RBE550: Motion Planning

Project 1: Throwing Points on a Disk

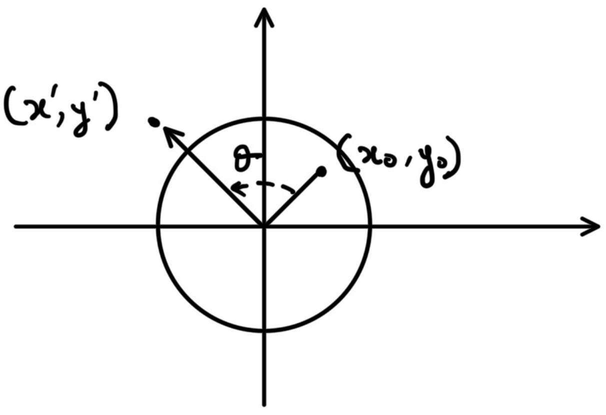
Student 1: Dhruv, Agrawal
Student 2: Shreyas, Khobragade

Project goals:

- To assess the understanding of fundamental concepts required for the course.
- To learn about key operations in OMPL, specifically sampling and collision checking for a simple point robot.

1 Theoretical Questions

1. **Consider A, a unit disc centered at the origin in the workspace $W = R^2$. Suppose A is described by the algebraic primitive $H = (x, y) \quad | \quad x^2 + y^2 \leq 1$. Demonstrate that rotating this primitive about the origin does not alter its representation. To prove this, show that any point within the rotated primitive H' is also within H, and vice versa.**



(a) Point rotation

Figure 1: Representation of a point in H when H is rotated by a certain angle

Let there be a point P (x_0, y_0) lying inside H. Assuming that the disc is rotated around an axis perpendicular to the workspace at the origin. After the rotation of H let the new coordinates of the point be $P^{'}(x^{'}, y^{'})$. Using the rotation matrices we can represent $(x^{'}, y^{'})$ as follows:

$$\begin{bmatrix} x^{'} \\ y^{'} \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} x_0 cos(\theta) - y_0 sin(\theta) \\ x_0 sin(\theta) + y_0 cos(\theta) \end{bmatrix}$$

$$\begin{aligned} x^{'} &= x_0 cos\theta - y_0 sin\theta \\ y^{'} &= x_0 sin\theta + y_0 cos\theta \end{aligned}$$

To check if $P^{'}(x^{'}, y^{'})$ is inside the unit circle, we need to verify if:

$$(x^{'})^2 + (y^{'})^2 \leq 1$$

$$\begin{aligned} (x^{'})^2 &= (x_0 cos\theta - y_0 sin\theta)^2 \\ (y^{'})^2 &= (x_0 sin\theta + y_0 cos\theta)^2 \end{aligned}$$

$$(x^{'})^2 = x_0^2cos^2\theta + y_0^2sin^2\theta - 2x_0y_0cos\theta sin\theta$$
$$(y^{'})^2 = x_0^2sin^2\theta + y_0^2cos^2\theta + 2x_0y_0cos\theta sin\theta$$

Adding the above two equations we get:

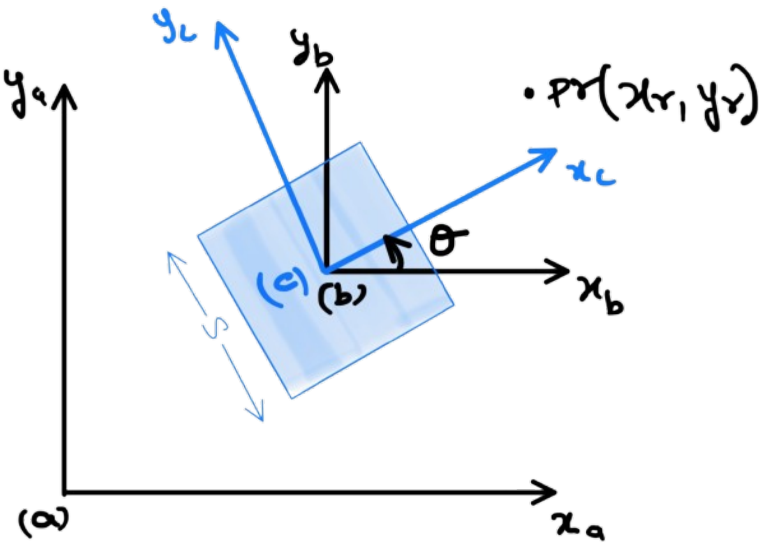
$$(x^{'})^2 + (y^{'})^2 = x_0^2cos^2\theta + y_0^2sin^2\theta - 2x_0y_0cos\theta sin\theta + x_0^2sin^2\theta + y_0^2cos^2\theta + 2x_0y_0cos\theta sin\theta$$

