RBE550: Motion Planning

Motion Planning HW1

Student 1: Harsh Chhajed
Student 2: Pranay Katyal

Q1

We consider a set of discrete points on the boundary of the unit disc (the unit circle, $x^2 + y^2 = 1$) and verify that their rotated counterparts still satisfy $x^2 + y^2 = 1$.

Defining the Discrete Points

Take $n = 4$ points on the unit circle for simplicity:

$$P = \{(0.5, -0.5), (0, 1), (0, 0.25), (0, 0.5)\}.$$

Rotation Matrix, around the Z-Axis

The rotation matrix for an angle $\theta = \frac{\pi}{4}$ (45°) is:

$$R = \begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} 0.7071 & -0.7071 \\ 0.7071 & 0.7071 \end{bmatrix}.$$

Rotating the points about the center using the Rotation matrix.

Using R, the rotated coordinates (x', y') of a point (x, y) are given by:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R \begin{bmatrix} x \\ y \end{bmatrix}.$$

Rotating $(0, 1)$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.7071 \\ 0.7071 \end{bmatrix}.$$

So, $(0, 1) \to \begin{bmatrix} -0.7071 \\ 0.7071 \end{bmatrix}$.

Rotating $(0, 0.5)$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} = \begin{bmatrix} -0.3536 \\ 0.3536 \end{bmatrix}.$$

So, $(0, 0.5) \to \begin{bmatrix} -0.3536 \\ 0.3536 \end{bmatrix}$.

Rotating $(0, 0.25)$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0.25 \end{bmatrix} = \begin{bmatrix} -0.1768 \\ 0.1768 \end{bmatrix}.$$

So, $(0, 0.25) \to \begin{bmatrix} -0.1768 \\ 0.1768 \end{bmatrix}$.

Rotating $(0.5, -0.5)$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 0.7071 \\ 0 \end{bmatrix}.$$

So, $(0.5, -0.5) \to \begin{bmatrix} 0.7071 \\ 0 \end{bmatrix}$.

Verifying whether they still satisfy the original equation of $H(x, y)$ primitive

For each rotated point:

1. $\begin{bmatrix} -0.7071 \\ 0.7071 \end{bmatrix} = (-0.7071)^2 + (0.7071)^2 = 1.$
2. $\begin{bmatrix} -0.3536 \\ 0.3536 \end{bmatrix} = (-0.3536)^2 + (0.3536)^2 = 0.25.$
3. $\begin{bmatrix} -0.1768 \\ 0.1768 \end{bmatrix} = (-0.1768)^2 + (0.1768)^2 = 0.0625.$
4. $\begin{bmatrix} 0.7071 \\ 0 \end{bmatrix} = (0.7071)^2 + 0^2 = 0.5.$

Conclusion

The rotated points satisfy $x^2 + y^2 \leq 1$, demonstrating that the unit disc remains invariant under rotation. The 4 points are still part of the disc, even after the rotation.

Q2

Pseudocode: Checking if a Point is Outside a Rotated Square Obstacle

Algorithm 1 Checking if a Point is Outside a Rotated Square

function isPointOutsideSquare(x, y, theta, s, x_r, y_r)
- $x\_t = x\_r - x$
- $y\_t = y\_r - y$
- $x' = x\_t \cdot \cos(\theta) + y\_t \cdot \sin(\theta)$
- $y' = -x\_t \cdot \sin(\theta) + y\_t \cdot \cos(\theta)$
- $half\_side = s/2$
- **if** $|x'| \leq half\_side$ AND $|y'| \leq half\_side$ **then**
  - **return** False ▷ The point is inside or on the edge of the square
- **else**
  - **return** True ▷ The point is outside the square

Explanation of the Pseudocode

The function `isPointOutsideSquare` checks whether a given point (x_r, y_r) lies outside a rotated square. The square is centered at the origin and has a side length s. The square is rotated by an angle θ with respect to the coordinate axes.

