Assignment 2: Report

Benyamin and Sebastian

November 2023

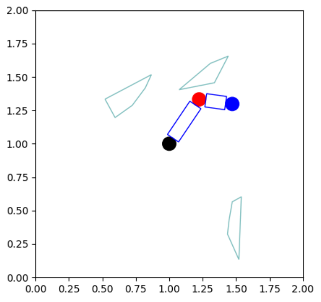
1 Motion Planning for a 2-link Planar arm

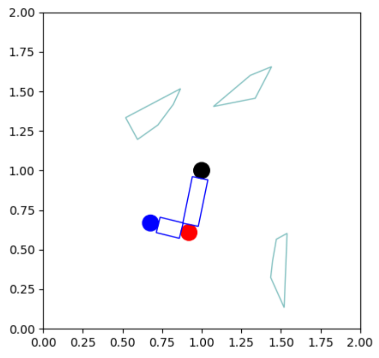
1.1 Sampling random collision-free configurations

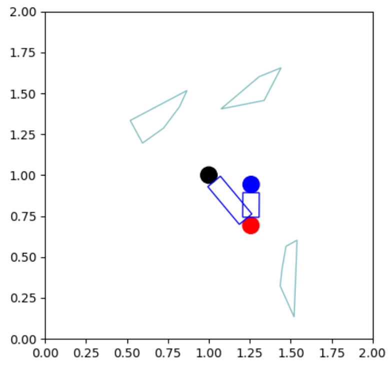
For your report randomly generate 5 collision-free configurations in this environment and include the corresponding visualizations in your report. Do so as well for one more environment from those that you generated as part of assignment. Briefly explain your implementation.

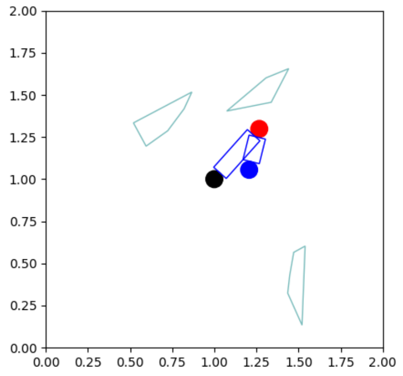
Really, there was only one major difference between my old code and this one, and that was randomly generating the arm in different configurations. So what I did was simply adding random.uniform(0, 360) to the angle, and if this caused a collision, I kept retrying it indefinitely.

