Mutual Information Based Tracking With Mobile Sensors

by

John A. Russ

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

Author .....
Department of Electrical Engineering and Computer Science
June 11, 2010

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

Certified by .....
John Fisher
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Mutual Information Based Tracking With Mobile Sensors

by

John A. Russ

Submitted to the Department of Electrical Engineering and Computer Science
on June 11, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

In order to utilize mobile sensor nodes in a sensing and estimation problem, one must carefully consider the optimal placement of those sensor nodes and simultaneously account for the cost incurred in moving the sensor nodes. We present an approximate dynamic programming approach to a tracking problem with mobile sensor nodes. We utilize mutual information as the objective for optimal sensor placement. We show how a constrained dynamic programming approach allows us to balance estimation quality against mobility costs. However this constrained optimization problem is NP-hard. We present a set of approximations that allow this dynamic program to be solved with polynomial complexity in the number of sensors. We present a greedy multiple time step planning algorithm that greedily selects the most informative paths over a fixed planning horizon. These approximation algorithms are verified via simulation to give a comparative analysis of estimate quality and mobility costs.

Thesis Supervisor: Daniela Rus
Title: Professor

Thesis Supervisor: John Fisher
Title: Principal Research Scientist

Acknowledgments

I would like to thank Professor Daniela Rus for her support and guidance throughout my graduate school experience. She allowed me the freedom to explore those areas where my passions drove me. I am grateful to Dr. John Fisher for providing me with an interesting problem and giving me everything I needed to take on the challenge. I am also grateful to all of my fellow lab mates in the DRL group for acting as a sounding board for my ideas and for keeping graduate life fun and interesting.

Most of all I would like to thank my wife, Emily. She has sustained me and supported me in all of my pursuits. I will be forever grateful to her, and I am excited to have her by my side as we move on to life's next adventure. I am also grateful to my children for always seeing the best in me and inspiring me to better myself and the world around me for their benefit.

Contents

List of Figures

Chapter 1

Introduction

1.1 Sensor Networks

Advancements in electronics, robotics, and computer science have made it possible to create networks of sensors and actuators that can autonomously observe and interact with their environment. These networks open the door to a vast array of fascinating and valuable applications. To design and operate such networks one must address problems with communications, control, reliability, and planning to name only a few. Regardless of the application or specific problem of focus, the goal is to ensure that the network is obtaining the information that will provide the best possible estimate and doing so in the most efficient manner possible.

These networks might consist of thousands of very small, simple sensor nodes that are capable of little more than taking very a simple measurement and communicating it back to neighboring nodes. These nodes could be spread around a large building to monitor environmental conditions, in a hospital to help monitor patients, around a fire or chemical spill to track the hazard as it progresses, on a battlefield to detect intruders, or along a river to detect flood conditions. At the other end of the spectrum, a network could just as easily consist of only a handful of highly capable nodes with multiple sensors, powerful processors, and complex actuators. Some examples would be a small team of unmanned aerial vehicles performing a search and rescue operation, a small team of unmanned underwater vehicles clearing a harbor of mines, or a team

of ground robots doing routine security surveillance.

1.2 Optimal Sensing Under Constraints

In each of these cases the collection of the desired information is constrained by the costs associated with obtaining it. The cost of taking a measurement can come in a variety of forms. For many networks the most significant cost may be the cost of communicating the measurement back for processing. Communication might be costly because of the actual energy expended by the radio or it might be costly because it further congests a limited communication channel. For a mobile sensor the energy consumed by moving to a new measurement location may be the primary concern. For others applications, such as search and rescue, the time required to obtain the information may be the primary concern. On the other hand the cost may actually be the monetary cost of purchasing and deploying more sensors. Regardless of what the specific costs are they all place constraints on what a given network is capable of achieving.

A significant but less obvious challenge is that of identifying and quantifying the value of potential measurements or sensor locations prior to actually taking the measurements. Common approaches consist of attempting to take measurements everywhere possible, taking measurements as close as possible to the object or phenomenon of interest, or selecting the sensor locations that maximize the entropy with respect to the other selected locations. We use a mutual information based approach. Mutual information is an ideal metric because, unlike the previously mentioned approaches, it relates directly to the quality of the *a posteriori* estimate rather than just relying on intuitive heuristics. Also mutual information has some desirable properties that allow us to develop a greedy algorithm with provable bounds on the performance.

We wish to select the optimal set of measurements to take, while simultaneously trying to minimize the cost of taking those measurements. However, this is an NP-hard problem [11, 15]. Left with only approximate solutions there are a variety of paths to take. Dividing the problem into two parts, one part for selecting the most

informative sensor locations and another part for selecting the lowest cost deployment, isn't feasible. This approach is problematic because the optimal sensor selection is often very costly, and likewise the lowest cost solution will often perform very poorly. Other approximation algorithms are designed around static sensor networks and planning for a single fixed deployment and therefore don't lend themselves to dynamic networks of mobile sensor nodes.

Our work is an extension of work done in [26, 27] for sensor management in a communications constrained static sensor network. In [26, 27] the authors develop an approximate dynamic programming approach that allows a static sensor network to balance estimation performance against the cost of communicating measurements and estimation parameters through the network. Their approach also facilitates planning sensor selections over a rolling time horizon rather than only considering a single time step. They demonstrate their algorithm with a simulated tracking exercise. Their problem was concerned with selecting the subset of sensors to be activated at each given time step and selecting which sensor would act as leader node for data fusion and processing.

In this thesis we extend their work to accommodate mobile sensor networks that have mobility cost constraints rather than communication constraints. We utilize a very similar formulation to [27] for the dynamic programming approach and for our tracking simulation, as we describe in Chapter 4 through 6. However, this dynamic programming formulation is intractable in its pure form. The complexity grows exponentially in the number of sensors. In [27] a set of approximations are utilized to allow a tractable solution to be found. However, these approximations rely on the structure of the leader node selection problem and the fact that the sensor nodes are static. For our mobile sensor network problem we are not concerned with the leader node selection problem and we cannot depend on sensor nodes being in the same locations over time. Therefore, new types of approximations are necessary before this dynamic programming approach can be utilized with mobile sensor networks.

1.3 Contributions

The main contributions of our work are the extension of this dynamic programming approach to mobile sensor networks, and the development of new approximations that are appropriate for mobile sensors. We model a mobile sensor network as a series of static networks. With each time slice consisting of a static network of virtual sensor nodes that represent the potential sites that the real mobile sensors can occupy during the next time step. We then utilize a slightly modified version of the dynamic programming approach from [27] to identify the ideal subset of virtual sensor nodes that the real sensor nodes should occupy at the next time step. Where the ideal subset consists of those locations that optimally balance the informational value of the measurements with the cost of moving our sensors to those locations. Our primary contribution is the development of two approximation algorithms that provide tractable solutions to this dynamic program even in the case of mobile sensor networks.

In Section 5.2.2 we present our decoupled multistep algorithm. This approximation reduces the complexity of the solution to being linear in the number of sensors. It also has the advantage of facilitating a decentralized distributed implementation. However, this approximation tends to overestimate the informational value of measurements, which tends to result in less efficient solutions.

In Section 5.2.3 we present our greedy multistep algorithm which relies on the key insight that we can greedily select entire paths of sensor locations rather than just greedily selecting individual sensor placements. The complexity of this greedy multistep algorithm grows as the square of the number of sensors. Therefore it is only slightly more computationally intensive than the decoupled algorithm, but has much better theoretical motivation and performs better in simulation.

Alternate approaches typically reduce the complexity by addressing only part of the problems our algorithms address. For example there are many information only based solutions that neglect to address cost constrained optimization at all. Others tackle the cost constrained sensor placement problem but do so only for static sensor

networks, and thus don't deal with the complexity of planning over a fixed time horizon. Those that do solve a bounded cost sensor placement problem typically suffer from severe restrictions, like only being able to plan for two or three steps ahead. Recent work in [24] presents a bounded cost path planning algorithm for data collection for spatial estimation applications. Their work is based on a recursive greedy algorithm [5] that recursively subdivides the measurement paths and greedily optimizes over the subpaths. Comparisons between our approach and other leading solutions are presented with the related work in Chapter 3.

We fully implemented each of our algorithms in a simulation of a tracking exercise with a team of autonomous kayaks. We demonstrate that our approach performs nearly as well as the optimal for a single step case where the optimal solution can be found explicitly. We also show that our multistep algorithms are able to achieve equivalent or better performance at a lower cost than purely information based, or myopic single step approaches.

1.4 Outline

This thesis will be organized as follows. Chapter 2 discusses optimal sensing in static and mobile networks, along with concepts for determining measurement quality, handling bounded costs, and creating multistep plans. Chapter 3 presents some of the related work and work that influenced our current approach. Chapter 4 shows how we chose to model and formulate our particular tracking problem and the estimators that we use. Chapter 5 presents the dynamic programming approach we employ along with the approximation algorithms we developed to provide tractable solutions. Chapter 6 explains our particular implementation and MATLAB simulations. Also the results and analysis of the simulations are discussed.

Chapter 2

Optimal Sensing

Optimal sensing is the problem of choosing which measurements should be taken such that the most accurate estimate is obtained. The nature of the measurement selection problem takes on a variety of forms. It could be selecting the locations of the sensors, selecting which subset of a larger set of sensors to utilize, or selecting when the sensors should take measurements, to name a few possibilities.

Solving the optimal sensing problem is important to any real sensor network. In the worst case the network will fail completely because it is unable to acquire the information needed to achieve its purpose. This could be from egregious errors like holes in the coverage or from more subtle errors because there is insufficient data to find an accurate or unique solution to the estimation problem. Even in the best case, if the optimal sensing problem isn't solved then the sensor network may be functioning but it will be doing so inefficiently. This could be because too much data is being collected. So energy and resources are being consumed collecting and processing redundant information. So an equally accurate estimate could be obtained at a much lower cost. On the other hand the network may be functioning far below its potential because a more optimal utilization of the same sensors and resources could produce a much more accurate estimate.

Before looking at solutions to this problem it will be useful to look more closely at the structure of sensor networks and what the criteria are for determining the optimality of a given network deployment.

2.1 Types of Sensor Networks

Sensor networks can be characterized in numerous ways based on their structure and capabilities. Distinctions are most frequently based on the number and capability of the individual nodes. On one extreme a "network" might consist of a single node that is very complex and highly capable. The various rovers sent to explore other planets are a prime example of this approach. On the other extreme a network might consist of thousands of tiny nodes with minimal capabilities. This type of network was popularized by the Smart Dust project at Berkeley [25, 20]. One can imagine such a network being used for environmental monitoring to track changes in temperature or on a battlefield to measure vibrations from intruders. Often it is the case that the capabilities of the individual nodes is inversely proportional to the number of nodes in the network. Each of these design decisions provides different advantages and disadvantages for sensing applications.

2.1.1 Smart vs. Dumb Sensors

The capabilities of an individual node usually fit into one of four areas: sensing, communication, computation, and actuation. By definition most sensors nodes will have the first three capabilities in some form, though it may be severely diminished. An object that is capable actuation to move around or manipulate its environment is usually considered to be a robot rather than a sensor node. For our purposes we will treat such multi-robot systems within the sensor network framework. So actuation is simply viewed as a capability that aids the sensor node in obtaining measurements. A smart sensor would be one that has superior function in some or all of these areas, and a dumb sensor would be one that has the minimal function necessary or is missing some of these capabilities completely.

Each of these capabilities will significantly influence the structure and solution to the optimal sensor placement problem. The design of a sensor network must consider the trade off between accomplishing its goals with smart sensors vs. dumb sensors. There are obviously exceptions but generally the trade off is between using a small

number of more capable smart sensors or using a larger number of less capable dumb sensors. Smarter sensors could take more accurate measurements or do so at longer ranges. A smart node may have multiple sensors or more complex sensors that provide information that can't be obtained via simpler methods. They could communicate at higher data rates or over longer distances. They may be able to leverage greater computation to perform adaptive processing or data fusion in the network before sending the data back. Actuation may allow them to move to obtain more accurate measurements.

However, all of these capabilities of smart nodes come with costs. The nodes themselves will have higher material costs. They will generally be larger in size and weight. Their greater complexity increases the risk of something breaking or failing. They will also consume more energy which limits the lifetime or range of the network. All of these costs tend to limit the size of a sensor network consisting of smart nodes.

A great deal of recent work has focused on moving in the other direction. Namely using dumb nodes in larger numbers to achieve the same purpose. These nodes generally have no actuation. They have simple sensors that are less accurate and may only provide a few bits of data. Their communication abilities are limited and usually rely on multihop communication to send information through their immediate neighbors. They may only have enough computational power to manage the sensor and communications hardware.

These networks with dumb sensors make up for their limited capabilities with their size and numbers. These nodes can be very small and cheap. Their size and cost allows them to be deployed in large numbers. The network can compensate for sensing, communication, and mechanical failures through redundancy. However, a large network has new problems. Wireless communications take place over the same channel so self interference can severely constrain the bandwidth available within the network. Recharging or replacing batteries can be difficult because of the number and locations of nodes. So they often must operate under severe energy constraints to maintain a useful lifetime for the network. These variations in sensor node capabilities strongly influence the optimal sensor placement problem. For example a large network

of dumb sensors may have more nodes than it is feasible to process data for. Too many measurements could overwhelm the bandwidth or the computational capacity of the network. In an energy constrained network using all the sensors at once may significantly reduce the lifespan of the network. So the optimal sensing problem is to select the optimal set of sensors to activate at a given time. In another application utilizing smarter sensor nodes, the estimate might be very sensitive to the precise location of the nodes. So optimal sensing problem in this case is primarily concerned with where the nodes should be deployed.

2.1.2 Static vs. Mobile Sensors

In this thesis we focus on an approximate solution to the optimal sensor placement problem for mobile sensor nodes. Most of the work on sensor placement has focused on static sensor networks. Making the jump from static to mobile sensor networks brings several advantages but also introduces several complications.

The two most significant advantages that mobility provides are the ability to handle dynamic situations, and to get the same work done with fewer nodes. Most of the phenomena that one would be interested in measuring via a sensor network are dynamic. They change over time and space. The usual remedy is to overcompensate and use more nodes than are needed so that unforeseen situations can be handled. Even more challenging for a static network are cases where the initial conditions are unknown. This is the norm for many military operations or search and rescue situations. In each case the environment may be new and foreign or may be undergoing significant changes. Mobile sensor nodes can move quickly to accommodate the unknown or dynamic elements of the environment or the phenomenon that is being measured.

Fewer mobile sensors can accomplish the same work as a larger group of static sensors for several reasons. First as mentioned above there is less need to overcompensate for the unknown. Rather than making sure all possibilities are covered, it is often sufficient to ensure that they can be covered by moving the necessary nodes. The following paradigm, which we use in our algorithmic development later, will help

illustrate another way a mobile node can often accomplish the work of multiple static nodes. A mobile sensor network can be viewed as if it is a static sensor network consisting of a large number of inactive virtual nodes. These virtual nodes represent all of the locations to which a real mobile node can potentially move. A virtual node is activated when a real node moves to that location and takes a measurement. In this paradigm it is easy to see that in many situations a single mobile node can collect the same data that a large group of static nodes would by periodically sampling the same locations the static nodes would occupy. This is particularly useful when the sampling rate is slow or in situations where a sensor is particularly complex or expensive and it simply isn't feasible to have a large number of them. Also in some situations a static network may interfere with the regular use of a space. Imagine a busy harbor patrolled by a single unmanned vehicle instead of hundreds of buoys floating around interfering with traffic.

Two of the main trade offs of utilizing mobile sensors are energy requirements and computational complexity. Using mobile nodes allows a network to use fewer nodes and to be more versatile and responsive to changes, but these benefits are paid for with increased energy or fuel requirements. The amount of energy used to move a sensor node will generally be orders of magnitude more than what is required for sensing, computation, and communications. This means that mobile nodes will need to recharge or refuel much more frequently than static nodes. Many current motes that are used in static sensor networks may last years on a single charge. But most mobile platforms would need to recharge in hours or days.

Mobility can also dramatically increase the computational requirements of the network, even if only the sensor placement problem is considered. With a static network the placement is determined when the nodes are deployed, so the problem only needs to be solved once. Returning to the paradigm of a mobile network consisting of snapshots of static networks, each time step can be thought of as a deployment of a static network. So not only does a mobile sensor network need to solve the problem many times, it needs to do it quickly enough to act on the solution and implement it before the next time step. Because of the computational complexity most work in optimal

sensor placement has focused on the simpler case of deploying static networks.

2.2 Measurement Quality

Optimal sensor locations cannot be found without first developing a rigorous definition of what constitutes an optimal measurement. In general the value of the information contained in a measurement will depend on what information is obtained from the other measurements being taken. We will assume that we have discrete set $\mathcal{V}$ of potential sensor locations. The challenge is to identify the set $\mathcal{S} \subseteq \mathcal{V}$ of n sensor locations that will provide the best possible measurements. Measurements taken from the optimal $\mathcal{S}$ should provide the most accurate estimate of the object state $\mathcal{X}$ that we are interested in.

2.2.1 Conditional Entropy

An intuitive and natural method of determining measurement quality is via the concept of conditional entropy. Conditional entropy is a measure of the remaining uncertainty about one random variable when conditioned on another random variable. For our problem conditional entropy is defined as follows,

$$H(\mathcal{X}|\mathcal{Z}_{\mathcal{S}}) = -\int p(\boldsymbol{x}, \boldsymbol{z}) \log p(\boldsymbol{x}|\boldsymbol{z})\, d\boldsymbol{x} d\boldsymbol{z}, \tag{2.1}$$

where $\mathcal{X}$ is a random variable representing the object state we would like to estimate, and $\mathcal{Z}_{\mathcal{S}}$ is a random variable representing the measurements taken at the sites in $\mathcal{S}$. The conditional entropy is simply the entropy of our object state conditioned on the measurements $\mathcal{Z}_{\mathcal{S}}$.

Decreasing the entropy of $\mathcal{X}$ is the same as decreasing our uncertainty about the current state. Therefore, the natural and intuitive approach would be to attempt to choose $\mathcal{S}$ such that the conditional entropy $H(\mathcal{X}|\mathcal{Z}_{\mathcal{S}})$ is minimized. Or more

rigorously, the optimal set $\mathcal{S}^*$ is defined as follows:

$$\mathcal{S}^* = \underset{\mathcal{S}\subset\mathcal{V}:|\mathcal{S}|=n}{\arg\min}\ H(\mathcal{X}|\mathcal{Z}_\mathcal{S}). \tag{2.2}$$

Unfortunately, finding this optimal subset $\mathcal{S}^*$ that minimizes the conditional entropy has been shown to be an NP-hard problem [13]. Fortunately, several approximate solutions have been proposed. We first present the maximum entropy criteria for comparison, and then we present the mutual information criteria that we utilize in our algorithms.

2.2.2 Maximum Entropy

The maximum entropy criteria is based on the concept that we should choose the set of sensor locations that has the maximum uncertainty about each other, or formally,

$$\mathcal{S}^* = \underset{\mathcal{S}\subset\mathcal{V}:|\mathcal{S}|=n}{\arg\max}\ H(\mathcal{Z}_\mathcal{S}). \tag{2.3}$$

The maximum entropy criteria arises because in some special cases the maximum entropy set will also minimize the conditional entropy of the variables of interest. This is the case when using a subset of sensors to estimate some spatial phenomenon over all of the uninstrumented locations in some region of interest [16]. However, as is pointed out in [22, 16] using this maximum entropy criteria is problematic in practice. We present the maximum entropy criteria here as a basis of comparison to allow us to explain some of the advantages of using the related but superior mutual information criteria.

