Robust Distributed Sensor Network Localization with Noisy Range Measurements

by

David Christopher Moore

B.S., California Institute of Technology (2003)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2005

Author .....
Department of Electrical Engineering and Computer Science
January 14, 2005

Certified by .....
Seth Teller
Associate Professor
Thesis Supervisor

Certified by .....
John J. Leonard
Associate Professor
Thesis Supervisor

Certified by .....
Daniela L. Rus
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Robust Distributed Sensor Network Localization with Noisy Range Measurements

by

David Christopher Moore

Submitted to the Department of Electrical Engineering and Computer Science
on January 14, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis describes a distributed, linear-time algorithm for localizing sensor network nodes in the presence of range measurement noise and demonstrates the algorithm on a physical network. We introduce the probabilistic notion of *robust quadrilaterals* as a way to avoid flip ambiguities that otherwise corrupt localization computations. We formulate the localization problem as a two-dimensional graph realization problem: given a planar graph with approximately known edge lengths, recover the Euclidean position of each vertex up to a global rotation and translation. This formulation is applicable to the localization of sensor networks in which each node can estimate the distance to each of its neighbors, but no absolute position reference such as GPS or fixed anchor nodes is available.

We implemented the algorithm on a physical sensor network and empirically assessed its accuracy and performance. Also, in simulation, we demonstrate that the algorithm scales to large networks and handles real-world deployment geometries. Finally, we show how the algorithm supports localization of mobile nodes.

Thesis Supervisor: Seth Teller
Title: Associate Professor

Thesis Supervisor: John J. Leonard
Title: Associate Professor

Thesis Supervisor: Daniela L. Rus
Title: Associate Professor

Acknowledgments

This work was funded by a grant from Project Oxygen and supported by a National Science Foundation Graduate Research Fellowship. Additional funding was provided in part by the Institute for Security Technology Studies (ISTS) at Dartmouth College, NSF award 0225446, ONR awards N00014-01-1-0675, N00014-02-C-0210, and N00014-03-1-0879, and DARPA TASK program award F30602-00-2-0585.

I would like to thank the Cricket project, especially Michel Goraczko, Bodhi Priyantha, and Hari Balakrishnan, for supplying hardware and programming assistance. Thanks also goes to the BMG (Building Model Generation) project for floorplans. Patrick Nichols wrote the constrained beacon graph generator. I am grateful to Erik Demaine for useful discussions and pointers to literature.

Most of all, I'd like to thank my advisors Seth Teller, Daniela Rus, and John Leonard for their invaluable guidance and support in producing this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Localization in sensor networks is defined as an algorithm that finds the Euclidean position for some or all of the nodes in the network. An example of the inputs and outputs to the localization problem is shown in Figure 1-1. Localization is an essential tool for the development of low-cost sensor networks for use in location-aware applications and ubiquitous networking [5, 23]. For example, the Global Positioning System (GPS) is a reliable technology for discovering the location of portable devices, however, it does not work indoors because a view of the sky is obstructed. Localization of sensor networks provides similar functionality to GPS, but it works indoors as well. A full-range of location-aware applications such as autonomous navigation, environmental monitoring, and "smart buildings" are realizable given a robust algorithm for sensor network localization.

This thesis describes a distributed algorithm for a specific type of sensor network localization in which nodes have the ability to estimate distance to nearby nodes, but such measurements are corrupted by noise. Distributed computation and robustness in the presence of measurement noise are key ingredients for a practical localization algorithm that will give reliable results over a large scale network. We formulate the problem as the following two-dimensional graph realization problem: given a planar graph with edges of known length, recover the Euclidean position of each vertex up to a global rotation and translation. This is a difficult problem for several reasons. First, there is often insufficient data to compute a unique position assignment for all nodes. Second, distance measurements are noisy, compounding the effects of insufficient data and creating additional uncertainty. Another problem is a lack of absolute reference points or anchor nodes[1] which could provide

[1]We use the term *anchor node* to refer to a node that has prior knowledge of its absolute position, either

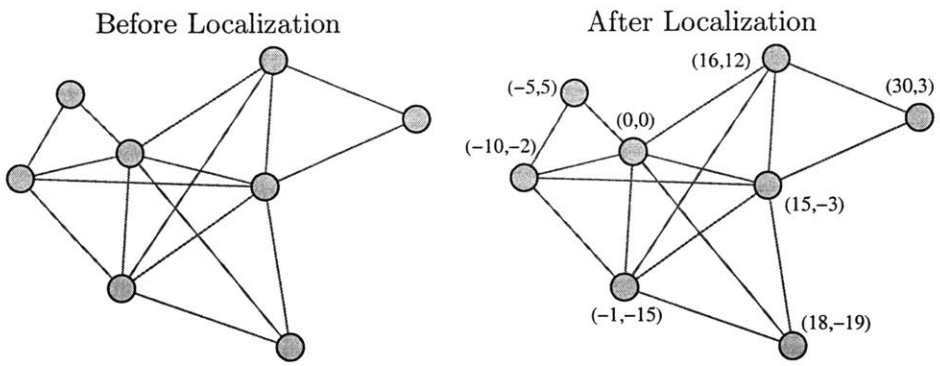


Figure 1-1: An example of the localization problem. The input to the algorithm (left) is a set of sensor network nodes with some set of known inter-node distances, depicted as straight lines. The output after localization is an assignment of Euclidean coordinates (2-dimensional in this case) to the network nodes.

a starting point for localization. Finally, it is difficult to devise algorithms that scale linearly with the size of the network, especially if data must be broadcast through the limited communications capacity of a wireless channel.

We present a distributed localization algorithm that gets around these difficulties by localizing nodes in regions constructed from *robust quadrilaterals*, a term that we formally define in Chapter 2. Localization based on robust quads attempts to prevent incorrect realizations of flip ambiguities that would otherwise corrupt localization computations. Furthermore, we show that the criteria for quadrilateral robustness can be adjusted to cope with arbitrary amounts of measurement noise in the system. The drawback of our approach is that under conditions of low node connectivity or high measurement noise, the algorithm may be unable to localize a useful number of nodes. However, for many applications, missing localization information for a known set of nodes is preferential to incorrect information for an unknown set.

A general result of our simulations is that even as noise goes to zero, nodes in large networks must have degree 10 or more on average to achieve 100% localization.

1.1 Algorithm Overview

At a high level, our network localization algorithm works as follows. Each node measures distances to neighboring nodes, then shares these measurements with the neighbors. This

by manual initialization or an outside reference such as GPS. This type of node is also called a *beacon*.

"one-hop" information is sufficient for each node to localize itself and its neighbors, which we call a cluster, in some local coordinate system. Coordinate transforms can then be computed between overlapping clusters to stitch them into a global coordinate system. Such stitching can be done in an on-line fashion as messages are routed through the network rather than attempting to solve for the full localization up front. Similar cluster-based approaches have been proposed before, but often suffer from gross localization errors due to graph flips that can compound over larger distances. Our novel use of robust quadrilaterals ensures that cluster-based localization does not suffer from such errors.

Figure 1-2 depicts an illustrative run of our algorithm. We find all sets of four nodes that are fully connected by distance measurements and are "well-spaced" such that even in the presence of measurement noise, their relative positions are unambiguous. We adopt this quadrilateral as the smallest subgraph that can be definitively realized, and define it as a *robust quad.* Additional robust quads can be "chained" to the original quad if they have 3 nodes in common with it. This approach allows each chained quad to localize its fourth node based on the 3 known positions common to the two quads using the standard technique of *trilateration* [4, 12]. This use of robust quadrilaterals enables our algorithm to tolerate noise by computing unique realizations for graphs that might otherwise be ambiguous.

Our algorithm has the following characteristics:

1. It supports noisy distance measurements, and is designed specifically to be robust under such conditions.

2. It is fully distributed, requiring no beacons or anchors.

3. It localizes each node correctly with high probability, or not at all. Thus, rather than produce a network with an incorrect layout, any nodes with ambiguous locations are not used as building blocks for further localization.

4. Cluster-based localization supports dynamic node insertion and mobility.

1.2 Related work

Eren et al. in [4] provide a theoretical foundation for network localization in terms of graph rigidity theory. They show that a network has a unique localization if and only if its

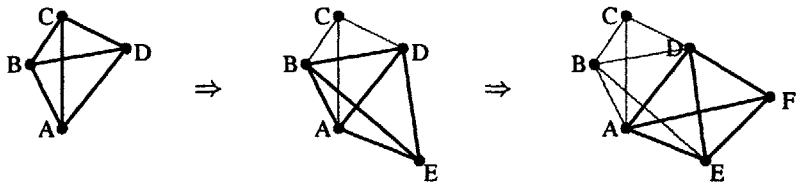


Figure 1-2: An example run of our algorithm to estimate the relative positions of node A's neighbors. Nodes ABCD form a *robust quad* because their realization is unambiguous even in the presence of noise. We select A as the origin of a local coordinate system and choose positions for B, C, and D that satisfy the six distance constraints. In the next step, node E is localized relative to the known positions of ABD using trilateration. This localization is unambiguous because ABDE also forms a robust quadrilateral. Continuing, the same procedure is used to localize node F which is part of the robust quad ADFE.

underlying graph is *generically globally rigid.* In addition, they show that a certain subclass of globally rigid graphs, *trilateration graphs*, can be constructed and localized in linear time. We take global rigidity and trilateration graphs one step further with robust quadrilaterals that provide unambiguous localizations and tolerate measurement noise.

In [18], Savvides et al. derive the Cramér-Rao lower bound (CRLB) for network localization, expressing the expected error characteristics for an ideal algorithm, and compare it to the actual error in an algorithm based on multilateration. They draw the important conclusion that error introduced by the algorithm is just as important as measurement error in assessing end-to-end localization accuracy. In [13] and [12], Niculescu and Nath also apply the CRLB to a few general classes of localization algorithms. Their *"Euclidean"* method is similar to our method of cluster localization in that it depends on the trilateration primitive. They also state the relevance of four-node quadrilaterals. In their case, the quads are constrained with five distance measurements — the sixth is computed based on the first five. Flip ambiguities are resolved using additional information from neighboring nodes. Their *"DV-coordinate"* propagation method presented in [12] is similar to our method in that clusters consisting of a node and its one-hop neighbors are first localized in local coordinate systems. Registration is then used to compute the transformations between neighboring coordinate systems. This idea of local clusters was also proposed by Čapkun et al. in [2]. However, neither algorithm considers how measurement noise can cause incorrect realization of a flip ambiguity.

A variety of other research attempts to solve the localization problem using some form of global optimization. Doherty et al. described a method using connectivity constraints

and convex optimization when some number of beacon nodes are initialized with known positions [3]. Ji and Zha use multidimensional scaling (MDS) to perform distributed optimization that is more tolerant of anisotropic network topology and complex terrain [9]. Priyantha et al. eliminate the dependence on anchor nodes by using communication hops to estimate the network's global layout, then using force-based relaxation to optimize this layout [15].

Other previous work is based on propagation of location information from known reference nodes. Bulusu et al. and Simic et al. propose distributed algorithms for localization of low power devices based on connectivity [1, 21]. Other techniques use distributed propagation of location information using multilateration [11, 19]. Savarese et al. use a two-phase approach using connectivity for initial position estimates and trilateration for position refinement [17]. Patwari et al. use one-hop multilateration from reference nodes in a physical experiment using both received signal strength (RSS) and time of arrival (ToA) [14]. Grabowski and Khosla maximize a likelihood estimator to localize a small team of robots, achieving some robustness by including a motion model in their optimization [6].

In this thesis we make several departures from previous research. Most importantly, no previous algorithm considers the possibility of flip ambiguities during trilateration due to measurement noise. Although the requirement of global rigidity as a means to avoid flips has been well established [4], the effects of measurement noise on global rigidity are not well understood. Our notion of robust quadrilaterals minimizes the probability of realizing a flip ambiguity incorrectly due to measurement noise. Error propagation during trilateration is derived in [13], but the potential for significant error due to flips is not considered. Secondly, like [2] and [12], we do not require anchor nodes, enabling localization of networks without absolute position information. This characteristic is important for localization in homogeneous ad-hoc networks, where any node may become mobile. Furthermore, manual beacon initialization can be error-prone or impossible, for example, in a sensor network deployed by a mobile robot.

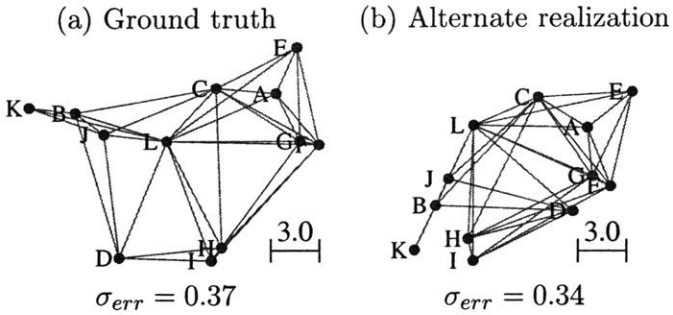


Figure 1-3: Example graph showing (a) true vertex positions and (b) an alternate realization of the graph in which inter-vertex distances, depicted as lines, are preserved almost exactly. The error metric σ_{err} is shown below each realization, with inter-vertex distances generated from a Gaussian distribution with a mean of the true distance and $\sigma = 0.35$. Thus, in this example, an incorrect realization of the graph fits the constraints *better* than the ground truth, showcasing why network localization is difficult.

1.3 Challenges of network localization

The difficulties inherent in localization can be easily demonstrated with an example. Consider the following metric that characterizes the error for a given localization,

$$\sigma_{err}^2 = \sum_{i=1}^{M} \frac{(\tilde{d}_i - \hat{d}_i)^2}{M} \tag{1.1}$$

where M is the number of distance measurements, $\tilde{d}_i$ is each distance computed from the localized positions, and $\hat{d}_i$ is each measured distance. Without ground truth, σ_{err} tells us how well a computed localization fits the constraints. Figure 1-3 shows why minimizing this error metric is insufficient for localization. In this example, two possible realizations shown for the network have similar values for σ_{err}, but the ground truth actually has higher error than an incorrect realization. This demonstrates the need for an algorithm that appropriately handles nodes with ambiguous positions by refusing to assign a position to any node that has more than one possible locus for its position. We now formally address why an algorithm based primarily on numerical optimization of the distance constraints fails.

