

Five Interactive Online Demonstrations
for Robot Motion Planning

by
Shreyas Poyrekar

A submitted to the Department of Department of Electrical and Computer
Engineering,
College of Engineering
in partial fulfillment of the requirements for the degree of

Master of Science
in Electrical Engineering

Chair of Committee: Dr. Aaron T. Becker
Committee Member: Dr. Mequanint Moges
Committee Member: Dr. Hung “Harry” Le

University of Houston
May 2020

Copyright 2020, Shreyas Poyrekar

ACKNOWLEDGMENTS

I express my special thanks to my guide Dr. Aaron T. Becker who gave me this opportunity to learn and utilize my skills of *Mathematica* to generate online demonstrations hope would be useful learning resources. I thank Arifa Sultana for giving her valuable inputs while developing them.

ABSTRACT

This thesis presents five interactive online demonstrations for learning and teaching robot motion-planning concepts. These include four published Wolfram Demonstration projects built for this Master of Science program, and an additional demonstration that is under review.

These Demonstrations illustrate robotic motion planning algorithms from S. M. LaValle’s classic book “Planning Algorithms”, namely, *Minkowski Sum of Convex Polygons*, *Visibility Polygon*, *Art Gallery Problem* and *Motion Planning Demonstrations*. The aim of using *Mathematica* to implement these demonstrations is to provide curated, online interactive tools that can be used worldwide to understand and teach concepts from robotics and engineering.

By generating these demonstrations, a useful learning resource is made available in the robotic field.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	iii
ABSTRACT	iv
LIST OF FIGURES	vi
1 INTRODUCTION	1
2 RELATED WORK	3
3 MINKOWSKI SUM OF CONVEX ROBOT AND OBSTACLE	5
3.1 Background	5
3.2 Algorithm	5
3.3 How to Use	8
3.4 Screenshots	10
4 VISIBILITY REGION OF A POLYGON	15
4.1 Background	15
4.2 Algorithm	15
4.3 How to Use	17
4.4 Screenshots	17
5 ART GALLERY PROBLEM	22
5.1 Background	22
5.2 Algorithm	22
5.3 How to Use	27
5.4 Screenshots	27
6 BASE CONVERSIONS FROM BASE 2 THROUGH 100 USING RADIX POINTS	37
6.1 Background	37
6.2 Algorithm	37
6.3 How to Use	41
6.4 Screenshots	42
7 MOTION PLANNING FOR TRANSLATING POLYGONAL ROBOT	45
7.1 Background	45
7.2 Algorithm	46
7.3 How to Use	48
7.4 Screenshots	50
8 CONCLUSION	57
BIBLIOGRAPHY	58

LIST OF FIGURES

1	Minkowski Sum of Convex Sum: “static” tab.	8
2	Minkowski Sum of Convex Sum: “move around Minkowski” tab. . .	9
3	Minkowski Sum of Convex Sum: Regular pentagon robot and obstacle.	10
4	Minkowski Sum of Convex Sum: Irregular convex robot and obstacle example 1 in “static” tab.	11
5	Minkowski Sum of Convex Sum: Irregular convex robot and obstacle example 1 in “move around minkowski” tab	12
6	Minkowski Sum of Convex Sum: Line robot and polygon obstacle example 2 in “static” tab.	13
7	Minkowski Sum of Convex Sum: Line robot and polygon obstacle example 2 in “move around minkowski” tab.	14
8	Visibility Region of a Polygon: Example 1	18
9	Visibility Region of a Polygon: Example 2	19
10	Visibility Region of a Polygon: Example 2 continued	20
11	Visibility Region of a Polygon: Example 2 completed algorithm . . .	21
12	Art Gallery Problem: Cubicle environment.	28
13	Art Gallery Problem: Irregular environment.	29
14	Art Gallery Problem: Movable obstacles environment.	30
15	Art Gallery Problem: Cubicle environment with 6 guards.	31
16	Art Gallery Problem: Irregular environment with 6 guards.	32
17	Art Gallery Problem: Irregular environment with two guards.	33
18	Art Gallery Problem: Movable obstacles environment with one guard inside the obstacle.	34
19	Art Gallery Problem: Cubicle obstacles environment with eight guards to cover the outside of the polygon.	35
20	Art Gallery Problem: Movable obstacles environment with seven guards.	36
21	Base Conversion: Input 228 in base 7 output in base 2.	41
22	Base Conversion: Input 9775.445 in base 100 output in base 36. . . .	42
23	Base Conversion: Input 0 in base 36 output in base 2.	43
24	Base Conversion: Input 100 in base 10 output in base 60.	44
25	Motion Planning: “Work space”	49
26	Motion Planning: “Configuration Space”	50
27	Motion Planning: Example 1 shows the work space of a triangle robot in a square boundary environment.	51
28	Motion Planning: Configuration space for Example 1.	52
29	Motion Planning: Example 2 shows the work space of a square robot in a triangular boundary environment.	53
30	Motion Planning: Configuration space for Example 2.	54
31	Motion Planning: Example 3 shows the work space of a square robot in a pentagonal boundary environment.	55
32	Motion Planning: Configuration space for Example 3.	56

1 Introduction

This thesis presents and explains five *Mathematica* demonstrations built to explain and explore robotics concepts [13, 10, 11, 12]. Each demonstration is a separate chapter that goes through the background of each demonstration; the algorithm is explained with short snippets of *Mathematica* code and provides information and description on how to use the demonstrations and include example screenshots that cover some of the boundary cases and conditions. Each of these demonstrations is useful in different aspects of robot motion planning.

Mathematica is a mathematical programming language used for simulation and visualization of engineering concepts. These demonstrations are online on the Wolfram *Mathematica* Demonstration website, <https://demonstrations.wolfram.com/>. It is an open-source resource sharing website to create and share such exciting demonstrations. This thesis should be read by those who wish to obtain detailed information on each of the demonstrations. Also, those who want to generate such demonstrations can find some valuable insights into building a demonstration.

Chapter 1 is the introduction, Chapter 2 informs about the related work in building Wolfram *Mathematica* demonstrations for education and visualization of the concepts. Chapter 3 presents the “*Minkowski sum of Convex Polygons*” demonstration, which is a visualization of the algorithm. The Minkowski sum is crucial in motion planning to find the optimal path for polygonal obstacles and robots. Chapter 4 explains the “*Visibility Region of a Polygon*” demonstration. The visibility region of a polygon is a sophisticated algorithm, and the demonstration serves to visualize how the visible area is generated. Chapter 5 shows the “*Art Gallery Problem*” demonstration, which uses the visibility region of a polygon algorithm to envision the number of guards needed to simultaneously see every position in a 2D polygonal art gallery. Chapter 6 describes the “*Base Conversions from Base 2 through 100 Using Radix Points*” demonstration, which is can convert a number from any base to any base (between 2 and 100) and can be used as a calculator to

find the conversions up to base 100. Chapter 7 is a motion planning demonstration “*Motion Planning for Translating Polygonal Robot*”, which describes how the shortest possible path for a polygonal robot is obtained in an environment with polygonal obstacles. Chapter 8 provides the conclusion on this thesis.

2 Related Work

The Internet is a good source for education and has many online resources sharing platforms available. The use of online video lectures and tests have increased, along with online degree programs in many fields.

An online demonstration can be a useful tool in illustrating concepts in both online and offline based courses. By interacting with the demos, students can gather valuable insights. As the online industry began, many interactive online games were built and shared. Most of these small applications were Java-based applets that ran at breakneck speed and used 3D hardware acceleration, which made them well-suited for computation-intensive visualization. Today, many of the online-education-based platforms are moving into web-based visualizations as an interactive tool.

“Wolfram Demonstration Project”¹ is such an online, demonstration-sharing, open-source platform that uses dynamic computation to illuminate concepts in science and engineering. It uses the *Mathematica* programming language to generate online demonstrations. There are currently 12,000+ interactive demonstrations available on the website.

Robot motion planning deals with the problem of computing a continuous path that connects the starting configuration of a robot to the goal configuration of the robot while avoiding all collisions with the obstacles in the work space. A simple approach to solving the motion planning is to uniformly sample the configuration space of the workspace, label each sample as ‘in collision’ or ‘free’, and then perform a breadth-first search to find a path that connects the starting configuration to the goal configuration, or determine that no path exists. Such approaches are called *resolution-complete* planners, because they are accurate up to the resolution (or granularity) of the uniform sampling. However these require a number of samples that increases exponentially with the resolution of the grid.

¹<https://demonstrations.wolfram.com/>

There are many grid-based algorithms that can solve low-dimensional path planning problems. Exact approaches do not depend on the resolution. however, exact algorithms have higher computation complexity and can be difficult to understand. Using interactive demonstrations to illustrate the algorithm can be useful in understanding and learning of these algorithms. The “Wolfram Demonstration Project” has many demonstrations available, listed under the robotics section of engineering and technology².

²<https://demonstrations.wolfram.com/topic.html?topic=Robotics&limit=20>

3 Minkowski Sum of Convex Robot and Obstacle

This chapter explains a demonstration built to teach how the *Minkowski sum* of a robot and an obstacle is constructed [12]. In Section 3.1 a brief explanation of *Minkowski sum* is provided. Section 3.2 describes the algorithm used and goes through the parts of the *Mathematica* code. Section 3.3 offers information on how to use the online demonstration, and Section 3.4 includes screenshot examples.

3.1 Background

The *Minkowski sum* was named after German mathematician Hermann Minkowski (1864–1909). Knowing this region simplifies robot motion planning. The *Minkowski sum* is the set of all coordinates in which one polygon overlaps another. The *Minkowski sum* is useful for robot navigation because, once it is computed, the problem of finding a collision-free path for a polygon-shaped robot moving through a work-space filled with obstacles is reduced to finding a path for a point-shaped robot through a configuration space populated by the *Minkowski sums* of the robot and each obstacle [12, 7].

3.2 Algorithm

The *Minkowski sum* is a convolution between the robot and the obstacle. The interior of this region is the set of robot coordinates where the translated robot intersects the obstacle region. The solution provided here is valid for any 2-D world with convex robots and convex obstacles. To handle non-convex polygons, they can be modelled as a union of convex polygons.

The procedure has four steps. First, compute the convex hull of both the robot and the obstacle polygons. The convex hull of polygon is a polygon converted such that no vertex bends inwards or forms a non-convex shape. The convex hull is computed by the Graham Scan Algorithm, which has $O(n \log(n))$ time complexity [4].

The time complexity of the algorithm is not dependent on the size of the hull. The following is the implementation in *Mathematica*. The scan starts from the bottom-most point, then sorts the points based on the angle between the bottom-most point and each other points. After sorting, step through all the remaining points and join all the points which form counterclockwise rotations. The function `clockwiseQ` takes three input points and returns true if the points form a clockwise rotation. `convexHullGrahamScan` takes all the points which form a non-convex polygon and returns a set of points which represent a convex polygon [4, 12].

```

1 clockwiseQ[p1: {x1_, y1_}, p2: {x2_, y2_}, p3: {x3_, y3_}] :=
2   Chop[((x2 - x1)*(y3 - y1) - (y2 - y1)*(x3 - x1))];

```

```

1 convexHullGrahamScan[pts_] :=
2 Module[ {n = Length@pts, sortedP = Sort[pts, #1[[2]] < #2[[2]] &],
3 p, m = 2, i},
4   p = Join[{sortedP[[1]]}, Sort[sortedP[[2 ;; n]], ArcTan[#1[[1]]
5     - sortedP[[1, 1]], #1[[2]] - sortedP[[1, 2]]] < ArcTan[#2[[1]]
6     - sortedP[[1, 1]], #2[[2]] - sortedP[[1, 2]]] &]];
7   For[i = 1, i <= n, i++,
8     While[(clockwiseQ[p[[m - 1]], p[[m]], p[[i]]] <= 0),
9       If[m > 2, m = m - 1, If[i == n, Break[], i += 1]];
10    m += 1; p[[{i, m}]] = p[[{m, i}]];
11    p[[1 ;; m]]]

```

The second step is to obtain the normals for the obstacle edges and the robot edges. The normals for the obstacles points outwards and point inward for the robot. The `normalVector` function returns a normal vector to the polygon side [12].

```

1 normalVector[{x2_, y2_}, {x1_, y1_}] :=
2   Normalize[{-y2 + y1, x2 - x1} ]

```

The third step is to merge the robot and obstacle normal's and sort them by increasing magnitude of their angles. The code below merges the robot and obstacle

normals. The `ArcTan` function is used to get the angle. An association is used to allocate keys to each side and the values are the normals and the angles [12].

```

1 orderOfNormals = Sort[{If[! (#[[1]][[1]] == 0 && #[[1]][[2]] == 0),
2   ArcTan#[[1]][[1]], #[[1]][[2]], None],
3   #[[1]], #[[2]], #[[3]], #[[4]]}
4   & /@ Join [robotAssignedNormal, obstacleAssignedNormal],
5   #1[[1]] < #2[[1]] &];
6 sortedVectors = Values[orderOfNormals];

```

The first point in the *Minkowski* sum is arbitrarily chosen as a point where the centroid of the robot lies at one of the vertex-vertex contacts of the obstacle and robot. The vertex-vertex contact is when either the robot vertex meets the obstacle edge or when the obstacle vertex meets the robot edge. The code below calculates the position of any arbitrary vertex-vertex contact obtained and calculates the point where the *Minkowski* sum starts [12].

```

1 (*Get the Sequence position where \[Alpha] and \[Beta] meet*)
2 pos = SequencePosition[sortedVectors[;; , 4]],
3 {"\[alpha]", "\[Beta]"}][[1, 1]];
4 orderOfNormalsKeys = RotateLeft[Keys@orderOfNormals, pos];
5
6 (*Calculate the Prev Value based on the position*)
7 prev = rc + sortedVectors[[pos + 1]][[5]][[1]] -
8   sortedVectors[[pos]][[5]][[1]];

```

Finally, the Minkowski sum is generated by adding each edge in the order specified by the normal's. Robot edges are added in the clockwise direction, obstacle edges in the counterclockwise direction [12].

```

1 (*Get the Minkowski by adding the vectors in the sorted order*)
2 minkowskiPointswithSide = {Chop[prev = prev - #[[3]], #[[4]]}
3 & /@ RotateLeft[sortedVectors, pos - 1];
4 minkowskiPoints = (#[[1]] & /@ minkowskiPointswithSide);

```

A significant observation is that every edge of the Minkowski sum polygon is a translated edge from either the obstacle or the robot polygon [12].

3.3 How to Use

This demo is available online³. The “View” Option lets you choose between the “Static” and “Move Around Minkowski” Tabs. In the “Static” tab, you can manage the parameters for the robot and obstacle polygons and visualize the calculation of the Minkowski sum. In the “Move Around Minkowski” tab, you can see an animation of the robot centroid moving around the Minkowski sum while avoiding entering the obstacle. In Figure 1 you can see the the “View” option set to “static” with the robot edges set to 3 and the obstacle edges set to 5..

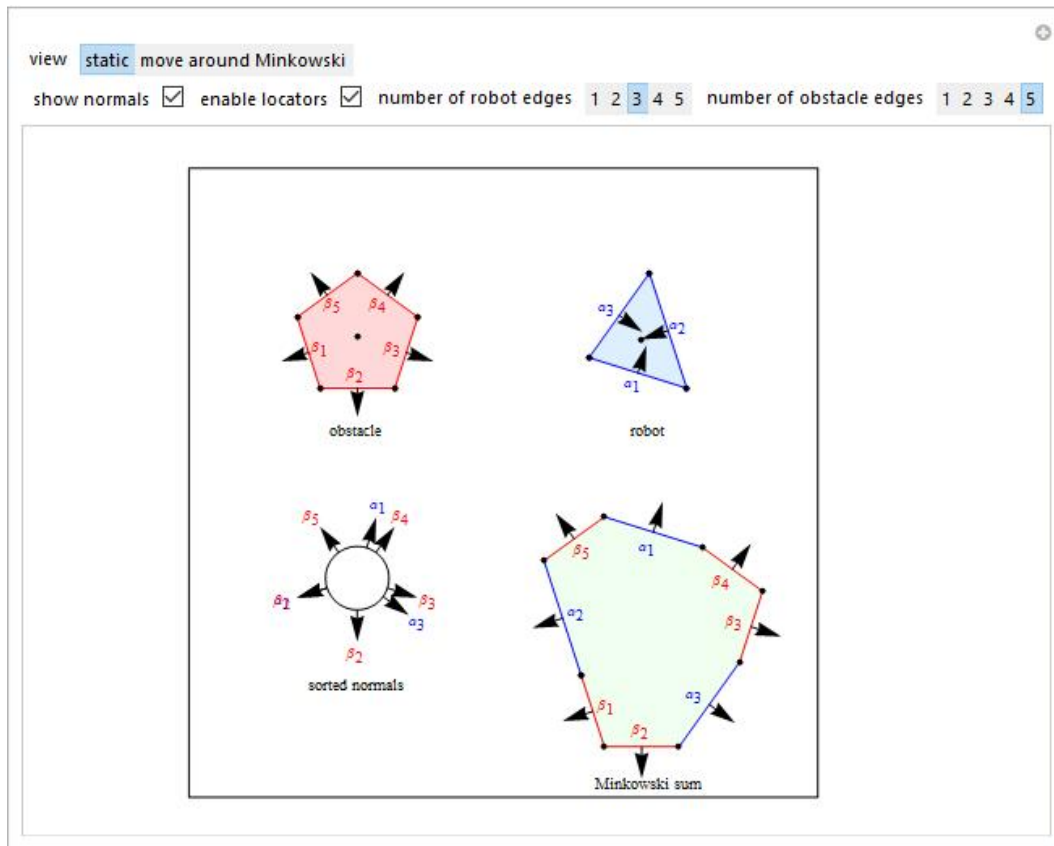


Figure 1: Minkowski Sum of Convex Sum: “static” tab.

In the “Static” tab, you can drag the locators to make any arbitrary two-dimensional robots and obstacles—the restriction to the locators is such that they always form a convex polygon. The sides of the robot and obstacles are labeled as

³demonstrations.wolfram.com/MinkowskiSumOfConvexRobotAndObstacle/

α and β , respectively. The “Show Normals” checkbox enables/disables the normals displayed on the manipulate window. The “Enable Locators” checkbox enables/disables the access to change the shape of the polygon. You can change the robot and obstacle edges from one up to five. In Figure 2 you can see the the “View” option set to “move around minkowski” for the configuration in Figure 1.

