Assignment 1

Ravi Raghavan
rr1133

Michelle Han
mmh255

October 8, 2023

1 Generate, visualize and store 2D polygonal scenes

1.1 Example Maps

The density of the maps was controlled by the amount of polygons in relation to the radii. Maps with greater radii (Fig. 1) required less polygons to fill the space compared to maps with smaller radii (Fig. 2). The shape of the polygons was controlled by the number of vertices. Maps with higher number of vertices resulted in polygons that were more round in appearance (Fig. 3). Maps with a lower amount of vertices appeared more sharp (Fig. 1 and 2). The four maps generated using *constructScene* and their corresponding parameters:

1.1.1 Big and Dense Polygons (see Fig. 1)

Number of Polygons: 10

Minimum Number of Vertices: 4	Maximum Number of Vertices: 6
Minimum Radius: 0.4	Maximum Radius: 0.5

1.1.2 Small and Dense Polygons (see Fig. 2)

Number of Polygons: 30

Minimum Number of Vertices: 4	Maximum Number of Vertices: 6
Minimum Radius: 0.1	Maximum Radius: 0.2

1.1.3 Small and Sparse Polygons (see Fig. 3)

Number of Polygons: 5

Minimum Number of Vertices: 15	Maximum Number of Vertices: 20
Minimum Radius: 0.05	Maximum Radius: 0.1

1.1.4 Big and Sparse Polygons (see Fig. 4)

Number of Polygons: 2
Minimum Number of Vertices: 5 Maximum Number of Vertices: 15
Minimum Radius: 0.5 Maximum Radius: 0.6

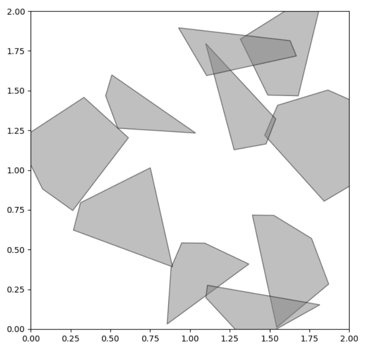


Figure 1: Big and dense polygons

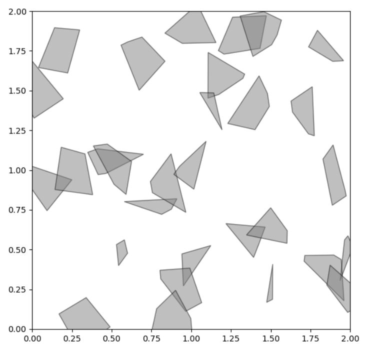


Figure 2: Small and dense polygons

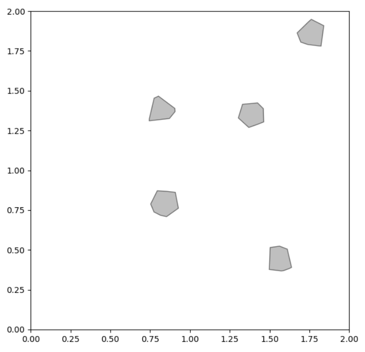


Figure 3: Small and sparse polygons

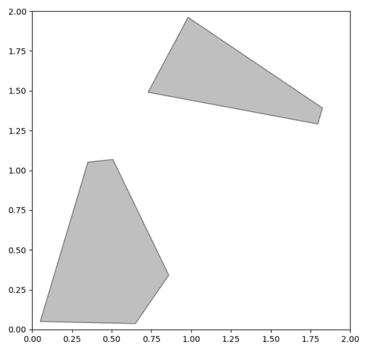


Figure 4: Big and sparse polygons

Rationale: Parameters were chosen accordingly for each combination of small/big and sparse/dense polygons. In other words, if we wanted to generate small and sparse polygons, we chose the parameters accordingly. Likewise, if we wanted to generate big and dense polygons, we chose the parameters accordingly.

1.2 Convex Polygon Letters

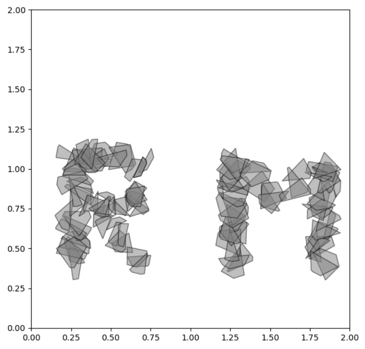


Figure 5: R and M generated using convex polygons

1.3 Convex Polygon Strategy

In order to generate convex polygons, the strategy outlined in the write up was followed.

1.3.1 Polygon Rotation

A randomly sampled vertex (*xcenter* , *ycenter*) was created inside the 2mx2m map. Using this vertex as the center, the minimum radius *rmin* was used to generate a circle of radius *rmin*. An angle, θ, is sampled from [0°,360°]. A vertex is calculated by:

$$x = xcenter + r * cos(\theta)$$

$$y = ycenter + r * sin(\theta)$$

The expressions determine where the vertice falls on the circle. This is repeated N times to get N vertices from [*nmin, nmax*]. The placement of the vertices along the circumference of a circle of radius *rmin* allows for the polygon to be rotated.

1.3.2 Extrude Vertices

To extrude a vertex, a radius r is sampled from [0, maximum vertices - minimum vertices]. The range [0, maximum vertices - minimum vertices] allows for a vertex to lie outside of the polygon's minimum radius. It allows for the circle that the new vertex lies on to be equal to or greater than radius *rmin*. The new position of the vertex is calculated by:

$$newx = x + r * cos(\theta)$$

$$newy = y + r * sin(\theta)$$

This process is repeated for all the vertices in the polygon.

1.3.3 Convexity of Polygons

To determine the convexity of a generated polygon, the *ConvexHull* function was used. The *makeConvex* function takes a polygon and applies *ConvexHull* to it. The result is the indices of vertices that form a convex polygon in counterclockwise order. This is stored in *hull.* The indices in *hull* are used to extract the vertices of the polygon that form a convex polygon into *vertices.* Another vertex, the first one, is appended to *vertices* so that the polygon is closed. *convexPolygons* is ordered in counterclockwise order. If the given polygon is already convex, then convexPolygons will receive the same amount of vertices as the original polygon.

