

### Homework 3 – Question 1

Adi Album 316563956 and Tomer Epshtein 313200750

#### Exercise 3.1 (A):

The aspect we chose to improve is the sampling method.

*prm\_basic.py*: The basic sampling method in *prm\_basic* is to randomly select landmarks inside a scene's bounding-box and verify each landmark is valid, do so until *num\_landamrk* landmarks have been chosen. These are the algorithm's landmarks.

The main disadvantage of this method occurs when there are narrow passages. The probability of sampling points inside narrow passages decreases as the passage becomes narrower.

We implemented a 'Gaussian sampling' method based on "The Gaussian Sampling Strategy for Probablistic Roadmap Planners" (Boor, Overmas, van der Strappen), as presented in class.

*prm\_gaussian\_sampling.py*: The sampling method:

We randomly sample a point, if it is valid we ignore it and sample another point.

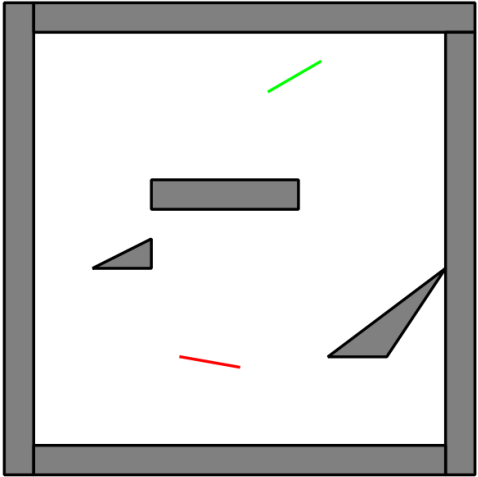
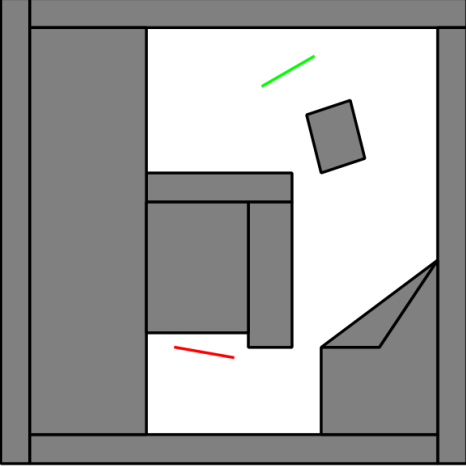
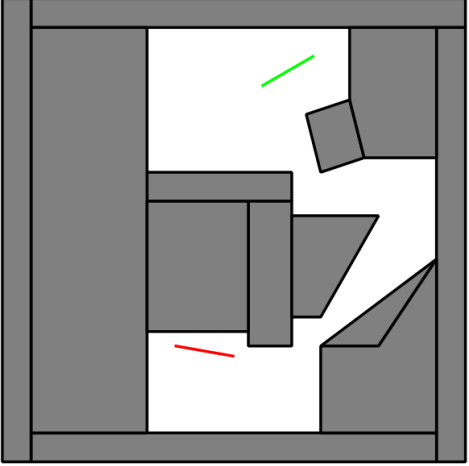
So we now have a non-valid point. We sample another point from a normal distribution centered at the current invalid point. If the new sampled point is valid – We add it to our set of landmarks.

This sampling method promotes points that are near obstacles, therefor the probability of sampling points near narrow passages increases.

The intuition behind this sampling method is that it has a higher probability of sampling points near obstacles. In many cases, these coincide with difficult passageways and often the shortest paths are those that occur near obstacles.