The robot is colored blue, and the obstacle is colored red. The Minkowski Sum is colored green with the edges colored red or blue depending on if the polygon edge was taken from the robot or the obstacle. The lower-left image shows the sorted normals. In the “Move Around Minkowski” tab, you can trace the region of the

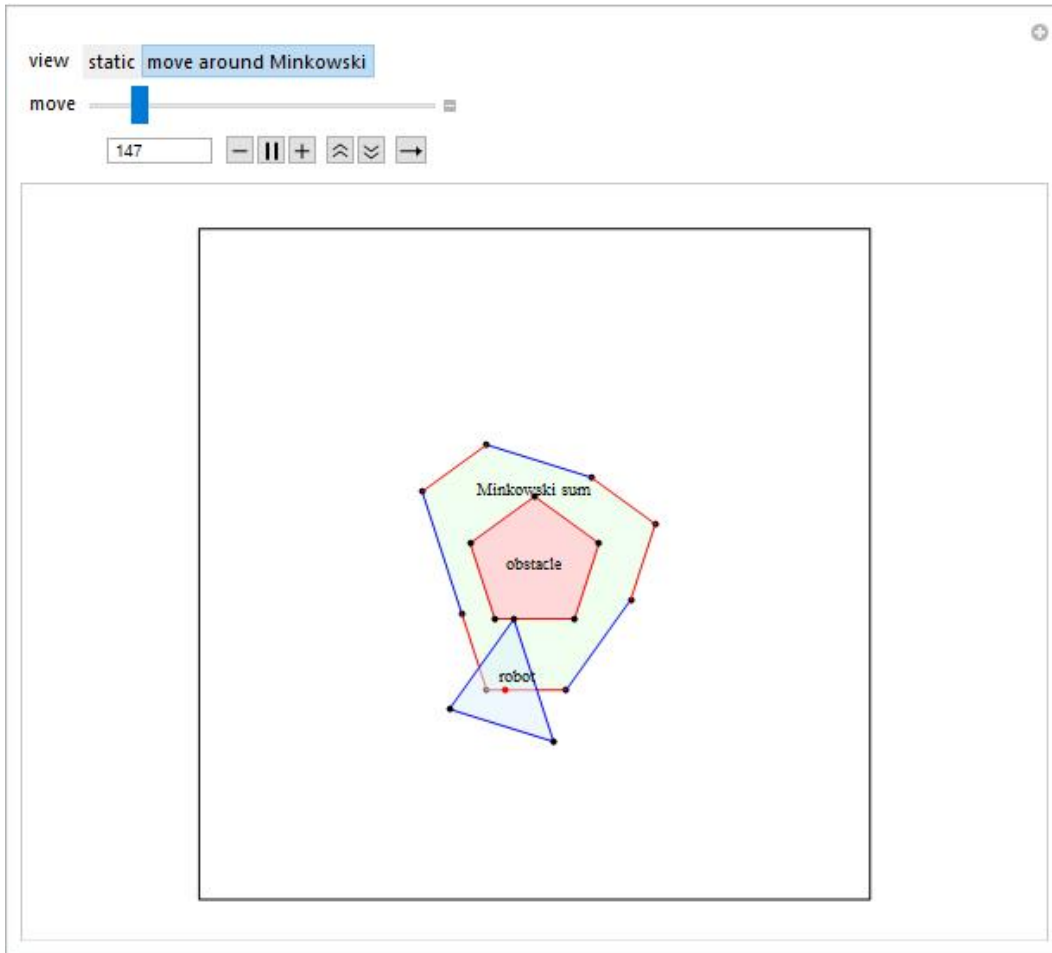


Figure 2: Minkowski Sum of Convex Sum: “move around Minkowski” tab.

Minkowski sum by choosing “move around Minkowski” slider to animate the robot moving around the edge of the obstacle.

3.4 Screenshots

Figure 3 shows a case where the robot and the obstacle are both regular pentagons. Figure 4 and 5 shows the screenshot example 1 for a three edged robot and five edged obstacle. Figure 6 and 7 shows the screenshot example 2 for a line robot and a four-edged obstacle.

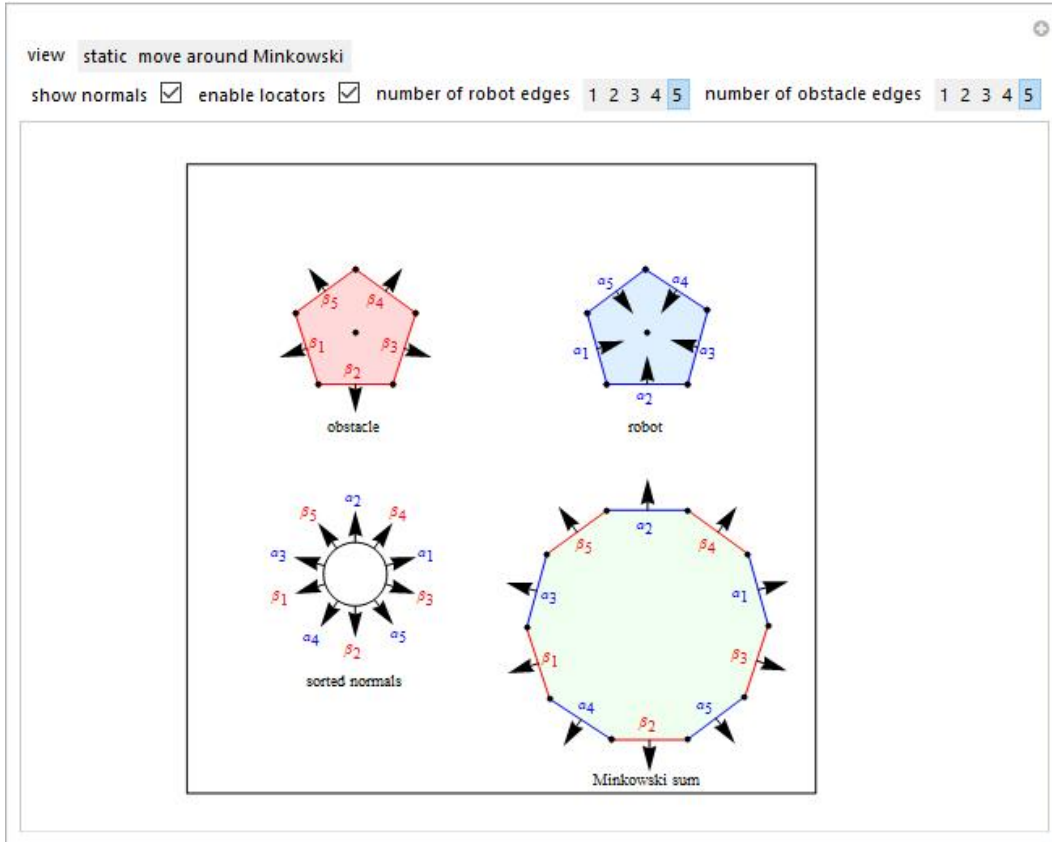


Figure 3: Minkowski Sum of Convex Sum: Regular pentagon robot and obstacle.

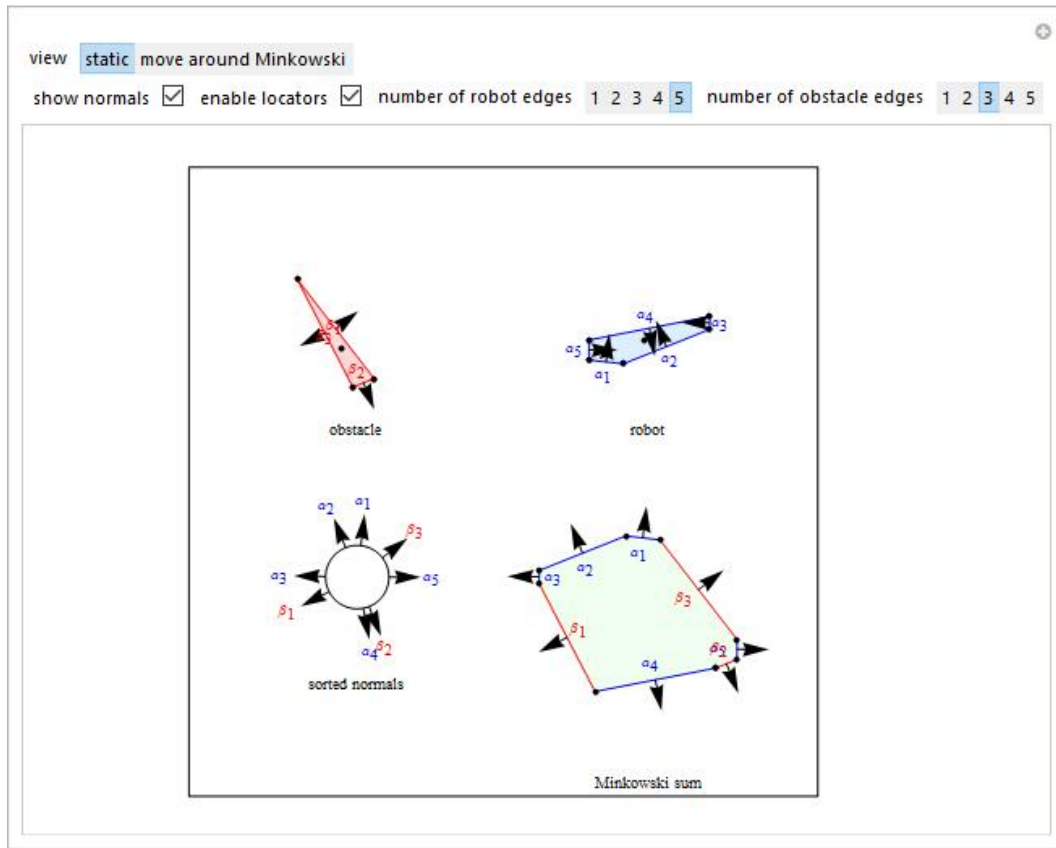


Figure 4: Minkowski Sum of Convex Sum: Irregular convex robot and obstacle example 1 in “static” tab.

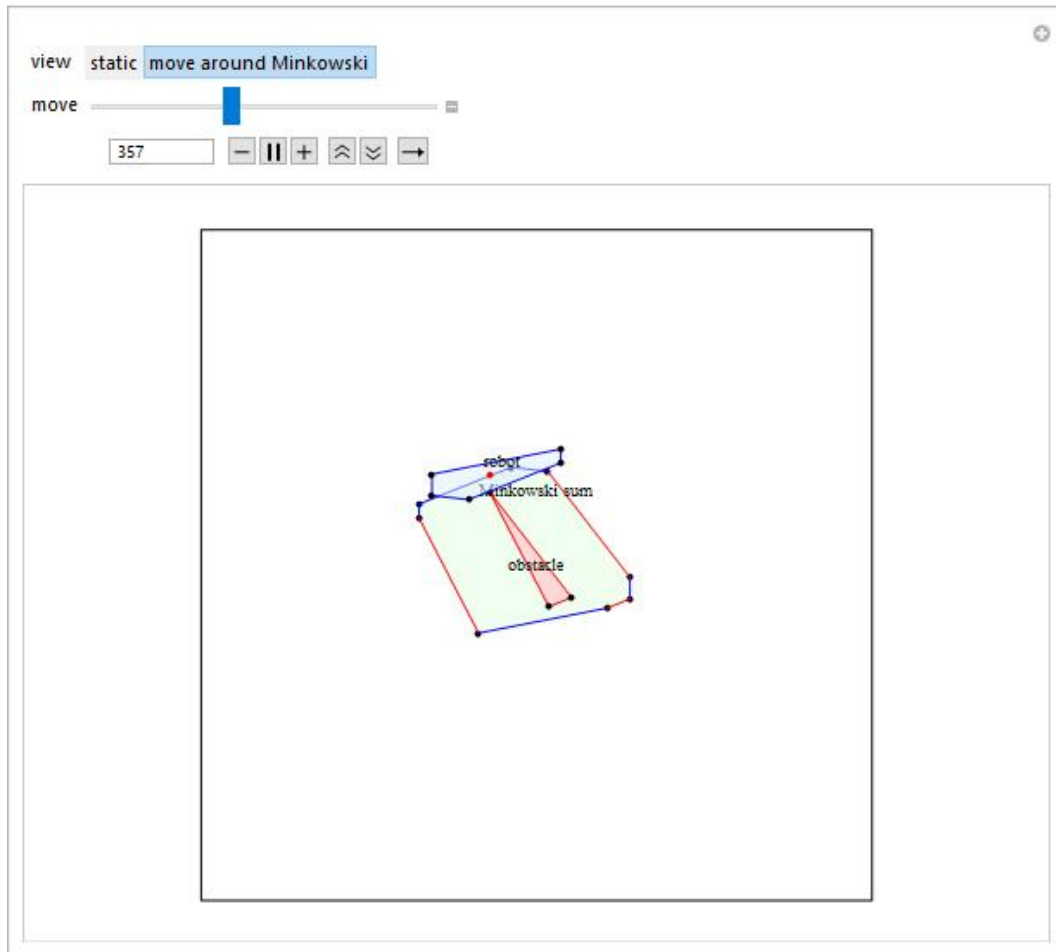


Figure 5: Minkowski Sum of Convex Sum: Irregular convex robot and obstacle example 1 in “move around minkowski” tab

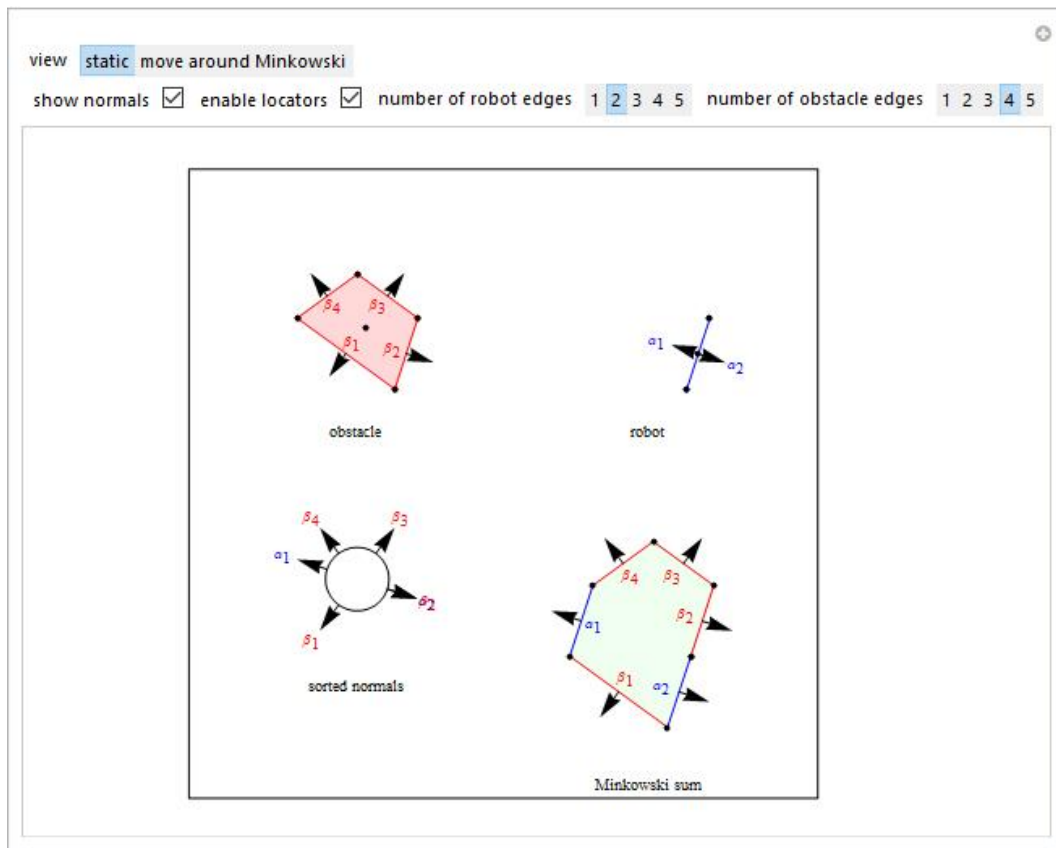


Figure 6: Minkowski Sum of Convex Sum: Line robot and polygon obstacle example 2 in “static” tab.

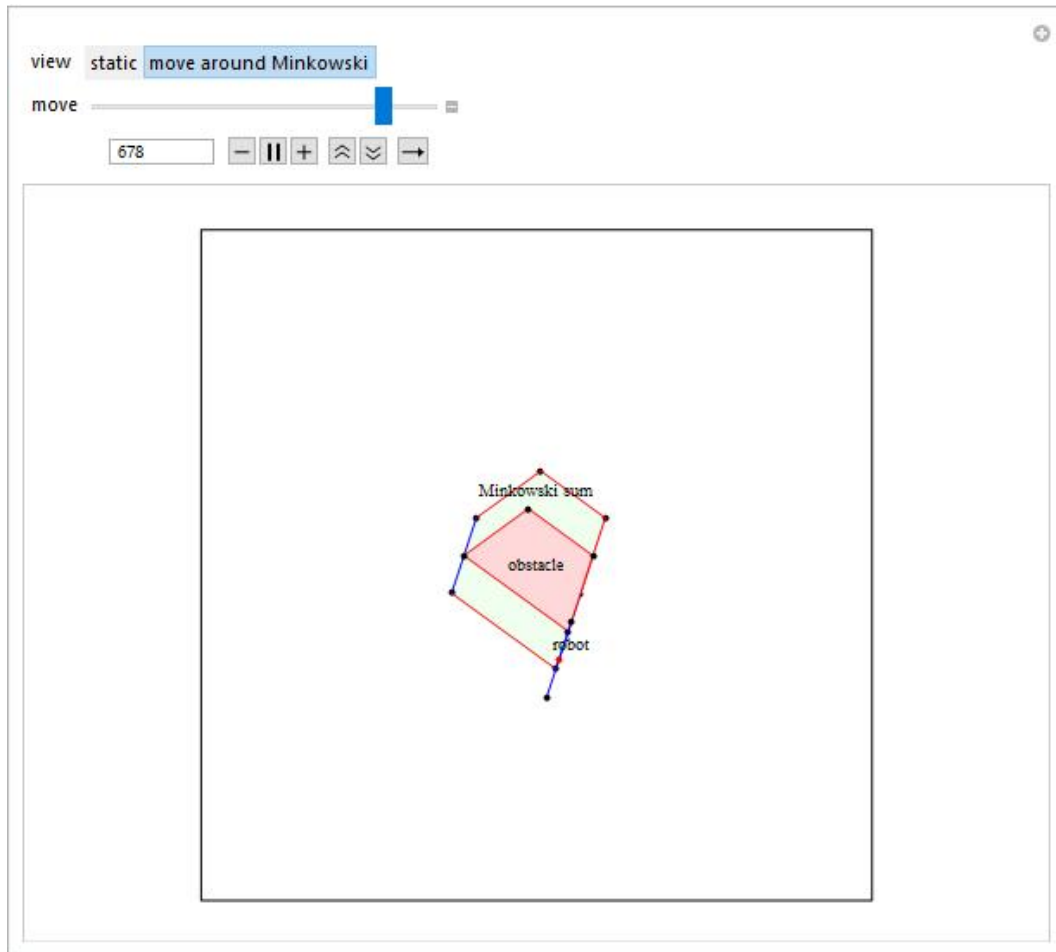


Figure 7: Minkowski Sum of Convex Sum: Line robot and polygon obstacle example 2 in “move around minkowski” tab.

4 Visibility Region of a Polygon

This chapter explains a demonstration built to teach how the visibility region inside a polygon from any point is constructed [13]. In Section 4.1, a brief explanation of the visibility region of a polygon is provided. Section 4.2 describes the algorithm used and goes through the parts of the *Mathematica* code. Section 4.3 offers the information to use the online demonstration, and Section 4.4 includes screenshot examples.

