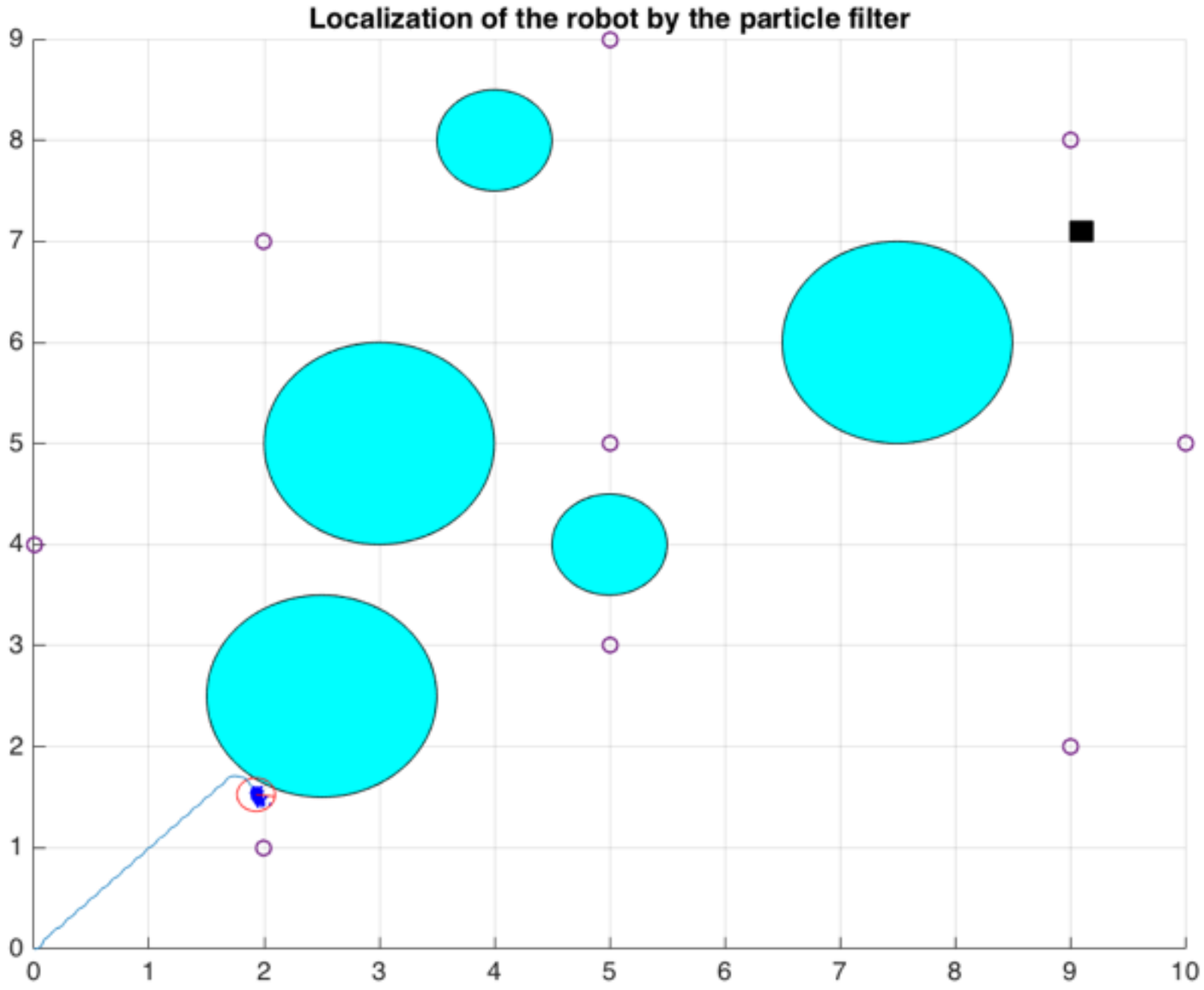


A* based motion planning and Particle filters

By Purushottam Sharma

CS685 Fall15 Final Project

Introduction

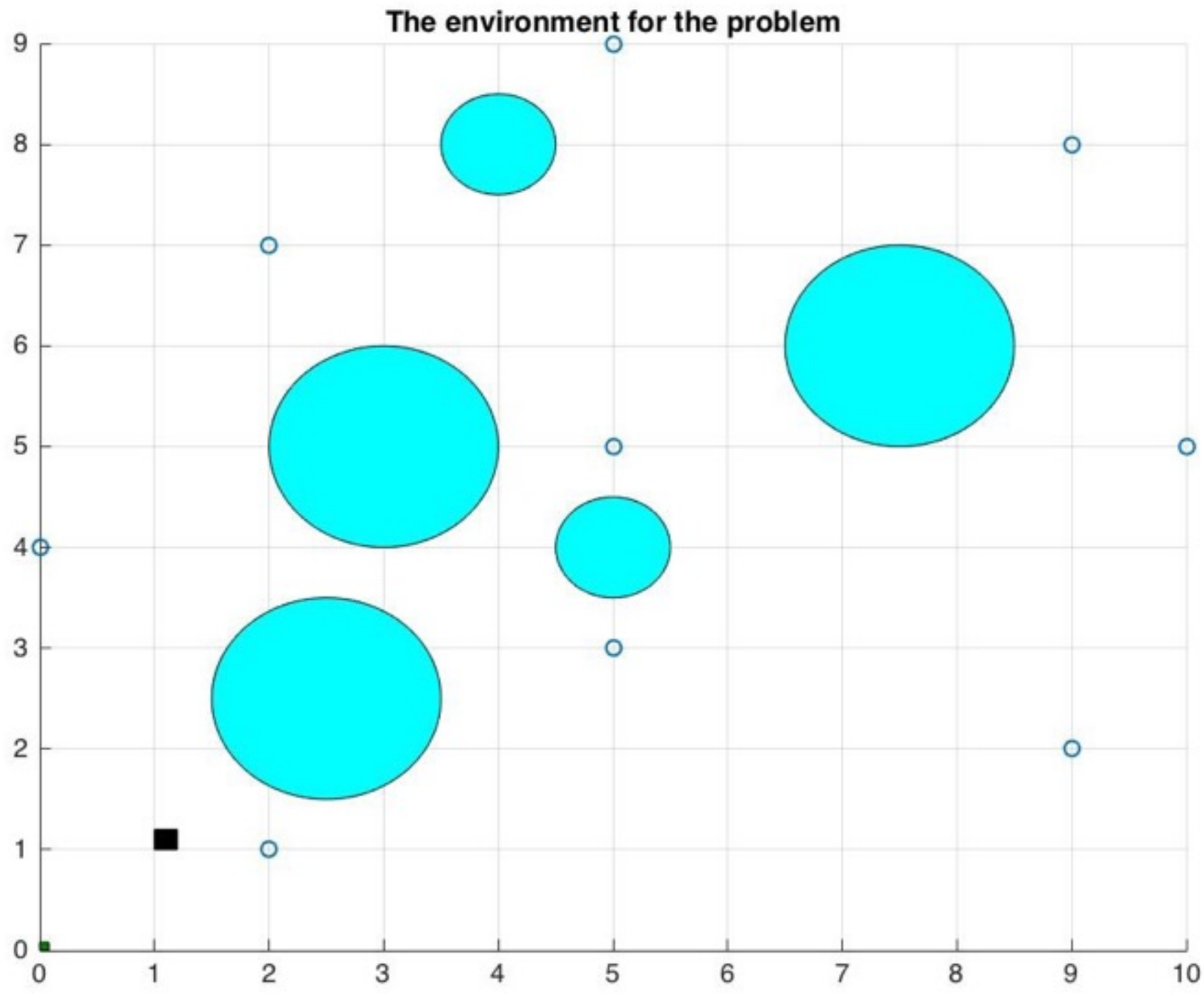
- The task to move robot from a start to a goal position is fundamental to robotics
- Obstacles may come in the way
- An efficient path needs to be taken
- we need to keep track of robot's location so that it does not get lost

Problem

Given a robot at an initial position, compute, how to gradually move it to a desired goal position over an efficient/shortest path while avoiding collision with obstacles. And track the robot with particle filter as it moves along the path.

The environment for the problem is 2D world, $W = R^2$ and $O \subset W$ is the obstacle region, which has a piecewise-linear (polygonal) boundary. The robot is a point that can move through the world, but must avoid touching the obstacles.

Environment



Motion planning

- Device some control strategy that directs the robot towards the goal position.

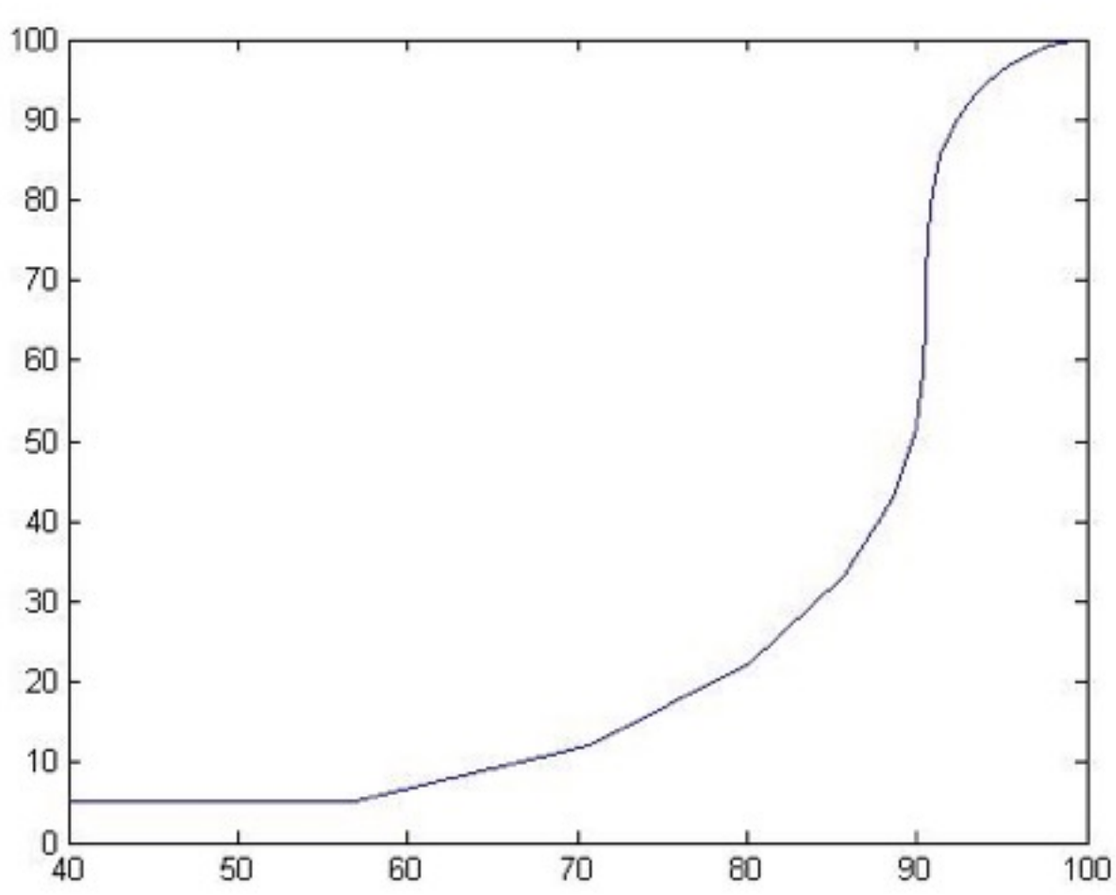
- Must avoid collision with the obstacles

- Must find an efficient path to the goal

Motion planning

Naive approach:

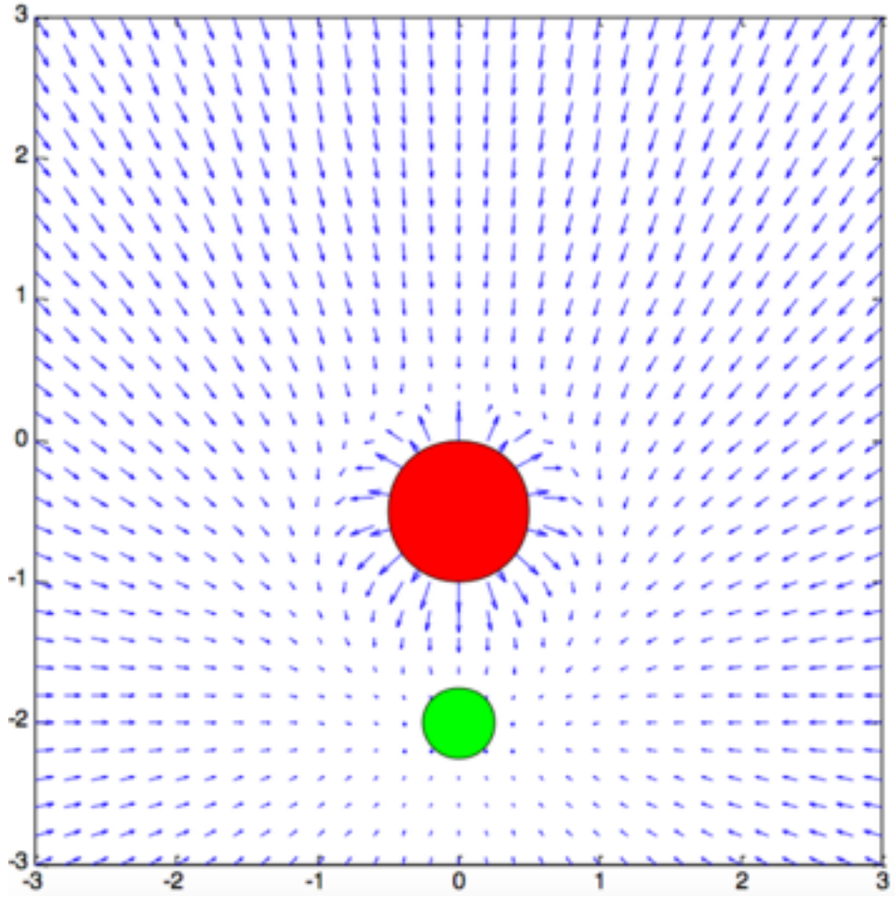
- Simply use the distance between the start and the goal to direct the robot
- Simple but inefficient
- Does not take obstacles into consideration



Motion planning

Potential fields method:

- Environment is represented as a potential field
- Assign repulsive field to obstacles and attractive to goal
- Negative gradient of potential at any point guides the robot

