**2013 IEEE International Conference on Robotics and Automation (ICRA)**
**Karlsruhe, Germany, May 6-10, 2013**

# K-Robots Clustering of Moving Sensors using Coresets

Dan Feldman*, Stephanie Gil*, Ross A. Knepper*, Brian Julian*†, and Daniela Rus*

***Abstract*— We present an approach to position $k$ servers (e.g. mobile robots) to provide a service to $n$ independently moving clients; for example, in mobile ad-hoc networking applications where inter-agent distances need to be minimized, connectivity constraints exist between servers, and no *a priori* knowledge of the clients' motion can be assumed. Our primary contribution is an algorithm to compute and maintain a small representative set, called a kinematic coreset, of the $n$ moving clients. We prove that, in any given moment, the maximum distance between the clients and any set of $k$ servers is approximated by the coreset up to a factor of $(1 \pm \varepsilon)$, where $\varepsilon > 0$ is an arbitrarily small constant. We prove that both the size of our coreset and its update time is polynomial in $k \log(n)/\varepsilon$. Although our optimization problem is NP-hard (i.e., takes time exponential in the number of servers to solve), solving it on the small coreset instead of the original clients results in a tractable controller. The approach is validated in a small scale hardware experiment using robot servers and human clients, and in a large scale numerical simulation using thousands of clients.**

## I. Introduction

A cooperative team of robots can provide a large range of services to moving clients, where these clients can be other robots (or even humans) that are performing independent tasks. Such services include environmental surveillance, system health monitoring, and communication coverage. Many interesting applications enabled by these services require that the team of robots be adaptive to their environment; the overlying control strategy must account for the motion of the clients that the robots are tasked to service. In addition, the quality of these services (e.g., communication strength, camera resolution) often attenuates with increasing separation between robots. Thus, we favor proximity-based solutions that consider the dynamical constraints of real platforms.

In this paper we consider the problem of controlling this team of $k$ robots (servers) to provide services to $n$ clients moving independently in a $d$-dimensional space, where $d \geq 1$ is constant. Our model assumes that: 1) the clients are free to move over arbitrary *a priori* unknown paths, 2) the maximum distance between any client and its nearest server should be small, and 3) additional constraints and control limitations can be applied over the feasible new locations of the servers, e.g., by restricting the maximum distance between servers. As a motivating example, consider the problem of controlling a team of aerial robots that provide ad-hoc network communincation to ground based clients. A

*All authors are with the Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139, USA, {dannyf, sgil, rak, bjulian, rus}@csail.mit.edu

† Brian Julian also with MIT Lincoln Laboratory, 244 Wood Street, Lexington, MA 02420, USA

This work is supported by MAST Project under ARL Grant W911NF-08-2-0004. This work is also sponsored by the Department of the Air Force under Air Force contract number FA8721-05-C-0002. The opinions, interpretations, recommendations, and conclusions are those of the authors and are not necessarily endorsed by the United States Government.

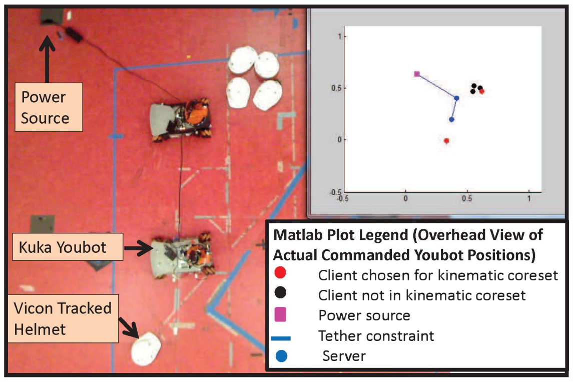
Power Source
Kuka Youbot
Vicon Tracked Helmet
Matlab Plot Legend (Overhead View of Actual Commanded Youbot Positions)
Client chosen for kinematic coreset
Client not in kinematic coreset
Power source
Tether constraint
Server

Fig. 1: Overhead view of our hardware setup of two Kuka Youbot robots and white helmets with vicon tracking markers that were worn by five adults, or "clients", moving around the motion capture room at a walking pace as shown in figure 4. This figure shows the constraint that the first Youbot was tethered to a power source at the top left corner of the environment and the second Youbot was tethered to the first Youbot for power. The Matlab plot in the top right corner demonstrates the two kinematic coreset points (red) and the optimal configuration of the Youbots (blue) computed for the client positions under the given tethering constraints.

client $p$ needing to route a message to another far away client $q$ can route its message through a path of servers $c_1, \cdots, c_i$, where $c_1$ is the closest server to $p$, and $c_i$ is the closest server to $q$. This approach requires that every client maintain a connection only to its closest server, rather than all the other clients, and that the servers remain connected. By dynamically repositioning the servers as the clients move, we can increase the communication coverage and reduce the number of required servers.

It is common knowledge that even the static and unconstrained versions of these problems (known as $k$-center) are NP-hard, i.e, take time exponential in $k$ to compute a solution. Intuitively, it should not be necessary to track every client individually to compute near optimal server placement. Hence, we concentrate on constructing a subset that accurately represents client locations such that the error compared with the optimal placement of servers can be bounded. A naïve uniform sampling of clients will often miss outliers that dominate the optimization's cost function, leading to high approximation costs. Thus, we need a better solution.

The main tool we propose to handle these problems as well as the dynamic and constrained versions is a *kinematic coreset*. This tool yields a sparse representative set of the $n$ clients that provably approximates their maximum distance to any possible positioning of the $k$ servers at any given time. Since our coresets are small and can be updated quickly, we are able to apply exact (optimal) solutions that would otherwise be intractable. This yields dynamic positioning of the servers that provably approximates the optimal solutions on the full set of clients. Since the running times are exponential in $k$, our coresets improve the performance even for a small number $n < 10$ of clients.

## A. Previous Work

Information and resource sharing amongst robot teams are common requirements for increased efficiency and thus inter-agent proximity is often favored. One common application is that of communication over mobile wireless networks where agent positions must be optimized to form a connected network. Both distributed and centralized solutions in either open or constrained environments have been examined in papers such as [3], [4], [8], [9], [15], [16].