4.1 Background

The parts of a polygon reachable by straight lines from a point without hitting a wall are called the *visibility region*. The visibility region is useful for finding accessible paths. The accessible path is the shortest visible path which a robot can translate. The visibility region is the set of points in a polygon that are visible from a reference point p . Imagine standing at point p holding a flashlight in a building whose walls are the polygon. The visibility region is the region illuminated by your flashlight [2].

Calculating the visibility region is necessary for solving the art gallery problem, a canonical computational geometry problem discussed in Chapter 5. The visibility region is also often used for robotic motion planning (see Chapter 7).

4.2 Algorithm

The visibility region of a polygon from a reference point p can be calculated as described in [2], which goes as follows: First, construct an infinite ray parallel to the x axis, starting from point p . The code below generates a line parallel to the x -axis; in this case, the infinite line is limited up to the workspace boundaries.

```

1  (*get a Line Parallel to x axis*)
2  infiniteLine = {p, {0.1,
3    0} + {Max@#[[1]] & /@ (Join[workspace, complexPoly])), p[[2]]}};

```

Next, find the intersection of all the polygon lines with this infinite ray and add each intersecting line to a list commonly called the J list. Sort the J list by distance from p . The closest line is visible at this intersection point. Next, sort all the polygon vertices by the polar angle from p :

```

1  (*Sort the lines based on the angle w.r.t to point p*)
2  sortedList =
3    Sort[Join[vertexList@(workspace), vertexList@(complexPoly)],
4    angleSortCond[p, #1, #2] &];
5
6  (*get the intersection of lines for the Infinteline*)
7  If[SegmentIntersectionQ[{infiniteLine, #}],
8    AppendTo[jList, Reverse@#] & /@ listofAllLines;
9
10 (*Store the sorted values in jList*)
11 jList = Sort[jList, distSortCond[infiniteLine, #1, #2, p] &];

```

Then step through each vertex in this loop. For each vertex, we must update the J list. Construct a ray to the vertex and add the lines that extend in the clockwise half-plane into the J list and remove the lines that extend in the counterclockwise half-plane from the J list. The current vertex is visible only if it is at the top of the J list. The `getClockwiseAngle` function returns the angle between two vectors and the `leftorRight` returns if point p lies on the right or left side of a line. Using these two functions we can find the lines that extend in the counterclockwise half-plane [13].

```

1  (*clockwise angle between two vectors*)
2  getClockwiseAngle[p1_, p2_, p3_] :=
3    Module[{a = (p3 - p2), b = (p1 - p2)},
4    AppendTo[a, 0]; AppendTo[b, 0];
5    If[(Cross[a, b][[3]]) < 0, Chop@(2 Pi - VectorAngle[a, b]),
6    Chop@VectorAngle[a, b]]

```

```

1  (*returns true if the point p lies on the right side else false*)
2  leftOrRight[a : {x1_, y1_}, b : {x2_, y2_},
3  p : {x_, y_}] := ((x \[Minus] x1) (y2 \[Minus]
4  y1) \[Minus] (y \[Minus] y1) (x2 \[Minus] x1)) >= 0

```

4.3 How to Use

This demo is available online⁴. Figure 9 shows a non-complex polygon with a square boundary and a blue colored reference point p . The visible region from the reference point p is colored light-yellow. A line parallel to the x -axis from the reference point is seen, it is required at the start of the algorithm to get the initial J -list. The edges of the polygons colored in green are the lines in the J -list. The slider “vertices sorted by angle” steps through all the sorted vertices of the polygon. As the slider is moved, the line from the reference point and the current point is updated and drawn as a blue line. The J -list is also updated as you move from one point to another, and the region covered thus far during this angular sweep from 0° is colored dark-yellow. This can be seen in Figure 9, which shows the region included in the visible region at each point. All the visible vertices are colored red. The reference point p is a locator which can be dragged by the use. users can move the reference point inside or outside of the polygon, and can move the slider and observe how the visible region is computed.

4.4 Screenshots

Example 2, pictured in Figure 9-11, shows the computation of the visible region and updated set of lines J -list as the slider is moved from point to point. Figure 11 shows the completed algorithm with the final set of lines in the J -list, which is similar to that at the start of the algorithm.

⁴demonstrations.wolfram.com/VisibilityRegionOfAPolygon/

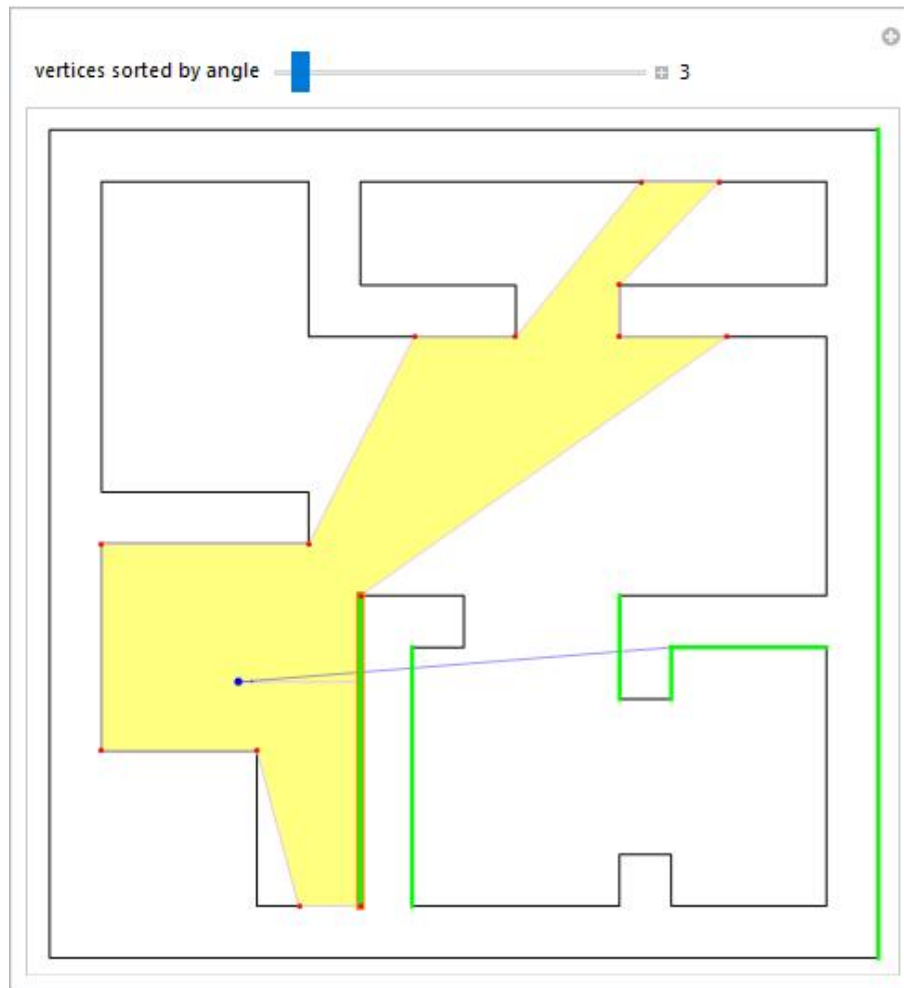


Figure 8: Visibility Region of a Polygon: Example 1

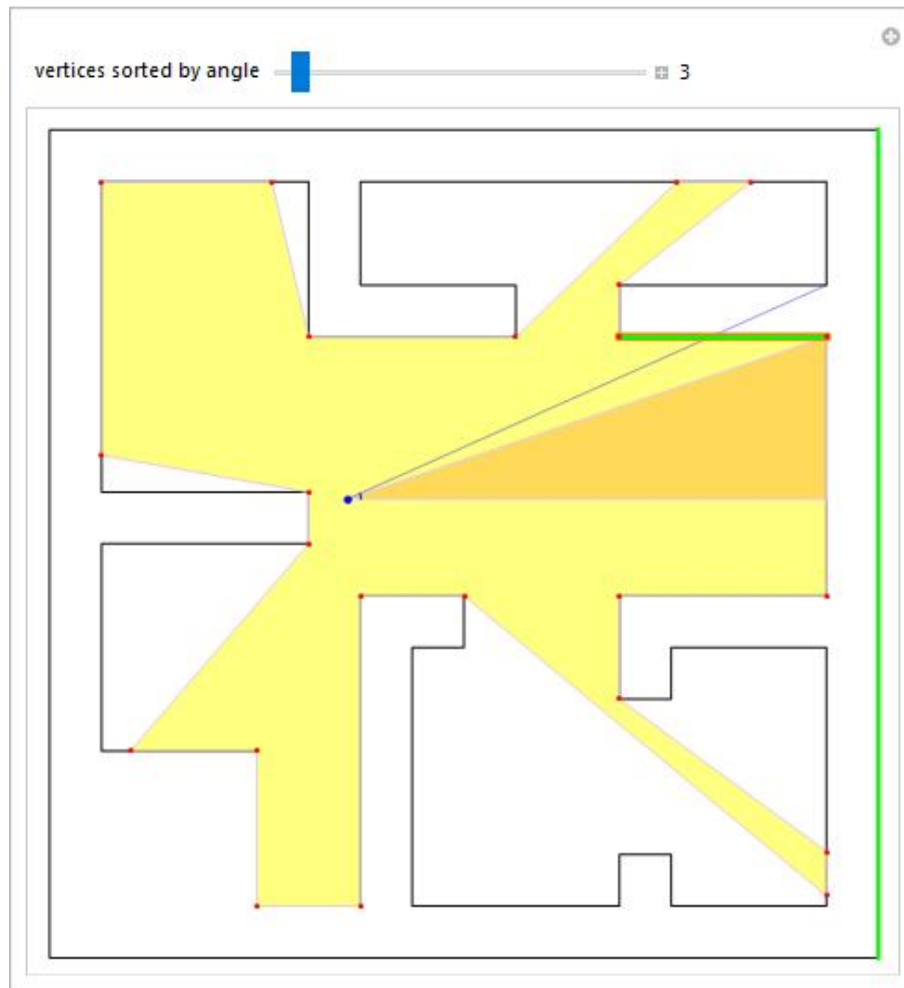


Figure 9: Visibility Region of a Polygon: Example 2

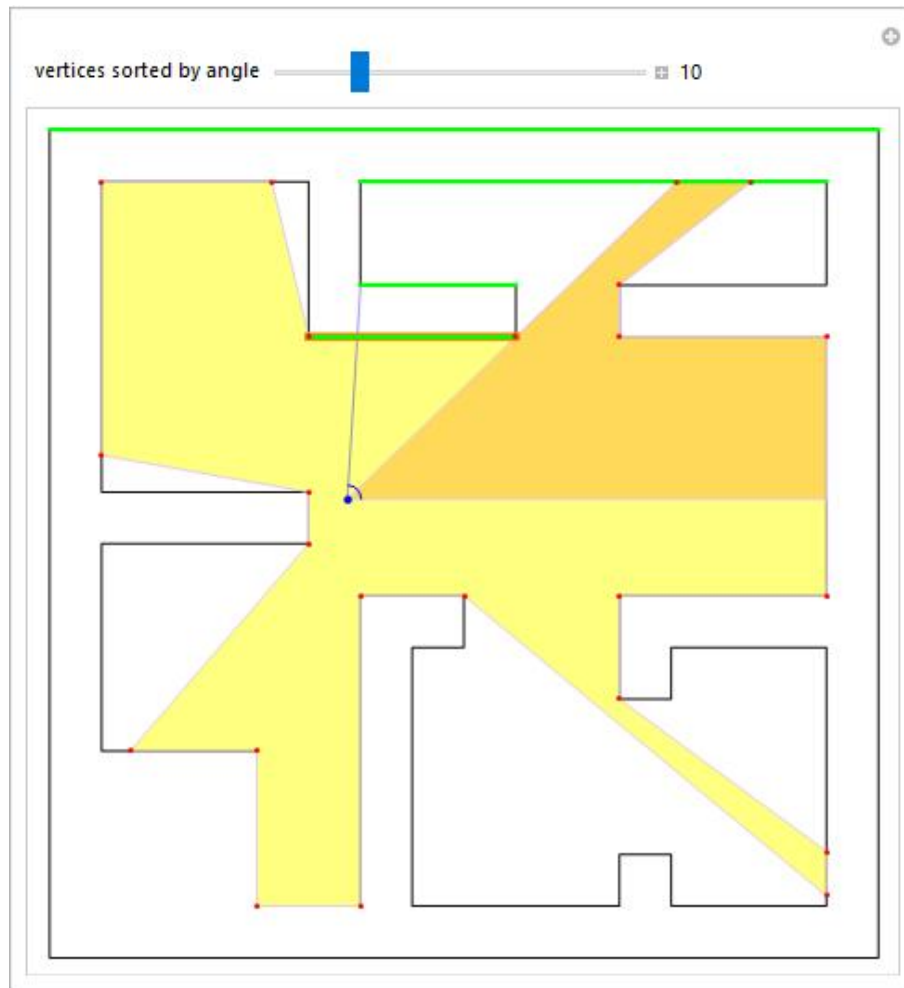


Figure 10: Visibility Region of a Polygon: Example 2 continued

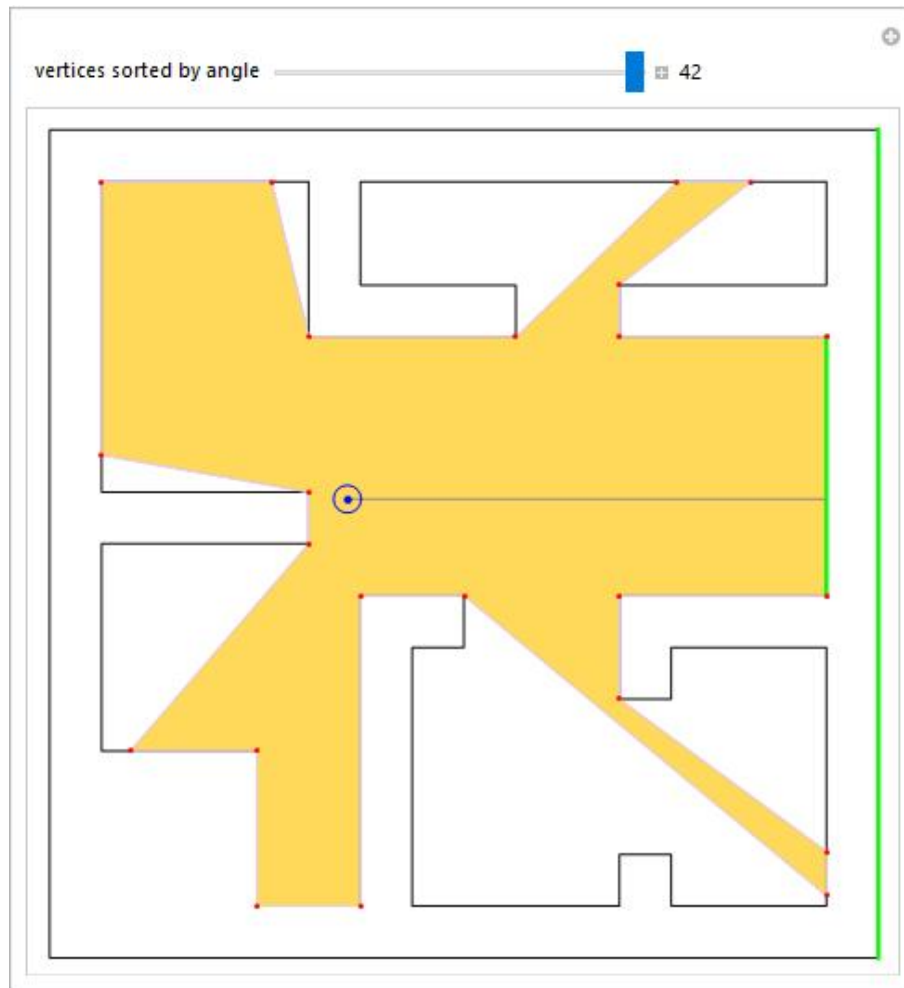


Figure 11: Visibility Region of a Polygon: Example 2 completed algorithm

5 Art Gallery Problem

This chapter explains a demonstration built to teach how the visibility polygon from multiple points can be used to fully cover a given polygon [11]. In Section 5.1 a brief explanation of the art gallery problem is provided. Section 5.2 describes the algorithm used and goes through the parts of the *Mathematica* code. Section 5.3 offers the information to use the online demonstration, and Section 5.4 includes screenshot examples.

5.1 Background

The art gallery problem, also called the museum problem, is a visibility region problem. Art gallery problems deal with the minimum number of guards or cameras required such that the union of each guard's visibility region covers a 2D polygonal region completely. The visibility graph is the set of intervisible points such that when a line drawn from these points does not intersect any obstacle. The art gallery problem can be resolved by getting the visibility graph.

Chvátal's art gallery theorem states that for a simple polygon with n vertices, $\lfloor n/3 \rfloor$ guards (or cameras) are sufficient to guard it [15]. Fisk's short proof indicates that the vertices of the triangulation graph of the polygon may be 3-colored, meaning that the vertices with any one color form a valid guard set [3]. Proofs provided by the [15, 5, 14] state that only $\lfloor n/4 \rfloor$ is required for an orthogonal polygon whose edges meet at right angles. This demonstration can provide a visual proof for the theorems above.