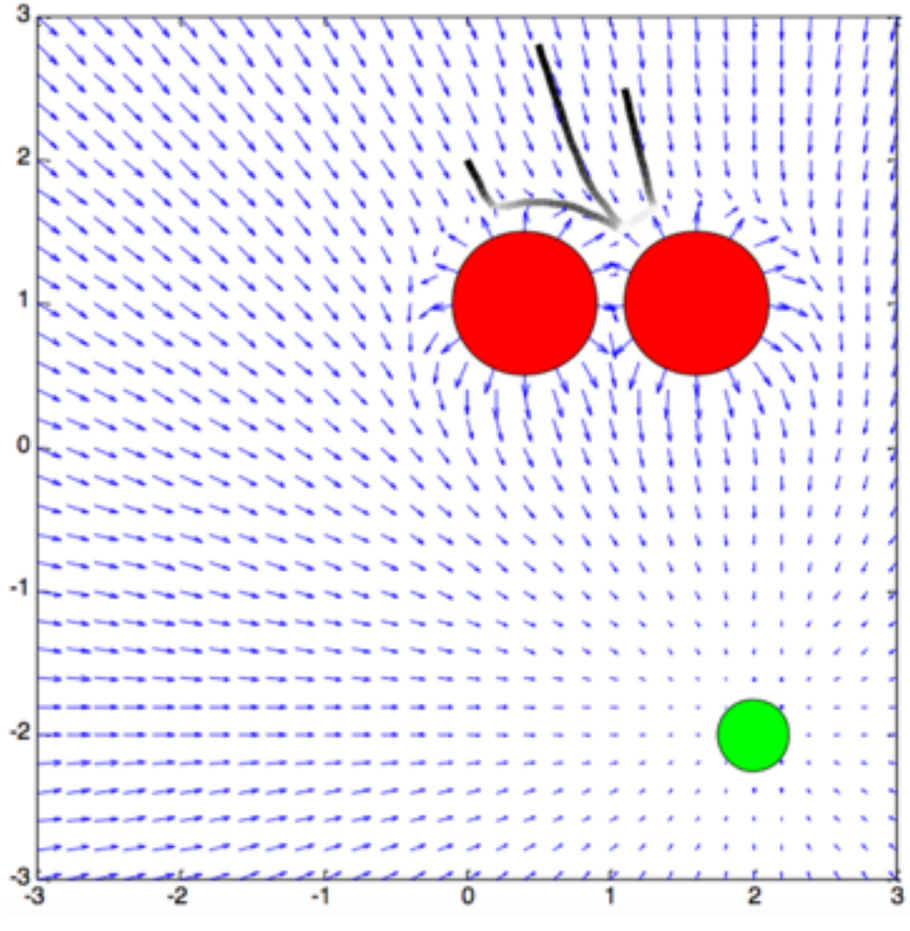


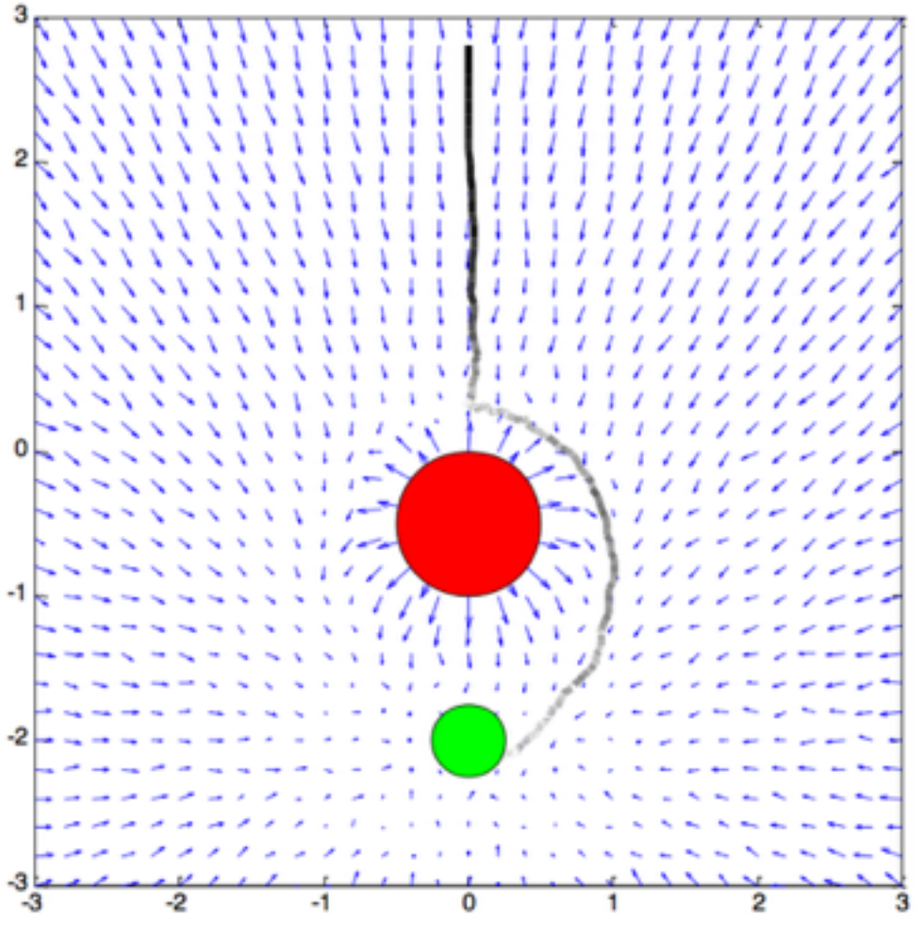
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.431.9504&rep=rep1&type=pdf

Motion planning

Potential fields method:

- Robot can get stuck in a local minima
- Does not always find the best path possible





http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.431.9504&rep=rep1&type=pdf

Motion planning

A* algorithm:

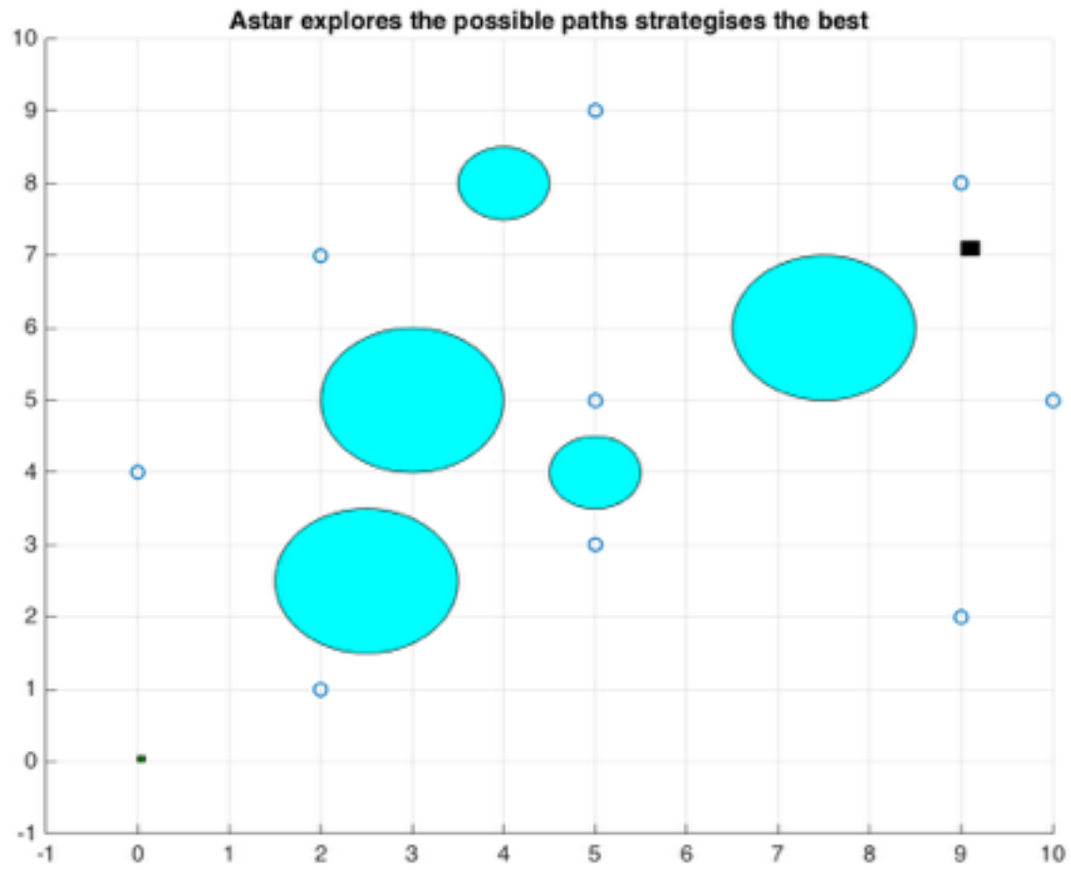
- A popular algorithm for finding shortest path in graphs
- Employs a cost function, which combines a heuristic estimate of the cost to reach a goal and the distance traveled from the initial node
- I modified the algorithm to fit the problem
    - The environment is represented as a grid. Each cell acts as a separate node.
    - To move across the grid from one cell to another, moves are defined: horizontal, vertical and diagonal
    - The heuristic is the euclidean distance. f(n) is the distance between current node n and goal. g(n) is the distance traveled between n and start.