This simplifies to:

$$(x^{'})^2 + (y^{'})^2 = x_0^2 + y_0^2$$

Since, $x_0^2 + y_0^2 \leq 1$, it follows that $(x^{'})^2 + (y^{'})^2 \leq 1$
Thus the point $P^{'}(x^{'}, y^{'})$ is inside the circle. This is true for any other point on the rotated disc.

2. **Let S be a square object parameterized by (x, y, θ, s), where (x, y) denotes its position in the X-Y plane, θ represents its rotation relative to its center, and s is the length of its sides. Given a random point $p_r = (x_r, y_r)$, provide a pseudocode algorithm to determine if p_r is inside the square. The function should return False if the point is inside or on the edge of the square, and True otherwise.**



(a) Point in square

Figure 2: Representation of a square and random point in world frame

Here (x_r, y_r) have been represented as (x_r^a, y_r^a) where a-frame is the world frame.
The b-frame is a frame at the origin of the square but having a rotation $\theta = 0$ with respect to the x-axis of a-frame.
The c-frame is nothing but the b-frame rotated at an angle θ.

Algorithm 1 Point in Square Algorithm

```
function PointInSquare($x_r^a$, $y_r^a$, $x^a$, $y^a$, $\theta$, $s$)
                                      ▷ $x_r^a, y_r^a$ are random point coordinates in a-frame
                                      ▷ $x^a, y^a$ are the central point coordinates of the square in a-frame
                                      ▷ $\theta$ is the rotation of the square with the x-axis of a-frame
                                      ▷ $s$ is the length of a side of the square

    $x_r^b = x_r^a - x^a$                                          ▷ express $x_r$ in b-frame
    $y_r^b = y_r^a - y^a$                                          ▷ express $y_r$ in b-frame
    $(x_r^c, y_r^c)$ = Rot($x_r^b, y_r^b$) to c-frame coordinates   ▷ rotate the b-frame by $\theta$ degrees to obtain c-frame
    if $x_r^c \leq s/2$ and $x_r^c \geq -s/2$ then
        if $y_r^c \leq s/2$ and $y_r^c \geq -s/2$ then return False
                                      ▷ If the coordinates are within the square or on the boundary then return False
    return True                       ▷ If the coordinates are outside the square then return True
```

2 Programming Component

1. **Implement a sampler that generates uniformly distributed points on a disk in R^2 space using OMPL. Complete the following tasks:**

   - **Implement the sampleNaive(ob::State *state) function in DiskSampler.cpp. Use the following process to perform naive sampling on a disk with a radius of 10:**
     - **Sample random polar coordinates r in $[0, 10]$ and θ in $[0, 2\pi)$.**
     - **Convert the polar coordinates to Cartesian coordinates (x, y) in R^2.**

     Evaluate whether these points are uniformly distributed on the disk. Explain why they may not be uniformly distributed and include the naive_samples.png figure in your report.

     The points are plotted on the disk using uniform random generators for polar coordinates (r, θ).
     The Fig3 below shows the output of the random generator.
     We can see that when r is close to the origin, the points are densely packed and they are sparsely packed when r is away from the origin.
     This is incorrect and not what we expected in a uniform distribution.
     θ is uniformly sampled and we can see from Fig3 that at any radius r within the bounds $[0, 2\pi)$ the samples are uniformly separated.
     Thus, θ does not play any role in this incorrect distribution.
     The reason for this discrepancy is that, there is an equal probability in a uniform random generator to choose any radius r, so the number of points in a circle of smaller radius will be equal to the number of points in a circle of bigger radius, which results in densely sampled circular area near the origin rather than a uniform distribution.