In graph theory, the problem of finding Euclidean positions for the vertices of a graph is known as the *graph realization problem.* Saxe showed that finding a realization is strongly NP-hard for the two-dimensional case or higher [20]. However, knowing the length of each graph edge does not guarantee a unique realization, because deformations can exist in the

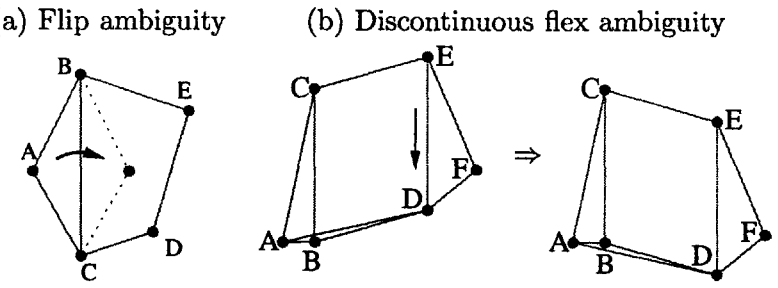


Figure 1-4: (a) Flip ambiguity. Vertex A can be reflected across the line connecting B and C with no change in the distance constraints. (b) Discontinuous flex ambiguity. If edge AD is removed, then reinserted, the graph can flex in the direction of the arrow, taking on a different configuration but exactly preserving all distance constraints.

graph structure that preserve edge lengths but change vertex positions. Rigidity theory distinguishes between *non-rigid* and *rigid* graphs. Non-rigid graphs can be continuously deformed to produce an infinite number of different realizations, while rigid graphs cannot. However, in rigid graphs, there are two types of *discontinuous* deformations that can prevent a realization from being unique [7]:

1. *Flip ambiguities* (Figure 1-4a) occur for a graph in a d-dimensional space when the positions of all neighbors of some vertex span a $(d-1)$-dimensional subspace. In this case, the neighbors create a mirror through which the vertex can be reflected.

2. *Discontinuous flex ambiguities* (Figure 1-4b) occur when the removal of one edge will allow part of the graph to be flexed to a different configuration and the removed edge reinserted with the same length. This type of deformation is distinct from *continuous flex ambiguities* which are present only in non-rigid graphs. In the remainder of this thesis, we use "flex ambiguity" to mean the discontinuous type.

Graph theory suggests ways of testing if a given graph has a unique realization by determining whether or not flip or flex ambiguities are present in a specific graph. However, *neither these specific tests nor the general principles of graph theoretic rigidity apply to the graph realization problem when distance measurements are noisy.* Since realizations of the graph will rarely satisfy the distance constraints exactly, alternative realizations can exist that satisfy the constraints as well as or better than the correct realization, even when the graph is rigid in the graph theoretic sense. In this situation, assuming one can model the measurement noise as a random process, it is desirable to localize only those vertices that

have a small *probability* of being subject to flip or flex ambiguity.

Our algorithm uses robust quadrilaterals as a building block for localization, adding an additional constraint beyond graph rigidity. This constraint permits localization of only those nodes which have a high likelihood of unambiguous realization. We present the algorithm itself in the next chapter, then justify it by deriving the worst-case error likelihood in Chapter 3.

Chapter 2

Approach

We describe an approach for localization of a sensor network in two-dimensional space. We define a node's *neighbors* to be those nodes that have direct bidirectional communications and ranging capability to it. Depending on the type of ranging mechanism used by the network, these two conditions may always be satisfied together. A *cluster* is a node and its set of neighbors.

The algorithm can be broken down into three main phases. The first phase localizes clusters into local coordinate systems. The optional second phase refines the localization of the clusters. The third phase computes coordinate transformations between these local coordinate systems. When all three phases are complete, any local coordinate system can be reconciled into a unique global coordinate system. Alternatively, the transformation between any connected pair of clusters can be computed on-line by chaining the individual cluster transformations as messages are passed through the network. The three phases of the algorithm are as follows:

PHASE I. CLUSTER LOCALIZATION Each node becomes the center of a cluster and estimates the relative location of its neighbors which can be unambiguously localized. We call this process *cluster localization.* For each cluster, one identifies all robust quadrilaterals and finds the largest subgraph composed solely of overlapping robust quads. This subgraph is also a trilateration graph as in [4]; the restriction to robust quads provides an additional constraint that minimizes the probability of realizing a flip ambiguity. Position estimates within the cluster can then be computed incrementally by following the chain of quadrilaterals and trilaterating along the way, as in

Figure 1-2.

PHASE II. CLUSTER OPTIMIZATION **(optional)** Refine the position estimates for each cluster using numerical optimization such as spring relaxation or Newton-Raphson with the full set of measured distance constraints. This phase reduces and redistributes any accumulated error that results from the incremental approach used in the first phase. It can be omitted when maximum efficiency is desired. Note that this optimization imposes no communications overhead since it is performed per cluster and not the network as a whole.

PHASE III. CLUSTER TRANSFORMATION Compute transformations between the local coordinate systems of neighboring clusters by finding the set of nodes in common between two clusters and solving for the rotation, translation, and possible reflection that best aligns the clusters.

This cluster-based approach has the advantage that each node has a local coordinate system with itself as the origin. The algorithm is easily distributed because clusters are localized using only distance measurements to immediate neighbors and between neighbors. Furthermore, if one node in the network moves, only the $\mathcal{O}(1)$ clusters containing that node must update their position information. The following sections describe the phases of the algorithm in more detail.

2.1 Cluster Localization

The goal of cluster localization is to compute the position of a cluster's nodes in a local coordinate system up to a global rotation and possible reflection. Any nodes that are not part of the largest subgraph of robust quads in the cluster will not be localized. However, after Phases I-III are complete, the positions of many of these unlocalized nodes can be computed using more error prone methods that do not rely on robust quads. We do not use such methods in this phase since inaccurate position estimates will be compounded by later phases of the algorithm. Our cluster-based localization strategy is similar to that proposed in [2] except that our use of robust quads specifically avoids flip ambiguities.

Quadrilaterals are relevant to localization because they are the smallest possible subgraph that can be unambiguously localized in isolation. Consider the 4 node subgraph

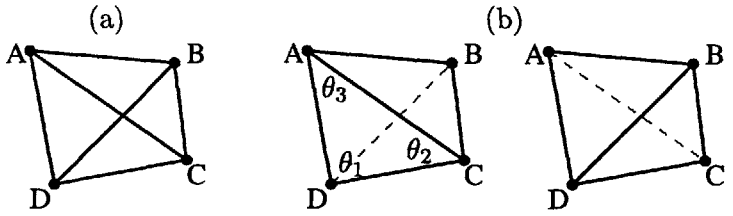


Figure 2-1: (a) The robust four-vertex quadrilateral. The characteristic features of this subgraph are that each vertex is connected to every other by a distance measurement and that knowing the locations of any three vertices is sufficient to compute the location of the fourth using trilateration. (b) Decomposition of the robust quadrilateral into four triangles.

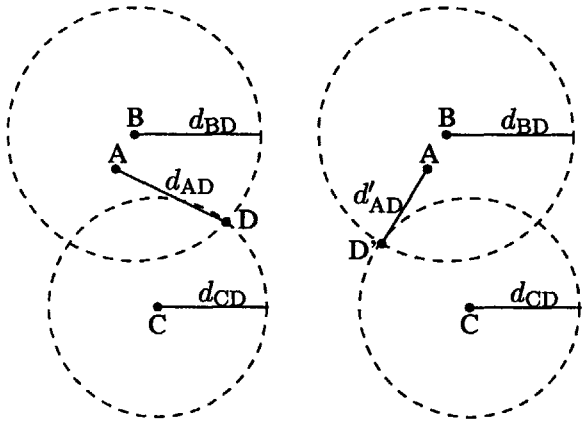


Figure 2-2: An example of a flip ambiguity realized due to measurement noise. Node D is trilaterated from the known positions of nodes A, B, and C. Measured distances d_{BD} and d_{CD} constrain the position of D to the two intersections of the dashed circles. Knowing d_{AD} disambiguates between these two positions for D, but a small error in d_{AD} (shown as d'_{AD}) selects the wrong location for D.

in Figure 2-1, fully-connected by 6 distance measurements. Assuming no three nodes are collinear, these distance constraints give the quadrilateral the following properties:

1. The relative positions of the four nodes are unique up to a global rotation, translation, and reflection. In graph theory terms, the quadrilateral is *globally rigid.*

2. Any two globally rigid quadrilaterals sharing three vertices form a 5-vertex subgraph that is also globally rigid. By induction, any number of quadrilaterals chained in this manner form a globally rigid graph.

Despite these two useful properties of the quadrilateral, global rigidity is not sufficient to guarantee a unique graph realization when distance measurements are noisy. Thus, we further restrict our quadrilateral to be *robust* as follows. The quadrilateral shown in

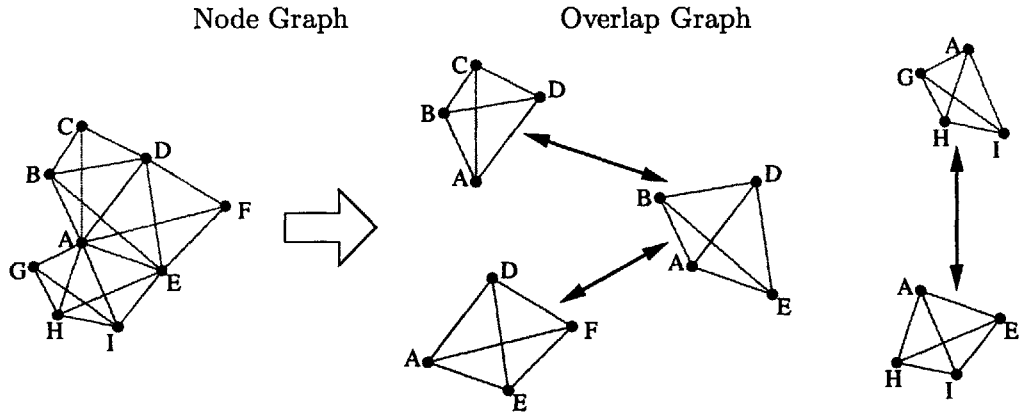


Figure 2-3: The duality between a cluster rooted at A and a graph of robust quads, which we call an *overlap graph*. In the overlap graph, each robust quadrilateral is a vertex. Edges are present between two quads whenever they share three nodes. Thus, if all four node positions are known for some quad, any neighboring quad in the overlap graph can use the three common nodes to trilaterate the position of the unknown node. A breadth-first search into the overlap graph from some starting quad, trilaterating along the way, localizes the cluster as described by Algorithm 2. Note that the overlap graph for a cluster can have distinct, unconnected subgraphs as shown in this example. Nodes that are unique to one subgraph cannot be localized with respect to those of an unconnected subgraph.

Figure 2-1a can be decomposed into four triangles: ΔABC, ΔABD, ΔACD, and ΔBCD, as shown in Figure 2-1b. If the smallest angle θ_i is near zero, there is a risk that measurement error, say in edge AD, will cause vertex D to be reflected over this sliver of a triangle as shown in Figure 2-2. Accordingly, our algorithm identifies only those triangles with a sufficiently large minimum angle as robust. Specifically, we choose a threshold $d_{\min}$ based on the measurement noise and identify those triangles that satisfy

$$b \sin^2 \theta > d_{\min}, \tag{2.1}$$

where b is the length of the shortest side and θ is the smallest angle, as robust. This equation bounds the worst-case probability of a flip error for each triangle. See Chapter 3 for a full derivation. We define a *robust triangle* to be a triangle that satisfies Equation 2.1. Furthermore, we define a *robust quadrilateral* as a fully-connected quadrilateral whose four sub-triangles are robust.

A key feature of our algorithm is that we use the robust quadrilateral as a starting point, and localize additional nodes by chaining together connected robust quads. Whenever two quads have three nodes in common and the first quad is fully localized, we can localize

the second quad by trilaterating from the three known positions. A natural representation of the relationship between robust quads is the *overlap graph*, shown in Figure 2-3. Since three vertices in common make it possible to localize two quads relative to each other, it is natural to represent the space as a graph of robust quads. Localization then amounts to traversing the overlap graph with a breadth-first search and trilaterating as we go, a linear time operation as in [4].

The entire algorithm for Phase I, cluster localization, is as follows:

1. Distance measurements from each one-hop neighbor are broadcast to the origin node so that it has knowledge of the between-neighbor distances.

2. The complete set of robust quadrilaterals in the cluster is computed (Algorithm 1) and the overlap graph is generated.

3. Position estimates are computed for as many nodes as possible via a breadth-first search in the overlap graph (Algorithm 2). At the start of the graph search, we choose positions for the first three nodes to fix the arbitrary translation, rotation, and reflection. We place the origin node at $(0, 0)$ to specify the global translation, the first neighbor on the x-axis to specify the global rotation, and the second neighbor in the positive y direction to specify the global reflection. The remaining nodes are trilaterated as they are encountered.

Appendix A contains an implementation of Algorithms 1 and 2 written in C that was used for the hardware localization experiments presented in Chapter 4. The code combines Algorithms 1 and 2 in such a way that the list of robust quads is generated simultaneously with the breadth-first search so that the list does not need to be stored explicitly. This space-saving optimization is important on board real sensor devices with limited memory.

