Section 4
Coverage Control

Shun-ichi Azuma

Nagoya University

www.ctrl.mae.nagoya-u.ac.jp

Outline

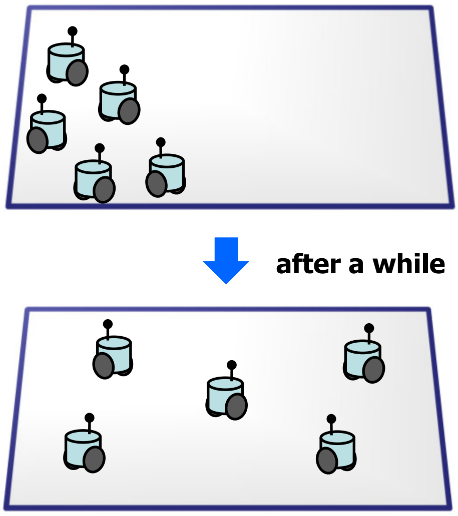
4.1 Coverage Problem

4.2 Coverage Controllers

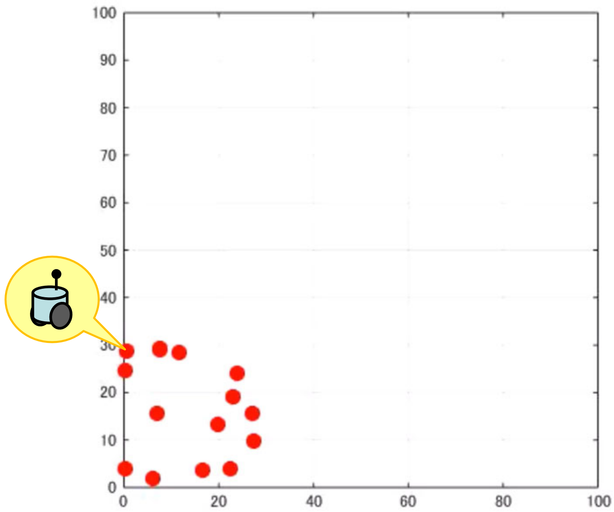
4.3 Python Implementation

What is Coverage?

- **What is consensus?**
  - **Consider a multi-agent system with n agents.**
  - **Each agent can know the relative position of neighbor agents and move depending on it**
  - **Coverage (or deployment): All the agents are placed in an environment so that the agents are distributed uniformly in some sense.**



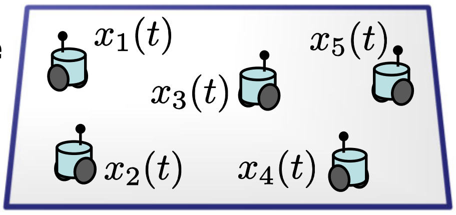
- **Coverage by 14 agents.**



- **Note that the final formation is unknown in designing the controller for coverage.**

Multi-agent Systems to Be Studied

- **Agents**
  - Consider n agents, which are called agent i (i=1,2,…,n)



  - Agent i is described by

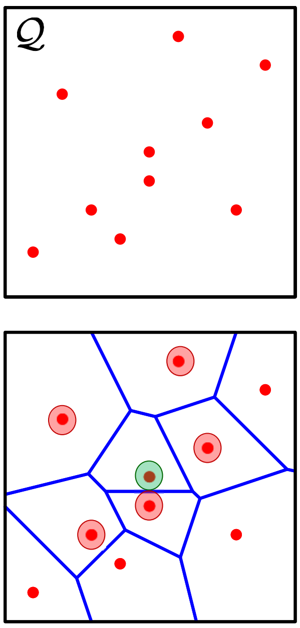
    $$\dot{x}_i(t) = f_i(x_i(t), u_i(t))$$

    where

    - $x_i(t) \in \mathrm{R}^m$ is the position, $u_i(t) \in \mathrm{R}^l$ is the input,
    - f_i is a function.

- **Neighbors**
  - Each agent has a proximity sensor, which measures the relative positions to some agents, called the neighbors.
  - The sensor has an ability to determine the Voronoi cell.

- **Voronoi Diagram**
  - Consider n distinct points in a set Q.
  - A Voronoi diagram is a partition of the set into n regions, such that
    (i) each region contains exactly one point,
    (ii) the point in the region is the nearest one to any locations in a region.
  - Each region containing a point is called the Voronoi cell of the point.

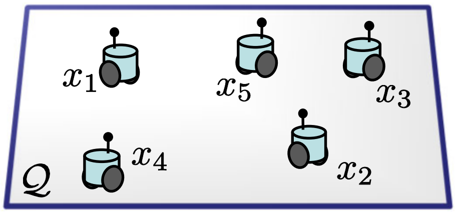


- **Information Agent i has**
  - (At least) the relative positions of the agents whose Voronoi cell is adjacent to the Voronoi cell of agent i.