5.2 Algorithm

The “Art Gallery Problem” is an NP-Hard problem. Computing the visibility graph of the points which represent the guards can solve the problem. This demonstration is similar to the “Visibility Region of a Polygon”, demonstration, except

that there are up to eight guards or reference points, and the visibility region from each of the points is computed for the same polygon.

```

1  (*This generates the visible region seen by a point at pm in an \
2  environment with the polygons in polys.*)
3  visiblePolys[polys_, pm : {x_, y_}] :=
4  Module[{p, listOfAllLines, infiniteLine, length = Length@polys,
5    orderedList, sortedList, jList = {}, visibleList = {}, wi = False,
6    wip = False, lvLine, a, b, c, abcList, ba, bc, pb, pbaAngle,
7    pbcAngle, startJlistInt, m, bcVisiInt, gLine, giList, giPoint,
8    giLine, abNotGlaInt, abNotVisInt, bcNotVisInt, i = 1, prevA, prevB,
9    prevC, postA, postB, postC, tb, index, theta, gFlag = False,
10    pprevbAngle, pbAngle
11  },
12  (*Check if point p lies on any of the lines in the environment,
13  if so then shift it by a very small offset.*)
14  p = pm;
15  Table[If[(pointOnSegmentQ[#, pm]),
16    p = pm - 0.00001 normalVector[#]] & /@ (lineList@
17    polys[[k]]), {k, 1, length}];
18  (*Check if point p lies on any of the vertices of the polygons in \
19  the environment, if so then shift it by a very small offset.*)
20  If[Or @@
21    Table[Or @@ ((index = Flatten@Position[polys[[k]], #];
22      pm == #) & /@ polys[[k]]), {k, 1, length}],
23    p = pm +
24      0.00001 (Flatten@polys[[1, index - 1]] +
25        Flatten@polys[[1, index + 1]]);
26  (*get a line parallel to x axis*)
27  infiniteLine = {p, {0.11,
28    0} + {MaximalBy[Flatten[polys, 1], First][[1, 1]], y}};
29  (*reverse the order of the list representing a polygon if point p \
30  is inside the polygon*)
31  orderedList =
32    Table[If[testpoint[polys[[i]], p], Reverse@polys[[i]],
33      polys[[i]]], {i, 1, Length@polys, 1}];
34  (* make a list of all lines*)
35  listOfAllLines = Flatten[Join[(lineList@#) & /@ orderedList], 1];
36  (*sort the lines based on the angle w.r.t to point p*)
37  sortedList =
38    Sort[Flatten[Join[(vertexList@N@#) & /@ orderedList], 1],
39      angleSortCond[p, #1, #2] &];
40  (*number of vertices in the environment*)
41  m = Length@sortedList;
42  (*get the intersection of lines for the infiniteLine*)
43  If[SegmentIntersectionQ[{infiniteLine, #}],
44    AppendTo[jList, Reverse@#]] & /@ listOfAllLines;

```

```

45  (*store the sorted values in jList*)
46  jList = Sort[jList, distSortCond[infiniteLine, #1, #2, p] &];
47  lvLine = First@jList;
48  (*find the line which is closest and also visible*)
49  Table[
50    If[SegmentIntersectionQ[{lvLine, jList[[i]]}],
51      startJlistInt = N@LineIntersectionPoint[{lvLine, jList[[i]]}];
52      If[noIntersection[p, startJlistInt, listOfAllLines],
53        AppendTo[visibleList, startJlistInt];
54        If[startJlistInt[[2]] > p[[2]], lvLine = jList[[i]]]]
55    ], {i, 1, Length@jList}];
56  (*loop around all the vertices of all the polygons*)
57  For[i = 1, i <= m, i++,
58    (*initialize all the variables*)
59    {a, b, c} = sortedList[[i, 1 ;; 3]];
60    {prevA, prevB, prevC} =
61      If[i != 1, sortedList[[i - 1, 1 ;; 3]], sortedList[[m, 1 ;; 3]];
62    {postA, postB, postC} =
63      If[i != m, sortedList[[i + 1, 1 ;; 3]], sortedList[[1, 1 ;; 3]];
64    pbAngle = getAngle[{p, b}]; pprevbAngle = getAngle[{p, prevB}];
65    (*for collinearity between current vertice,
66    next vertex and point p then make them non-collinear*)
67    tb = N@
68      If[pointOnSegmentQ[{p, postB}, b],
69        If[postB == a || pbaAngle <= Pi/3,
70          b + 0.00001 normalVector[{a, p}],
71          b - 0.00001 normalVector[{postB, p}]], b];
72    (*get all the angles and the lists*)
73    abcList = {a, tb, c};
74    ba = {b, a}; bc = {b, c}; pb = {p, b};
75    pbaAngle = Re@getClockwiseAngle[p, b, a];
76    pbcAngle = Re@getClockwiseAngle[p, b, c];
77    (*conditions to determine whether a vertex is visible or not*)
78    If[intersectInteriorQRev2[p, abcList], wi = False,
79      If[i == 1 || Not@pointOnSegmentQ[pb, a],
80        If[noIntersection[p, b, jList],
81          If[i != 1,
82            If[pbAngle != pprevbAngle, wi = True,
83              If[b == prevA || b == prevC || b == postA || b == postC ||
84                gFlag, wi = True, wi = False]], wi = True], wi = False],
85        If[wip, If[noIntersection[b, a, jList], wi = True, wi = False],
86          wi = False]]];
87    (* if the vertices are visible*)
88    If[wi,
89      (*Check if there is an intersection with latest visible line \
90      with the bc line. The intersection is appended in visible list.*)
91      If[SegmentIntersectionQ[{lvLine, bc}],
92        bcVisiInt = N@LineIntersectionPoint[{lvLine, bc}];

```

```

93     If[noIntersection[p, bcVisiInt, listOfAllLines],
94         AppendTo[visibleList, bcVisiInt]]
95     ];
96     (*Glancing blow condition.*)
97     If[(glancingBlow[p, abcList]), gFlag = False, gFlag = True ];
98     (* If glancing blow and previous vertex does not lie on the line \
99     pb*)
100    If[gFlag && (Not@pointOnSegmentQ[pb, prevB]),
101        (*extend the glancing line pb*)
102        gLine = extendedLine[p, tb];
103        (*check for intersection between all the lines in Jlist and the \
104        extended line. Get the intersection point nearest to p*)
105        giList = Sort[(DeleteCases[
106            (If[
107                SegmentIntersectionQ[{gLine, #}] ||
108                And[b != #[[2]], pointOnSegmentQ[gLine, #[[2]]] ||
109                And[b != #[[1]], pointOnSegmentQ[gLine, #[[1]]]],
110
111                If[SegmentIntersectionQ[{gLine, #}], {N@
112                    LineIntersectionPoint[{gLine, #}], #},
113                If[And[b != #[[2]],
114                    pointOnSegmentQ[gLine, #[[2]]], {#[[2]], #},
115                If[And[b != #[[1]],
116                    pointOnSegmentQ[gLine, #[[1]]], {#[[1]], #}
117                ]]] & /@
118            DeleteCases[DeleteCases[listOfAllLines, {b, _}], {_, b}]
119            ), Null]),
120            EuclideanDistance[p, #1[[1]]] <
121            EuclideanDistance[p, #2[[1]]] &];
122
123    If[giList != {},
124        (*giPoint is the closest point of intersection between the \
125        extended glancing line and the lines in Jlist.
126        Where the giLine is the line for the intersection point *)
127        giPoint = giList[[1, 1]]; giLine = giList[[1, 2]];
128        (*condition to decide the order of appending the vertex and the \
129        intersection point*)
130        If[(pointOnSegmentQ[lvLine, giPoint] &&
131            Chop[getAngle[pb]] != 0) || (i == 1 && pbaAngle <= Pi/2)
132            || (SegmentIntersectionQ[{gLine,
133                lvLine}] && (LineIntersectionPoint[{gLine, lvLine}] ==
134                giPoint)) || giPoint == lvLine[[2]],
135            visibleList = Join[visibleList, {giPoint, b}]; lvLine = ba,
136            visibleList = Join[visibleList, {b, giPoint}]; lvLine = giLine]]
137        ,(*if not glancing blow then just append the vertex*)
138        If[(pbAngle != pprevbAngle ), AppendTo[visibleList, b]];
139        lvLine = ba;
140        (*Check if there is an intersection with any line in Jlist with \

```

```

141 line ba. The intersection is appended in visible list. *)
142 Table[If[SegmentIntersectionQ[{ba, jList[[i]]}],
143   abNotGlaInt = N@LineIntersectionPoint[{ba, jList[[i]]}];
144   If[noIntersection[p, abNotGlaInt, listofAllLines],
145     AppendTo[visibleList, abNotGlaInt];
146     lvLine = jList[[i]]], {i, 1, Length@jList, 1}];
147 ] (*If it is not a visible vertex*)
148 (*check if there is an intersection of latest visible line with \
149 the pb line. *)
150 , If[SegmentIntersectionQ[{pb, lvLine}],
151   (*If the line ba lies on the left side of line pb then check \
152 the intersection of line ba with the latest visible line.
153 Append the intersection in visible list. *)
154   If[pbaAngle < pbcAngle,
155     If[Not@leftOrRight[p, b, a] ,
156       If[SegmentIntersectionQ[{lvLine, ba}],
157         abNotVisInt = N@LineIntersectionPoint[{lvLine, ba}];
158         If[noIntersection[p, abNotVisInt, listofAllLines],
159           AppendTo[visibleList, abNotVisInt]; lvLine = ba]]];
160     (*If the line bc lies on the left side of line pb then check \
161 the intersection of line bc with the latest visible line.
162 Append the intersection in visible list. *)
163     If[pbcAngle < pbaAngle,
164       If[Not@leftOrRight[p, b, c],
165         If[SegmentIntersectionQ[{lvLine, bc}],
166           bcNotVisInt = N@LineIntersectionPoint[{lvLine, bc}];
167           If[noIntersection[p, bcNotVisInt, listofAllLines],
168             AppendTo[visibleList, bcNotVisInt]; lvLine = bc]]];
169     (*If there is no intersection of latest visible line with the \
170 pb line then search in Jlist for a line intersecting with latest \
171 visible line and append the intersection in visible list. *)
172     If[SegmentIntersectionQ[{#, lvLine}],
173       AppendTo[visibleList, LineIntersectionPoint[{#, lvLine}]];
174       lvLine = #] & /@ jList
175   ];
176 ];
177 (*Update the Jlist & wip*)
178 wip = wi;
179 (*choose a line to be added in Jlist if the line is in the \
180 counterclockwise direction*)
181 If[pbaAngle < Pi, AppendTo[jList, ba]];
182 If[pbcAngle < Pi, AppendTo[jList, bc]];
183 (*remove the line which is already in Jlist and has the ending \
184 vertex same as the current point in the loop*)
185 jList = (DeleteCases[jList, {_, b}]);
186 ]; (*return the visible list*)
187 visibleList
188 ]

```


The function `visiblePolys` returns the visible polygon for a reference point `pm` and a 2D environment represented by a set of polygons `poly`, is *Mathematica* function used in this demonstration. The function `visiblePolys` follows the steps to compute the visibility region of a polygon that are described in Chapter 4.

5.3 How to Use

This demonstration is available online⁵. In this demonstration, you have an option to change the number of guards, and you can choose up to eight guards. There are three different environments to select. In *Mathematica*, a *locator* is a movable point⁶. Each guard is represented by a small disk-shaped locator, and the region covered by it is colored in a color similar to the color of the locator. All the guards are locators and can be dragged in the manipulate window.

In Figure 12, you can see the cubicle environment, Figure 13 shows the irregular environment, and Figure Figure 14 shows the movable obstacles environment in which two obstacles regular polygon (square and triangle) which are movable. In Figures 12—14 there is only one guard, drawn with a blue color.

5.4 Screenshots

Figure 15 shows a screenshot of the demonstration having six guards in the “cubicle” environment. The region covered by both the light blue guard and the yellow colored guard is shown in an olive green color. Figure 16 shows the “irregular” environment with six guards that completely cover the interior and the exterior of the polygon. Figure 17 shows “irregular” environment with two guards to completely cover the interior of the polygon. Figure 18 shows the “movable obstacles” environment with two guards, one inside the square obstacle. Figure 19 shows the “cubicle” environment with eight guards to completely cover the exterior of the cubicle polygon. Figure 20 shows the “movable obstacles” environment with seven guards, one

⁵demonstrations.wolfram.com/ArtGalleryProblem/

⁶<https://reference.wolfram.com/language/ref/Locator.html>

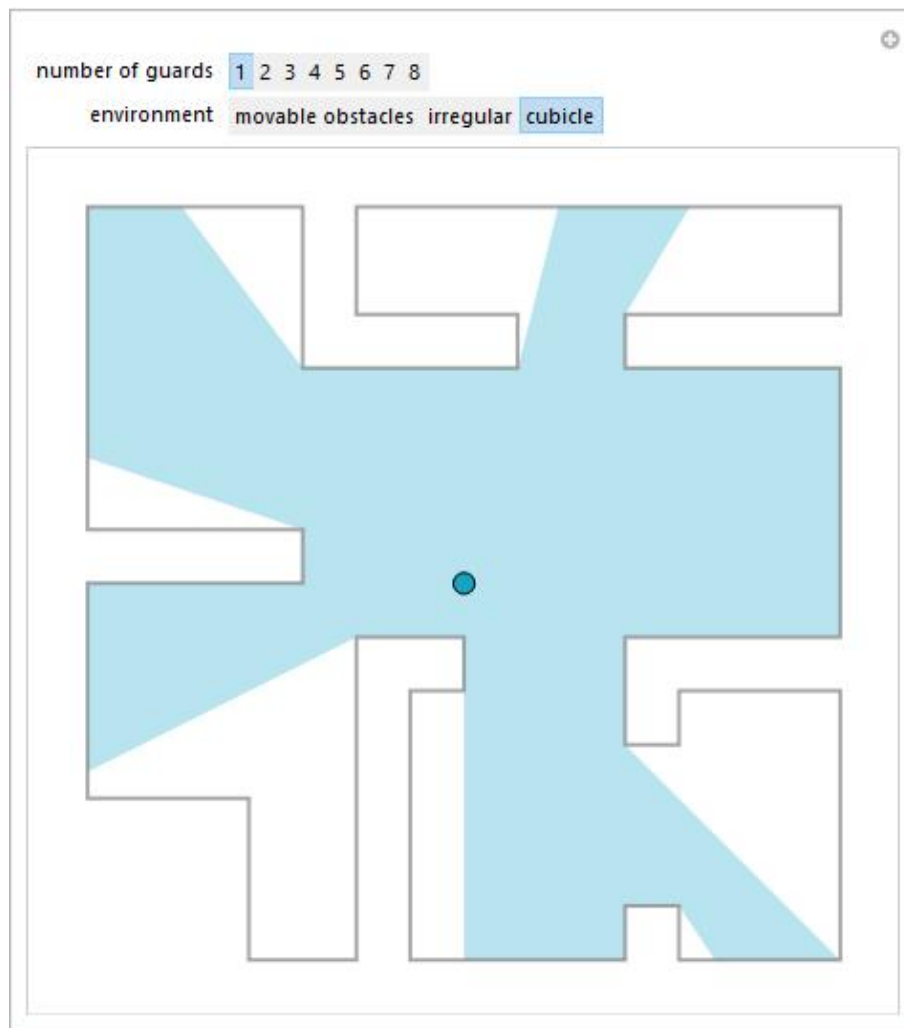


Figure 12: Art Gallery Problem: Cubicle environment.

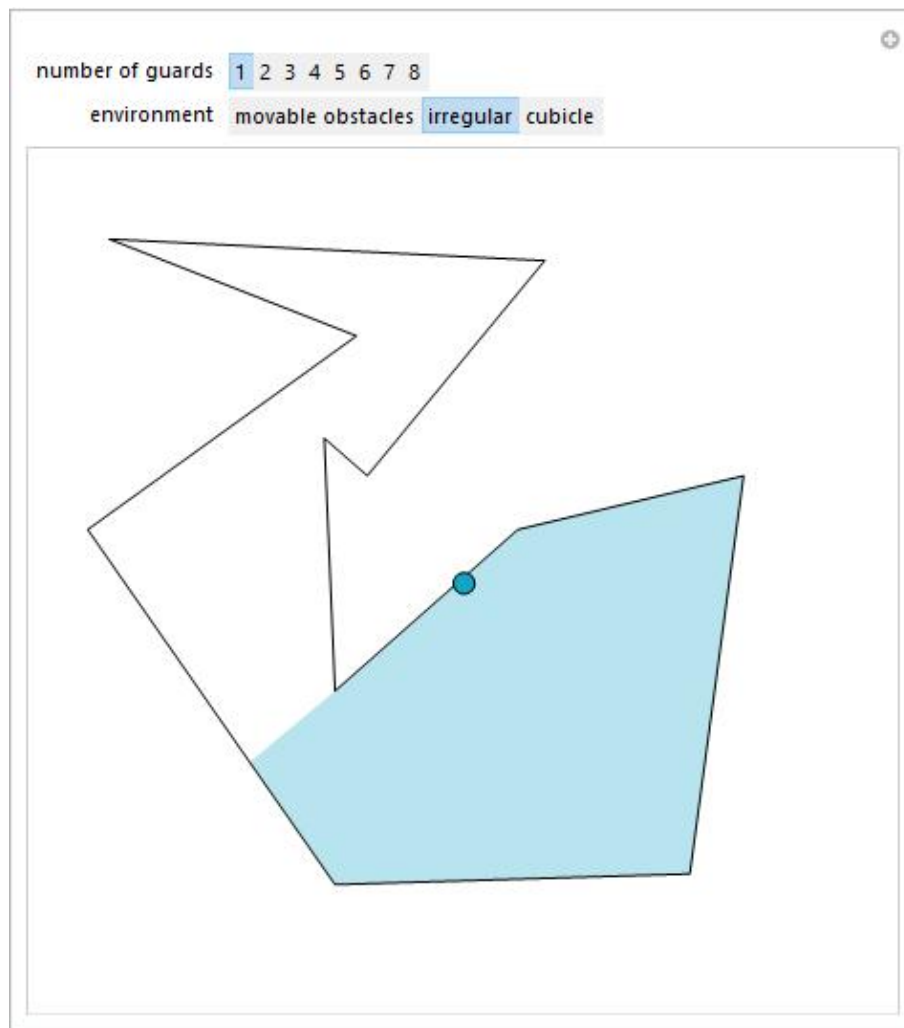


Figure 13: Art Gallery Problem: Irregular environment.

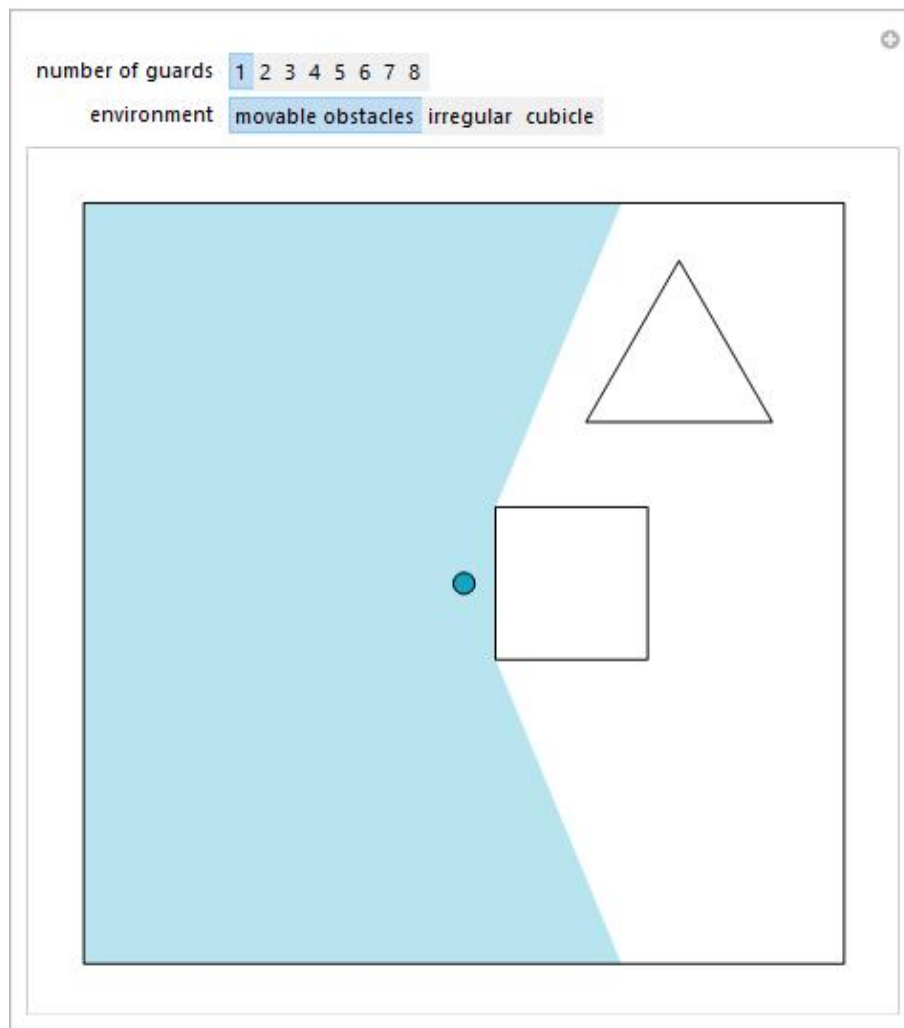


Figure 14: Art Gallery Problem: Movable obstacles environment.