Seeking the set of sensor locations that is maximally uncertain about each other turns out to be an intuitive approach. We might get the most bang for the buck by taking measurements in the places that we collectively know the least about, *i.e.* where the uncertainty and entropy is the highest. This should ensure the sensors are distributed evenly through the space and avoid situations where we have multiple sensors collecting redundant information. However, just like finding the minimum

conditional entropy subset, finding the maximum entropy subset is NP-hard [13].

Approximation algorithms do exist. The most commonly used algorithm put forth in [18, 8] uses a greedy approach to iteratively select the maximum entropy locations. The reasoning for this approach can be seen clearly by dividing up the objective function using the chain rule for entropy. Let $\mathcal{S}_i = \{s_0, s_1, \ldots, s_i\}$ be the set of selected sensor locations at the ith iteration. Then the entropy $H(\mathcal{Z}_{\mathcal{S}_i})$ of that set of locations can be broken down as follows:

$$H(\mathcal{Z}_{\mathcal{S}_i}) = H(\mathcal{Z}_{s_i}|\mathcal{Z}_{\mathcal{S}_{i-1}}) + \ldots + H(\mathcal{Z}_{s_2}|\mathcal{Z}_{\mathcal{S}_1}) + H(\mathcal{Z}_{s_1}|\mathcal{Z}_{\mathcal{S}_0}). \tag{2.4}$$

Therefore if we start with $\mathcal{S}_0 = \emptyset$, at each step we simply choose the next location that has the highest entropy when conditioned on the set of previous measurements.

$$s_i = \arg\max_s H(\mathcal{Z}_s|\mathcal{Z}_{\mathcal{S}_{i-1}}) \tag{2.5}$$

This algorithm is straightforward and distributes the sensors in a natural and intuitive way. If we assume that sensors are most informative about the region closely surrounding the sensor. This algorithm should tend to place sensors in locations far apart from each other where the least information is available. This should provide good even coverage of the space. However, in practice this can be problematic.

In Figure 2-1 an example of an experimental deployment of temperature sensors is shown [16]. From the figure we can see that this tendency to place sensors far apart from each other frequently results in deployments with a large portion of the sensors along the boundaries of the region. Typically we are not interested in the information outside the boundary or there is simply no information available (*e.g.* a wall). Therefore sensors placed along the boundary are only partially utilized and in a sense "wasted".

This shortcoming of maximum entropy deployments is highlighted in [22] along with some heuristics to mitigate the effect. The problem arises from the fact that the selection criteria, namely maximizing $H(\mathcal{Z}_{\mathcal{S}})$, only considers the selected sensor locations. It doesn't directly consider the state $\mathcal{X}$ that we want to estimate.

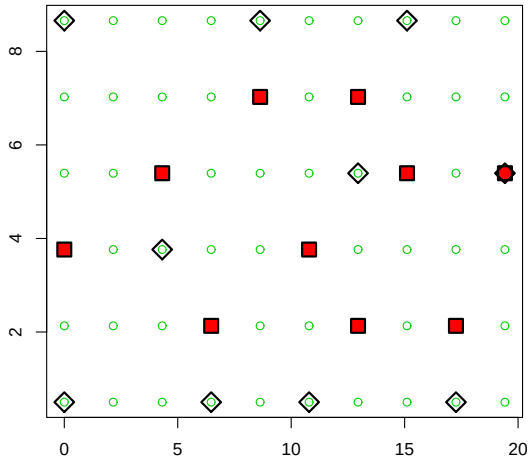

Figure 2-1: Two different deployments of sensor nodes in the same environment. The diamonds are for the maximum entropy based deployment and the squares are for the mutual information based deployment. Figure originally from [16].

2.2.3 Mutual Information Metric

Mutual information is a related metric that overcomes some of the shortcomings of using maximum entropy [4, 14]. Mutual information between the object state $\mathcal{X}$ and the measurements $\mathcal{Z}_\mathcal{S}$ is defined as the expected reduction in entropy of the object state $\mathcal{X}$ when conditioned on the measurements $\mathcal{Z}_\mathcal{S}$. Or formally mutual information is defined as follows:

$$I(\mathcal{X}; \mathcal{Z}_\mathcal{S}) = H(\mathcal{X}) - H(\mathcal{X}|\mathcal{Z}_\mathcal{S}). \tag{2.6}$$

Recall, from Equation (2.2) that our goal is to minimize the conditional entropy $H(\mathcal{X}|\mathcal{Z}_\mathcal{S})$. Since the first term in Equation (2.6) is independent of the measurements $\mathcal{Z}_\mathcal{S}$, selecting the set $\mathcal{S}^*$ that minimizes $H(\mathcal{X}|\mathcal{Z}_\mathcal{S})$, is the same set that will maximize the mutual information $I(\mathcal{X}; \mathcal{Z}_\mathcal{S})$ [10]. So we want to select the set of sensors that maximize the mutual information between the measurements and the object state we

are trying to estimate, or specifically,

$$\mathcal{S}^* = \underset{\mathcal{S}\subset\mathcal{V}:|\mathcal{S}|=n}{\arg\max}\, I(\mathcal{X};\mathcal{Z}_\mathcal{S}) \tag{2.7}$$

$$= \underset{\mathcal{S}\subset\mathcal{V}:|\mathcal{S}|=n}{\arg\max}\, H(\mathcal{X}) - H(\mathcal{X}|\mathcal{Z}_\mathcal{S}). \tag{2.8}$$

As can be seen in Figure 2-1 using mutual information as the selection criteria leads to a more reasonable uniform distribution throughout the interior of the space. This way each sensor is fully utilized and we don't end up wasting sensing capacity along the boundaries. This is because mutual information is directly related to the estimate of the object state $\mathcal{X}$. Whereas the maximum entropy selection criteria in Equation (2.5) ended up only focusing on the entropy of the actual measurement sites. This only indirectly affects the entropy of the object state $\mathcal{X}$.

Mutual information is an ideal metric for our sensor placement problem, because it results in minimizing the conditional entropy of our object state. However, it still poses challenges to use in practice. One of the challenges is the need to have a good model for $P(\mathcal{X})$, which is needed for the entropy calculations. We utilize a particle filter, as described in Section 4.2, for this purpose. Second, like maximum entropy, finding a set of sensor locations that optimizes mutual information has been shown to be an NP-hard problem [13, 14].

Approximation Algorithm

In addition to having a performance advantage over other metrics, mutual information also has several nice mathematical properties that have aided in developing approximation algorithms with good provable bounds on performance. We will cover some of these properties here and introduce a simple approximation algorithm that we will build off in our solution to the mobile sensor tracking problem. The algorithms will be discussed in more rigorous detail in Chapter 5 .

Recently research [11, 14] points out that mutual information has the desirable property of being submodular and monotonic which allows us to use a greedy algorithm that is a $(1 - 1/e)$OPT approximation. In simple terms the approximation

algorithm says we should sequentially select the sensors. At each stage choosing the sensor that has the highest mutual information given the previously selected sensors. So at each stage we would select,

$$s_i = \arg\max_s I(\mathcal{X}; \mathcal{Z}_s | \mathcal{Z}_{\mathcal{S}_{i-1}}) \tag{2.9}$$

$$= \arg\max_s H(\mathcal{X}|\mathcal{Z}_{\mathcal{S}_{i-1}}) - H(\mathcal{X}|\mathcal{Z}_{\mathcal{S}_{i-1}}, \mathcal{Z}_s) \tag{2.10}$$

$$= \arg\max_s H(\mathcal{Z}_s|\mathcal{Z}_{\mathcal{S}_{i-1}}) - H(\mathcal{Z}_s|\mathcal{X}, \mathcal{Z}_{\mathcal{S}_{i-1}}) \tag{2.11}$$

$$= \arg\max_s H(\mathcal{Z}_s|\mathcal{Z}_{\mathcal{S}_{i-1}}) - H(\mathcal{Z}_s|\mathcal{X}), \tag{2.12}$$

where we have removed the conditioning on the measurements from the other sensors $\mathcal{Z}_{\mathcal{S}_{i-1}}$ from the second term in Equation (2.12) because we make the conservative assumption that the measurements are independent when conditioned on the object state $\mathcal{X}$ [27].

Comparing Equation (2.12) with the selection criteria for greedy maximum entropy algorithm in Equation (2.5) we see that the mutual information criteria contains the same term, $H(\mathcal{Z}_s|\mathcal{Z}_{\mathcal{S}_{i-1}})$, which was the sole consideration of the maximum entropy criteria. As previously stated, maximizing this term will tend to favor sensors which are far apart and have high entropy with regard to one another. Notice in Equation (2.12) that the mutual information algorithm contains an additional term, $H(\mathcal{Z}_s|\mathcal{X})$. In order to maximize mutual information we will want to minimize this second term in Equation (2.12). Because of the conditioning on $\mathcal{X}$, minimizing this term will tend to favor sensors that are informative about our object state $\mathcal{X}$. So maximizing mutual information involves the delicate balance of selecting sensors that have information about the object state but don't share information in common with the other previously selected sensor locations. As can be seen in Figure 2-1 this second term helps avoid the problem of pushing all the sensors to the boundaries that maximum entropy suffers from, and is what can give mutual information based deployments performance advantages in practice.

In [19] it was shown that a greedy selection algorithm like the one above will be at

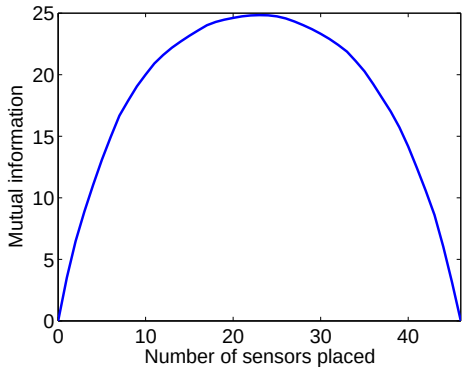

Figure 2-2: Mutual information as a function of the number of sensors placed in the example deployment from Figure 2-1. Figure originally from [16].

least $(1 - 1/e)$OPT if the objective function is monotonic and submodular. In order to show monotonicity and submodularity for this formulation we will rely on the well known "information never hurts" property of entropy, which states that the entropy of $\mathcal{A}$ will never increase given we observe some variable $\mathcal{B}$. More formally we have $H(\mathcal{A}|\mathcal{B}) \leq H(\mathcal{A})$ [7].

For monotonicity we must show that $I(\mathcal{X}; \mathcal{Z}_s|\mathcal{Z}_{\mathcal{S}_{i-1}}) \geq 0, \forall s$. Utilizing the "information never hurts" property we know,

$$I(\mathcal{X}; \mathcal{Z}_s|\mathcal{Z}_{\mathcal{S}_{i-1}}) = H(\mathcal{X}|\mathcal{Z}_{\mathcal{S}_{i-1}}) - H(\mathcal{X}|\mathcal{Z}_{\mathcal{S}_{i-1}}, \mathcal{Z}_s) \geq 0. \tag{2.13}$$

Therefore, each time we add an additional sensor, we will add a nonnegative quantity of information and thus our mutual information metric will be monotonic. This seems straightforward and obvious, but care should be taken to ensure monotonicity because other formulations can result in mutual information metrics that won't always increase monotonically.

A common example can be found in [16] where they are utilizing a set of sensor nodes $\mathcal{S}$ to estimate a spatial phenomenon over the uninstrumented locations $\mathcal{V} \backslash \mathcal{S}$. So they are concerned with the mutual information between the instrumented locations and the uninstrumented locations, $I(\mathcal{V} \setminus \mathcal{S}; \mathcal{S})$. In this scenario, each time a sensor s is added to $\mathcal{S}$, it is also removed from $\mathcal{V} \setminus \mathcal{S}$. This results in a mutual information

function that grows like the one shown in Figure 2-2. Clearly mutual information is not monotonic in this case. However, in [16] they show that as long as the deployment of sensors is small enough mutual information can be considered monotonic (*e.g.* the first half of the plot in Figure 2-2).

Submodularity is informally the same concept as that of diminishing returns. For a submodular set function, the increase from adding an additional member to the set will be inversely proportional to the size of the set. So if we have two sets of sensors $\mathcal{S}$ and $\mathcal{S}'$ where $\mathcal{S} \subseteq \mathcal{S}'$. Then we should gain more information by adding a sensor s to $\mathcal{S}$ than we would if we add it to $\mathcal{S}'$. In general mutual information will not be a submodular function, but under the previously stated assumption that measurements are conditionally independent given the object state, it has been shown that mutual information will be submodular [11, 14]. Since mutual information satisfies the requirements of monotonicity and submodularity [19], the greedy approximation can be guaranteed to be within $(1 - 1/e)\text{OPT}$ or approximately 63% of the optimal deployment.

2.3 Measurement Costs

Up to this point we have only considered one side of the sensor selection problem, that of choosing the most informative sensor locations. However, if this is all that is considered there is an implicit assumption that all of those measurements can be obtained for the same cost. For some applications this may essentially hold true, but can't be assumed to be true in general. In most cases differing costs will be incurred to take and process different measurement. There are computational costs for signal processing, etc. There are communication costs incurred to transmit the data for processing and reporting. And in mobile sensor networks there is a significant cost incurred to move the sensing platform. In static networks the communications cost are usually orders of magnitude higher than the cost of computation [21, 20]. In which case it is standard to only focus on the most significant cost and ignore the others. In this work we are primarily focusing on utilizing mobile nodes in a tracking exercise.

Since the energy required for mobility is significantly higher than that required for communications or computation we will focus solely on the cost of movement.

If movement costs were ignored then the solution to our tracking scenario would be to move all sensors as close as possible to the target being tracked. This is obviously a waste and would reduce the effectiveness of the sensor network. In a tracking scenario these costs could be time or energy. If it takes too long to get to a specific location then the opportunity to get the needed measurement will be lost. Naturally moving around unnecessarily will also waste energy or fuel and limit the lifetime of the network. Typically there will be a clear constraint on costs that a robot or sensor platform can incur. It may be limited by amount of energy in its battery, it may be limited by the total distance it can travel, or by the amount of time it has to finish the tasks. We will attempt to solve the challenge of selecting the most informative locations for the mobile sensors while subject to these constraints on the cost of moving. The specific modeling of costs and problem formulation will be discussed in Chapter 4.

2.4 Multistep Planning

There are several approaches to planning the movements of these mobile sensors. One method which we will look at would be to take a myopic solution that only considers the next time step. This can be done in a straightforward manner by treating the next time step as a deployment of a static network and simply deploying the nodes optimally according to the current conditions. This approach can directly leverage some of the previously mentioned greedy algorithms with little modification.

These myopic approaches can be problematic for a mobile sensor network, particularly when cost constraints are also being considered. As previously mentioned a mobile sensor network is typically more sparsely distributed and relies on mobility to provide adequate coverage. If we only consider the next time step a node will usually find that moving will incur a cost and no beneficial information is gained, so the optimal thing to do is to sit still. So there will be a bias towards doing nothing

and the network can fail to accomplish its goals.

Planning ahead is meant to remedy this by identifying opportunities where moving early on will incur some cost that is offset by obtaining information we need later on. We will plan ahead over a fixed time horizon by using an approximate dynamic programming approach that builds off of work done for static networks in [26, 27]. Unfortunately, in its pure form the dynamic program is intractable. The complexity grows exponentially in the number of nodes and in the length of the time horizon. So even for a small number of nodes and a very short planning horizon the computation will take too long for it to be actionable in a tracking exercise. In the original work in [27] the structure of the problem is slightly different than ours. They were selecting which subset of sensors to activate in a static network, and they were also selecting a leader node for data fusion and processing. Approximations were introduced that rely on the static nature of the network and the structure of the leader node selection problem to prune down the possible plans, making the problem tractable. However, these approximations are not applicable without the structure of the leader node selection or if the nodes are moving. Our main contributions are new approximations to this dynamic program that are appropriate for our mobile sensor network problem. The key insight is to utilize a greedy selection over whole paths instead of individual locations.

Chapter 3

Related Work

Our contributions in this thesis build off of several recent developments in the sensor network and tracking communities. Here we will highlight related work, in particular those contributions that provide the framework for the algorithms we developed for the mobile sensor tracking problem. We look at work loosely organized into three areas that are critical for the problem of tracking with mobile sensors, namely sensor management, bounded costs, and planning.

3.1 Information Theoretic Sensor Management

Recent improvements in power efficiency, miniaturization, and computation have been mirrored by increased interest and research on sensor networks and sensor management. A great deal of work has been done on optimal sensor placement in static sensor networks. As previously discussed most of the information theoretic metrics, such as conditional entropy and mutual information, lead to NP-hard optimization problems. That is why even though information theoretic approaches to sensor placement are popular, they have consisted of mostly greedy approaches and heuristics.

Guestrin and Krause [11, 14] were the first to present an elegant greedy algorithm for the sensor selection problem with nice provable bounds. As discussed in Section 2.2.3, they were able to utilize the submodularity of mutual information to show that their simple greedy selection will be at least $(1 - 1/e)$ times the optimal. However,

this greedy algorithm doesn't automatically extend to the case where we do multistep look ahead planning like we would for tracking with mobile sensors. We developed a modified version of this greedy algorithm that works appropriately with mobile sensors and while planning over a finite time horizon.

Hoffmann and Tomlin [12] present a mutual information based algorithm for mobile sensors that utilizes particle filters to avoid dependence on Gaussian assumptions or extended Kalman filters. They mitigate the complexity by only considering single node and pairwise node mutual information calculations. This is the same as implicitly assuming the sensor measurements have zero mutual information between each other and thus can operate independently of each other. This can allow the network to function in a distributed rather than a centralized fashion and will scale well. This is one assumption that we consider and implement in our simulation for comparison.

3.2 Bounded Cost Optimizations

Much of the work in information based sensor management tends to neglect the issue of cost and focuses on optimizing for information only. For tracking with mobile sensors, the movement costs must be considered if the results are going to be feasible in practice.

Zhao *et al.* [29] discusses a variety of different information based metrics for use in tracking in static sensor networks. They propose a distributed algorithm where each node chooses which of its neighbors to hand off to based on local estimates of information utility. They do acknowledge the costs of communication and computation but the costs are not addressed directly in their formulation.

Krause and Guestrin [15] extended their previous work to also consider communication costs while solving for optimal sensor placements for a static network. They present a clustering algorithm that utilizes their greedy algorithm and then optimizes costs of over the clusters. They prove that they can achieve similar bounds under this cost constrained scenario as well.

Williams and Fisher [26, 27] develop an approximate dynamic programming ap-

proach to solve the constrained cost sensor management problem. They develop a dynamic program that allows the information utility and communications costs budget to be broken down and optimized on a per-stage basis. Interestingly this also provides a solution to the dual problem of having a fixed estimation quality requirement and then optimizing the costs while subject to that requirement. In its standard form the dynamic program is intractable. In addition to the choosing which sensors to activate at each time step, they also select a leader node for data fusion and processing. The authors introduce an approximation that prunes down the possible plans to make the problem more tractable. However, this pruning technique relies on the particular structure of the leader node selection problem and the static nature of the network, and therefore doesn't carry over to our mobile sensor network scenario. They demonstrate their algorithm with a specific implementation in a tracking exercise. We also use a dynamic programming approach for our tracking problem that builds directly off of their work.

