Coverage Control Overview

Simon Hu

1 Coverage Control Overview

1.1 Background

Let $\mathbf{p} = (p_1, p_2, \dots, p_N) \in \mathbb{R}^N$ be a vector whose elements are the positions of the i-th agent, given by $p_i \in \mathbb{R}^d$ where $d = 1, 2, \dots$. The goal of the coverage control algorithm is to solve the following problem.

$$\max_{\mathbf{p}\in\Xi^N} \mathcal{H}_\varphi(\mathbf{p}, t) := \max_{\mathbf{p}\in\Xi^N} \int_\Xi \min_{i=1,2,\dots,N} \|\boldsymbol{\xi} - \mathbf{p}_i\|_2^2 \, \varphi(\boldsymbol{\xi}, t) \ \mathrm{d}\boldsymbol{\xi} \tag{1.1}$$

In other words, the goal is to find the optimal configuration of agent positions $\mathbf{p}$ so that the desired area to cover, which is encoded in $\varphi : \mathbb{R}^N \times \mathbb{R}^+ \to \mathbb{R}$, is covered by all agents, which ensuring that agents are assigned an area, which is encoded in $\|\boldsymbol{\xi} - \mathbf{p}\|_2^2$, that is maximal with respect to Ξ. We will discuss how this φ fits in with the tracking framework, but for now note that it is just a regular old function.

The Voronoi partition $\mathcal{V} \equiv \mathcal{V}(\Xi, \mathbf{p})$ of Ξ given the current agent positions $\mathbf{p}$ is a set $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_N\}$. Here, each cell $\mathcal{V}_i$ is given by

$$\mathcal{V}_i = \left\{ \xi \mid \|\xi - p_i\|_2^2 \le \left\|\xi - p_j\right\|_2^2 \ \forall\, i \neq j, \ \forall\, j = 1, 2, \dots, n \right\}.$$

An example of a Voronoi partition is shown in Figure 1. Using the definition of a Voronoi partition, we can rewrite (1.1) as

$$\max_{\mathbf{p}\in\Xi^N} \mathcal{H}_\varphi(\mathbf{p}, t) := \max_{p\in\Xi^N} \sum_{i=1}^{N} \int_{\mathcal{V}_i} \|\boldsymbol{\xi} - \mathbf{p}_i\|_2^2 \, \varphi(\boldsymbol{\xi}, t) \ \mathrm{d}\boldsymbol{\xi}. \tag{1.2}$$

In other words, the problem reduces to finding the configuration of agents so that effective coverage is maintained, but each agent is also assigned an area (i.e. a cell) that it is responsible for. There are two advantages of using the Voronoi partitions. First, the min term inside the integral is removed by the construction of the Voronoi cells. Second, any algorithm that uses these partitions will be *distributed*, which means agents only need to use information from its Voronoi neighbors, defined by agents that share a cell boundary (i.e. agents i and j are neighbors if and only if $\partial\mathcal{V}_i \cap \partial\mathcal{V}_j \neq \emptyset$), so that communications between agents can be reduced. However, for the purposes of this project, the algorithm that has been implemented should run on a *centralized* computer and then information about the waypoints, should be shared with the agents through the centralized computer.

The mass M_i and centroid C_i of the i-th cell is given by

$$M_i = \int_{\mathcal{V}_i} \varphi(\boldsymbol{\xi}, t) \ \mathrm{d}\boldsymbol{\xi}, \quad C_i = \frac{1}{M_i} \int_{\mathcal{V}_i} \boldsymbol{\xi} \cdot \varphi(\boldsymbol{\xi}, t) \ \mathrm{d}\boldsymbol{\xi}. \tag{1.3}$$

2 Approach

2.1 Dimensionality

For the purposes of this project, we only consider $d = 3$, i.e. 3-D Euclidean space, though the algorithm extends to an arbitrary number of dimensions. Orientation information may be available so that $d \neq 3$, we only use the position information to compute the waypoints that the agents should travel to. After all, our algorithm spits out only the waypoints that the agents should travel to. It is to our understanding that there exist a controller that will bring the robot to the waypoint.

2.2 Domain Model

Furthermore, it is assumed that the domain Ξ is convex and *extended* to a rectangular polygon. Since this extension is done, it is up to the user to determine whether or not the waypoint is inside the feasible set of the *true* operating domain. For example, in the case where $d = 2$, and the operating domain is given by the vertices $(0, 0), (1, 0), (1, 1)$ then Ξ is set to be the rectangle given by the vertices $(0, 0), (0, 1), (1, 1), (1, 0)$, and is used for all computations. However, the waypoint $(0, 1)$ may not be an area that is accessible to the agents. Thankfully, for the purposes of this project we have been allowed to assume that the operating domain can be modeled as free water (i.e. a very large rectangular polygon) and thus our assumption is allowed. Regardless, this scenario would not happen, unless a RED agent goes outside the operating domain.