2.2 Computing Inter-Cluster Transformations

In Phase III, the transformations between coordinate systems of connected clusters are computed from the finished cluster localizations. This transformation is computed by finding the rotation, translation, and possible reflection that bring the nodes of the two local coordinate systems into best coincidence [8]. After Phase I is complete for the two clusters, the positions of each node in each local coordinate system are shared. As long as

Algorithm 1 Finds the set of robust quadrilaterals that contain an origin node i. Each quad is stored as a 4-tuple of its vertices and is returned in the set $Quads_i$. We assume that distance measurements have already been gathered as follows: $Meas_j$ is a set of ordered pairs (k, d_{jk}) that represent the distance from node j to node k. $d_{\min}$ is the robustness threshold, computed from the measurement noise as described in Chapter 3.

```
1: for all pairs (j, d_ij) in Meas_i do
2:   for all pairs (k, d_jk) in Meas_j do
3:     Remove (j, d_kj) from Meas_k
4:     for all pairs (l, d_kl) in Meas_k do
5:       for all pairs (m, d_lm) in Meas_l do
6:         if m ≠ j then
7:           continue
8:         Retrieve (k, d_ik) from Meas_i
9:         Retrieve (l, d_il) from Meas_i
10:        if IsRobust(d_jk, d_kl, d_lj, d_min) AND IsRobust(d_ij, d_ik, d_jk, d_min) AND
           IsRobust(d_ij, d_il, d_lj, d_min) AND IsRobust(d_ik, d_il, d_kl, d_min) then
11:          Add (i, j, k, l) to Quads_i
12:    Remove (k, d_jk) from Meas_j
```

Algorithm 2 Computes position estimates for the cluster centered at node i. This algorithm does a breadth-first search into each disconnected subgraph of the overlap graph created from $Quads_i$ and finds the most complete localization possible. At the end of this algorithm, $Locs_{best}$ is a set containing pairs $(j, \mathbf{p})$ where $\mathbf{p}$ is the estimate for the x-y position of node j. Any neighbors of i not present in $Locs_{best}$ were not localizable.

```
1: Locs_best := ∅
2: for each disconnected subgraph of the overlap graph do
3:   Locs:= ∅
4:   Choose a quad from the overlap graph.
5:   p_0 := (0, 0) {Position of the origin node}
6:   p_1 := (d_ab, 0) {First neighbor sets x-axis}
7:   α := (d_ab^2 + d_ac^2 − d_bc^2) / (2 d_ab d_ac)
8:   p_2 := (d_ac α, d_ac √(1 − α^2)) {Localize the second neighbor relative to the first}
9:   Add (a, p_0), (b, p_1), and (c, p_2) to Locs
10:  for each vertex visited in a breadth-first search into the overlap graph do
11:    if the current quadrilateral contains a node j that has not been localized yet then
12:      Let p_a, p_b, p_c be the x-y positions of the three previously localized nodes.
13:      p' :=Trilaterate(p_a, d_aj, p_b, d_bj, p_c, d_cj)
14:      Add (j, p') to Locs
15:  if length(Locs) >length(Locs_best) then
16:    Locs_best :=Locs
```

there are at least three non-collinear nodes in common between the two localizations, the transformation can be computed. By testing if these three nodes form a robust triangle, we simultaneously guarantee non-collinearity and the same resistance to flip ambiguities as Phase I of the algorithm.

2.3 Node Mobility

A key goal that motivates the design of our localization algorithm is the ability to handle mobile nodes. The approach we have described is intrinsically static, that is, it assumes distance measurements are constant in time. However, with moving nodes, if we can guarantee that the set of measurements was taken at a single instant, the algorithm still produces the correct localization. If the measurements were taken at different times, the localization result will be inaccurate because the graph realization they describe never existed at any time. Thus, for accurate localization under mobility, each execution of the algorithm *must have all measurements to or from a mobile node measured at a single instant.*

Another issue with mobility is that measurement noise and outliers can have a more severe effect on localization accuracy than in the static case. For example, it is difficult to detect whether changes in the measurement over time are due to the movement of the node or measurement noise. Spurious outliers are equally difficult to disambiguate. Our solution to this problem is to filter the measurements with a Kalman filter, which attempts to probabilistically differentiate between noise and motion by knowing the characteristics of each.

2.3.1 Kalman Filtering

The measurement Kalman filter has state $s_i = \begin{bmatrix} d_i & v_i \end{bmatrix}^\top$ at time i where d_i is the node-to-node distance and v_i is the rate of change of that distance. There is one such filter for each pair of nodes. The basic assumption behind the Kalman filter is that, over time, the state varies according to

$$s_{i+1} = H_i s_i + n_i \tag{2.2}$$

where H_i is some matrix that expresses the time-dependence of the state and n_i is noise. Similarly, the dependence of measurements on the state can be described by

$$x_i = F_i s_i + \eta_i \tag{2.3}$$

where x_i is the measured value, F_i expresses the dependence of the measurement on the state, and η_i is noise. Since our state consists of distance and its rate of change, and measurements estimate the distance, these matrices are

$$H_i = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{2.4}$$

$$F_i = \begin{bmatrix} 1 & 0 \end{bmatrix}. \tag{2.5}$$

The filtering process consists of three steps: *prediction*, *gain*, and *update*, which are executed each time a measurement is made. These three steps are as follows to transition from state $i-1$ to state i:

Prediction:

$$\hat{s}_{i|i-1} = H_{i-1}\hat{s}_{i-1} \tag{2.6}$$

$$P_{i|i-1} = H_{i-1}P_{i-1}H_{i-1}^{\top} + Q_{i-1} \tag{2.7}$$

Gain:

$$K_i = P_{i|i-1}F_i^{\top}\left(F_i P_{i|i-1} F_i^{\top} + \Lambda_i\right)^{-1} \tag{2.8}$$

Update:

$$\hat{s}_i = \hat{s}_{i|i-1} + K_i\left(x_i - F_i\hat{s}_{i|i-1}\right) \tag{2.9}$$

$$P_i = \left(I - K_i F_i\right) P_{i|i-1} \tag{2.10}$$

where $\hat{s}_i$ is the current estimate of s_i, P_i is the covariance matrix of $\hat{s}_i$, and x_i is the new measurement. Matrix Q_i is the *process noise*, describing how the state can vary over time, and Λ_i is the *measurement noise*, describing the expected noise of each distance

measurement. Mathematically, $Q_i = E[n_i n_i^\top]$ and $\Lambda_i = E[\eta_i \eta_i^\top]$. For our case, we will assume

$$Q_i = \begin{bmatrix} \Delta t^2 & \Delta t \\ \Delta t & 1 \end{bmatrix} \sigma_v^2 \tag{2.11}$$

where σ_v^2 is the variance of distance's rate of change and we will assume Λ_i is equal to the variance of the underlying measurement system, determined empirically. The value of σ_v depends on the type of distance we are estimating. Between two nodes known to be stationary, σ_v is set to nearly zero, allowing little to no motion. Furthermore, we restrict the $\hat{v}_i$ portion of the state estimate to be zero. These restrictions express our knowledge that the distance estimate is not expected to change over time.

For distance measurements between a stationary node and a mobile node, or between two mobile nodes, we use a much larger value of σ_v that expresses the large uncertainly we have in how the distance estimate changes over time. The precise value depends on the speed and unpredictability of the mechanism that is moving the node. For the experiments in Chapter 4 the value of Λ_i is 2.0 cm and the value of σ_v is 20 cm/s for moving nodes and 0.33 cm/s for stationary nodes.

Phase I, cluster localization, of the algorithm is performed using the distance estimates $\hat{d}_i$ computed by this Kalman filter, not the distance measurements x_i directly. Appendix B contains C code that performs this filtering.

Other approaches to object tracking in a sensor network have been proposed in which an Extended Kalman Filter is used to simultaneously filter noise and perform tracking, by treating the node's Euclidean position as the filter state [22]. Although this is a promising approach, we do not consider it here because the amount of state and computation required becomes infeasible on board low-power sensor networks.

2.3.2 Outlier Rejection

In our experiments presented in Chapter 4, time difference of arrival between ultrasound and radio signals is the underlying distance measurement technology. These measurements are subject to occasional outliers that do not fit the Gaussian noise assumption intrinsic to the Kalman filter. We attempt to detect and eliminate such outliers using the Mahalanobis distance computed from the current filter state and an incoming measurement. We define e_i as the difference between the predicted measurement and the actual measurement at time

step i. Thus, $e_i = x_i - F_i \hat{s}_{i|i-1}$. The Mahalanobis distance is

$$M = e_i^\top \left(F_i P_{i|i-1} F_i^\top + \Lambda_i \right) e_i. \tag{2.12}$$

For any new measurement, M is computed and thresholded. If it is above the threshold, the measurement is rejected as an outlier and the filter state is not updated. Otherwise, the measurement is incorporated as usual. A common threshold for M is 3.0, which considers any measurement outside the 1% likelihood of Gaussian noise to be an outlier.

In some situations, a series of measurements are detected as outliers even though they are correct. This can occur due to an unexpected movement of the nodes or a change in the environment. Thus, if we detect more than three sequential outliers, the Kalman filter is reinitialized from scratch. This reset allows the network to adapt to unexpected changes in node configuration without requiring manual intervention.

The code in Appendix B performs the outlier rejection and filter reinitialization as described here.

2.3.3 Optimization

Localization using trilateration alone can be inaccurate since it uses exactly three distance constraints when in fact many more constraints may be available. Using the maximum number of constraints is especially important in the presence of mobility due to its extra noise. Thus, we have found it important to apply Phase II, the optimization step, to introduce these extra constraints after Phase I computes an initial estimate using trilateration.

Applying nonlinear optimization to the entire cluster can be too computationally expensive on low-power sensor nodes. In this situation, good results can be obtained by optimizing only the positions of mobile nodes, keeping the results of Phase I for the static nodes. This reduction in the dimensionality of the optimization problem significantly reduces the amount of computation required.

Chapter 3

Analysis

3.1 Proof of Robustness

In order for Algorithm 2 to produce a correct graph realization, we must ensure that our use of robust quads prevents both flip and flex ambiguities. Since distance measurements may have arbitrary noise we cannot guarantee a correct realization in all cases — instead we can only predict the probability of having no flips based on our definition of robustness. It is difficult to quantify this probability for an entire graph, so instead we focus on the probability of an individual error. That is, we define an "error" as the realization of a single robust quad with one vertex flipped or flexed from its correct location. By deriving the worst-case probability of error, we will prove our first theorem:

Theorem 1 *For normally-distributed distance measurement noise with standard deviation σ, we can construct a robustness test such that the worst-case probability of error is bounded.*

First, we prove that the use of robust quadrilaterals rules out the possibility of flex ambiguities as seen in Figure 1-4b. This kind of flex ambiguity occurs only when a rigid graph becomes non-rigid by the removal of a single edge [7]. If the graph is such that no single edge removal will make it non-rigid, the graph is *redundantly rigid*, and no flex ambiguities are possible. The robust quad has six edges. By removing any edge, we are left with a 5-edged graph, which must be rigid according to the following theorem [10]:

Theorem 2 (Laman's Theorem) *Let a graph G have exactly $2n-3$ edges where n is the number of vertices. G is generically rigid in $\mathbb{R}^2$ if and only if every subgraph G' with n'*

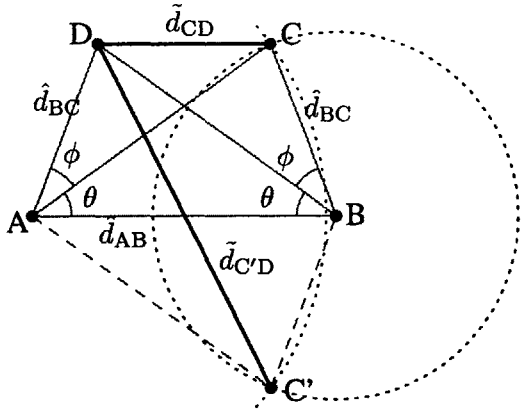


Figure 3-1: A diagram of a quadrilateral for deriving the worst-case probability of flip error. Vertex C is being trilaterated from the known positions of vertices A, B, and D. Its distance to vertex D is used to disambiguate between the two possible locations C and C′ by testing which of $\tilde{d}_{CD}$ and $\tilde{d}_{C'D}$ is closer to the measured distance between C and D.

vertices has $2n' - 3$ *or fewer edges.*

Our robust quad with its missing edge has 4 vertices and 5 edges, satisfying the condition in Laman's Theorem. Since every 3-vertex subgraph has 3 or fewer edges and every 2-vertex subgraph has 1 or fewer edges, the 5-edged quad is rigid. Thus, the 6-edged robust quad is redundantly rigid. Therefore, flex ambiguities are impossible for a graph constructed of robust quads.

Unlike flex ambiguities, flips cannot be ruled out based on the graph structure alone. Since distance measurements are noisy, they may cause a vertex to be incorrectly flipped in a computed realization. Thus, we derive the worst-case probability of realizing a flipped vertex. Figure 3-1 depicts the scenario in which a vertex could become incorrectly flipped. In this example, vertex C is being trilaterated with respect to the known positions of vertices A, B, and D. Temporarily ignoring vertex D, we can pinpoint C to two possible locations: C and C′, the intersection points of two circles centered at A and B, of radius $\hat{d}_{AC}$ and $\hat{d}_{BC}$. To disambiguate this possible flip, we use the known distance to vertex D as follows. We compute distances $\tilde{d}_{CD}$ and $\tilde{d}_{C'D}$. Whichever distance is closer to the measured distance $\hat{d}_{CD}$ will determine whether C or C′ is selected during trilateration.

The probability of an incorrect flip is equal to the probability that the measured distance $\hat{d}_{CD}$ will be closer to the incorrect distance $\tilde{d}_{C'D}$ than to the correct distance $\tilde{d}_{CD}$. Note that the problem has an intrinsic symmetry: namely, disambiguating the position of C based

on D is equivalent to disambiguating D based on C. Assuming the random measurement noise is zero-mean, there must be a measurement error of magnitude $\geq \frac{1}{2}(\tilde{d}_{C'D} - \tilde{d}_{CD})$ for an incorrect flip to be realized. We can derive this value from the graph in Figure 3-1. For simplicity, we constrain the figure to be left-right symmetric, although the probability of error will only decrease by breaking this symmetry. In this problem, we take the values of $\hat{d}_{AB}$, θ, and ϕ as given. We will later eliminate ϕ by maximizing the error with respect to it. First, we compute the values of $\tilde{d}_{CD}$ and $\tilde{d}_{C'D}$ as: $\tilde{d}_{CD} = \frac{\hat{d}_{AB}\sin\phi}{\sin(2\theta+\phi)}$ and $\tilde{d}_{C'D} = \frac{\hat{d}_{AB}}{\sin(2\theta+\phi)}\sqrt{\sin^2\phi + 4\sin^2(\theta+\phi)\sin^2\theta}$. Combining these yields

$$d_{err} = \frac{\tilde{d}_{C'D} - \tilde{d}_{CD}}{2} \tag{3.1}$$

$$= \hat{d}_{AB}\frac{\sqrt{\sin^2\phi + 4\sin^2(\theta+\phi)\sin^2\theta} - \sin\phi}{2\sin(2\theta+\phi)}. \tag{3.2}$$

Since we are interested in the worst-case probability of error, we minimize d_{err} with respect to ϕ by taking the partial derivative of d_{err} and setting it equal to zero. We find that d_{err} is minimized when $\phi = \frac{\pi}{2} - 2\theta$. This can be substituted into Equation 3.2 and the resulting equation simplified to yield

$$d_{err} = \hat{d}_{AB}\sin^2\theta. \tag{3.3}$$

Thus, if the true distance is d and the measured distance is a random variable X, then the worst-case probability of error is $P(X > d + d_{err})$. If measurement noise is zero-mean Gaussian with standard deviation σ, the worst-case probability of error is

$$P(X > d + d_{err}) = \Phi\left(\frac{d_{err}}{\sigma}\right) \tag{3.4}$$

where $\Phi(x)$ connotes the integration of the unit normal probability density function from x to infinity. This equation tells us that for arbitrary measurement noise with standard deviation σ, we can choose a threshold $d_{\min}$ for the robustness test. Only those triangles for which $b\sin^2\theta > d_{\min}$, where b is the shortest side and θ is the smallest angle, will be treated as robust. By choosing $d_{\min}$ to be some constant multiple of σ, we bound the probability of error. This proves Theorem 1.