3.3 Finite Horizon Planning

There are obvious advantages to being able to do look ahead planning in sensor network management. However, the computation costs are generally prohibitive because the problem grows quickly with the number of sensors and the length of the time horizon that we are planning over. So even for a small number of sensors and even two or three steps ahead the amount of computation can be prohibitive.

Chhetri *et al.* [6] propose a solution for sensor scheduling in a tracking problem. They suggest using a brute force approach to examine all of the possible plans. The experiments only extend from one to three time steps ahead.

Roy and Earnest [23] suggest a mutual information based approach to search for a target with mobile sensors. They develop a method for creating a multistep look ahead plan by clustering over the particles in a particle filter and attempting to find the most informative path over those clusters.

As mentioned previously Williams and Fisher [26, 27] propose utilizing a dynamic

programming approach which allows them to create a sensor management plan over a finite time horizon. The approximations they developed allow them to plan over much longer horizons. They show experiments with plans extending over 60 time steps. In [26, 28] they also extend some of the performance guarantees developed by Guestrin and Krause in [14] to sequential planning problems. One of the approximations is to utilize a greedy sensor subset selection at each time step. However, as we discuss in Chapter 5 this tends to be less than ideal when planning motion paths for mobile sensors. We seek methods that optimize over the paths rather than doing greedy optimizations at each time step.

Singh *et al.* [24] extends the work of Chekuri and Pal [5] to the multi-robot path planning domain. They present an algorithm for finding the most informative paths for a multi-robot team subject to bounded path costs. They demonstrate their algorithm with experiments carried out for lake and river monitoring. This is one of the few works that like us tries to find the most informative paths for a set of mobile sensors while being subject cost constraints. They accomplish this by recursively breaking down the paths and optimizing over the subpaths. They choose to take an offline planning approach. For Gaussian models the mutual information only depends on the covariance and therefore can be calculated offline. So the complete plan is produced ahead of time and then carried out. This offline approach isn't appropriate for a tracking exercise because of the dynamics of the object being tracked are unknown.

3.4 Comparison With Our Work

The goal of our work is to create real time plans for the most informative paths for a set of mobile nodes to follow while subject to a fixed budget for path costs. The work by Singh *et al.* [24] is the only one of the related works that attempts to take on almost all of the same challenges in terms of mobility, cost constraints, look ahead planning, and information theoretic objectives. However, we are taking on a tracking problem which poses a different set of challenges, and as such we have

a different structure. In Chapter 4 we will present our formulation of the tracking problem. Then in Chapter 5 we present the dynamic programming structure that we use, which closely follows the work of Williams and Fisher [27]. We also present several greedy approximations that build off of the greedy sensor selection algorithm and bounds developed by Guestrin and Krause [11, 14, 16].

Chapter 4

Problem Formulation

Here we will present the particular problem formulation we will use for object tracking with mobile sensor nodes. It should be noted that our algorithms are generally applicable to estimation problems that employ mobile sensors, and don't depend on the specific structure or features of tracking or this particular formulation. We wish to present a concrete formulation now to aid in clarity and understanding as we present our algorithms, and for use in the simulation presented in Chapter 6. Our contributions in this thesis are an extension of the work done in [26, 27]. As such, we utilize the same formulation for object dynamics, sensor modeling, and the particle filter estimator as that found in [27]. The primary distinction in problem structure being that they utilize a communication constrained static sensor network and we will use motion constrained mobile sensor nodes, which will require the use of different approximations in the algorithms presented in Chapter 5.

4.1 Modeling

4.1.1 Target Object Model

Our goal is to utilize the mobile sensor network to create and maintain an estimate of the current state of the object being tracked. Let $\boldsymbol{x}_k \in \mathcal{X}$ be the state of the object being tracked at time k. For our specific experiments in this thesis we will track the

position and velocity in two dimensions of the target object, *i.e.* $\boldsymbol{x}_k = [p_x\, v_x\, p_y\, v_y]^T$. Velocity of the object is modeled as a random walk with constant diffusion strength q. Positions are simply the integral of the velocities. It is assumed that the object dynamics progess according to a linear Gaussian model, as follows:

$$\boldsymbol{x}_{k+1} = \mathbf{F}\boldsymbol{x}_k + \boldsymbol{w}_k \tag{4.1}$$

where $\boldsymbol{w}_k \sim \mathcal{N}\{\boldsymbol{w}_k; \mathbf{0}, \mathbf{Q}\}$[1] is a Gaussian white noise process, and $\mathbf{F}$ and $\mathbf{Q}$ are known matrices. If we let T be the sampling interval, then our discrete-time model defines these matrices as follows [17]:

$$\mathbf{F} = \begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{Q} = q \begin{bmatrix} \frac{T^3}{3} & \frac{T^2}{2} & 0 & 0 \\ \frac{T^2}{2} & T & 0 & 0 \\ 0 & 0 & \frac{T^3}{3} & \frac{T^2}{2} \\ 0 & 0 & \frac{T^2}{2} & T \end{bmatrix}. \tag{4.2}$$

4.1.2 Sensor Model

We will now define the measurement model used for our mobile sensor network. Let $\mathcal{S} = \{1, 2, \ldots, N_s\}$ be the set of N_s mobile sensor nodes. Let $\boldsymbol{z}_k^s$ be a measurement taken at time k from sensor $s \in \mathcal{S}$. Then the measurement model for sensor s is

$$\boldsymbol{z}_k^s = \boldsymbol{h}(\boldsymbol{x}_k, s) + \boldsymbol{v}_k^s \tag{4.3}$$

where $\boldsymbol{v}_k^s \sim \mathcal{N}\{\boldsymbol{v}_k^s; \mathbf{0}, \mathbf{R^s}\}$ is a Gaussian white noise process. Also $\boldsymbol{v}_k^s$ is independent of $\boldsymbol{w}_k$ and $\boldsymbol{v}_k^j$ for $j \neq s$. Each sensor has a known noise covariance $\mathbf{R}^s$. For the experiments in this thesis we use range measurements, represented by a known nonlinear function $\boldsymbol{h}(\cdot, s)$, for each sensor s. We define $\boldsymbol{h}(\cdot, s)$ as follows:

$$\boldsymbol{h}(\boldsymbol{x}_k, s) = \frac{a}{\|\mathbf{L}\boldsymbol{x}_k - \boldsymbol{y}_k^s\|_2^2 + b}. \tag{4.4}$$

[1] We use the notation $\boldsymbol{x} \sim \mathcal{N}\{\boldsymbol{x}; \boldsymbol{\mu}, \mathbf{P}\}$ to indicate that the $\boldsymbol{x}$ is normally distributed, that is $p(\boldsymbol{x}) = \mathcal{N}\{\boldsymbol{x}; \boldsymbol{\mu}, \mathbf{P}\} = |2\pi\mathbf{P}|^{-1/2} \exp\{-1/2(\boldsymbol{x} - \boldsymbol{\mu})^T \mathbf{P}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})\}$

The terms a and b can be adjusted to model the specific SNR (signal to noise ratio) of each sensor s, and the rate at which SNR decreases with distance. The $\|\mathbf{L}\boldsymbol{x}_k - \boldsymbol{y}_k^s\|_2^2$ expression accounts for the actual distance from the sensor to the target object being tracked. The first term $\mathbf{L}\boldsymbol{x}_k$ represents the position of the target at time k, where $\mathbf{L}$ is simply the matrix that selects the target object's position from the object's state $\boldsymbol{x}_k$. The second term $\boldsymbol{y}_k^s$ represents the position of sensor s at time k.

While the nonlinear measurement model is used to form the actual estimates, we utilize a simplified linear approximation during the planning process. Specifically the linearized model is only used to estimate the expected information gain. When the actual measurements are taken, precise inference is implemented via a particle filter as described in Section 4.2. While we don't rely on any specific structure of the measurement model, we do assume that it is sufficiently smooth that it can be approximated around a nominal point $\boldsymbol{x}^0$ with a Taylor series expansion as follows:

$$\boldsymbol{z}_k^s \approx \boldsymbol{h}(\boldsymbol{x}^0, s) + \mathbf{H}^s(\boldsymbol{x}^0)(\boldsymbol{x}_k - \boldsymbol{x}^0) + \boldsymbol{v}_k^s \tag{4.5}$$

$$\mathbf{H}^s(\boldsymbol{x}^0) = \nabla_{\boldsymbol{x}} \boldsymbol{h}(\boldsymbol{x}, s)|_{\boldsymbol{x}=\boldsymbol{x}^0}. \tag{4.6}$$

For the specific range measurement in Equation (4.4) that we use in our experiments the linearization will be:

$$\mathbf{H}(\boldsymbol{x}^0) = \frac{-2a}{(\|\mathbf{L}\boldsymbol{x}^0 - \boldsymbol{y}_k^s\|_2^2 + b)^2}(\mathbf{L}\boldsymbol{x}^0 - \boldsymbol{y}_k^s)^T\mathbf{L}. \tag{4.7}$$

As discussed in Chapter 5 this linear approximation will allow for a simplification to the planning algorithm for future sensor paths.

4.1.3 Cost Model

Our algorithm is indifferent to the specific type of costs, but we do assume that there is some cost associated with taking each measurement. That cost could be the time to acquire the data, the energy or bandwidth to communicate the measurements, the computational costs of processing the data, or in our case the cost of moving a sensor

node. We limit ourselves to movement costs because the energy expenditures for moving sensor nodes will be much higher than the energy required for computation and communications. Also it is directly related to the placement of mobile sensors which is the problem we are primarily interested in.

At each time step we have N_s mobile sensors that will need to be placed. As previously mentioned, at any moment we can take a snapshot of our mobile sensor network, and we can treat it as a static network consisting of real and virtual sensor nodes. The real nodes represent the current locations of our mobile sensing platforms. For each real sensor node s_k we create a discrete set of N_v virtual nodes $\mathcal{V}_{k+1}^s$ representing all of the possible locations that the real sensor s_k can go to at the next time step $k+1$. Obviously the locations in $\mathcal{V}_{k+1}^s$ will be constrained by the nature of the mobile sensor node and the controls available to it and the current location of s_k at time k. Our solution relies on a discrete dynamic programming approach and thus we require the $\mathcal{V}_{k+1}^s$ be a finite discrete set of locations.

For our simulations we chose to discretize the space in a uniform hexagonal grid. Therefore at each time step a node s_k can choose to remain at its current location or to move to one of its six neighboring virtual nodes arranged in a hexagonal grid. Thus for our simulations $\mathcal{V}_{k+1}^s$ consisted of $N_v = 7$ locations, representing the six nearest neighbors of s_k on the hexagonal grid and the current location of s_k (representing the control choice to stand still).

We assume that the energy cost C_k^s of moving a node s_k to its current location $\boldsymbol{y}_k^s$ at time k from its previous location $\boldsymbol{y}_{k-1}^s$ at time $k-1$ is proportional to the distance between the two locations, specifically

$$C_k^s \propto \|\boldsymbol{y}_k^s - \boldsymbol{y}_{k-1}^s\|_2 \tag{4.8}$$

where $\boldsymbol{y}_k^s$ is constrained to be the location of one of the virtual nodes in $\mathcal{V}_k^s$, *i.e.* the virtual nodes in the neighborhood of s_{k-1}. Note that this implies that the cost of remaining in the same place will be zero. In our planning algorithm we will also be interested in costs over paths of virtual nodes. We assume that the path costs are

likewise proportional to the length of the path and therefore simply equal to the sum of the costs of the individual segments.

Other more complex cost structures could also be used. For example there may be different amounts of energy required when changing direction or starting and stopping. Or perhaps, communication costs will be significant when compared to mobility costs, or they could be highly sensitive to node to node distances. So a weighted sum of the movement and communication costs could be used. However, the structure of the solution would be the same. The only requirement is that we need to be able assign some cost to each discrete move. While it is true that the structure may not change, the complexity could change because considering additional costs can increases the number of options that must be evaluated at each stage.

4.2 Tracking Estimator

Our planning algorithm for sensor placement uses mutual information as the objective function, and therefore is not dependent on the specific estimator that the measurements are fed to. A variety of estimators could be used that would perform better by having the optimal set of measurements in terms of mutual information. For our tracking simulations we chose to use a particle filter to form our estimate of the object state of the target. A particle filter is particularly appropriate for this application. The fact that each individual sensor only focuses on its local region and each has a nonlinear measurement model as described in Equation (4.4) can result in significant multimodality. A particle filter estimator is well suited to handle such multimodal behavior.

While the algorithms are not dependent on this particular estimator, we present the specific details of our implementation for completeness and clarity. Let the conditional pdf of the target object state $\boldsymbol{x}_k$ conditioned on the previous measurements $\boldsymbol{z}_{0:k}$ received up to and including the time k be denoted as $p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k})$. Then we

approximate $p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k})$ with a set of N_p weighted samples (or particles) as follows:

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k}) \approx \sum_{i=1}^{N_p} w_k^i \delta(\boldsymbol{x}_k - \boldsymbol{x}_k^i). \tag{4.9}$$

Next we want to infer what this pdf will be at the next time step. Specifically, we want to calculate $p(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{0:k+1})$. This is accomplished with the standard Sequential Importance Sampling (SIS) algorithm [1] with resampling at each time step. This algorithm prescribes how to produce a new sample $\boldsymbol{x}_{k+1}$ for time $k+1$ for each of the current samples $\boldsymbol{x}_k^i$. We get $\boldsymbol{x}_{k+1}$ by sampling the distribution $q(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k^i, \boldsymbol{z}_{k+1})$, which is obtained from the linearization of the measurement model from Equation (4.3) for $\boldsymbol{z}_{k+1}$. We make the linearization around the point $\mathbf{F}\boldsymbol{x}_k^i$ using Equation (4.5). We can obtain this distribution, $q(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k^i, \boldsymbol{z}_{k+1})$, directly using an extended Kalman filter equations [17]. At time $k+1$ we have:

$$p(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k^i) = \mathcal{N}(\boldsymbol{x}_{k+1}; \mathbf{F}\boldsymbol{x}_k^i, \mathbf{Q}). \tag{4.10}$$

Then we update the distribution with the next measurement $\boldsymbol{z}_{k+1}$ using the extended Kalman filter update equation to get

$$q(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k^i, \boldsymbol{z}_{k+1}) = \mathcal{N}(\boldsymbol{x}_{k+1}; \hat{\boldsymbol{x}}_{k+1}^i, \mathbf{P}_{k+1}^i) \tag{4.11}$$

where

$$\hat{\boldsymbol{x}}_{k+1}^i = \mathbf{F}\boldsymbol{x}_k^i + \mathbf{K}_{k+1}^i[\boldsymbol{z}_{k+1} - \boldsymbol{h}(\mathbf{F}\boldsymbol{x}_k^i, s)] \tag{4.12}$$

$$\mathbf{P}_{k+1}^i = \mathbf{Q} - \mathbf{K}_{k+1}^i \mathbf{H}^s(\mathbf{F}\boldsymbol{x}_k^i)\mathbf{Q} \tag{4.13}$$

$$\mathbf{K}_{k+1}^i = \mathbf{Q}\{\mathbf{H}^s(\mathbf{F}\boldsymbol{x}_k^i)\}^T[\mathbf{H}^s(\mathbf{F}\boldsymbol{x}_k^i)\mathbf{Q}\{\mathbf{H}^s(\mathbf{F}\boldsymbol{x}_k^i)\}^T + \mathbf{R}^s]^{-1}. \tag{4.14}$$

Then a new particle $\boldsymbol{x}_{k+1}^i$ is created by sampling from the distribution in Equation

(4.11). The weight w_{k+1}^i for the new particle is calculated using

$$w_{k+1}^i = c w_k^i \frac{p(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}^i) p(\boldsymbol{x}_{k+1}^i|\boldsymbol{x}_k^i)}{q(\boldsymbol{x}_{k+1}^i|\boldsymbol{x}_k^i, \boldsymbol{z}_{k+1})} \tag{4.15}$$

where c is a constant used to normalize the weights and ensure that they sum to one (*i.e.* $\sum_{i=1}^{N_p} w_{k+1}^i = 1$), and

$$p(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}^i) = \mathcal{N}\{\boldsymbol{z}_{k+1}; \boldsymbol{h}(\boldsymbol{x}_{k+1}^i, s), \mathbf{R}^s\}. \tag{4.16}$$

The final resulting approximation for the pdf of $\boldsymbol{x}_{k+1}$ conditioned on all the measurements $\boldsymbol{z}_{0:k+1}$ is

$$p(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{0:k+1}) \approx \sum_{i=1}^{N_p} w_{k+1}^i \delta(\boldsymbol{x}_{k+1} - \boldsymbol{x}_{k+1}^i). \tag{4.17}$$

At any time we can produce a Gaussian approximation of the current estimate of the target object's state. This is done by producing a moment-matched Gaussian representation of the particle distribution by calculating the mean and covariance from the particles as follows:

$$\boldsymbol{\mu}_k = \sum_{i=1}^{N_p} w_k^i \boldsymbol{x}_k^i \tag{4.18}$$

$$\mathbf{P}_k = \sum_{i=1}^{N_p} w_k^i (\boldsymbol{x}_k^i - \boldsymbol{\mu}_k)(\boldsymbol{x}_k^i - \boldsymbol{\mu}_k)^T. \tag{4.19}$$

Chapter 5

Approximate Dynamic Programming Algorithms

There a numerous challenges that must be addressed in order to provide a solution to the tracking problem laid out in the previous chapter. First, there are complexity challenges associated with the chosen objective function, namely mutual information. For example finding an optimal subset of n sensor locations that maximizes mutual information is an NP-complete problem [13, 14]. Second, we wish to optimize mutual information subject to constraints on the cost. This dual optimization problem adds to the complexity required of a solution. Third, we want a solution that provides a plan over a multistep time horizon. This will in general significantly increase the computational load and lead to tractability challenges.

A dynamic programming approach will very naturally accommodate the cost constrained optimization problem as well as the dynamic multistep planning problem we are interested in [3]. However, this will only provide us with a structure for the solution. In its pure form the dynamic program will not be tractable for anything more than a few sensors even for single time step planning.

First we will present our dynamic programming approach to the tracking problem. We follow closely the formulation for the dynamic program used in [26, 27]. Their problem focuses on sensor selection in a communication constrained static sensor network. Their dynamic program is designed to solve for which sensors should be

activated at each time step and which sensor should act as the leader node for data fusion and processing. We make appropriate changes to reflect our use of a mobile sensor network. We are not concerned with the leader node selection problem at all, and we want to select a sensor location for all of our real sensors not just some subset of them.

Next we will present our main contributions, which are some specific approximations that we developed to create a tractable solution to this dynamic program. The approximations used for static sensor networks and for the leader node selection problem don't carry over to our network with mobile sensors. We will present a set of new approximation algorithms that allow a tractable solution to this dynamic program to be found for mobile sensor nodes. We consider two sets of approximations. One relies on decoupling the sensor nodes and the other relies on a greedy selection of sensor paths. We also consider multiple versions that only plan one step ahead or that create a multistep plan over some fixed horizon of N steps.