2 2D collision checking for convex polygons

2.1 Implementation Details

To implement the collision checking approach, I used three methods. The first method was a naive method. The second and third methods were optimized methods. I implemented this collision checking approach three ways so that I could do a comparison between all implementations to see how the optimized approaches fares with respect to the overall runtime.

2.1.1 Naive Approach

The naive approach I implemented was rather simplistic. The first thing I did was to check all pairs of line segments(i.e. the line segments represent the edges) between the two polygons for collisions.

To check if Line Segment A and Line Segment B intersect, I parameterized both line segments and solved for the point of intersection using Cramer's Rule.

So, for example, if a line segment went from $(1, 2)$ to $(3, 5)$, I would parameterize this line segment as $x = 1 + 2s$ and $y = 2 + 3s$. Likewise, if the second line segment went from $(5, 6)$ to $(9, 11)$, I would parameterize this line segment as $x = 5 + 4t$ and $y = 6 + 5t$. Then, I would set the x parameterized equations equal to each other and, likewise, set the y parameterized equations equal to each other. Hence, my system of linear equations became:

$$1 + 2s = 5 + 4t$$

$$2 + 3s = 6 + 5t$$

Solving this system of equations was done using Cramer's Rule. Once I got my solution for s and t, if such a solution existed, then I would check to see if $0 \leq s, t \leq 1$. I want to make sure the intersection points is contained within the line segments!

Once I finished checking the edges for collisions, I checked to see if any of the vertices of one polygon are inside the other. To check if a point p was inside Polygon A, I picked a vertex of Polygon A, let's call this vertex V, and established a polar coordinate system centered about V. Then, I calculated the angle of each of the other vertices of A with respect to V, essentially constructing a polar coordinate system with respect to the V. I made sure to go in counterclockwise order so that the polar angles I calculated would be in increasing order. This essentially splits up A into "sections" in the polar coordinate system. Finally, I calculated the polar angle of the point p with respect to V. Using binary search, since the polar angles are in increasing order, I determined which "section" p belonged to. If p didn't belong to any section, obviously it wasn't contained in A. However, if p did belong to any of the sections, I had to check the distance between V and p to make sure that it belonged within the polygon. If line segment from V to p didn't intersect A or p was on one of the edges of A, then I knew that p was contained within A. Else, p was not contained within A.

Overall, if I had cases where there was either an intersection between one line segment from one polygon and a line segment from the other polygon or one vertex of one polygon was inside the other polygon, then I would return True to indicate that we had a collision. If I did not encounter either of these two cases, I would return False to indicate that there is NO collision.

2.1.2 Optimized Approach

To implement the optimized approach, my methodology can be broken down into two major portions. Let's say we are trying to detect if Polygon P and Q collide. The first thing I did was to construct bounding boxes around each polygon. Essentially, a bounding box for a polygon is akin to drawing a rectangle around the polygon that covers all its corners

Once the bounding boxes are constructed, when checking for collision, the first thing I check to see is if the bounding boxes for P and Q intersect. If no intersection is detected, I simply return False to indicate that there is NO collision.

However, if there is a collision detected between the bounding boxes of P and Q, I need to verify this collision. To do this verification, I use the Separating Axis Theorem(SAT).

Essentially, the main principle behind the Separating Axis Theorem is as follows. If a line/axis exists such that the projections of two polygons onto this line don't overlap, then the polygons don't collide.

For each edge of both polygons, I computed the normal vector to that edge. Then, I projected each polygon onto that normal vector. If there was a gap in the projections(i.e. there was no overlap in the projections), then I could conclude that there was NO intersection between the two polygons. Else, if there was no such gap, then there was an intersection between the two polygons.

2.1.3 Alternative Optimized Approach

I implemented a third optimized approach which was to use Minkowski Differences to detect collisions. Given two polygons P and Q, they collide iff their Minkowski Difference contains the origin.

To compute the Minkowski Difference of polygons P and Q, the first thing I did was multiply the vertices of Polygon Q by -1 to get $-Q$. Then, I would essentially compute the Minkowski Sum of P and $-Q$.

To do this, I initialized a pointer to the vertex with the lowest y coordinate for P and $-Q$ respectively. Then, for each edge from P and $-Q$ that I was on, I would compare their polar angles(i.e. angles with respect to the horizontal, positive x axis). The edge that had the lower polar angle would be added to my new polygon. I would keep iterating through this process until all edges from both polygons were added to the new polygon. Throughout this process, the pointers would simply be incremented so I could keep track of which edge I am adding to my new polygon!

Once my new polygon was computed, it represented the Minkowski Difference! Now, I just had to determine if the point $(0, 0)$ was contained within this new polygon. To do this, I established a new polar coordinate system where the reference point was the first vertex in the numpy array for the new polygon. Let's call this reference point V. Then, I calculated the angle of each of the other vertices with respect to V, essentially constructing a polar coordinate system with respect to V. I made sure to go in counterclockwise order so that the polar angles I calculated would be in increasing order. This essentially splits up the polygon into "sections" in the polar coordinate system. Finally, I calculated the polar angle of $(0, 0)$ with respect to V. Using binary search, since our polar angles are in increasing order, I determined which "section" $(0, 0)$ belonged to. If it didn't belong to any section, obviously it wasn't contained in the new polygon. However, if it did belong to any of the sections, I had to check the distance between V and $(0, 0)$ to make sure that it belonged within the polygon. If line segment from V to $(0, 0)$ didn't intersect the polygon or $(0, 0)$ was on one of the edges of the polygon, then I knew that $(0, 0)$ was contained within the polygon. Else, $(0, 0)$ was not contained within the polygon.

2.1.4 Runtime Comparisons

Scene #	Naive Method	Bounding Box + SAT	Minkowski Difference
1	0.0008450031	0.000544428825378418	0.0007897854
2	0.0048280478	0.00035381317138671875	0.0023546457
3	0.0029128313	0.0014295339584350585	0.0024139881
4	0.0026724100	0.000854659080505371	0.0028937340
5	0.0011903286	0.00010688304901123047	0.0010263920
6	0.0106782198	0.0018508195877075194	0.0079920769
7	0.0032384157	0.0001322031021118164	0.0012509346
8	0.0073652029	0.00230255126953125	0.0050377607

From this table of runtime comparisons, we can observe that the Minkowski Difference algorithm performs better than the Naive Method for 7 out of 8 of the scenes. We can observe that the Bounding Box + SAT Algorithm performed better than the Naive Method for all 8 scenes. Furthermore, the Bounding Box + SAT Algorithm performed better than the Minkowski Difference for all 8 scenes.