#### Exercise 3.1 (B):

We designed 3 scenes for our rod robot, with difficulty levels increasing:

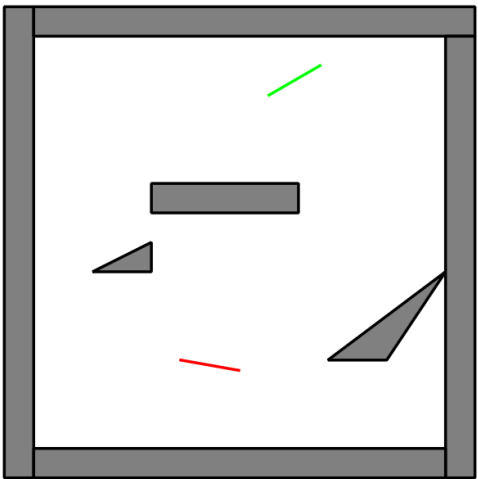
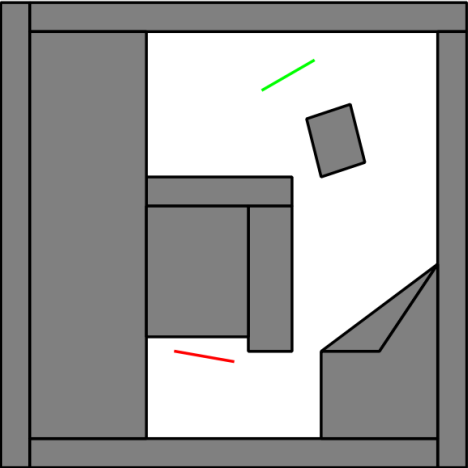
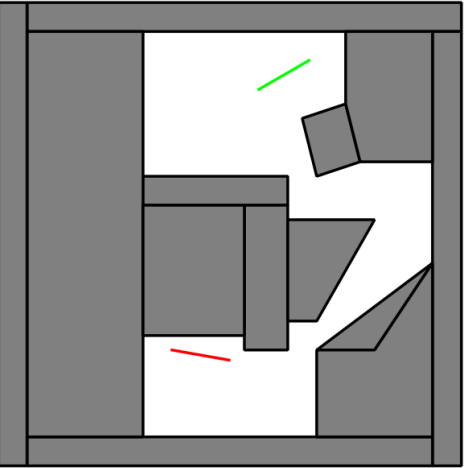
<i>our_easy_robot_maze.json</i>	<i>our_medium_robot_maze.json</i>	<i>our_difficult_robot_maze.json</i>
		

### Exercise 3.1 (C):

#### Our empirical definitions:

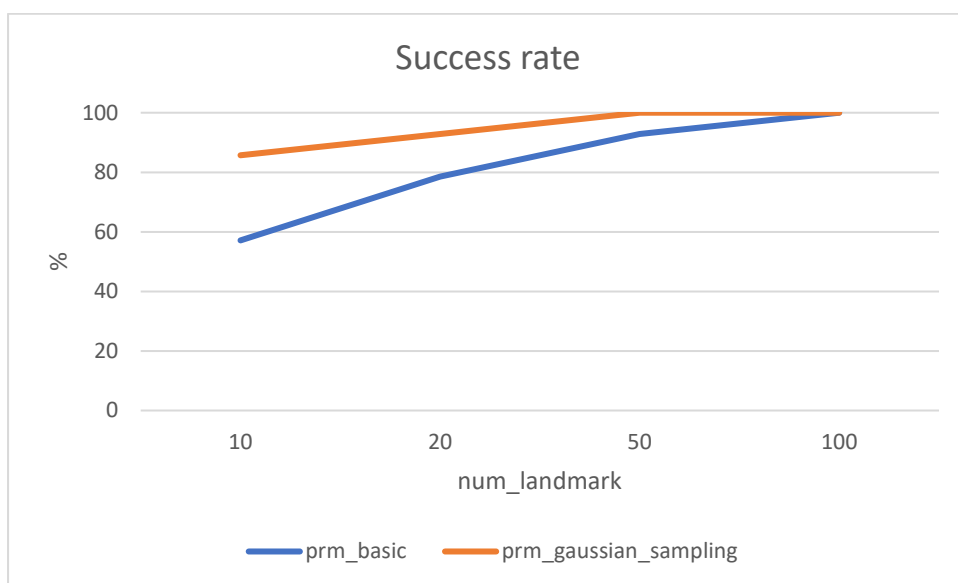
- Success rate: Percentage of runs with a successful path from start position to target position, of 14 attempts
- Running time: The average amount of time (in secs) it took the program to run on 14 attempts
- Number of samples needed to reach a solution when one exists: Number of landmarks needed for 3 consecutive runs with a successful path.