5.1 Dynamic Programming Approach

A dynamic programming approach provides the framework to produce a plan that will optimize the system performance according to the objective function we choose. In order to find an efficient approximate solution to the dynamic program it is generally useful to formulate the objective function in such a way that it can be broken down into a per-stage cost or reward for each stage of the decision tree. Then the overall objective function can be expressed as the sum of the per-stage costs. Additionally we can incorporate a per-stage cost constraint term that will allow us to optimize our objective function subject to the constraint on cost.

5.1.1 Objective Function

We will start with our previously stated objective of finding the subset of sensor locations that will maximize mutual information. We will break this down into per-sensor and per-stage formulations. Then we will augment this objective function

with our per-stage costs to produce the final augmented objective function. This final objective function will allow us to consider for each control decision the trade off between being rewarded with additional information and the cost incurred to obtain that information.

First let us consider the mutual information objective for the case of a set of sensors $\mathcal{S}_k$, where the set $\mathcal{S}_k$ is the set of real sensor nodes at time k. As discussed in Section 4.1.3, at the time step $k-1$ every real sensor $s_{k-1} \in \mathcal{S}_{k-1}$ will only be allowed to move to one of the locations of the virtual nodes associated with s_{k-1}, specifically those in $\mathcal{V}_k^s$. We will use the notation $s_k \in \mathcal{V}_k^s$ to indicate this constraint on the locations of s_k. Then according to the discussion on mutual information in Section 2.2 our objective at time k should be to maximize the mutual information between the current measurements $\boldsymbol{z}_k^{\mathcal{S}_k}$ and the target object state $\boldsymbol{x}_k$ conditioned on all the previous measurements, where $\boldsymbol{z}_k^{\mathcal{S}_k}$ are the set of measurements taken from the sensors in $\mathcal{S}_k$ at time k. Or more formally,

$$\mathcal{S}_k = \underset{\mathcal{S}_k : s_k \in \mathcal{V}_k^s}{\arg\max}\, I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1}) \tag{5.1}$$

where $\boldsymbol{z}_{0:k-1}$ are all of the previous measurements from all sensors. The complexity of this optimization problem arises from the fact that we must compute $I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1})$ for every possible $\mathcal{S}_k$. The number of possibilities for $\mathcal{S}_k$ grows exponentially with the number of sensors N_s and the number of discrete control choices N_v (*i.e.* the number of virtual nodes) each mobile sensor node is allowed to make. In general there will be $O(N_v^{N_s})$ possible sets to consider for $\mathcal{S}_k$.

Because of the additive chain rule for entropy, we can break down the mutual information objective in Equation (5.1) into the sum of the contributions made by each individual sensor in $\mathcal{S}_k$, as follows:

$$I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1}) = \sum_{j=1}^{N_s} I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k^j} | \boldsymbol{z}_{0:k-1}, \boldsymbol{z}_k^{\mathcal{S}_k^{1:j-1}}) \tag{5.2}$$

where $\mathcal{S}_k^j$ is the jth element of $\mathcal{S}_k$ and $\mathcal{S}_k^{1:j-1}$ is the set of the first $j-1$ sensors in

$\mathcal{S}_k$. This formulation is equivalent to the original and doesn't actually change the complexity of the optimization at all. However, this additive form lends itself to the same greedy approximation algorithm discussed in Section 2.2.3 for the maximum mutual information sensor subset selection problem. Which is precisely one of the single-step algorithms we will consider in the next section.

However, we are also interested in the more general problem of optimizing for mutual information over a fixed time horizon of N steps. In this case we are interested in not only the mutual information between the current measurements and the currents object state but also the mutual information between future object states and future measurements. Specifically in the multistep case our object function becomes,

$$I(\boldsymbol{x}_{k:k+N-1}; \boldsymbol{z}_{k:k+N-1}^{\mathcal{S}_{k:k+N-1}} | \boldsymbol{z}_{0:k-1}) \tag{5.3}$$

where with a slight abuse of notation $\boldsymbol{z}_{k:k+N-1}^{\mathcal{S}_{k:k+N-1}}$ is substituted for the sets of all future measurements, $\{\boldsymbol{z}_k^{\mathcal{S}_k}, \ldots, \boldsymbol{z}_{k+N-1}^{\mathcal{S}_{k+N-1}}\}$. This objective can be broken into substages for the selection of each sensor as we did in Equation (5.2). It can also be broken down additionally into substages for each time step as follows:

$$I(\boldsymbol{x}_{k:k+N-1}; \boldsymbol{z}_{k:k+N-1}^{\mathcal{S}_{k:k+N-1}} | \boldsymbol{z}_{0:k-1}) = \sum_{i=k}^{k+N-1} \sum_{j=1}^{N_s} I(\boldsymbol{x}_i; \boldsymbol{z}_i^{\mathcal{S}_i^j} | \boldsymbol{z}_{0:i-1}, \boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}}) \tag{5.4}$$

where time steps are indexed with i and sensors are indexed with j. Again this formulation allows us to isolate the contribution from each sensor at each time step to the total mutual information from all sensors at all times. However, it does not reduce the overall complexity, which will now grow exponentially with N as well, specifically there are $O(N_v^{N_s N})$ possibilities to evaluate.

5.1.2 Constrained Dynamic Programming

Currently the objective function only considers part of our ultimate goal, that of choosing the most informative sensor locations. We also wish to consider the trade off in terms of mobility costs to reach those sensor locations. A standard approach to

handle this trade off is to optimize for mutual information subject to a constraint on the movement costs. First we will present in general terms the constrained dynamic programming approach that can solve this constrained optimization problem. Then we will show our specific implementation for the objective function and costs we previously discussed. Our approach is derived directly from that used in [27], except we don't need to consider the leader node selection problem and we utilize mobile sensor nodes.

General Approach

Ultimately our objective function and cost constraints will be unified into a single cost function that we will seek to minimize over an N step rolling time horizon. The state of the dynamic program will consist of the current pdf of the target object state conditioned on the previous measurements and the previous controls. We define the conditional belief state as $\mathbb{X}_k \triangleq p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k-1})$. The control taken at each time k is $u_k = \mathcal{S}_k$, where $\mathcal{S}_k$ is again the set of sensor locations utilized at time k. Let $\mu_k(\mathbb{X}_k)$ be the control policy for time k, and $\pi_k = \{\mu_k, \ldots, \mu_{k+N-1}\}$ be the set of policies over the N-step planning horizon. We want to find the control policy that gives the optimal solution to the following constrained minimization problem:

$$\min_{\pi} E\left[\sum_{i=k}^{k+N-1} g(\mathbb{X}_i, \mu_i(\mathbb{X}_i))\right] \tag{5.5}$$

$$\text{s.t. } E\left[\sum_{i=k}^{k+N-1} G(\mathbb{X}_i, \mu_i(\mathbb{X}_i))\right] \leq M \tag{5.6}$$

where $g(\mathbb{X}_i, \mu_i(\mathbb{X}_i))$ is the per-stage cost function that we wish to minimize and $G(\mathbb{X}_i, \mu_i(\mathbb{X}_i))$ is the per-stage contribution to the constraint function. As in [27], we can handle this constrained optimization through a Lagrangian relaxation, which is a standard approximation approach for discrete optimization problems. This results

in the following Lagrangian function:

$$L_k(\mathbb{X}_k, \pi_k, \lambda) = E\left[\sum_{i=k}^{k+N-1} g(\mathbb{X}_i, \mu_i(\mathbb{X}_i)) + \lambda\left(\sum_{i=k}^{k+N-1} G(\mathbb{X}_i, \mu_i(\mathbb{X}_i)) - M\right)\right]. \quad (5.7)$$

We can now do a standard unconstrained optimization of this Lagrangian function. The optimization of the Lagrangian function over the primal variable π_k gives us the following dual function:

$$J_k^D(\mathbb{X}_k, \lambda) = \min_{\pi_k} L_k(\mathbb{X}_k, \pi_k, \lambda). \quad (5.8)$$

We then solve the dual optimization problem by maximizing the dual function over $\lambda \geq 0$,

$$J_k^L(\mathbb{X}_k) = \max_{\lambda \geq 0} J_k^D(\mathbb{X}_k, \lambda). \quad (5.9)$$

This dual optimization problem can be solved using a standard subgradient method [3]. Like [27] we take advantage of the fact that we are using a rolling time horizon to avoid making evaluations for many different values of the dual variable λ. We make a plan for the next N steps then we only take one step and plan again for the next N steps. Since we only take one step before replanning we can assume the value of λ will not change significantly between each planning stage. Therefore we can plan using a single value of λ then make a simple update to λ, as described in [27], before the next planning stage.

Notice that the dual function $J_k^D(\mathbb{X}_k, \lambda)$ in Equation (5.8) has the same form as an unconstrained dynamic program, but the per-stage cost has been modified so that the original constraint is incorporated into the per-stage cost function, with the following structure:

$$\bar{g}(\mathbb{X}_k, u_k, \lambda) = g(\mathbb{X}_k, u_k) + \lambda G(\mathbb{X}_k, u_k). \quad (5.10)$$

This ability to capture the per-stage objective and per-stage constraint in a single cost function is key to the approximation algorithms that will follow later. In its pure form this dynamic program will be intractable because our problem grows exponentially in

N_s, and N, specifically there are $O(N_v^{N_s N})$ possible solutions that would all need to be evaluated. Fortunately the structure in Equation (5.10) will allow us to greedily select sensor locations by optimizing on a per-stage basis and thus significantly reduce the complexity.

Constrained Mobility Formulation

We can now show the specific implementation of this dynamic programming approach for our problem of tracking with with energy constrained mobile sensors. From our discussion in Section 5.1.1, we know that at each stage we wish to select a set of sensor locations $\mathcal{S}_k$ that will maximize mutual information. So from Equation (5.2) our per-stage objective function will be

$$g(\mathbb{X}_k, u_k) = -I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1}) \tag{5.11}$$

$$= -\sum_{j=1}^{N_s} I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k^j} | \boldsymbol{z}_{0:k-1}, \boldsymbol{z}_k^{\mathcal{S}_k^{1:j-1}}). \tag{5.12}$$

Notice we use the negative of mutual information because our dynamic program formulation was structured to minimize a cost function subject to a constraint. So our objective "cost" function is the negative of our reward, mutual information. It would have been equally valid to formulate the dynamic program as a maximization of a reward function subject to the negative of the current constraint. Since our reward is the information we obtain from the measurements taken by the sensors in $\mathcal{S}_k$ at their current locations, the associated cost will be the cost to move from their previous locations in $\mathcal{S}_{k-1}$ to their current locations. Our per-stage constraint contribution will simply be the cost associated with moving all the sensors from their locations during the previous time step $k-1$ to their current locations,

$$G(\mathbb{X}_k, u_k) = \sum_{j=1}^{N_s} C_k^j \tag{5.13}$$

We can now substitute the per-stage cost and per-stage constraint into Equation

(5.10) to get the augmented per-stage cost function of

$$\bar{g}(\mathbb{X}_k, u_k, \lambda) = -\sum_{j=1}^{N_s} I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k^j} | \boldsymbol{z}_{0:k-1}, \boldsymbol{z}_k^{\mathcal{S}_k^{1:j-1}}) + \lambda \left(\sum_{j=1}^{N_s} C_k^j \right) \tag{5.14}$$

$$= \sum_{j=1}^{N_s} \left[-I \left(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k^j} | \boldsymbol{z}_{0:k-1}, \boldsymbol{z}_k^{\mathcal{S}_k^{1:j-1}} \right) + \lambda C_k^j \right]. \tag{5.15}$$

Notice that using the formulation in Equation (5.15) we can calculate the augmented cost for every individual sensor selection. This allows us to balance the estimation quality against motion costs for each individual sensor placement at each time step. This is the basis of our greedy optimizations. We can greedily choose the sensor with the lowest augmented cost conditioned on the previously placed sensors.

To properly solve the unconstrained dual optimization problem in Equation (5.8) (for a specific value of the Lagrange multiplier λ), we can use the following recursive dynamic programming equation:

$$J_i^D(\mathbb{X}_i, \lambda) = \min_{u_i} \left\{ \bar{g}(\mathbb{X}_i, u_i, \lambda) + E_{\mathbb{X}_{i+1}|\mathbb{X}_i, u_i} J_{i+1}^D(\mathbb{X}_{i+1}, \lambda) \right\} \tag{5.16}$$

for the time indexes $i \in \{k : k + N - 1\}$, and terminate at time $k + N$ with

$$J_{k+N}^D(\mathbb{X}_{k+N}, \lambda) = -\lambda M \tag{5.17}$$

where M is the total budget for our movement costs. The belief state $\mathbb{X}_{i+1}$ is calculated using the particle filter update equations describe in Section 4.2.

It should also be noted that this same formulation can be used to solve the dual problem where we seek to optimize movement costs subject to a constraint on our estimate quality. Going through the whole derivation will result in precisely the same solution structure as Equation (5.15) excepted the Lagrangian multiplier is on the mutual information term instead of the cost term [27].

5.1.3 Linearized Gaussian Approximation

In order to evaluate the dynamic program in Equation (5.16) for complex or nonlinear measurement models, we would need to simulate the measurements from each of the $N_v^{N_s}$ possible configurations of the sensors in $\mathcal{S}_k$ at each time step. This could be done with a particle filter where we generate N_p particles to represent the measurements in $\boldsymbol{z}_k^{\mathcal{S}_k}$. This would result in a tree structure where for each of the leaves at the previous stage we must generate $N_v^{N_s} N_p$ particles to simulate all of the possible measurements at the next stage. So for a planning horizon of N the complexity will be $O(N_v^{N_s N} N_p^N)$. Clearly this will be intractable for even small values of N and N_p.

When the dynamics and measurement models are linear and Gaussian then the problem is dramatically simplified. Lets see what our mutual information objective will look like under a linear Gaussian assumption. Our measurement model will be:

$$\boldsymbol{z}_k^{\mathcal{S}_k} = \mathbf{H}_k^{\mathcal{S}_k} \boldsymbol{x}_k + \boldsymbol{v}_k^{\mathcal{S}_k} \tag{5.18}$$

where the *a priori* distribution of $\boldsymbol{x}_k$ is $\mathcal{N}\{\boldsymbol{x}_k; \boldsymbol{\mu}_k, \mathbf{P}_k\}$. We want to calculate mutual information as follows:

$$I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1}) = H(\boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1}) - H(\boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{x}_k). \tag{5.19}$$

Based on linear Gaussian assumptions we know that $\boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{x}_k \sim \mathcal{N}\{\boldsymbol{z}_k^{\mathcal{S}_k}; \mathbf{H}_k^{\mathcal{S}_k} \boldsymbol{x}_k, \mathbf{R}^{\mathcal{S}_k}\}$, so we can calculate

$$H(\boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{x}_k) = \frac{1}{2} \log \left| 2\pi e \mathbf{R}^{\mathcal{S}_k} \right|. \tag{5.20}$$

Also, we know $\boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1} \sim \mathcal{N}\{\boldsymbol{z}_k^{\mathcal{S}_k}; \mathbf{H}_k^{\mathcal{S}_k} \boldsymbol{\mu}_k, \mathbf{H}_k^{\mathcal{S}_k} \mathbf{P}_k {\mathbf{H}_k^{\mathcal{S}_k}}^T + \mathbf{R}^{\mathcal{S}_k}\}$ for a linear Gaussian measurement model [17], and therefore we can calculate

$$H(\boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1}) = \frac{1}{2} \log \left| 2\pi e \left(\mathbf{H}_k^{\mathcal{S}_k} \mathbf{P}_k {\mathbf{H}_k^{\mathcal{S}_k}}^T + \mathbf{R}^{\mathcal{S}_k} \right) \right|. \tag{5.21}$$

Now we can combine these two terms to get a final result for our mutual information

objective function,

$$I(\boldsymbol{x}_k; \boldsymbol{z}_k^{\mathcal{S}_k} | \boldsymbol{z}_{0:k-1}) = \frac{1}{2} \log \left[\frac{\left| \mathbf{H}_k^{\mathcal{S}_k} \mathbf{P}_k \mathbf{H}_k^{\mathcal{S}_k T} + \mathbf{R}^{\mathcal{S}_k} \right|}{|\mathbf{R}^{\mathcal{S}_k}|} \right]. \tag{5.22}$$

Based on the fact that $\mathbf{P}_k$, $\mathbf{H}_k^{\mathcal{S}_k}$, and $\mathbf{R}^{\mathcal{S}_k}$ are all independent of the specific measurements (they only depend on the specific sensors and their locations) we see that the mutual information objective will only depend on the specific control u_k chosen and not on the specific measurements that result from that control [26]. This means that if we have linear Gaussian dynamics and measurement models then we won't need to simulate any measurements. We can calculate our objective function solely using the relevant covariances. This reduces the complexity of solving the dynamic program in Equation (5.16) from $O(N_v^{N_s N} N_p^N)$ to just $O(N_v^{N_s N})$.

For our tracking experiment our range measurements are based on the nonlinear measurement model in Equation (4.4). However, using the Taylor series expansion from Equation (4.5) will allow us to utilize the above simplifications. This is a common approach known as a linearized Kalman filter [17, 27].

Unfortunately this only reduces some of the complexity. As previously noted, evaluating our dynamic program will still have a complexity of $O(N_v^{N_s N})$. So the tree representing our dynamic program will grow exponentially in both the number of sensor N_s and the length of our planning horizon N. In order to develop a plan quickly enough that it can be acted upon in a real-time exercise like tracking, we will need to introduce some sort of approximation that will make it possible to find a solution quickly.

5.2 Approximation Algorithms

We will now look at some of the approximation algorithms that we developed for this problem. The single time step or myopic algorithms only consider a plan for the next step (*i.e.* $N = 1$). These approaches are fairly straightforward and mirror approximations previously mentioned for use in static sensor networks. The algo-

rithms for planning over multiple time steps require more creativity to produce an approximation that has good performance and is still efficient enough to be useful.

5.2.1 Brute Force

While not technically an approximation, the brute force approach is useful in providing the foundation for the other approximations and stands as a basis for comparison. The end product of our the dynamic programming approach was the recursive dynamic program in Equation (5.16). Combining that with Equation (5.15) we can see the specific dynamic program we will be solving for our tracking problem, *e.g.*

$$J_i^D(\mathbb{X}_i, \lambda) = \min_{u_i} \left\{ \bar{g}(\mathbb{X}_i, u_i, \lambda) + E_{\mathbb{X}_{i+1}|\mathbb{X}_i, u_i} J_{i+1}^D(\mathbb{X}_{i+1}, \lambda) \right\} \tag{5.23}$$

$$= \min_{u_i} \left\{ \sum_{j=1}^{N_s} \left[-I\left(\boldsymbol{x}_i; \boldsymbol{z}_i^{\mathcal{S}_i^j} | \boldsymbol{z}_{0:i-1}, \boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}} \right) + \lambda C_i^j \right] + E_{\mathbb{X}_{i+1}|\mathbb{X}_i, u_i} J_{i+1}^D(\mathbb{X}_{i+1}, \lambda) \right\} \tag{5.24}$$

for the time indexes $i \in \{k : k+N-1\}$, and we terminate at time $k+N$ with $J_{k+N}^D(\mathbb{X}_{k+N}, \lambda) = -\lambda M$.