2.3 RED Agent Model

A RED agent is modeled as a Gaussian distribution, $W \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ so that the PDF of W is given by

$$\varphi(w) = \frac{1}{\sqrt{(2\pi)^3 |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(w - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (w - \boldsymbol{\mu}) \right). \tag{2.1}$$

This $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are *provided* to you by an algorithm called Event-Triggered Distributed Data Fusion (ET-DDF) which is developed by Luke Barbier from Colorado University in Boulder. For scenarios with multiple RED agents, the following model can be used:

$$\varphi(w) = \sum_{k=1}^{K} \frac{\pi_k}{\sqrt{(2\pi)^3 |\boldsymbol{\Sigma}_k|}} \exp\left(-\frac{1}{2}(w - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (w - \boldsymbol{\mu}_k) \right) \tag{2.2}$$

where K is the number of RED agents present. Note that this is just a Gaussian mixture model, though there is no constraint on π_k as φ does not need to be a proper probability distribution. This φ is the same exact φ that is used in equation (1.3) *but* we have restricted φ to be in the class of probability densities. This makes sense, as φ encodes the object we want to track, and we have modeled φ as a Gaussian distribution to play well with ET-DDF. Note that due to numerical integration issues, we enforce that the variance $\boldsymbol{\Sigma}$ have a lower limit (i.e. a minimum lower threshold of covariance) since the computation of M_i in equation (1.3) requires an integral over φ, so if φ is too small due to $\mathcal{V}_i$ containing a very little contribution from φ (i.e. when agent i is very very far away from the RED agent) then we run into divide-by-zero issues and the centroid computation blows up. There is no secret to choosing this minimum threshold but it should depend on the size of Ξ. Alternatively, you can first find a scaling function $g : \Xi \to \Xi'$ which effectively scales down Ξ to a smaller domain. Scaling is always injective, so the final waypoints can be returned for the non-scaled operating domain. This helps with reducing finite-time blow-up issues mentioned above.

2.4 Dynamic Coverage Control

The original algorithm presented in [1] does not cover the case where the φ is dynamically changing. Since this is the case for the moving RED agent, the original algorithm was adapted according to [2]. The authors in [2] use derivative information about φ to obtain better waypoints. Unfortunately, this information is not returned to us by ET-DDF, and thus we must turn to finite-differencing schemes to estimate them. The first order finite-differences scheme used is given by

$$\dot{\varphi} = \frac{p_i - p_{i-1}}{t_i - t_{i-1}}. \tag{2.3}$$

Using this information, the updated mass, center of mass, and their derivatives are given by

$$\dot{M}_i = \int_{\mathcal{V}_i} \dot{\varphi}(\boldsymbol{\xi}, t) \ \mathrm{d}\boldsymbol{\xi}, \quad \dot{C}_i = \frac{1}{M_i} \left(\int_{\mathcal{V}_i} \boldsymbol{\xi} \cdot \varphi(\boldsymbol{\xi}, t) \ \mathrm{d}\boldsymbol{\xi} - M_i C_i \right) \tag{2.4}$$

and the M_i and C_i are the same as given before, in equation (1.3). Note that this is still the Cortés' algorithm but modified to include derivative information. The update law, using the derivative information is given by

$$\dot{p}_i = \dot{C}_i - \left(\kappa + \frac{\dot{M}_i}{M_i} \right) (p_i - C_i). \tag{2.5}$$

In the above, $\kappa > 0$ is some small gain. There is no easy way to choose this κ, but in the implementation it is chosen so that the overall control has unit 2-norm. To compute the new waypoints, we can use this update law and perform a gradient descent algorithm, i.e. $p_{i+1} = p_i + \alpha \dot{p}_i$ where α is a parameter that can be tuned.

2.5 Solving the Optimization Problem

To solve the maximization problem, agents move towards the centroid of their Voronoi cell [1]. Intuitively, the agents move towards the area they need to cover since the φ acts as an attracting force that draws the center of mass towards the desired coverage area. Algorithm 1 describes one step of the coverage control algorithm. This single-step algorithm can then be run over and over again until tracking is completed and the agents can go back to observing the environment.

To reduce the complexity of the code required to solve the problem, we consider a simple 3-D projection algorithm. Essentially, instead of using the C++ package **qhull** to compute the FULL 3-D Voronoi partition, we simply project the BLUE agent positions to the plane defined by the height of the RED agent, and save the projection information (to be used later in determining the waypoints). This creates a 2-D domain over which the 2-D geometric computations can be carried out. However, since we are working in 2-D the coverage control output will update the waypoints of the BLUE agents for the first two dimensions but not the third. But don't fear, the third dimension is recovered using the projection heights, which brings the BLUE agents closer to the RED agents.