1. **Translation to Origin:** The first two lines of the pseudocode translate the point (x_r, y_r) to the origin of the rotated square. This is done by subtracting the reference coordinates (x, y) from the point's coordinates:

$$x_t = x_r - x, \quad y_t = y_r - y.$$

2. **Rotation Transformation:** The next two lines apply a rotation by the angle $-\theta$ using the standard 2D rotation matrix. The transformed point (x', y') is computed as:

$$x' = x_t \cdot \cos(\theta) + y_t \cdot \sin(\theta), \quad y' = -x_t \cdot \sin(\theta) + y_t \cdot \cos(\theta).$$

   After applying the rotation, the point's new coordinates (x', y') are now in the reference frame of the rotated square.

3. **Checking Point Location:** The function checks if the rotated point (x', y') lies within the bounds of the square. The square's half-side length is $s/2$. If both $|x'|$ and $|y'|$ are smaller than or equal to $s/2$, the point is inside or on the edge of the square, and the function returns `False`. Otherwise, if either $|x'|$ or $|y'|$ exceeds $s/2$, the point lies outside the square, and the function returns `True`.

Using MATLAB to test this code for 3 points:

```
function isPointOutsideSquare(x, y, theta, s, x_r, y_r)
    % Translate the point to the origin of the rotated square
    x_t = x_r - x;
    y_t = y_r - y;

    % Apply the rotation transformation
    x_prime = x_t * cos(theta) + y_t * sin(theta);
    y_prime = -x_t * sin(theta) + y_t * cos(theta);

    % Half-side length of the square
    half_side = s / 2;

    % Check if the point is inside or outside the square
    if abs(x_prime) <= half_side && abs(y_prime) <= half_side
        disp('The point is inside or on the edge of the square.');
    else
        disp('The point is outside the square.');
    end
end

    % Parameters
    s = 2 * sqrt(2);   % Side length of the square (2 sqrt(2))
    theta = pi / 4;     % Rotation angle (45 degrees)

    %  Test points
    testPoints = [
        -3.5, -2.5;  % Inside the square
        -3 + sqrt(2), -2; % On the edge of the square
        2, 2         % Outside the square
    ];

    % Reference point (origin for the rotated square)
    x  = -3; y = -2;

    % Loop through test points
    for i = 1:size(testPoints, 1)
        x_r = testPoints(i, 1);
        y_r = testPoints(i, 2);
        fprintf('Testing point (%f, %f):\n', x_r, y_r);
        isPointOutsideSquare(x, y, theta, s, x_r, y_r);
        fprintf('\n');
    end
```

We got the result as:

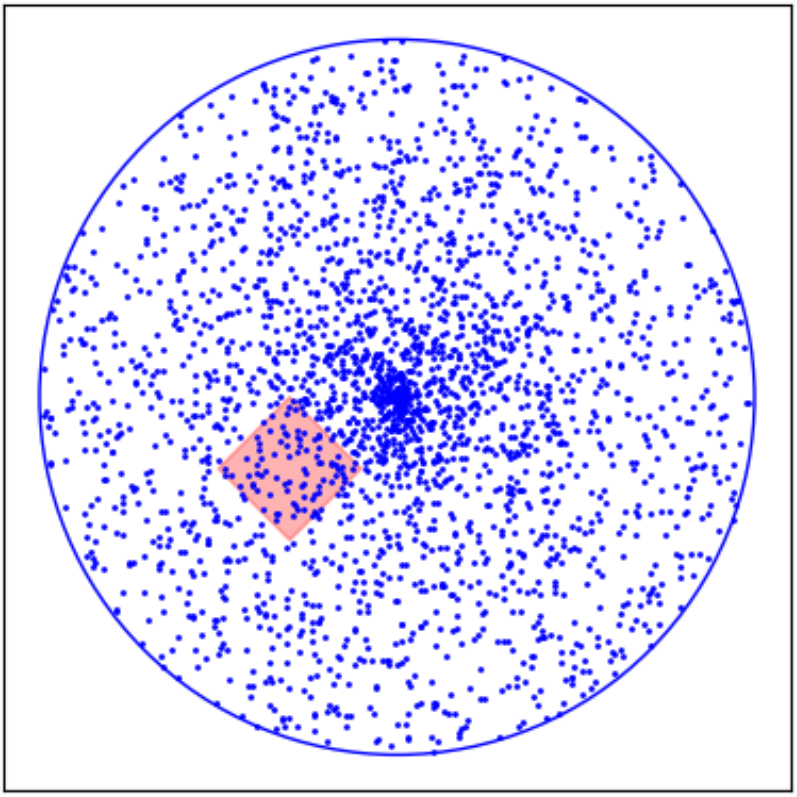
```
    >> obstaclecolidecheck
Testing point (-3.500000, -2.500000):
The point is inside or on the edge of the square.

Testing point (-1.585786, -2.000000):
The point is inside or on the edge of the square.

Testing point (2.000000, 2.000000):
The point is outside the square.
```