The challenging aspect of using mutual information as our objective can be seen by looking more closely at the mutual information term in Equation (5.24),

$$I\left(\boldsymbol{x}_i; \boldsymbol{z}_i^{\mathcal{S}_i^j} | \boldsymbol{z}_{0:i-1}, \boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}} \right). \tag{5.25}$$

At time step i the measurement from the jth sensor will be $\boldsymbol{z}_i^{\mathcal{S}_i^j}$. From the expression above we see that the mutual information between that measurement and the object state $\boldsymbol{x}_i$ is dependent on all of the previous measurements from all of the sensors, $\boldsymbol{z}_{0:i-1}$, and on current measurements already selected for other sensors during this time step, $\boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}}$. It is intuitive that the amount of information gained from some measurement will be highly dependent on what information has already been obtained from any other measurements. However, this strong interdependence of mutual information between different measurements makes it difficult to divide up or

simplify the problem.

The only way to fully evaluate the dynamic program in Equation (5.24) is to simply calculate all of the possible solutions in a brute force manner. This analysis will lead to a recursive tree structure. The root node at time k would represent the current state, which consists of our current set of nodes $\mathcal{S}_k$. Then from the root node there will be a branch for each possible configuration of sensor nodes at time $k+1$. We have N_s sensors and each one can move to N_v locations at the next step, which means there are $N_v^{N_s}$ possible configurations for $\mathcal{S}_{k+1}$. Then at the next time step we would do the same analysis for each of those $N_v^{N_s}$ tree nodes. To brute force evaluate all of the solutions over an N-step planning horizon would result in enumerating $N_v^{N_s N}$ plans, where each plan would consist of a unique set of N-step paths for the N_s sensors in $\mathcal{S}_k$. This simply isn't feasible to be accomplished in real time like we need for a tracking exercise.

5.2.2 Decoupled

We could dramatically simplify the dynamic program if we could decouple the sensors such that we could plan for each one independent of the others. In other words, for planning each sensor wouldn't consider any of the future measurements from other sensors and would only consider the mutual information between its own measurements and the object state, *i.e.* if our objective was as follows,

$$I\left(\boldsymbol{x}_i; \boldsymbol{z}_i^{\mathcal{S}_i^j} | \boldsymbol{z}_{k:i-1}^{\mathcal{S}_i^j}\right) \tag{5.26}$$

where $\boldsymbol{z}_{k:i-1}^{\mathcal{S}_i^j}$ are future measurements (after time k) from only the jth sensor. Note that we still incorporate all of the previous measurements from all sensors, $\boldsymbol{z}_{0:k-1}$, in our actual estimates. It is only for planning ahead and estimating the information value of future measurements that we decouple the sensors.

With this objective each sensor could then plan its future sensor placements independent from the other sensors. Thus, decoupling the sensors like this would result in evaluation of the dynamic program being linear in the number of sensors N_s rather

than being exponential in N_s. It would also allow for a more practical and efficient distributed implementation where each sensor node could do its own decentralized planning and only the resulting measurements would need to be shared.

However, we must consider whether this is a valid approximation to make. By removing the conditioning on $\boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}}$ (*i.e.* the measurements of the other sensors) we are making an implicit assumption that they don't affect the value of the mutual information between $\boldsymbol{x}_i$ and $\boldsymbol{z}_i^{\mathcal{S}_i^j}$. From the definitions of mutual information and entropy, we can see that this would hold true if measurements from one sensor are independent of the measurements from the others. Obviously this won't be the case unless they are independent of $\boldsymbol{x}$ as well. In which case we wouldn't be able to use them to help estimate $\boldsymbol{x}$. However, when the information from the measurements is not highly redundant, like when the sensor nodes are very sparsely distributed, this could be a good approximation.

When using this approximation the error will always tend towards overestimating the amount of mutual information. This follows from the submodularity property of mutual information. From submodularity we know that adding a measurement to a small set of measurements will always cause a larger increase in mutual information than if the same measurement is added to a larger superset of measurements. With this approximation each sensor is only considering its own measurements when in fact they will be combined with a larger set of measurements from all the sensors. Therefore from submodularity we will always be overestimating the mutual information when the measurements are not independent.

For our application it is better that the error is biased this way than if it was biased towards underestimating mutual information. This way we are biased towards collecting too many measurements (*i.e.* redundant measurements). So we will tend towards being inefficient or perhaps missing an opportunity to take a measurement that wouldn't have been redundant. If the we were biased the other way we would be erring towards taking too few measurements, because we would tend to think the measurement wasn't valuable. In this situation our sensor network could be dysfunctional rather than just being inefficient.

5.2.3 Greedy

In Section 2.2 when we introduced the concept of mutual information, we showed that one of its desirable qualities are that there is an efficient greedy algorithm. In [11, 14] the authors presented this greedy algorithm for use in the optimal sensor subset selection problem with static sensor networks. They proved that the greedily selected subset will provide mutual information that is at least $(1-1/e)$OPT or more than 63% of the optimal amount. We develop a greedy multistep algorithm that is modeled after and motivated by this this greedy sensor subset selection algorithm. However, our objective function does not meet all of the criteria to guarantee the same theoretical bounds. We will show that our objective function is in fact submodular. Even though our algorithm will only utilize nonnegative values of the objective function there are scenarios where the function itself can take on negative values. Therefore we cannot ensure monotonicity for all values of the objective function, which is required to guarantee the theoretical $(1 - 1/e)$OPT bound. We do show, via our simulations in Chapter 6, that in practice our greedy multistep algorithm performs much better than the single step and decoupled algorithms.

The greedy sensor selection algorithm relied on the fact that the mutual information objective could be broken down into the following form:

$$I(\boldsymbol{x}; \boldsymbol{z}^{\mathcal{S}}) = \sum_{j=1}^{N_s} I(\boldsymbol{x}; \boldsymbol{z}^{\mathcal{S}^j} | \boldsymbol{z}^{\mathcal{S}^{1:j-1}}). \tag{5.27}$$

This structure allows sensors to be selected sequentially such that each selection maximizes the corresponding term in the objective function. This heuristic greedy algorithm is formally described in Algorithm 1.

If we look again at the augmented objective function from our dynamic programming solution, we see that it has the same structure.

$$\bar{g}(\mathbb{X}_i, u_i, \lambda) = \sum_{j=1}^{N_s} \left[-I\left(\boldsymbol{x}_i; \boldsymbol{z}_i^{\mathcal{S}_i^j} | \boldsymbol{z}_{0:i-1}, \boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}}\right) + \lambda C_i^j \right]. \tag{5.28}$$

Algorithm 1 Greedy Sensor Subset Selection Algorithm

Input: N_s, $\mathcal{V}$
Output: Sensor subset $\mathcal{S} \subseteq \mathcal{V}$
  $\mathcal{S} \leftarrow \emptyset$
  for $j = 1$ to N_s **do**
    $s^* \leftarrow \arg\max_{s \in \mathcal{V} \setminus \mathcal{S}} I(\boldsymbol{x}; \boldsymbol{z}^s | \boldsymbol{z}^{\mathcal{S}})$
    $\mathcal{S} \leftarrow \mathcal{S} \cup s^*$
  end for

Therefore we should be able to use the same greedy algorithm to select our sensors.

However, before we adapt this greedy algorithm to our problem, there are two issues that must be addressed. First, because we are solving a cost constrained problem, we ended up with an augmented objective function. This objective function is not based purely on mutual information, because we augmented it with the λC_k^j cost constraint. Therefore, we will need to investigate how this augmented cost term affects submodularity and monotonicity. Second we are utilizing mobile sensors and planning over a rolling time horizon, and it isn't immediately clear how well the greedy sensor selection algorithm will carry over to this different problem structure.

Note that Equation (5.28) is only the per-stage objective function. So we would be making a greedy sensor subset selection at each time step in our plan, not over the whole planning horizon. So at each time stage i we will have N_s substages indexed by j where we greedily select a single sensor that optimizes the correlating term of our objective function. Specifically, at stage i and substage j we would maximize the following substage objective:

$$\widetilde{g}(\mathbb{X}_i, S_i^j, \lambda) = I\left(\boldsymbol{x}_i; \boldsymbol{z}_i^{\mathcal{S}_i^j} | \boldsymbol{z}_{0:i-1}, \boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}}\right) - \lambda C_i^j. \tag{5.29}$$

Recall that the theoretical bounds for the greedy algorithm relied on the fact that the objective function was monotonic and submodular. Looking again at Equation (5.29), it is clear that our augmented objective isn't strictly monotonic. Any time the cost term λC_i^j is larger than the mutual information term, our objective function will be negative and thus we can't guarantee monotonicity. Intuitively that is what our objective function was designed to do. It was suppose to help us identify when the

cost of moving a sensor is higher than the value of the information we get from the measurement at the new location. The cost can be higher than the information gained for multiple reasons. The most obvious reason being that some moves could simply be prohibitively expensive. However, a more subtle problem is the fact that the value of information from a particular measurement always decreases as we obtain more and more measurements. So over time our measurements will contribute less and less information and the mobility costs will come to dominate the objective function. At which point, even for relatively cheap moves the costs will outweigh the information gain, because there simply isn't much information left to obtain.

In practice our greedy algorithm will never select options that result in negative values of the objective function. These are precisely the sensor selections we want to reject because the information is not worth the cost incurred to obtain it. Note that for every substage, among the N_v possible controls, we always include the option of not moving, *i.e.* staying at the same virtual node. From Equation (4.8), we see that this option always incurs a cost of zero. Combined with the fact that mutual information is always nonnegative, we know that at every substage there will always be an option such that the our substage objective in nonnegative. Since we are maximizing $\widetilde{g}(\mathbb{X}_i, S_i^j, \lambda)$ and we know there is a control at each substage such that $\widetilde{g}(\mathbb{X}_i, S_i^j, \lambda) \geq 0$, our cumulative reward will always be nonnegative. Even though our selected sensor locations will always result in a nonnegative objective in practice, this is insufficient to prove the $(1-1/e)$OPT bound. For that bound to hold we would need to show that our augmented objective is nonnegative for all possible inputs [19, 11], which we cannot do without putting additional constraints on the cost structure or the Lagrangian formulation.

Fortunately, we can still show that our augmented objective is submodular. Recall that a set function F is submodular if for $\mathcal{S} \subseteq \hat{\mathcal{S}}$,

$$F(\hat{\mathcal{S}} \cup s) - F(\hat{\mathcal{S}}) \leq F(\mathcal{S} \cup s) - F(\mathcal{S}) \tag{5.30}$$

In other words adding s to the larger set $\hat{\mathcal{S}}$ has less impact than adding s to the smaller

set $\mathcal{S}$. This is often referred to as the property of diminishing returns. Because of the fact that the incremental cost of adding a specific sensor location to a set is invariant of the size of the set, our substage objective will reduce to the same submodularity property that mutual information has. For $\mathcal{S} \subseteq \hat{\mathcal{S}}$ we have the following expansion for the set $\hat{\mathcal{S}}$,

$$\widetilde{g}(\mathbb{X}, \hat{\mathcal{S}} \cup s, \lambda) - \widetilde{g}(\mathbb{X}, \hat{\mathcal{S}}, \lambda) \tag{5.31}$$

$$I(\boldsymbol{x}; \boldsymbol{z}^{\hat{\mathcal{S}} \cup s}) - \lambda C^{\hat{\mathcal{S}} \cup s} - I(\boldsymbol{x}; \boldsymbol{z}^{\hat{\mathcal{S}}}) + \lambda C^{\hat{\mathcal{S}}} \tag{5.32}$$

$$I(\boldsymbol{x}; \boldsymbol{z}^{\hat{\mathcal{S}}}) + I(\boldsymbol{x}; \boldsymbol{z}^{s} | \boldsymbol{z}^{\hat{\mathcal{S}}}) - \lambda C^{\hat{\mathcal{S}}} - \lambda C^{s} - I(\boldsymbol{x}; \boldsymbol{z}^{\hat{\mathcal{S}}}) + \lambda C^{\hat{\mathcal{S}}} \tag{5.33}$$

$$I(\boldsymbol{x}; \boldsymbol{z}^{s} | \boldsymbol{z}^{\hat{\mathcal{S}}}) - \lambda C^{s} \tag{5.34}$$

and in the same manner we have the following expansion for the set $\mathcal{S}$,

$$\widetilde{g}(\mathbb{X}_i, \mathcal{S} \cup s, \lambda) - \widetilde{g}(\mathbb{X}_i, \mathcal{S}, \lambda) \tag{5.35}$$

$$I(\boldsymbol{x}; \boldsymbol{z}^{s} | \boldsymbol{z}^{\mathcal{S}}) - \lambda C^{s} \tag{5.36}$$

Now we must show that the expression in (5.34) is less than or equal to the one in (5.36). Making the comparison we have,

$$I(\boldsymbol{x}; \boldsymbol{z}^{s} | \boldsymbol{z}^{\hat{\mathcal{S}}}) - \lambda C^{s} \leq I(\boldsymbol{x}; \boldsymbol{z}^{s} | \boldsymbol{z}^{\mathcal{S}}) - \lambda C^{s} \tag{5.37}$$

$$I(\boldsymbol{x}; \boldsymbol{z}^{s} | \boldsymbol{z}^{\hat{\mathcal{S}}}) \leq I(\boldsymbol{x}; \boldsymbol{z}^{s} | \boldsymbol{z}^{\mathcal{S}}) \tag{5.38}$$

Equation (5.38) is simply stating that adding the measurement from s to those of $\hat{\mathcal{S}}$ will add less information than adding the measurement from s to $\mathcal{S} \subseteq \hat{\mathcal{S}}$. We know this is true because of the fact that mutual information is submodular. This proves that our augmented substage objective from Equation (5.29) is in fact submodular. While we can't guarantee strict monotonicity, the fact that our substage objective function is submodular and that we only utilize nonnegative selections in practice, suggests that a greedy approach will still give favorable results.

For our multistep planning problem our greedy algorithm candidate is shown in

Algorithm 2.

Algorithm 2 Greedy Multistep Algorithm Candidate

Input: N_s, N, $\mathcal{V}$, $\mathcal{S}_0$
Output: N sensor subsets $\mathcal{S}_{1:N} \subseteq \mathcal{V}$
  for $i = 1$ to N **do**
    $\mathcal{S}_i \leftarrow \emptyset$
    for $j = 1$ to N_s **do**
      $\mathcal{V}^{\mathcal{S}_{i-1}^j} \leftarrow \{\text{virtual nodes in neighborhood of } \mathcal{S}_{i-1}^j\}$
      $s^* \leftarrow \arg\max_{s \in \mathcal{V}^{\mathcal{S}_{i-1}^j}} I\left(\boldsymbol{x}_i; \boldsymbol{z}_i^s | \boldsymbol{z}_{0:i-1}, \boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}}\right) - \lambda C_i^j$
      $\mathcal{S}_i^j \leftarrow s^*$
    end for
  end for

Everything looks great, until we take a closer look to see what is really going on. Let us use the notation $\mathcal{S}_{k:k+i}$ to denote a sequence of sensor node configurations $\mathcal{S}_k \rightarrow \mathcal{S}_{k+1} \rightarrow \ldots \rightarrow \mathcal{S}_{k+i}$. Then we can think of $\mathcal{S}_{k:k+i}$ as a set of N_s planned paths that the sensors will follow from time k to time $k+i$, where the jth sensor node will move along the path $\mathcal{S}_{k:k+i}^j$. According to Algorithm 2 at each stage i we should greedily select a set of sensor locations $\mathcal{S}_i$. So at the end of stage i of our planning, instead of having $O(N_v^{N_s i})$ possible paths to consider we have aggressively narrowed it down to one path for each of the N_s sensors, namely $\mathcal{S}_{k:k+i}$.

This means this multistep planning algorithm isn't planning over our N step horizon at all. It is aggressively narrowing decision tree down to one branch at each level. If there is only one option after the first step of planning, then there is no point in planning any further ahead, because we already know what we are going to do on the next step. In other words Algorithm 2 is equivalent to just doing the greedy single time step algorithm found in Algorithm 3.

This is a very important distinction to make. We are trying to optimize over the entire N-step planning horizon, but we are really only doing the greedy optimizing on a per-stage basis. The optimal single step solution can be an arbitrarily bad approximation for the optimal N-step solution. For example imagine a scenario where the target object being tracked moves out of the range of any of the sensor nodes, and none of the sensor nodes can get back in range over the next time step. In this case

Algorithm 3 Greedy Single Step Algorithm

Input: N_s, $\mathcal{V}$, $\mathcal{S}_{k-1}$
Output: $\mathcal{S}_k \subseteq \mathcal{V}$
  $\mathcal{S}_k \leftarrow \emptyset$
  for $j = 1$ to N_s **do**
    $\mathcal{V}^{\mathcal{S}^j_{k-1}} \leftarrow \{\text{virtual nodes in neighborhood of } \mathcal{S}^j_{k-1}\}$
    $s^* \leftarrow \arg\max_{s \in \mathcal{V}^{\mathcal{S}^j_{k-1}}} I\left(\boldsymbol{x}_k; \boldsymbol{z}^s_k | \boldsymbol{z}_{0:k-1}, \boldsymbol{z}_k^{\mathcal{S}_k^{1:j-1}}\right) - \lambda C^j_k$
    $\mathcal{S}^j_k \leftarrow s^*$
  end for

the optimal single step solution would be for every sensor node to not move at all and stay where it currently is. That is because every other possible move over the next time step won't provide any additional information but they would all incur a cost. However, the optimal N-step solution would be able to find solutions that incurred the early cost to get back in range because those costs would be compensated for by the information gained at the later stages.

What we really want is a way to find a plan consisting of paths that are optimized over our entire N-step planning horizon, rather than creating a plan of N independently optimized single stages as the greedy single step algorithm would do. The key insight to making this possible is to note that our objective has the same structure over entire paths not just only over nodes. In other words we can take our per-stage objective function and sum it up over the entire path. Then we use that cumulative function as our new per-path objective function. We just need to show that this per-path objective function has the same structure as our previous per-stage objective. If that is true, then we can utilize the same algorithm to greedily select entire paths $\mathcal{S}^j_{k:k+N}$ rather than just greedily selecting individual nodes $\mathcal{S}^j_k$. If we plug our augmented per-stage objective function from Equation (5.28) back in to the original Lagrangian function in Equation (5.7) we see the cumulative objective for our dynamic program has the following form:

$$-\sum_{i=k}^{k+N}\sum_{j=1}^{N_s}\left[I\left(\boldsymbol{x}_i; \boldsymbol{z}_i^{\mathcal{S}_i^j} | \boldsymbol{z}_{0:i-1}, \boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}}\right) - \lambda C_i^j\right]. \tag{5.39}$$

Again we see that our augmented objective can be split into the per-stage and per-sensor contributions of each selection. All we need to do is change the order of summation and we can see our new per-path objective function:

$$\sum_{i=k}^{k+N} \left[I\left(\boldsymbol{x}_i; \boldsymbol{z}_i^{\mathcal{S}_i^j} | \boldsymbol{z}_{0:i-1}, \boldsymbol{z}_i^{\mathcal{S}_i^{1:j-1}} \right) - \lambda C_i^j \right]. \tag{5.40}$$

Previously we were summing over j first, which is the same as selecting locations for each sensor at time step i before moving to the next time step $i+1$. Now we are summing over i first, which is the same as selecting an entire path of locations from time k to time $k+N$ for a specific sensor $\mathcal{S}^j$, before moving on to the next sensor $\mathcal{S}^{j+1}$. This per-path objective has the identical structure to our per-stage objective in Equation (5.15). Because the incremental costs of adding an entire path is still independent of the size of the previous set of selected paths and the additive property of mutual information, the proof for submodularity of the per-path objective follows the proof for the per-stage objective exactly. With the only difference being that there are now N terms instead one term.