The problems in this paper are variants and generalizations of the classic $k$-center problem [14], where we wish to minimize the maximum distance from a set of $n$ clients to a set of $k$ servers. Gonzalez [10] suggested a 2-approximation for the $k$-center problem that takes time $O(nk)$. An exact solution to this problem generally requires time exponential in $k$ [10] and has given rise to many static approximation methods including static *coresets*, or sparse representative sets [1], [6], [7], [13]. These coresets can be constructed in time that is linear in both $n$ in $k$, and returns a small set of size roughly $k/\varepsilon^d$, i.e., independent of $n$. We then run exhaustive search algorithms, approximations, or heuristics on these small representative sets.

In our previous work [7] we constructed (static) coresets for a class of problems framed as a *connected k-center* problem, where we wish to compute the positions of $k$ servers that minimize both the maximum distance from any client to its closest server, and the longest edge in the minimum Euclidean spanning tree over the servers (ie. connected centers). Unfortunately, this requires computation time that grows exponentially in the number $k$ of servers. This motivates the approach of reducing the number of clients $n$, in order to get a faster approximated solution (although still exponential in $k$).

The focus of this paper is to improve the static coreset approximation to explicitly account for the dynamic nature of the client vehicles under constraints. For the original $k$-center without any constraint, such coresets were suggested in [2], [5], [12]. In particular, the work by Timothy Chan in [2] has a similar focus as it derives a dynamic coreset for the $k$-center problem that can be updated in time $\log^{O(1)} n$ for constant $k$ and $\varepsilon$.

The current work differs from [2] in that our coreset can be used for approximating the distances of the clients to *any* $k$ servers, rather than only to the $k$-center of the clients or their coresets; see Eq. 4 for our formal definition. This makes our coreset useful for solving the $k$-center problem with additional constraints, or when maximum client-server distance is only part of the optimization function.

In particular, we use the coreset to solve the class of *connected $k$-center* problems from [7], where centers must maintain connectivity over a Euclidean minimum spanning tree (see Eq. (2)), and are subject to maximum velocity constraints. This class of problems is important for practical scenarios such as ad-hoc network formation where vehicles have control input limitations such as maximum velocity limits.

In this work we derive a *kinematic coreset* with the properties that it 1) can be updated quickly and adapt to client motion, 2) provides consistency such that the same coreset can be maintained for marginal client motion and 3) can provide approximation error bounds for the $k$-center and connected $k$-center problems as well as constrained versions of these problems. Our system contains the first implementation of kinematic coresets, and include several improvements to the state of the art, both in term of theoretical guarantee and practical usage.

## B. Our Contributions

*1) **Kinematic Coreset**:* Our main new technical tool is an algorithm to compute a kinematic coreset, or sparse subset, of clients to be used as input for computing server positions to form a connected communication network for the set of moving client vehicles. This coreset differs from the *static* coreset of our previous work in [7] in that the kinematic coreset can be updated quickly, is reactive to client movement, and also provides consistency across iterations such that for marginal client movement the same coreset can be maintained.

We show how to maintain a coreset $S$ for the set $P$ of $n$ clients such that in every given moment it holds that 1) for any position of the $k$ servers, the maximum distance between a client and its closest server is the same for $P$ and $S$, up to a multiplicative factor of $(1+\varepsilon)$ where $\varepsilon > 0$ is an arbitrarily small given constant reflecting the desired accuracy and 2) the size of $S$ and its update time per client position update is only polynomial in $k \log(n)/\varepsilon$. In particular, our kinematic coreset extends both the current theory and generalizes the practical application for coresets in the following ways:

i) *Theory*: Our coreset is applicable to a general class of problems related to the classic $k$-center problem where, in addition, centers are subject to constraints such as maintenance of a connected Euclidean spanning tree, and maximum velocity or control input limitations.

ii) *Practical Application*: Our coreset provides consistency across iterations by updating only to reflect client movement that effects the cost of the network beyond a specified threshold; thus resulting in greater stability of the center positioning.

*2) **Experimental Results**:* We test the computational efficiency and approximation error bound properties of our kinematic coresets on large problem sizes of up to $n = 2000$ in simulation as well as on a small scale hardware implementation using two Kuka Youbot robots (servers) that must react online to five clients moving over *a priori* unknown trajectories.

## C. Paper Roadmap

The structure of this paper is as follows. We define our representative set and a *static* algorithm for finding this representative set in Section III. We define key characteristics of an algorithm that computes a representative set with error bounds on optimal server placement for general communication problems defined in Section IV. In Section III we present an algorithm for computing *kinematic* updates to our representative set such that this set can be updated quickly and where points critically influencing the cost are tracked such that we maintain desired error bounds with respect to optimal server placement computed over the full set of clients. Finally in Section V-B we provide empirical evaluation showing the time complexity and accuracy of our kinematically updated representative set for practical implementations where routing vehicles have both connectivity constraints and control effort constraints.

## II. Problem Statement

Given $k$ moving clients, find server locations so that the entire heterogeneous system of servers and clients is connected, or serviced. We assume control of the server vehicles but do not assume control over the clients. Our approach is to determine the kinematic coreset, or sparse representative set, of clients and control the servers to track this coreset.

In the geometry literature, the servers are called *centers* and the clients are called *points*, and the problem is related to the $k$-center or connected $k$-center problems. In the $k$-center problem we are given an integer $k \geq 1$ and a set $P$ of $n$ points in $\mathbb{R}^d$. We wish to compute a set of centers with positions $\{c_1, \ldots, c_k\} \in C \subseteq \mathbb{R}^d$ such that the maximum point-center distance is minimized, i.e, minimize

$$r(P, C) := \max_{p \in P} \mathrm{Dist}(p, C). \quad (1)$$

The *connected* $k$-center cost, $rb(P, C)$, additionally optimizes the distance between neighboring centers as defined by the minimum Euclidean spanning tree $T^*(C)$ over $C$.

$$rb(P, C) = \max\{r(P, C), b(C)\} \quad (2)$$
$$b(C) := \max_{(c,c') \in T^*(C)} \mathrm{dist}(c, c').$$

For a given Euclidean spanning tree $T(C)$ the minimum communication power needed to maintain connectivity amongst centers is the worst case mutual connectivity captured by the *bottleneck edge* $b(C)$, or longest edge, of $T(C)$. Figure 2 shows some of the main differences between $k$-center and connected $k$-center solutions. In particular this schematic shows that 1) even if the optimal clustering (point assignments to centers) is known, one cannot “divide and conquer” by applying the $k = 1$ solution to each cluster, due to the connectivity constraints between the centers themselves, see Figure 2(a), and 2) although the $k = n$ case has a trivial solution for the $k$-center problem, the same is not true for the connected $k$-center problem due to the coupled effects of point clustering and a connected spanning tree over the centers that must be optimized simultaneously, see Figure 2(b). The interested reader is referred to [7] for an in-depth explanation of the connected $k$-center problem.