Thus, the function efficiently determines if a point is inside or outside a rotated square by translating, reverse rotating, and checking the sampled point's position relative to the square's boundaries.

Q1 - coding

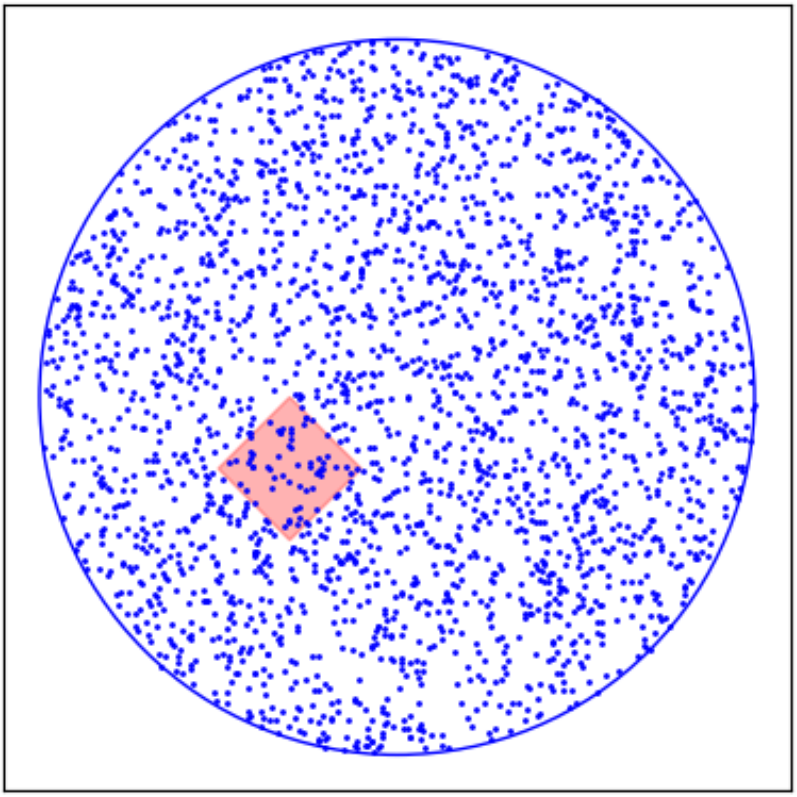
Implementing the sampleNaive

```
    bool DiskSampler::sampleNaive(ob::State *state)
{
    // ******* START OF YOUR CODE HERE *******//
    double sampleR = DiskSampler::rng_.uniformReal(0,10);
    double sampleTheta = DiskSampler::rng_.uniformReal(0,2*M_PI - std::numeric_limits<double>::epsilon());

    double x = sampleR * cos(sampleTheta);
    double y = sampleR * sin(sampleTheta);

    auto *r2state = state->as<ob::RealVectorStateSpace::StateType>();
    r2state->values[0] = x;
    r2state->values[1] = y;


    // ******* END OF YOUR CODE HERE *******//

    //The valid state sampler must return false if the state is in-collision
    return isStateValid(state);
}
```