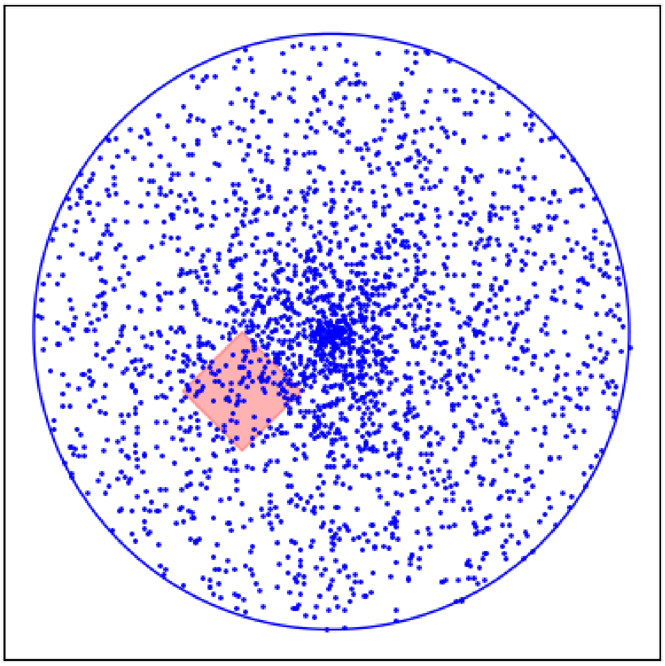


Figure 3: Naive Sampling

   - **Implement the sampleCorrect(ob::State *state) function in DiskSampler.cpp. Use a correct sampling process to generate uniformly random points on the disk. (Uniform here means samples drawn from a uniform distribution over the area of the disk, not a grid pattern). Many methods can achieve this, but the most elegant solution involves a single code change. Describe what you changed and why, and include the correct_samples.png figure in your report.**

From Fig3, it is evident that the Naive approach is incorrect.
The correct solution is obtained by taking the square root of the uniform random sample for the radius parameter within the modified bounds $(0, r^2)$.

For uniform random distribution, the number of samples in the circle should linearly increase with radius or the distance from the centre.
However, in the case of naive sampling, the number of samples remain constant in any radius.
Taking square root of value of radius ensures that as the distance from centre increases, the points sampled by random sample generator also increase, thereby keeping the density of sampled points uniform over the whole circle.
However, this results in points sampled in $(0, \sqrt{r})$ bounds resulting in uniform distribution over a circle of radius $\sqrt{r}$.
But we require the points to be sampled in the range $(0, r)$.(In our case $r = 10$).
Hence, we take random sample over a modified bound $(0, r^2)$.

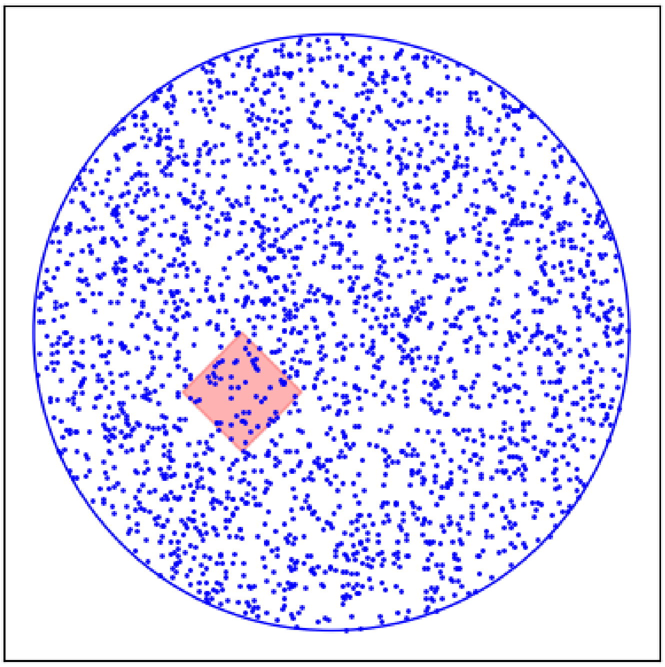


Figure 4: Correct Sampling

2. **Implement a collision checker for a point and a translated and rotated square using the algorithm proposed in Theoretical Question 2. Complete the following task:**