We calculate these properties for everyone of the following mazes:

<i>our_easy_robot_maze.json</i>	<i>our_medium_robot_maze.json</i>	<i>our_difficult_robot_maze.json</i>
		

We will first analyze our solver's performance on the easy maze *our\_easy\_robot\_maze.json*. For each of the following  $num\_landmark \in \{10, 20, 50, 100\}$ , run 14 times and calculate the average of the following parameters:

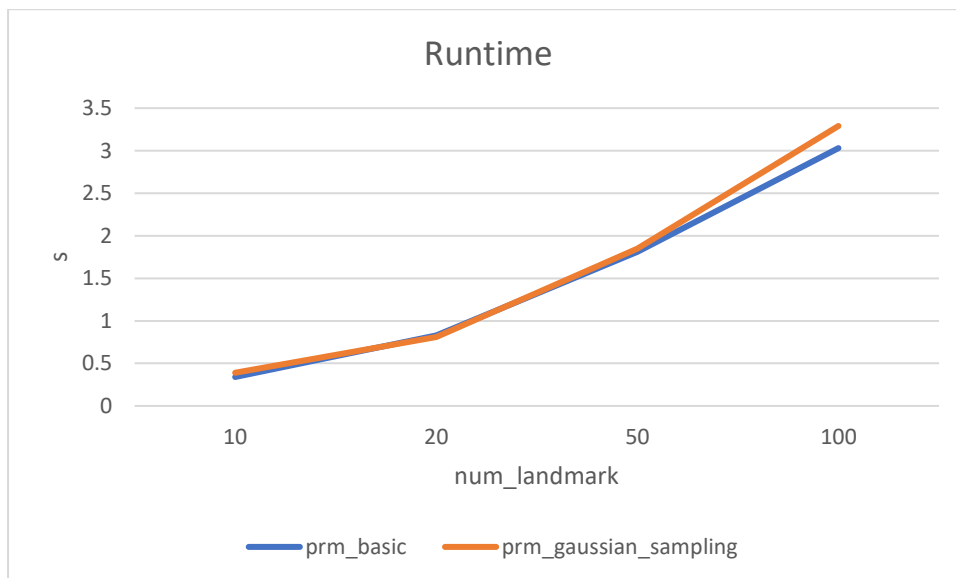
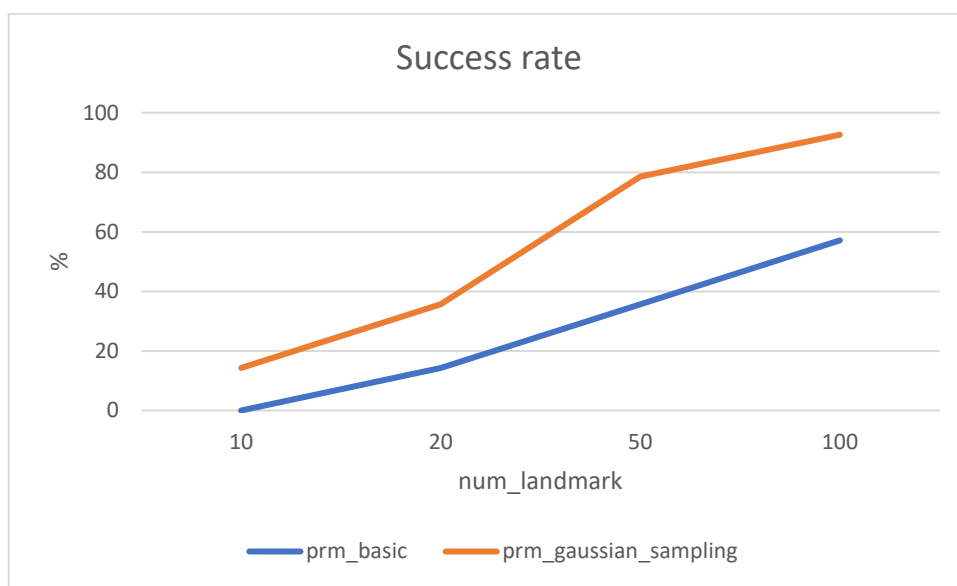
<i>num_landmark</i>		10	20	50	100
<i>prm_basic.py</i>	Success rate	57.13%	78.57%	92.86%	100%
	Running time	0.41s	1.18s	2.14s	3.60s
<i>prm_gaussian_sampling.py</i>	Success rate	85.71%	92.86%	100%	100%
	Running time	0.44s	1.20s	1.96s	3.82s