2.2 Polygon Scenes With Collision Detection

Note: To generate the below scenes, I used the collision detection algorithms that we have implemented in the corresponding python files for Problem 2. Refer to the README.md for more information about where to locate these python files.

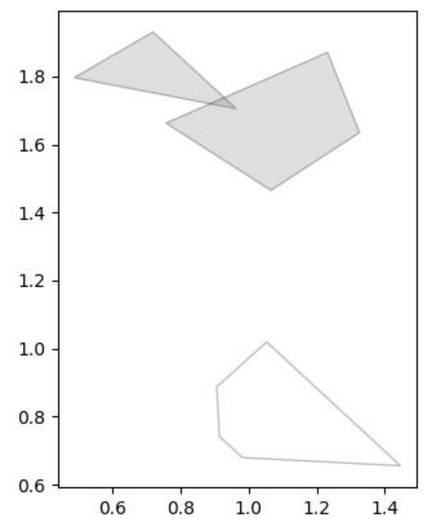


Figure 6: First Scene for Problem 2

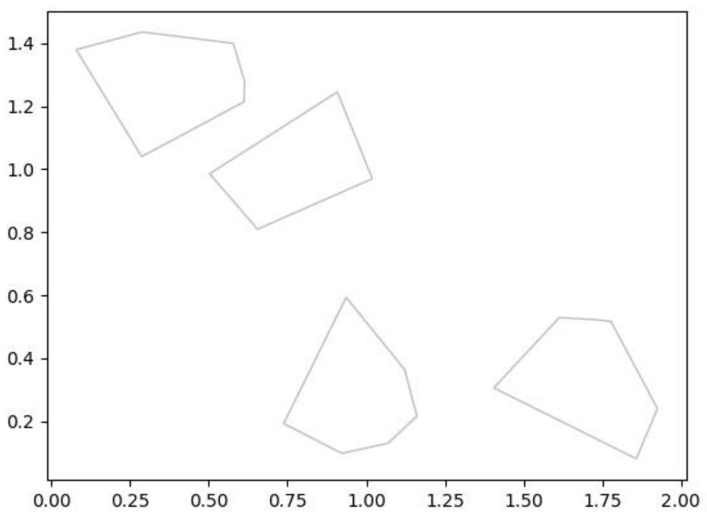


Figure 7: Second Scene for Problem 2

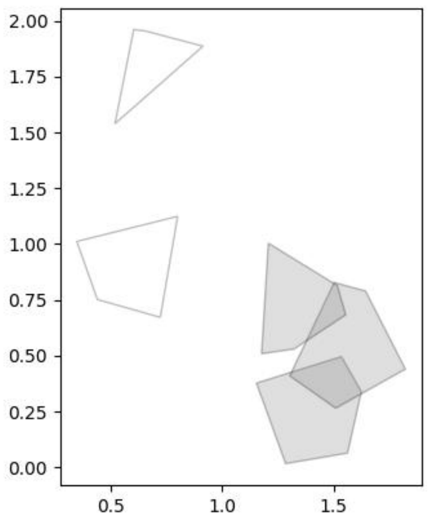


Figure 8: Third Scene for Problem 2

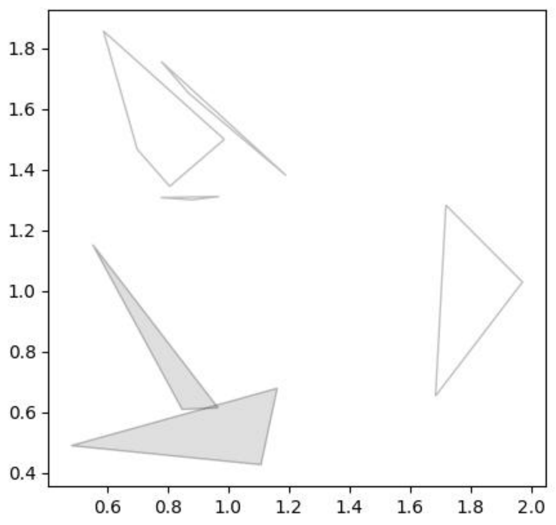


Figure 9: Fourth Scene for Problem 2

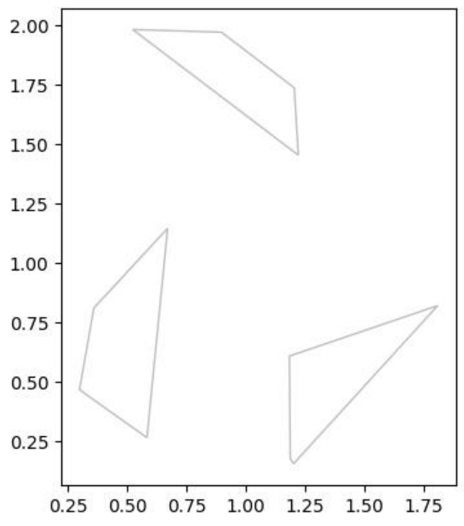


Figure 10: Fifth Scene for Problem 2

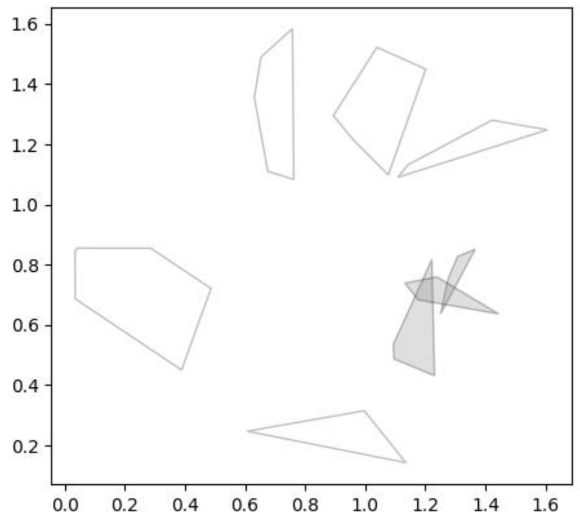


Figure 11: Sixth Scene for Problem 2

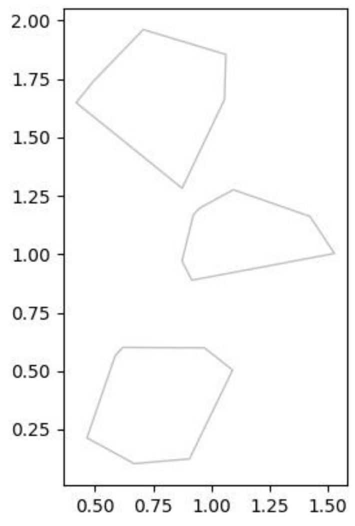


Figure 12: Seventh Scene for Problem 2

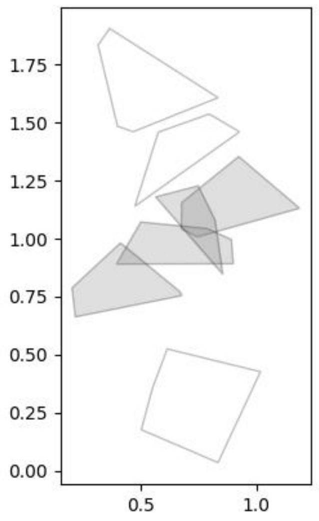


Figure 13: Eighth Scene for Problem 2

3 Collision-free Navigation for a 2D rigid body

3.1 Instructions For Running Program

All the logic for Problem 3 is contained in the file 2d_rigid_body.py
To run the animation of the 2D Rigid Body in an environment with Polygonal Obstacles, please do as such:
Run the following from the terminal, "python3 2d_rigid_body.py –function Rigid"

To run the logic for Generating the Free Configuration Space, please do as such:
Run the following from the terminal, "python3 2d_rigid_body.py –function Minkowski"

3.2 Rigid Body Animations

- Left Arrow: Move Rigid Body backward along the long axis of the rigid body
- Right Arrow: Move Rigid Body forward along the long axis of the rigid body
- Up Arrow: Rotate Rigid Body counterclockwise around its center
- Down Arrow: Rotate Rigid Body clockwise around its center

3.3 Plots of Free Configuration Space for Scene 1

Note to Grader: All of the following plots of the free configuration space were generated by using the Minkowski Difference Algorithm. Although the assignment stated to use the Minkowski Sum Algorithm, an important observation to make here is that our rectangular rigid body is symmetrical. Hence, using the Minkowski Sum or Minkowski Difference Algorithm are both valid approaches to computing the free configuration space!

When visualizing the free configuration space, the red polygons represent the original polygonal obstacles in the original workspace. For each red polygonal obstacle, the surrounding green boundaries represent the representation of this obstacle in the Configuration Space.