Implementing the sampleCorrect

```
    bool DiskSampler::sampleCorrect(ob::State *state)
{
    // ******* START OF YOUR CODE HERE *******//
    double sampleR = sqrt(DiskSampler::rng_.uniformReal(0, 100));
    double sampleTheta = DiskSampler::rng_.uniformReal(0,2*M_PI - std::numeric_limits<double>::epsilon());

    double x = sampleR * cos(sampleTheta);
    double y = sampleR * sin(sampleTheta);

    auto *r2state = state->as<ob::RealVectorStateSpace::StateType>();
    r2state->values[0] = x;
    r2state->values[1] = y;


    // ******* END OF YOUR CODE HERE *******//


    //The valid state sampler must return false if the state is in-collision
    return isStateValid(state);
}
```



Inference

From what we have observed when we sample the points' R and Theta, using the *rng_* .uniformReal , when R is small, essentially closer to the center of the disc, since the area from which the point is being sampled is small, the sampled points are closer to each other and thus the density of sampled points as we go near the center of the disc

increases, this also explain why, as R increases the sampling area increases and therefore the distance between the sampled points starts increase and they get visibly less dense.

The work around it to sample correctly, and uniformly was to sample not at R but we sample R^2 and then later find the square root of it to get R. This works as The number of samples is proportional to the area of each radial ring in the disk. The radial distance R is more evenly distributed across the entire disk, instead of being clustered around the center. This helps prevent the bias towards the center of the disc.