The same considerations still exist for monotonicity as did with the per-stage objective. The only caveat being the interpretation was different. In the previous case the greedy algorithm would reject sensor locations for which the costs was higher than the expected information gain. Likewise we will now select only those paths that cumulatively have more information than they do cumulative costs. Again in practice our objective function will be nondecreasing, but we cannot guarantee the it will be nondecreasing for all possible inputs. However, note that since we are using the cumulative costs and cumulative information reward over the entire path as our objective, we can now consider individual nodes along the path that would have been rejected under the per-stage objective. That is, we don't have to maintain monotonicity as we sum along the path. We can take on extra costs and have the objective go negative as long as the cumulative objective function of the whole path is nonnegative.

In simple terms our multiple time step greedy algorithm consists of each sensor

evaluating the per-path objective in Equation (5.40) for each length N path it could take. Then at each step we select one path from all of the paths from all of the sensors, that maximizes the per-path objective. Then we fix that path and iterate conditioning on the measurements from the previously selected paths. The greedy multistep algorithm is formally explained in Algorithm 4.

Algorithm 4 Greedy Multistep Algorithm

Input: N_s, N_v, N, $\mathcal{V}$, $\mathcal{S}_k$
Output: N_s sensor paths $\mathcal{S}_{k:k+N}$

```
S_{k+1:k+N} ← ∅
for j = 1 to N_s do
    Get all length N paths from the j remaining sensors in S_k
    P ← GetPaths(S_k^{j:N_s}, N_v, N)
    p* ← arg max_{p∈P} Σ_{i=k}^{k+N} [ I( x_i; z_i^{S_i^j} | z_{0:i-1}, z_i^{S_i^{1:j-1}} ) − λ C_i^j ]
    S_{k:k+N}^j ← p*
end for
```

This algorithm will be much more efficient than the brute force evaluation of the dynamic program. The complexity is reduced because we only condition on the specific paths that were previously selected. This way on the jth iteration each sensor node is only conditioning on the measurements from the $j-1$ specific paths that were previously selected, rather than on all of the possible paths from the previous j sensor nodes. This results in the complexity being $O(N_s^2 N_v^N)$ rather than $O(N_v^{N_s N})$. In the next subsection we will discuss some ways to reduce the complexity from the exponential N_v^N term.

Since we are optimizing over the whole N-step planning horizon rather than over single stages we expect this multistep algorithm will perform better than the single stage approximation algorithms. We also expect it to outperform the decoupled algorithm because mutual information between the other sensors' measurements is taken into consideration. This allows the greedy algorithm to choose more efficient paths that are collecting new information rather than redundant measurements as the decoupled algorithm can tend to do.

5.2.4 General Simplifications

There are a couple of simplifications that can be applied to all of our approximation algorithms. While we were able to address the exponential dependence on the number of sensor nodes N_s, all of the multistep algorithms still had an N_v^N complexity term. This term represents the work required to brute force evaluate every possible path from each of the nodes. Notice these paths are going to be very dense and most of them will be very similar to a large number of their neighboring paths. Also note that we utilize a rolling time horizon so we replan after taking each step. Therefore we don't need to be concerned about sampling so densely at the farthest regions of our planning horizon, because we will replan multiple times as we get closer to it. Using rapidly exploring random trees or some other more efficient method of selecting the set of paths we optimize over is an easy way to eliminate the $O(N_v^N)$ term from the complexity.

A second simplification arises from submodularity. Notice that in several of the algorithms when we iterate, we will end up evaluating the same nodes or paths over and over again. Because of submodularity we know that the amount of information that a measurement contributes to the cumulative total will decrease as we get more and more measurements. In the greedy algorithms we iterate and at each step we condition on the measurements selected in the previous steps. Therefore our objective function for any measurement will only go down on each iteration. It will never increase more than it was on the previous step. This means that once the objective function hits zero for a given node or path (*i.e.* the costs outweigh the information gain) then we can exclude that node or path from consideration for the rest of the current planning session.

This basically reduces the effective number of sensors that we are working with. This will be extremely beneficial in allowing our algorithms to scale to larger networks. Consider our tracking experiment in a very large network, where only a small subset of the sensors will be within close enough range to obtain measurements. All of the nodes that are out of range over our planning horizon will end up with an optimal

objective function equal to zero, meaning they should just stay right where they are for now. Fortunately, this will be noticed on the first iteration and they can be excluded from the rest of the computation. This reduces the effective network size to always being only the small cluster of nodes that are currently in range.

Chapter 6

Simulation

6.1 Autonomous Robotic Kayaks

Our mobile sensor network tracking problem and simulation was motivated by and based on a real network of autonomous robotic kayaks or SCOUTs (Surface Craft for Oceanographic and Undersea Testing) developed at the MIT Department of Ocean Engineering [9]. The SCOUT vehicles were specifically designed for experimentation with multi-robot cooperative autonomy. Each SCOUT houses a battery powered propulsion system, a complete main computer system, GPS, and Wi-Fi or other RF communications system. Each of these vehicles is a highly capable sensor node with the ability to autonomously move to a location, take measurements, process data, plan future controls, and communicate results.

These type of autonomous vehicles are routinely utilized in tracking applications. A team of SCOUTs are often deployed on the surface to help track and localize a submersible vehicle that doesn't have access to GPS data for self localization. Additionally they can be utilized in a more traditional tracking application where we want to identify and track any vehicle that enters a region of interest, such as a harbor.

The SCOUT vehicles are controlled by a well developed software stack known as MOOS-IvP [2]. MOOS-IvP is a collection of open source software modules that handle all of the control and autonomous behaviors for the vehicles. For example there are modules for collision avoidance and waypoint traversal.

Figure 6-1: MIT SCOUT autonomous kayak [9].

Figure 6-2: Main on board computer and 802.11b (Wi-Fi) radio [9].

Our simulation models a team of these autonomous kayaks executing a tracking exercise. We focus solely on the sensor path planning and tracking estimation portion of the problem. So the output of our simulation needs to provide a set of locations or waypoints for each of the sensors and an estimate of the current position and velocity of the object being tracked. In practice, collision avoidance and other such behaviors would be handled by separate MOOS-IvP modules.

6.2 Tracking Simulation

In order to make more concrete comparisons of performance and accuracy of the algorithms discussed in discussed in Chapter 5, we implemented each of them in our computer simulation. The simulations were performed in MATLAB and the programming code for the simulation is included in Appendix A.

We simulated a tracking exercise in which a team of mobile sensor nodes are tracking a single moving target. The algorithms in this thesis will extend to multiple target tracking problems, but doing so would require the data association problem to be solved. In order to avoid the completely separate issues of data association we limit ourselves to tracking a single target object.

Our tracking exercise was carried out in a two dimensional region of 100x100 units. We track a target that enters the region along a random trajectory at a uniform random location on the border of the region. The target then proceeds across the region in a random walk according to the dynamics described in Section 4.1.1. We randomly generated 100 target trajectories and tested each of the algorithms against the same 100 targets in separate simulations. So each algorithm ran through 100 complete simulations, and we average the results for our analysis.

For each simulation the kayaks start at uniform random locations within the bounded region. Each kayak represents a single sensor node that can take range measurements according to the measurement model in Section 4.1.2. As mentioned in Chapter 4, we utilize a hexagonal grid to model the discretized control space for each kayak. So at each time step the kayak can move to one of its six nearest neigh-

bors on the hexagonal grid or it can choose to remain at its current location. The simulations consisted of 200 time steps. At each step each sensor node takes a measurement. Then it produce an N-step plan of its movement for future time steps. However, we use a rolling time horizon so only the first move of the plan is executed, before we replan at the next time step. Each mobile sensor node has its motion costs constrained to a budget of $3N$, where N is the length of the planning horizon.

The path planning for each sensor was carried out according to the respective algorithms described in Chapter 5. The actual estimate of the target position and velocity was maintained via a particle filter as described in Section 4.2. We use the entropy of the belief state as modeled by the particle filter as the primary performance metric. The cost was simply calculated by summing up the total distance moved by all sensor nodes during the experiment.

Figures 6-3 and 6-4 show a typical simulation at the beginning and midway points respectively. The kayaks are represented by the ellipsoids randomly distributed through the region. We plot a random selection of the particles to help visualize the current state of the particle filter estimate. We also denote the true location of the target with a + symbol and we take the mean position over all particles and denote it with a $\times$ symbol.

In Figure 6-3 we see the state shortly after the simulation has commenced. Very few measurements have been taken and the kayaks have not had a chance to move into the optimal measurement locations. The cloud of particles is widely dispersed giving a clear indication that the current estimate is not well formed and has very high entropy.

In Figure 6-4 we see the state of the simulation midway through. Now kayaks have been able to move into the ideal positions and take sufficient measurements to obtain a much more accurate estimate. Note the particle cloud is now very small and dense and the mean position estimate coincides with the true target position, all indicating an accurate estimate with low entropy. Also note that the kayak movements have been very efficient. Only those kayaks that are nearby and able to obtain accurate measurements have moved into new positions. Also note that the first kayak

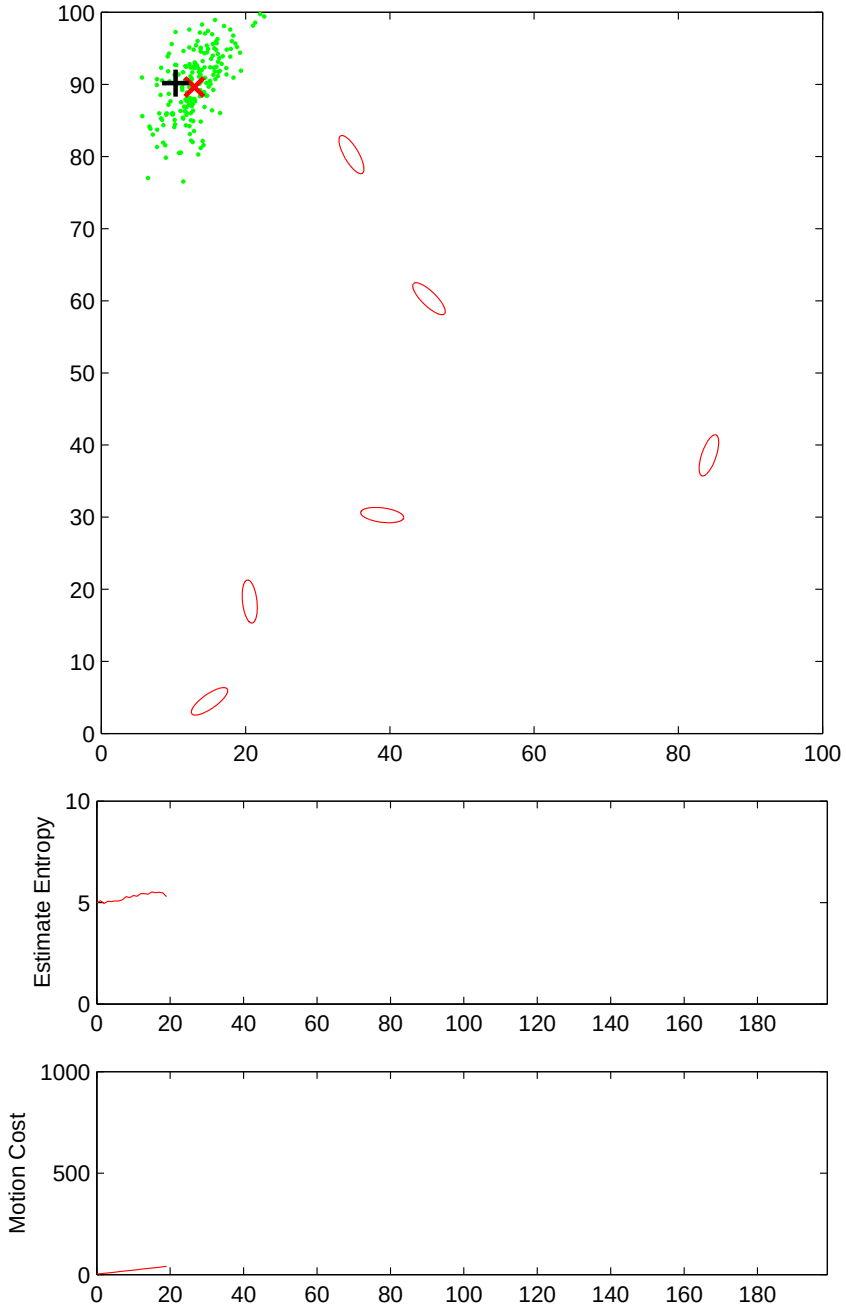

Figure 6-3: Typical simulation at the beginning before kayaks have had the chance to move into position or take very many measurements. The cloud of dots are sample particles from the current particle filter. The true position of the target is marked by a + and the current mean of all particles is marked with a ×.

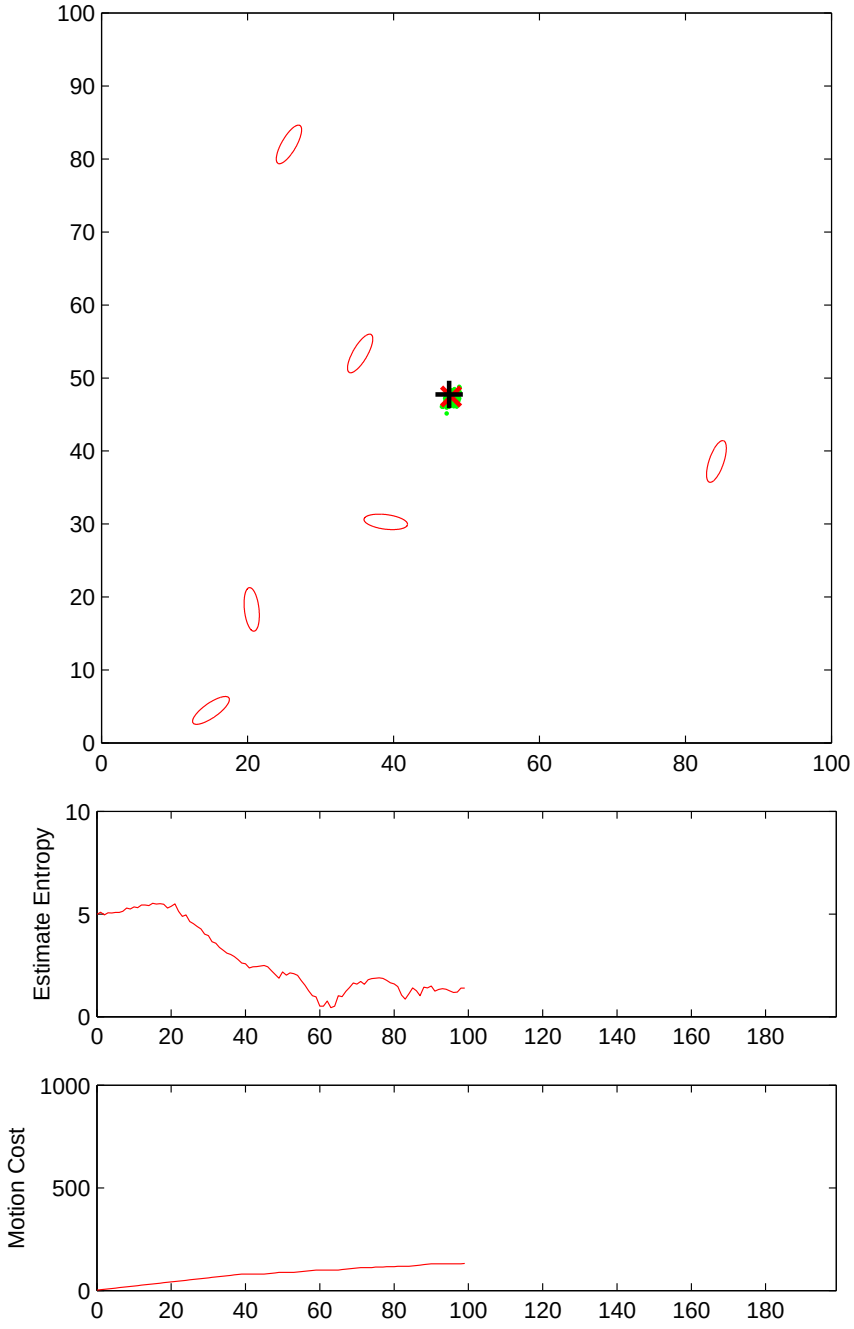

Figure 6-4: Later on in a typical simulation after kayaks have had the chance to move into position and take more measurements.

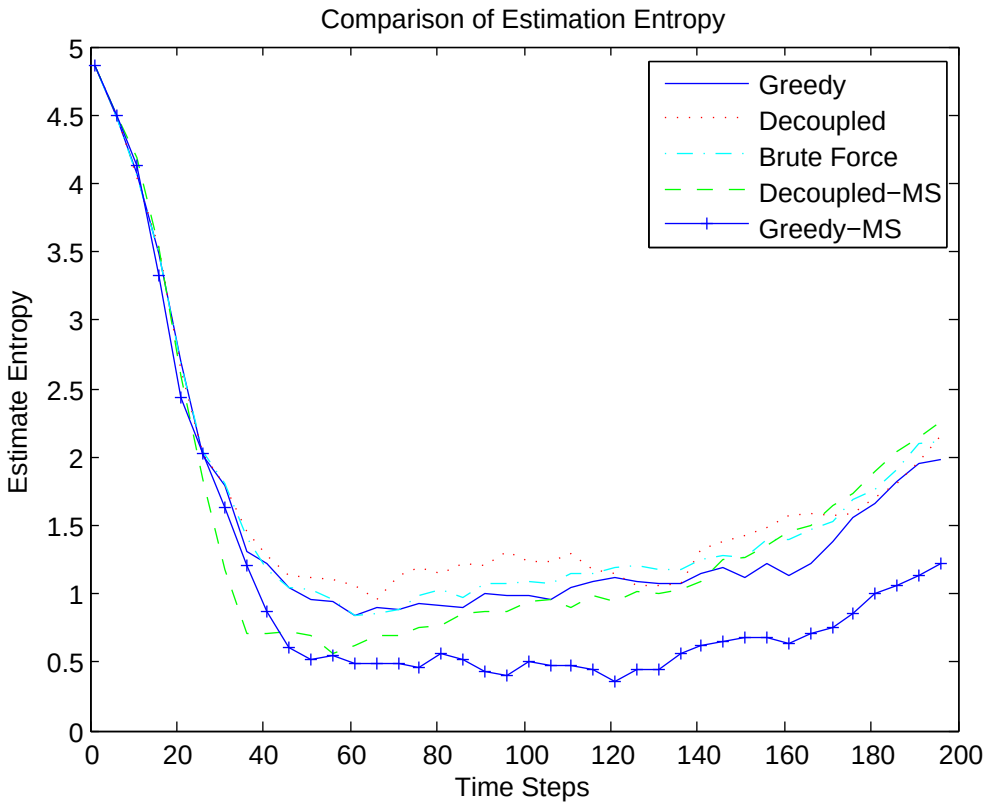

Figure 6-5: A comparison of average estimation entropy over the full 200 time steps of the simulations. We use "-MS" to indicate multistep algorithms that plan over a rolling $N = 10$ time step horizon.

that engaged the target stopped following when it was no longer needed and naturally allowed the other kayaks to take over. We will see later that in unconstrained information only algorithms we don't naturally achieve these types of efficiencies.