For the simulation results presented in this thesis, $d_{\min}$ was chosen to be 3σ. For Gaussian noise, this bounds the probability of error for a given robust quadrilateral to be

less than 1%. However, for the typical case, the probability is significantly less than 1%, thus posing minimal threat to the stability of the localization algorithm.

3.2 Computational Complexity

It is important that any distributed localization algorithm be scalable to large networks. In this section we discuss the computational and communications efficiency of the algorithm presented in Chapter 2. In general, finding a realization of a graph is NP-hard [20]. We are able to do it in polynomial time because our algorithm purposefully avoids nodes that may have position ambiguities (i.e., flips or flexes) at the cost of failing to find all possible realizations. It is these ambiguities which cause the general case to blow up combinatorially. Our algorithm grows linearly with respect to the number of nodes when there are $\mathcal{O}(1)$ neighbors per node. Furthermore, since this computation is distributed across the network, each node performs $\mathcal{O}(1)$ computation. If the node degree is not constant, each node's computation varies with the third power of the number of neighbors.

Algorithm 1, which finds the set of robust quadrilaterals in a local cluster, has worst-case runtime $\mathcal{O}(m^4)$ where m is the maximum node degree. It can be implemented with $\mathcal{O}(m^3)$ runtime using better data structures, as is done with the implementation in Appendix A. In practice, the algorithm is much more efficient because each neighbor is generally not connected to every other neighbor. In this algorithm, we simply enumerate the robust quadrilaterals in the cluster, thus the worst-cast number of robust quadrilaterals is $\binom{m}{3}$, which is $\mathcal{O}(1)$ for a graph of bounded degree.

Algorithm 2, which solves for position estimates for one cluster, has runtime $\mathcal{O}(q)$ where q is the number of robust quadrilaterals. In the worst case, $q = \binom{m}{3}$.

Finding the inter-cluster transformations for one cluster has runtime $\mathcal{O}(m^2)$. We are finding m transformations, each of which may take $\mathcal{O}(m)$ time to compute because the registration problem takes linear time in the number of overlapping vertices. Again, for a graph of bounded degree, these computations take $\mathcal{O}(1)$ time.

The only stage of the algorithm that entails communication overhead is the initial step where each node shares its measured distances with its neighbors. If we assume that non-overlapping clusters do not share the same channel (due to range limitations), the communications overhead is $\mathcal{O}(m^2)$ because m^2 measurements are being shared. In practice, this

is implemented by each node sending one packet of constant size for distance measurement and one packet of $\mathcal{O}(m)$ size to share other measurements.

Chapter 4

Experimental Results

In order to measure the effectiveness of our algorithm on real sensor networks, we implemented it on-board a functioning sensor network. The network is constructed of Crickets (Figure 4-1) a hardware platform developed and supplied by MIT [16]. Crickets are hardware-compatible with the Mica2 Motes developed at Berkeley with the addition of an Ultrasonic transmitter and receiver on each device. This hardware enables the sensor nodes to measure inter-node ranges using the time difference of arrival (TDoA) between Ultrasonic and RF signals. Although the Crickets can achieve ranging precision of around 1 cm on the lab bench, in practice, the ranging error can be as large as 5 cm due to off-axis alignment of the sending and receiving transducers.

4.1 Evaluation Criteria

One criteria by which we evaluate the performance of the algorithm is how the computed localization differs from known ground truth. This error is expressed as

$$\sigma_p^2 = \sum_{i=1}^{N} \frac{(\hat{x}_i - x_i)^2 + (\hat{y}_i - y_i)^2}{N} \tag{4.1}$$

where N is the number of nodes, $\hat{x}_i$ and $\hat{y}_i$ compose the localized position of node i, and x_i and y_i compose the true position of node i. This metric is simply the mean-square error in Euclidean 2D space.

It is useful to compare σ_p^2 to the mean-square error in the raw distance measurements, since the error model of the measurements determines the minimum achievable σ_p of an

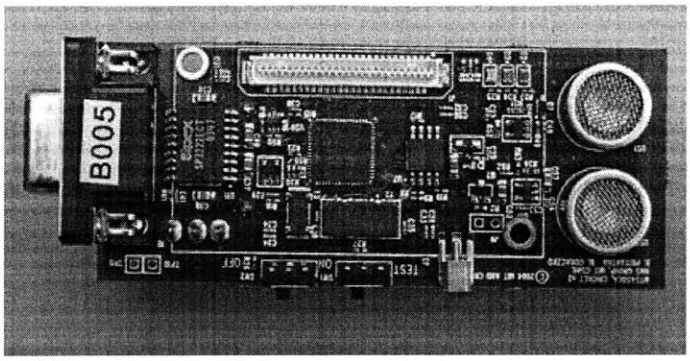


Figure 4-1: A photograph of a Cricket device. The hardware is compatible with a Berkeley Mica2 Mote with the addition of an ultrasonic transmitter and receiver, the silver components on the right. The Cricket estimates distance by measuring the time difference of arrival between an ultrasonic chirp which travels at the speed of sound, and a radio packet which travels at the speed of light.

ideal localization algorithm [18]. The mean-square error of the distance measurements is

$$\sigma_d^2 = \sum_{i=1}^{M} \frac{(\hat{d}_i - d_i)^2}{M} \tag{4.2}$$

where M is the number of inter-node distances, $\hat{d}_i$ is the measured value of distance i, and d_i is the true value of distance i.

Another useful metric is the proportion of nodes successfully localized by the algorithm. Let L_i be the number of nodes successfully localized in the cluster centered at node i, and k_i be the total number of nodes in this cluster. In one cluster, the proportion of nodes localized is L_i/k_i. For the entire network, we define the *cluster success rate* as

$$\bar{R} = \frac{1}{N} \sum_{i}^{N} \frac{L_i}{k_i}. \tag{4.3}$$

This metric tells us the average percentage of nodes that were localizable per cluster.

Our final metric conveys the proportion of nodes in the *entire* network that could be localized into a single coordinate system. Since some clusters will have transformations between them and others will not, the network may split into separate subgraphs, each of which is localized with respect to all its nodes, but is not rigidly localized with respect to the other subgraphs. We call these subgraphs *forests*. Naturally, it is desirable for there to be only a single forest that contains every node in the network. Thus, another useful metric

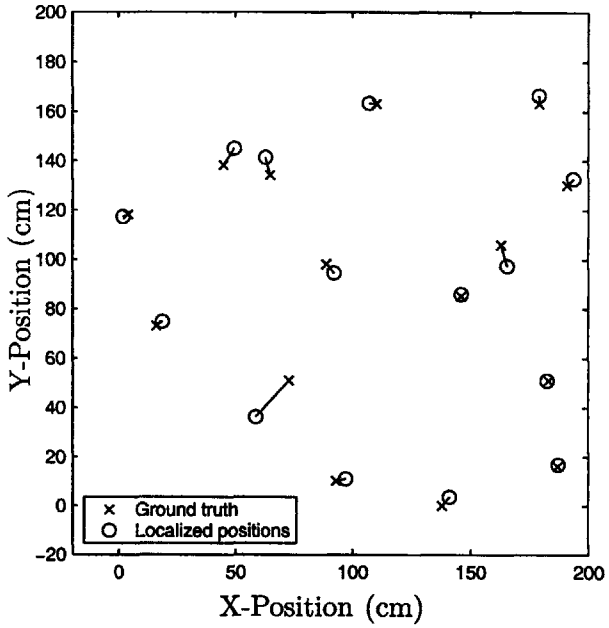


Figure 4-2: A comparison of node positions as localized by our algorithm to the true positions of the nodes on a physical Cricket cluster. Positions are computed by Phase I of the algorithm, cluster localization. The experiment involved 16 nodes, one of which could not localize; thus only 15 are shown.

is the *largest forest size*, which is the number of nodes in the largest forest. This metric can be expressed as a percentage $\tilde{R}$ by dividing by the total number of nodes in the network.

4.2 Accuracy Study: Hardware Deployment

Figure 4-2 shows the results of the first experiment. In this experiment, 16 crickets were placed in a pseudo-random, 2-dimensional arrangement. Ground truth was measured manually with the aid of a grid on the surface. The small circles depict the positions of each node as computed by the localization algorithm running on-board the cricket in the bottom-most position of the figure. The positions shown are for Phase I of the algorithm, where positions are trilaterated using robust quadrilaterals. No least-squares optimization was performed. The true position of each node is shown with an "x." A line between the two points shows the amount of positioning error.

The error metrics for this experiment are shown in Table 4.1. The fact that σ_p (the localization error) is only slightly larger than σ_d (the measurement error) tells us that our

metric	value
σ_d	5.18 cm
σ_p	7.02 cm
$\bar{R}$	$15/16 = 0.94$
$\tilde{R}$	$15/16 = 0.94$

Table 4.1: Error metrics for the localization results of Figure 4-2.

metric	value
σ_d	4.38 cm
σ_p	6.82 cm
$\bar{R}$	0.97
$\tilde{R}$	$38/40 = 0.95$

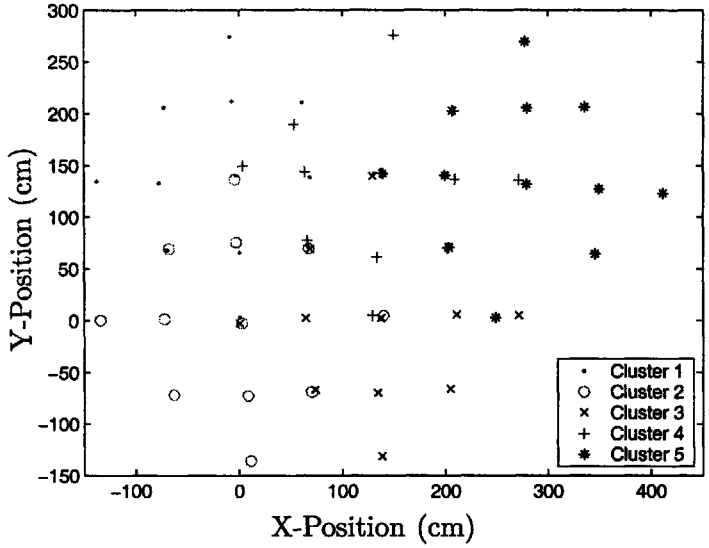
Table 4.2: Error metrics for the localization results of Figure 4-3.

algorithm performed well relative to the quality of the distance measurements available. In addition, all nodes but one were successfully localized, indicating that the algorithm provided good localization coverage of the cluster.

A second experiment, the results of which are shown in Figure 4-3, demonstrates both Phase I and Phase III of the algorithm. Once again, for simplicity, the optional least-squares relaxation phase is omitted. A total of 40 nodes were placed in a 5×6 meter region. The RF and ultrasound ranges of each Cricket were arbitrarily restricted so that only 12 neighbors were rangeable from each node. Then, cluster localization was performed separately on five nodes, each being the origin of its own cluster as shown in Figure 4-3a. Phase I of the algorithm, running on each of the five clusters, localized its nodes in a local coordinate system. Transformations between each pair of coordinate systems with at least three nodes in common were computed by Phase III of the algorithm. Figure 4-3b shows the localized positions of each node as small circles, overlaid with the ground truth. The localized positions of three nodes are used to bring the entire network into registration with the global coordinate system used by ground truth.

The error metrics for the Figure 4-3 experiment are shown in Table 4.2. As in the first experiment, these results show that localization error was not much greater than measurement error.

(a) The five clusters present in the network



(b) Localized network versus ground truth

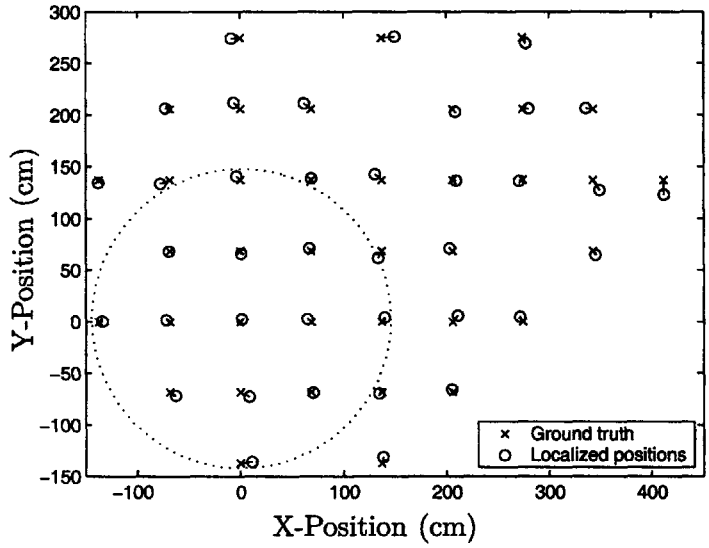


Figure 4-3: (a) The separate clusters that combined to form the complete network localization. (b) The localized positions of 40 Crickets in a physical network. The two "holes" in the network are where two nodes could not localize, and thus only 38 are shown. The coordinate transformations between each cluster were computed and used to render the localized positions in the single coordinate system seen here. Ground truth positions are overlaid, with lines showing the amount of error for each node. The dotted line depicts the maximum communications radius.

