configs = np.array(q_init)
    linspace_configs = linspace_configs[None, :]

    if np.isnan(q_goal.all()):
        while True:
            q_goal = np.round((2)**(np.random.random()) * q_init)
            if (np.abs(q_goal) >= 0.0).all() and (np.abs(q_goal) <= 180.0
                break

    q_random = np.array(q_init)

    while np.linalg.norm((q_goal - q_random), norm_type) >= max_rot:
        q_random = np.random.randint(0, 180, 4).astype(float)

        if np.linalg.norm((q_random - linspace_configs[-1]), norm_type) >
            continue
        if (np.abs(au.angle_difference(q_random, linspace_configs[-1])) >
            continue
```

```
        linspace_configs  = np.vstack((linspace_configs, q_random))

    return [State(angles) for angles in linspace_configs]
```

In [ ]:
```
config_space = rrt_creation(10, 2,
                q_init=np.array([0., 0., 0., 0.]),
                q_goal=np.array([-180., -60., 72., -60.]))
for state in config_space:
    print(state.angles)
    env.state = state
    condition, point = env.check_collision(env.state)
    if (condition):
        print(
            f'Attention!!! Collision detected '
            f'in link {point['link_segment']} '
            f'joint on coordinates: {np.round(point['collision_point'], 2
        )
    else:
        print('No collision detected')
    env.render(plt_show=False)
```

Few functions implemented for usage in main algorithm

In [ ]:
```
def compute_weighted_distance(state_a, state_b):
    scaling_factors = np.array([1.2, 1.1, 1.05, 1.0])
    angle_diffs = au.angle_difference(state_a, state_b)
    absolute_diffs = np.abs(np.array(angle_diffs))
    weighted_diffs = scaling_factors * absolute_diffs
    total_distance = np.sum(weighted_diffs)
    return total_distance


def reconstruct_array_path(goal_node):
    """Reconstruct the path from the goal node to the root."""
    path = []
    current_node = goal_node
    while current_node is not None:
        path.append(current_node.value.tolist())
        current_node = current_node.parent
    return path[::-1]


def generate_random_state(q_goal, state_space_root, method=1,
                          min_bounds=-180, max_bounds=180):
    """
    Generate a random state (q_random) using one of the specified methods

    :param q_goal: Goal configuration as a numpy array.
    :param state_space_root: The root configuration (numpy array).
    :param method: Integer specifying the method to use:
                   1 - Global Sampling
                   2 - Goal Sampling
```

```
                     3 - Informed Sampling (Heuristic-Based)
    :param min_bounds: Minimum bounds for each dimension.
    :param max_bounds: Maximum bounds for each dimension.
    :return: A random state as a numpy array.
    """
    if method == 1:
        # Global Sampling
        return np.random.uniform(min_bounds, max_bounds, size=q_goal.shap

    elif method == 2:
        # Goal Sampling
        return np.random.uniform(
                np.min([state_space_root.value, q_goal], axis=0),
                np.max([state_space_root.value, q_goal], axis=0)
                )

    elif method == 3:
        # Informed Sampling (Heuristic-Based)
        distance_to_goal = np.linalg.norm(q_goal - state_space_root.value
        lower_bound = np.maximum(q_goal - distance_to_goal, min_bounds)
        upper_bound = np.minimum(q_goal + distance_to_goal, max_bounds)
        return np.random.uniform(lower_bound, upper_bound, size=q_goal.sh

    else:
        raise ValueError("Invalid method selected. Choose 1, 2, or 3.")
```

In [ ]:
```
class Node:
    def __init__(self, value, parent=None):
        self.value = value
        self.children = []
        self.parent = parent


    def add_child(self, child_node):
        """Add a child node to the current node."""
        self.children.append(child_node)


    def __repr__(self):
        return f"Node({self.value})"
```

In [ ]:
```
class Tree:
    def __init__(self, root_value):
        self.root = Node(root_value)


    def add_node(self, parent_value, child_value):
        """Add a child node under the specified parent node."""
        parent_node = self.find_node(self.root, parent_value)
        if parent_node:
            parent_node.add_child(Node(child_value))
```

```
    def find_node(self, current_node, value):
        """Recursively search for a node with the given value."""
        if np.array(current_node.value == value).all():
            return current_node
        for child in current_node.children:
            found_node = self.find_node(child, value)
            if found_node:
                return found_node
        return None


    def find_child(self, current_node):
        return True


    def find_neighbor(
        self, current_node,
        value, boundary=0.5,
        norm_type=1) -> bool:
        """Find and connect a neighbor within a specified boundary."""
        if np.linalg.norm(np.array([current_node.value - value]), ord=nor
            new_node = Node(value, parent=current_node)
            current_node.add_child(new_node)
            return True

        for child in current_node.children:
            found_neighbor = self.find_neighbor(child, value, boundary)
            if found_neighbor:
                return found_neighbor
        return False


    def display(self, node=None, level=0):
        """Display the tree structure."""
        if node is None:
            node = self.root
        print(' ' * level * 4 + str(node.value))
        for child in node.children:
            self.display(child, level + 1)
```

In [ ]:

```
class Tree:
    def __init__(self, root_value):
        self.root = Node(root_value)  # Initialize the tree with a root n

    def find_node(self, current_node, value):
        """Recursively search for a node with the given value."""
        if np.array_equal(current_node.value, value):
            return current_node
        for child in current_node.children:
            found_node = self.find_node(child, value)
            if found_node:
                return found_node
```

```
        return None

    def find_neighbor(
        self, current_node,
        value, boundary=0.5,
        norm_type=1,
        custom_norm:bool = False) -> bool:
        """Find and connect a neighbor within a specified boundary."""
        if custom_norm:
            if compute_weighted_distance(current_node.value, value) <= bo
                new_node = Node(value, parent=current_node)
                current_node.add_child(new_node)
                return True
        else:
            if np.linalg.norm(np.array([current_node.value - value]), ord
                new_node = Node(value, parent=current_node)
                current_node.add_child(new_node)
                return True

        for child in current_node.children:
            found_neighbor = self.find_neighbor(child, value, boundary, n
            if found_neighbor:
                return found_neighbor
        return False

    def find_all_leaf_nodes(self, current_node=None, leaves=None):
        """
        Find all leaf nodes (nodes without children) in the tree.
        """
        if leaves is None:
            leaves = []
        if current_node is None:
            current_node = self.root

        if not current_node.children:  # If no children, it's a leaf
            leaves.append(current_node)
        else:
            for child in current_node.children:
                self.find_all_leaf_nodes(child, leaves)
        return leaves

    def add_to_nearest_leaf(
            self, q_random, boundary,
            env: ManipulatorEnv, norm_type=1):
        """
        Add a new node to the nearest leaf node in the direction of q_ran

        :param q_random: Target state (numpy array).
        :param boundary: Maximum allowable distance for the new node.
        :param norm_type: Norm type to use for distance calculation.
        """
        # Find all leaf nodes
        leaf_nodes = self.find_all_leaf_nodes()
```

```
        # Find the nearest leaf to q_random using the specified norm
        nearest_leaf = None
        min_distance = float("inf")
        for leaf in leaf_nodes:
            distance = np.linalg.norm(q_random - leaf.value, ord=norm_typ
            if distance < min_distance:
                min_distance = distance
                nearest_leaf = leaf

        # Steer from the nearest leaf toward q_random, constrained by bou
        if nearest_leaf is not None:
            diff = q_random - nearest_leaf.value
            step = np.clip(diff, -boundary, boundary)  # Limit the step s
            new_state = nearest_leaf.value + step

            #collision check
            env.state = State(new_state)
            condition, _ = env.check_collision(env.state)
            if condition:
                return None

            new_node = Node(new_state, parent=nearest_leaf)
            nearest_leaf.add_child(new_node)
            return new_node
        return None
```

RRT algorithm **without collision**

In [ ]:

```
leaf_init = np.array([0., 0., 0., 0.])
# leaf_goal = np.array([-180., -60., 72., -60.])
leaf_goal = np.array([10., 11., 12., 10.])

state_tree = Tree(leaf_init)

def rrt_array_algos(state_space: Tree, q_goal: np.ndarray, bound=1):

    while state_space.find_neighbor(state_space.root, q_goal, boundary=bo
        # q_random = np.random.uniform(
        # np.min([state_space.root.value, q_goal], axis=0),
        # np.max([state_space.root.value, q_goal], axis=0)
        # )
        q_random = generate_random_state(q_goal=q_goal,
                                        state_space_root=state_space.root
                                        method=2)

        # ic(q_random)
        state_space.find_neighbor(state_space.root, np.round(q_random, 2)
        # state_space.display()

    # state_space.display()

    # Find the goal node
```

```
    goal_node = state_space.find_node(state_space.root, q_goal)
    if goal_node:
        # Reconstruct the path
        path = reconstruct_array_path(goal_node)
        print(f"Shortest path: {path}")
    else:
        print("Goal node not found!")