6.3 Comparative Analysis

Figure 6-5 shows the estimate entropy over the 200 time steps of the experiment, where averages were taken over the full 100 simulations. We simulated the single time step planning algorithms for the decoupled, greedy, and brute force algorithms. We also simulated multiple time step planning algorithms for the greedy and decoupled algorithms. We used a planning horizon of $N = 10$ steps for the simulations in Figure 6-5. All the algorithms follow a similar pattern where the entropy is reduced rapidly

early on as the first measurements come in. Also because the target is initially moving towards the sensors as it enters the region, the sensor nodes are easily able position themselves for the optimal measurements by moving towards a spot that will intercept the target's current trajectory. We see the entropy increase slightly at the end as the target moves past the sensor nodes and they have to handle a more challenging chase style problem as it moves away out of the region.

As predicted the multistep algorithms are able to achieve better estimates by planning ahead and moving accordingly. The greedy multistep being the most noteworthy by outperforming the others by approximately 30% on average. Notice that the decoupled algorithm performs well initially but then falls behind the rest of the algorithms by the end of the simulation. This behavior can be explained by the fact that the decoupled algorithm will tend to result in sensors moving such that they take redundant measurements because they don't consider the measurements of other sensors during planning. The longer the planning horizon the more this error is compounded and it becomes more likely that multiple sensors will try to move to locations that provide redundant information and to miss alternate sites that would provide new information. This results in poorly placed sensors and a higher entropy estimate. Whereas the greedy algorithm considers the future measurements of the other sensors, resulting in the appropriate value being placed on locations that will result in new unique information about the target object.

Also note that we use $N_s = 5$ sensors for these experiments to allow us to compute the brute force solution over a single time step. From the results we see that the single step greedy algorithm is able to achieve very similar performance but only requires $O(N_v N_s^2)$ calculations versus $O(N_v^{N_s})$ for the brute force method. Also even with only $N_s = 5$ sensors the brute force method took approximately 30 minutes to complete a single 200 step simulation making it infeasible for real-time planning with current processors.

It is important to remember that estimation performance is only half the story for our problem. We were interested in balancing estimation quality against our cost constraints. In Figure 6-6, we see how the algorithms compare when measured against

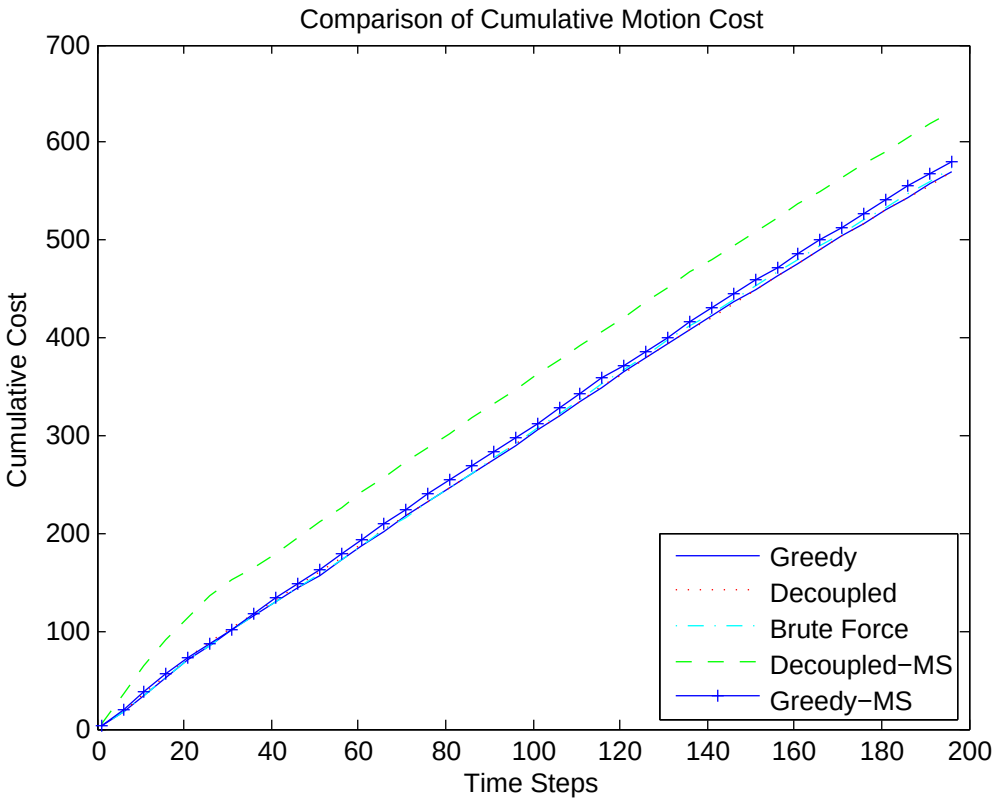

Figure 6-6: A comparison of the average cumulative motion costs over the full 200 time steps of the simulation. We use "-MS" to indicate multistep algorithms that plan over a rolling $N = 10$ time step horizon.

cumulative motion costs. With the exception of the decoupled multistep algorithm all of the algorithms performed almost identically on the cumulative cost criteria. This isn't too surprising since all of the algorithms were approximating the same dynamic program with the same objective function. So the distinction came down to which algorithms are more successful at selecting good measurements sites for the costs that they do incur.

As predicted the decoupled multistep algorithm incurs higher motion costs. Note that this is especially true at the very beginning. This again is because of a failure to consider future measurements of the other sensors. Initially when there are no measurements, because of submodularity, any new measurement will have premium value. Since every sensor assumes it is the only one taking a measurement, it overestimates the value of that measurement and can justify incurring a much higher cost (*i.e.* moving towards the target from farther away). Early on most of the sensors tend to move towards the target, until enough measurements are taken and the estimate improves enough that some of the sensors realize that their measurements won't provide valuable information and they make the correct choice to stop wasting their resources.

Also note from Figure 6-6 that the greedy multistep algorithm incurs a slightly higher cost than the single step algorithms. This makes sense because planning ahead allows the greedy algorithm to identify opportunities where slightly more movement can result in much better measurements. Looking at Figure 6-5 again we see that this slight increase in cost facilitated a much larger increase in estimate accuracy.

Because of the added complexity of the cost constrained optimization problem, it is common to consider information only solutions without regard for the costs of taking measurements. Figure 6-7 and Figure 6-8 demonstrate the contrast between using an information only solution and using a cost constrained solution as we provide. As would be expected, in Figure 6-7 we see that we can achieve better estimation performance by changing to a mutual information only objective. In our tracking problem we could achieve approximately 40% better estimation quality by using the same greedy algorithm without cost constraints. However, as Figure 6-8 shows, achieving

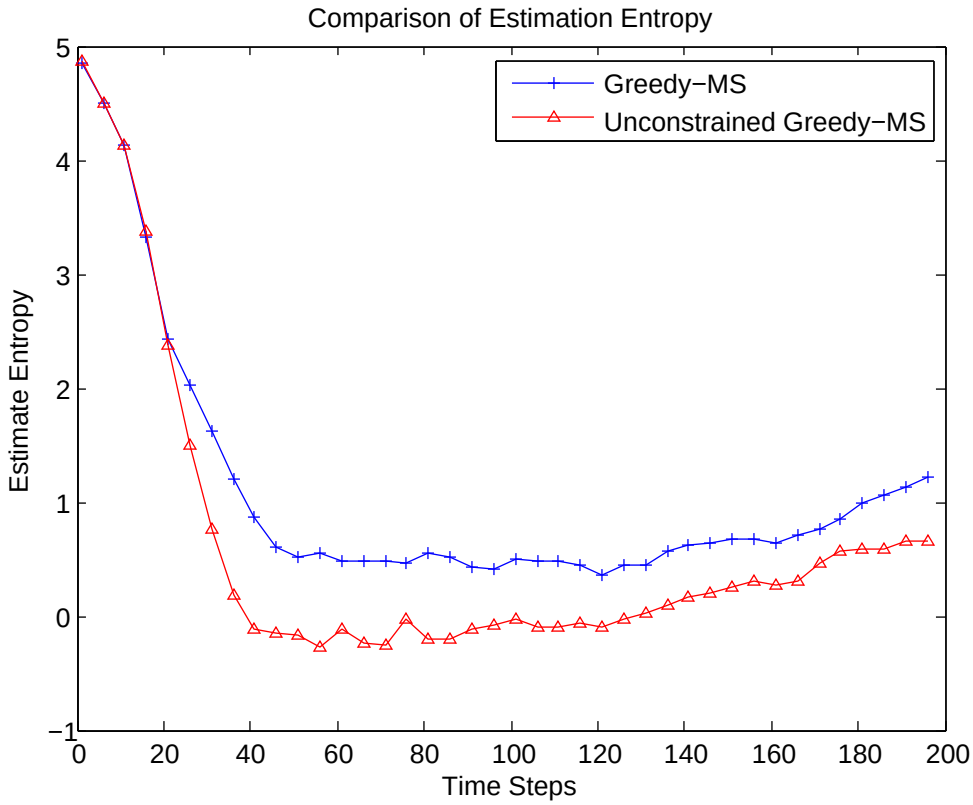

Figure 6-7: A comparison of average estimate entropy for cost constrained versus unconstrained solutions.

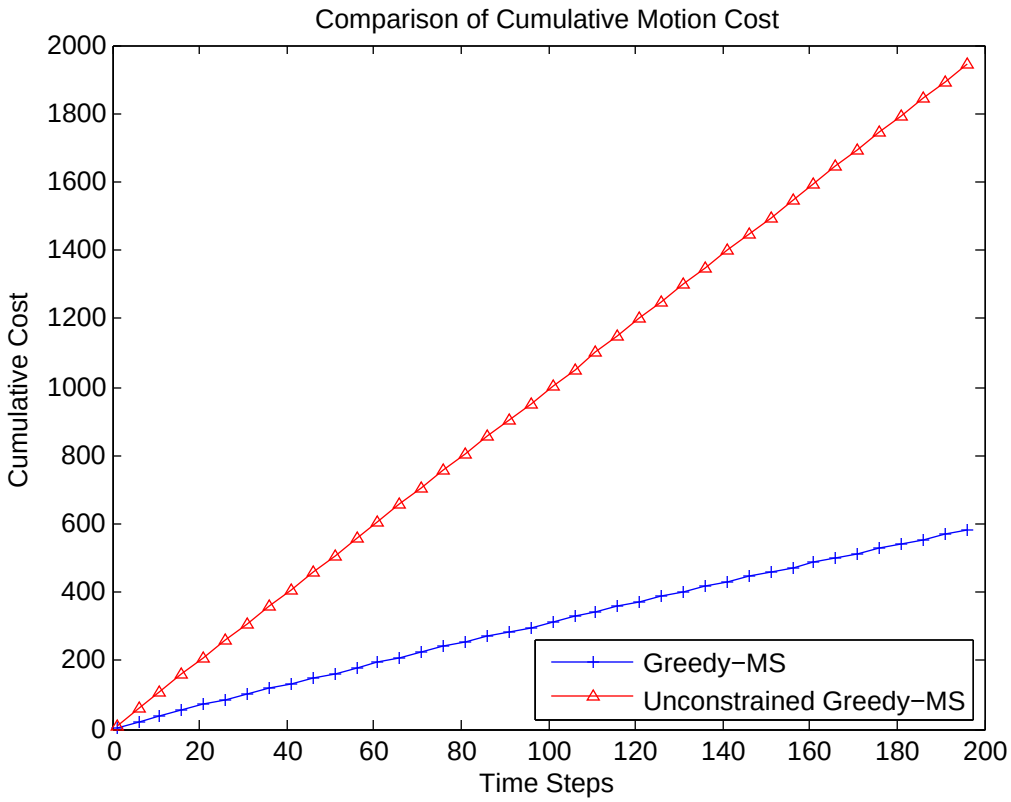

Figure 6-8: A comparison of average cumulative motion costs for cost constrained versus unconstrained solutions.

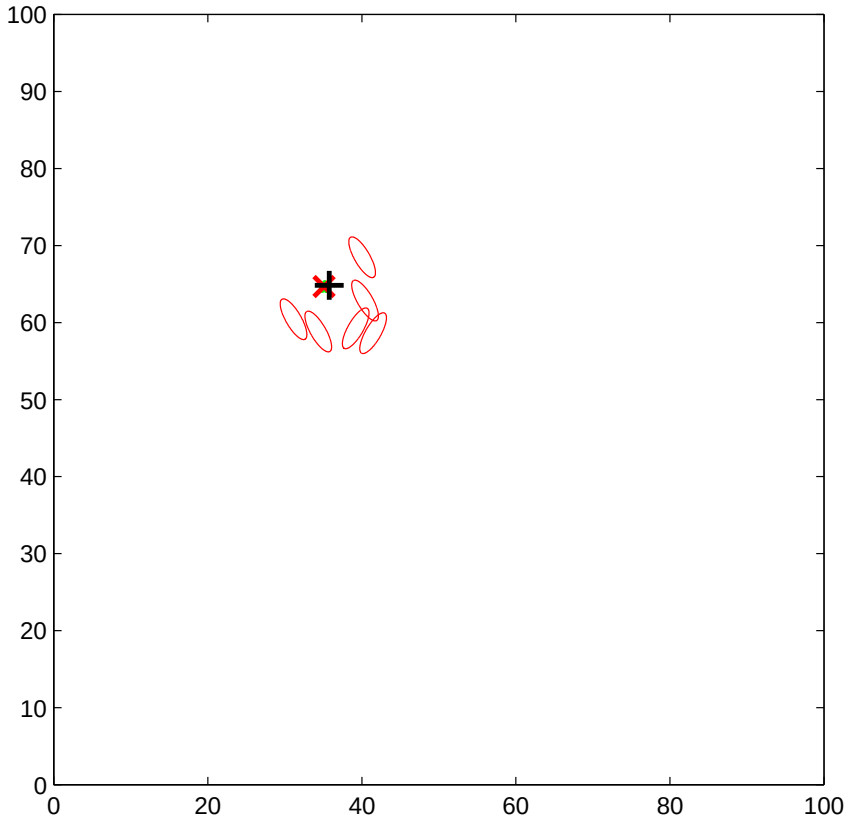

Figure 6-9: Typical simulation for the unconstrained greedy multistep algorithm. Notice that without cost constraints all of the kayaks tend to move towards the target.

that extra 40% in estimate quality incurs over 300% more in costs. In other words, our cost constrained solution would allow a mobile sensor network to work for over 3 times as long with only slightly degraded estimation performance.

We can see more explicitly how the unconstrained information only algorithms behave and understand why it is so costly by looking at an example simulation of the unconstrained greedy multistep algorithm. In Figure 6-9 we see the typical behavior of the unconstrained algorithm. Notice that all of the kayaks have moved towards the target object and surrounded it to obtain as much information as possible. Having all of the kayaks constantly moving in sync with the target requires a great deal of energy. Each additional kayak may provide only a small amount of additional information from its measurements but it is incurring a large cost to obtain it.

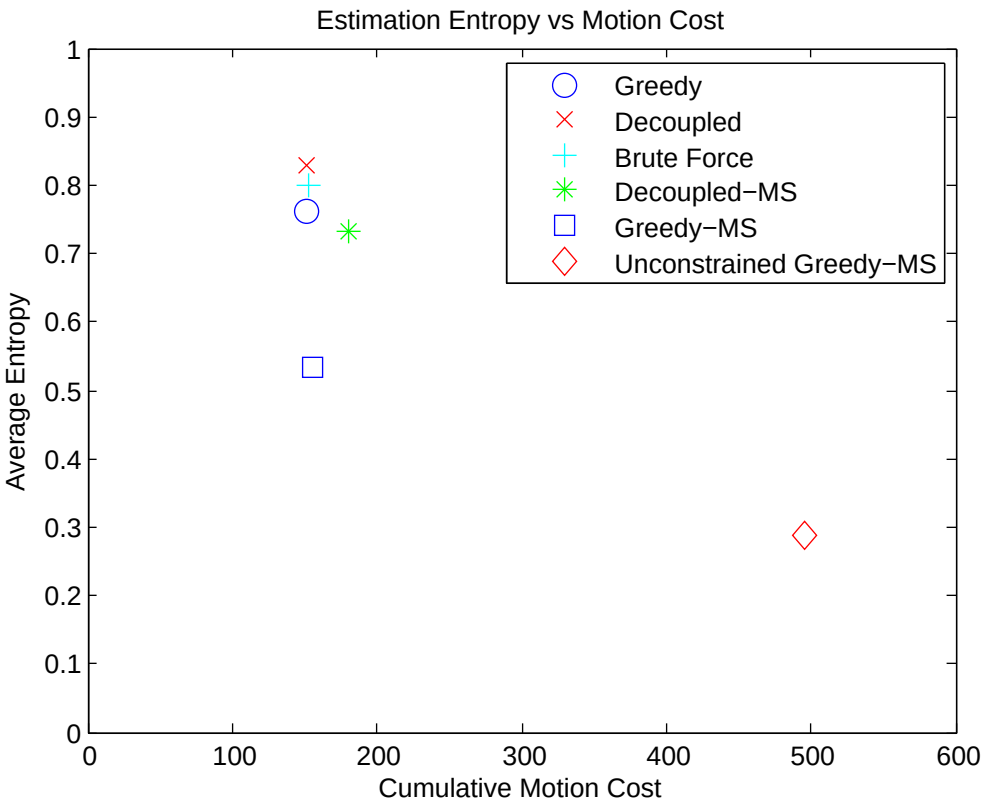

Figure 6-10: A comparison of average estimate entropy versus cumulative motion costs.

In Figure 6-10 we plot the average estimation entropy versus the cumulative motion costs for each of the algorithms. The ideal algorithm would have very low estimation entropy and very low cost and so it would occupy the lower left quadrant of this plot. From this figure it is clear that the greedy multistep algorithm is superior to the single step and decoupled approximations. It achieves significantly better estimation accuracy without incurring additional cost. From the figure we also see again that an unconstrained formulation can achieve superior estimation accuracy but at a significantly higher cost. It should be noted that as the motion cost budget is increased, (*i.e.* the cost constraints are loosened) the greedy multistep algorithm will gradually move towards the unconstrained solution. So if there is extra budget for the costs it will naturally be utilized to increase the estimation accuracy.

Chapter 7

Conclusion

7.1 Significant Findings

In this thesis we presented a constrained dynamic programming framework for planning mobile sensor node placements in a tracking problem. We developed a series of approximate dynamic programming algorithms that allowed this NP-hard problem to be solved in an efficient manner. We then verified and compared our algorithms via simulation in a tracking exercise.

We showed that traditional single step greedy algorithms used in static sensor networks don't readily translate over to mobile sensor networks where we want to plan sensor placements over some future time horizon. Our most significant contribution and finding was the extension of the greedy heuristic algorithm towards the greedy selection of length N paths rather than individual sensor node placements. By selecting sensor paths we can greedily optimize over the whole planning horizon rather than only over a single step.