	Number of samples needed to reach a solution when one exists
<i>prm_basic.py</i>	12
<i>prm_gaussian_sampling.py</i>	5

Next we'll analyze our solver's performance on the medium maze *our\_medium\_robot\_maze.json*. For each of the following  $num\_landmark \in \{10, 20, 50, 100\}$ , run 14 times and calculate the average of the following parameters:

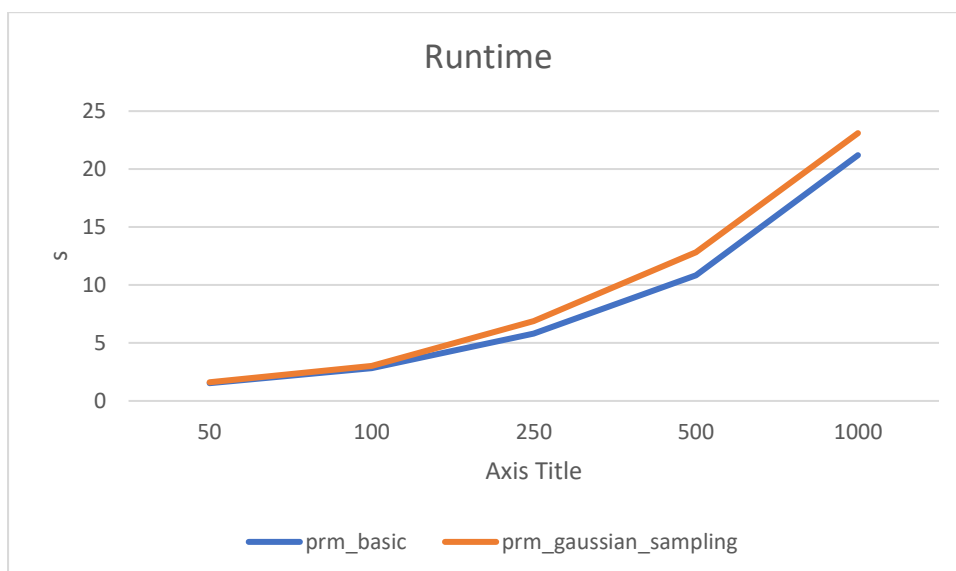
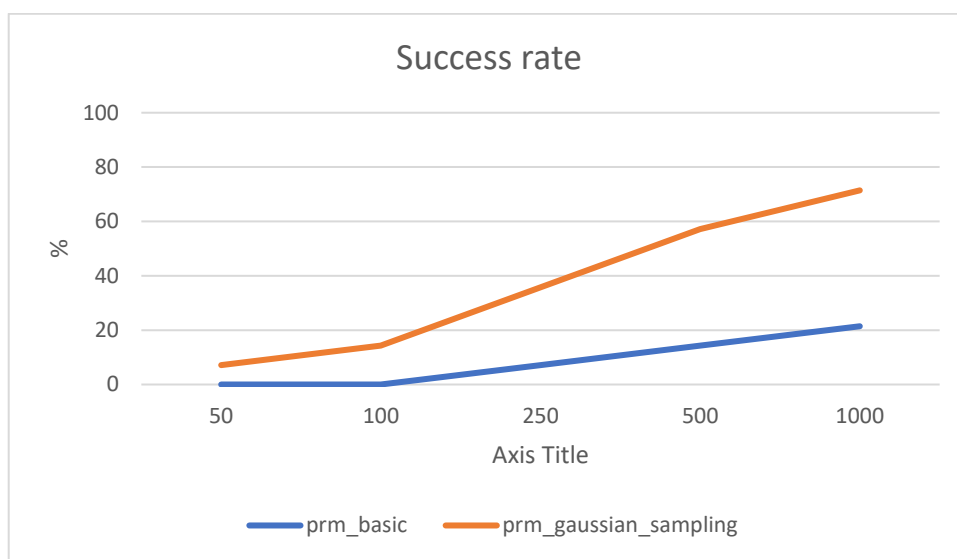
<i>num_landmark</i>		10	20	50	100
<i>prm_basic.py</i>	Success rate	0%	14.3%	35.71%	57.14%
	Running time	0.34s	0.83s	1.81s	3.03s
<i>prm_gaussian_sampling.py</i>	Success rate	14.3%	35.71%	78.6%	92.6%
	Running time	0.39s	0.81s	1.85s	3.29s