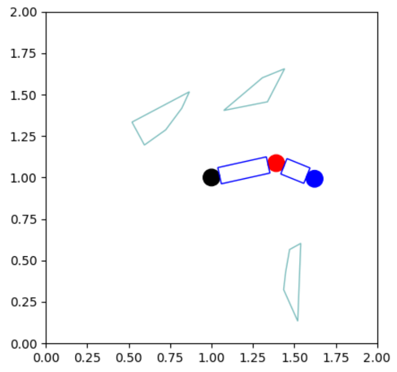
ARMPOLYGON.NPY EXAMPLES



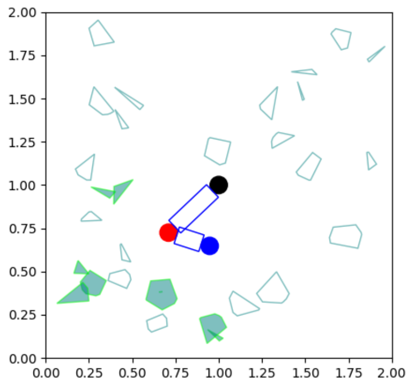


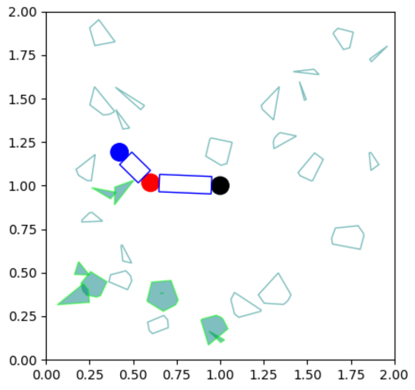


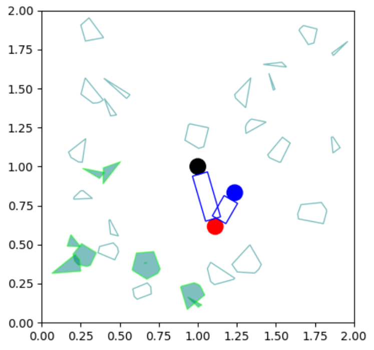


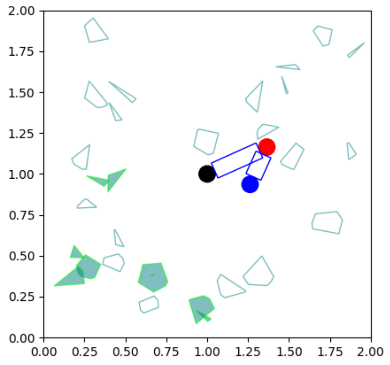


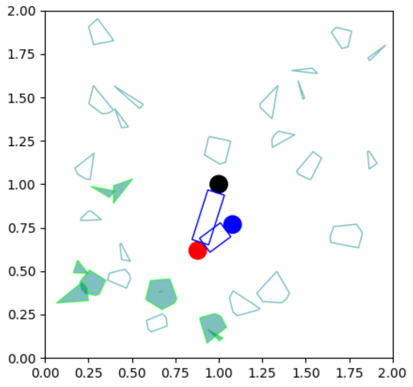
TEST1.NPY EXAMPLES



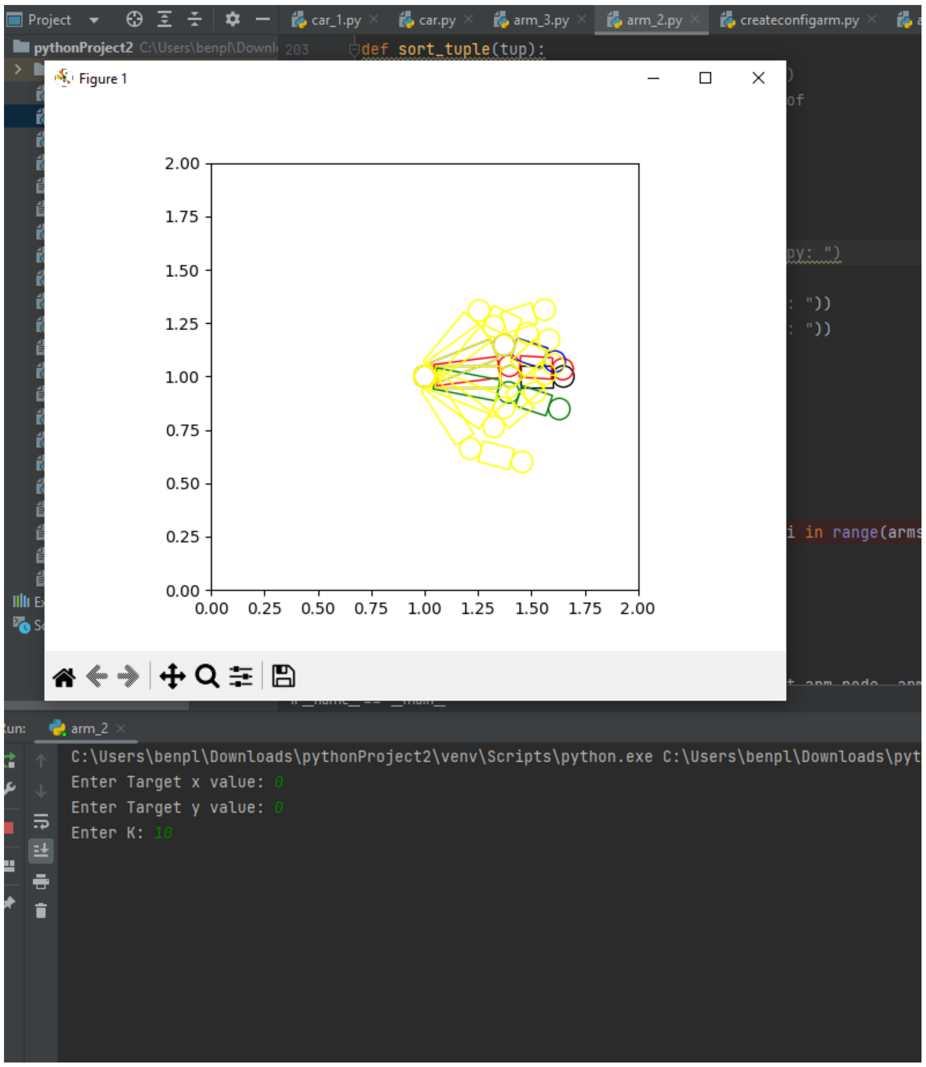






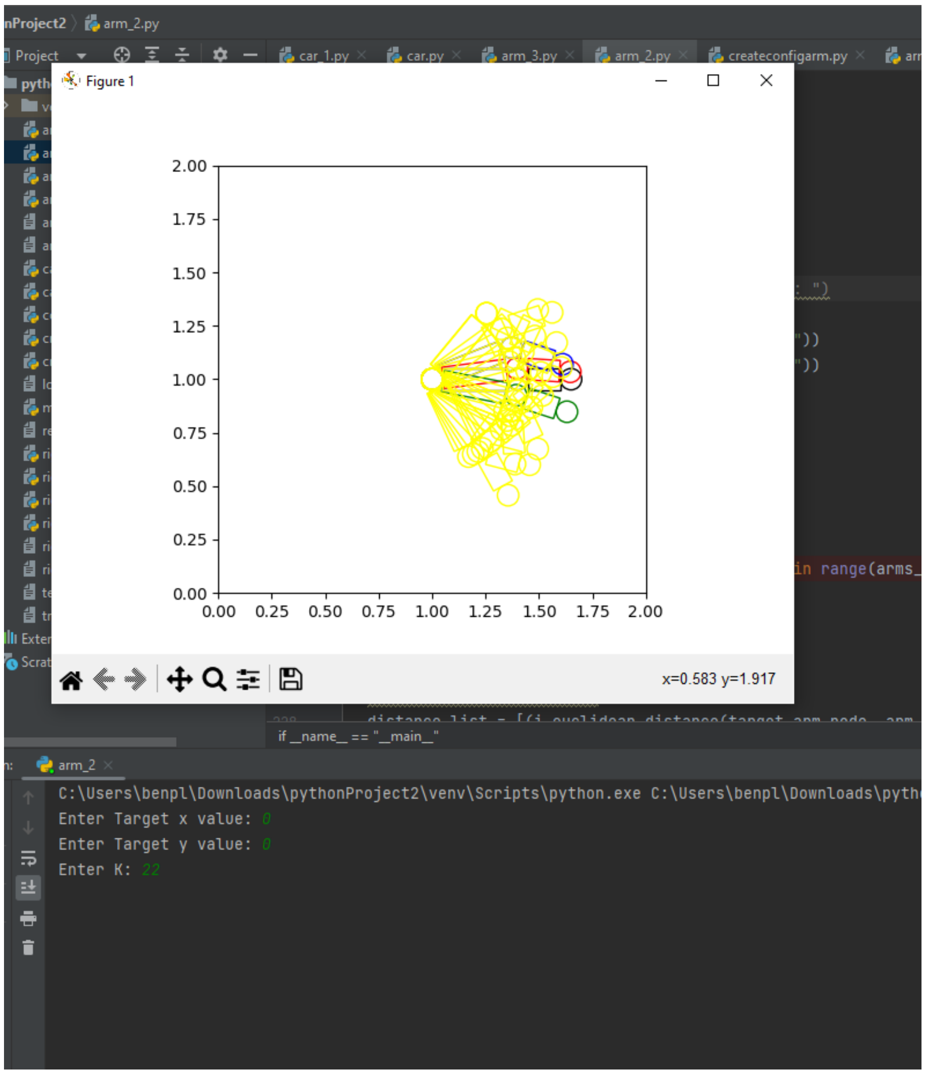


1.2 Nearest neighbors with linear search approach

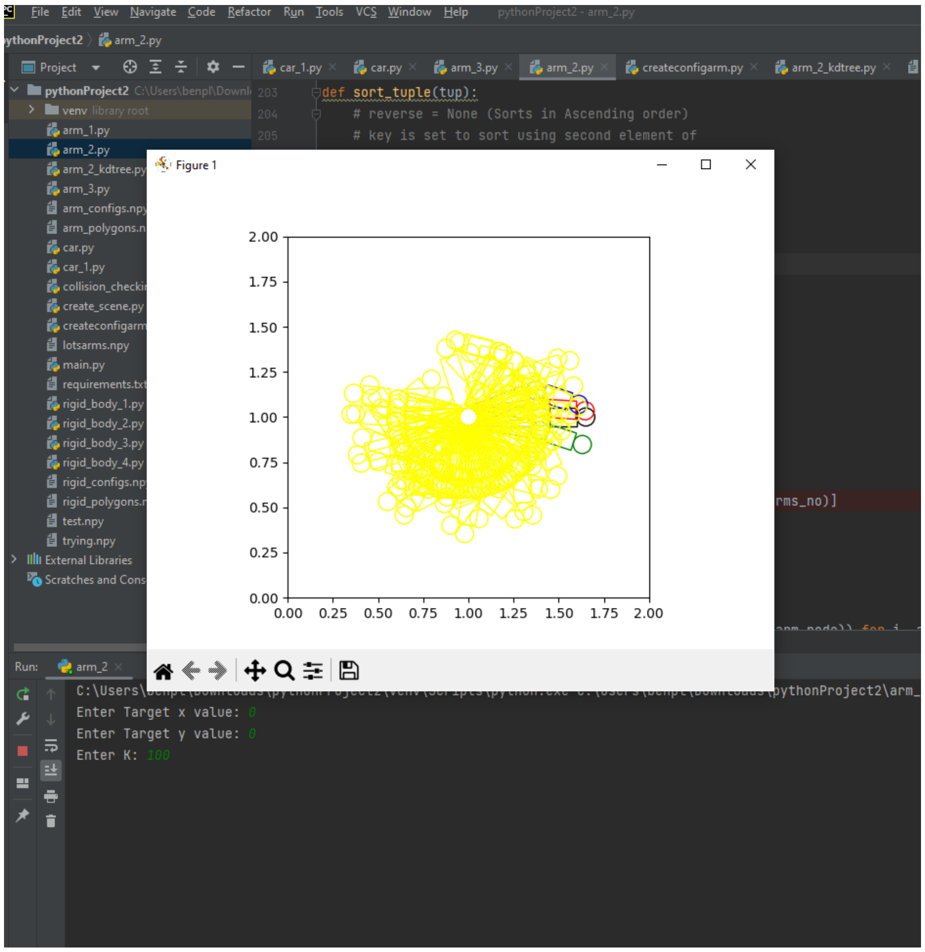


```
def sort_tuple(tup):
```

```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads\pyt
Enter Target x value: 0
Enter Target y value: 0
Enter K: 10
```



```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads\pyth
Enter Target x value: 0
Enter Target y value: 0
Enter K: 22
```