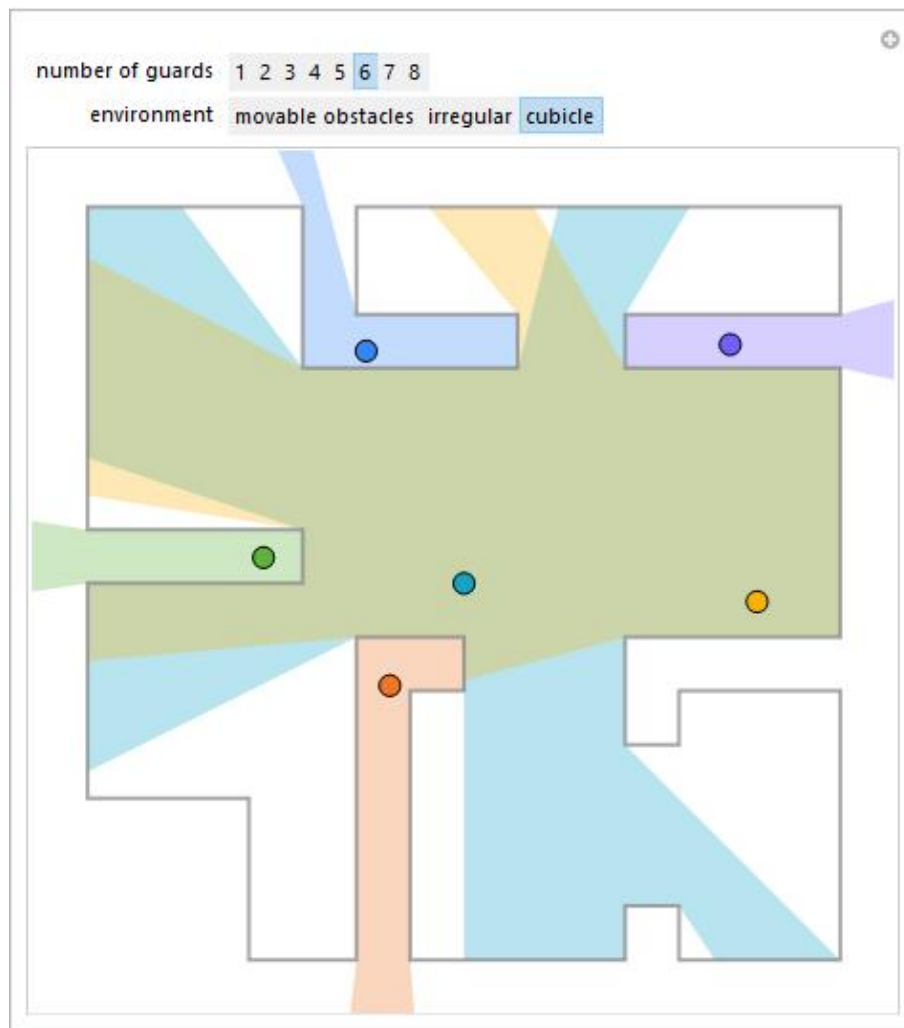


Figure 15: Art Gallery Problem: Cubicle environment with 6 guards.

guard inside each obstacle, and the rest used to cover the exterior.

Covering the exterior of a polygon is known as the *Fortress problem*, and requires $\lceil n/2 \rceil$ guards [3, 1].

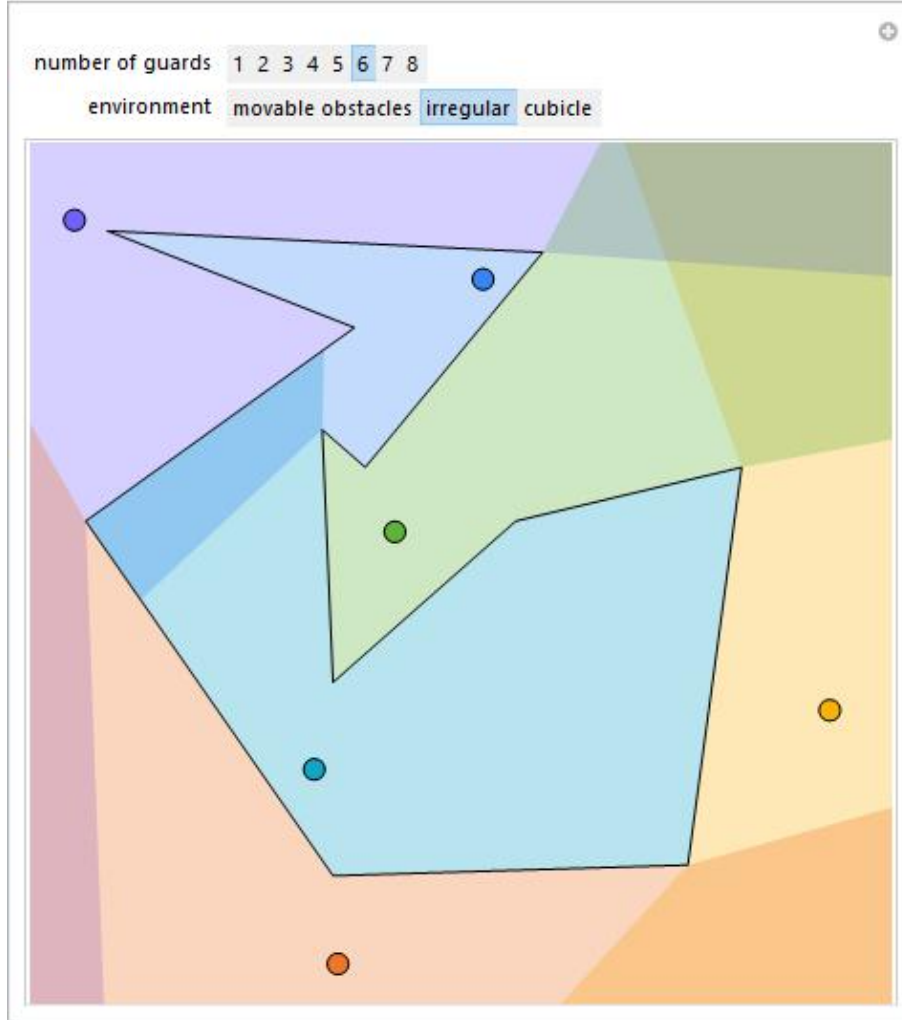


Figure 16: Art Gallery Problem: Irregular environment with 6 guards.

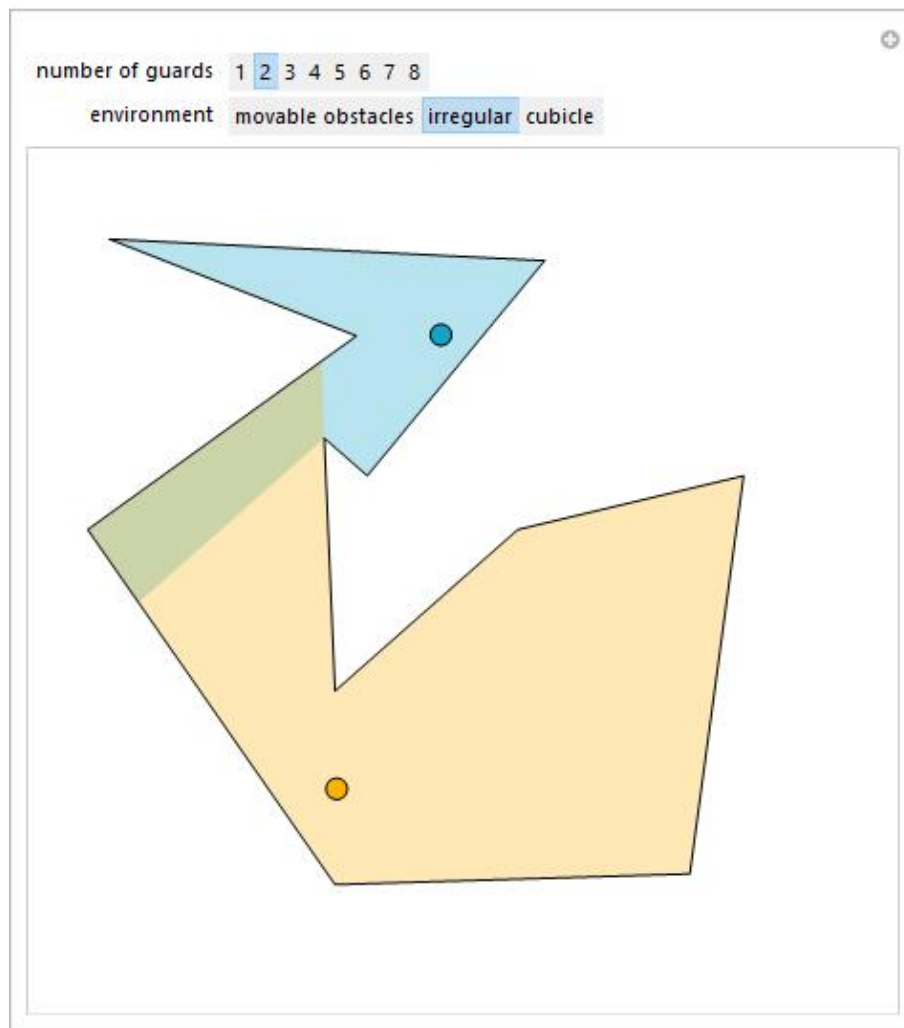


Figure 17: Art Gallery Problem: Irregular environment with two guards.

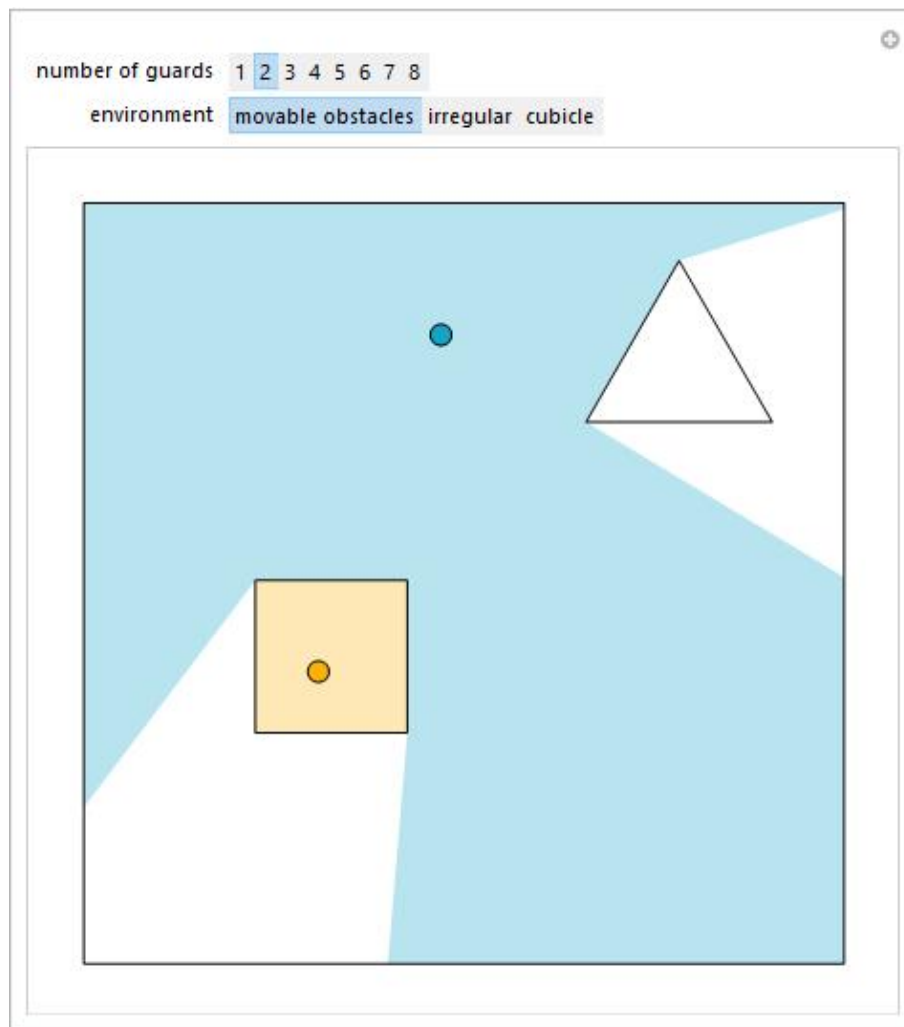


Figure 18: Art Gallery Problem: Movable obstacles environment with one guard inside the obstacle.

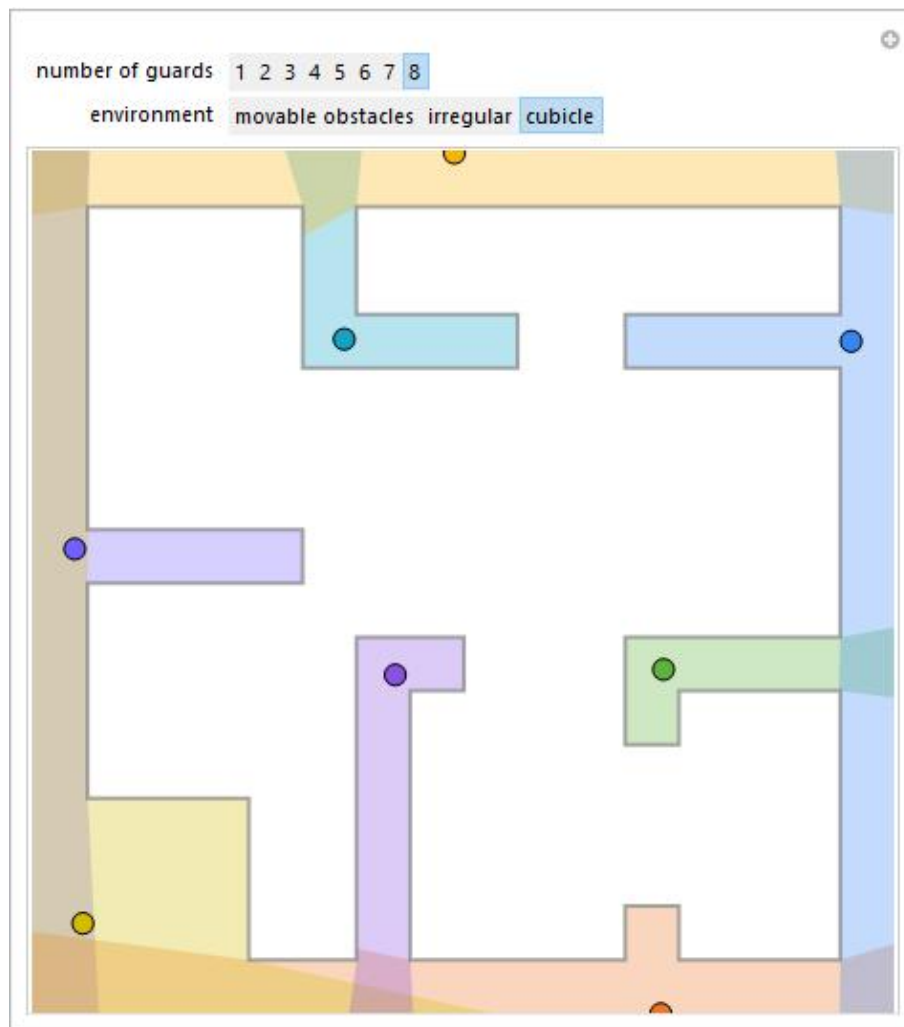


Figure 19: Art Gallery Problem: Cubicle obstacles environment with eight guards to cover the outside of the polygon.

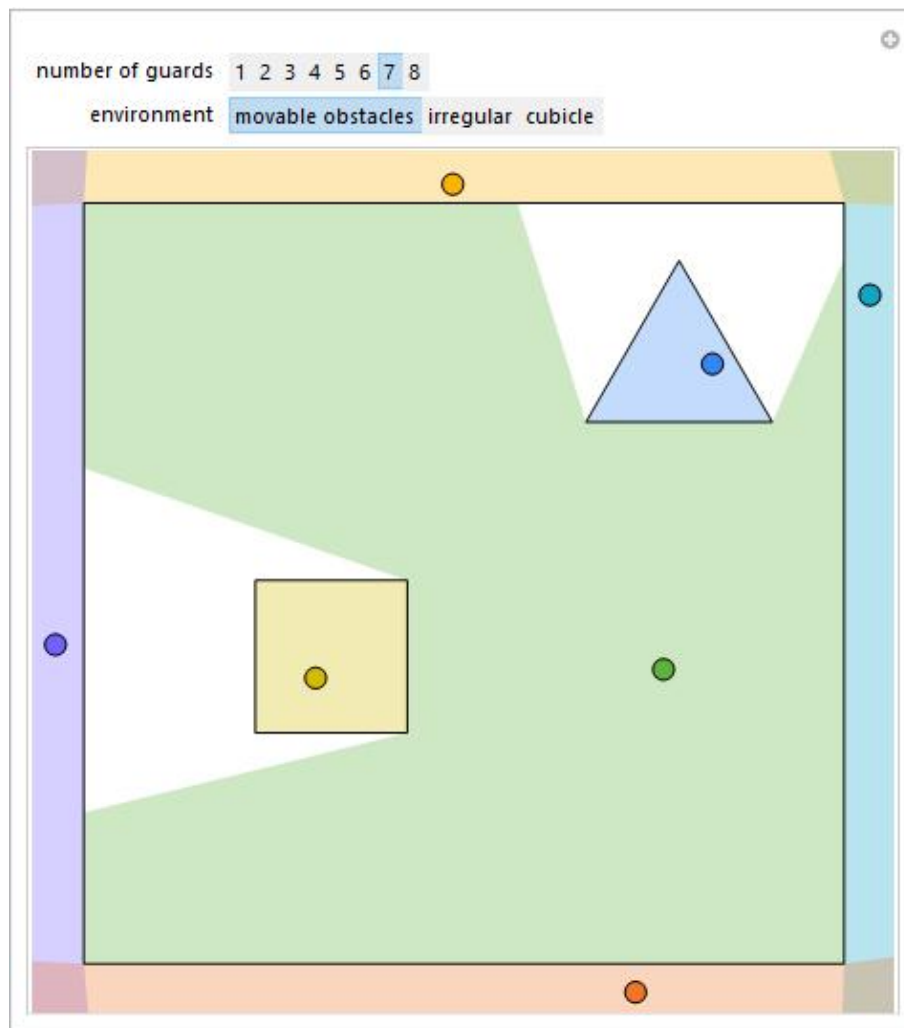


Figure 20: Art Gallery Problem: Movable obstacles environment with seven guards.