4.3 Scalability Study: Simulated Deployment

We have tested the three phases of our algorithm on a variety of simulated networks in order to evaluate its scalability beyond the physical experiments performed in Section 4.2. In this thesis, we present simulation results in two network environments: a square region without obstructions, shown in Figure 4-4a, and an environment based on the actual floorplan of an office building, shown in Figure 4-5a. The square environment places 100 nodes uniformly and randomly in a two-dimensional region, where connectivity is only available between nodes that are within a maximum ranging distance. The building environment randomly places 183 nodes with the additional requirement that nodes obstructed by walls cannot range to each other. This exercises the algorithm's ability to deal with obstructions. The floorplan has three rooms and one hallway, and is approximately square with each side 10 m long.

When evaluating the algorithm's performance, we are interested in how both node degree and measurement noise affect the results. Node degree was varied by changing the maximum ranging distance. We also consider three different degrees of measurement noise:

1. Zero noise, where all measurements are exact. Simulations without noise give an upper bound on how much localization is possible for a network. Without noise, any unlocalizable nodes must be due to disconnection or non-rigidity in the graph structure.

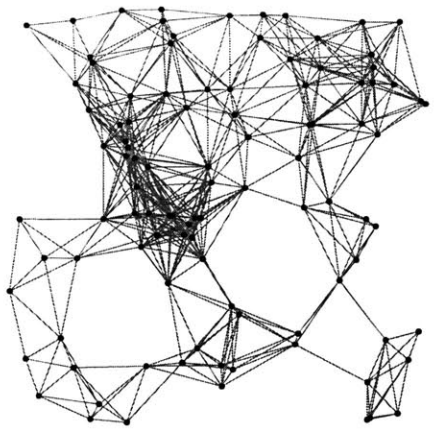
2. Noise with $\sigma_d = 1$ cm, similar to that of a Cricket device in ideal circumstances.

3. Noise with $\sigma_d = 10$ cm. This figure is to designed to simulate sensor networks with more imprecise ranging capability.

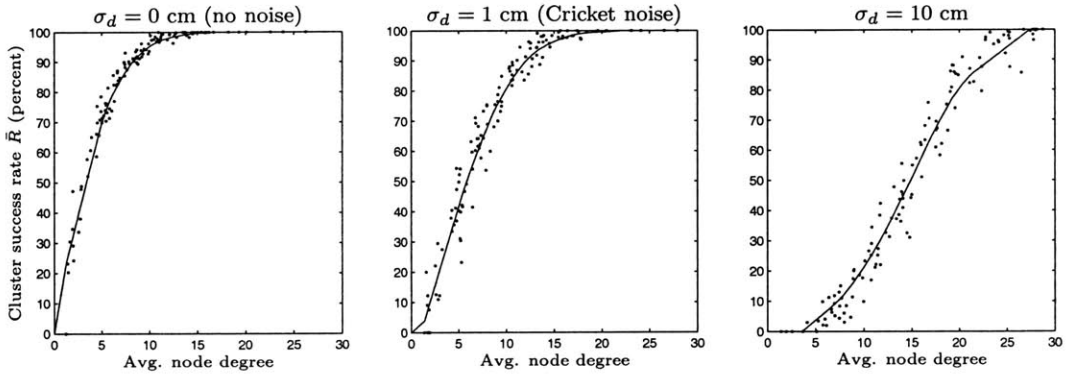
Figure 4-4 shows the simulation results for the square environment. Each data point on the plots represent a single run of the simulation, which localizes as many nodes as possible. As one would expect, the ability of the algorithm to localize goes down as the measurement noise increases. Interestingly, the algorithm is nearly as effective with $\sigma_d = 1$ cm noise as with zero noise. With more noise, the algorithm is still effective, but the requirements for node degree are higher.

Figure 4-5 shows the simulation results for the building environment. Note that the largest forest size $\tilde{R}$ rarely obtained 100% even with high node degree due to obstruction

(a)



(b) PLOTS OF CLUSTER SUCCESS RATE, $\bar{R}$, VERSUS NODE DEGREE



(c) PLOTS OF LARGEST FOREST SIZE, $\tilde{R}$, VERSUS NODE DEGREE

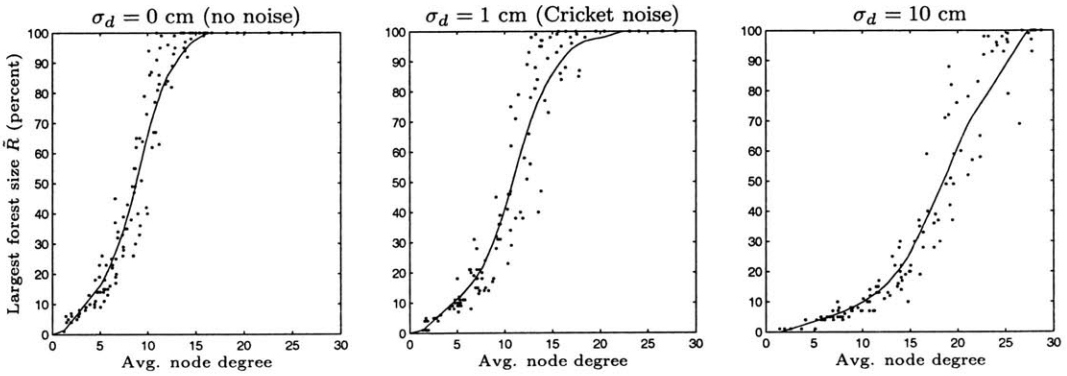
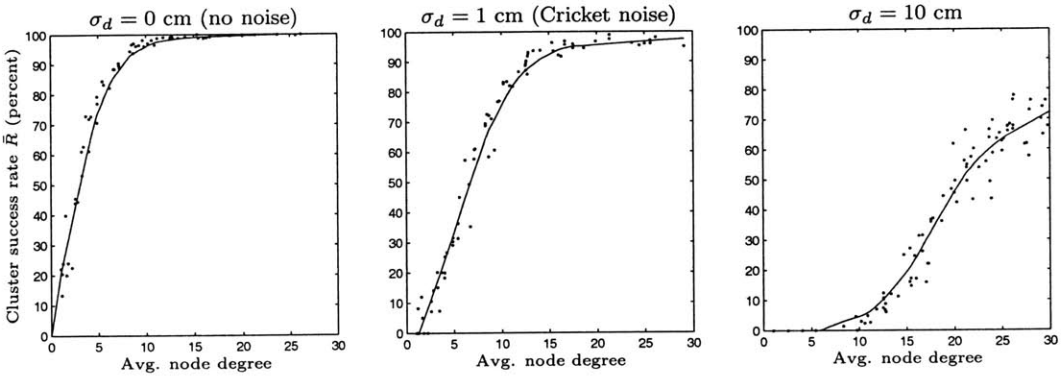


Figure 4-4: (a) One instance of the simulated sensor network. Each graph edge represents a distance measurement that a node can perform. Different runs in the results used the same node positions with varying maximum measurement radius. (b) The cluster success rate $\bar{R}$ versus the average node degree for three different levels of measurement noise. Each data point shows the value of these quantities for a single simulation run. A moving average of the data points is overlaid on each plot. (c) The size of the largest forest $\tilde{R}$ versus average node degree for three different levels of measurement noise.

(a)

(b) PLOTS OF CLUSTER SUCCESS RATE, $\bar{R}$, VERSUS NODE DEGREE



(c) PLOTS OF LARGEST FOREST SIZE, $\tilde{R}$, VERSUS NODE DEGREE

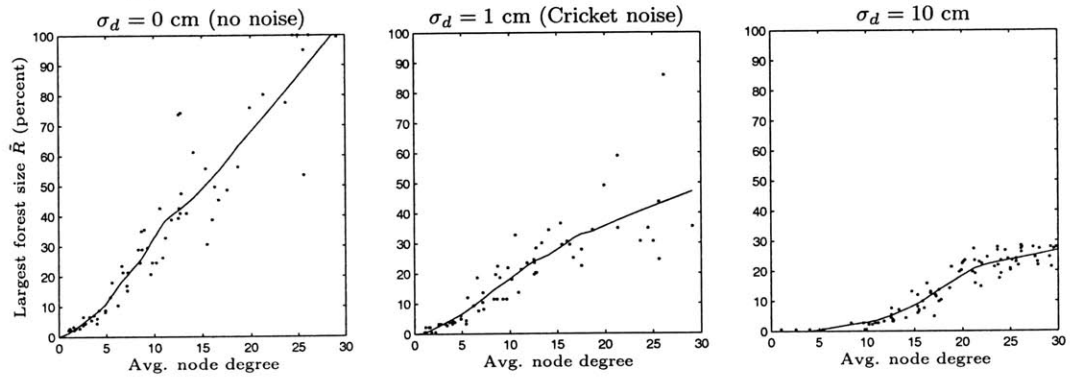


Figure 4-5: (a) The office floorplan used for sensor network simulation. Dark lines are the walls of the building and light-colored lines represent the graph edges between nodes. Each edge represents a distance measurement that a node can perform. Measurements cannot be taken through walls. (b) The cluster success rate $\bar{R}$ versus the average node degree for three different levels of measurement noise. Each data point shows the value of these quantities for a single simulation run. A moving average of the data points is overlaid on each plot. (c) The size of the largest forest $\tilde{R}$ versus average node degree for three different levels of measurement noise.

(a) Our algorithm (b) Algorithm without robust quads

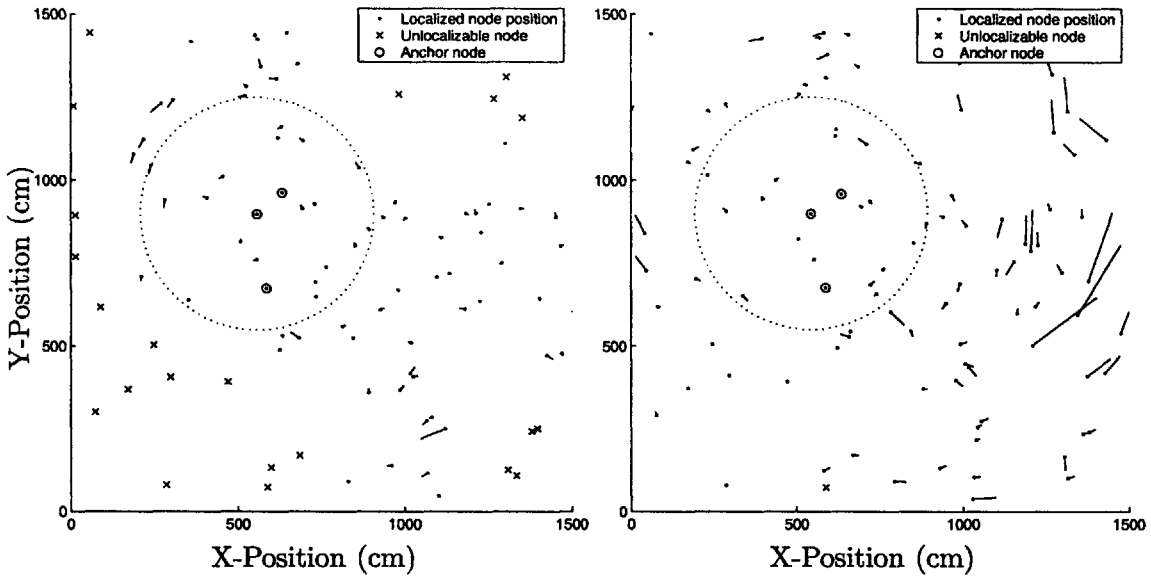


Figure 4-6: (a) Our algorithm's localized positions for a simulated network compared to ground truth. Lines show the amount of error for each node's position. The three nodes used to compute the transformation to the ground truth's coordinate system are shown with small circles. The large dotted circle depicts the maximum ranging distance of a node. (b) Localization of the same network using basic trilateration without checking for quad robustness.

by walls. In a practical deployment, nodes would have to be strategically placed around doorways to achieve 100% forest size.

4.4 Error Propagation

Cluster-based localization algorithms generally suffer from poor error propagation characteristics because they have no absolute reference points as constraints. We show that our approach, using robust quads, significantly reduces the amount of error propagated over approaches based on basic trilateration.

Figure 4-6a shows localization results of our algorithm after Phase I and III on a simulated network of 100 nodes. Nodes were randomly placed within the square region, each with a maximum ranging distance of 350 cm. Distance measurements were corrupted by Gaussian noise with $\sigma_d = 5.0$ cm. In order to compare to ground truth, we pick three nodes as "anchors". These nodes are used solely for transforming between the separate coordinate

metric	Our algorithm			w/o robust quads
σ_d	**1.0 cm**	**3.0 cm**	**5.0 cm**	**5.0 cm**
σ_p	4.43 cm	14.39 cm	16.22 cm	54.87 cm
$\bar{R}$	0.91	0.85	0.79	0.95
$\tilde{R}$	0.93	0.87	0.75	0.99
Shown in:			Figure 4-6a	Figure 4-6b

Table 4.3: Error metrics of four simulation runs of the network in Figure 4-6.

systems of the algorithm and ground truth, and are not used by the algorithm at run-time. The anchor nodes are closely-spaced so that errors can accumulate towards the edges of the network. In contrast, Figure 4-6b shows localization results for the same network, but with an algorithm that uses trilateration alone and does not check for quad robustness.

Table 4.3 contains the error metrics of four simulation runs for the network in Figure 4-6. Each was run with a different amount of measurement noise, σ_d. The error metrics for the simulation without robust quads are also shown. This comparison demonstrates that robust quads significantly reduce error propagation.

4.5 Localization of Mobile Nodes

An advantage of our algorithm is that it handles node mobility well because each cluster localization can be recomputed quickly. Even on a low power device, the cluster localization phase can take less than one second for 15–20 neighbors. Thus, as nodes move, Phase I can simply be repeated to keep up. Furthermore, by excluding mobile nodes from the transformation computation in Phase III, it does not need to be repeated.

In practice, recomputing the cluster localization repeatedly does not produce good results due to additional measurement noise introduced by motion. This degradation is especially apparent with the Cricket system because motion in the environment often introduces extra sound which can interfere with acoustic ranging. In Section 2.3 we introduced several techniques to deal with this extra noise. All three techniques: Kalman Filtering, outlier rejection, and non-linear optimization were used in our experiments with mobility.