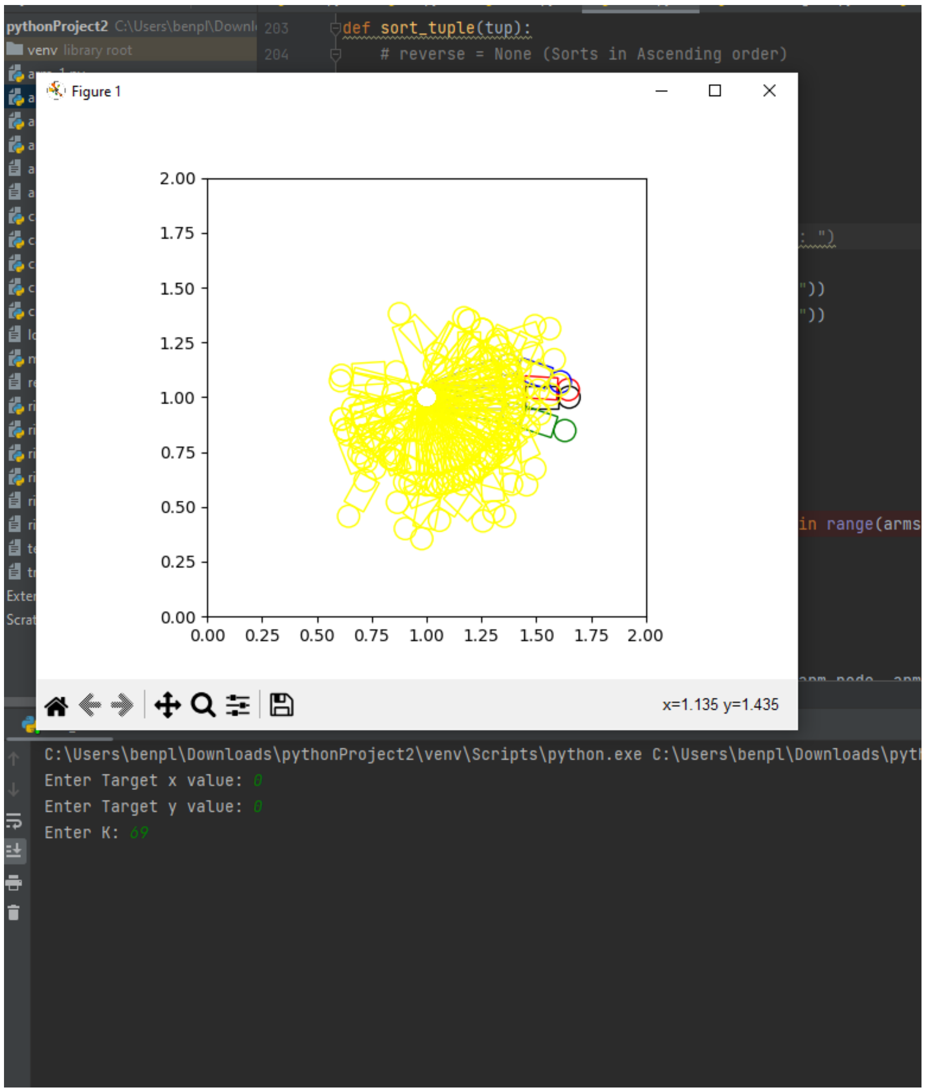


```
def sort_tuple(tup):
    # reverse = None (Sorts in Ascending order)
    # key is set to sort using second element of
```

```
rms_no)]
```

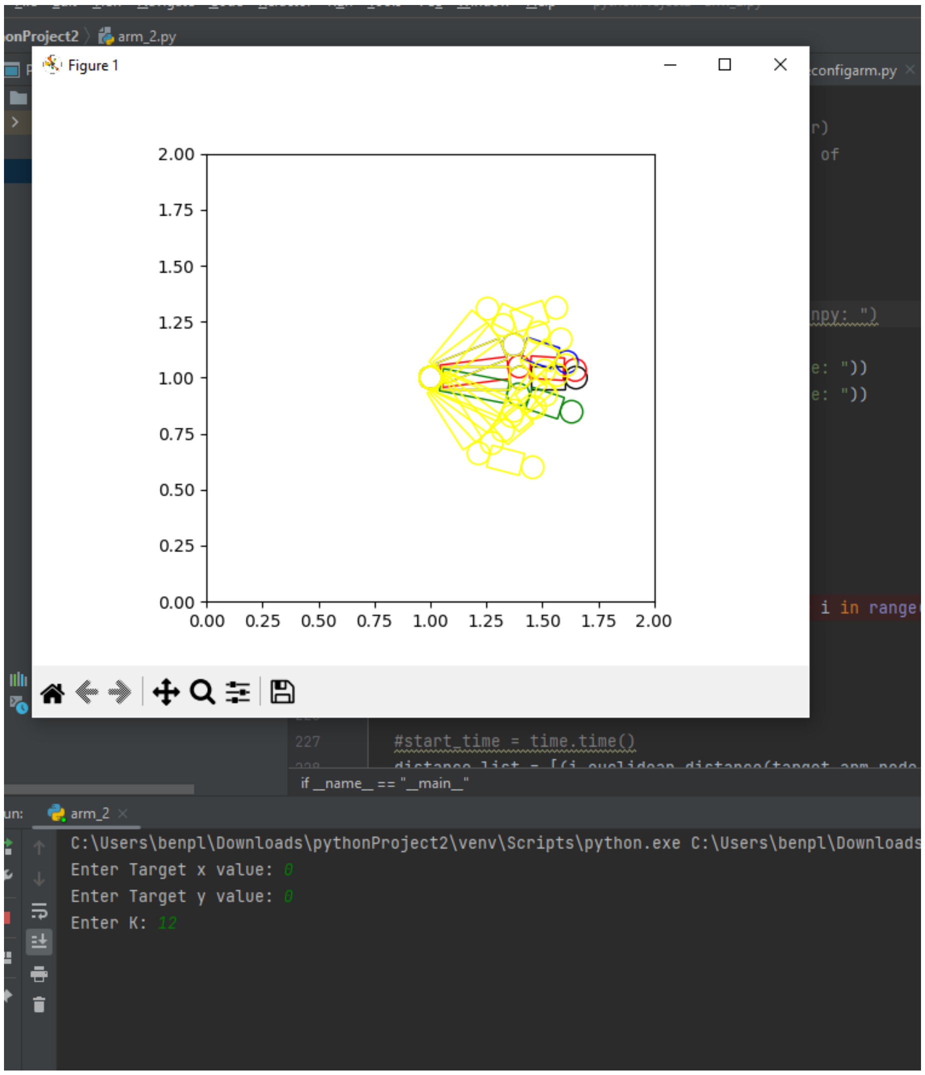
```
arm_node)) for i
```

```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads\pythonProject2\arm_
Enter Target x value: 0
Enter Target y value: 0
Enter K: 100
```



```
def sort_tuple(tup):
    # reverse = None (Sorts in Ascending order)
```

```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads\pyt
Enter Target x value: 0
Enter Target y value: 0
Enter K: 69
```



```
#start_time = time.time()
```

```
if __name__ == "__main__"
```

```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads
Enter Target x value: 0
Enter Target y value: 0
Enter K: 12
```