6 Base Conversions from Base 2 through 100 Using Radix Points

This chapter explains the demonstration built to teach base conversion [10]. In section 6.1 a brief explanation of the base conversion is provided. Section 6.2 describes the algorithm used and goes through the parts of the *Mathematica* code. Section 6.3 offers the information to use the online demonstration, and Section 6.4 includes screenshot examples.

6.1 Background

The *base* is the unique scalar used to represent numbers in a positional numeral system. *Radix* is a synonym for the base of a number in arithmetic. Binary, Octal, Decimal, Hexadecimal are commonly used number systems, corresponding to base 2, 8, 10, and 16. The *radix point* is the symbol ‘.’ used to separate the whole number from the fractional part. In base 10, the radix point is called the “decimal point”, and in base 2 it is called the “binary point”. Every positive integer can be expressed in the form as

$$a_b = r_m b^m + r_{m-1} b^{m-1} + \dots + r_1 b + r_0 , \quad (1)$$

$$\text{where,} \quad 0 < r_m < b, \text{ and } 0 \leq r_i < b \quad (2)$$

$$\text{and} \quad i = 0, 1, \dots, m - 1 \text{ and } r \in \mathbb{Z}^+. \quad (3)$$

Here $b \in \mathbb{Z}^+$ is the base of representation and a is the number in the base form [8].

6.2 Algorithm

A number written in a base form can be converted into any other base form. Given an integer in base b , the decimal value is obtained by multiplying the i^{th} digit by $b^{(i-1)}$ and summing the result. Digits to the left of the radix point are

multiplied by b to increasing powers, starting at b^0 . Digits to the right of the radix are multiplied by decreasing powers of b .

For example:

$$ab4d.e6_{16} = a \times 16^3 + b \times 16^2 + 4 \times 16^1 + d \times 16^0 + e \times 16^{-1} + 6 \times 16^{-2} \quad (4)$$

$$= 10 \times 16^3 + 11 \times 16^2 + 4 \times 16^1 + 13 \times 16^0 + 14 \times 16^{-1} + 6 \times 16^{-2} \quad (5)$$

$$= 10 \times 4096 + 11 \times 256 + 4 \times 16 + 13 \times 1 + \frac{14}{16} + \frac{6}{256} \quad (6)$$

$$= 40960 + 2816 + 64 + 14 + 0.875 + 0.0546875 \quad (7)$$

$$= 43854.9296875 \quad (8)$$

To convert a decimal number to base b , let n be the integer part of the decimal number and f be the fractional part. First compute the integer part. Divide n by b ; the remainder is the least significant digit of the integer. The process repeats: divide the quotient by b , and use the remainder as the next most significant digit of the integer part. The process terminates when the quotient is 0, and the last remainder is the most significant digit. Next, compute the fractional part. Let $a = f \times b$; a is always less than $2b$. The integer part of a is the next digit to the right of the radix point of the number in base b . This digit may be zero. Continue multiplying the fractional part of a by b until the fractional part is zero or a sufficient resolution is reached.

For example, 0.44_{10} to base 6:

$$0.4 \times 6 = \mathbf{2}.64, \text{ giving the answer to one digit precision of } \mathbf{0.2}_6$$

$$0.64 \times 6 = \mathbf{3}.84, \text{ giving the answer to two digit precision of } \mathbf{0.23}_6$$

$$0.84 \times 6 = \mathbf{5}.04, \text{ giving the answer to three digit precision of } \mathbf{0.235}_6$$

$$0.04 \times 6 = \mathbf{0}.24, \text{ giving the answer to four digit precision of } \mathbf{0.2350}_6$$

$$0.24 \times 6 = \mathbf{1}.44, \text{ giving the answer to five digit precision of } \mathbf{0.23501}_6$$

$$0.44 \times 6 = \mathbf{2}.64, \text{ giving the answer to six digit precision of } \mathbf{0.235012}_6$$

$$0.64 \times 6 = \mathbf{3}.84, \text{ giving the answer to seven digit precision of } \mathbf{0.2350123}_6$$

...

The remainder repeats:

0.2350123501235012350123501235012350123501235012350123501235012350...6.

The *Mathematica* function `parseNumericStringBase` below takes the input base in decimal as a string and returns the number in the output base specified.

```

1 parseNumericStringBase[strIn_, base_, baseOut_, res_] :=
2   (*convert strIn, which is a string of a number in base
3   'base' (and may have a '.' with fractional values) into *)
4   Module[{ complement, charstonum, parsedNum, parsedStrDrop
5     , periodPosition =
6       Flatten@Position[Characters[strIn],
7         "." ] (*get the decimal position*)
8     , baseVals =
9       Join @@ CharacterRange @@@ {{"0", "9"}, {"a", "z"}, {"A",
10        "Z"}, {"\[Alpha]", "\[Omega]"}, {"\[CapitalAlpha]",
11        "\[CapitalNu]"} } (*list of all base values 0 to \[Omega]*)
12     , outputNum = "NaN" (*when no output*)
13   } ,
14   charstonum =
15     AssociationThread[
16       baseVals -> Range[0, 99]]; (*associate char with num*)
17   (*check if there is no more than 1 "."*)
18   If[ Length[periodPosition] > 1 , Return[{StringForm["\n(\n*
19   StyleBox["`.\`", \nFontColor->RGBColor[1, 0, 0]] \n Error:\nNo more
20   than 1 "\`.\`" in the input number", strIn], outputNum}]];
21   (*check that the string only has values from 1 to base *)
22   complement =
23     Complement[Characters[strIn], {"."}, baseVals[[1 ;; base]]];
24   If[Length[complement] > 0, Return[{StringForm["\n(\n*
25   StyleBox["`.\`", \nFontColor->RGBColor[1, 0, 0]] \n Error:\nThe
26   value(s) `` were in this string.\nFor base ``, the only valid
27   characters are "\n`", strIn, complement, base, baseVals[[1 ;; base]],
28     outputNum}]];
29   parsedStrDrop =
30     StringDrop[strIn, periodPosition]; (*remove decimal point*)
31   If[Length[periodPosition] == 0,
32     periodPosition = {StringLength[parsedStrDrop] + 1}];
33   parsedNum =
34     Characters[parsedStrDrop] /.
35       charstonum; (*Get respective num for char*)
36   (*find the decimal number*)
37   outputNum =
38     Total@N@{baseRtoP[parsedNum, base, periodPosition, #] & /@ (Range@
39     Length[parsedNum])};
40   (*return the num in required base form*)

```

```

41 Return[ {Pane[
42   Style[StringForm["` = \n`", Subscript[strIn, base],
43     pr[convertToBase[N@outputNum, baseOut, res], Infinity]],
44   LineIndentMaxFraction -> 0], 250], outputNum]]
45 ]

```

```

1 baseRtoP[n_, b_, p_, i_] :=
2 Return@(n[[i]]*(N@b^((First@p) - i - 1)))

```

The function `baseRtoP` computes the formula, as mentioned in equation 6.1 - 6.2. `convertToBase` shown below is the *Mathematica* function which uses the internal built-in function `BaseForm` to convert any base less than 36 to decimal 10, and the rest from 36 to 100 are converted using the algorithm mentioned above. All the input and output performed in the *Mathematica* functions use string manipulation [10].

```

1 (*convert decimal to any base up to base 100*)
2 convertToBase[numIn10_, base_, decimalRes_] :=
3 Module[{p, r, a = 0, b, n = IntegerPart[numIn10],
4   m = FractionalPart[numIn10], intOut = "", fOut = "",
5   baseVals =
6     Join @@ CharacterRange @@@ {"0", "9"}, {"a", "z"}, {"A",
7       "Z"}, {"\[Alpha]", "\[Omega]"}, {"\[CapitalAlpha]",
8         "\[CapitalNu]"}},
9   If[base < 36, Return@baseRes[numIn10, base, decimalRes],
10   While[m > 0 && a < decimalRes, a = a + 1; b = N@(m*base);
11     p = N@IntegerPart[b]; (*convert the fraction part*)
12     m = N@FractionalPart[b];
13     fOut = StringJoin[fOut, baseVals[[p + 1]]];
14   While[n > 0, {n, r} =
15     QuotientRemainder[n, base]; (*convert the Integer part*)
16     intOut = StringJoin[baseVals[[r + 1]], intOut]];
17   If[StringLength[intOut] < 1, intOut = "0"];
18   Return@(Subscript[
19     StringJoin[intOut, If[FractionalPart[numIn10] != 0, ".", ""],
20     fOut], ToString@base)]]

```

6.3 How to Use

This demo is available online⁷. In this demonstration, there are three sliders “input base”, “output base”, “resolution”. The input and output base can be up to 100. Because string input is not allowed for online demonstrations, a keypad entry method is used instead. The keypad is shown on the left side of the demonstration, and the keys in it change as per the input base. There is a “delete” key to delete the previously entered digit, and a “clear” key to clear all the input.

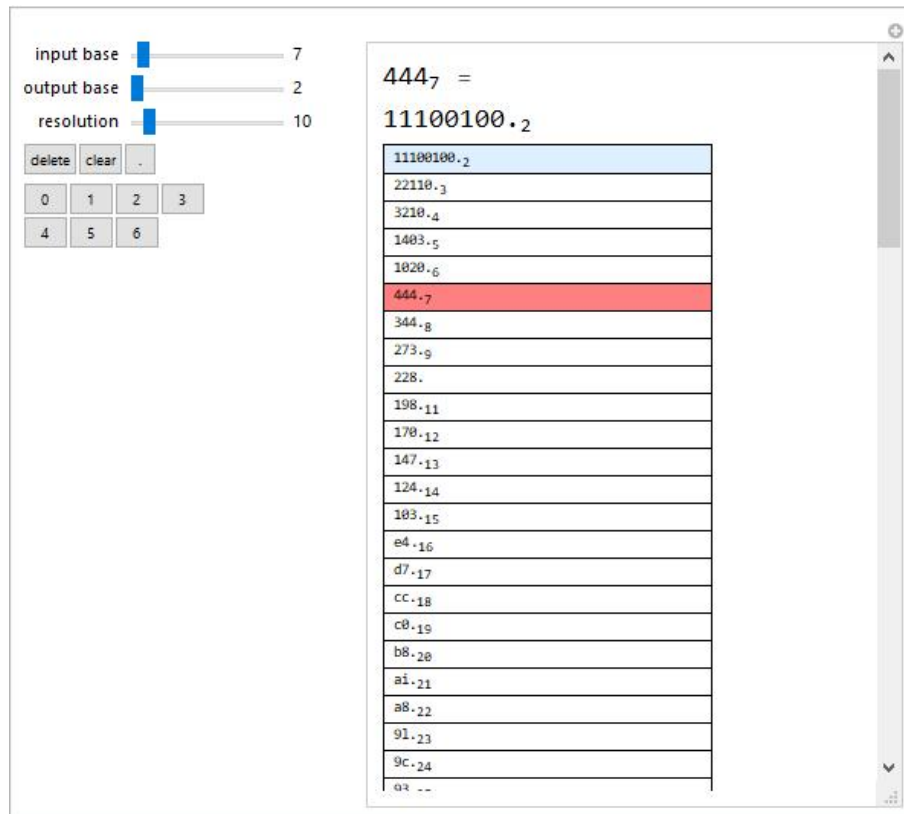


Figure 21: Base Conversion: Input 228 in base 7 output in base 2.

The resolution of the output can be up to 10 values past the radix point. The manipulate window shows entered input highlighted in blue and respective output highlighted in red, along with a list of the converted number in each base up to 100. The figure 21 shows an example where the input is 228 in base 7, and the output is in base 2 with the output resolution of 10 places.

⁷demonstrations.wolfram.com/BaseConversionsFromBase2Through100UsingRadixPoints/

6.4 Screenshots

Figure 22 shows an example with the input is 9775.445 in base 100 with output in base 36. Figure 23 shows an example with the input is 0 in base 36 with output in base 2. Figure 24 shows an example with the input is 100 in base 10 with output in base 60. All the examples have a resolution set to 10 places.

input base: 100
output base: 36
resolution: 10

delete clear .

0	1	2	3
4	5	6	7
8	9	a	b
c	d	e	f
g	h	i	j
k	l	m	n
o	p	q	r
s	t	u	v
w	x	y	z
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P
Q	R	S	T
U	V	W	X
Y	Z	α	β
γ	δ	ε	ζ
η	θ	ι	κ
λ	μ	ν	ξ
ο	π	ρ	ς

$\Delta_{\xi} \cdot IO_{100} =$
7jj.g0px4bip3k₃₆

10011000101111.0111000111 ₂
111102001.1100001012 ₃
2120233.1301322320 ₄
303100.2103030303 ₅
113131.2400415304 ₆
40333.3054305430 ₇
23057.3436560507 ₈
14361.4003572175 ₉
9775.445
7387.4993277105 ₁₁
57a7.540b62a688 ₁₂
45ac.5a288500b
13
37c3.63311973cc ₁₄
2d6a.6a1d1d1d1c ₁₅
262f.71eb851ec
16
1ge0.79a4e6384
17
1c31.80345dc46
18
1819.88c4g10g4
19
148f.8120

Figure 22: Base Conversion: Input 9775.445 in base 100 output in base 36.

0	1	2	3
4	5	6	7
8	9	a	b
c	d	e	f
g	h	i	j
k	l	m	n
o	p	q	r
s	t	u	v
w	x	y	z

36 =

0.2

0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0.
0.11
0.12
0.13
0.14
0.15
0.16
0.17
0.18
0.19
0.20
0.21
0.22
0.23
0.24
a --

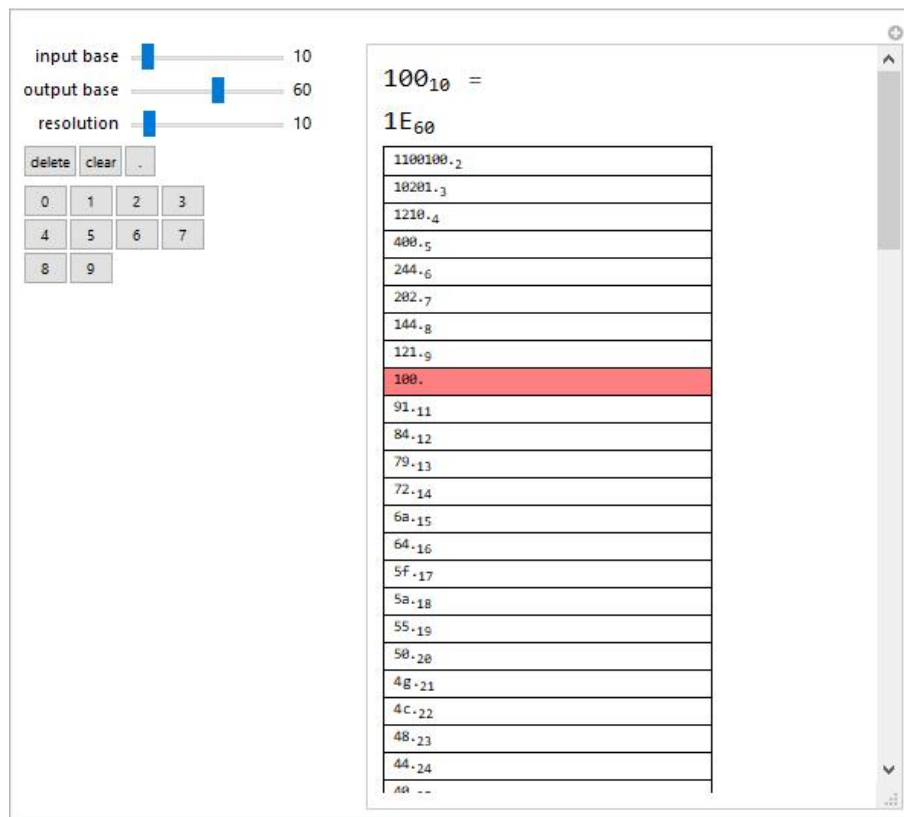


Figure 24: Base Conversion: Input 100 in base 10 output in base 60.

7 Motion Planning for Translating Polygonal Robot

This chapter explains a demonstration built to teach motion planning for a translating polygonal robot. The demonstration has been submitted, but is yet to be published⁸. In Section 7.1 a brief explanation of the motion planning is provided. Section 7.2 describes the algorithm used and goes through the parts of the *Mathematica* code. Section 7.3 offers the information on how to use the online demonstration, and Section 7.4 includes screenshot examples.

7.1 Background

Robot motion planning deals with the problem of computing a continuous path that connects the starting configuration of a robot to the goal configuration of the robot while avoiding collision with any of the obstacles in the work-space. Motion planning requires computing if a given path would cause a collision with an obstacle. Planning in the work-space is challenging because collision checks seemingly require sweeping the polygonal robot along the path, and continuously checking for collision. A common trick in motion planning is to perform the planning in the configuration space for the robot. In the configuration space, the robot's configuration is represented by a point at its (x, y) center. The polygonal obstacles map to regions in the configuration space. These obstacle regions are calculated using the Minkowski sum. The shortest path is then the line from the initial configuration to the destination configuration that does not intersect the obstacle regions in the configuration space. This same line represents the shortest path in both the configuration space and the work-space [7].