One of the requirements stressed in Section 2.3 was that distances measured to a mobile node must be taken at a single instant. In our experiments with the Cricket platform, we

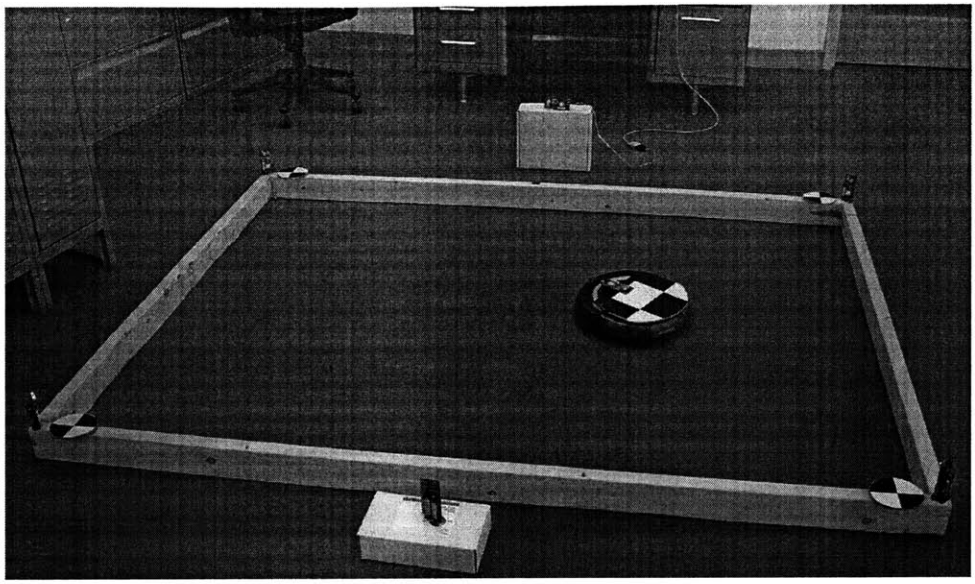
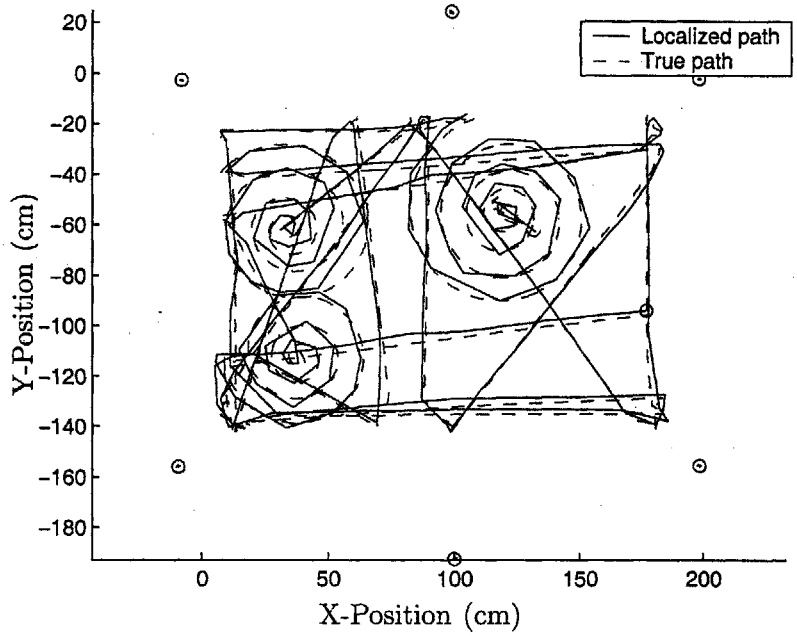


Figure 4-7: The experimental setup for mobile robot localization, consisting of six stationary nodes and one mobile node.

achieve this by measuring the distance to moving nodes using only ultrasound pulses generated by the moving node itself. This way, all other nodes will sense the same physical pulse and generate a measurement estimate for the same moment in time. These measurements are then shared within each cluster.

Figure 4-7 shows our experimental setup for localizing a mobile node. Six stationary nodes were deployed in a roughly circular configuration around the sides of a wooden frame. A node was attached to an autonomous robot placed in the center of the frame. Once activated, the robot randomly traversed the rectangular space. The localization as computed by the sensor network was logged over time and manually synchronized with a calibrated video camera. The video was post-processed to obtain the ground truth robot path with sub-centimeter accuracy. This path is compared to the path computed by the localization algorithm in Figure 4-8. The localization algorithm computed a position estimate for the robot roughly once per second for 3 minutes. Since discrete computations were made, each of these separate localizations could be compared to ground truth. The mean-square error, σ_p, computed from these values is 2.59 cm. Thus, our localization algorithm is shown to be successful at localizing networks with mobile nodes.

(a) Localized path of mobile node



(b) Magnitude of localization error vs. time

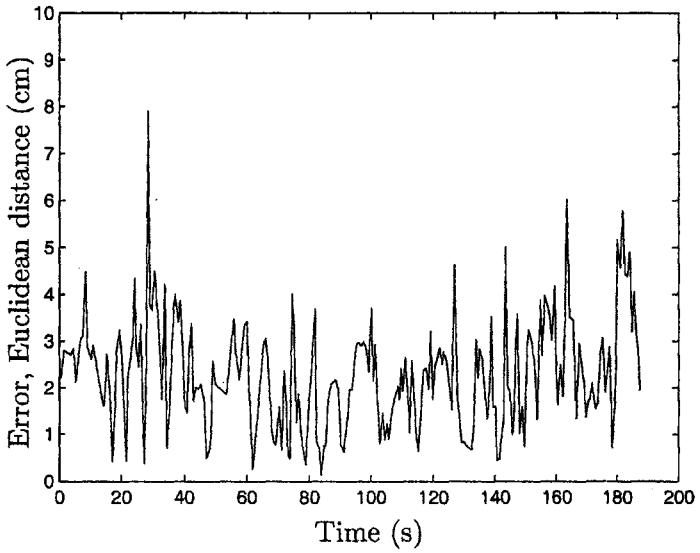


Figure 4-8: (a) The path of a mobile node computed by our localization algorithm compared to ground truth over a 3 minute period. A sensor node was attached to a mobile robot (an autonomous floor vacuum) that randomly covered a rectangular space. Six static nodes, depicted as circles, were used to localize this mobile node over time. Ground truth (dashed) was obtained from calibrated video. (b) The Euclidean distance between the mobile node's localized position and ground truth over time.

Chapter 5

Conclusion

We have demonstrated an algorithm that successfully localizes nodes in a sensor network with noisy distance measurements, using no beacons or anchors. Simulations and experiments showed the relationship between measurement noise and ability of a network to localize itself. As long as the error model of the measurement noise is known, the algorithm copes with this noise by refusing to localize those nodes which have ambiguous positions. Furthermore, even with no noise, each node in the network must have approximately degree 10 or more before 100% node localization can be attained. As noise increases, so will the connectivity requirements. The Cricket platform has a moderate amount of noise and thus exercises our algorithm's tolerance for noisy distance measurements. We have also shown that the algorithm adapts to node mobility by filtering the underlying measurements.

For future work, we are interested in extending our physical experiments to even larger node deployments that also include obstructions. Secondly, we would like to use the principle of robust quads to compute the optimal placement of additional nodes so that a partially localized graph becomes fully localizable. We would also like to extend the algorithm to the three-dimensional case by deriving a test for 3D robustness analogous to the test described herein for 2D robustness. Finally, it would be useful to further refine our approach to allow "passive" mobile nodes to localize without transmitting.