    return path


linspace_final = rrt_array_algos(state_tree, leaf_goal, bound=2)
```

In [ ]:
```
linspace_final = np.array(linspace_final)

for state in [State(angles) for angles in linspace_final]:
    print(state.angles)
    env.state = state
    condition, point = env.check_collision(env.state)
    if (condition):
        print(
            f'Attention!!! Collision detected '
            f'in link {point['link_segment']} '
            f'joint on coordinates: {np.round(point['collision_point'], 2
        )
    else:
        print('No collision detected')
    env.render(plt_show=False)
```

Main implementation

In [ ]:
```
def rrt_array_algos_with_collision(
    state_space: Tree,
    q_goal: np.ndarray,
    obstacles: np.ndarray,
    collision_threshold: float = 0.1,
    node_bound=1,
    max_iterations=30000
):
    """
    RRT algorithm with collision avoidance and fallback to add nodes to t

    :param state_space: The Tree object representing the state space.
    :param q_goal: The goal configuration as a numpy array.
    :param obstacles: Array representing obstacles in the environment.
    :param collision_threshold: Threshold for collision detection.
    :param node_bound: Maximum allowable distance for connecting nodes.
    :param max_iterations: Maximum iterations to grow the tree.
    :return: The path from start to goal as a list of configurations.
    """
    skip = 0
    test_env = ManipulatorEnv(
        obstacles=obstacles,
```

```
        initial_state=State(state_space.root.value),
        collision_threshold=collision_threshold
    )

    for iteration in range(max_iterations):
        # Check if we can directly connect to the goal
        if state_space.find_neighbor(state_space.root, q_goal, boundary=n
            print(f"Goal reached in {iteration + 1} iterations.")
            break

        # Generate a random configuration
        q_random = generate_random_state(
            q_goal=q_goal,
            state_space_root=state_space.root,
            method=1  # Global sampling
        )

        # Perform collision check
        test_env.state = State(q_random)
        condition, point = test_env.check_collision(test_env.state)
        if condition:
            skip += 1
            print(f"Skipped {skip} due to collision at {q_random}.")
            continue

        # Try to add the random configuration to the tree
        added = state_space.find_neighbor(state_space.root, np.round(q_ra
        if not added:
            # Fallback: Add a new node to the nearest leaf in the tree
            new_node = state_space.add_to_nearest_leaf(
                q_random, boundary=node_bound,
                env=test_env, norm_type=1
            )
            if new_node:
                print(f"Node added to the nearest leaf: {new_node.value}"

    # Find the goal node
    goal_node = state_space.find_node(state_space.root, q_goal)
    if goal_node:
        # Reconstruct the path
        path = reconstruct_array_path(goal_node)
        print(f"Shortest path: {path}")
        return path
    else:
        # print("Goal node not found!")
        return None

# Example input data
leaf_init = np.array([0.0, 0.0, 0.0, 0.0])
leaf_goal = np.array([-180.0, -60.0, 72.0, -60.0])

state_tree = Tree(leaf_init)
```

```
linspace_final_without_collision = rrt_array_algos_with_collision(
    state_space=state_tree,
    q_goal=leaf_goal,
    obstacles=np.array(data["obstacles"]),
    collision_threshold=data["collision_threshold"],
    node_bound=15
)
```

In [ ]:
```
linspace_final = np.array(linspace_final_without_collision)

for state in [State(angles) for angles in linspace_final]:
    print(state.angles)
    env.state = state
    condition, point = env.check_collision(env.state)
    if (condition):
        print(
            f'Attention!!! Collision detected '
            f'in link {point['link_segment']} '
            f'joint on coordinates: {np.round(point['collision_point'], 2
        )
    else:
        print('No collision detected')
    env.render(plt_show=False)
```

In [ ]:
```
from anim import animate_plan

animate_plan(env, [State(angles) for angles in linspace_final])
```

Task 2.C

On average, for our initial and goal points, RRT takes 2-4 minutes and provide 130-150 states trajectory.

But, to achieve this, it was required a lot spaces, based on log we can say that it was skipped at lease 21600 states due to collision. It's a lot.

RRT doesn't guaranty that result will be optimal plan. Moreover, it's more chance that result will be not optimal. But, in my own opinion, RRT also is not good idea for multi-state agents. Because on sampling step it leads to complicated check-conditions and looking for neighbors. It something like "we have a lot computational resource - let's use it on maximum". Also, very important for manipulators and robotics in global, if we can provide robust, repeatable algorithm. And I don't found RRT such algorithm as well.

Task 2.D

With the introduction of the weighted distance function that emphasizes the angles of the first joints, the planner now prioritizes optimizing the motion of these initial joints in the manipulator. This change in focus alters the trajectory structure, making the movements of the first joints more predictable, while potentially leading to less optimal movements in the more distant joints, which carry less weight in the calculations.

From a practical standpoint, employing a weighted metric allows for consideration of the varying significance of different joints for specific tasks. However, this approach may compromise the overall optimality of the trajectory. This rephrasing maintains the original meaning while presenting it in a new way.

In [ ]:
```
def rrt_array_algos_with_collision(
    state_space: Tree,
    q_goal: np.ndarray,
    obstacles: np.ndarray,
    collision_threshold: float = 0.1,
    node_bound=1,
    max_iterations=30000
):
    """
    RRT algorithm with collision avoidance and fallback to add nodes to t