My code uses inputs instead of arguments for this problem due to not knowing how to parse arguments at that time. Later on I figured it out.

Introduction: The objective of this program is to identify and visualize the three nearest robot configurations to a specified target configuration. Robot configurations are defined by joint angles, and the process involves computing the Euclidean distance between the target and all configurations. The implementation adopts a naive linear search approach to ascertain the nearest neighbors.

Distance Metric: The utilized distance metric is the Euclidean distance between two configurations with angle subtraction for topology.

Robot Configuration Space: The robot is modeled as a series of connected joints, and its configuration space is defined by the potential combinations of joint angles. Joint angles are expressed in degrees, and the configurations are stored in a 2D matrix where each row corresponds to a distinct configuration.

Implementation: The program reads robot configurations from a file, creates instances of the robot arm class for each configuration, and computes the Euclidean distance between the target configuration and all stored configurations. The three nearest neighbors are identified, and the robot at the target configuration and its neighbors are visualized using Matplotlib.

1.3 Interpolation along the straight line in the C-space

As requested, attached in the zip files are three .mp4 animations that generate a plot, animating the robot arm moving from the start to the goal configuration step by step.

The interpolation resolution used in all three animations is determined by the variable numsteps, which is set to 100. Inputting a higher number of steps would make the animation appear smoother, as there would be more intermediate frames showing the arm's movement.

All three animations take different configurations as arguments in the settings of the file.

- The animation `arm_1.3-1.mp4` takes `[-0.5,-0.5]` as the start and `[1.5,1.5]` as the goal.
- The animation `arm_1.3-2.mp4` takes `[-0.5,-2.5]` as the start and `[2.5,1.5]` as the goal.
- The animation `arm_1.3-3.mp4` takes `[-0.5,-2.5]` as the start and `[2.5,-0.3]` as the goal.

1.4 Implement RRT

As requested, attached in the zip files is the .mp4 animation that generate a plot animating the arm moving along the solution path inside the original workspace from the start and the configuration to the goal configuration without colliding with any obstacles. In the last part of the same .mp4 video is also the animation that shows the tree data structured generated by RRT growing over the search process in the arm's configuration space.

In our example, arguments such as: "–start 0 0 –goal -2.9 0 –map "arm-polygons.npy" " were inputted thought the file settings of the script rather the command prompt.

- The animation `"1.4 all animations under 1 minute.mp4"` takes `[0,0]` as the start and `[-2.9,0]` as the goal.

Introduction:
The task was to implement the Rapidly-exploring Random Tree (RRT) algorithm to find a collision-free path for a robotic arm within a given 2D configuration space. The configuration space, defined by the 'arm_polygons.npy' file, contains obstacles that the arm must navigate around.

RRT Implementation:
The RRT algorithm was implemented to grow a tree from the start configuration towards the goal. Nodes were added iteratively by selecting random points in the configuration space and connecting them to the nearest existing node if no collision occurs along the connecting path. A step size of 0.3 units was used for each extension, and the goal was sampled occasionally to guide the search.

Collision Checking:
Collision checking was performed using a 'check_joint' function that tested if the arm's joints or connecting links intersected with any obstacles. The collision check was integrated into the tree expansion to ensure that only valid configurations were added to the tree.

Tree Visualization:
An animation was generated to show the growth of the RRT. Each node and its connection to the parent node were visualized with green lines, and the goal configuration was marked with a red point. The visualization highlighted the exploration process of the algorithm within the arm's configuration space.

Path Animation:
Upon finding a path, a separate animation was created to display the arm's movement from the start to the goal configuration. The arm's configuration was updated frame by frame to follow the solution path, ensuring no collisions occurred with the environment.

Conclusion:
The RRT algorithm successfully generated a collision-free path within the provided configuration space. The implementation respected the topology of the space and treated the boundaries correctly. The animations provided a clear visualization of the algorithm's process and the final solution path.

Figures:
1. Animation of the RRT tree growth.
2. Animation of the arm following the solution path.