Appendix A

Localization Code

The following code is a reference implementation of Phase I (cluster localization) of the algorithm written in C. This code has been used as part of a TinyOS NesC module to perform localization directly on board a Cricket node. The `FindQuads()` function is the entry point for the algorithm.

```
/* Distances between each pair of nodes.  This array does not
 * need to be complete.  Any element set to zero is assumed
 * to be not present.  Since each pair appears twice in this
 * 2-D array, we use half for temporary storage and the other
 * half for storage of the original values.  The macros below
 * make that convenient. */
uint16_t d_norm[MAX_NEIGHBORS+1][MAX_NEIGHBORS+1];

/* Data access macros to get distances from d_norm */
#define GET_D(i,j)  (((i)<(j))?d_norm[i][j]:d_norm[j][i])
#define GET_D_ORIG(i,j)  (((i)>(j))?d_norm[i][j]:d_norm[j][i])
#define CLEAR_D(i,j)  (((i)<(j))?(d_norm[i][j]=0):(d_norm[j][i]=0))

/* Number of nodes available for localization. */
uint8_t norm_count;

/* Maps node indices for our internal use to node indices used
 * by external modules in the system. */
uint8_t nodes[MAX_NEIGHBORS+1];

/* Information about the nodes */
struct NodeInfo {
    uint8_t id[4];      /* Its 4-byte ID */
    uint8_t status;     /* status of our distance cache */
    uint8_t next;       /* pointer to next node for hash table collisions */
} * info;
```

```
/* Array of localized positions.  There are two copies since we have
 * to make several passes until the best set of localizations is
 * found. */
struct node_pos {
    uint8_t set;        // whether a position is set
    double x, y;        // the localized position
} pos[2][MAX_NEIGHBORS];

/* Number of localized positions in pos[] */
uint8_t pos_num[2];

/* Types of errors encountered during trilateration */
#define ERROR_NONINTERSECT  0xfe
#define ERROR_BASELINE      0xff

/* Since the origin of the cluster is always at (0,0) and we don't
 * want to waste storage for that, this macro provides a convenience
 * for reading the coordinates of an arbitrary node. */
#define GET_POS(_i,_x,_y) \
    do { \
        if ((_i) == norm_count) { \
            _x = 0; \
            _y = 0; \
        } else { \
            _x = pos[1][_i].x; \
            _y = pos[1][_i].y; \
        } \
    } while (0)

/* Quality of trilateration flags */
#define LOC_NONE    0
#define LOC_ROBUST  1
#define LOC_WEAK    2


/* The entry point of the localization algorithm.  Picks a
 * robust quad as a starting point and then continues the
 * localization from there.  Other starting points are also
 * considered in case they yield more localizations.  When
 * this function is run, d_norm[][], info[], norm_count, and
 * nodes[] must already be set. */
void FindQuads()
{
    uint8_t j, k, l;
    uint8_t i = norm_count;

    UARTOutput(OUT_DEBUG, "Localizing...\n");
```

```
pos_num[0] = 0;
ring_init();

/* Loop through all neighboring nodes as a possible second node */
for (j=0; j<norm_count; j++) {                                          80
    uint16_t dij = GET_D(i,j);
    /* Only consider nodes with a distance to the origin */
    if (dij == 0)
        continue;
    /* Only consider static nodes */
    if (info[nodes[j]].status & IS_MOVING)
        continue;
    /* Loop through all nodes as a possible third node */
    for (k=0; k<norm_count; k++) {
        uint16_t djk = GET_D(j,k);                                      90
        uint16_t dik = GET_D(i,k);
        /* Only consider third nodes that form a fully-connected
         * triangle with the origin and second node. */
        if (k==j || djk==0 || dik==0)
            continue;
        if (info[nodes[k]].status & IS_MOVING)
            continue;
        /* The triangle must also be robust */
        if (!is_robust(dij,dik,djk))
            continue;                                                   100
        /* Three nodes are enough to define the coordinate system,
         * so we do it. */
        init_pos(i, j, k, dij, dik, djk);

        /* Loop through all nodes as a possible fourth node */
        for (l=0; l<norm_count; l++) {
            uint16_t dkl = GET_D(k,l);
            uint16_t dlj, dil;
            /* Only consider fourth nodes that form a fully-connected
             * quad with the origin, first, and second nodes. */       110
            if (l==k || l==j || dkl==0)
                continue;
            dlj = GET_D(l,j);
            if (dlj == 0)
                continue;
            dil = GET_D(i,l);
            if (dil == 0)
                continue;
            /* Check the quad for robustness, localize it, and
             * enqueue if possible. */                                 120
            check_quad_and_enqueue(i, j, k, l,
                    djk, dij, dik, dkl, dlj, dil);
```

```
            }
            /* Since we've now enumerated all quads that contain
             * edge j-k, we remove it from future consideration. */
            CLEAR_D(j,k);

            /* Do the breadth-first search, localizing as many
             * nodes as possible with this starting point. */
            if (!ring_empty())                                          130
                quad_bfs(i);

            /* Update the localization state and abort if we've
             * localized all nodes. */
            update_pos();
            if (pos_num[0] == norm_count)
                break;
        }
        if (pos_num[0] == norm_count)
            break;                                                      140
    }

    /* Print localization results */
    UARTOutput(OUT_INFO, "Localized %d of %d neighbors:\n",
            pos_num[0], norm_count);
    if (pos_num[0] > 0) {
        for (l=0; l<norm_count; l++) {
            if (pos[0][l].set) {
                UARTOutput(OUT_INFO, "  %02d: x=%5.1f  y=%5.1f ",
                        nodes[l], pos[0][l].x/DISTANCE_MULT_US,         150
                        pos[0][l].y/DISTANCE_MULT_US);
                if (pos[0][l].set == 2)
                    UARTOutput(OUT_INFO, "*");
                if (info[nodes[l]].status & IS_MOVING) {
                    UARTOutput(OUT_INFO, "m");
                }
                UARTOutput(OUT_INFO, "\n");
            }
        }
    }                                                                   160
    UARTOutput(OUT_INFO, "done\n");
}

/* The heart of the localization algorithm.  This function does
 * a breadth-first search into the graph by following robust
 * quads that overlap by three nodes.  Thus, by starting with
 * known positions of three nodes in the quad, we are able to
 * localize the fourth, and then continue the search until a
 * maximum number of nodes have been localized.  Calling
 * this function assumes that the queue has already been        170
```

```
 * initialized with at least one quad, and those quads in the
 * queue have three nodes with known positions.
 *
 * Arguments:
 *      i           The origin node (present in all quads)
 */
void quad_bfs(uint8_t i)
{
    uint8_t v[3];
    uint8_t p, l;                                                    180
    uint16_t djk, dij, dik, dkl, dlj, dil;

    /* Continue the search until no more quads are available */
    while (!ring_empty()) {
        ring_dequeue(v);
        /* Any quad we dequeue has already been localized when
         * it was enqueued.  Thus we immediately begin by
         * enumerating all quads that border the dequeued quad.
         * Consider the triangle composed of nodes in the quad
         * that are _not_ the origin node.  This triangle has          190
         * three edges.  One edge has already has all its bordering
         * quads fully enumerated in previous passes.  The other
         * two edges (adjacent to the most recently localized node)
         * have not, so we enumerate all their bordering quads. */

        /* Loop through the two un-enumerated edges */
        for (p=0; p<2; p++) {
            /* Get the first three edges of any bordering quad.
             * These edges are present in _all_ bordering quads. */
            djk = GET_D_ORIG(v[!p],v[2]);                            200
            dij = GET_D(i,v[!p]);
            dik = GET_D(i,v[2]);
            /* Loop through all nodes as the possible fourth node
             * in the quad. */
            for (l=0; l<norm_count; l++) {
                /* Fourth node must be different than the first three */
                if (l==v[2] || l==v[!p])
                    continue;
                dkl = GET_D(l,v[2]);
                dlj = GET_D(l,v[!p]);                                210
                /* Fourth node must form a fully-connected quad */
                if (dkl==0 || dlj==0)
                    continue;
                dil = GET_D(i,l);
                if (dil == 0)
                    continue;
                /* Check the quad for robustness, localize the
                 * fourth node if possible, and enqueue it. */
```

```
                check_quad_and_enqueue(i, v[!p], v[2], l,
                        djk, dij, dik, dkl, dlj, dil);

                /* Let any pending tasks run */
                TOSH_flush_tasks();
            }
            /* Since we've enumerated all quads bordering this edge,
             * we clear it to avoid searching the same edge in the
             * future. */
            CLEAR_D(v[!p],v[2]);
        }
        /* Abort the search early if all nodes have been localized */
        if (pos_num[1] == norm_count)
            break;
    }
}

/* Given a new quad that we've encountered in the breadth-first
 * search:
 *  1) check if it's robust
 *  2) if so, trilaterate the unknown vertex and enqueue the
 *       quad for later use in the BFS.
 * If the quad is not robust, we can still trilaterate the unknown
 * vertex, but we don't enqueue it.
 * Also, if the trilateration is found to have a baseline ambiguity
 * we don't enqueue it.
 *
 * Arguments:
 *       i,j,k,l          The four vertices of the quad
 *       djk, dij, etc.   The six distances of the quad
 */
void check_quad_and_enqueue(uint8_t i, uint8_t j, uint8_t k, uint8_t l,
        uint16_t djk, uint16_t dij, uint16_t dik,
        uint16_t dkl, uint16_t dlj, uint16_t dil)
{
    double x, y;
    uint8_t ret;

    /* Check if three of the quads triangles are robust.  The fourth
     * triangle has already been checked in earlier stages of the
     * algorithm. */
    if (is_robust(dij,dil,dlj) &&
            is_robust(dik,dil,dkl) && is_robust(djk,dkl,dlj)) {
        /* Only bother trilaterating if the node's position isn't
         * already known. */
        if (pos[1][l].set != LOC_ROBUST) {
            ret = trilaterate(i, j, k, dil, dlj, dkl, &x, &y);
            if (!ret) {
```

```
                pos[1][l].set = LOC_ROBUST;
                pos[1][l].x = x;
                pos[1][l].y = y;
                pos_num[1]++;                                          270
            }
            /* If there's a baseline ambiguity, store the solved position
             * but return immediately without enqueuing the quad. */
            else if (ret == ERROR_BASELINE && pos[1][l].set == LOC_NONE) {
                pos[1][l].set = LOC_WEAK;
                pos[1][l].x = x;
                pos[1][l].y = y;
                UARTOutput(OUT_DEBUG, "Weak trilat of %d\n", nodes[l]);
                return;
            }                                                          280
            else {
                UARTOutput(OUT_DEBUG, "Trilat of %d rejected\n",
                        nodes[l]);
                return;
            }
        }
        UARTOutput(OUT_DEBUG, "RQ: %02d %02d %02d\n",
                nodes[j], nodes[k], nodes[l]);

        /* Only use for further trilateration if node is not moving */     290
        if (!(info[nodes[l]].status & IS_MOVING))
            ring_enqueue(j, k, l);
    }
    else if (pos[1][l].set == LOC_NONE) {
        /* If the quad is not robust, we can still trilaterate, but
         * we don't use it for further localization (by not enqueuing
         * it). */
        ret = trilaterate(i, j, k, dil, dlj, dkl, &x, &y);
        if (!ret) {
            pos[1][l].set = LOC_WEAK;                                  300
            pos[1][l].x = x;
            pos[1][l].y = y;
            UARTOutput(OUT_DEBUG, "Very weak trilat of %d\n", nodes[l]);
        }
    }
}

/* Initialize the first few three nodes of a localization.  This
 * assumes they have already been checked for robustness and
 * connectivity.  We merely solve for their underconstrained              310
 * positions, thus defining a local coordinate system.
 *
 * Arguments:
 *      i, j, k         The indices of the origin and two other nodes
```

```
 *      dij, dik, djk   Distances between nodes
 */
void init_pos(uint8_t i, uint8_t j, uint8_t k,
        uint16_t dij, uint16_t dik, uint16_t djk)
{
    double alpha;                                                    320
    uint8_t l;

    /* Reset the position array */
    for (l=0; l<MAX_NEIGHBORS; l++) {
        pos[1][l].set = LOC_NONE;
    }

    /* The first node defines (0,0) implicitly */

    /* The second node defines the x-axis */                         330
    pos[1][j].set = LOC_ROBUST;
    pos[1][j].x = dij;
    pos[1][j].y = 0.0;

    /* The third node defines the positive direction of the y-axis.
     * Its position is computed using law of cosines. */
    alpha = (square(dij)+square(dik)-square(djk))/(2.0*dij*dik);
    pos[1][k].set = LOC_ROBUST;
    pos[1][k].x = alpha*dik;                    /* dik * cos(th) */
    alpha = 1 - square(alpha);                                       340
    if (alpha < 0)
        UARTOutput(OUT_WARNING, "Warning: negative square root\n");
    pos[1][k].y = sqrt(alpha)*dik;     /* dik * sin(th) */

    pos_num[1] = 2;

}

/* Check the temporary list of positions with the best-so-far
 * list of positions.  If the temporary list is better, use it     350
 * instead. */
void update_pos() {
    if (pos_num[1] > pos_num[0]) {
        /* If the temporary list has more localized positions,
         * replace the best-so-far list. */
        pos_num[0] = pos_num[1];
        memcpy(pos[0], pos[1], sizeof(pos[1]));
        UARTOutput(OUT_DEBUG, "Solved for %d (best)\n", pos_num[1]);
    }
    else                                                             360
        UARTOutput(OUT_DEBUG, "Solved for %d\n", pos_num[1]);
}
```

```
/* The threshold for triangle robustness in units of length:
 * sound-microseconds.  Ideally, this value should be about three
 * standard deviations of the measurement noise of the system. */
#define ROBUST_THRESH       90

/* Given the lengths of the three sides of a triangle, return 1
 * if the triangle is robust. */                                    370
uint8_t is_robust(uint16_t a, uint16_t b, uint16_t c)
{
    double min, d, e;
    double costh;

    if (a <= b && a <= c)
        min = a, d = b, e = c;
    else if (b <= a && b <= c)
        min = b, d = a, e = c;
    else                                                             380
        min = c, d = a, e = b;

    costh = (square(d)+square(e)-square(min))/(2.0*d*e);
    if (min*(1-square(costh)) < ROBUST_THRESH)
        return 0;
    else
        return 1;
}

/* Solves for the position of a node using the known positions      390
 * of three nodes and three distances to those nodes.  This
 * function basically evaluates trilaterate1() thrice, finding
 * the centroid of the three positions returned by trilaterate1().
 *
 * Arguments:
 *      i1, i2, i3      Indices of the three known nodes
 *      r1, r2, r3      Distances to the three nodes
 *      x, y            Return pointers for the solved position
 */
uint8_t trilaterate(uint8_t i1, uint8_t i2, uint8_t i3,          400
        uint16_t r1, uint16_t r2, uint16_t r3,
        double * x, double * y)
{
    double xt, yt;
    uint8_t ret, bl, bad=0;
    ret = trilaterate1(i1, i2, i3, r1, r2, r3, x, y);
    if (ret == ERROR_NONINTERSECT)
        return ERROR_NONINTERSECT;
    if (ret == ERROR_BASELINE)
        bad = 1;                                                     410
```

```
    bl = ((ret >> 1) & 1) | ((ret << 1) & 2) | (~ret & 4);
    ret = trilaterate1(i1, i3, i2, r1, r3, r2, &xt, &yt);
    if (ret == ERROR_NONINTERSECT)
        return ERROR_NONINTERSECT;
    /* In addition to checking for a baseline error, we make sure
     * that each trilateration solves for a position in the same
     * region as the last (see comments in trilaterate1() about the
     * 7 possible regions). That's what all this bitshifting
     * nonsense is about. */
    if (ret == ERROR_BASELINE || ret != bl)                          420
        bad = 1;
    *x += xt;
    *y += yt;
    bl = ((~bl >> 2) & 1) | (~bl & 2) | ((~bl << 2) & 4);
    ret = trilaterate1(i2, i3, i1, r2, r3, r1, &xt, &yt);
    if (ret == ERROR_NONINTERSECT)
        return ERROR_NONINTERSECT;
    if (ret == ERROR_BASELINE || ret != bl)
        bad = 1;
    *x += xt;                                                        430
    *y += yt;
    *x /= 3.0;
    *y /= 3.0;
    /* In the case of baseline errors, we still solve for the
     * localization, but we note the error for higher-level
     * processing. */
    if (bad)
        return ERROR_BASELINE;

    return 0;                                                        440
}

/* Solves for the position of a node using the known positions
 * of three nodes and three distances to those nodes. The
 * technique here is quite simplified - the distances and positions
 * of the first two nodes define two circles that intersect (in
 * the normal case) at two points. We choose one of these two
 * points as the answer based on the distance to third node.
 * This is just a helper function for trilaterate() which does
 * it more accurately.                                             450
 *
 * Arguments:
 *      i1, i2, i3      Indices of the three known nodes
 *      r1, r2, r3      Distances to the three nodes
 *      x, y            Return pointers for the solved position
 */
uint8_t trilaterate1(uint8_t i1, uint8_t i2, uint8_t i3,
        uint16_t r1, uint16_t r2, uint16_t r3,
```

```
    double * x, double * y)
{                                                                    460
    double dsq, gam, xa, ya, xb, yb, xt1, xt2, yt1, yt2, d1, d2;
    double x_1, y_1, x_2, y_2, x_3, y_3;
    uint8_t ret;

    /* Get the positions of the three known nodes */
    GET_POS(i1, x_1, y_1);
    GET_POS(i2, x_2, y_2);
    GET_POS(i3, x_3, y_3);

    /* Find the two intersection points of the two circles. */           470
    dsq = square(x_2-x_1) + square(y_2-y_1);
    gam = (square((double)r2+r1) - dsq)*(dsq - square((double)r2-r1));
    if (gam < 0) {
        UARTOutput(OUT_DEBUG, "Nonintersecting circles\n");
        return ERROR_NONINTERSECT;
    }
    gam = sqrt(gam);
    xa = -(square(r2)-square(r1))*(x_2-x_1)/(2.0*dsq) + (x_1+x_2)*0.5;
    ya = -(square(r2)-square(r1))*(y_2-y_1)/(2.0*dsq) + (y_1+y_2)*0.5;
    xb = (y_2-y_1)*gam/(2.0*dsq);                                    480
    yb = (x_2-x_1)*gam/(2.0*dsq);
    xt1 = xa - xb;
    xt2 = xa + xb;
    yt1 = ya + yb;
    yt2 = ya - yb;

    /* Disambiguate between the two points using the third node. */
    d1 = sqrt(square(xt1-x_3) + square(yt1-y_3));
    d2 = sqrt(square(xt2-x_3) + square(yt2-y_3));
    if (fabs(d1-r3) < fabs(d2-r3)) {                                 490
        *x = xt1;
        *y = yt1;
    }
    else {
        *x = xt2;
        *y = yt2;
    }

    /* The three nodes of known position define a triangle in 2-space.
     * Consider the three sides of this triangle, and extend the       500
     * length of each edge to infinity. This divides the plane
     * into 7 distinct regions. The position we computed above is
     * in one of these 7 regions. The point of all the computation
     * below is to compute which region. We do this by finding
     * "which side" we are on of each edge of the triangle. This is
     * done for each of the three edges and the result is a 3-bit
```

```
     * bitmask where each bit tells us which side we are on.  This
     * is the return value of the function.
     *
     * We also check if the solved position is "close" to one of the
     * edges (within 2 standard deviations of the measurement noise).
     * If so, we return an error that indicates potential for
     * localization ambiguity.
     */
    gam = (x_2-x_1)*(*y-y_1) - (y_2-y_1)*(*x-x_1);
    if (fabs(gam)/sqrt(dsq) < 2*ROBUST_THRESH)
        return ERROR_BASELINE;
    ret = (gam > 0);

    gam = (x_3-x_1)*(*y-y_1) - (y_3-y_1)*(*x-x_1);
    dsq = square(x_3-x_1) + square(y_3-y_1);
    if (fabs(gam)/sqrt(dsq) < 2*ROBUST_THRESH)
        return ERROR_BASELINE;
    ret |= (gam > 0) << 1;

    gam = (x_3-x_2)*(*y-y_2) - (y_3-y_2)*(*x-x_2);
    dsq = square(x_3-x_2) + square(y_3-y_2);
    if (fabs(gam)/sqrt(dsq) < 2*ROBUST_THRESH)
        return ERROR_BASELINE;
    ret |= (gam > 0) << 2;
    return ret;
}

/* Head/tail pointers for the robust quad queue */
uint16_t qhead, qtail;

/* The queue of robust quads */
struct quad {
    uint8_t v[3];   // 3 vertices of the quad (4th is always
                    // the cluster origin)
} q[800];

/* Initialize the ring-buffer holding a queue of robust quads */
void ring_init()
{
    qhead = qtail = 0;
}

/* Return 1 if the queue is empty */
uint8_t ring_empty()
{
    return (qhead == qtail);
}
```

```
/* Enqueue an element (3 vertices) onto the quad queue. */
result_t ring_enqueue(uint8_t i, uint8_t j, uint8_t k)
{
    if ((qhead+1) == qtail || (qhead==(QUEUE_LEN-1) && qtail==0)) {
        UARTOutput(OUT_ERROR, "ring_enqueue(): queue is full\n");
        return FAIL;                                                    560
    }
    q[qhead].v[0] = i;
    q[qhead].v[1] = j;
    q[qhead].v[2] = k;
    qhead++;
    if (qhead == QUEUE_LEN)
        qhead = 0;
    return SUCCESS;
}
                                                                        570
/* Dequeue an element (3 vertices) from the quad queue. */
result_t ring_dequeue(uint8_t * v)
{
    if (ring_empty())
        return FAIL;

    v[0] = q[qtail].v[0];
    v[1] = q[qtail].v[1];
    v[2] = q[qtail].v[2];
    qtail++;                                                            580
    if (qtail == QUEUE_LEN)
        qtail = 0;
    return SUCCESS;
}
```