	Number of samples needed to reach a solution when one exists
<i>prm_basic.py</i>	90
<i>prm_gaussian_sampling.py</i>	30

Lastly we'll analyze our solver's performance on the difficult maze *our\_difficult\_robot\_maze.json* For each of the following  $num\_landmark \in \{50, 100, 250, 500, 1000\}$ , run 14 times and calculate the average of the following parameters:

<i>num_landmark</i>		50	100	250	500	1000
<i>prm_basic.py</i>	Success rate	0%	0%	7.14%	14.3%	21.43%
	Running time	1.53s	2.82s	5.80s	10.84s	21.20s
<i>prm_gaussian_sampling.py</i>	Success rate	7.14%	14.3%	35.72%	57.14%	71.43%
	Running time	1.61s	3.01s	6.88s	12.83s	23.1s



	Number of samples needed to reach a solution when one exists
<i>prm_basic.py</i>	6100
<i>prm_gaussian_sampling.py</i>	2350

#### Summary:

- Our empirical results correlate well with our proposed hypothesis: The success rate of *prm\_gaussian\_sampling* is significantly better than *prm\_basic*. This difference becomes greater as the scene becomes more difficult and complex.
- Our empirical results show a minor increase in runtime for *prm\_gaussian\_sampling* over *prm\_basic*. This seems to be consistent without direct relation to the scene's complexity.
- Our empirical results also show a major decrease in ('minimum') number of samples needed to reach a solution when one exists. This too becomes more dramatic as the scene becomes more complex.

### Exercise 3.1 (D):

We implemented two different distance measures and integrated them in two solvers:

*prm\_gaussian\_sampling\_dist\_1.py* and *prm\_gaussian\_sampling\_dist\_2.py*.

The original distance measure given in *prm\_basic.py* is the Euclidean distance between the rod's single endpoint. This has significant disadvantages that occur from the fact it only considers the translational position of one endpoint of the rod.