Let $C^*$ denote the set that minimizes $r(P, C)$ over every set $C$ of $k$ centers. For $\varepsilon > 0$, a $(k, \varepsilon)$-*coreset* for $P$ is a subset $S$ such that for every point $p \in P$ we have

$$\mathrm{Dist}(p, S) \leq \varepsilon r(P, C^*). \quad (3)$$

In particular, by the triangle inequality, for every set $C$ of $k$ centers,

$$(1 + \varepsilon) r(P, C) \geq r(S, C) \geq (1 - \varepsilon) r(P, C). \quad (4)$$

We are interested in such a set $S$ that is as small as possible.

By the above definition, computing the optimal positioning of $k$ centers (i.e, $k$-center) for the coreset $S$, would yield a $(1+\varepsilon)$ approximation $\tilde{C}$ such that $r(P, \tilde{C}) \leq (1+\varepsilon) r(P, C^*)$ to the optimal solution of the original set $P$. Moreover, this holds for any optimal positioning with additional constraints on the feasible positioning of the centers. Formally, for a (possibly infinite) set of candidate solutions $\mathbb{C}$ of centers in $\mathbb{R}^d$, we have by (4) that the optimal solution of $S$,

$$r(S, C^*) := \min_{C \in \mathbb{C}} r(S, C)$$

is an approximate optimal solution for $P$

$$r(S, C^*) \geq (1 - \varepsilon) \min_{C \in \mathbb{C}} r(P, C).$$

A *kinematic* $(k, \varepsilon)$-*coreset* is a data structure that *dynamically* updates the coreset whenever a point updates its new position . More precisely, the data structure consists of a $(k, \varepsilon)$-coreset $S$ for the set of points $P$, and an update method $\mathrm{MOVE}(p, p')$ that gets a point's position $p \in P$ and replaces it by $p' \in \mathbb{R}^d$. That is, both $P$ and $S$ are updated in each call to the MOVE method. We wish to maintain such a set $S$ of size as small as possible, and also to minimize the execution time of a call to MOVE.

## III. Algorithms

In Algorithms 2–5 we define the main procedure MOVE for updating the coreset, together with its sub-routines. We also provide a sample optimization problem that we run on the coreset in our experiments, for computing a set of centers that is close to the points, with additional restriction on maximum distance between centers. The procedure $\mathrm{INIT}(P)$ (Algorithm 1) is called once with the initial position of points set $P$. It runs the static version of our coreset construction from [6]. The data structure maintains the coreset in each call to $\mathrm{MOVE}(p, p_a)$, and correctness follows from Theorem 4.1.

To minimize the changes to the coreset, our data structure maintains a partition of the points into $O(\log n)$ resolution levels. The first level is large and represents the “main stream" or large dense clusters of the points that are not sensitive to a small fraction of points that may change their position. The last level is very small and consists of few “outliers" or isolated points that change their location frequently. Each of the $O(\log n)$ levels has its own coreset of $m = O(k \log(n)/\varepsilon)$ points. This yields a coreset of size $O(k \log^2(n)/\varepsilon)$. When a point updates its location the method $\mathrm{MOVE}(p, p_a, i)$ is called. The parameter $p$ denotes the last recorded position, and $p_a$ denotes the new position. The last recorded position can be saved on the point or center side. The data structure then computes which of the following actions should be taken:

**No update (Line 3).** Our data structure maintains the distances of each point to its closest coreset point at its level. These distances are stored in a binary heap for fast updates. A binary heap has the property that a value in a heap's node is always larger than its childs node. Therefore, the root of the heap contains the largest distance from a point to the coreset of the heap in that level. When a point updates its new location, it also sends a pointer to its node in the heap. If the heap is still valid after the change (the new distance is still larger then the node's child and smaller than its parent) no other action is taken. This is the fastest update type and takes constant $O(1)$ time.

**Heap update (Line 4).** When the updated distance of a point to the closest coreset point in its level does not preserve

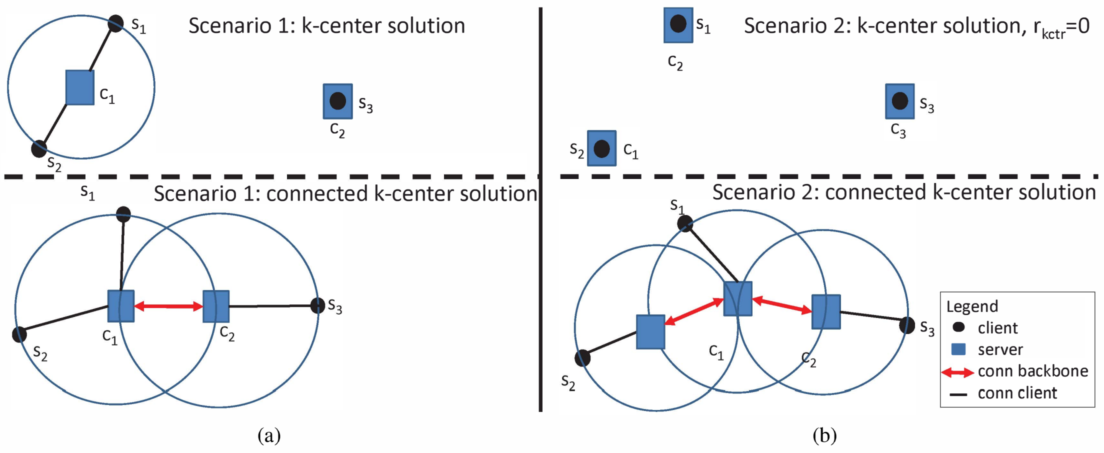
Scenario 1: k-center solution
Scenario 2: k-center solution, r_kctr=0
Scenario 1: connected k-center solution
Scenario 2: connected k-center solution
Legend
client
server
conn backbone
conn client
(a)
(b)

Fig. 2: Schematic drawings showing the differences between the $k$-center and connected $k$-center solutions for the $n$=3, $k$=2 case (a) and the $n$=3, $k$=3 case (b). Note that points have the same positions for all scenarios depicted above.

the structure of the heap (the new distance is smaller than the node's childs or larger than its parent), we need to "heapify" the node down or up the heap to preserve its structure. The update time depends on the number $\ell$ of such switches with other nodes. Since the height of the heap is $O(\log n)$ these changes take time $\ell \leq O(\log n)$.