Appendix B

Filtering Code

The following code is a reference implementation of the Kalman filtering step used to reduce noise in node-to-node distance measurements. It is run prior to the localization code shown in Appendix A. This code has been used as part of a TinyOS NesC module to perform localization directly on board a Cricket node. The `new_measurement()` function performs the filtering.

```
/* Information about each neighboring node */
struct NodeInfo {
    uint8_t id[4];      /* Its 4-byte ID */
    uint8_t status;     /* status of our distance cache */
    uint8_t next;       /* pointer to next node for hash table collisions */
} neighbors[MAX_NEIGHBORS+1];

/* Values for the status field of neighbors[] */
#define EMPTY           0xff
#define RELAY_DATA      0x1
#define OLD_MEAS        0x2
#define OLD_RELAY       0x4
#define IS_MOVING       0x8
#define IS_NEW          0x10

/* dists[][] records pairwise distances between every pair of nodes */
int16_t dists[MAX_NEIGHBORS+1][MAX_NEIGHBORS+1];

/* Macros for getting and setting the recorded distance. */
#define SET_DIST(a,b,x)     dists[a][b] = (x)
#define GET_DIST(a,b)       dists[a][b]

/* Number of measurements recorded before the Kalman filter is started. */
#define NUM_INIT_MEAS   5
```

```
/* Variance of rate of change of distance for filtering
 * (in sound-microseconds per second squared) */
#define VEL_VARIANCE          100    // static case
#define VEL_VARIANCE_MOTION  360000 // mobile case
/* Variance of underlying measurements (in sound-microseconds squared) */
#define MEAS_VARIANCE         3600                                    30
/* Maximum measurable distance (in sound-microseconds) */
#define MAX_DIST              30000

/* Special value of distance we use to keep track of outliers so we
 * can debug their effect on the system. */
#define OUTLIER_DIST    1

/* Clamps a value between two extremum */
#define CLAMP(x,a,b)        (((x)<(a))?(a):(((x)>(b))?(b):(x)))
                                                                    40
/* Kalman filter state for each distance measured to a neighbor. */
struct NodeFiltInfo {
    union {
        uint16_t meas_time; // timestamp of last measurement
        uint16_t n;         // number of initial measurements
    };
    uint8_t bad_count;      // number of consecutive outliers
    int16_t vel;            // rate of change of distance
    union { // used either for covariance or filter initialization
        float P[4];         // current covariance matrix               50
        uint16_t d[8];      // initial measurements
    };
} filt[MAX_NEIGHBORS];

/* Possible flags for the arguments to new_measurement() */
#define CR_MOVING   1


/* Records a new distance measurement by doing any necessary
 * Kalman filtering and then storing the value.                        60
 * Arguments:
 *      id          Unique ID of the node to which we are measuring
 *      x           Distance measured (in sound-microseconds)
 *      timestamp   Value of the millisecond system clock
 *      flags       Any relevant flags (either 0 or CR_MOVING)
 */
result_t new_measurement(uint8_t * id, uint16_t x, uint16_t timestamp,
        uint8_t flags)
{
    uint8_t node;                                                   70
    float * P;
    float newP[4];
```

```
int16_t v;
float d, a, dt;
float K[2];
uint8_t moving = 0;
float mdist;

/* Obvious outliers are rejected. */
if (x > MAX_DIST)                                                    80
    return SUCCESS;

/* Find the ID in our list, or create a new entry if we haven't
 * seen this node before. */
node = node_find(id);
if (node == EMPTY) {
    uint16_t * sd;
    node = node_add(id);
    if (node == EMPTY)
        return FAIL;                                                 90

    /* The node is new to us, so we initialize the Kalman
     * filter.  We do this by taking the median of the first
     * NUM_INIT_MEAS measurements as the initial state for
     * the filter.  Since we only have one measurement so far,
     * we add it to the list. */

    /* Official distance and velocity are not known yet
     * (so set them to zero). */
    SET_DIST(SELF, node, 0);                                         100
    filt[node].vel = 0;
    /* Number of recorded distances is 1 */
    filt[node].n = 1;
    sd = filt[node].d;
    sd[1] = sd[2] = sd[3] = sd[4] = sd[5] = sd[6] = sd[7] = 0;
    /* Known distance */
    sd[0] = x;

    neighbors[node].status |= IS_NEW;
    filt[node].bad_count = 0;                                        110
    return SUCCESS;
}

/* Clear the aging flag */
neighbors[node].status &= ~OLD_MEAS;

if (flags & CR_MOVING)
    neighbors[node].status |= IS_MOVING;
else
    neighbors[node].status &= ~IS_MOVING;                            120
```

```
/* A convenience pointer */
P = filt[node].P;

/* If the filter is still being initialized */
if (neighbors[node].status & IS_NEW) {
    uint8_t n = filt[node].n;
    uint16_t * sd = filt[node].d;
    /* Incorporate the new distance */
    sd[n] = x;                                                    130
    n++;

    if (n == NUM_INIT_MEAS) {
        /* Enough measurements are recorded to initialize the
         * Kalman filter */
        uint8_t i, j;
        uint16_t t;
        /* Bubble sort the initial measurements */
        for (i=0; i<n; i++) {
            for (j=i+1; j<n; j++) {                               140
                if (sd[j] < sd[i]) {
                    t = sd[i];
                    sd[i] = sd[j];
                    sd[j] = t;
                }
            }
        }
        /* Pick the median as the initial filter state */
        SET_DIST(SELF, node, sd[(n-1)>>1]);
                                                                  150
        /* Initialize the covariance */
        P[0] = P[1] = P[2] = P[3] = 0;
        neighbors[node].status &= ~IS_NEW;

        /* Record the timestamp */
        filt[node].meas_time = timestamp;
    }
    else {
        filt[node].n = n;
    }                                                             160
    return SUCCESS;
}

/* If there have been three outliers in a row, reset the filter. */
if (filt[node].bad_count == 3) {
    P[0] = P[1] = P[2] = P[3] = 0;
    filt[node].meas_time = timestamp;
    SET_DIST(SELF, node, x);
```

```
    filt[node].vel = 0;
    filt[node].bad_count = 0;                                          170
    return SUCCESS;
}

/* Time since last measurement in seconds */
dt = (float)(timestamp - filt[node].meas_time)/1024.0;
v = filt[node].vel;

/* KALMAN FILTERING BEGINS HERE */

/* Prediction step for the filter state */                             180
d = (float)GET_DIST(SELF, node) + v*dt;

/* Moving distances get a much larger process noise than
 * static distances. */
if ((neighbors[node].status & IS_MOVING) ||
        (neighbors[SELF].status & IS_MOVING)) {
    moving = 1;
    a = VEL_VARIANCE_MOTION;
}
else                                                                   190
    a = VEL_VARIANCE;

/* Prediction step for covariance. */
newP[0] = P[0] + P[2]*dt + P[1]*dt + P[3]*dt*dt + a*dt*dt;
newP[1] = P[1] + P[3]*dt                        + a*dt;
newP[2] = P[2] + P[3]*dt                        + a*dt;
newP[3] = P[3]                                  + a;

/* Gain step */
a = newP[0] + MEAS_VARIANCE;                                           200
K[0] = newP[0] / a;
K[1] = newP[2] / a;

/* Compute the Mahalanobis distance of the measurement */
mdist = ((float)x - d)/sqrt(a);

if (fabs(mdist) <= 3.0) {
    /* Incorporate the measurement into the filter if it is within
     * a Mahalanobis distance of 3.0 */
                                                                       210
    /* Update step for the filter state (distance) */
    a = CLAMP(d + K[0]*((float)x - d), 0, MAX_DIST);
    SET_DIST(SELF, node, a);

    if (moving) {
        /* Update step for the filter state (velocity) */
```

```
            v = CLAMP(v + K[1]*((float)x - d), -MAX_DIST, MAX_DIST);
            filt[node].vel = v;

            /* Record the distance in the moving node tracker                    220
             * if necessary */
            if (neighbors[node].status & IS_MOVING)
                new_tracked_dist(timestamp, node, SELF, a);
        }
        else {
            /* If the distance is static, force the velocity to zero. */
            filt[node].vel = 0;
        }

        /* Update step for the covariance */                                     230
        P[0] = (1.0 - K[0])*newP[0];
        P[1] = (1.0 - K[0])*newP[1];
        P[2] = -K[1]*newP[0] + newP[2];
        P[3] = -K[1]*newP[1] + newP[3];

        filt[node].meas_time = timestamp;
        filt[node].bad_count = 0;
    }
    else {
        /* Reject the measurement as an outlier */                               240
        filt[node].bad_count++;
        /* Record the distance in the moving node tracker
         * if necessary */
        if (neighbors[node].status & IS_MOVING)
            new_tracked_dist(timestamp, node, SELF, OUTLIER_DIST);
    }

    return SUCCESS;
}
```

Bibliography

[1] Nirupama Bulusu, John Heidemann, and Deborah Estrin. GPS-less low cost outdoor localization for very small devices. *IEEE Personal Communications Magazine*, 7(5):28–34, October 2000.

[2] Srdan Capkun, Maher Hamdi, and Jean-Pierre Hubaux. GPS-free positioning in mobile ad-hoc networks. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, 2001.

[3] Lance Doherty, Kristofer S. J. Pister, and Laurent El Ghaoui. Convex position estimation in wireless sensor networks. In *Proc. IEEE INFOCOM*, Anchorage, AK, April 2001.

[4] T. Eren, D. Goldenberg, W. Whiteley, Y. R. Yang, A. S. Morse, B. D. O. Anderson, and P. N. Belhumeur. Rigidity, computation, and randomization in network localization. In *Proc. IEEE INFOCOM*, March 2004.

[5] Deborah Estrin, Ramesh Govindan, and John Heidemann. Embedding the internet: introduction. *Commun. ACM*, 43(5):38–41, 2000.

[6] Robert Grabowski and Pradeep Khosla. Localization techniques for a team of small robots. In *Proc. IEEE IROS*, Maui, Hawaii, October 2001.

[7] Bruce Hendrickson. Conditions for unique graph realizations. *SIAM J. Comput.*, 21(1):65–84, 1992.

[8] B. K. P. Horn. Closed form solution of absolute orientation using unit quaternions. *Journal of the Optical Society A*, 4(4):629–642, April 1987.

[9] Xiang Ji and Hongyuan Zha. Sensor positioning in wireless ad-hoc sensor networks using multidimensional scaling. In *Proc. IEEE INFOCOM*, March 2004.

[10] G. Laman. On graphs and rigidity of plane skeletal structures. *J. Engineering Math*, 4:331–340, 1970.

[11] Radhika Nagpal, Howard Shrobe, and Jonathan Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *Proc. IPSN*, pages 333–348, Palo Alto, CA, April 2003.

[12] Dragos Niculescu and Badri Nath. DV based positioning in ad hoc networks. *Kluwer journal of Telecommunication Systems*, pages 267–280, 2003.

[13] Dragos Niculescu and Badri Nath. Error characteristics of ad hoc positioning systems (APS). In *Proc. 5th ACM MobiHoc*, Tokyo, May 2004.

[14] Neal Patwari, Alfred O. Hero III, Matt Perkins, Neiyer S. Correal, and Robert J. O'Dea. Relative location estimation in wireless sensor networks. *IEEE Trans. Signal Process.*, 51(8):2137–2148, August 2003.

[15] Nissanka B. Priyantha, Hari Balakrishnan, Erik Demaine, and Seth Teller. Anchor-free distributed localization in sensor networks. Technical Report 892, MIT Lab. for Comp. Sci., April 2003.

[16] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The Cricket location-support system. In *Proc. 6th ACM MobiCom*, Boston, MA, August 2000.

[17] Chris Savarese, Jan Rabaey, and Koen Langendoen. Robust positioning algorithms for distributed ad-hoc wireless sensor networks. In *USENIX Annual Tech. Conf.*, pages 317–327, Monterey, CA, June 2002.

[18] Andreas Savvides, Wendy Garber, Sachin Adlakha, Randolph Moses, and Mani B. Srivastava. On the error characteristics of multihop node localization in ad-hoc sensor networks. In *Proc. IPSN*, pages 317–332, Palo Alto, CA, April 2003.

[19] Andreas Savvides, Chih-Chieh Han, and Mani B. Srivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Proc. 7th ACM MobiCom*, pages 166–179, Rome, Italy, 2001.

[20] J. B. Saxe. Embeddability of weighted graphs in k-space is strongly NP-hard. In *Proc. 17th Allerton Conf. Commun. Control Comput.*, pages 480–489, 1979.

[21] Slobodan N. Simic and Shankar Sastry. Distributed localization in wireless ad hoc networks. Technical Report UCB/ERL M02/26, UC Berkeley, December 2001.

[22] Adam Smith, Hari Balakrishnan, Michel Goraczko, and Nissanka Priyantha. Tracking moving devices with the cricket location system. In *Proc. 2nd ACM MobiSys*, pages 190–202, Boston, MA, June 2004.

[23] Seth Teller, Jiawen Chen, and Hari Balakrishnan. Pervasive pose-aware applications and infrastructure. *IEEE Computer Graphics and Applications*, pages 14–18, July/August 2003.