We implemented two different distance measures:

1. *prm\_gaussian\_sampling\_dist\_1.py*:

Here we calculated the Euclidean distance of the two endpoints of the rod

This can be seen in the following code:

```
# distance used to weigh the edges
def edge_weight(p, q):
    base_rod_position = [p[0], p[1], p[0] + length.to_double() * math.cos(p[2]), p[1] + length.to_double() * math.sin(p[2])]
    target_rod_position = [q[0], q[1], q[0] + length.to_double() * math.cos(q[2]), q[1] + length.to_double() * math.sin(q[2])]

    sd = math.sqrt((base_rod_position[0]-target_rod_position[0])**2 +
                  (base_rod_position[1] - target_rod_position[1])**2 +
                  (base_rod_position[2] - target_rod_position[2])**2 +
                  (base_rod_position[3] - target_rod_position[3])**2)

    return sd
```

$p[0]$  and  $q[0]$  are the  $x$ -axis of an endpoint of rod's base and target positions

$p[1]$  and  $q[1]$  are the  $y$ -axis of an endpoint of the rod's base and target positions

$p[0] + length * \cos(p[2])$  and  $q[0] + length * \cos(q[2])$  are calculations that determine the  $x$ -axis of the other endpoint of the rod's base and target positions

$p[1] + length * \sin(p[2])$  and  $q[1] + length * \sin(q[2])$  are calculations that determine the  $y$ -axis of the other endpoint of the rod's base and target positions

Next we calculate a 4-dimensional Euclidean distance of the 4 above pairs.

2. *prm\_gaussian\_sampling\_dist\_2.py*:

Here we calculate a weighed distance between the Euclidean distance of an endpoint of the rod and the rotational distance between the rods.

```
# distance used to weigh the edges
def edge_weight(p, q, is_clockwise):
    rod_translational_dist = math.sqrt((p[0] - q[0])**2 + (p[1] - q[1])**2)
    rod_rotational_dist = path_angular_dist(p[2], q[2], is_clockwise)
    w_t, w_r = 1, 1
    weighted_dist = w_t * rod_translational_dist + w_r * rod_rotational_dist
    return weighted_dist
```

The weights of the weighted combination are given as hyperparameters:

$w_t, w_r = 1, 1$ .

The *path\_angular\_distance* is the calculation of distance, in radians, between the two given angles:

```

# Given two angles theta_1, theta_2 in range [0, 2pi) and a direction (clockwise, or anti-clockwise),
# compute the distance between theta_1 and theta_2 in that direction. Result must be in range [0, 2pi)
def path_angular_dist(theta_1, theta_2, is_clockwise):
    # calculate clockwise distances:
    if theta_2 - theta_1 < -math.pi:
        angular_dist = theta_1 - theta_2
    elif theta_2 - theta_1 <= 0:
        angular_dist = theta_1 - theta_2
    elif theta_2 - theta_1 < math.pi:
        angular_dist = 2*math.pi - theta_2 + theta_1
    else:
        angular_dist = 2*math.pi - theta_2 + theta_1

    if not is_clockwise:
        if angular_dist != 0:
            angular_dist = 2 * math.pi - angular_dist

    return abs(angular_dist)

```

These calculations are straightforward when viewing the different cases.

### Analysis of the differences between the distance measures:

According to the experiments ran on the different scenes we generated:

The first distance measure, defined by the combined Euclidean distance of both endpoints generates a general good balance between rotation and translation of the rod. Minimizing this distance measure generates paths that seem moderately “wiggly” – Trying to minimize the overall rotation and translation together.

This method’s advantages are: No apparent hyperparameter, maintains a natural balance between rotation and translation.

This method’s disadvantages are: For a small rod, the movement is quite chaotic in the rotational sense. The ‘fine’ that the algorithm pays for aggressive rotations becomes smaller as the stick’s length decreases. This causes what looks like chaotic movement.

The second distance measure, defined by the weighted distance of the translational Euclidean distance of the rod’s endpoint and the rotational distance of the rod’s positions. Minimizing this distance measure generates paths that generally seem to either translate or rotate in motion between landmarks.

This method’s advantages are: Stability across rod lengths, control (via hyperparameter weights) of the rotation-translation tradeoff.

This method’s disadvantages are: Hyperparameters (How to choose optimal hyperparameters? Translational and rotational distances aren’t scaled accordingly. Hyperparameters are chosen empirically).