3.3.1 Robot is Oriented at a 0 Degree Angle

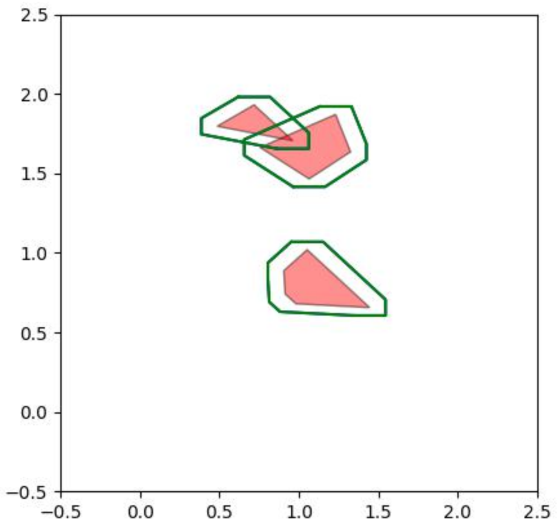


Figure 14: 0 Degree Orientation for Robot

3.3.2 Robot is Oriented at a 45 Degree Angle

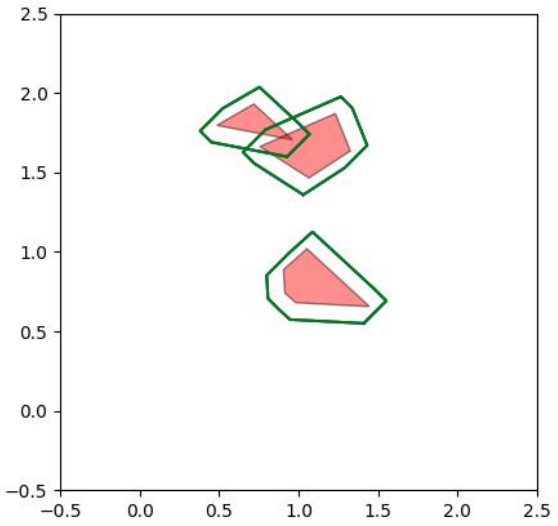


Figure 15: 45 Degree Orientation for Robot

3.3.3 Robot is Oriented at a 90 Degree Angle

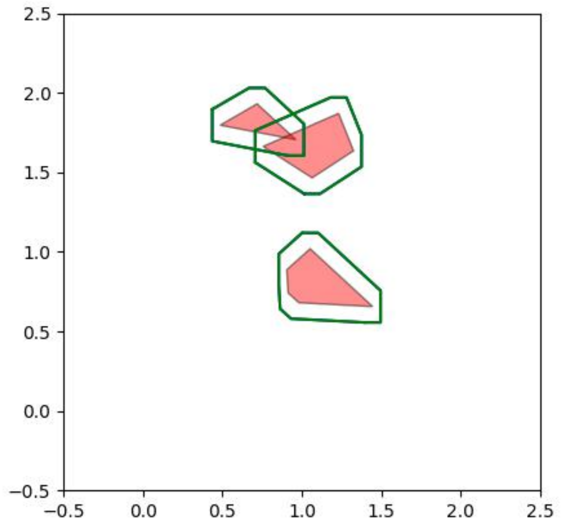


Figure 16: 90 Degree Orientation for Robot

3.4 Plots of Free Configuration Space for Scene 2

3.4.1 Robot is Oriented at a 0 Degree Angle

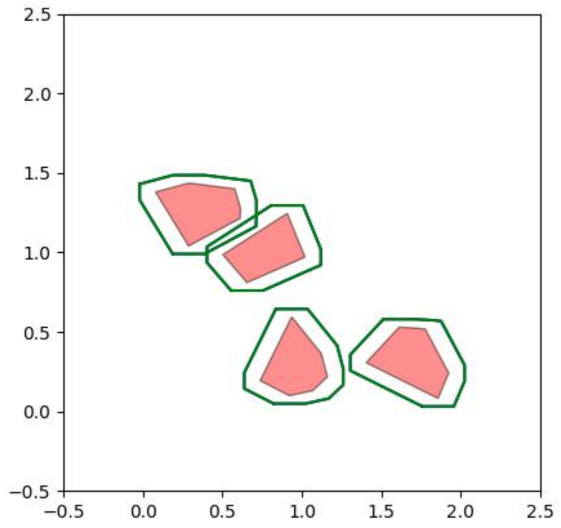


Figure 17: 0 Degree Orientation for Robot

3.4.2 Robot is Oriented at a 45 Degree Angle

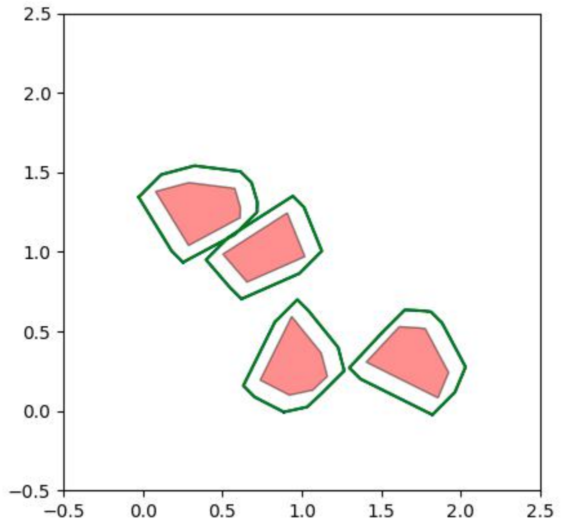


Figure 18: 45 Degree Orientation for Robot

3.4.3 Robot is Oriented at a 90 Degree Angle

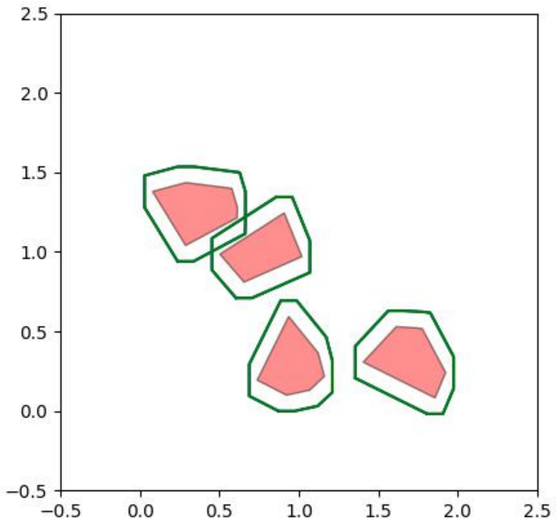


Figure 19: 90 Degree Orientation for Robot