**Level update (Lines 7–10).** After a series of heap updates, a point may be the farthest from its level coreset, and reaches the root of its level's heap. In this case, if a new point is added to the level with a smaller distance to the coreset, we remove the point in the root to a different resolution level, or even $\ell$ levels. The update time for such a change is $O(\ell)$ where $\ell \leq \log_2 n$ is the difference between the current level and the new level of the point.

**Coreset update (Line 6).** Every level maintains its coreset, which is a uniform random sampling of size $k$ from its points positions during different times ("snapshots"). That is, when a point position is chosen for the coreset, the point itself may continue to move, but its "recorded" coreset point is static until it is removed from the coreset. We thus call the coreset points "virtual points".

When too many points (constant factor) have entered or left the level's heap using heap updates, the coreset should also be updated. Updating a point in the coreset may affect all the points that it serves in the level, and also next levels. However, since the coreset is a random sample, update should occur very rarely in the higher levels (which contain large clusters) and may occur frequently in the lower levels (the small sets of outliers). Based on this observation, we prove that the overall expected running time of such an update is at most $O(\log n)$.

Algorithm 3 handles the case where $p$ is inserted or deleted from the $i$th level. That is, $p$ was one of the $|Q_i|$ closest points to $S_i$ but not after the call to MOVE, or vice versa. Intuitively, $p$ has left its cluster and has moved from the "main stream" toward a different level of resolution. In this case, we insert (respectively, remove) $p$ to its new (respectively, old) heap and continue recursively to update $p$ in the next level.

In case the size of the heap of $Q_i$ is not $cP_i$ for some $c \in (1/4, 3/4)$ (see Line 2 in Fig. 1), then we also need to balance the heaps by moving the root $r$ of one of the heaps to the other one and recursively update this change in the next pair $(Q_{i+1}, S_{i+1})$.

Finally, we handle the case where there is no $c \in (1/4, 3/4)$ such that $|S_i| \in c(k + \log n)$. That is, $p_i$ was removed or inserted to $S_i$. In this case, we recompute all the data structures that correspond to the pairs $(Q_i, S_i), \cdots, (Q_{|D|}, S_{|D|})$. As we prove in our main theorem, this event is rare (happens with probability at most $1/n$).

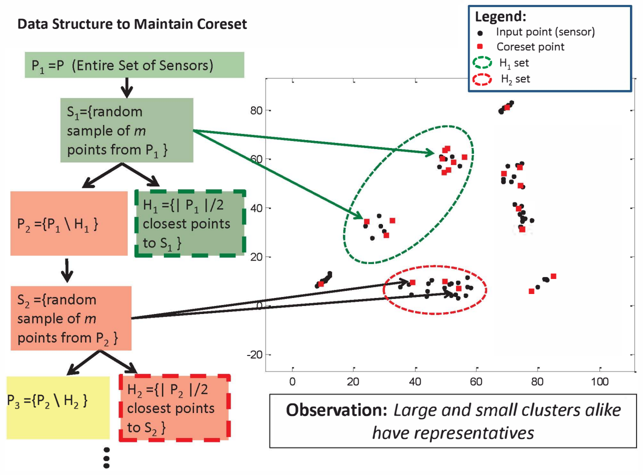
Data Structure to Maintain Coreset
P_1 =P (Entire Set of Sensors)
S_1={random sample of m points from P_1 }
P_2 ={P_1 \ H_1 }
H_1 ={| P_1 |/2 closest points to S_1 }
S_2 ={random sample of m points from P_2 }
P_3 ={P_2 \ H_2 }
H_2 ={| P_2 |/2 closest points to S_2 }
Legend:
Input point (sensor)
Coreset point
H_1 set
H_2 set
Observation: Large and small clusters alike have representatives

Fig. 3: Typical test scenario where points move randomly between clusters forming large and small clusters. The red squares are sampled input points from a $(k, \varepsilon)$-coreset demonstrating that most clusters (both large and small) are fairly sampled in contrast to uniform random sampling where several small clusters are often missed thus adversely affecting approximation quality. On the left, key properties of the construction of a $(k, \varepsilon)$-coreset are explained.

## IV. ANALYSIS

In this section we prove that the algorithms presented in the previous section provide a kinematic coreset $S$ that 1) is updated in time polynomial in $\frac{k \log(n)}{\varepsilon}$ for an input set of $n$ sensors, and 2) captures information about points most

**Algorithm 1:** INIT$(P, m)$

```
Input:  A set $P$ of $n$ points, and an integer $m \geq 1$
Output: A set $S$ that satisfies Theorem 4.1.
1 $i \leftarrow 1$; $P_1 \leftarrow P$
2 while $|P_i| > m$ do
3     $S_i \leftarrow$ A uniform random sample of $m$ points from $P_i$, with replacement.
4     $H_i \leftarrow$ A set of $c_i|P_i|$ points $p \in P_i$ with the smallest distance Dist$(p, S_i)$ for some $c_i \in (1/4, 3/4)$.
5     $P_{i+1} \leftarrow P_i \setminus H_i$
6     $i \leftarrow i + 1$
7 $S_i \leftarrow P_i$; $H_i \leftarrow P_i$
8 $S \leftarrow S_1 \cup \cdots \cup S_i$
9 return $S$
```

critically influencing the cost by reactively updating itself to maintain an upper bound on the approximation error as compared to the optimal cost by a factor of at most $(1+\varepsilon)$.