$$cost = f(n) + g(n)$$

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

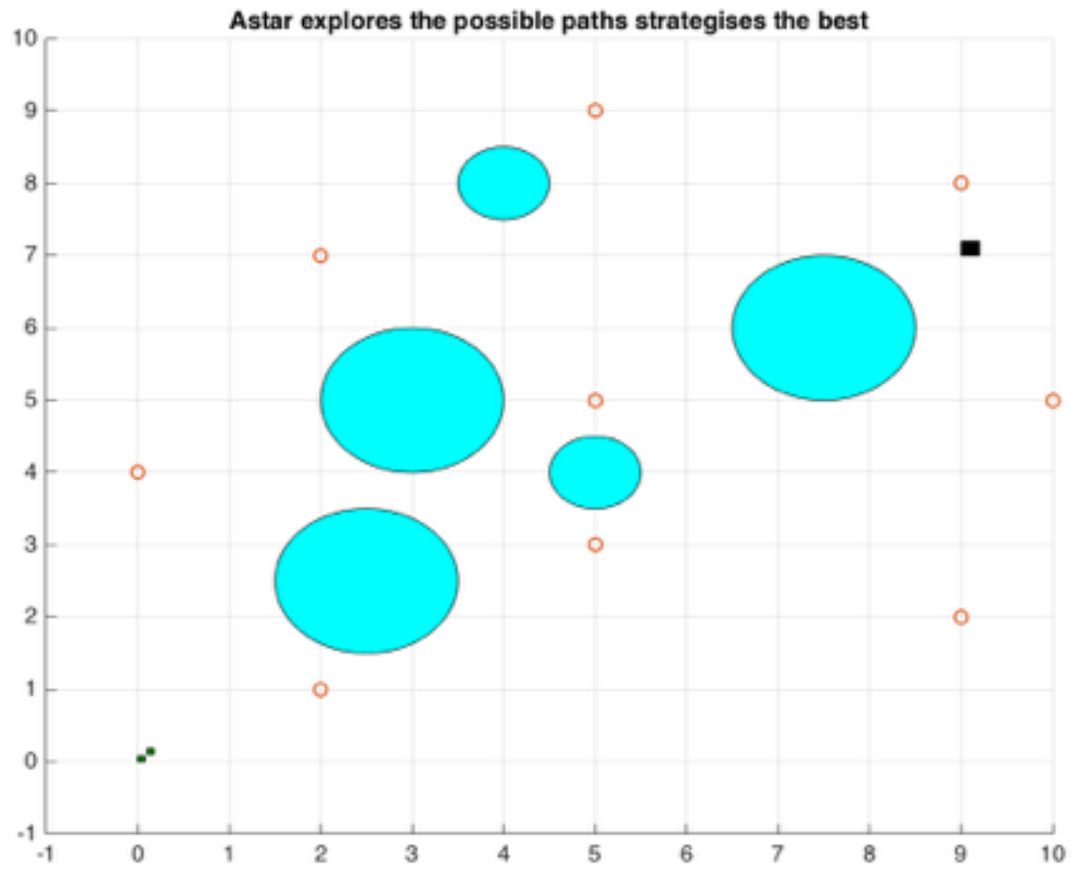


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

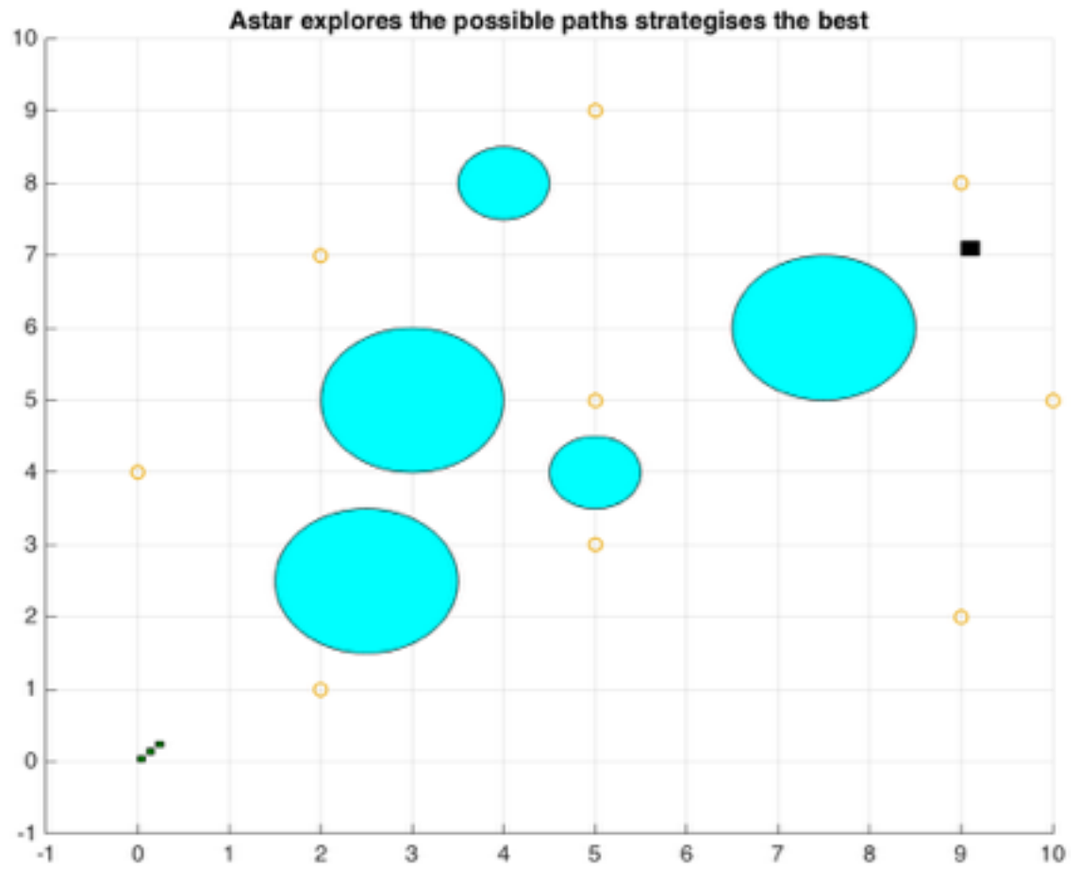


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

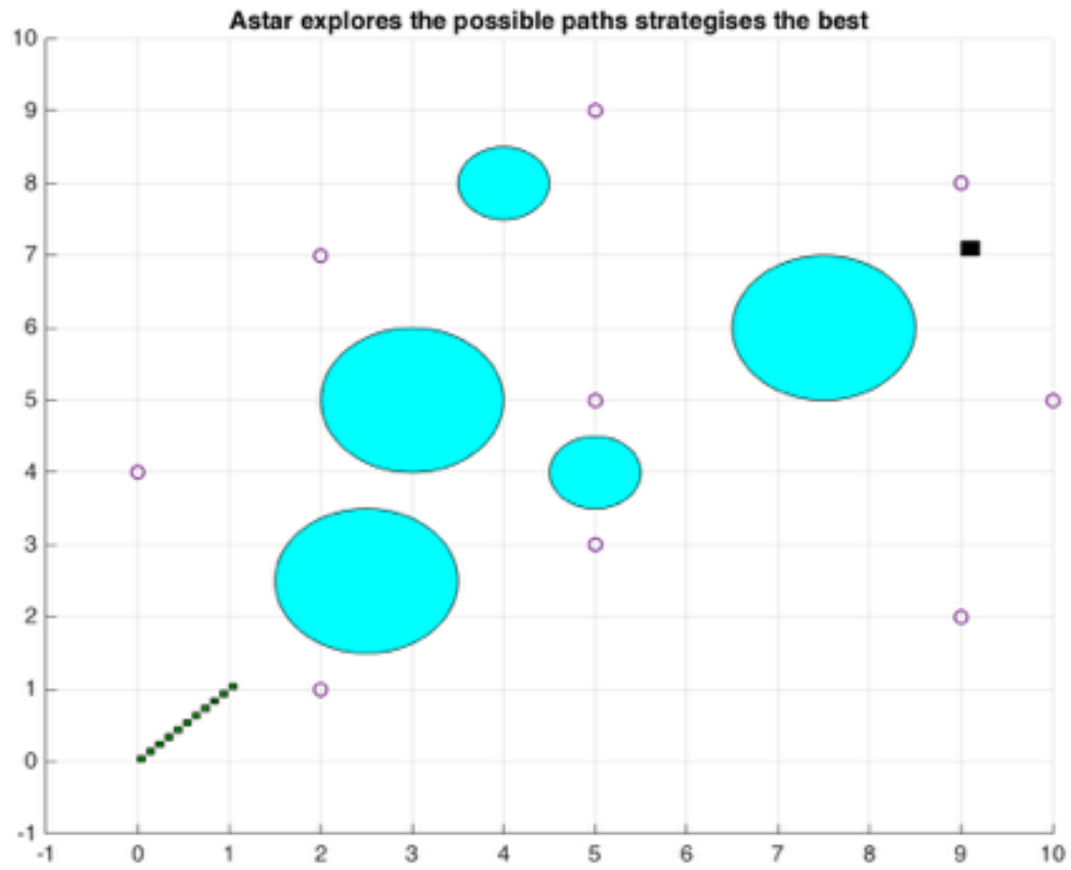


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

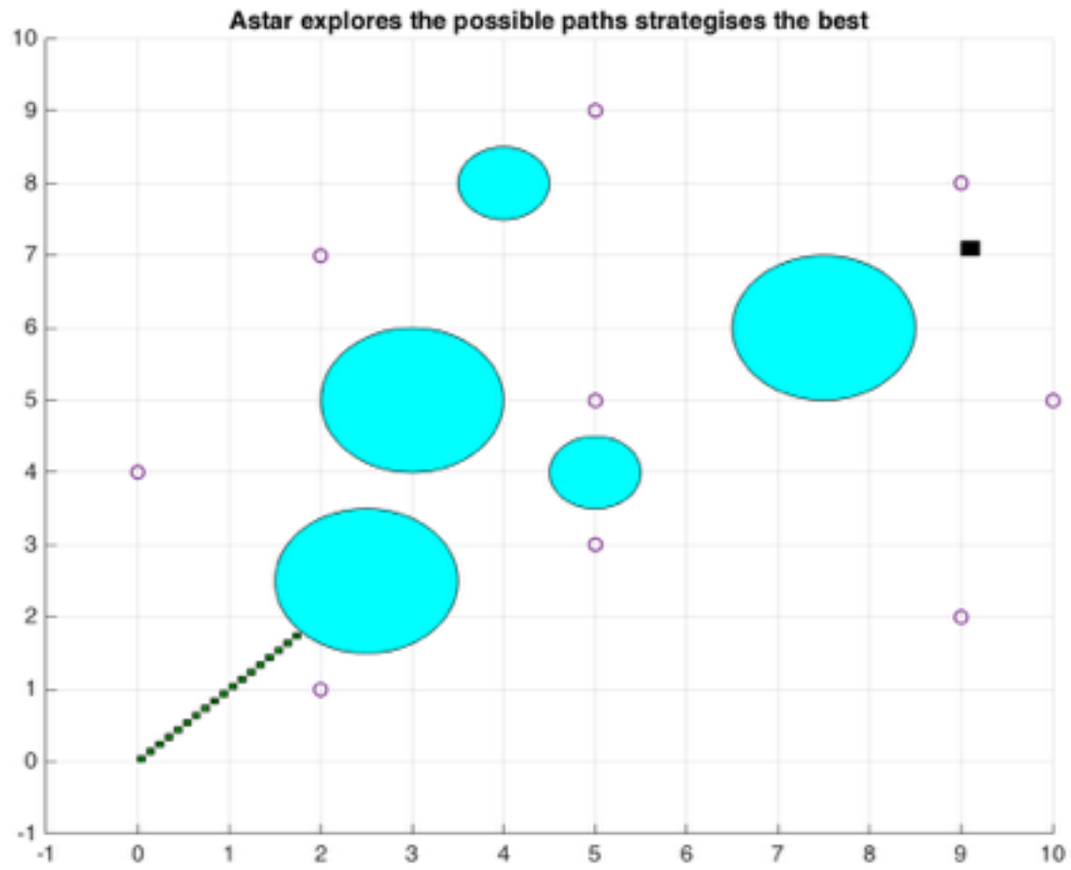


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

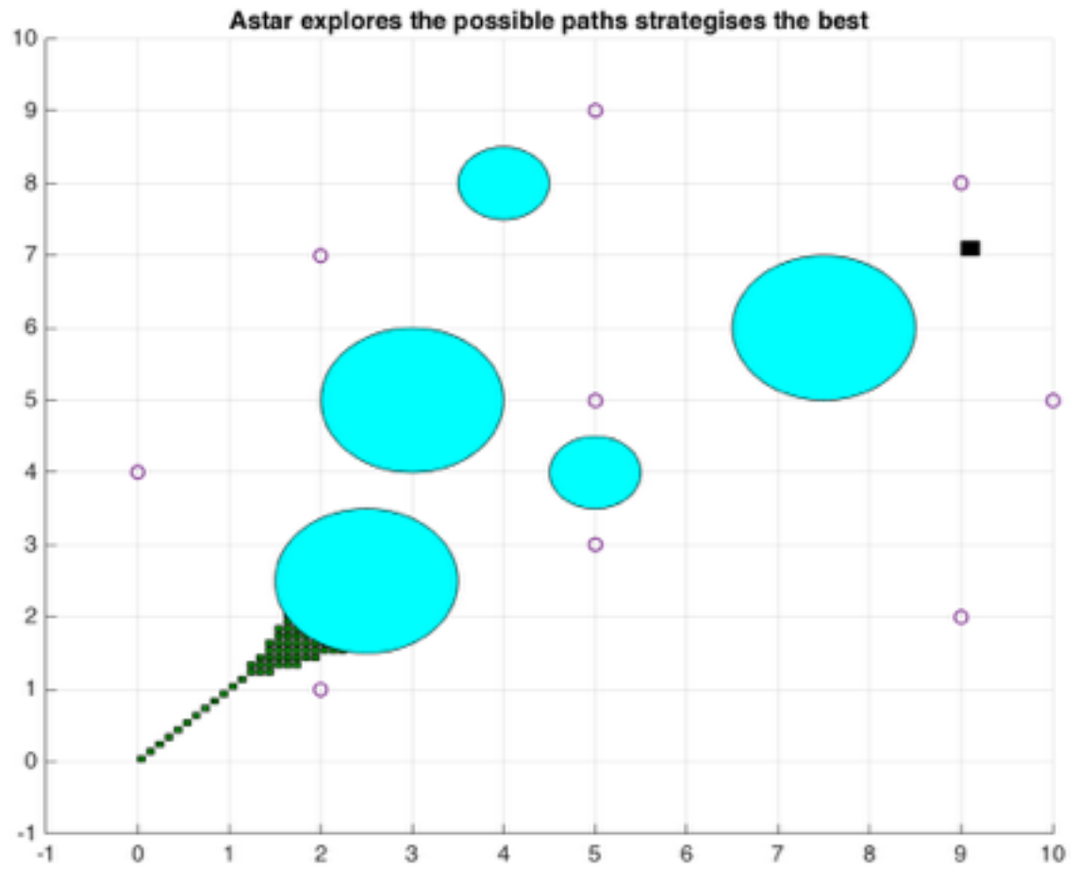


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

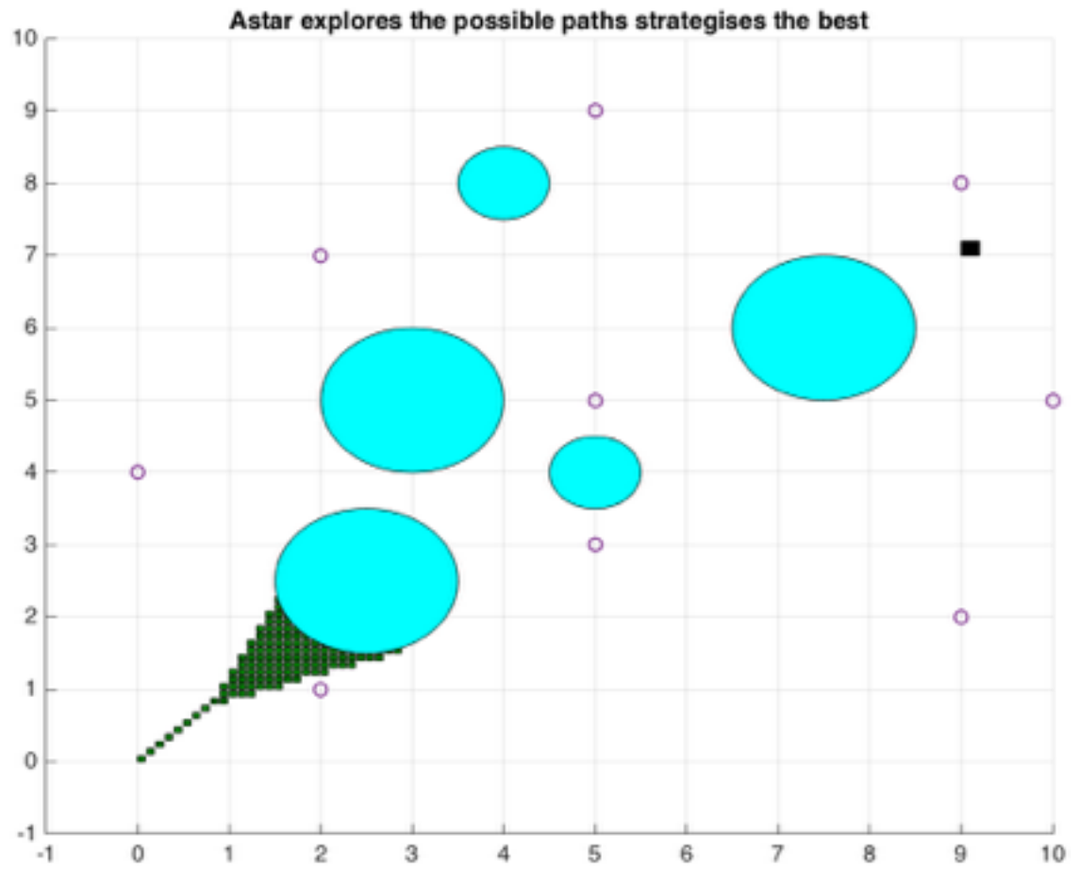


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