3.5 Plots of Free Configuration Space for Scene 3

3.5.1 Robot is Oriented at a 0 Degree Angle

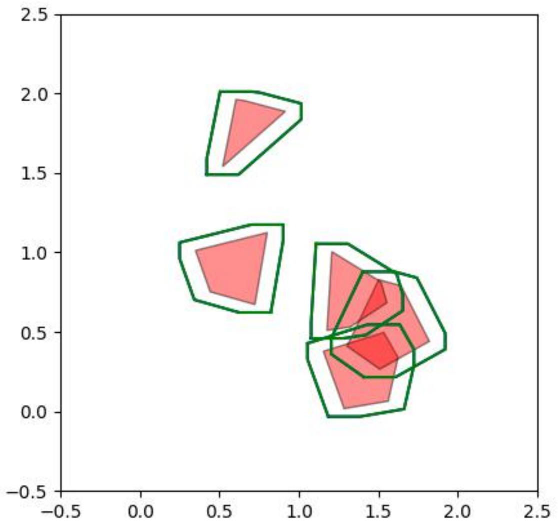


Figure 20: 0 Degree Orientation for Robot

3.5.2 Robot is Oriented at a 45 Degree Angle

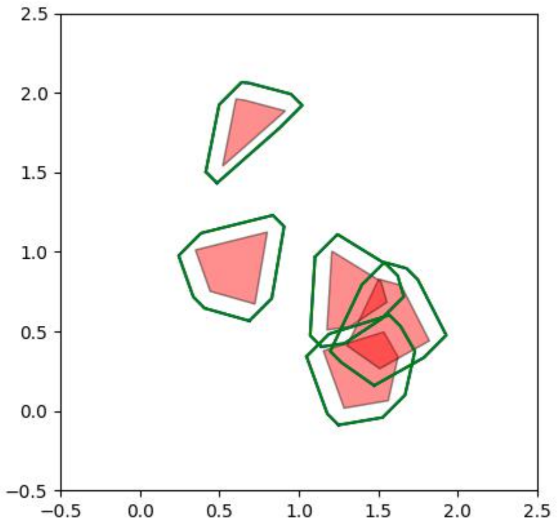


Figure 21: 45 Degree Orientation for Robot

3.5.3 Robot is Oriented at a 90 Degree Angle

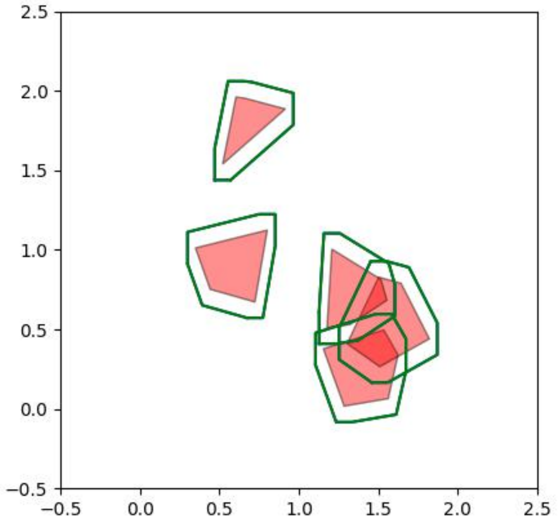


Figure 22: 90 Degree Orientation for Robot

3.6 Plots of Free Configuration Space for Scene 4

3.6.1 Robot is Oriented at a 0 Degree Angle

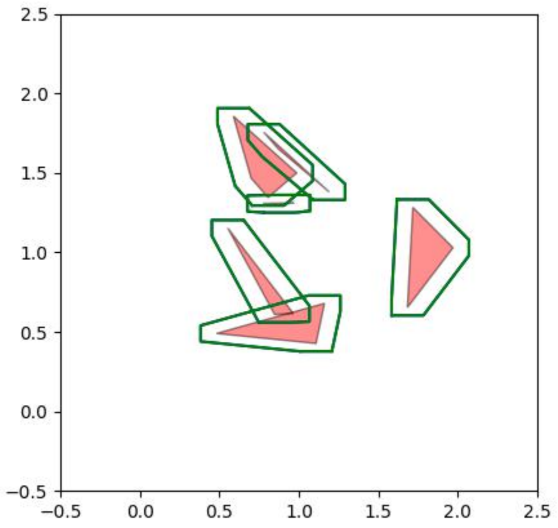


Figure 23: 0 Degree Orientation for Robot

3.6.2 Robot is Oriented at a 45 Degree Angle

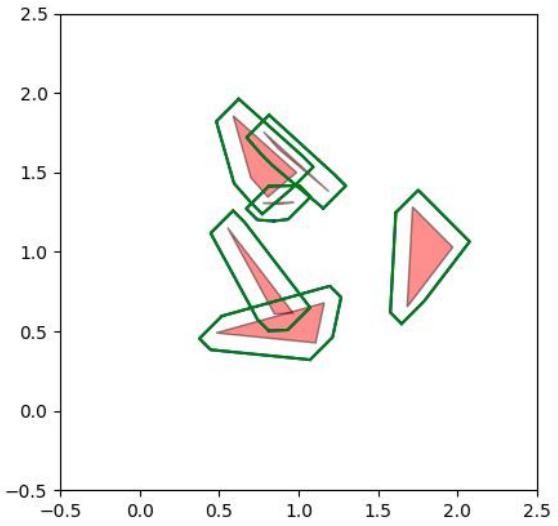


Figure 24: 45 Degree Orientation for Robot

3.6.3 Robot is Oriented at a 90 Degree Angle

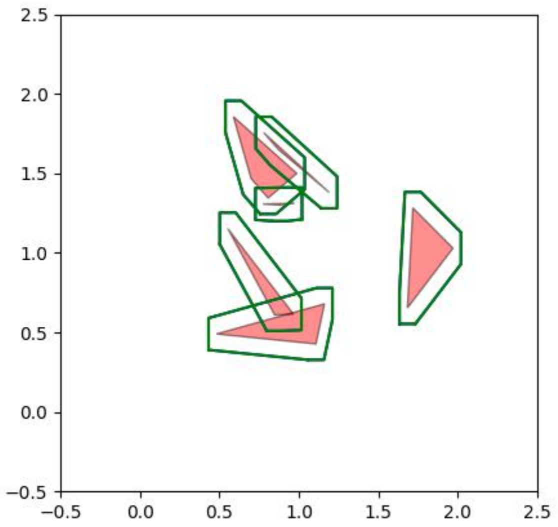