First we show that a coreset computed by the static approach in Algorithm INIT provides the desired $(1+\varepsilon)$, $\varepsilon \in (0, 1/2)$, error bound on the optimal cost for the $k$-centers. A corollary to this provides that any algorithm obeying three key properties of the INIT algorithm also produces a $(k, \varepsilon)$-coreset for our communication costs. Proposition 4.6 further generalizes this claim so that any $(k, \varepsilon)$-coreset for $P$ is also a coreset for any perturbed set $P_A$ as long as the magnitude of the perturbation is bounded below some constant factor. Finally we prove our main result: that our Algorithm MOVE for updating a *kinematic* coreset in time polynomial in $\frac{k \log(n)}{\varepsilon}$ indeed satisfies Corollary 4.2 and thus provides the error guarantees for an arbitrarily moving set of sensors.

*Theorem 4.1:* Let $P$ be a set of $n$ points, $k \geq 1$ be an integer and $\varepsilon > 0$ be a constant. Let $S$ denote the output of the algorithm INIT$(P, m)$ for an appropriate $m = O(k \log n / \varepsilon^2)$. Then, with arbitrarily high probability of at least $1 - 1/n$, $S$ is a $(k, \varepsilon)$-coreset for $P$ of size $|S|$ that is polynomial in $\frac{k \log(n)}{\varepsilon}$.

*Proof:* The algorithm INIT is a small modification to the Static BiCriteria algorithm from [6] and thus the proof from that paper holds with minor modification to account for a different constant factor $c \in (\frac{1}{4}, \frac{3}{4})$ of points taken from $P_i$ at every iteration. For brevity we will not repeat the proof here. ■

A result of this theorem is that any algorithm that maintains the properties of the coresets $S_i$ from Algorithm INIT as an invariant also produces a $(k, \varepsilon)$-coreset for $P$. In particular we state the following Corollary:

*Corollary 4.2:* Let $P$ be a set of $n$ points, $j \geq 1$ be an integer, and $(H_1, \cdots, H_j)$ be a partition of $P$. Let $S \subseteq P$ and $(S_1, \cdots, S_j)$ be a partition of $S$. Let $m = O(k \log n / \varepsilon^2)$ be defined as in the previous theorem. Suppose that the following properties 4.3-4.5 hold for every $i = 1, \cdots, j-1$:

*Property 4.3:* $S_i$ is a random sample of size $|S_i| \geq m$ from $P_i$

*Property 4.4:* $H_i$ is the set of $c|P_i|$ points $p \in P_i$ with

**Algorithm 2:** MOVE$(p, p_a, i)$:
Move $p \in P_i$ to its actual position $p_a$.

```
Input: A virtual point $p \in P_i$, its actual position $p_a$, and an integer $i \geq 1$.
1  $h_i \leftarrow \max_{q \in H_i}$ Dist$(q, S_i)$
   /* $S_i$ is the coreset of level $i$.              */
2  if $p_a, p > h_i$ then
       MOVE$(p_a, p, i+1)$
       /* Check next levels recursively               */
3  else if $p \in H_i$ and dist$(p_a, p) \leq$ Dist$(p, S_i)/2$ then
       return /* No update                            */
4  else if dist$(p, S_i)$, dist$(p_a, S_i) \leq h_i$ then
       Replace $p$ with $p_a$ in $H_i$ /* Heap update  */
5      if $p \in S_i$ then
6          Reconstruct levels $i, i+1, i+2, \ldots$
           /* Coreset update                          */
       return
   /* At this line $p \in H_i$ and $p_a \notin H$,      */
   /* or vice versa                                   */
7  if dist$(p, S_i) \leq h_i$ then
8      /* $p \in H_i$ but $p_a \notin H$                */
       UPDATE$(p, i, \text{delete})$
9      UPDATE$(p_a, i, \text{insert})$
   else
       /* $p_a \in H_i$ but $p \notin H_i$              */
       UPDATE$(p_a, i, \text{delete})$
10     UPDATE$(p, i, \text{insert})$
11 return
```

**Algorithm 3:** UPDATE$(p, i, \textit{action})$
Insert/Delete $p$ from $P_i$

```
Input: A point $p \in P_i$, and $action \in \{\text{insert}, \text{delete}\}$
1 UPDATESAMPLE$(p, i, action)$
2 if $S_i$ was changed during the execution of previous line then
3     Reconstruct levels $i, i+1, i+2, \ldots$
4     return
5 Insert/Delete $p$ to/from its heap $H \in \{H_i, \overline{H}_i\}$.
  In case of ties, choose smallest heap.
6 if $H = \overline{H}_i$ then
7     UPDATE$(p, i+1, action)$
8 BALANCE$(i)$
9 return
```

**Algorithm 4:** BALANCE$(i)$
Balance the pair of heaps at level $i$

```
Input: A coreset level $i \geq 1$
1 if $|\overline{H}_i| \notin [1/4, 3/4]$ then
2     $p \leftarrow$ root of the larger heap in $\{H_i, \overline{H}_i\}$
3     UPDATE$(p, i, \text{delete})$
4     UPDATE$(p, i, \text{insert})$
      /* $p$ is inserted to the smaller heap             */
```

**Algorithm 5:** UPDATESAMPLE$(p, i, action)$:
Update the sample $S_i$ with the deletion/insertion of $p$

```
Input:  A point $p$ that should be inserted/deleted from
        $P_i$ according to $action \in \{\text{insert}, \text{delete}\}$.
1 if $action = $ insert then
2     $P_i \leftarrow P_i \cup p$
3     $r(p) \leftarrow$ A random number, sampled uniformly over
          the interval $[0, 1]$
  else
4     $P_i \leftarrow P_i \setminus \{p\}$
5 Remove from $S_i$ every $q \in S_i$ such that
  $r(q) > k \log_2 n / |P_i|$
6 Insert to $S_i$ every $q \in P_i$ such that $r(q) \leq k \log_2 n / |P_i|$
```

**Algorithm 6:** RELAXATION$(P, C, \gamma)$
Compute connected centers that are attainable from $C_{t-1}$

```
Input:  A set $P$ of $k$ points, the current set $c'_1, \cdots, c'_k$
        of centers, and max velocity bound $\gamma > 0$
Output: A set $C$ of $k$ centers and their cost $r$
1 for $i \leftarrow 1$ to $k$ do
      /* Uniquely assign each center to a
         close point                          */
2     $p_i \leftarrow \arg\min_{p \in P} \|c'_i - p\|$
3     $P \leftarrow P \setminus \{p_i\}$
4 $(C, r) \leftarrow \arg \min_{C=\{c_1,\cdots,c_k\} \subseteq \mathbb{R}^2, r \geq 0} r$   s.t. $\forall i = 1, \cdots, k$
      $\|c_i - p_i\| \leq r,$
      $\|c_i - c'_i\| \leq \gamma.$
5 return $(C, r)$
```

the smallest $\text{Dist}(p, S_i)$, for some $c \in (1/4, 3/4)$.