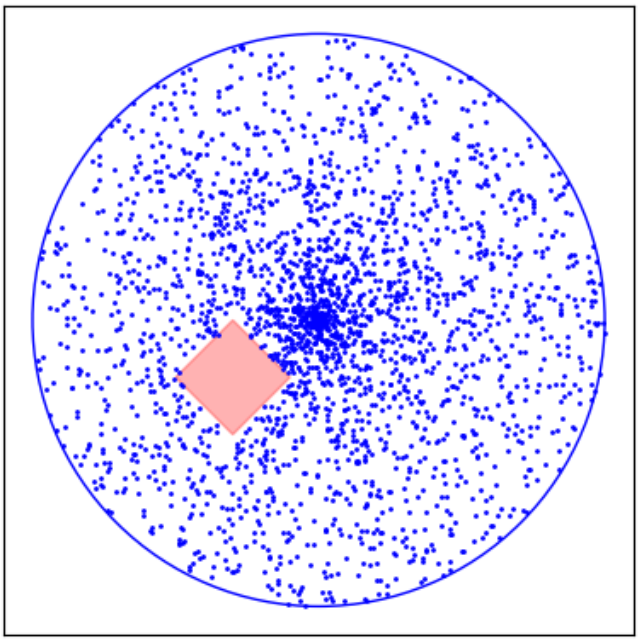
Q2 - Coding

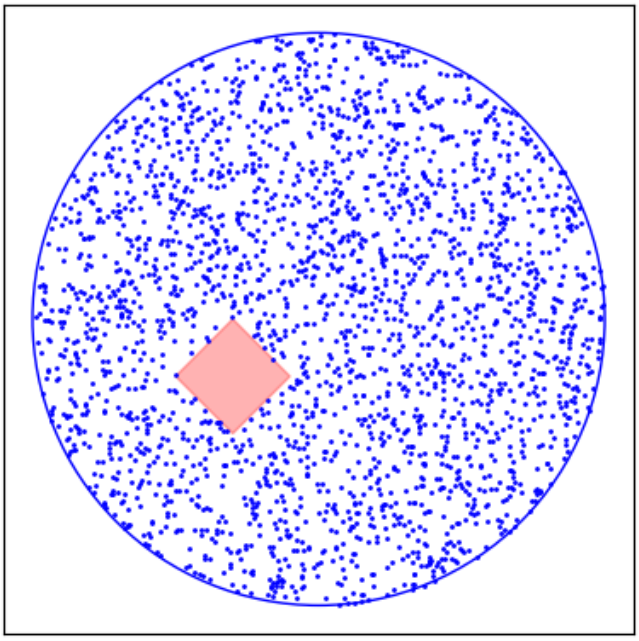
Implementing a collision checker, isStateValid and isPointOutsideSquare

```
    bool isPointOutsideSquare(double x, double y, double theta, double s, double x_r, double y_r){
        // Translate the point relative to the square's center
        double x_t = x - x_r;
        double y_t = y - y_r;

        // Rotate the point back (inverse rotation)
        double x_prime = x_t * std::cos(theta) + y_t * std::sin(theta);
        double y_prime = -x_t * std::sin(theta) + y_t * std::cos(theta);

        // Check if the point is outside the square
        double half_side = s / 2.0;
        return !(std::abs(x_prime) <= half_side && std::abs(y_prime) <= half_side);
}


bool isStateValid(const ob::State *state) {

    // cast the abstract state type to the type we expect
    const auto *r2state = state->as<ob::RealVectorStateSpace::StateType>();
    double x = r2state->values[0];
    double y = r2state->values[1];

    // A square obstacle with and edge of size 2*sqrt(2) is located in location [-3,-2,] and rotated pi/4
    // Fill out this function that returns False when the state is inside/or the obstacle and True otherwi

    // ******* START OF YOUR CODE HERE *******//

    // Square parameters
    double square_size = 2 * std::sqrt(2);
    double square_center_x = -3.0;
    double square_center_y = -2.0;
    double square_rotation = M_PI / 4.0;


    return isPointOutsideSquare(x , y, square_rotation, square_size, square_center_x, square_center_y);

    return true;
    // ******* END OF YOUR CODE HERE *******//
}
```

Conclusions drawn

Upon implementing our algorithm from $Q2$ and making changes to make it adapt to the code file available. We were now able to successfully sample the points while avoiding the square object.

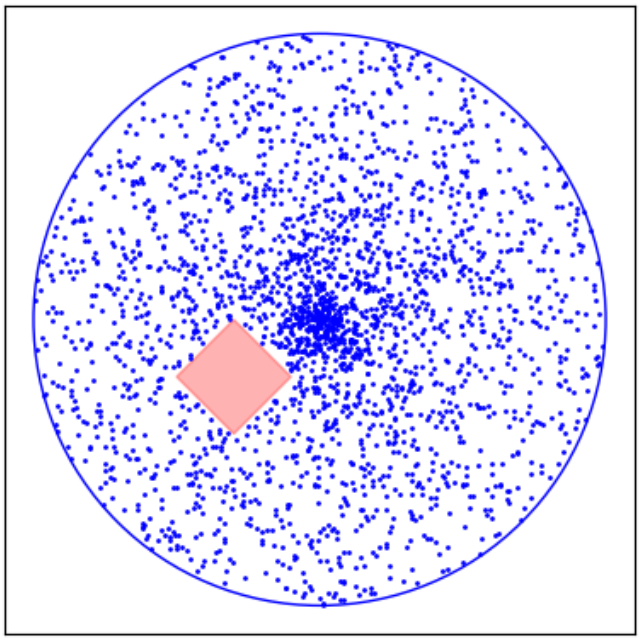




One thing we did extra was we made the square virtually bigger than it actually is as some points were sampled right next to its edges, but since the points weren't just a pixel but a whole disc they still had some of their part inside the edges of the square, but once I added this buffer region, no points were sampled in that way.

```
// Square parameters
double square_size = 2 * std::sqrt(2) + 0.3;
```

naiveSample



correctSample