Figure 25: 90 Degree Orientation for Robot

3.7 Plots of Free Configuration Space for Scene 5

3.7.1 Robot is Oriented at a 0 Degree Angle

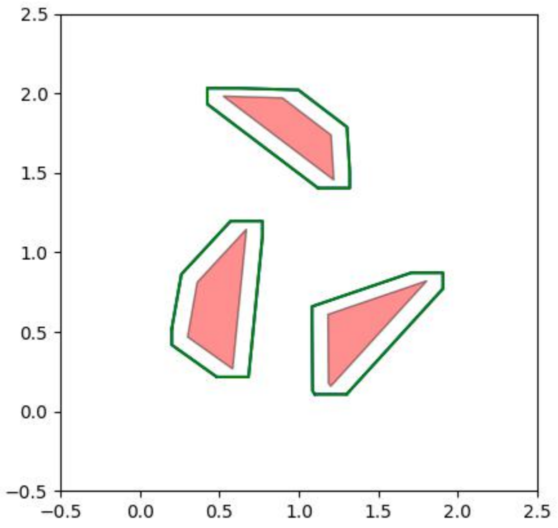


Figure 26: 0 Degree Orientation for Robot

3.7.2 Robot is Oriented at a 45 Degree Angle

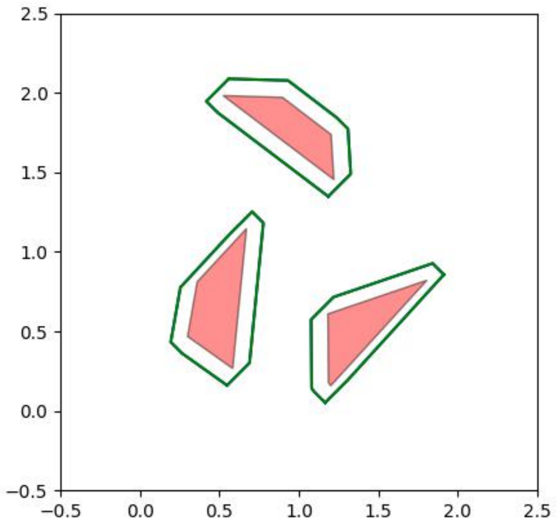


Figure 27: 45 Degree Orientation for Robot

3.7.3 Robot is Oriented at a 90 Degree Angle

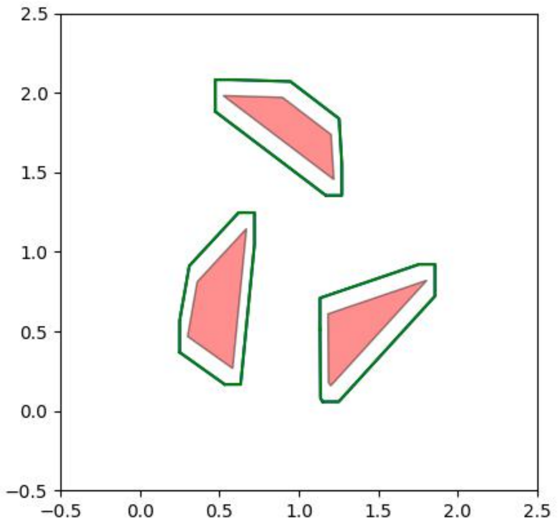


Figure 28: 90 Degree Orientation for Robot

3.8 Plots of Free Configuration Space for Scene 6

3.8.1 Robot is Oriented at a 0 Degree Angle

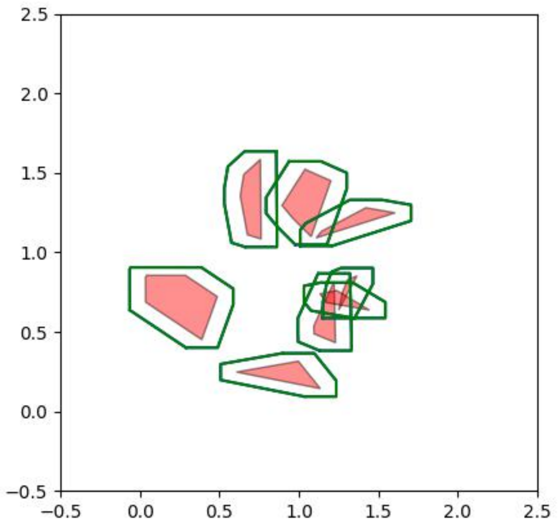


Figure 29: 0 Degree Orientation for Robot

3.8.2 Robot is Oriented at a 45 Degree Angle

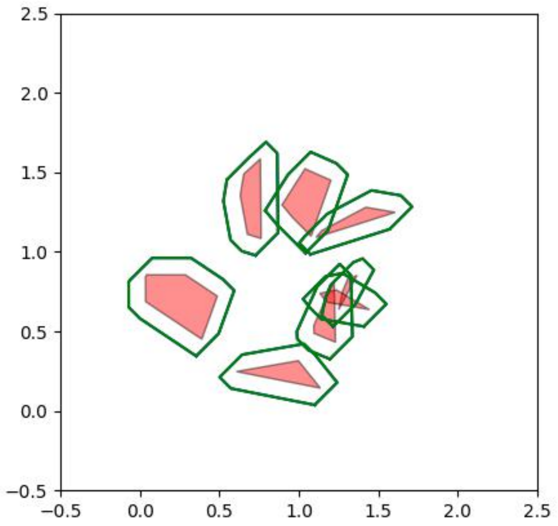


Figure 30: 45 Degree Orientation for Robot