Coverage Problem

- **Objective Function**
  - **Coverage of a set $\mathcal{Q}$:**

    $x_1, \ldots, x_n$ **are uniformly distributed in the set $\mathcal{Q}$.**



  - **Objective function:**

$$J(x) := \int_{\mathcal{Q}} \boxed{\min_{i \in \{1,2,\ldots,n\}} \|q - x_i\|^2} dq$$

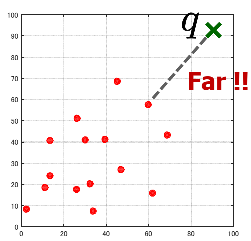
The square of the distance between q and the nearest agent.

= The badness of q in the sense that q is far from any agents

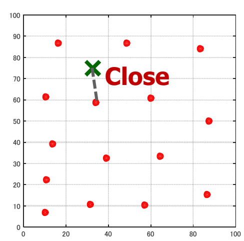
- $q \in \mathbb{R}^m$ **: A location in $\mathcal{Q}$**
- $x = [x_1^\top \ x_2^\top \ \cdots \ x_n^\top]^\top$

⇨ $J(x)$ **is the sum of the badness of all location q for x**

- **Example**
  - **Which formation has a larger value of J?**



$>$



Some q is far from any red point. **Any q is near to some red point.**

- **Coverage Problem**
  - **For the multi-agent system, find $u_i \ (i = 1, 2, \ldots, n)$ such that $\lim_{t \to \infty} J(x(t)) = \min_{x \in \mathcal{Q}^n} J(x)$.**

Outline

4.1 Coverage Problem

4.2 Coverage Controllers

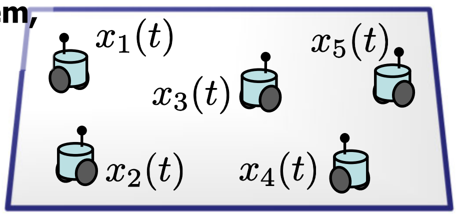
4.3 Python Implementation

Coverage Controllers

- **Multi-agent System to Be Considered**
  - **Consider a multi-agent system, where agent i is given by**

    $\dot{x}_i(t) = u_i(t)$

    for $x_i(t), u_i(t) \in \mathbb{R}^2$.



- **Gradient-based Controllers**
  - **Gradient-based controller:** $u_i(t) = -\dfrac{\partial J}{\partial x_i}(x(t))$
  - **Controlled Dynamics:** $\dot{x}_i(t) = -\dfrac{\partial J}{\partial x_i}(x(t))$

- **Collective Dynamics**

$$\left.\begin{aligned} \dot{x}_1(t) &= -\frac{\partial J}{\partial x_1}(x(t)) \\ &\;\vdots \\ \dot{x}_n(t) &= -\frac{\partial J}{\partial x_n}(x(t)) \end{aligned}\right\} \quad \dot{x}(t) = -\frac{\partial J}{\partial x}(x(t))$$

⇨ A gradient system for the objective function

- The solution converges to a stationary point of J under reasonable conditions.
- The stationary point often corresponds to a local or global minimum, which implies coverage.
- Questions: (1) Explicit form of the controller?
  (2) Computable by agent i's information?

Explicit Form of Gradient Controller

- **Explicit Form of Gradient**

Lemma

$$\frac{\partial J}{\partial x_i}(x) = k\Big(x_i - \mathrm{cent}(\mathcal{C}_i(x))\Big)$$

- $\mathcal{C}_i(x)$: Voronoi cell of the position of Agent i
- $\mathrm{cent}(\mathcal{C}_i(x))$: Centroid of $\mathcal{C}_i(x)$
- k : A positive scalar (depending on $\mathcal{C}_i(x)$)

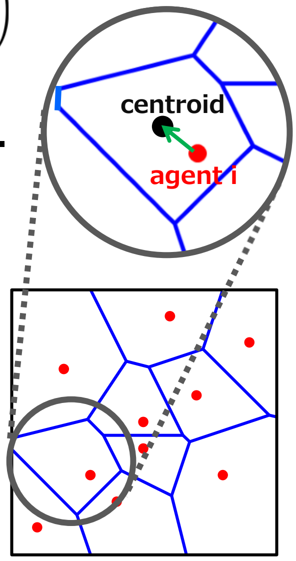
- **Gradient-based Controllers**

$$u_i(t) = -\frac{\partial J}{\partial x_i}(x(t)) = -k\Big(x_i(t) - \mathrm{cent}(\mathcal{C}_i(x(t)))\Big)$$

Remarks

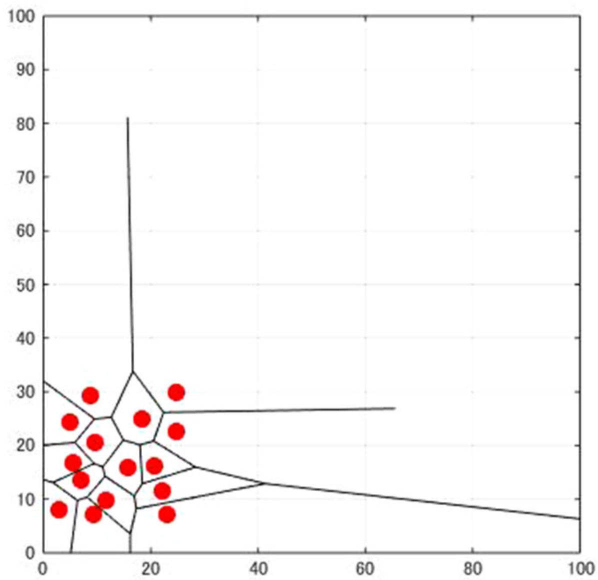
$$u_i(t) = -k\Big(x_i(t) - \mathrm{cent}(\mathcal{C}_i(x(t)))\Big)$$