Algorithm 1 Coverage Control Single Step

```
Input: Agent positions $p \in \mathbb{R}^3$, coverage function $\varphi$, operating domain $\Xi$, RED agent $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$
Output: New agent waypoints $p^* \in \mathbb{R}^3$.
Set $p^* = p$.
Project all points to the plane defined by the height of the RED agent.
Compute projection distances.
Compute the 2-D Voronoi partition $\mathcal{V}(\Xi, p) = (\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_N)$ of the projection.
for each cell $\mathcal{V}_i$, $i = 1$ to $N$ do
    Compute the mass and center of mass, and their derivatives according to equation (2.4).
    Using the control law (2.5) and the projection distances, compute the new waypoints using gradient descent.
    Set $p_i^*$ to the result of the above.
end for
Return $p^*$, the new agent waypoints.
```

3 Python Implementation Details

3.1 Structure of Code

Most of the code that you will use for this project is contained in a helper file called **pyvoro3d.py** which is contained under **vorutils**, which is short for Voronoi utilities. The entire package contains utility functions that you can use to create simulations of the coverage control algorithms. Examples of simulations are contained in the **simutils**, short for simulation utilities. In those examples, you can see how to create a simulation using the helper files. What is left to be done, is to turn this entire algorithm into a ROS node that can be used by the robots.

There is another package in there containing the helper file **pyvoro.py**. This is the old 2-D tracking code and is mostly reserved for use by Dimitris Boskos, post-doc with Sonia and Jorge. You are free to look and use it, since the essence of the code is also ported into **pyvoro3d.py** but be careful with the usage of those functions.

3.2 Geometric Operations

The Python modules/sub-modules **shapely** and **scipy.spatial** are used to handle geometric operations. Since much of the computational overhead was reduced in moving from full 3-D to a simplified version **shapely**, which is more suitable for 2-D geometric computations, is used in favor of **scipy.spatial**. However **scipy.spatial** contains the code that computes the infinite Voronoi partitions, so we still use it.

3.3 Computing the Integral

Numerical integration is not easy, especially when we are dealing with non-uniform domains. Currently, the integration strategy is to use **scipy.integrate** which uses quadrature rules to compute the integrals. Unfortunately, this means that the limits of the integration, according to their documentation, must be rectangular. Thus, any convex Voronoi cell that is spit into their integration scheme automatically gets turned into the smallest bounding box of the Voronoi cell. So your integration scheme will be incredibly inaccurate if you are unlucky, but this inaccuracy does help prevent the integrals from blowing up.

The scheme proposed initially was to create a Delaunay triangulation (or just regular triangulation) of the domain and compute the integrals over the triangles, which can be done using many schemes that abuse Barycentric coordinates. Then, their values are summed to obtain the integral over the entire Voronoi cell. However, when this was implemented, many of these integral computations suffered from finite-time blow up in the integral, since the agents were so far away from the RED agent, the contribution from φ was essentially zero. This is definitely an area for work to be done, and you should look into **quadpy**, which handles these quadrature computations and **scipy.spatial** for the triangulation.

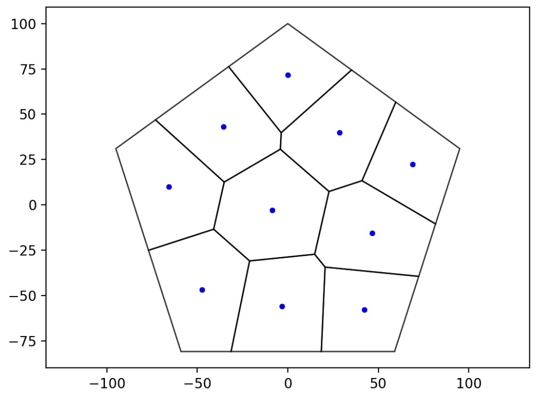

Figure 1: An example of a Voronoi partition. The Agents are represented as dots.

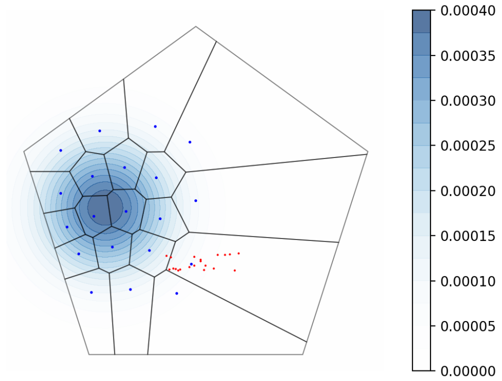

Figure 2: A brief snapshot of the coverage control algorithm. The Agents (dots) cover the region defined by the dark blue area. The red dots represent their starting location.

References

[1] J. Cortés, S. Martínez, T. Karatas, and F. Bullo, “Coverage control for mobile sensing networks,” *IEEE Transactions on Robotics and Automation*, vol. 20, pp. 243–255, April 2004.

[2] Y. Diaz-Mercado, S. G. Lee, and M. Egerstedt, *Human–Swarm Interactions via Coverage of Time-Varying Densities*, pp. 357–385. Cham: Springer International Publishing, 2017.