3.8.3 Robot is Oriented at a 90 Degree Angle

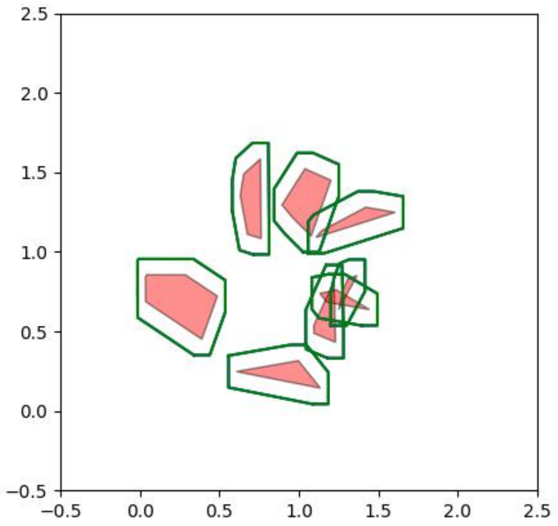


Figure 31: 90 Degree Orientation for Robot

3.9 Plots of Free Configuration Space for Scene 7

3.9.1 Robot is Oriented at a 0 Degree Angle

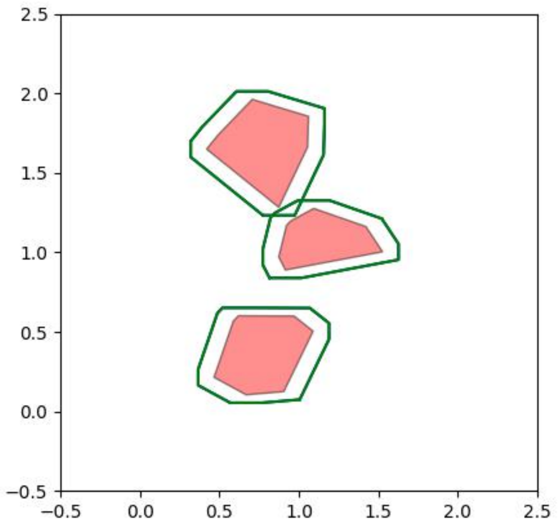


Figure 32: 0 Degree Orientation for Robot

3.9.2 Robot is Oriented at a 45 Degree Angle

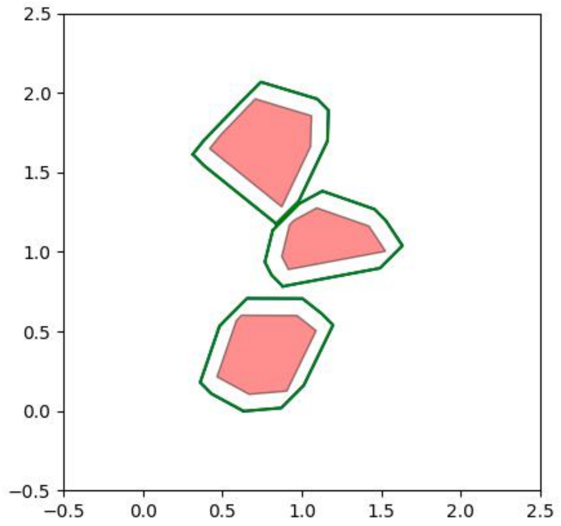


Figure 33: 45 Degree Orientation for Robot

3.9.3 Robot is Oriented at a 90 Degree Angle

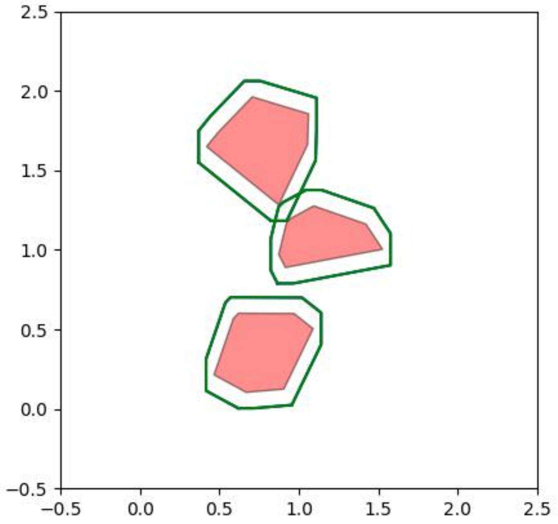


Figure 34: 90 Degree Orientation for Robot

3.10 Plots of Free Configuration Space for Scene 8

3.10.1 Robot is Oriented at a 0 Degree Angle

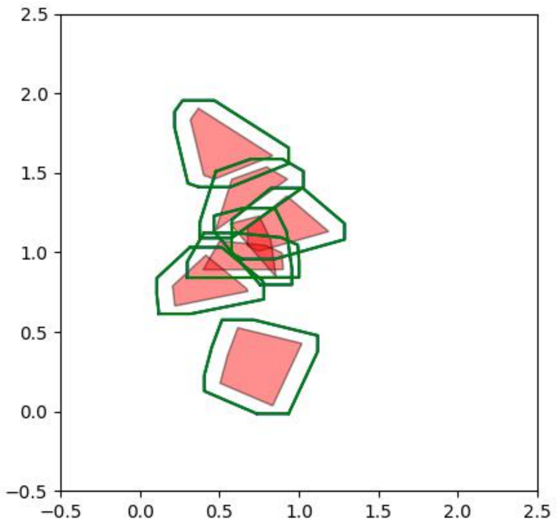


Figure 35: 0 Degree Orientation for Robot

3.10.2 Robot is Oriented at a 45 Degree Angle

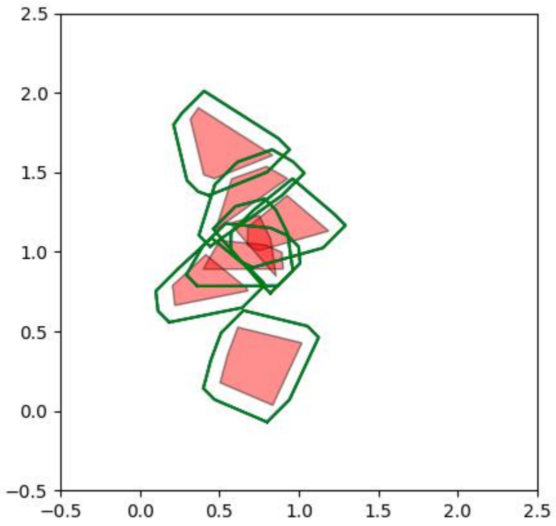