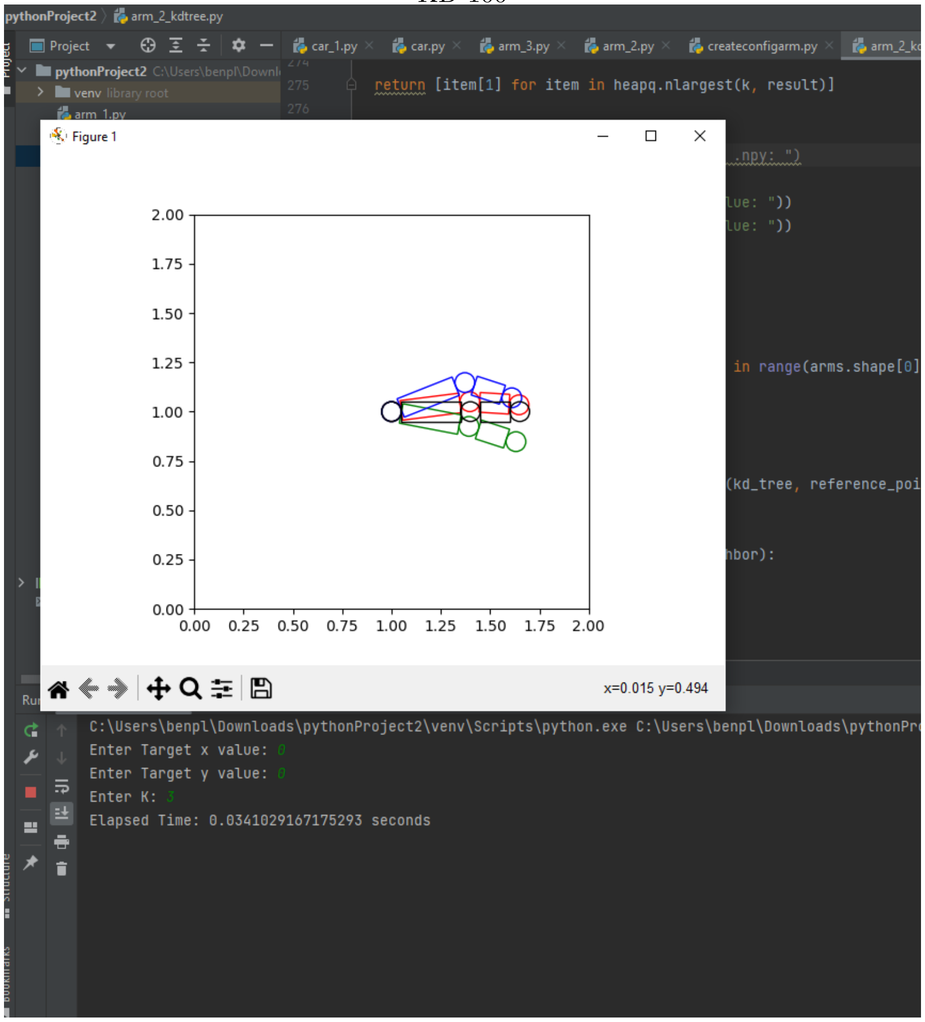
Extra Credit: KD-Tree implementation and Comparison

of Nearest Neighbor Methods

To begin with, I'll report my conclusions on the statistics. The Linear search approach takes longer to run as the number of configurations in a file increases. Specifically, Linear search took 0.0269 seconds for 100 configurations and 0.030917 seconds for 1000 configurations.
On the other hand, the KD tree approach appears to be less affected by an increase in the number of configurations.
With 100 configurations, it ran in 0.0341 seconds, and with 1000 configurations, it ran in 0.0349 seconds.This is notably less than a second, in contrast to Linear search, which took an additional 3 seconds to process 1000 configurations.

Configuration of 100 arms

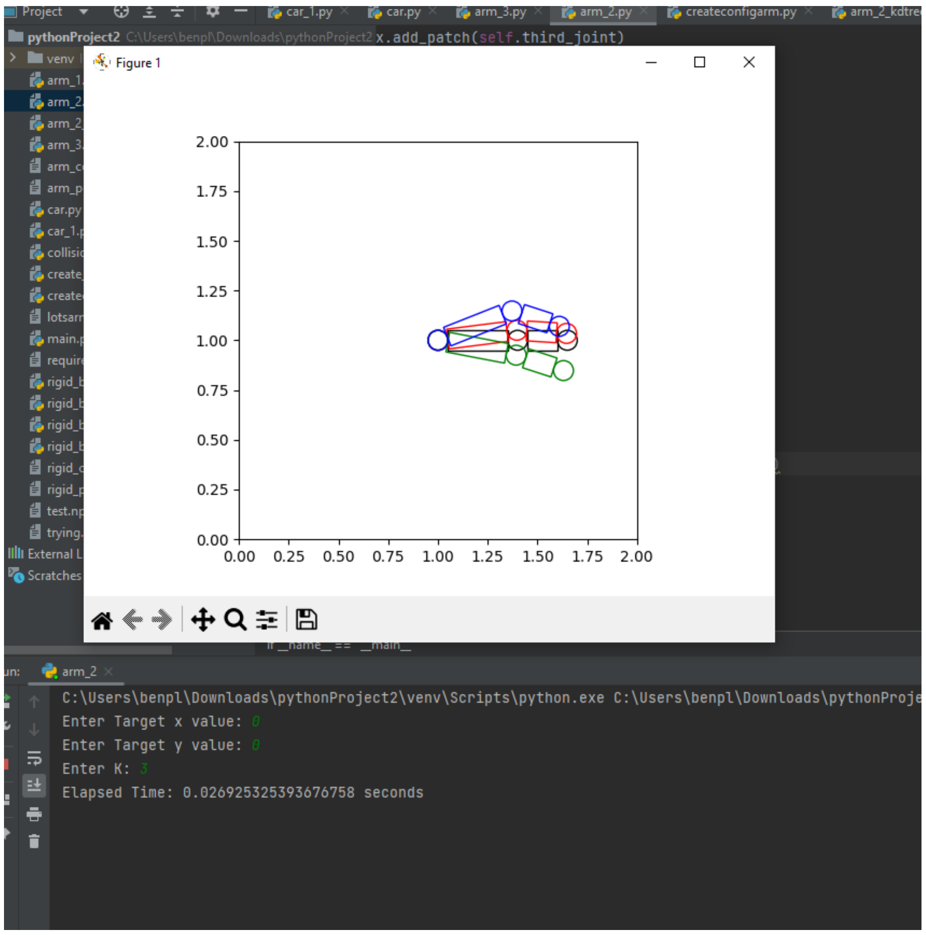
KD 100



```
return [item[1] for item in heapq.nlargest(k, result)]
```

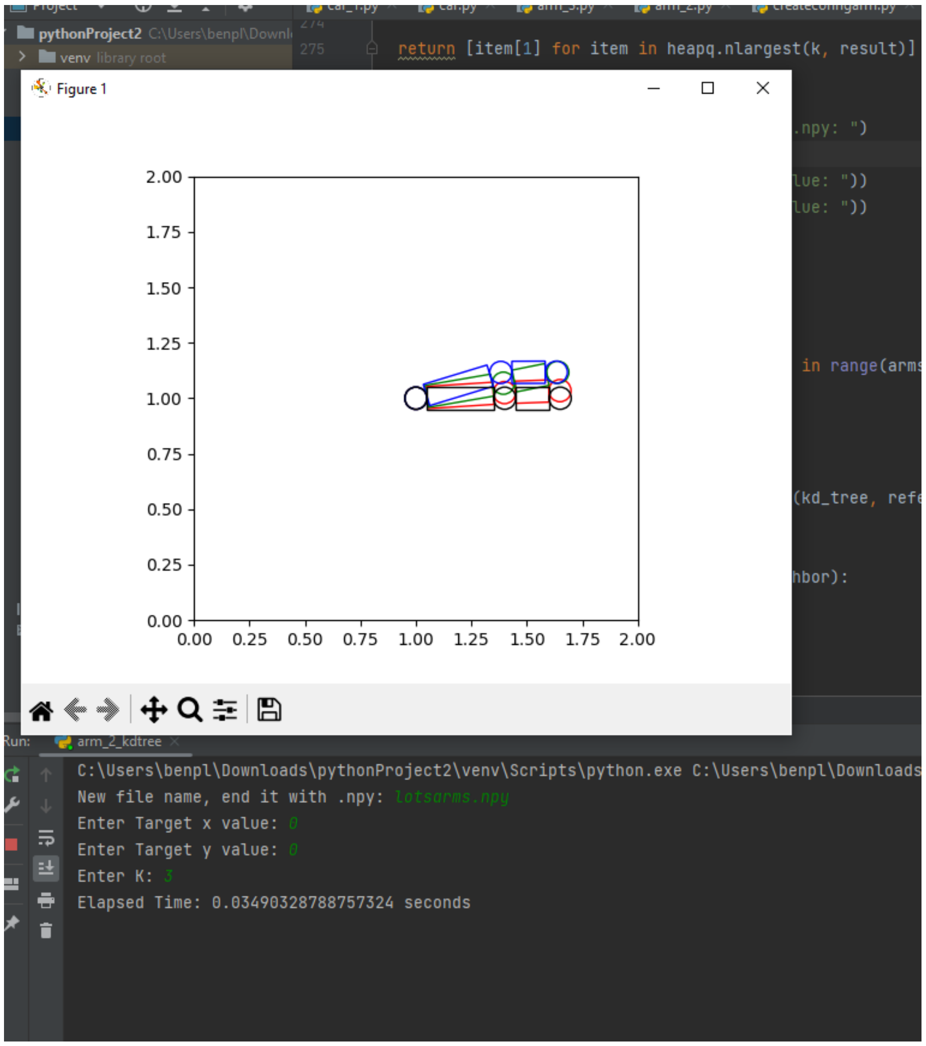
```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads\pythonPr
Enter Target x value: 0
Enter Target y value: 0
Enter K: 3
Elapsed Time: 0.0341029167175293 seconds
```