*Property 4.5:* $P_{i+1} = P_i \setminus H_i$

Then $S$ is a $(k, \varepsilon)$-coreset for $P$

Since it is inefficient and costly for the coreset to change as point vehicles move over small distances that do not have a significant effect on cost, we must be able to show that a coreset $S$ for an input set $P$ is also a coreset for a perturbed set $P_A$ if the perturbation is small in magnitude. The following proposition defines tolerable perturbations such that this property holds and is a key component of our kinematic update algorithm MOVE.

*Proposition 4.6 (Coresets for Perturbed Sets):* A $(k, \varepsilon)$-coreset, $S$, for an input set $P$, is also a $(k, \varepsilon)$-coreset for any other set $P_A$ if for every $q \in P_A$ there exists a *unique* point $p \in P$ such that $dist(p, q) \leq \frac{1}{2} Dist(p, S_i)$, where $i$ is arbitrary and corresponds to the coreset level (line 2 from Algorithm INIT) to which $p$ belongs. Then the coreset assumption $Dist(q, S) \leq O(1)\varepsilon opt \ \forall q \in P_A$ holds for all points $q \in P_A$ up to a constant factor $O(1)$ where $S = \cup_{i=1}^{\log(n)} S_i$.

*Proof:* Let $p$ be the unique virtual position of a point with actual position $q \in P_A$. The virtual position $p$ of a point is the last recorded position of $p$ and the set $P$ contains all the virtual points' positions. From the Proposition assumption we have that every $q \in P_A$ has a virtual point $p \in P$ such that $dist(p, q) \leq \frac{1}{2}\text{Dist}(p, Q_i)$. Indeed for some $p \in H_i$ for any level $i$, if the above claims are satisfied we have

$$\begin{aligned}\text{Dist}(q, Q_i) &\leq \text{dist}(q, p) + \text{Dist}(p, Q_i) \\ &\leq \frac{1}{2}\text{Dist}(p, Q_i) + \text{Dist}(p, Q_i) \leq \frac{3}{2}\varepsilon opt\end{aligned}$$

where the first line follows from the triangle inequality, the second line follows from our assumption, and the last inequality proves the proposition. ■

Lastly we prove our main results on the time complexity and accuracy of our kinematic coreset resulting from a call to the MOVE algorithm. The proof for the following theorem is an extension of similar proofs from [5], [6] and we omit it due to lack of space.

*Theorem 4.7:* Let $P'$ be a set of $n$ points, $k \geq 1$, and $\varepsilon \in (0, 1/2)$. Let $P$ denote the set of points after a call to INIT$(P', k/\varepsilon^2)$ followed by a finite sequence of calls to the MOVE algorithm. Then, the following holds (i) $S$ is a $(k, \varepsilon)$-coreset of $P$, (ii) $|S|$ is of size polynomial in $\frac{k \log(n)}{\varepsilon}$, (iii) The expected execution time of each such call to MOVE is polynomial in $\frac{k \log(n)}{\varepsilon}$, using an appropriate implementation.

*Proof:* (Sketch) (i) The main observations are that UPDATESAMPLE maintains a random sample $S_i$ from $P_i$. Therefore, after moving a point using the MOVE procedure, the coreset has the same properties as the output of the INIT algorithm and thus must be a $(k, \varepsilon)$-coreset for $P$.
(ii) by the two last lines of the procedure UPDATESAMPLE, we have that $S_i$ contains all the points in $P_i$ that were assigned random values $r(p) \in (0, 1)$ that are less than $k \log_2 n / |P_i|$. By Hoeffding-Chernoff inequality, the expected size of $S_i$ is $\Theta(k \log n)$. Since we have $O(\log n)$ such sample sets $S_i$, the overall size of the coreset is polynomial in $O(k \log n)$. (iii) Since the update takes expected $O(\log n)$ time and there are $O(\log n)$ levels, the overall update time is polynomial in $\log n$ if none of the sample sets are updated. By Lines 5 and 6 of Algorithm 5, the keys $r(q)$ of the samples $q \in S$ have values at most $k \log n / |P_i|$. When we insert or delete a point from $|P_i|$ and update the sample, this threshold changes very little. The probability that the random number $r(q) \in (0, 1)$ falls within this gap for one of the points $q \in P_i$ is $1/|P_i|$, meaning that indeed $S$ changes very rarely (once every $O(|P_i|)$ updates). ■

## V. EMPIRICAL RESULTS

We extensively test the computational time efficiency of computing a kinematic coreset, and of computing a $k$-center or connected $k$-center cost over this coreset, as well as the resulting approximation costs as a function of desired coreset size. Simulation results over large, up to $n = 2000$ points, data sets show the asymptotic properties of our coresets. We also implement our algorithm on a small $n = 5$ example problem in a hardware implementation to provide intuition behind why coresets work and demonstrate online adaptation of robots to *a priori* unknown point movement.

### A. Hardware Experiment

We implemented our kinematic coreset algorithm for a heterogeneous system consisting of $n = 5$ human points and