Figure 36: 45 Degree Orientation for Robot

3.10.3 Robot is Oriented at a 90 Degree Angle

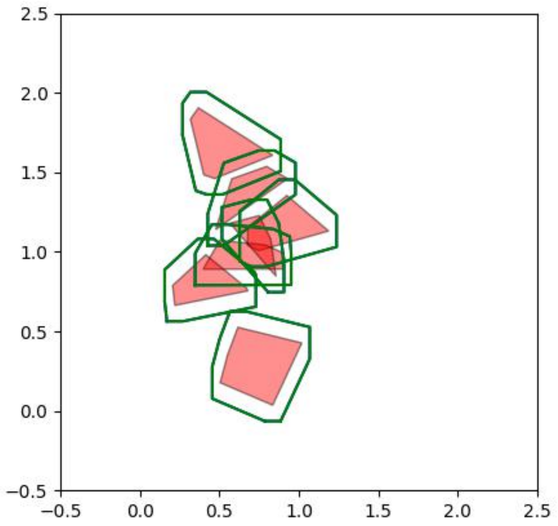


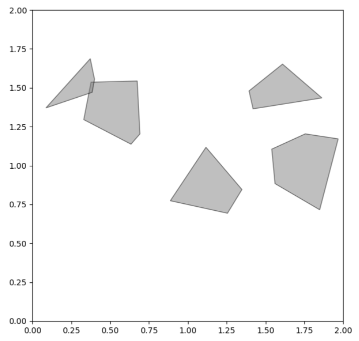
Figure 37: 90 Degree Orientation for Robot

4 Collision-free Movement of a Planar Arm

4.1 Controlling Planar Arm

To control the first joint, the up key was for increments and the down key was for decrements. To control the second joint, the right key was for increments and the left key was decrements. The step size was $\pi/30$.

4.2 Visualization of Scenes



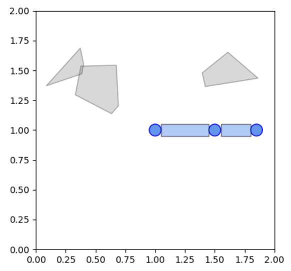


Figure 38: Polygons that collide with arm at start are removed

4.3 Collision-Free Configuration Space

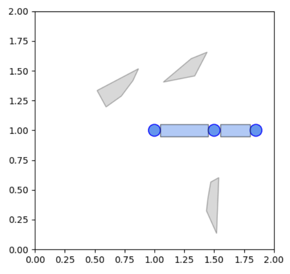


Figure 39: Arm collision-free

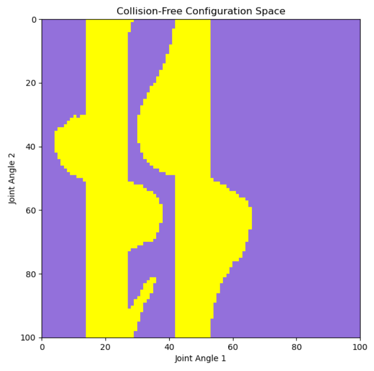


Figure 40: Configuration Space of fig.15