Linear search approach 100



```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads\pythonProje
Enter Target x value: 0
Enter Target y value: 0
Enter K: 3
Elapsed Time: 0.026925325393676758 seconds
```

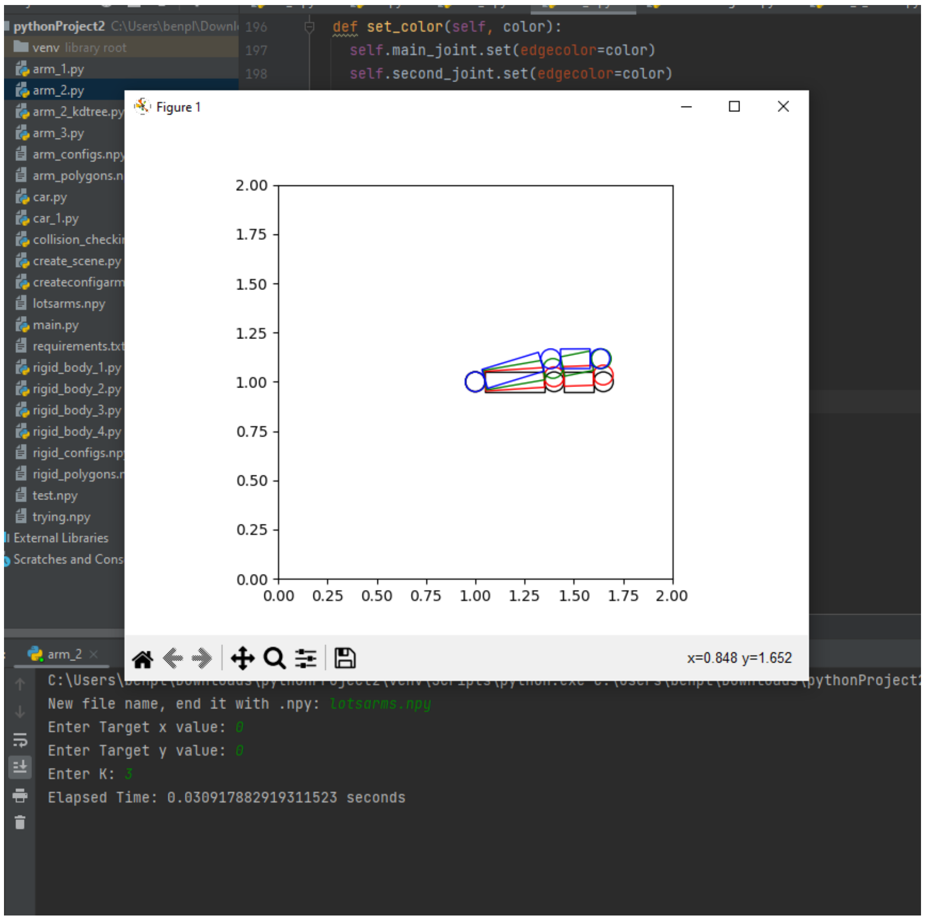
configuration of 1000 arms
KD 1000



```
return [item[1] for item in heapq.nlargest(k, result)]
```

```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads
New file name, end it with .npy: lotsarms.npy
Enter Target x value: 0
Enter Target y value: 0
Enter K: 3
Elapsed Time: 0.03490328788757324 seconds
```

Linear search approach 1000



```python
def set_color(self, color):
    self.main_joint.set(edgecolor=color)
    self.second_joint.set(edgecolor=color)
```

```
C:\Users\benpl\Downloads\pythonProject2\venv\Scripts\python.exe C:\Users\benpl\Downloads\pythonProject2
New file name, end it with .npy: lotsarms.npy
Enter Target x value: 0
Enter Target y value: 0
Enter K: 3
Elapsed Time: 0.030917882919311523 seconds
```

Custom Distance Metric:
A critical aspect of the implementation is the design of a custom distance metric suitable for joint angles. The implemented metric calculates the angular difference between two sets of joint angles, reflecting the true dissimilarity between arm configurations.
The metric is designed to handle the periodicity of joint angles, ensuring accurate representation of distances in the non-Euclidean space.

KD-Tree Construction:
The KD-Tree construction process is adapted to accommodate the unique characteristics of joint angles. A recursive algorithm is employed, splitting the data along alternating dimensions at each level.
The KD-Tree captures the cyclic nature of joint angles, providing an efficient hierarchical structure for subsequent nearest neighbor searches.

Nearest Neighbor Search:
During the search for nearest neighbors, a priority function is utilized that

respects the non-Euclidean nature of the joint space. The priority function is tailored to the custom distance metric, ensuring that the KD-Tree accurately reflects the proximity of different joint configurations. The search algorithm explores the tree efficiently, taking advantage of the specialized distance metric to identify the nearest neighbors.

Results and Validation:
The implementation is validated using a dataset of 2-link planar arm configurations. Comparative analysis against a linear search demonstrates the efficiency gains achieved through the KD-Tree approach. The KD-Tree is particularly advantageous in scenarios involving large datasets, where the logarithmic time complexity of search operations significantly outperforms linear search.

Challenges and Considerations:
Adapting the KD-Tree to a non-Euclidean topology presented challenges in designing a distance metric that both captures the nuances of joint angles and aligns with the principles of KD-Tree construction. Balancing the trade-off between accuracy and efficiency in the distance metric was crucial, requiring iterative adjustments and fine-tuning.