- This controller moves the agent i to the direction of the centroid of $\mathcal{C}_i(x)$.
- The input can be computed by the information of agent i, because $x_i(t) - \mathrm{cent}(\mathcal{C}_i(x(t)))$ can be computed by its information.
- Strictly speaking, k is a time-varying scalar; however we can replace it with a constant value from the viewpoint of reducing J.

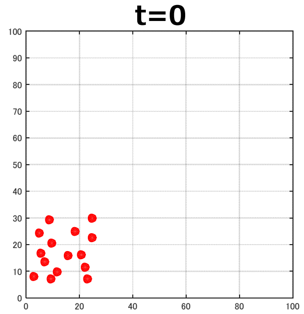


Example

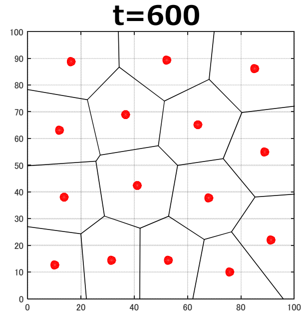
- $\mathcal{Q} = [0, 100]^2$
- 14 agents, denoted by "●"
- Solid lines: Boundary of Voronoi cells

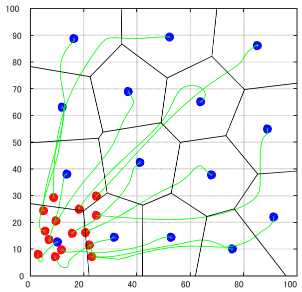


t=0



t=600





Trajectories

Outline

4.1 Coverage Problem

4.2 Coverage Controllers

4.3 Python Implementation

Python Code

- **Sample code: 4_coverage.py**

https://drive.google.com/drive/folders/1Y0Fu2b3tDSOXsP_YV5NpaGGPeujlrkf1?usp=sharing

Python Code

```
from scipy.integrate  import odeint
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import Voronoi, voronoi_plot_2d
from shapely.geometry import Polygon, Point
```

Part 1: Definition of a multi-agent system as an ODE

```
def MAS(x,t):

    x = np.array(x).reshape(-1, 2)
    dxdt = []

    # Definition of agent i
    for i in range(len(x)):

        # Control input of agent i
        cent, vor = voronoicentroid(np.array(x).reshape(-1, 2),workspace)
        u_i = -1 *(x[i] - cent[i])

        # Dynamics of agent i
        dxdt.append(u_i.tolist())

    return sum(dxdt,[])
```

Part 2: Computation of the Centroid of a Voronoi Cell

```
def voronoicentroid(x,workspace):

    wb = workspace.bounds
    wl = workspace.length
    D = np.array([[wb[0] - wl * 10, wb[1] - wl * 10],
                  [wb[2] + wl * 10, wb[1] - wl * 10],
                  [wb[0] - wl * 10, wb[3] + wl * 10],
                  [wb[2] + wl * 10, wb[3] + wl * 10]])

    points = np.append(x, D, axis=0)

    vor = Voronoi(points)
    vcentroid = x

    for i in range(len(x)):
        poly = [vor.vertices[v] for v in vor.regions[vor.point_region[i]]]
        i_cell = workspace.intersection(Polygon(poly))
        vcentroid[i] = i_cell.centroid.coords[0]

    return  vcentroid, vor
```

Part 3: Solve the ODE

```
workspace = Polygon([[0, 0], [1, 0], [1, 1], [0, 1]])
x0 = np.array([[0.1, 0.1], [0.2, 0.1], [0.25, 0.3], [0.35, 0.2], [0.3, 0.3],
        [0.3, 0.5], [0.4, 0.15], [0.4, 0.3], [0.4, 0.4], [0.5, 0.4]])
t = np.arange(0, 50, 0.01)
x = odeint(MAS, np.array(sum(x0.tolist(),[])), t)
```

Part 4: Plot the result

```
for i in range(len(x)):
    if i%100 == 0:
        cent, vor = voronoicentroid(np.array(x[i]).reshape(-1, 2),workspace)
        voronoi_plot_2d(vor)
        plt.gca().set_aspect('equal')
        plt.gca().set_xlim([0, 1])
        plt.gca().set_ylim([0, 1])
        plt.show()
```

$$u_i(t) = -k\Big(x_i(t) - \mathrm{cent}(\mathcal{C}_i(x(t)))\Big)$$