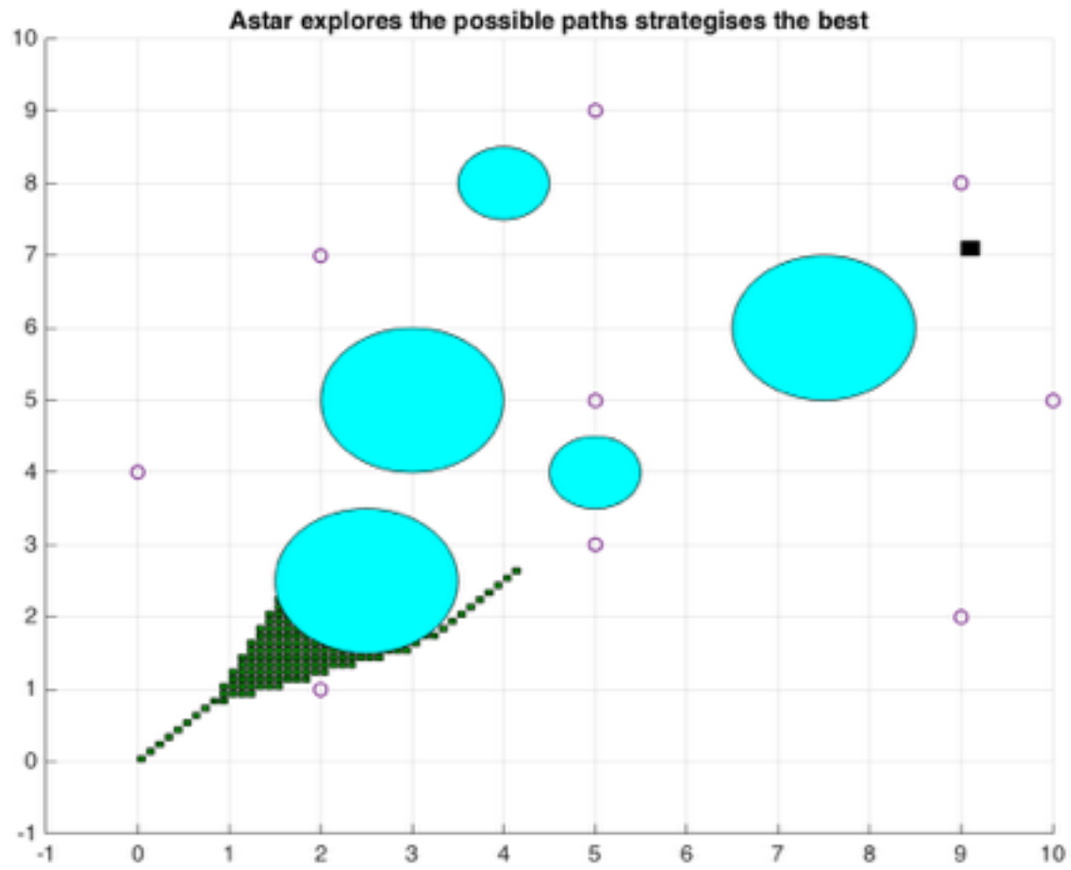


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

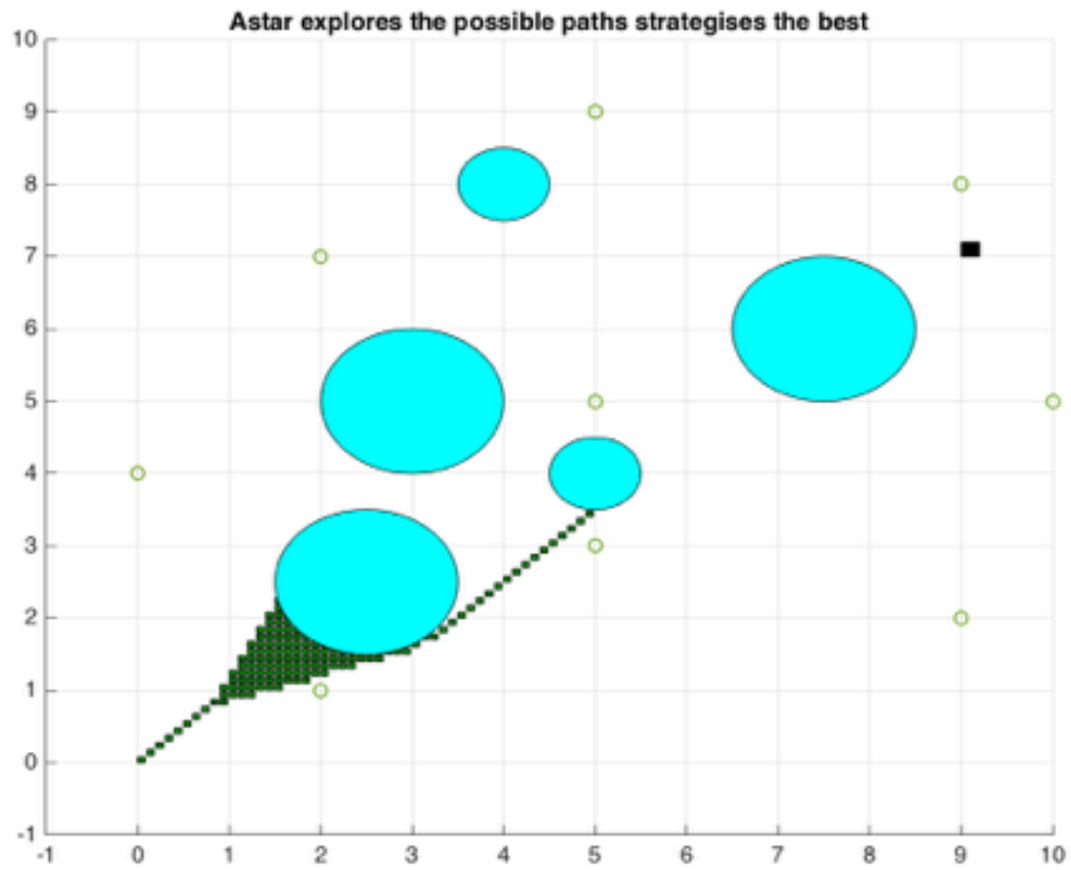


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

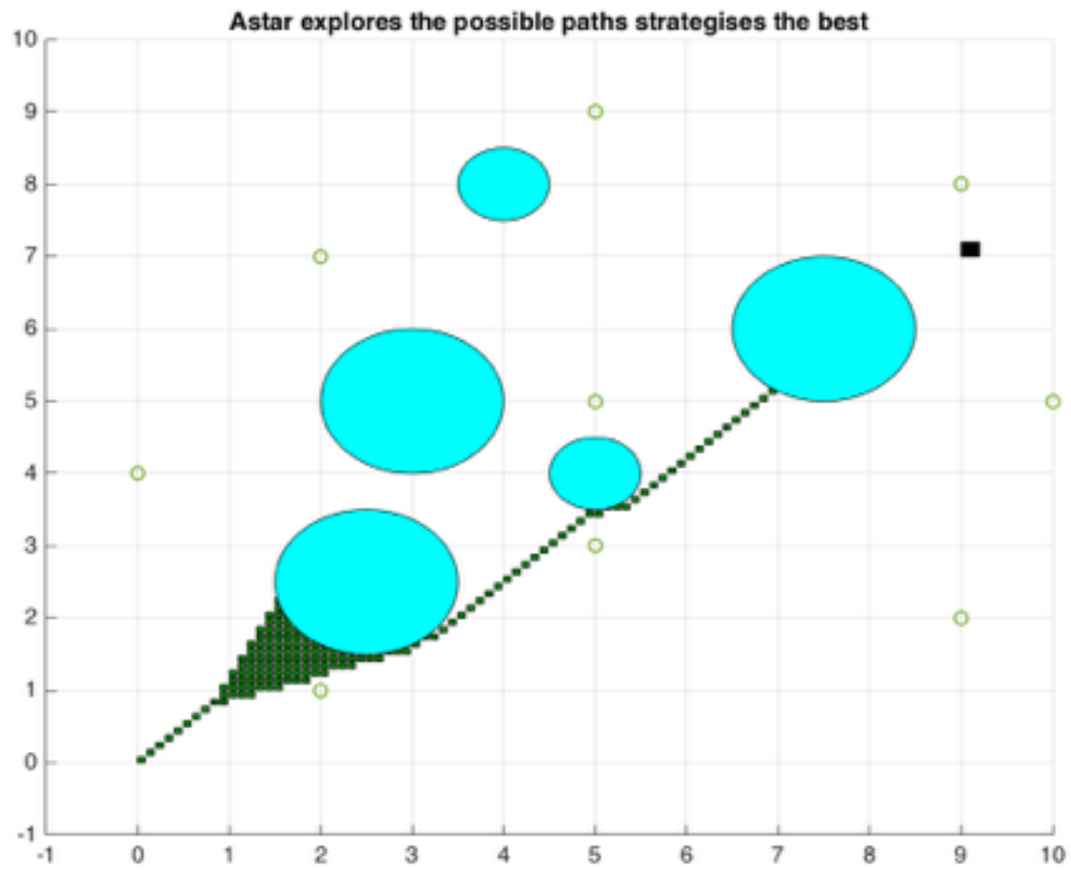


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

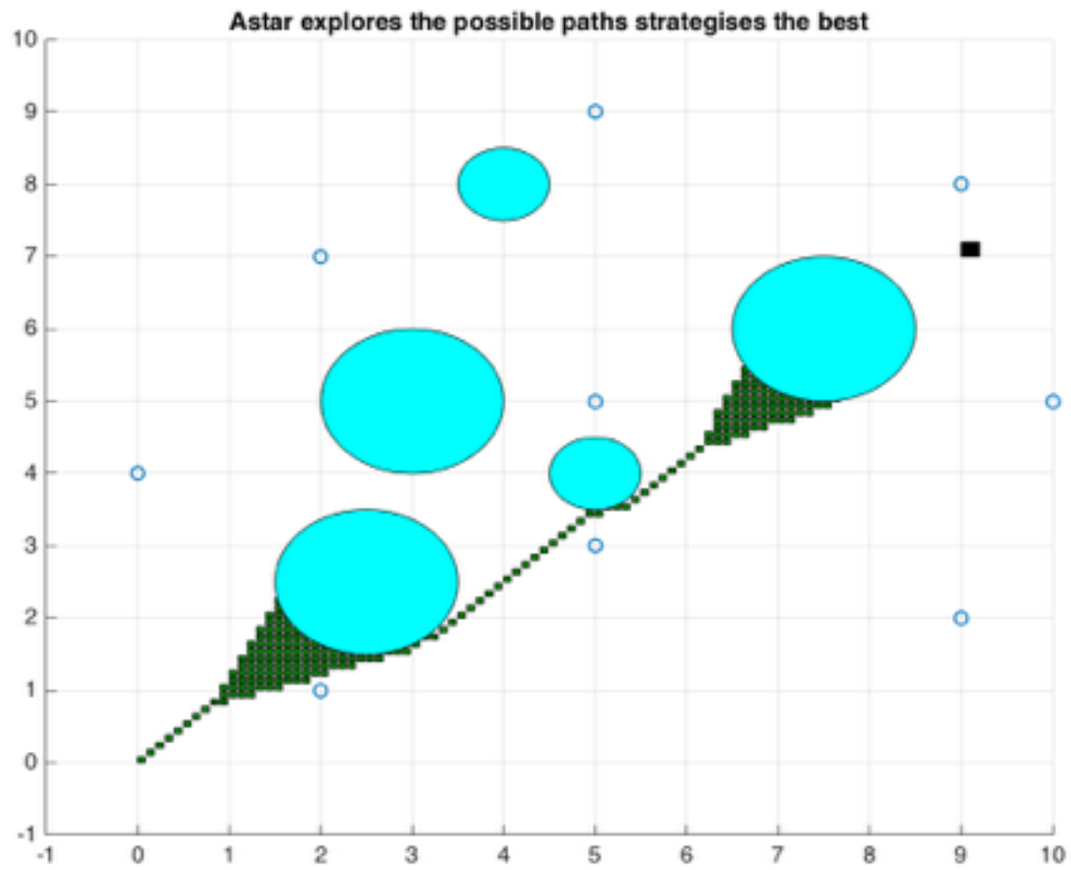


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

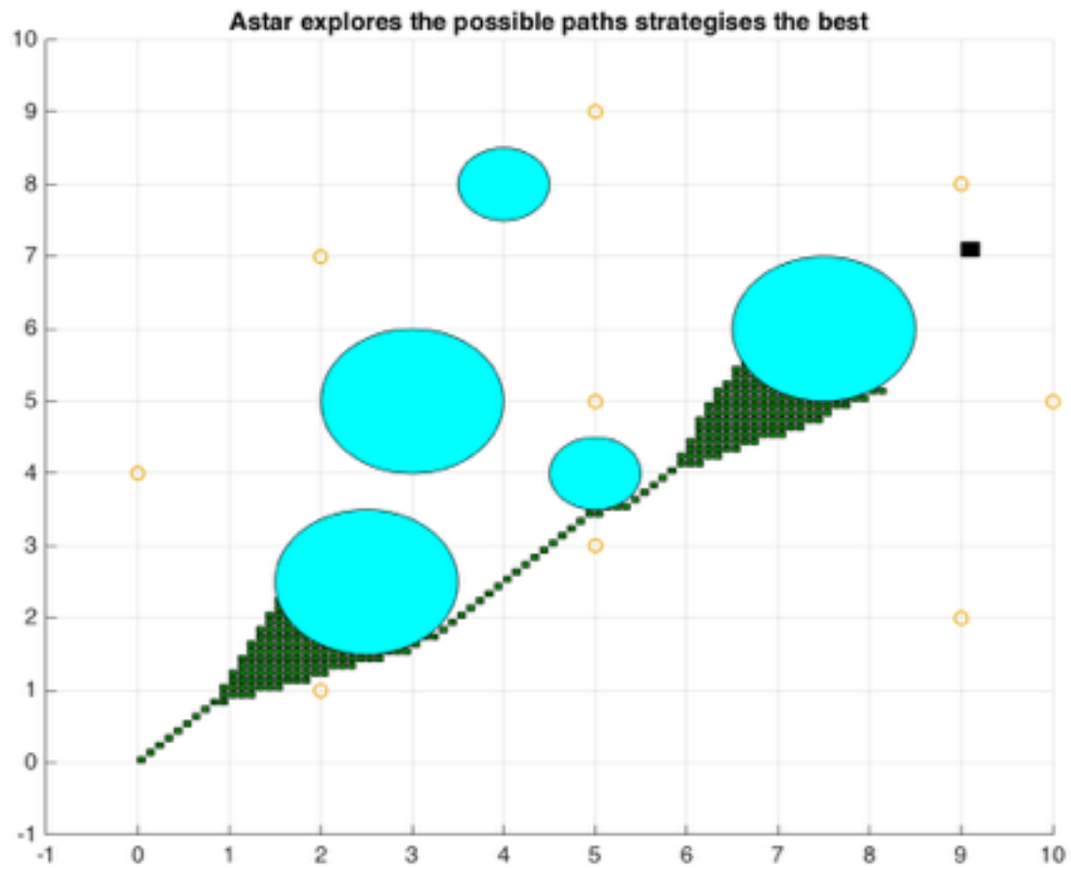


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

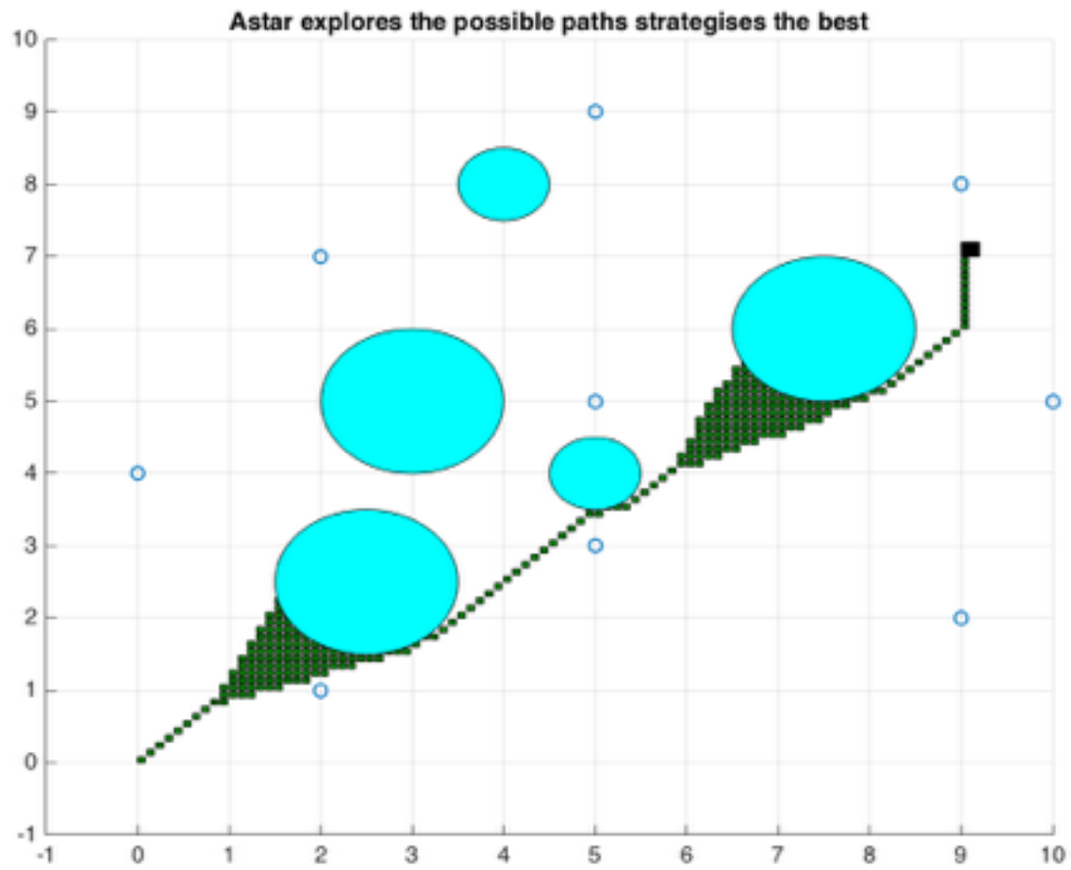


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```

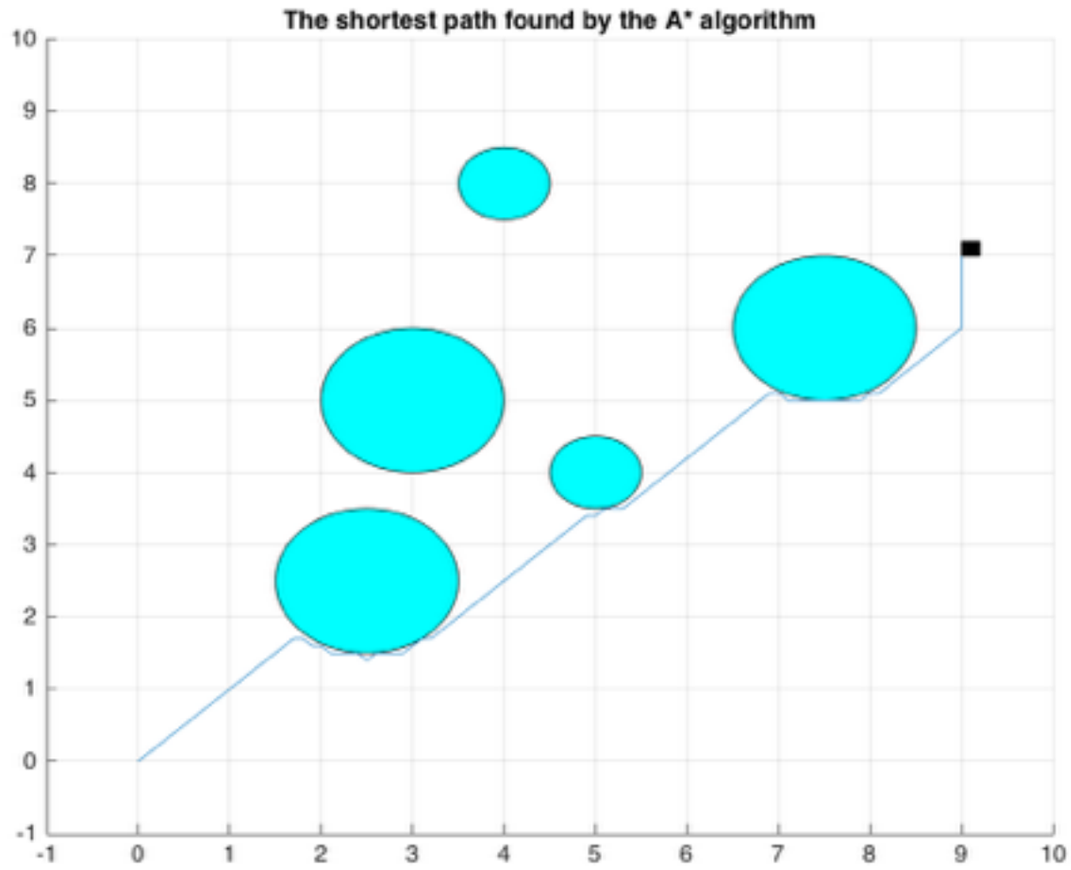


The shortest path found form start(0,0) to goal(9,7)