   - **Implement the isStateValid(ob::State *state) function in DiskSampler.cpp. This function should check for collisions with a square obstacle of edge size $2 * \sqrt{2}$, located at (-3,-2) and rotated by $\pi/4$. Use the visualize.py script to verify the correctness of your implementation. Include the final figure produced by the visualization script in your report.**

     The algorithm developed in Theoretical question 2 is implemented in the isStateValid function in DiskSampler.cpp.
     This function checks for collision of the random points with the pink square.
     If the points are in collision then, the points are not plotted.
     The algorithm is developed by translating and rotating the world frame of reference to the center of the pink square.
     Distance of each point in the space is calculated assuming the new frame as reference frame.
     If the x or y coordinate of the random point is within the length of side of the square then it is collision with the square.
     This is implemented for both the Naive sampling approach as well as the Correct sampling approach. The

results are shown below:

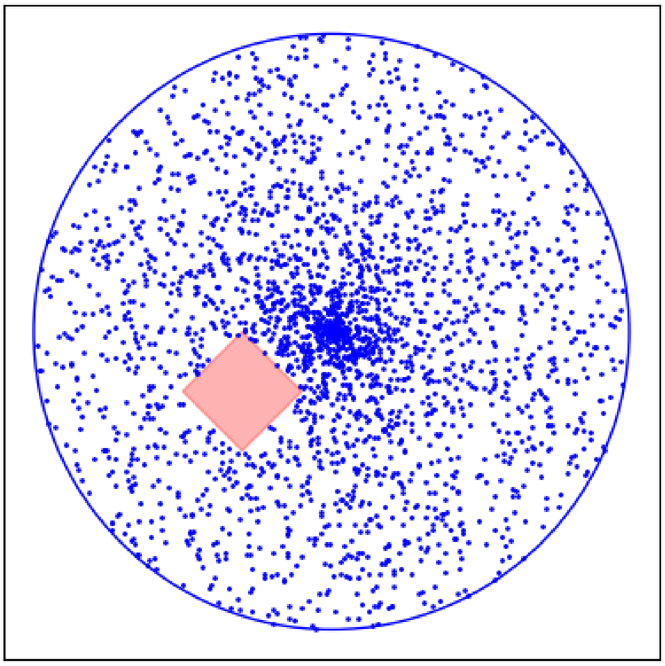


Figure 5: Naive Sampling with collision checker

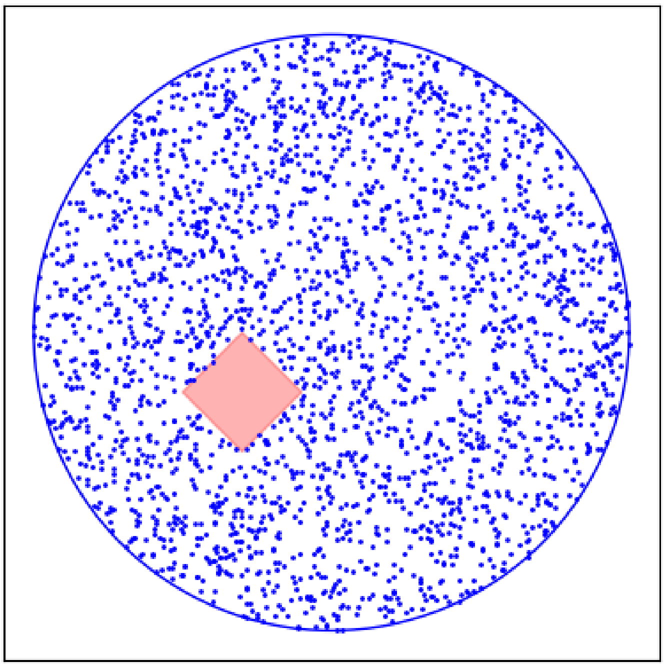


Figure 6: Correct Sampling with collision checker