    :param state_space: The Tree object representing the state space.
    :param q_goal: The goal configuration as a numpy array.
    :param obstacles: Array representing obstacles in the environment.
    :param collision_threshold: Threshold for collision detection.
    :param node_bound: Maximum allowable distance for connecting nodes.
    :param max_iterations: Maximum iterations to grow the tree.
    :return: The path from start to goal as a list of configurations.
    """
    skip = 0
    test_env = ManipulatorEnv(
        obstacles=obstacles,
        initial_state=State(state_space.root.value),
        collision_threshold=collision_threshold
    )

    for iteration in range(max_iterations):
        # Check if we can directly connect to the goal
        if state_space.find_neighbor(state_space.root,
                                     q_goal,
                                     boundary=node_bound,
                                     custom_norm=True):
            print(f"Goal reached in {iteration + 1} iterations.")
            break

        # Generate a random configuration
        q_random = generate_random_state(
            q_goal=q_goal,
            state_space_root=state_space.root,
            method=1  # Global sampling
        )
```

```
        # Perform collision check
        test_env.state = State(q_random)
        condition, point = test_env.check_collision(test_env.state)
        if condition:
            skip += 1
            print(f"Skipped {skip} due to collision at {q_random}.")
            continue

        # Try to add the random configuration to the tree
        added = state_space.find_neighbor(state_space.root,
                                          np.round(q_random, 2),
                                          boundary=node_bound,
                                          custom_norm=True)
        if not added:
            # Fallback: Add a new node to the nearest leaf in the tree
            new_node = state_space.add_to_nearest_leaf(
                q_random, boundary=node_bound,
                env=test_env, norm_type=1
            )
            if new_node:
                print(f"Node added to the nearest leaf: {new_node.value}"

    # Find the goal node
    goal_node = state_space.find_node(state_space.root, q_goal)
    if goal_node:
        # Reconstruct the path
        path = reconstruct_array_path(goal_node)
        print(f"Shortest path: {path}")
        return path
    else:
        # print("Goal node not found!")
        return None

# Example input data
leaf_init = np.array([0.0, 0.0, 0.0, 0.0])
leaf_goal = np.array([-180.0, -60.0, 72.0, -60.0])

state_tree = Tree(leaf_init)

linspace_final_without_collision = rrt_array_algos_with_collision(
    state_space=state_tree,
    q_goal=leaf_goal,
    obstacles=np.array(data["obstacles"]),
    collision_threshold=data["collision_threshold"],
    node_bound=20
)
```

In [ ]:
```
linspace_final = np.array(linspace_final_without_collision)

for state in [State(angles) for angles in linspace_final]:
    print(state.angles)
    env.state = state
```

```
    condition, point = env.check_collision(env.state)
    if (condition):
        print(
            f'Attention!!! Collision detected '
            f'in link {point['link_segment']} '
            f'joint on coordinates: {np.round(point['collision_point'], 2
        )
    else:
        print('No collision detected')
    env.render(plt_show=False)
```

In [ ]:
```
animate_plan(env, [State(angles) for angles in linspace_final], video_out
```

TASK 2.E

Increasing the RRT step size led to a decrease in both computation time and the number of trajectory points, as the planner navigates the space more assertively, which in turn minimizes the number of intermediate configurations. However, a larger step size may overlook critical configurations, particularly in intricate areas with tight passages between obstacles. This can heighten the risk of errors, such as crossing into obstacles, due to a lack of detail in the trajectory. Generally, a larger pitch is advantageous for simpler environments, but it necessitates careful consideration when dealing with dense or confined spaces. Striking a balance between pitch and safety is essential.

In [ ]:
```
# Example input data
leaf_init = np.array([0.0, 0.0, 0.0, 0.0])
leaf_goal = np.array([-180.0, -60.0, 72.0, -60.0])

state_tree = Tree(leaf_init)

linspace_final_without_collision = rrt_array_algos_with_collision(
    state_space=state_tree,
    q_goal=leaf_goal,
    obstacles=np.array(data["obstacles"]),
    collision_threshold=data["collision_threshold"],
    node_bound=30
)
```

In [ ]:
```
linspace_final = np.array(linspace_final_without_collision)

for state in [State(angles) for angles in linspace_final]:
    print(state.angles)
    env.state = state
    condition, point = env.check_collision(env.state)
    if (condition):
        print(
            f'Attention!!! Collision detected '
            f'in link {point['link_segment']} '
```

```
            f'joint on coordinates: {np.round(point['collision_point'], 2
        )
    else:
        print('No collision detected')
    env.render(plt_show=False)
```