⁸<https://demonstrations.wolfram.com/preview.html?draft/54742/000124/MotionPlanningForTranslatingPolygonalRobot>

7.2 Algorithm

This Demonstration computes the path for a polygonal robot that can translate inside a bounded workspace populated by polygonal obstacles. A valid path for a 2D robot in a 2D environment can be computed in four steps as follows:

Firstly, check whether the robot's initial or final position collide or enter any obstacle or are out of the workspace. If so, then there exists no possible path. This is done by using the `testpoint` function shown below.

```
1  (*true if pt is inside polygon*)
2  testpoint[poly_, pt_] :=
3  Round[(Total@
4      Mod[(# - RotateRight[#]) &@ (If[(pt - #) != {0.0, 0.0},
5          ArcTan @@ (pt - #), 0.0] & /@ poly), 2 Pi, -Pi]/2/Pi)] != 0.0
```

Secondly, compute the Minkowski sum of all the obstacles and the inverse Minkowski sum of the boundary. The Minkowski sum is the polygon generated from the robot and the obstacle such that the robot centroid can move along this polygon without colliding or entering the obstacle. Computing the Minkowski sum requires that the robot and the obstacles are convex polygons. The Minkowski sum algorithm used is the same as explained in chapter 3. The inverse Minkowski is used to find the configuration space of the robot. The `configBoundaryFunc` function returns the configuration space of the robot at a point r .

```
1  (*Compute the configuration space defined by the outer boundary.*)
2  configBoundaryFunc[borderpoly_, robotPoly_, r_] :=
3  Module[{borderpolyA, boundMidpts, boundNormal, boundaryOffsets, ip1},
4      borderpolyA = Join[borderpoly, {First@borderpoly}];
5      boundMidpts = ((borderpolyA[[#]] + borderpolyA[[# + 1]])/2.0) & /@
6          Range@Length[borderpoly];
7      boundNormal =
8          Normalize[{borderpolyA[[# + 1, 2]] - borderpolyA[[#, 2]],
9              borderpolyA[[#, 1]] - borderpolyA[[# + 1, 1]]}] & /@
10          Range@Length@borderpoly;
11      boundaryOffsets =
12          Table[First[
13              MaximalBy[(# - r[[1]]) & /@ robotPoly[[1]],
```

```

14      N[Dot[boundNormal[[i]], #]] &]], {i, 1, Length[borderpoly]};
15      (ip1 = If[# < Length[borderpoly], # + 1, 1];
16      lineInt[{borderpoly[[#]] - boundaryOffsets[[#]],
17      boundMidpts[[#]] - boundaryOffsets[[#]]}, {borderpoly[[ip1]] -
18      boundaryOffsets[[ip1]],
19      boundMidpts[[ip1]] - boundaryOffsets[[ip1]]}) & /@
20      Range@Length[borderpoly]
21    ]

```

Thirdly, draw visible bitangent lines from the vertex of each Minkowski polygon and the centroids of the robot's initial and final positions. Bitangent lines are the lines whose ends are reflex vertices. Reflex vertices are the vertices of the polygons whose exterior angles are greater than π (also called *convex vertices*). The function **reflex** returns true if the three vertices of a polygon form a reflex vertex.

```

1  (*Function Gives whether a point is reflex or not, Chop is used to \
2  round of to the nearest integer. From Motion Plannning Book by Steven \
3  M.LaValle*)
4  reflex[{x1_, y1_}, {x2_, y2_}, {x3_, y3_}] := Chop[Det[({
5      {1.0, x1, y1},
6      {1.0, x2, y2},
7      {1.0, x3, y3}
8      }))] > 0.0

```

The **biTangent** function shown below returns true if the line drawn from the vertices of two different polygons is a bitangent line.

```

1  (*If the given points are reflex then the function returns whether \
2  the line between the two points is bitangent or not. From Motion \
3  Plannning Book by Steven M.LaValle*)
4  biTangent[{p1_, p2_, p3_}, {p4_, p5_, p6_}] :=
5  Or[Xor[reflex[p1, p2, p5], reflex[p3, p2, p5]],
6  Xor[reflex[p4, p5, p2], reflex[p6, p5, p2]]]

```

To obtain the visible bitangent lines, calculate the visibility graph, and compute the intersection of the visibility graph and the bitangent lines. Form a graph of the visible bitangent lines and the Minkowski polygon lines. The visibility graph is generated using the visibility region of a polygon, as described in Chapter 4.

Function `visBiLineRev2` computes the intersection of the lines which are bitangent and exist in the visibility graph.

```

1 visBiLineRev2[congifpoly_] :=
2   With[{bitangentLine =
3     N[Flatten[
4       biTangents2polyRev1#[[1]], #[[2]]] & /@
5       Subsets[congifpoly, {2}], 1], 3]},
6     DeleteCases[Flatten[Table[Table[
7       Intersection[{#, configpoly[[j, i]]] & /@
8       N[
9         visiblePolys[
10          configpoly, (congifpoly[[j, i]] -
11            0.001*(congifpoly[[j,
12              If[i + 1 > Length@congifpoly[[j]], 1, i + 1]]] +
13              configpoly[[j, If[i - 1 == 0, -1, i - 1]]] -
14              2*congifpoly[[j, i]])), 3]
15        ], bitangentLine], {i, 1, Length@congifpoly[[j]]}], {j, 1,
16          Length@congifpoly}], 2], Null]]

```

Finally, calculate the shortest path from the generated graph using the A* search algorithm using a list of adjacent nodes.

7.3 How to Use

A draft of this demo is available online⁹. In this demonstration, you can change the initial and the final position of the robot by dragging the locators. The obstacle locations can be changed. As seen in F 25 shows the workspace, the initial robot position is colored in light-blue and the final place as light-green. The boundary is colored light-yellow. The Minkowski sum of each obstacle with the robot is overlapped on each obstacle. The obstacles are light-red in color. The robot sides and the boundary sides can be changed from three to five. The visible bitangent lines are colored in blue, the glancing lines from the robot start and end to the Minkowski vertices are colored in orange and purple. The final path computed is green in color. The robot, which is orange in color, moves along the path when the “progress” slider

⁹<https://demonstrations.wolfram.com/preview.html?draft/54742/000124/MotionPlanningForTranslatingPolygonalRobot>

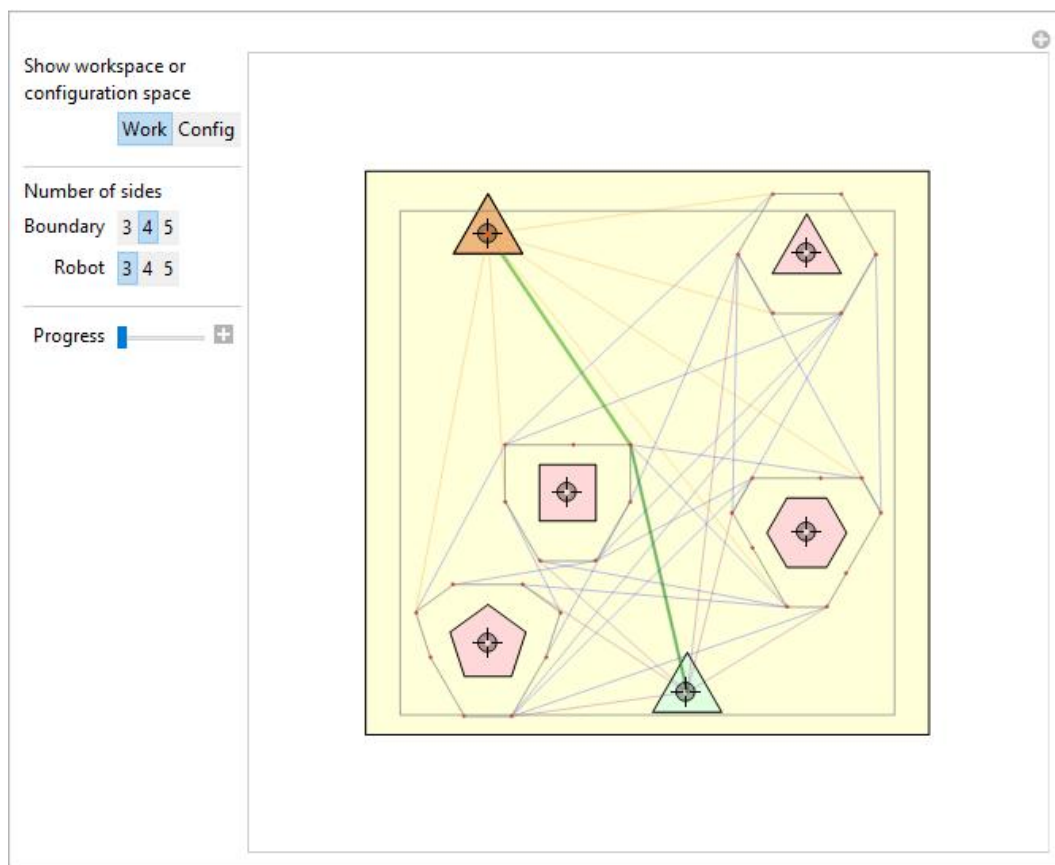


Figure 25: Motion Planning: “Work space”

is moved. If the robot enters the obstacle or is outside the boundary, there is no path, and the robot start or goal turns red. Figure 26 shows the configuration space in grey having the Minkowski sums in the space. If the progress slider is moved, a green color dot representing the robot centroid is moved along the final path.

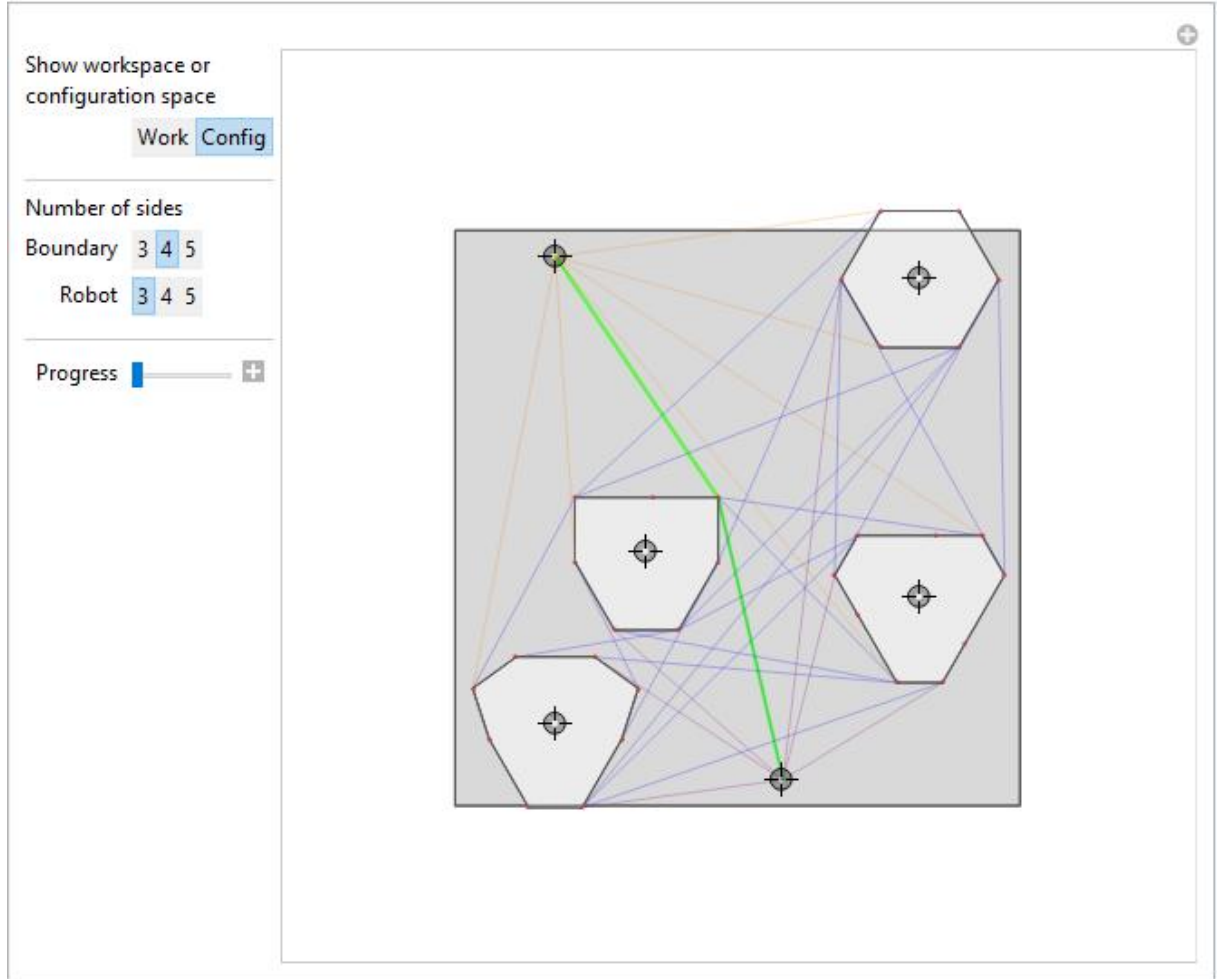


Figure 26: Motion Planning: “Configuration Space”

7.4 Screenshots

Example 1, as in Figure 27—28 shows a triangular robot in a four-sided boundary environment in both work space and the configuration space. The slider is in between the initial position and the end goal seen in the figures as an instance of the robot in orange.

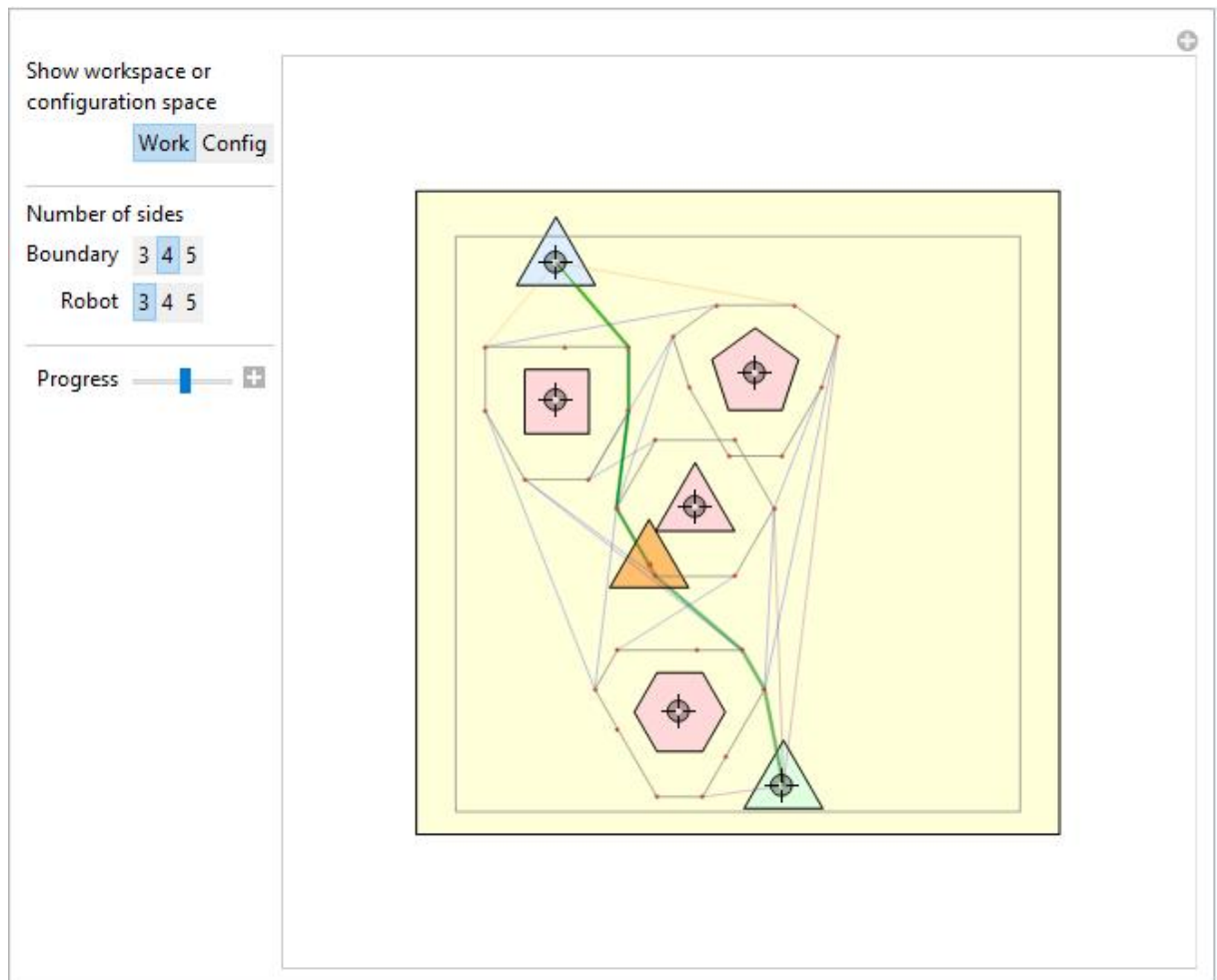


Figure 27: Motion Planning: Example 1 shows the work space of a triangle robot in a square boundary environment.

Example 2, as in Figure 29—30, shows a square robot in a three-sided boundary environment in both work space and the configuration space. The Minkowski sum for the same obstacles with different robot sides can be seen in figures 27 and 29.

Example 3, as depicted in Figure 31—32, shows a square robot in a five-sided boundary environment in both works pace and the configuration space. The Minkowski difference can be seen in each of the work space as grey border, which is the configuration space of the robot.

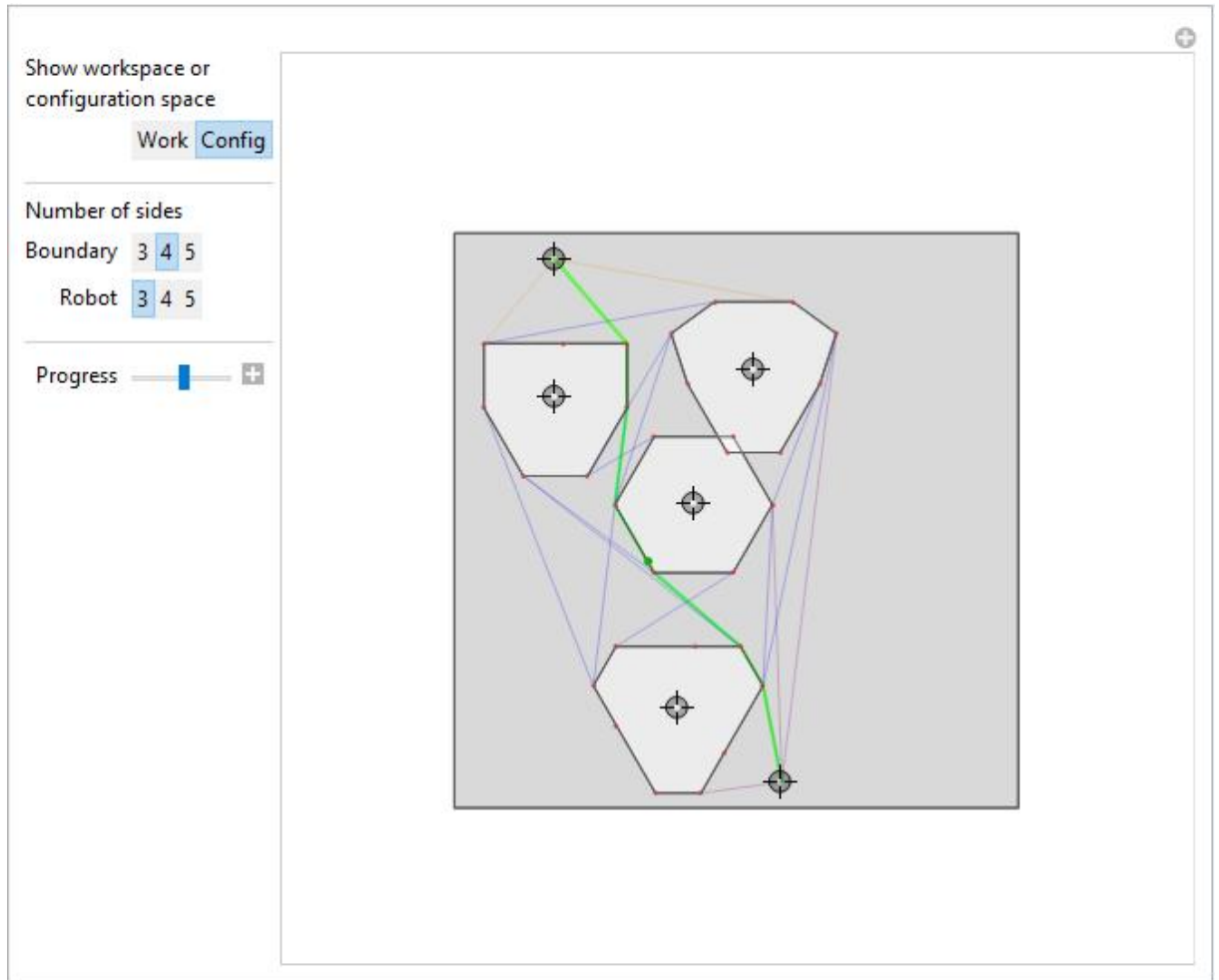


Figure 28: Motion Planning: Configuration space for Example 1.