Motion planning

A*(start, goal, obstacles)

```
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.          current_node ← minimum cost node from
                                OpenSet
9.          add current_node to ClosedSet
10.         remove current_node from OpenSet
11.
12.       for every move in Moves_possible
13.           new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.         add to path the node X from the ClosedSet
                  whose ID == last_parentID
25.         last_parentID ← X.parentID
26.
27. return path
```



The shortest path found form start(0,0) to goal(9,7)

Motion planning

- Move the robot over the path found
    - Use the kinematics equations and control law.(goTo( ))
- Derive the odometry from the robot poses

$$\delta_{trans} = \sqrt{(\bar{x}'-\bar{x})^2 + (\bar{y}'-\bar{y})^2}$$

$$\delta_{rot1} = \text{atan2}(\bar{y}'-\bar{y}, \bar{x}'-\bar{x}) - \bar{\theta}$$

$$\delta_{rot2} = \bar{\theta}'-\bar{\theta} - \delta_{rot1}$$

Particle filter

- Use particle filter to track the robot when it moves along the path
- Odometry calculated earlier is fed to the particle filter along with the sensor data
- The particles are initialized and algorithm is run
- The algorithm has three major steps
  - Sample the next generation for particles using the distribution
  - Compute the importance weights :
    weight = target distribution / proposal distribution
  - Resampling: “Replace unlikely samples by more likely ones”

Particle filter

1. Algorithm **particle_filter**(S_{t-1}, u_{t-1} z_t):
2. $S_t = \emptyset, \quad \eta = 0$
3. **For** $i = 1 \ldots n$ ***Generate new samples***
4. Sample index $j(i)$ from the discrete distribution given by w_{t-1}
5. Sample x_t^i from $p(x_t \mid x_{t-1}, u_{t-1})$ using $x_{t-1}^{j(i)}$ and u_{t-1}
6. $w_t^i = p(z_t \mid x_t^i)$ ***Compute importance weight***
7. $\eta = \eta + w_t^i$ ***Update normalization factor***
8. $S_t = S_t \cup \{< x_t^i, w_t^i >\}$ ***Insert***
9. **For** $i = 1 \ldots n$
10. $w_t^i = w_t^i / \eta$ ***Normalize weights***

http://cs.gmu.edu/~kosecka/cs685/cs685-discrete-particle-filter.pdf

Particle filter

1. Algorithm **systematic_resampling**(*S*,*n*):

2. $S' = \emptyset, c_1 = w^1$
3. **For** $i = 2 \ldots n$ — ***Generate cdf***
4. $c_i = c_{i-1} + w^i$
5. $u_1 \sim U[0, n^{-1}], i = 1$ — ***Initialize threshold***

6. **For** $j = 1 \ldots n$ — ***Draw samples …***
7. **While** ($u_j > c_i$) — ***Skip until next threshold reached***
8. $i = i + 1$
9. $S' = S' \cup \{ < x^i, n^{-1} > \}$ — ***Insert***
10. $u_{j+1} = u_j + n^{-1}$ — ***Increment threshold***

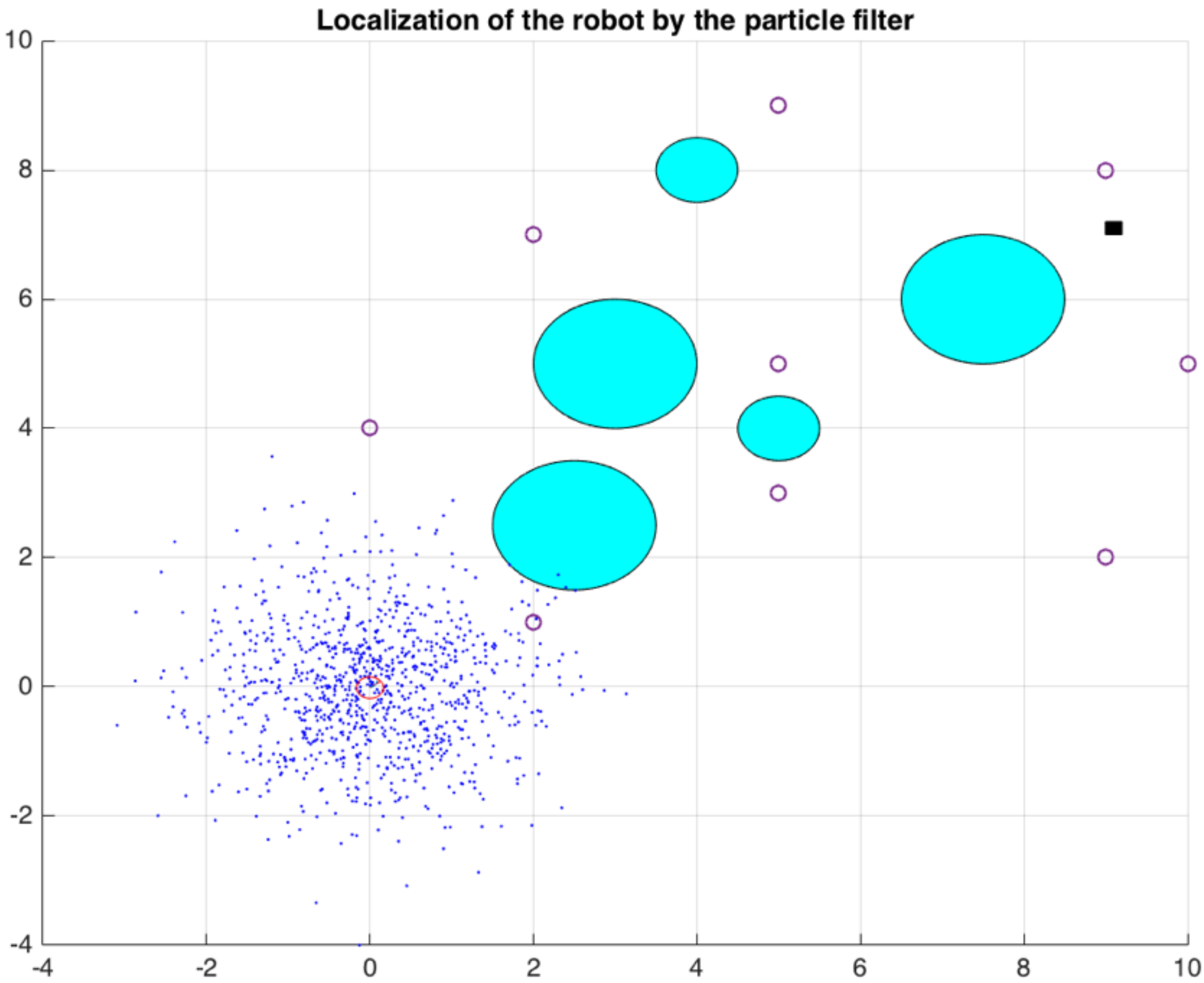
11. **Return** *S'*

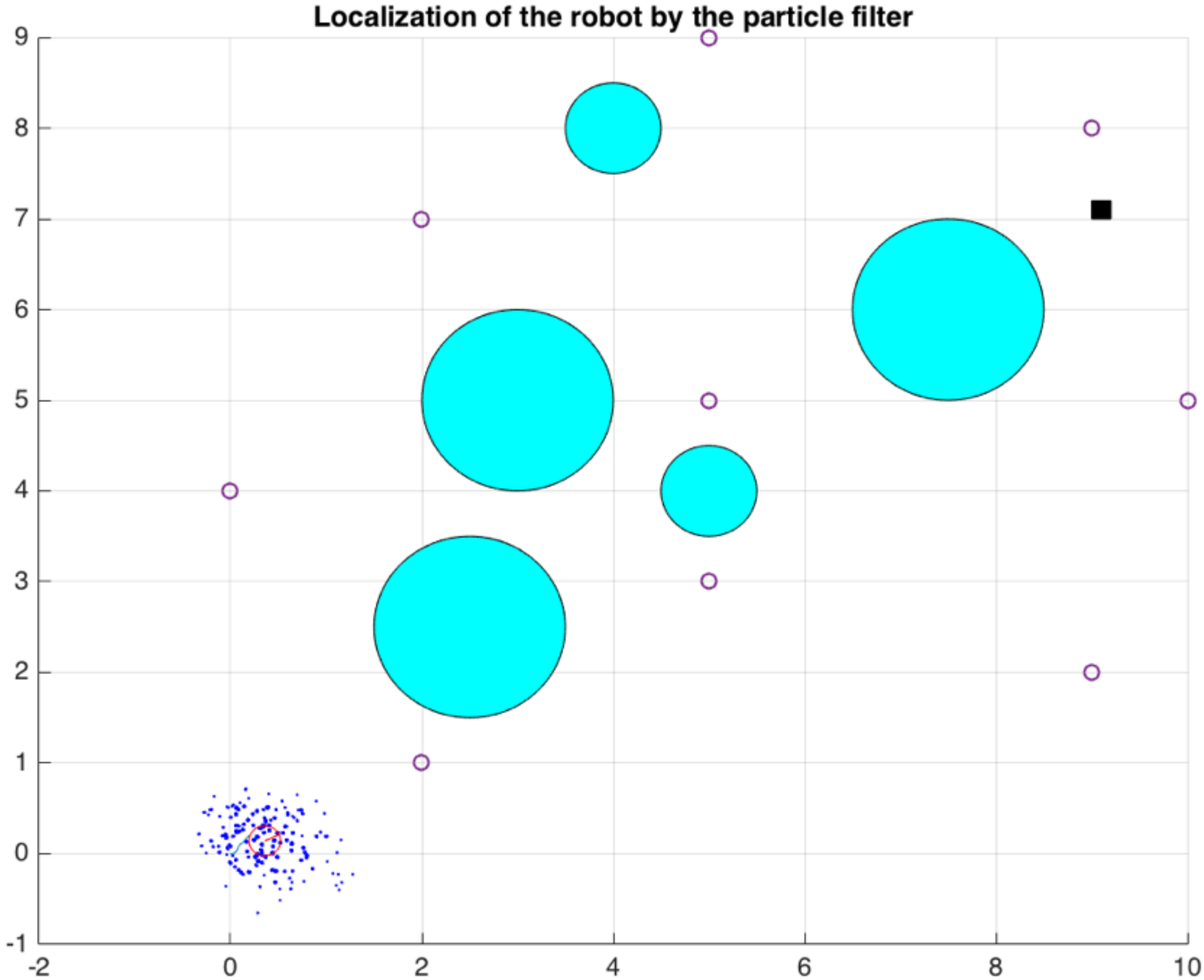
Also called **stochastic universal sampling**

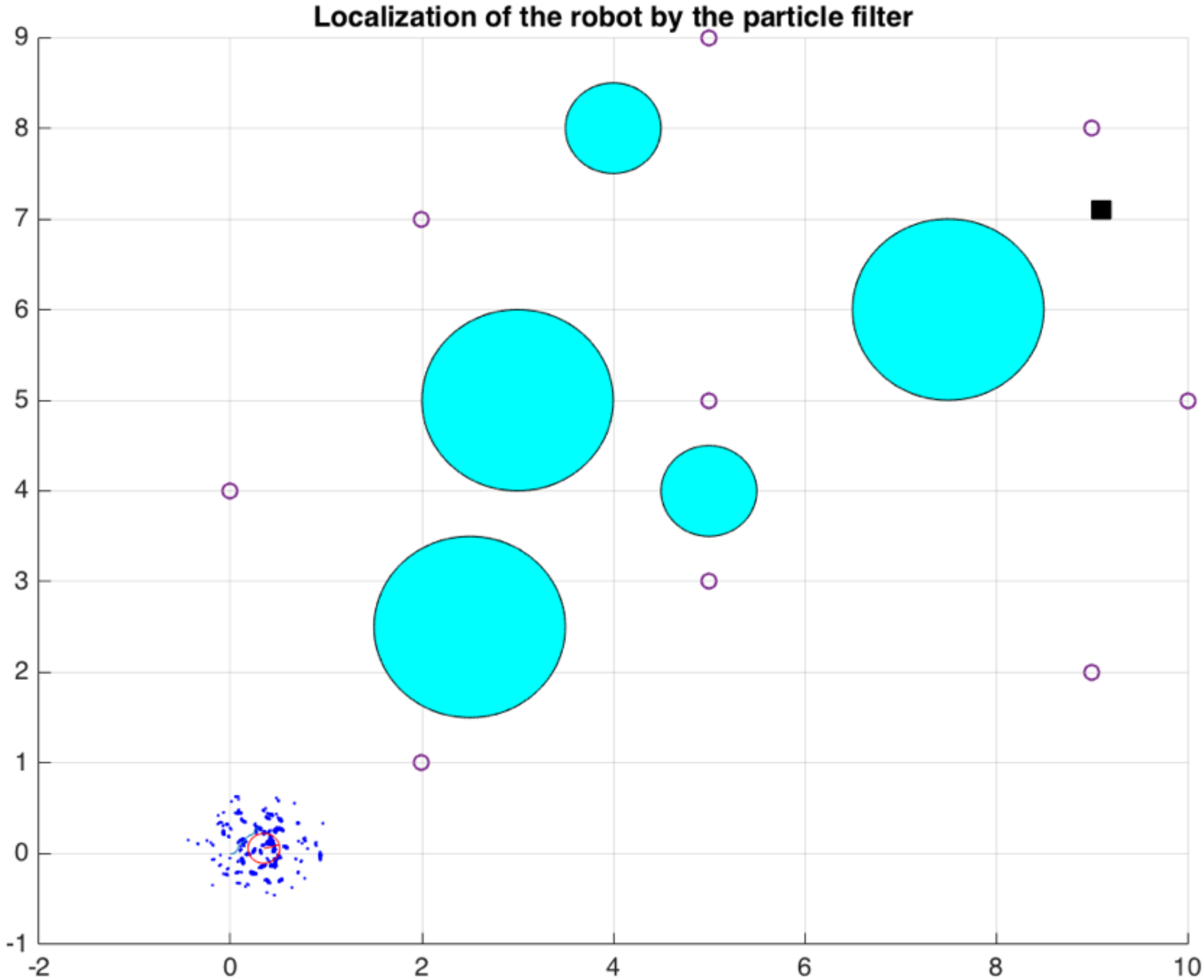
http://cs.gmu.edu/~kosecka/cs685/cs685-discrete-particle-filter.pdf