Conclusion:
The customized KD-Tree implementation serves as a robust solution for optimizing nearest neighbor searches in the non-Euclidean joint space of a 2-link planar arm. The successful adaptation of the KD-Tree to handle cyclic joint angles demonstrates its potential impact on enhancing robotic motion planning and control.

---

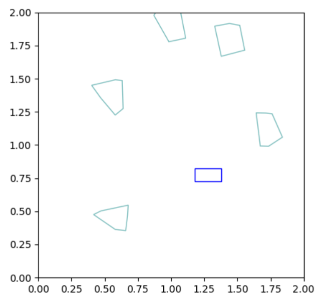
2 Motion planning for a rigid body in 2D

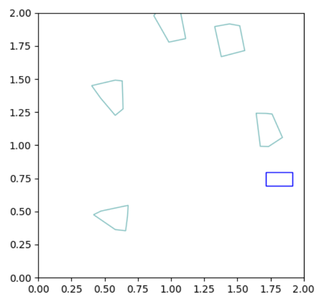
2.1 Sampling random collision-free configurations

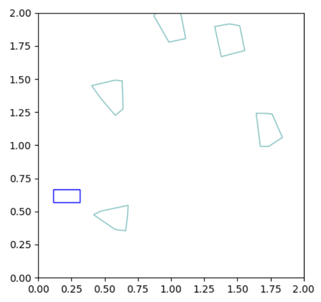
Just like with 1.1, here are various examples of how the program generates a plot with a random collision-free configuration of the 2d rigid body(car).
Just like 1.1, random.uniform is used but this time the it is used to get a random y and x value that are within the scope of the visible plot. If the position is one that causes a collision, then random.uniform is applied until a position that is free of collision is found.
For this section, the .npy files can be inputted thought the the command prompt. Below are two examples: rigid polygons.npy and trying.npy.

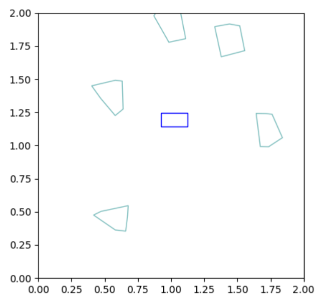
---

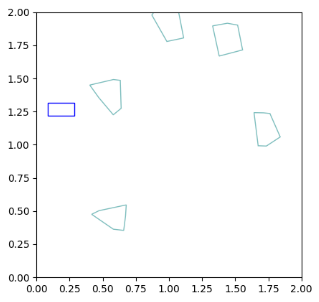
rigid polygons.npy EXAMPLES


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00

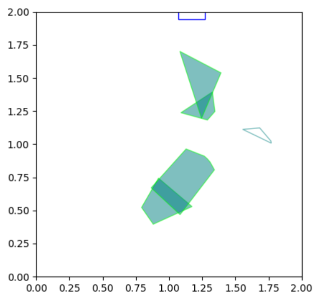

2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00

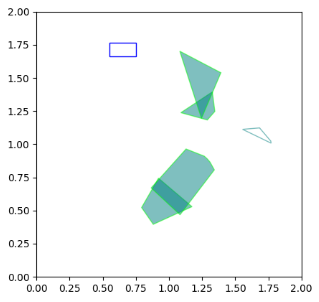

2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00

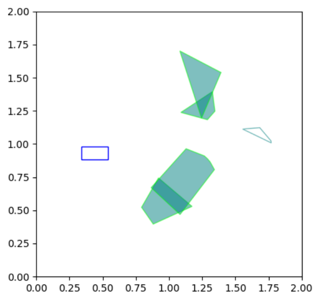


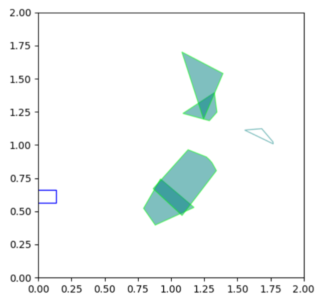
---

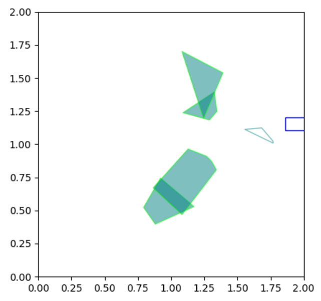
trying.npy EXAMPLES





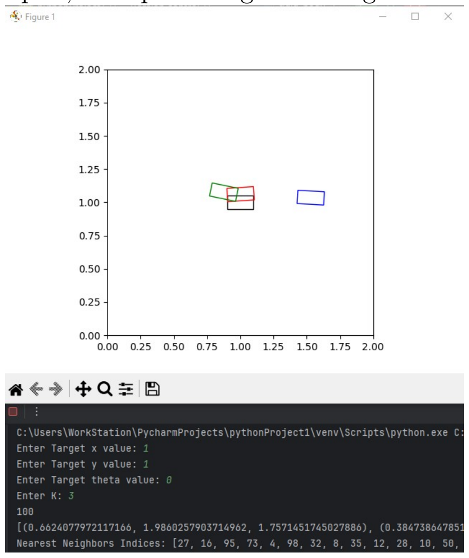




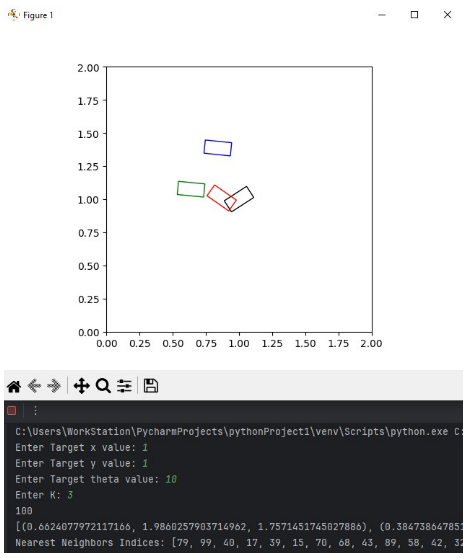


2.2 Nearest neighbors with linear search approach

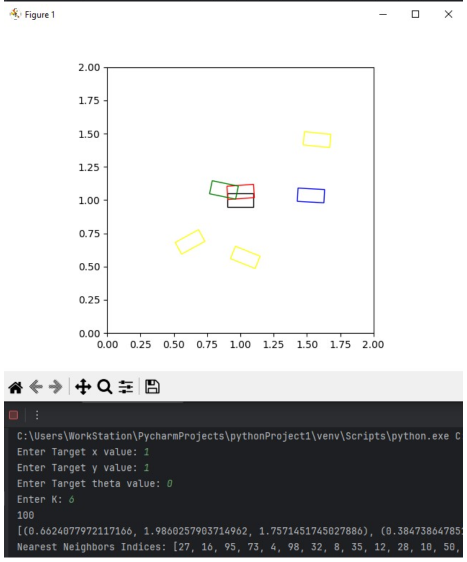
For the next examples, all input was given thought the command prompt