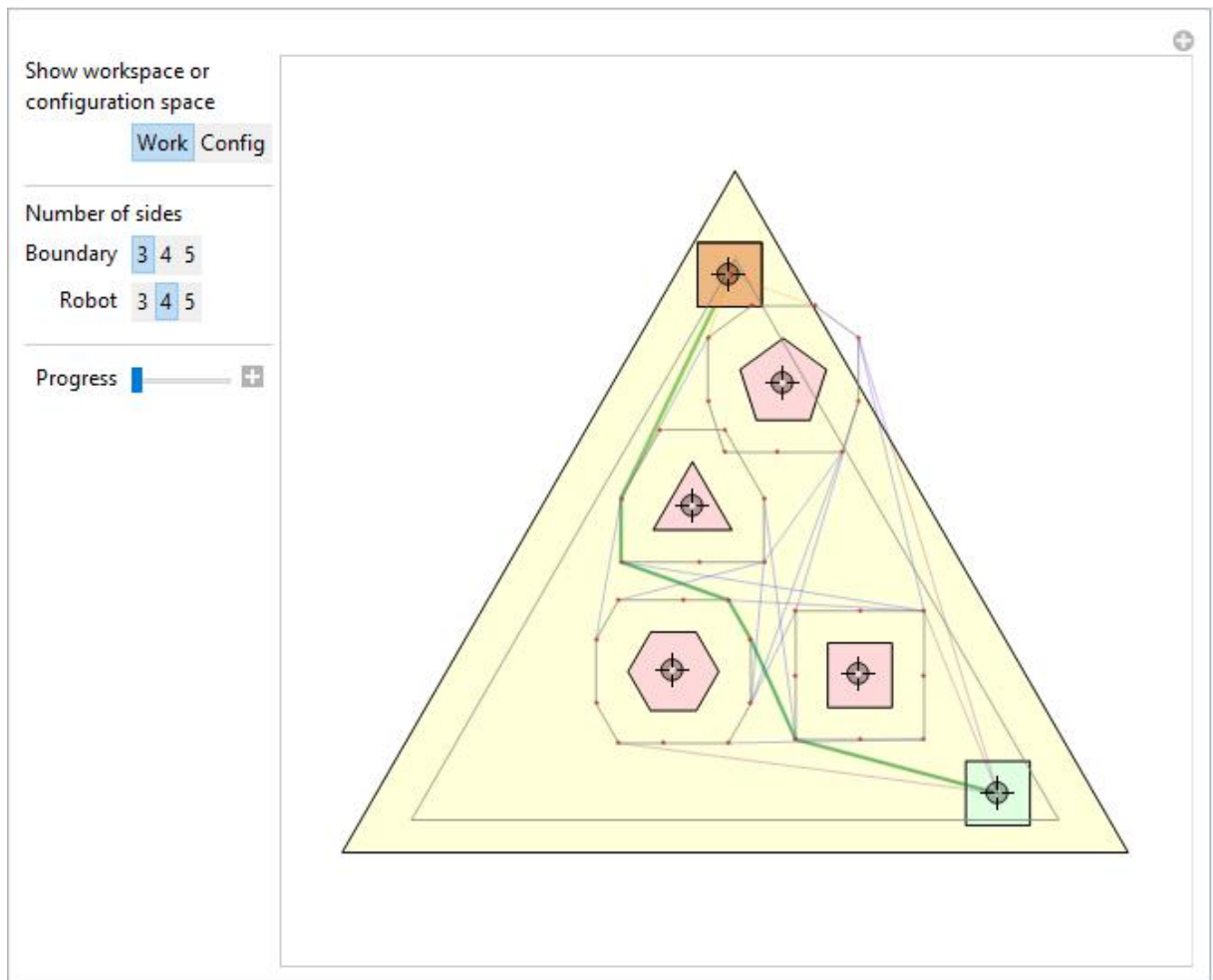


Figure 29: Motion Planning: Example 2 shows the work space of a square robot in a triangular boundary environment.

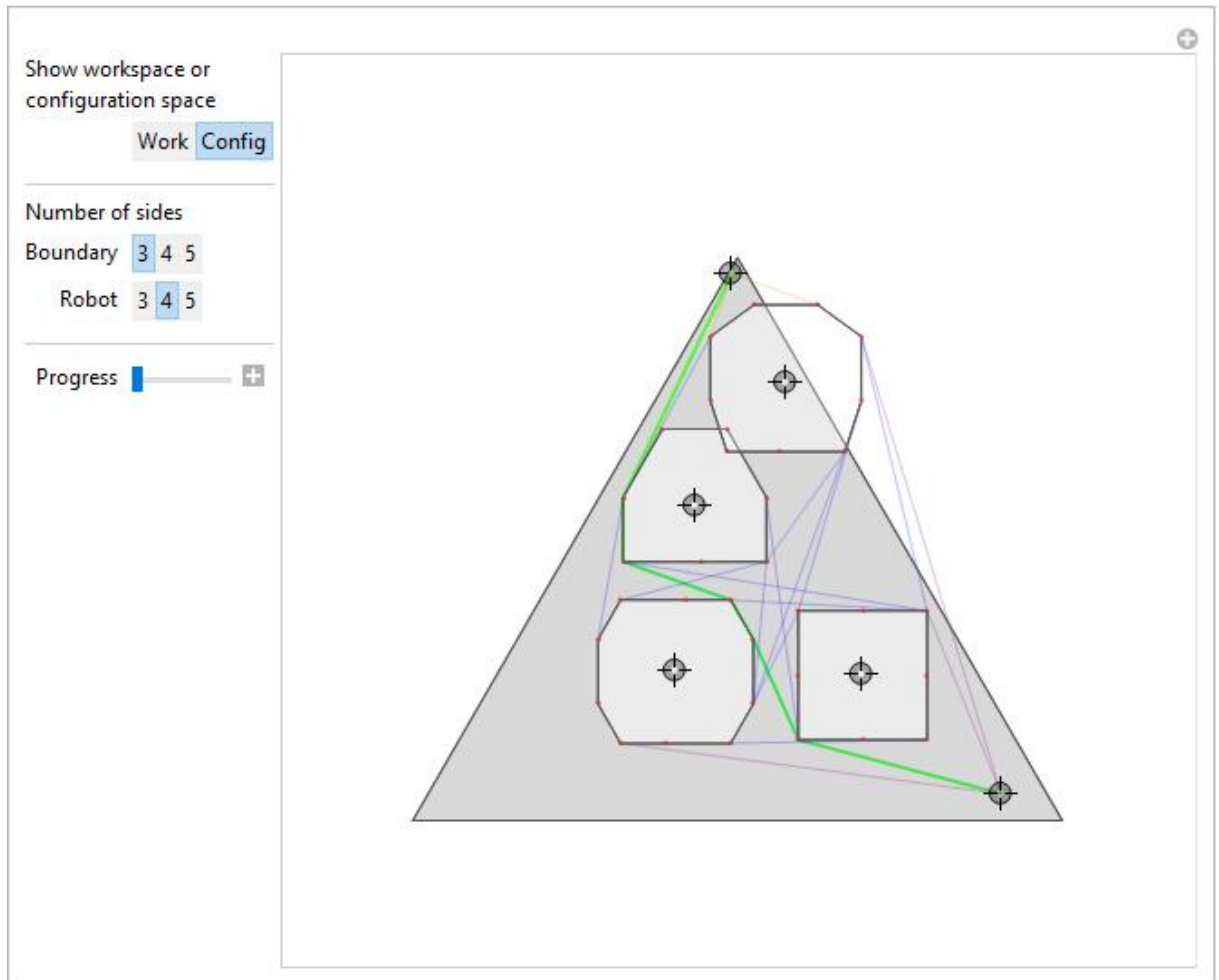


Figure 30: Motion Planning: Configuration space for Example 2.

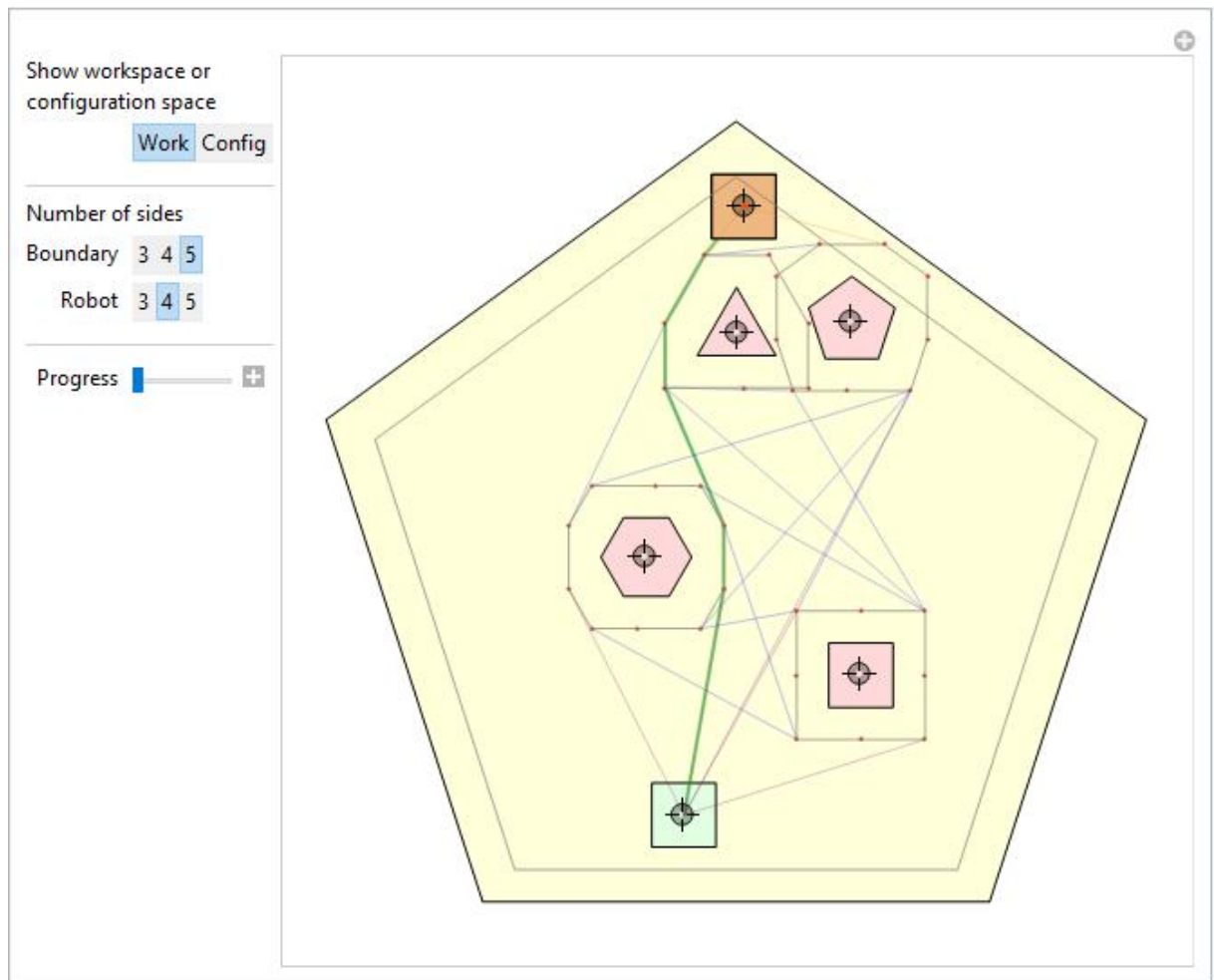


Figure 31: Motion Planning: Example 3 shows the work space of a square robot in a pentagonal boundary environment.

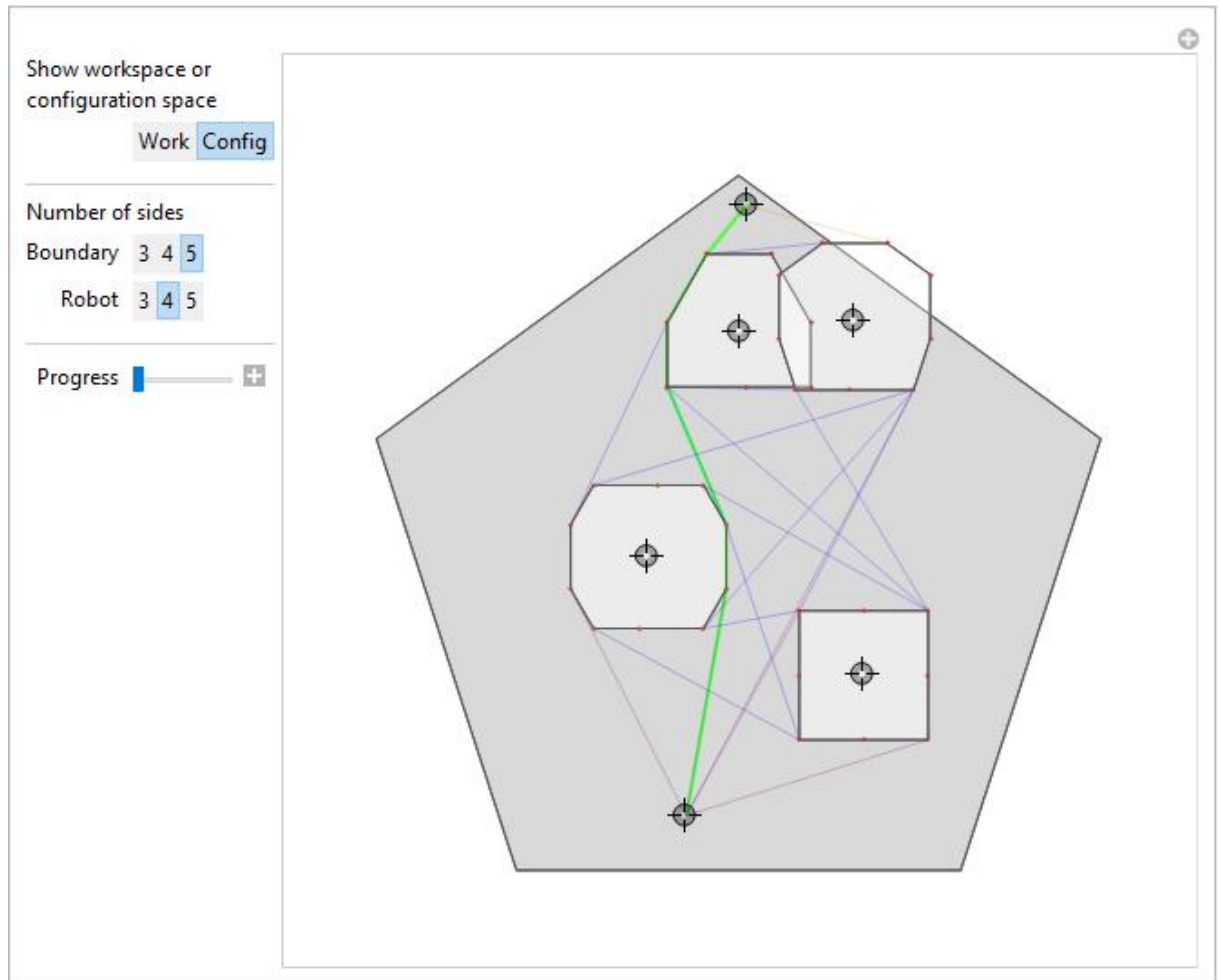


Figure 32: Motion Planning: Configuration space for Example 3.

8 Conclusion

For this thesis, five Mathematica demonstrations were generated, out of which four demonstrations are already published on the Wolfram demonstration project, with the fifth demonstration currently under review.

Students who examine demonstrations after understanding the concepts are more likely to remember them, and learning concepts is on the student making correct observations [9]. Online demonstrations provide comprehensive views of the concepts. Students who watch online videos learn more and enjoy going through it themselves. We can conclude that online demonstrations are better interactive learning tools [6].

These online demonstrations help visualize and illustrate complex science, engineering and technological concepts. In robotics, these online demonstrations help in understand the robot configuration space and observe different boundary conditions and can also simulate the workspace. Complex robotic algorithms like “Minkowski sum of convex polygons”, “visibility graph”, and “A-star algorithms” can be learned, and specific cases can be studied. Having such online demonstrations for learning is a resource for students and enthusiasts all around the world.

Bibliography

- [1] CHVATAL, V. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B* 18, 1 (1975), 39–41.
- [2] D.-T., L. *Proximity and reachability in the plane*. PhD thesis, Ph. D. Thesis, Dept. of Electrical Engineering, University of Illinois at ... , 1979.
- [3] FISK, S. A short proof of chvátal’s watchman theorem.
- [4] GRAHAM, R. L. An efficient algorithm for determining the convex hull of a finite planar set. *Info. Pro. Lett.* 1 (1972), 132–133.
- [5] KAHN, J., KLAWE, M., AND KLEITMAN, D. Traditional galleries require fewer watchmen. *SIAM Journal on Algebraic Discrete Methods* 4, 2 (1983), 194–206.
- [6] KESTIN, G., MILLER, K., MCCARTY, L. S., CALLAGHAN, K., AND DESLAURIERS, L. Comparing the effectiveness of online versus live lecture demonstrations. *Physical Review Physics Education Research* 16, 1 (2020), 013101.
- [7] LAVALLE, S. M. *Planning algorithms*. Cambridge university press, 2006.
- [8] MANO, M. M., AND KIME, C. R. *Logic and Computer Design Fundamentals 2nd Edition Updated*, vol. 6. Chapter, 2001.
- [9] MILLER, K., LASRY, N., CHU, K., AND MAZUR, E. Role of physics lecture demonstrations in conceptual learning. *Physical review special topics-physics education research* 9, 2 (2013), 020113.
- [10] POYREKAR, S., AND BECKER, A. T. Base conversions from base 2 through 100 using radix points, Feb 2020.

- [11] POYREKAR, S., SULTANA, A., AND BECKER, A. T. Art gallery problem, Sept 2019.
- [12] POYREKAR, S., SULTANA, A., AND BECKER, A. T. Minkowski sum of convex robot and obstacle, Apr 2019.
- [13] POYREKAR, S., SULTANA, A., AND BECKER, A. T. Visibility region of a polygon, Sept 2019.
- [14] SACK, J.-R., AND TOUSSAINT, G. T. Guard placement in rectilinear polygons. In *Machine Intelligence and Pattern Recognition*, vol. 6. Elsevier, 1988, pp. 153–175.
- [15] URRUTIA, J. Art gallery and illumination problems. In *Handbook of computational geometry*. Elsevier, 2000, pp. 973–1027.