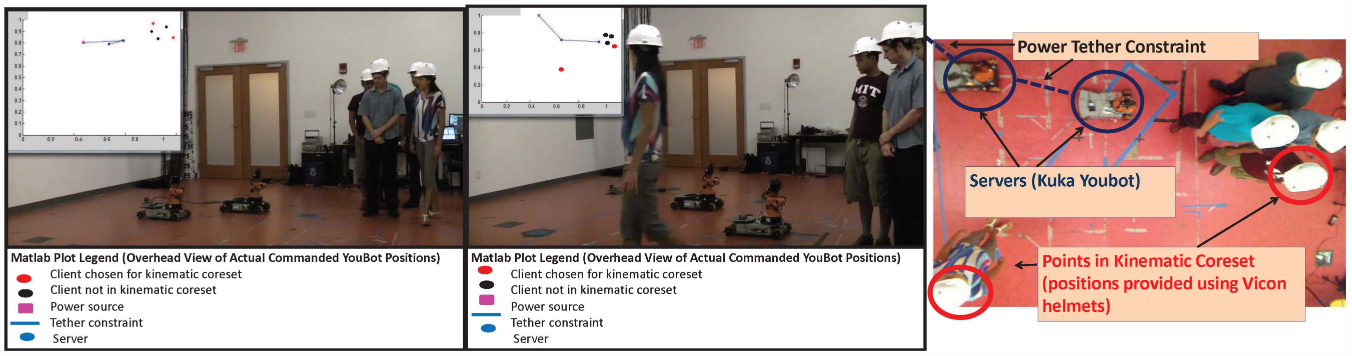
Matlab Plot Legend (Overhead View of Actual Commanded YouBot Positions)
Client chosen for kinematic coreset
Client not in kinematic coreset
Power source
Tether constraint
Server
Matlab Plot Legend (Overhead View of Actual Commanded YouBot Positions)
Client chosen for kinematic coreset
Client not in kinematic coreset
Power source
Tether constraint
Server
Power Tether Constraint
Servers (Kuka Youbot)
Points in Kinematic Coreset (positions provided using Vicon helmets)

Fig. 4: Side and overhead views of hardware experiments for heterogeneous Kuka Youbot (center) and human (point) systems. Arbitrary intial positions with coreset (left), centers tracking moving point (middle), overhead view of points divided into two clusters and resulting coreset (right). Matlab plots show computed kinematic coreset points (red), commanded Youbot positions (blue), and power tether constraints (blue line).

$k = 2$ robot centers. The five points were instructed to walk for 10 minutes within the sensing envelope of a Vicon motion capture system, where their positions were sent in realtime to a single Intel Core 2 Duo 2.4 GHz computer running our algorithm. No *a priori* knowledge of the points' movements was provided to the two centers, which were Kuka Youbot omnidirectional ground robots running the Robot Operating System. Figure 1 shows an overhead view of our hardware setup.

Using Matlab R2012a and the CVX convex optimization software [11], kinematic coresets of both two and three points were maintained and used to calculate the connected $k$-center costs from Equation (2). Figure 4 shows the movement of the points around the room and the resulting choice of the two point coreset $S$ maintained by repeated calls to MOVE. In addition, the optimal cost over the five points was calculated and used for positioning the centers as described in [7]. These optimal cost calculations of $n^{O(k)}$ computation time were made possible due to the small number of agents in the system, and thus were used to evaluate our algorithm's performance. Table I shows the computation time and solution accuracy of our algorithm compared to the optimal connected $k$-center solution and a naïve sampling strategy.

With our experiment, we were able to demonstrate the ability of our algorithm to detect newly formed clusters. The plots in Figure 4 show that although the coreset is only of two points, a representative point (red) is found in every cluster of points, which is a driving factor for the low resulting error of $\varepsilon = 0.14$ with respect to the optimal solution. In addition, the center position computation takes 2.2 s, which is a factor of $97\times$ faster. In contrast, a sample set of two points chosen randomly often misses one of the point clusters, thus resulting in a higher $\varepsilon = 0.5$ approximation cost. We expect that for the case where all points are equally distanced, the solution computed over a kinematic coreset would produce similar approximation costs to that of a uniform random sample. However, the clustering of points often arises in practice, especially for large data sets. This small scale implementation demonstrates that the properties we prove for large systems similarly holds for small systems where the $O(\cdot)$ notation is irrelevant.

| Metric | KC 2 Pts | U 2 Pts | KC 3 Pts | U 3 Pts |
|---|---|---|---|---|
| AvgCost/OPT | 1.14 | 1.50 | 1.02 | 1.30 |
| VarCost | 0.10 | 0.16 | 0.08 | 0.11 |
| OPTtime/Time | 97 | 102 | 19 | 19 |
| VarTime (sec) | 0.29 | 0.34 | 0.41 | 1.19 |

TABLE I: This tables summarizes the result of our hardware experiment employing two centers and five points. Computing new center positions over a kinematic coreset (KC) of two points is $97\times$ faster with approximation cost of $\varepsilon = 0.14$ compared to performing computation over entire input set of five points. In contrast, naïvely sampling two input points at random (U) produces an approximation cost of $\varepsilon = 0.5$ at comparable computational speed. Calculations of center positions using three points shows similar trends.

## B. Numerical Simulation

We present empirical results for update time, and quality of the coreset $S$ against different input set sizes $n$. Our test scenario is of an input set of points, $P$, moving randomly between depots located at three corners of the environment. In particular, at the beginning of a run we randomly select a subset of 10 points from all depots that choose with equal probability one other depot to move to. This is representative of a situation where points are scouting three areas of major interest where some vehicles may be recalled to other areas of higher interest. We compare the performance of *uniform random sampling, static bicriteria* and *kinematic coresets* for maintaining a representative set of the input $P$. We specify our three compared methods below where $\text{poly}(x)$ means "polynomial in x":

***Uniform Random Sampling***: the sample set $S$ is a uniform sample of $m = \text{poly}(\frac{k\log(n)}{\varepsilon})$ points from the input set $P$.

***Static Coreset***: the sample set $S$ in this case is of cardinality $m = \text{poly}(\frac{k\log(n)}{\varepsilon})$ and is a $(k, \varepsilon)$-coreset returned from Algorithm INIT computed from scratch every time the positions of input set $P$ are updated.

***Kinematic Coreset***: the sample set $S$ in this case is of cardinality $m = \text{poly}(\frac{k\log(n)}{\varepsilon})$ and is a $(k, \varepsilon)$-coreset that is updated using Algorithm MOVE each time the positions of the input points in $P$ are updated.