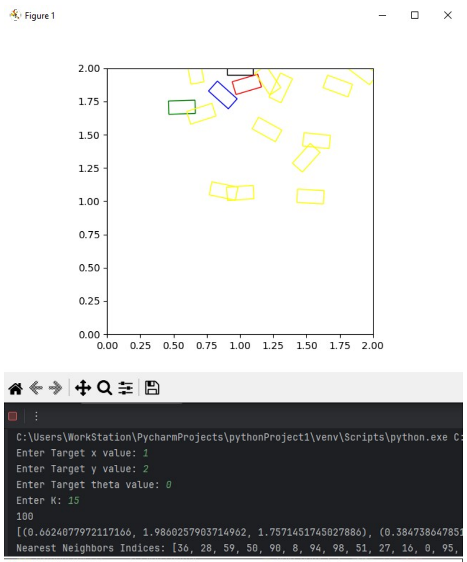
```
C:\Users\WorkStation\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:
Enter Target x value: 1
Enter Target y value: 1
Enter Target theta value: 0
Enter K: 3
100
[(0.6624077972117166, 1.9860257903714962, 1.7571451745027886), (0.384738647851
Nearest Neighbors Indices: [27, 16, 95, 73, 4, 98, 32, 8, 35, 12, 28, 10, 50,
```



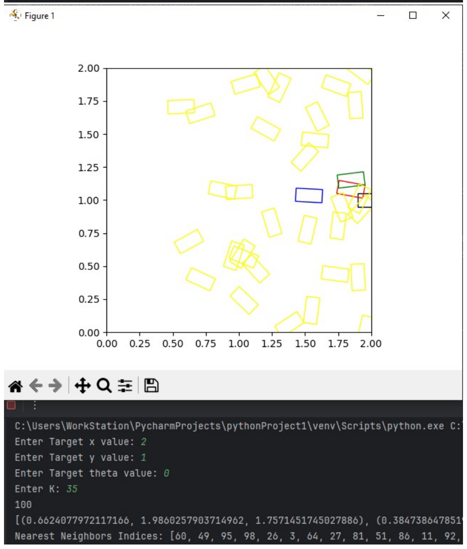
```
C:\Users\WorkStation\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:
Enter Target x value: 1
Enter Target y value: 1
Enter Target theta value: 10
Enter K: 3
100
[(0.6624077972117166, 1.9860257903714962, 1.7571451745027886), (0.38473864785
Nearest Neighbors Indices: [79, 99, 40, 17, 39, 15, 70, 68, 43, 89, 58, 42, 32
```



```
C:\Users\WorkStation\PycharmProjects\pythonProject1\venv\Scripts\python.exe C
Enter Target x value: 1
Enter Target y value: 1
Enter Target theta value: 0
Enter K: 6
100
[(0.6624077972117166, 1.9860257903714962, 1.7571451745027886), (0.38473864785
Nearest Neighbors Indices: [27, 16, 95, 73, 4, 98, 32, 8, 35, 12, 28, 10, 50,
```



```
C:\Users\WorkStation\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:
Enter Target x value: 1
Enter Target y value: 2
Enter Target theta value: 0
Enter K: 15
100
[(0.6624077972117166, 1.9860257903714962, 1.7571451745027886), (0.384738647851
Nearest Neighbors Indices: [36, 28, 59, 50, 90, 8, 94, 98, 51, 27, 16, 0, 95,
```



```
C:\Users\WorkStation\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:
Enter Target x value: 2
Enter Target y value: 1
Enter Target theta value: 0
Enter K: 35
100
[(0.6624077972117166, 1.9860257903714962, 1.7571451745027886), (0.384738647851
Nearest Neighbors Indices: [60, 49, 95, 98, 26, 3, 64, 27, 81, 51, 86, 11, 92,
```

Introduction: The task for problem 2.2 is to determine and visualize the three nearest configurations of a 2D rigid body, based on a given target configuration. A rigid body in this context is a rectangle that can rotate and translate in a 2D plane. The target configuration and the nearest configurations are defined by the x and y coordinates of the body's geometric center and its orientation θ. A weighted distance metric combining Euclidean distance and rotational difference is used to locate the nearest configurations.

Distance Metric: The distance metric D for this problem is a weighted average of translational distance d_t and rotational distance d_r, where $\alpha = 0.7$.

It is computed as $D = \alpha d_t + (1-\alpha)d_r$. This metric accounts for both the spatial and angular proximity of configurations, which is essential for the accurate positioning of rigid bodies.

Rigid Body Configuration Space: The configuration space for the rigid body is a set of potential (x, y, θ) combinations within the workspace limits. This space is discretized and represented as a matrix, where each row corresponds to a particular configuration. The workspace limits are defined to be within the [0, 2] range for both x and y coordinates.

Implementation: The 'rigid_body_2.py' script reads a list of potential configurations from the 'rigid_configs.npy' file and calculates the weighted distance to the target configuration for each. It then selects the three configurations with the smallest distances. These configurations, along with the target, are visualized using Matplotlib, with distinct colors and shapes for clarity.

Visualization The rigid body at the target configuration is highlighted, and its three nearest neighbors are indicated, in a 2D plot. The plot clearly demonstrates the relative proximity of each neighbor to the target. Example outputs are provided, showcasing the functionality of the script in various scenarios with different target configurations and sets of neighbors.

2.3 Interpolation along the straight line in the C-space

As requested, attached in the zip files is the .mp4 animation that generate a plot, animating the 2d rigid body(car) moving from the start to the goal configuration step by step.

The interpolation resolution used in the animation is determined by the variable STEP, which is set to 0.1 unit per step. Inputting a higher number for the value of STEP would make the animation go faster but less smooth. Inputting a lower number for the value of STEP would make the animation take more time but would make it look smoother as it is using more intermediate frames for showing the rigid body's movement.

The animation takes configurations as arguments in the settings of the file as opposed to the command prompt.

- The animation `"2.3 Animation.mp4"` takes `[0.5,0.5,0]` as the start and `[1.2,1.0,0.5]` as the goal.

---

3 Motion Planning for a first-order car

3.1 Implement the dynamics model of the car, and use a keyboard to control it

As requested, attached in the zip files is the .mp4 animation that generates a plot with a car(rectangle) that can be maneuvered. The animation shows a plot with a blue rectangle that follows regular car movement dynamics when arrow keys are pressed.

- The animation "car$_{E}XAMPLE.mp4\text{''}takesplotsabluerectangle(car)thatdisplaysregularcarmovementdynamicsandcanbem$

3.2 Integration of dynamics for constant control

As requested, attached in the zip files is the .mp4 animation that generates a plot with car(rectangle) that can receive start state as well as constant control and generate the sequence of states the robot(car) takes over the five seconds. The examples takes arguments through the configuration settings of the file script instead of the command prompt as shown in the video.

- The animation "car$_{2}.mp4\text{''}takes$[0.5,0.2]$asthestartand$[1.2,1.0,0.5]$asthegoal.$