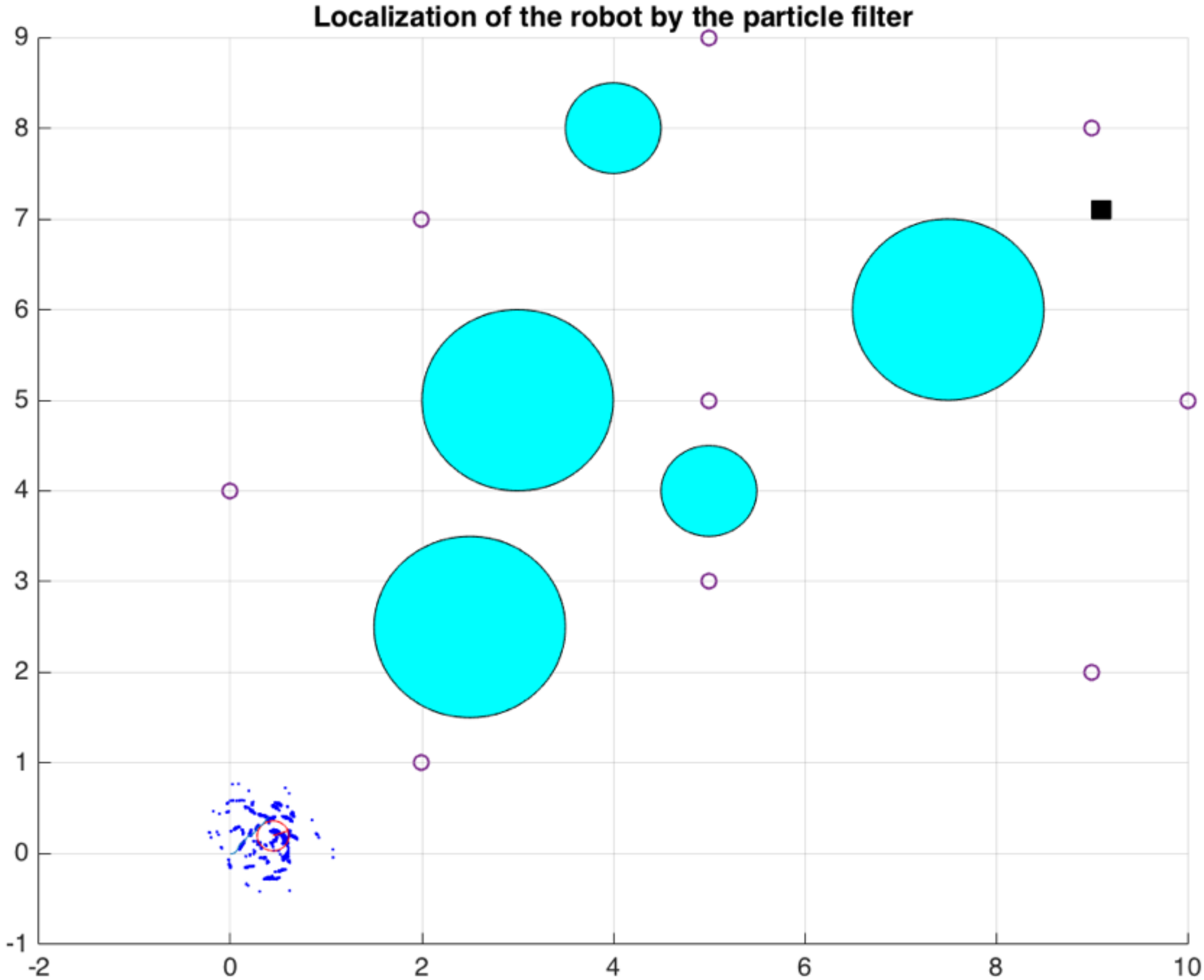
Particle filter

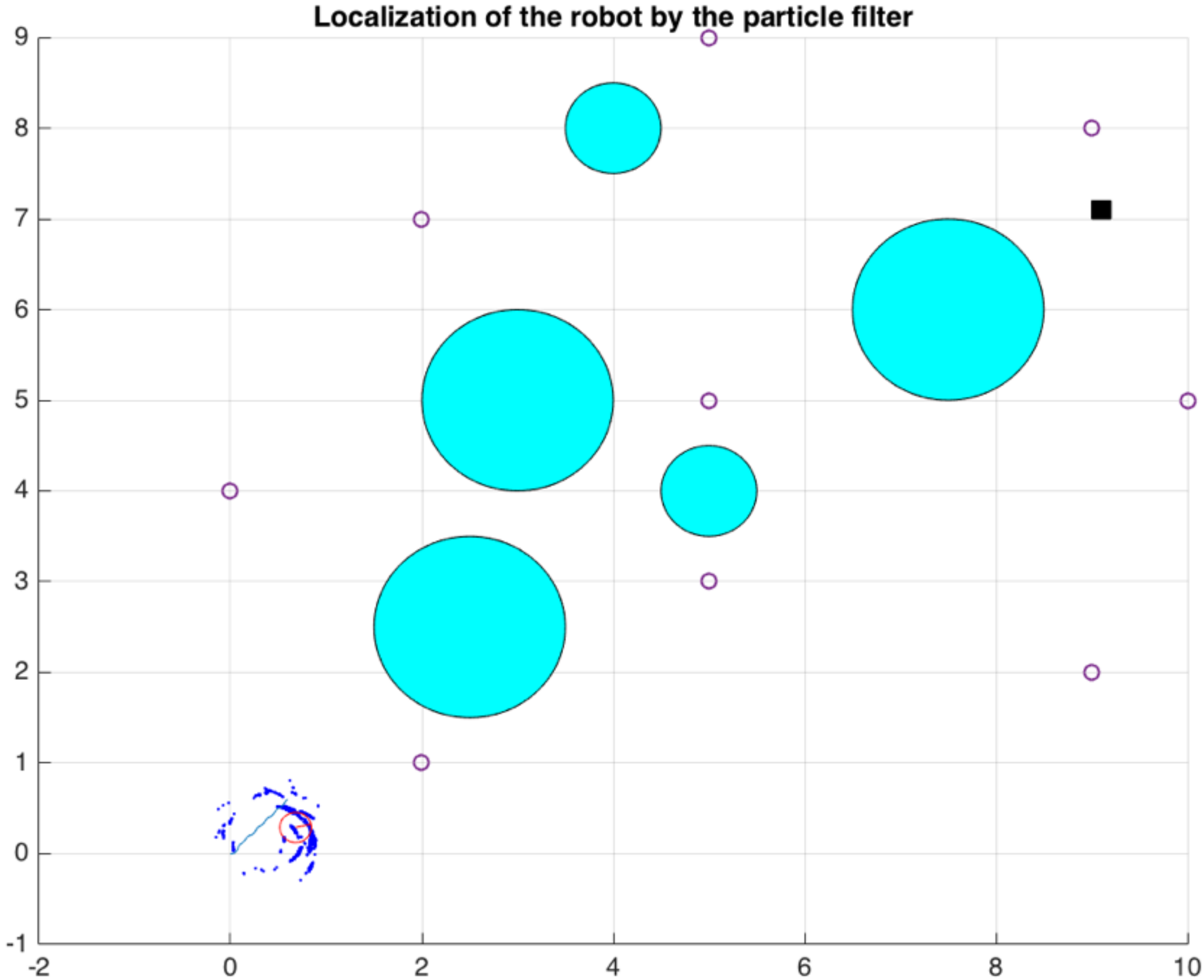
- The Particle filter was employed to track the robot
- The odometry and the precomputed sensor data was provided
- The algorithm initializes the particles and assigns the weights according to the sensor data provided
- However, the sensor data does not help the filter to locate the robot correctly. The sensor readings are not consistent with the measurements during robots motion

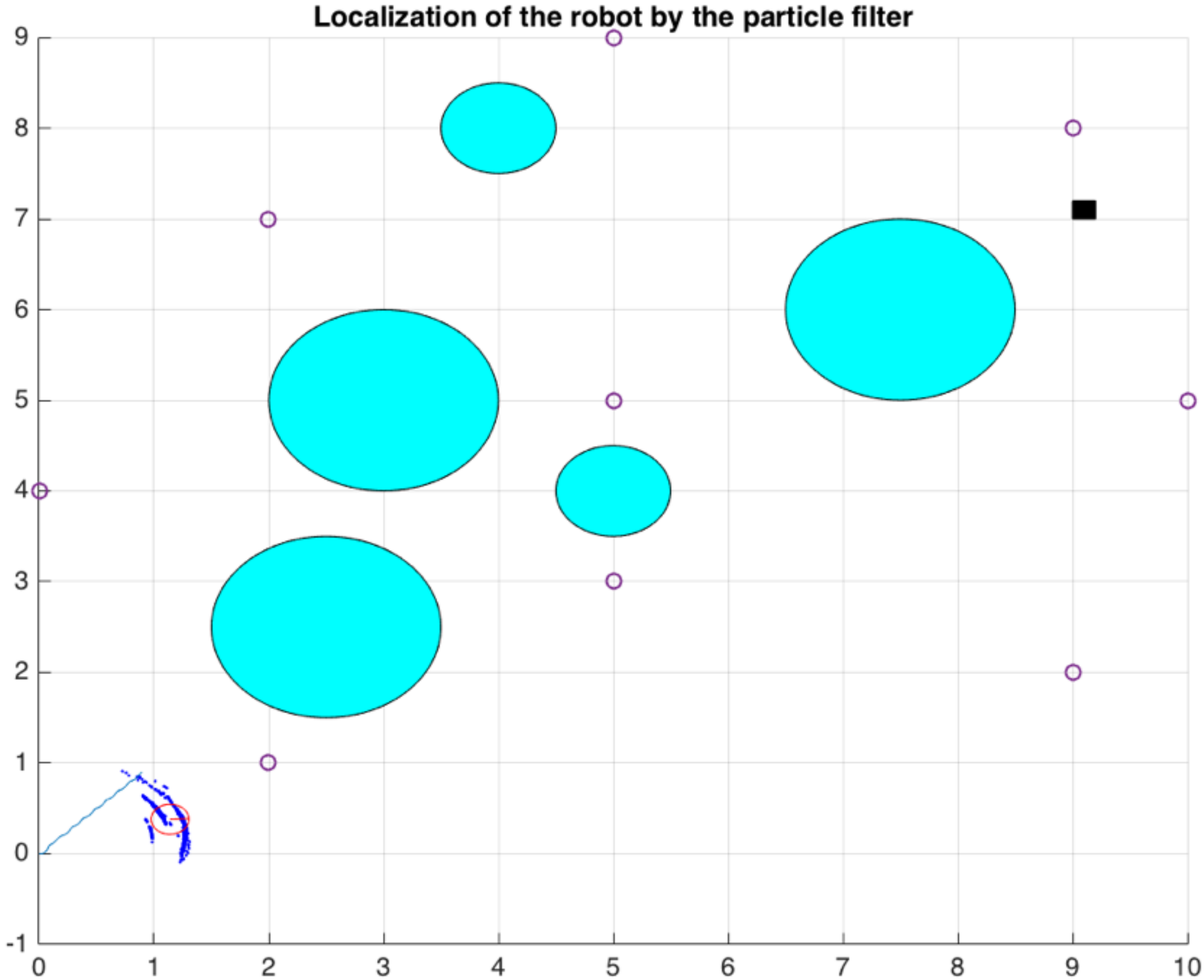

Localization of the robot by the particle filter
10
8
6
4
2
0
-2
-4
-4
-2
0
2
4
6
8
10