We measure performance between all three methods in three different ways. First we compare the time needed to update the representative set $S$. Secondly, we compute the coreset cost $\text{dist}_{p\in \text{P}}(p, S)$ which is how well the entire input

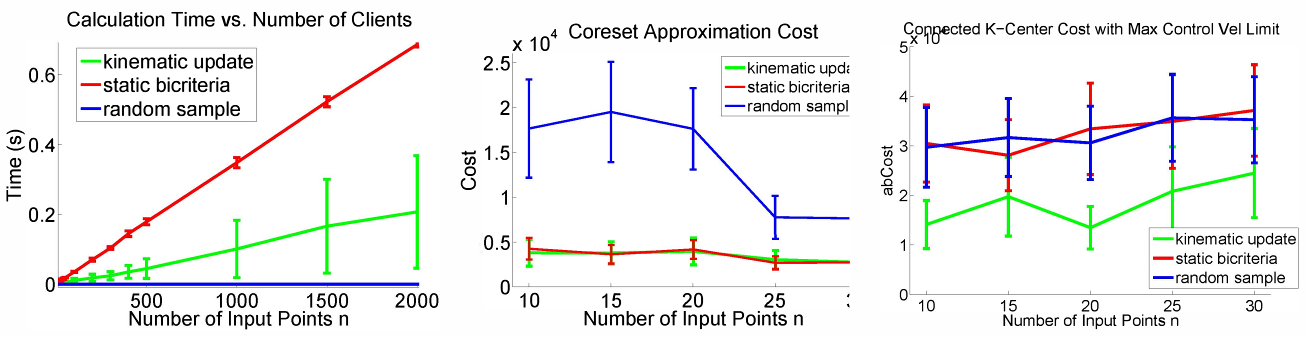

Fig. 5: These plots show calculation time to compute updated representative set after each position update for points in the input set $P$ for up to $n = 2000$ input points averaged over 500 runs, and the cost over each representative set after each position update for points in the input set $P$ for up to $n = 20$ input points averaged over 500 runs. Results show that kinematic updates perform comparably to using a static coreset in terms of accuracy and random sampling performs up to $5\times$ worse than both of these methods. The last plot shows the case of center velocity constraints where a kinematically updated coreset outperforms a statically updated coreset since in the former case there is consistency between iterations.

set $P$ is approximated by $S$. Lastly, we analyze the connected $k$-center cost from Equation (2) that takes into account a communication constraint between centers, with an added dynamic constraint on the vehicles that limits how far the centers can move between consecutive iterations, holds for any physical system. Calculation of the connected $k$-center cost demonstrates that kinematically updated coresets are the most cost effective for physical systems that cannot tolerate arbitrarily different solutions (since centers cannot move infinitely fast).

Figure 5 bolsters our main time complexity result from Theorem 4.7 and indeed demonstrates that the updates for the kinematic coreset are updated much faster, providing a larger computational complexity advantage over the updates for the Static Bicriteria coreset as $n$ increases. Additionally, Figure 5 demonstrates that the coreset which is updated kinematically provides similar $k$-center cost as compared to the Static Bicriteria coreset in stark contrast to a purely random sample of the input point set which has minimal computational complexity but performs up to $5\times$ worse than both the kinematic and static bicriteria algorithms.

For our simulation we do not compute the exact connected $k$-center cost which takes exponential time in $k$ to compute as discussed in [7], rather we compute a relaxation where we pair every coreset point in $S$ to a unique center as described in Algorithm 6. Figure 5 shows that the kinematic coreset performs *better* than the coreset computed using Static BiCriteria for a cost that takes into account displacement constraints on the centers between iterations. This is because the Move algorithm updates the coreset intelligently as points move whereas the static bicriteria calculates a new coreset from scratch each iteration and thus has no consistency between iterations.

## VI. Conclusion

In this paper we have provided an algorithm for maintaining a sparse set of representative clients that is updated as the client vehicle team moves arbitrarily through the environment. Additionally we present theory that guarantees that our representative set can be updated in time polynomial in $(k \log(n)/\varepsilon)$ and provide the same error bounded approximate $k$-center cost as the case of computing the entire representative set from scratch using static coresets from our previous work [7]. Our empirical results additionally show that for systems of practical interest that have physical limitations on how fast server vehicles can move, updating the existing coreset kinematically is favorable over computing a static coreset since consistency is maintained over consecutive iterations.

## References

[1] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *Journal of the ACM*, 51(4):606–635, 2004.
[2] T.M. Chan. Dynamic coresets. *Discrete & Computational Geometry*, 42(3):469–488, 2009.
[3] Alejandro Cornejo, Fabian Kuhn, Ruy Ley-Wild, and Nancy Lynch. Keeping mobile robot swarms connected. In *Proceedings of the 23rd international conference on Distributed computing*, DISC'09, pages 496–511, Berlin, Heidelberg, 2009. Springer-Verlag.
[4] D.Spanos and R.Murray. Motion planning with wireless network constraints. In *Proceedings of the ACC*, 2005.
[5] D. Feldman and Daniel Golovin. Dynamic bicriteria approximations. *Manuscript*, 2011.
[6] Dan Feldman, Amos Fiat, Micha Sharir, and Danny Segev. Bi-criteria linear-time approximations for generalized k-mean/median/center. In *Symposium on Computational Geometry'07*, pages 19–26, 2007.
[7] S. Gil, D. Feldman, and D. Rus. Communication coverage for independently moving robots. *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems*, 2012.
[8] Stephanie Gil, Samuel Prentice, Nicholas Roy, and Daniela Rus. Decentralized control for optimizing communication with infeasible regions. In *Proceedings of the 15th International Symposium on Robotics Research*, 2011.
[9] PR Giordano, A. Franchi, C. Secchi, , and H. Bulthoff. Bilateral teleoperation of groups of uavs with decentralized connectivity maintenance. In *Proceedings of RSS*, 2011.
[10] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38, 1985.
[11] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 1.21. `../../cvx`, April 2011.
[12] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *Proc. 36th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 291–300, 2004.
[13] S. Har-Peled and K. R. Varadarajan. High-dimensional shape fitting in linear time. *Discrete & Computational Geometry*, 32(2):269–288, 2004.
[14] Dorit S. Hochbaum. Easy solutions for the k-center problem or the dominating set problem on random graphs. In *Analysis and Design of Algorithms for Combinatorial Problems*, volume 109, pages 189 – 209. North-Holland, 1985.
[15] N.Michael, M. M. Zavlanos, V. Kumar, and G.Pappas. Maintaining connectivity in mobile robot networks. *Experimental Robotics*, pages 117–126, 2009.
[16] O. Tekdas, W. Yang, and V. Isler. Robotic routers: Algorithms and implementation. *Int. Journal of Robotics Research*, 29(1), 2010.