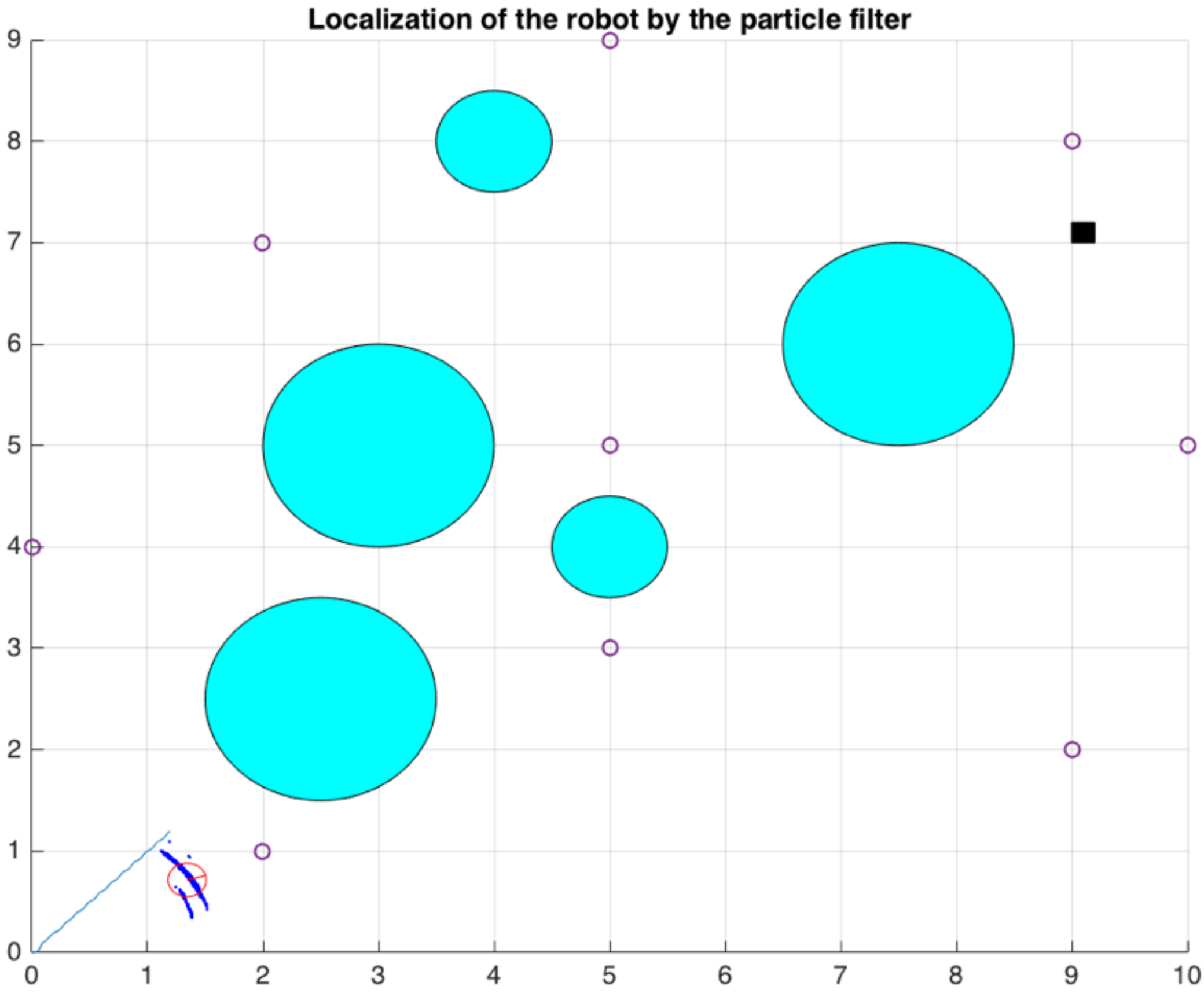

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
-1
-2
0
2
4
6
8
10

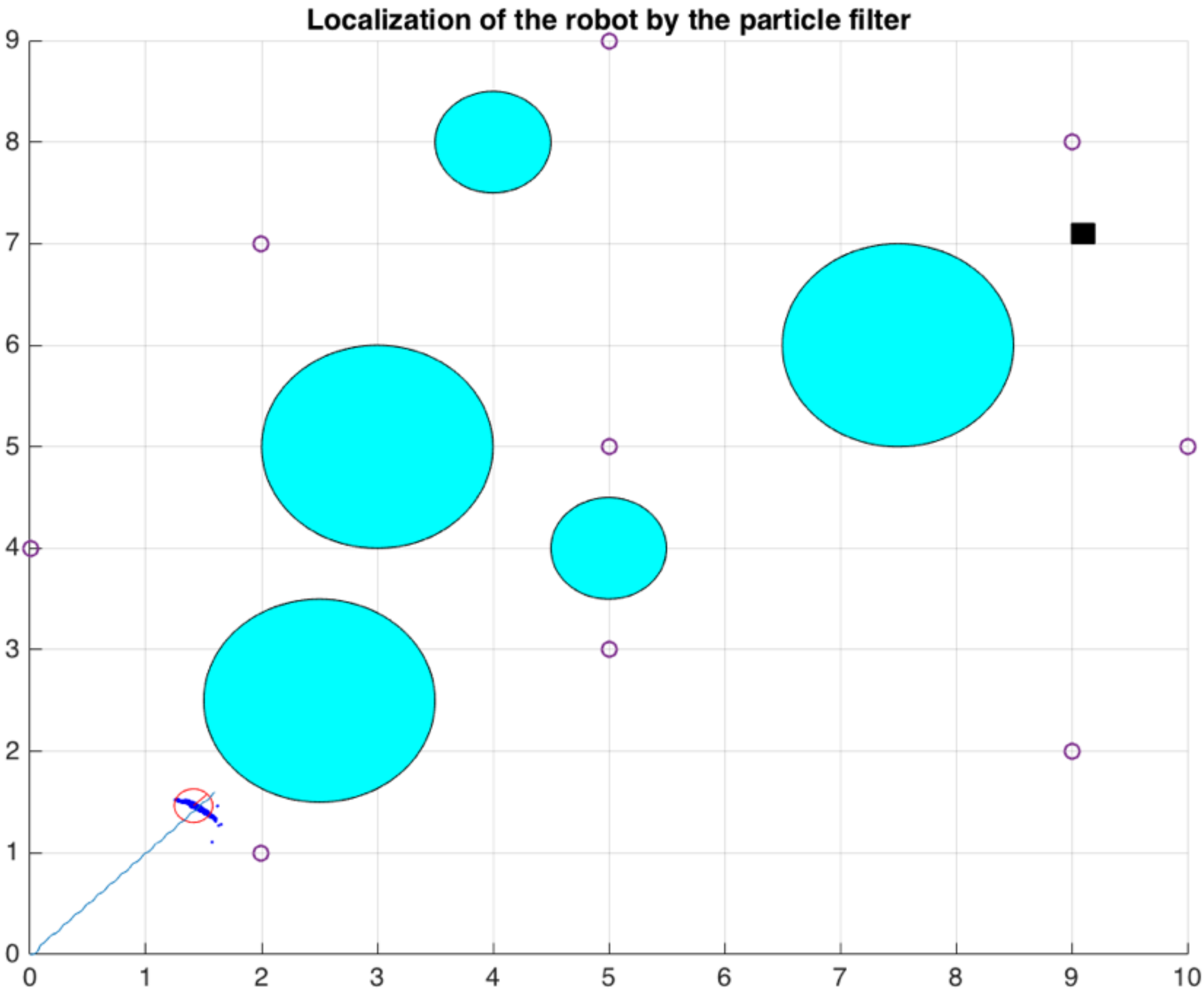

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
-1
-2
0
2
4
6
8
10

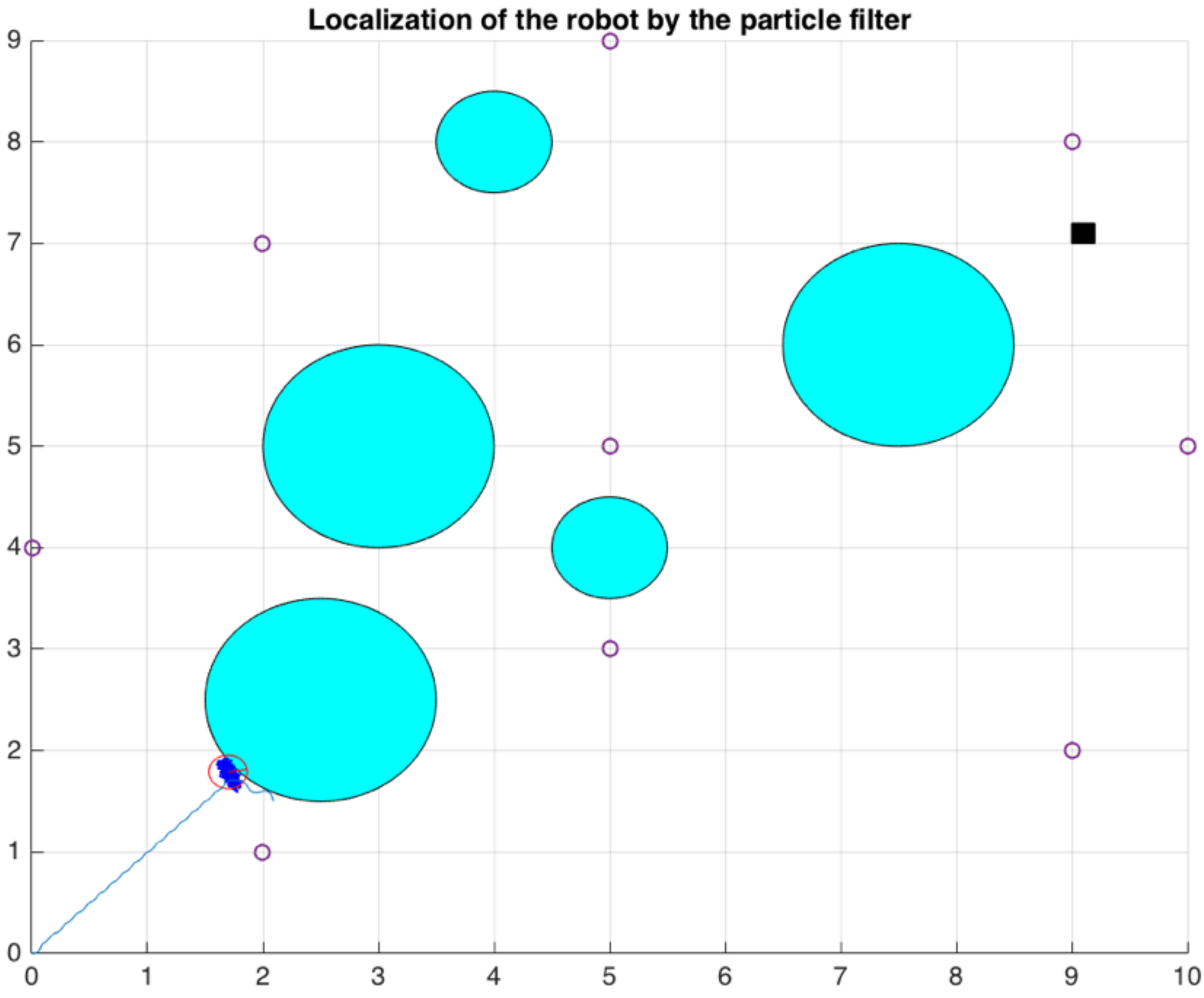

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
-1
-2
0
2
4
6
8
10

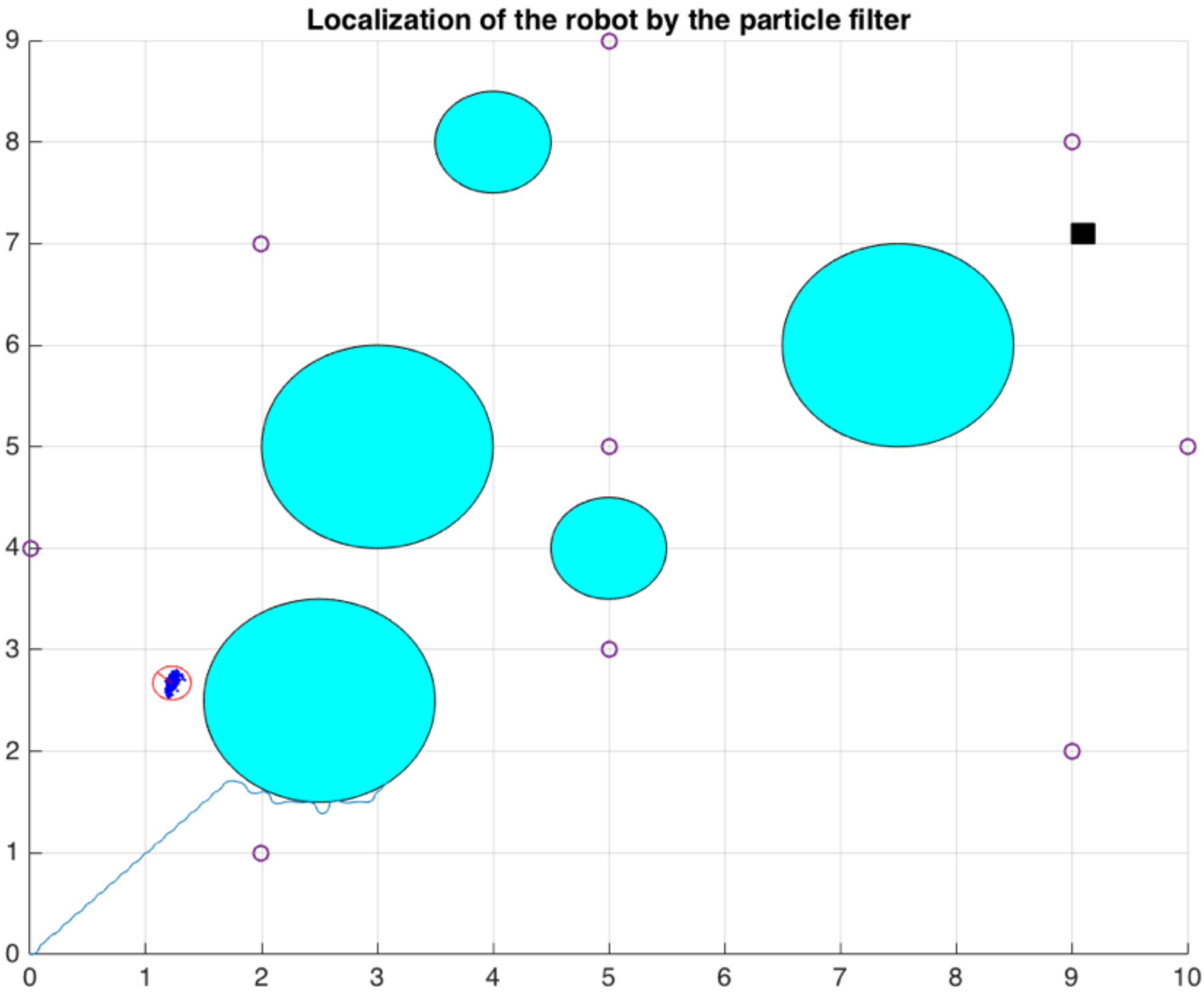

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
-1
-2
0
2
4
6
8
10

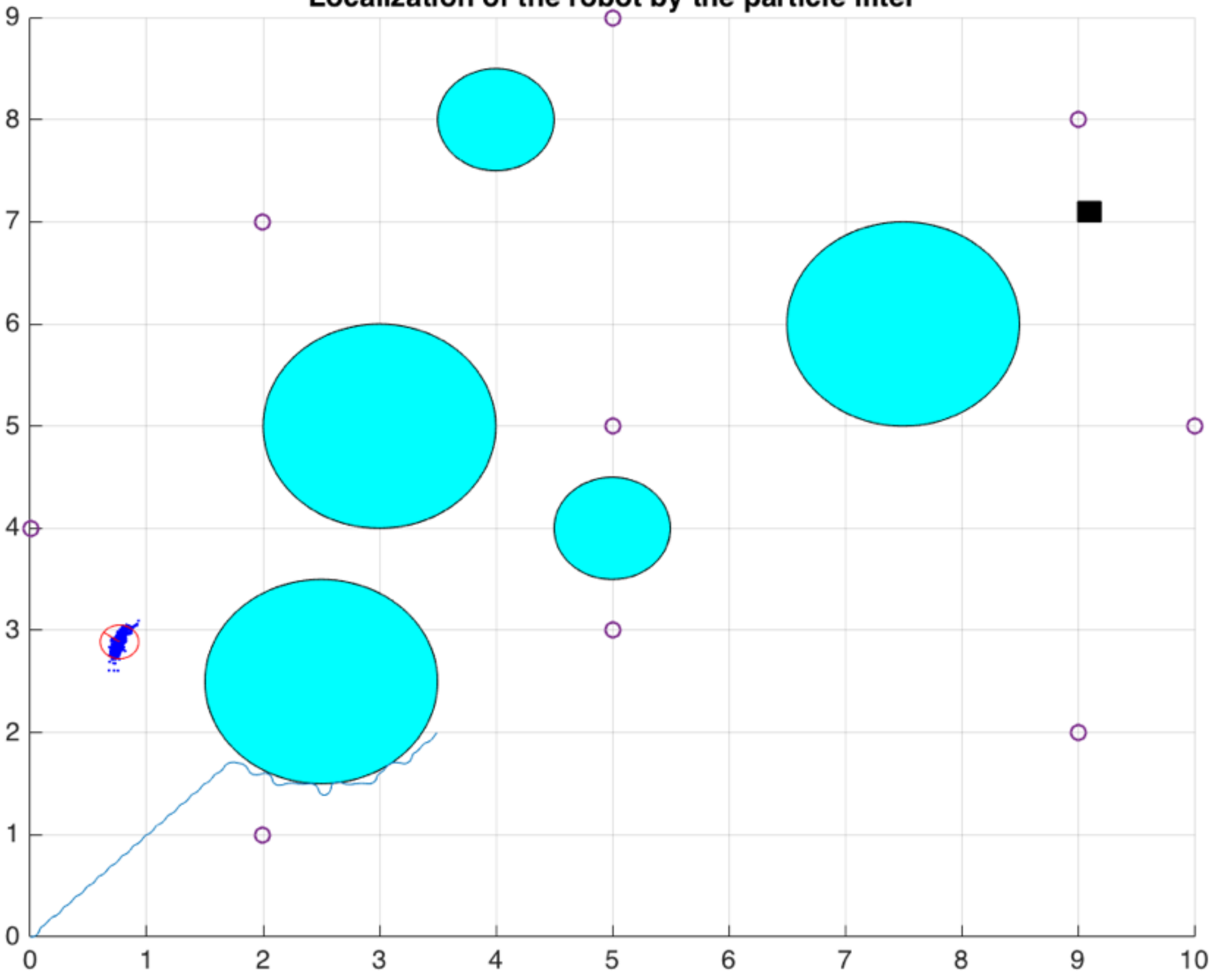

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
-1
0
1
2
3
4
5
6
7
8
9
10

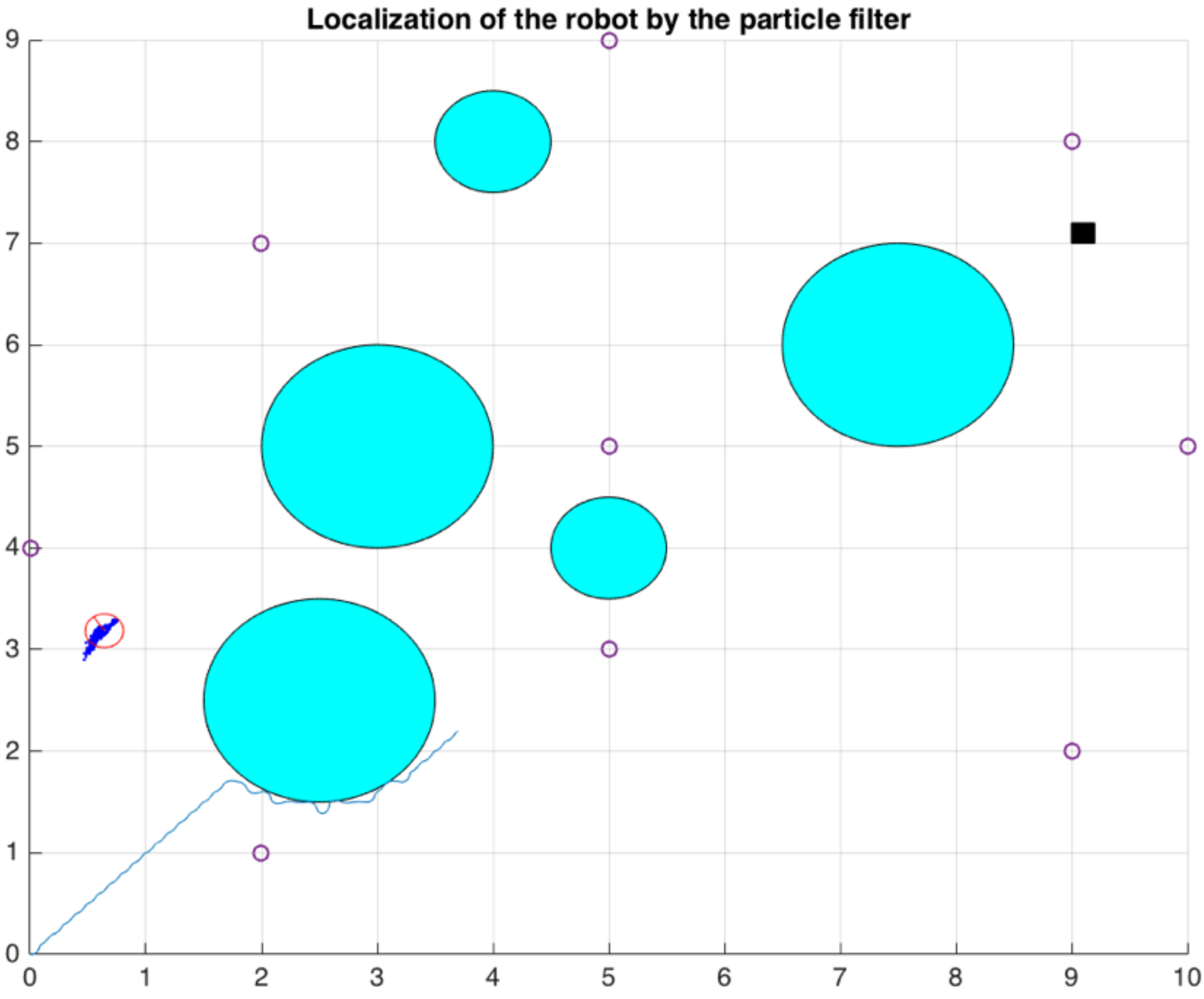

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10

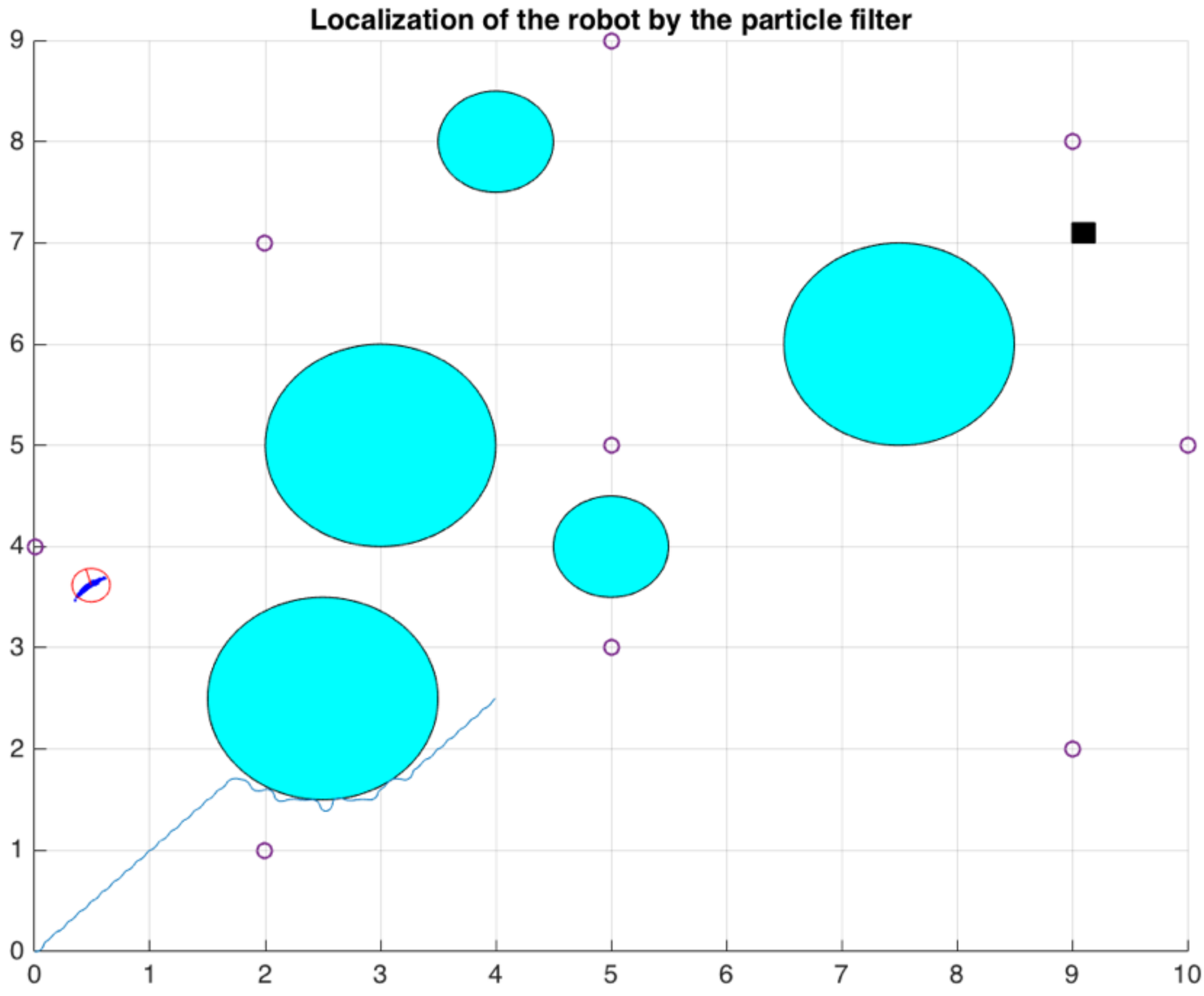

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10

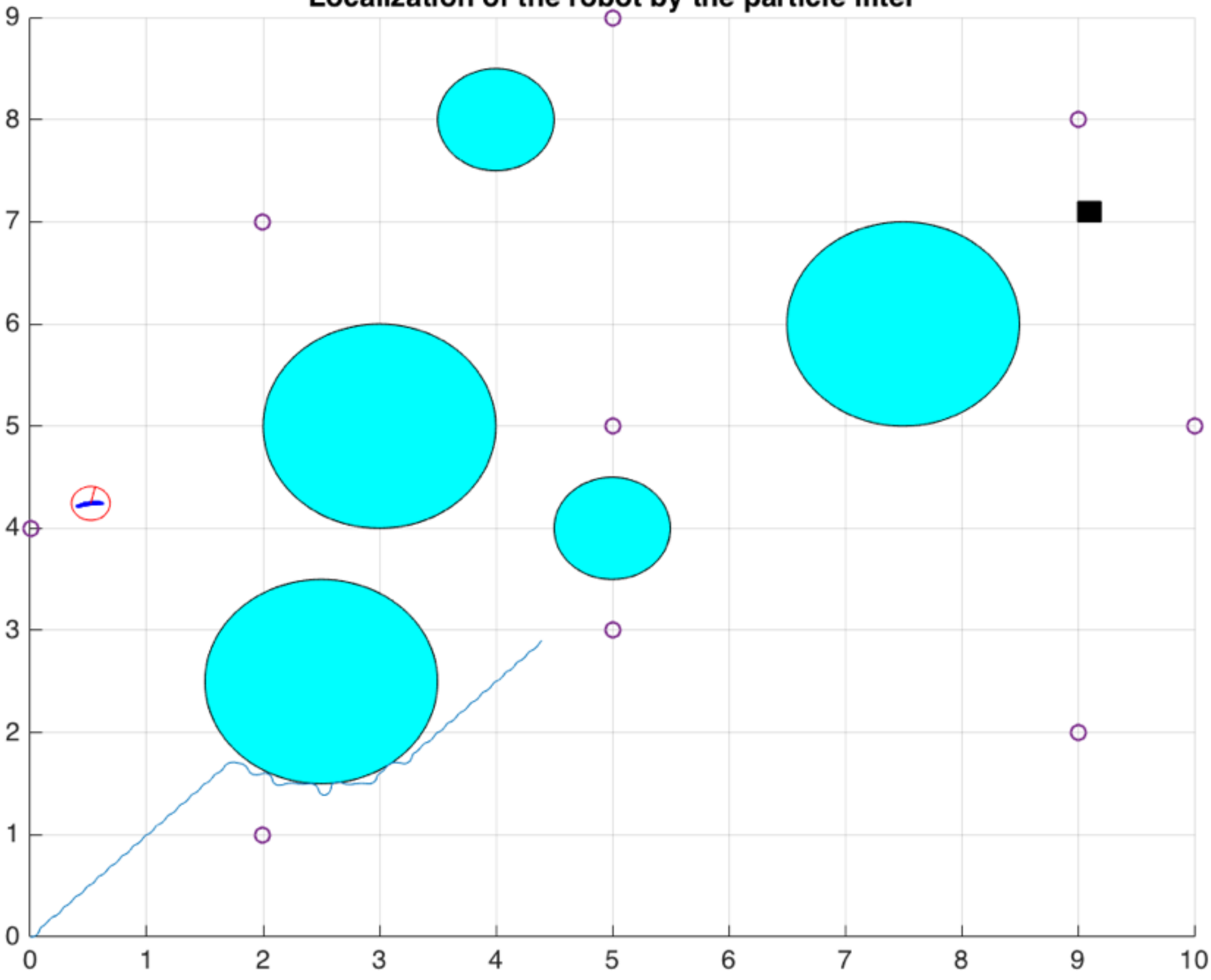

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10

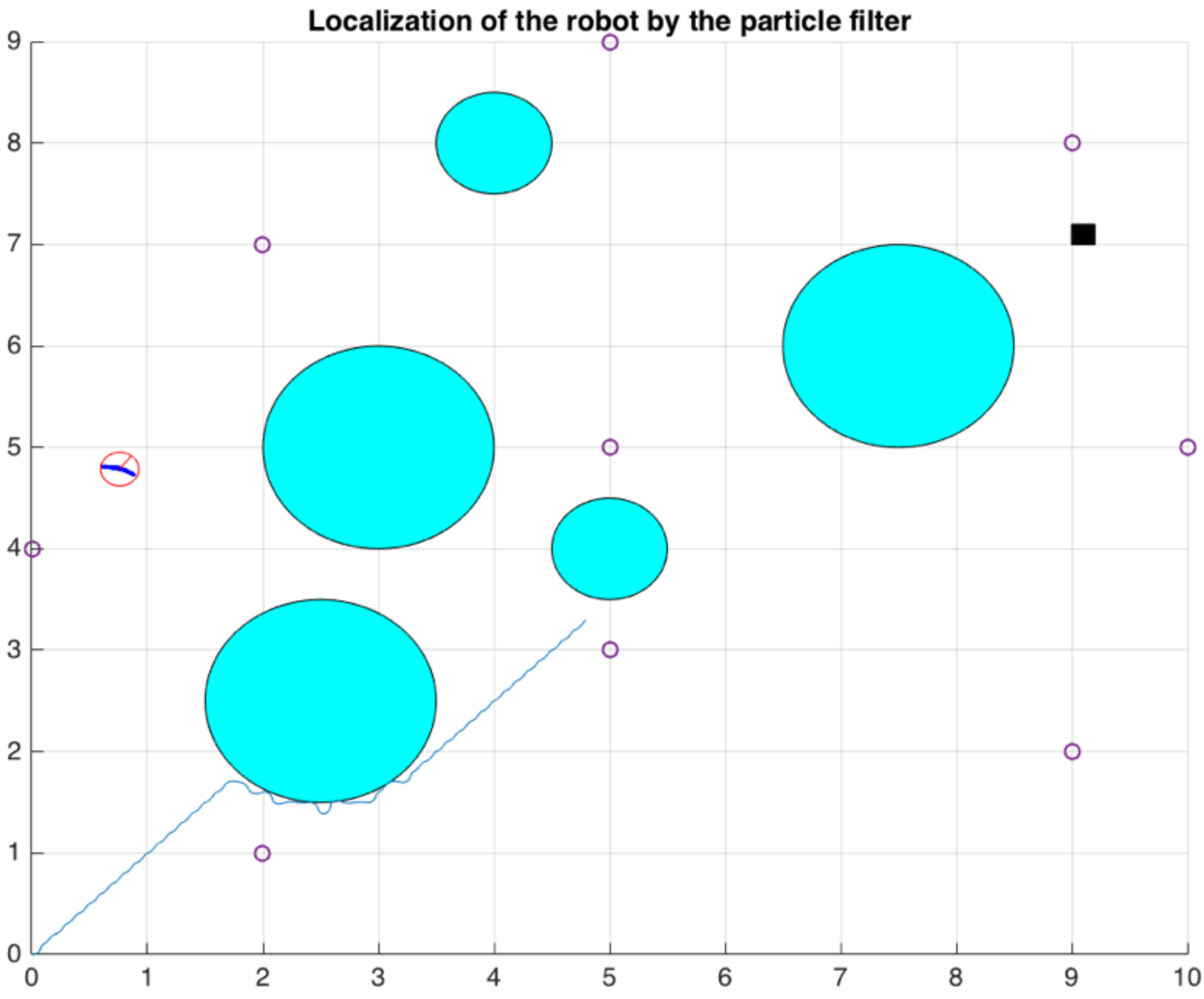

Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10


Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10


Localization of the robot by the particle filter
0
1
2
3
4
5
6
7
8
9
10


Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10


Localization of the robot by the particle filter
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10


Localization of the robot by the particle filter
0
1
2
3
4
5
6
7
8
9
10

Results

- The A* approach applied to the robot's motion planning provided great strategy to direct robot to goal efficiently

- The path found was used to move the robot to the goal position

- Tracking was integrated with the motion planning and Particle filter was used to track robot. Odometry for the filter was derived form robots motion

- The precomputed sensor data provided sensor readings that were too inconsistent with measurements and did not assign the weights correctly to the particles

- The believed position of the robot drifts away from the actual location of the robot

Conclusion

- The A* approach works very well to find the shortest path and guide the robot to traverse an efficient path to reach from the start to the goal position. Further work can be done to improve the system such that its faster and can also work in dynamic environment. D* is an algorithm that works with dynamic environment. Rapidly-Exploring Random Trees (RRT) aims at quickly exploring for the path

- Particle filters offer a simple and elegant solution the the problem of robot localization. The sensor data should be as consistent as possible for more precise localization. But there is always some noise in the motion and sensor that gives rise to uncertainty.