Our simulations verified validity of our approach. First, we confirmed that for single time step planning the greedy algorithm performed nearly as well as the optimal single step results found via brute force calculation. Second, we showed that our greedy multistep algorithm was able to outperform myopic single step planning algorithms by choosing more informative sensor placements. Also it was able to do so at roughly the same cost as single step algorithms. Third, we showed that our

cost constrained approach allowed us to significantly reduce the cost expended moving sensors around without significantly reducing the estimation quality compared to unconstrained solutions.

7.2 Lessons Learned

7.2.1 Intelligent Path Selection

Our formulation was structured around the concept that we would have some discrete set of controls possible at each time step. We then generate potential sensor paths by exploring all of the possible controls at each time step. While this formulation is natural and intuitive, it also ends up being excessive for our purposes. If we look at all of the resultant paths generated by such an approach many of them will be nearly identical to groups of other paths, where the entire paths overlap with the exception of one or two nodes being different. This means we have a very dense set of candidate paths and we must consider many nearly redundant paths.

Also note that we are utilizing a rolling time horizon and we replan after each time step. Therefore, as we get to the later time steps along our path it is less important to sample the set of possible positions so densely, because we are going to replan several times before we get to those points. Another way of thinking about this is that we only need the later steps in our paths to be sufficient enough to get us pointed in the right direction. As we get closer to those points we will replan and can explore a more exhaustive set of possibilities in that area.

Our multistep planning algorithm relies on making a greedy selection over a set of possible paths. Since this algorithm is primarily concerned with optimizing over paths rather than optimizing on a per-stage basis, it would behoove us to be more intelligent about selecting which paths we wish to consider. Specifically we want the minimum set of paths such that they thoroughly represent the potential sensor configurations over the planning horizon. The work on randomized path planning and exploration from robotics could be brought to bear on this problem.

7.2.2 Efficient Implementation

During the implementation of the the greedy multistage algorithm we ran into challenges with memory requirements because we had to consider the future measurement of the other sensor nodes. Each time we incorporate a new measurement it changes the covariance matrix for our Kalman filter as shown in Equation (4.13). Therefore if we are going to consider the future measurements from other sensors we end up needing these covariance matrices for all of the other measurements at all of the time steps. Storing all of these covariances can be prohibitive.

Fortunately, at each time step we can recreate the needed covariance using the Kalman filter update equations. At each step we simply incorporate the measurements for that time step from the previously selected paths. At most this requires N_s updates of the Kalman filter equation (Equation (4.13)) at each time step. This allows us to avoid an $O(N_s N_v^N)$ memory requirement by making $O(N_s N)$ simple calculations.

7.3 Future Work

We now have the theoretical framework for our approximate dynamic programming algorithms in place and we have verified them via simulation. The next step will be to extend this work with a real implementation and real experiments. As a first step the simulation could be used to provide waypoints for a real team of SCOUT kayaks, and likewise receive real measurements from the kayaks. The next step would be then to create a full MOOS-IvP module that could perform the tracking and path planning algorithms on board. Then fully autonomous experiments could be performed with all computation and communication happening on board the kayaks.

As noted in the lessons learned section there is still work to be done on the theoretical side as well. The most promising avenue would be to explore methods of creating the set of candidate paths for the greedy multistep algorithm to consider. Randomized path planning algorithms are promising in their ability to rapidly and efficiently sample space surrounding each sensor node. Another advantage of this

approach is that we could adjust the number of paths we wish to consider based the amount of time we have between steps and the length of our planning horizon. This would allow our planning to be as quick or as thorough as time permits.

The algorithms presented don't rely specifically on the structure of the tracking problem. It would be valuable to explore other problem spaces where mobile sensor nodes are deployed in sensing and estimation exercises. We would like to explore extending the greedy path selection paradigm to these other mobile sensing frameworks.

Appendix A

Simulation MATLAB Code

```
function [cost,reward,weightedCost,config,Pfinal] = ...
  exhaustiveSearch(sensor,numBearings,Hlin,P,R,C_,entR,...
  lambdai,lambdac,rat,curConfig,curCost,curReward,curWeightedCost)

% Performs an exhaustive (aka brute force) search over all possible
% solutions to the dynamic program over the next time step.

if sensor == 0
  cost = curCost;
  reward = curReward;
  weightedCost = curWeightedCost;
  config = curConfig;
  Pfinal = P;

else
  tmpWeightedCost = inf;
  for bearing = 1:numBearings
    % Calculate MI from sensor
    H = Hlin(:,:,sensor,bearing,1);

    S = H*P*H' + R(sensor);
    sensorMI = 0.5*(log(2*pi*exp(1)) + log((S))) - entR(sensor);
```

```
    sensorCost = rat*C_(sensor,bearing,1);

    % Add in information and cost for this sensor and bearing
    newWeightedCost = curWeightedCost ...
      + lambdac*sensorCost - lambdai*sensorMI;

    % Update the model
    Si = 1/S;
    Pnew = P - P*H'*Si*H*P;

    curConfig(sensor,bearing) = true;
    [cost,reward,weightedCost,config,Pfinal] = exhaustiveSearch(...
      sensor-1,numBearings,Hlin,Pnew,R,C_,entR,lambdai,lambdac,rat,...
      curConfig,curCost+sensorCost,curReward+sensorMI,newWeightedCost);

    if weightedCost < tmpWeightedCost
      tmpCost = cost;
      tmpReward = reward;
      tmpWeightedCost = weightedCost;
      tmpConfig = config;
      tmpP = Pfinal;
    end

    curConfig(sensor,bearing) = false;
  end

  cost = tmpCost;
  reward = tmpReward;
  weightedCost = tmpWeightedCost;
  config = tmpConfig;
  Pfinal = tmpP;
end
```

```
function [cost,reward,weightedCost,config,P] = ...
  greedySearch(numSensors,numBearings,Hlin,...
  P,R,C_,entR,lambdai,lambdac,rat)

% Performs single time step greedy approximation. Sequentially
% selects the the sensor that provides the minimum weightedCost,
% until a path has been selected for each sensor.

consider = true(numSensors,numBearings);
config = false(numSensors,numBearings);
reward = 0;
cost = 0;
weightedCost = 0;

while (any(consider)),
  % Calculate the information gain from each sensor
  sensorCost = NaN + zeros(numSensors,numBearings);
  sensorMI = NaN + zeros(numSensors,numBearings);

  for j = 1:numSensors
    for k = 1:numBearings,
      if (consider(j,k)),
        % Calculate MI from sensor
        H = Hlin(:,:,j,k,1);

        S = H*P*H' + R(j);
        sensorMI(j,k) = 0.5*(log(2*pi*exp(1))+log(S)) - entR(j);
        sensorCost(j,k) = rat*C_(j,k);
      end;
    end;
  end;

  % Select the sensor with the lowest cost
  [C,index] = min(lambdac*sensorCost - lambdai*sensorMI);
  [c,k] = min(C);
  j = index(k);
```

```
  % If the best option has a positive cost (i.e. the cost of
  % taking the measurement is more than the information gained)
  % then abort
  if (c > 0),
    break;
  end;

  % Update the model
  H = Hlin(:,:,j,k,1);
  Si = 1/(H*P*H' + R(j));
  P = P - P*H'*Si*H*P;

  % Mark sensor to be utilized
  config(j,k) = true;

  % Update the list of sensors to consider
  consider(j,:) = false;

  % Add in information and cost for this sensor and bearing
  reward = reward + sensorMI(j,k);
  cost = cost + sensorCost(j,k);
  weightedCost = weightedCost ...
    + lambdac*sensorCost(j,k) - lambdai*sensorMI(j,k);

end
```

```
function [cost,reward,weightedCost,config,Pfinal,pathLog] = ...
  greedySearchNStep(model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,...
  N,timeStep,locStep,sensor,bearing,pathLog,config,...
  curCost,curReward,curWeightedCost)
% Performs the multiple time step greedy approximation algorithm.
% All possible length N paths are expanded for each node. At each
% iteration the path with the lowest weightedCost is selected.
% Completes, when a path has been selected for each sensor.

numSensors = model.numSensors;
numBearings = model.numNeighbors;
F = model.F;
Qd = model.Qd;

% Finished recursing, return path results
if timeStep == N
  cost = curCost;
  reward = curReward;
  weightedCost = curWeightedCost;
  Pfinal = P;
  pathLog(timeStep) = locStep;

% Start initial recursion by going in each direction from each sensor
elseif timeStep == 0
  bestWeightedCost = inf;

  % for any sensor that hasn't already chosen a path
  for j = find(¬any(config'))
   for k = 1:numBearings
    [pathCost,pathReward,pathWeightedCost,config,pathPfinal,pathLog] ...
      = greedySearchNStep(model,Hlin,P,R,C_,entR,lambdai,lambdac, ...
      rat,N,timeStep+1,1,j,k,pathLog,config,...
      curCost,curReward,curWeightedCost);

      % set current best path
      if pathWeightedCost < bestWeightedCost
```

```
      % erase old best path
      if bestWeightedCost ≠ inf;
        pathLog(bestSensor,bestBearing,:) = 0;
      end
      bestWeightedCost = pathWeightedCost;
      bestSensor = j;
      bestBearing = k;
    else
      % not best path so erase from log
      pathLog(j,k,:) = 0;
    end;
  end;
end;

% Mark sensor to be utilized
config(bestSensor,bestBearing) = true;

% If all sensors have not chosen a path yet then run through
% the algorithm again with the remaining sensors
if ¬all(any(config'))
  [cost,reward,weightedCost,config,Pfinal,pathLog] = ...
    greedySearchNStep(model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,N,...
    0,0,0,0,pathLog,config,0,0,0);
else % every sensor has chosen a path, so finish
  Pfinal = P; % reset to original P
  reward = 0;
  cost = 0;
  weightedCost = 0;

  % Only taking first step along the path for each sensor
  for j = find(any(config')) % each sensor
    k = find(config(j,:)); % chosen bearing
    H = Hlin(:,:,j,k,1);
    S = H*Pfinal*H' + R(j);
    Si = 1/S; %
    Pfinal = Pfinal - Pfinal*H'*Si*H*Pfinal;
```

```
      % Add in information and cost for this sensor and bearing
      sensorReward = 0.5*(log(2*pi*exp(1))+log(S)) - entR(j);
      sensorCost = rat*C_(j,k,1);
      reward = reward + sensorReward;
      cost = cost + sensorCost;
      weightedCost = weightedCost ...
        + lambdac*sensorCost - lambdai*sensorReward;
    end;
    % end of recursion with final values being returned
  end;
else
% Recurse by going forward in the same direction or by staying still

  pathLog(sensor,bearing,timeStep) = locStep;

  % Use previously fixed measurements  at this timestep
  % plus this potential measurement
  curConfig = config;
  curConfig(sensor,bearing) = true; % add current sensor and bearing
  chosenSensors = any(curConfig');
  H = zeros(1,model.numState,numSensors);

  for j = find(chosenSensors)
    k = find(curConfig(j,:)); % chosen bearing
    H(:,:,j) = Hlin(:,:,j,k,pathLog(j,k,timeStep));

    S = H(:,:,j)*P*H(:,:,j)' + R(j);

    sensorReward = 0.5*(log(2*pi*exp(1))+log(det(S))) - entR(j);
    sensorCost = rat*C_(j,k,pathLog(j,k,timeStep));
    curReward = curReward + sensorReward;
    curCost = curCost + sensorCost;
    curWeightedCost = curWeightedCost ...
      + lambdac*sensorCost - lambdai*sensorReward;
```

```
    %update model and move forward one time step
    Si = 1/S;
    P = P - P*H(:,:,j)'*Si*H(:,:,j)*P;
  end
  P = F*P*F' + Qd;

  % continue along the same bearing (locStep+1)
  [tmpCost,tmpReward,tmpWeightedCost,config,tmpPfinal,pathLog] = ...
    greedySearchNStep(model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,...
    N,timeStep+1,locStep+1,sensor,bearing,pathLog,config,...
    curCost,curReward,curWeightedCost);

  % stay still for the next time step (locStep+0)
  [cost,reward,weightedCost,config,Pfinal,pathLog] = ...
    greedySearchNStep(model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,...
    N,timeStep+1,locStep,sensor,1,pathLog,config,...
    curCost,curReward,curWeightedCost);

  if tmpWeightedCost < weightedCost
    cost = tmpCost;
    reward = tmpReward;
    weightedCost = tmpWeightedCost;
    Pfinal = tmpPfinal;
  end
end
```

```
function [cost,reward,weightedCost,config,P] = independentSearch(...
  numSensors,numBearings,Hlin,P,R,C_,entR,lambdai,lambdac,rat)

% Performs the single time step decoupled approximation. Each sensor
% independently selects the measurement with the lowest weightedCost
% for the next time step

config = false(numSensors,numBearings);
reward = 0;
cost = 0;
weightedCost = 0;

sensorCost = NaN + zeros(numSensors,numBearings);
sensorMI = NaN + zeros(numSensors,numBearings);

% Calculate the information gain from each sensor
for j = 1:numSensors
  for k = 1:numBearings,
    % Calculate MI from sensor
    H =(Hlin(:,:,j,k,1));

    S = H*P*H' + R(j);
    sensorMI(j,k) = 0.5*(log(2*pi*exp(1))+log(S)) - entR(j);
    sensorCost(j,k) = rat*C_(j,k);
  end;
end;

% Select the sensor with the lowest cost
[C,I] = min(lambdac*sensorCost - lambdai*sensorMI,[],2);

for j = 1:numSensors
  % Mark sensor to be utilized
  k = I(j);
  config(j,k) = true;

  % Update the model
```

```
  H = Hlin(:,:,j,k,1);
  S = H*P*H' + R(j);
  Si = 1/S; % if numMeas > 1, this would have to be inv(.)
  P = P - P*H'*Si*H*P;

  % Add in information and cost for this sensor and bearing
  sensorReward = 0.5*(log(2*pi*exp(1))+log(S)) - entR(j);
  reward = reward + sensorReward;
  cost = cost + sensorCost(j,k);
  weightedCost = weightedCost ...
    + lambdac*sensorCost(j,k) - lambdai*sensorReward;
end;
```

```
function [cost,reward,weightedCost,config,Pfinal] = ...
independentSearchNStep(model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,N,...
  timeStep,locStep,sensor,bearing,curConfig,...
  curCost,curReward,curWeightedCost)

% Performs multiple time step decoupled approximation algorithm. Each
% sensor indepentently expands potential paths and chooses the path
% with the minimum pathWeightedCost.

numSensors = model.numSensors;
numBearings = model.numNeighbors;
F = model.F;
Qd = model.Qd;

config = false(numSensors,numBearings);
reward = 0;
cost = 0;
weightedCost = 0;

% Finished recursing, return path results
if timeStep == N
  cost = curCost;
  reward = curReward;
  weightedCost = curWeightedCost;
  config = curConfig;
  Pfinal = P;

% Start initial recursion by going in each direction from each sensor
elseif timeStep == 0
  pathWeightedCost = zeros(1,numBearings);
  for j = 1:numSensors
    for k = 1:numBearings,
      [pathCost,pathReward,pathWeightedCost(k),pathConfig,pathPfinal] ...
        = independentSearchNStep(model,Hlin,P,R,C_,entR,...
        lambdai,lambdac,rat,N,1,1,j,k,...
        curConfig,curCost,curReward,curWeightedCost);
```

```
    end;
    % Select the path for each sensor with the lowest cost
    % sum over all N steps
    [C,k] = min(pathWeightedCost);

    % Mark sensor to be utilized
    config(j,k) = true;

    % Assuming only taking first step along the path

    % Update the model
    H = Hlin(:,:,j,k,1);
    S = H*P*H' + R(j);
    Si = 1/S;
    Pfinal = P - P*H'*Si*H*P;

    % Add in information and cost for this sensor and bearing
    sensorReward = 0.5*(log(2*pi*exp(1))+log(S)) - entR(j);
    sensorCost = rat*C_(j,k,1);
    reward = reward + sensorReward;
    cost = cost + sensorCost;
    weightedCost = weightedCost ...
      + lambdac*sensorCost - lambdai*sensorReward;
  end;

% Recurse by going forward in the same direction or by staying still
else
  % Calculate MI from sensor
  H = Hlin(:,:,sensor,bearing,locStep);

  S = H*P*H' + R(sensor);

  sensorReward = 0.5*(log(2*pi*exp(1))+log(S)) - entR(sensor);
  sensorCost = rat*C_(sensor,bearing,1);
  curReward = curReward + sensorReward;
  curCost = curCost + sensorCost;
```

```
  curWeightedCost = curWeightedCost ...
    + lambdac*sensorCost - lambdai*sensorReward;

  %update model and move forward one time step
  Si = 1/S;
  P = P - P*H'*Si*H*P;
  P = F*P*F' + Qd;

  % continue along the same bearing
  [tmpCost,tmpReward,tmpWeightedCost,config,tmpPfinal] = ...
    independentSearchNStep(model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,...
    N,timeStep+1,locStep+1,sensor,bearing,curConfig,...
    curCost,curReward,curWeightedCost);

  % stay still for the next time step
  [cost,reward,weightedCost,config,Pfinal] = independentSearchNStep(...
    model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,...
    N,timeStep+1,locStep,sensor,bearing,curConfig,...
    curCost,curReward,curWeightedCost);

  if tmpWeightedCost < weightedCost
    cost = tmpCost;
    reward = tmpReward;
    weightedCost = tmpWeightedCost;
    Pfinal = tmpPfinal;
  end
end
```

```
function [reward,cost,selection,seqEnt,model] = ...
  dynamicProgram(P,mu,model,N,lambdai,lambdac,rat,searchType)
% Implements an approximate dynamic program that utilizes the
% algorithm indicated by searchType.
%
% Returns the reward and cost obtained in the optimization
% and the selected sensors positions for the next time step
%
% P contains initial covariance (previous decision time step)
% mu contains initial linearization mean (prev decis step)
% model contains dynamics and sensing model parameters
% N is the horizon length
% lambdai is largange multiplier for information
% lambdac is lagrange multiplier for cost

% Extract model parameters from structure
numState = model.numState;
numSensors = model.numSensors;
numNeighbors = model.numNeighbors;
F = model.F;
Qd = model.Qd;
Qdi = inv(Qd);

% Propagate mean and covariance to new time step
mu = F*mu;
P = F*P*F' + Qd;

% Extend nominal trajectory
x0 = zeros(numState,N);
mu0 = mu;
for i = 1:N,
  x0(:,i) = mu0;
  mu0 = F*mu0;
end

% Find locations of virtual nodes neighboring the current nodes,
```

```
% including the virtual node representing the current position.

numBearings = numNeighbors; % all neighbors and staying still
vSensorNodeLoc = zeros(2,numSensors,numBearings,N);
% motion costs
C_ = zeros(numSensors,numBearings,N);
dist = sqrt(sum(model.neighborLocDiff.*model.neighborLocDiff));

for j = 1:numSensors,
  for k = 1:N,
    vSensorNodeLoc(:,j,:,k) = repmat(model.sensor{j}.location,1,...
    numBearings) + k*model.neighborLocDiff;
    C_(:,:,k) = repmat(k*dist,numSensors,1);
  end
end

% Calculate linearizations of measurement model  for each virtual node
% along given target trajectory
Hlin = zeros(1,numState,numSensors,numBearings,N);
HlinCurrent = zeros(1,numState,numSensors);
R = zeros(1,numSensors);
entR = zeros(1,numSensors);
for j = 1:numSensors,
  for k = 1:numBearings,
    for l = 1:N,
      numMeas = size(model.sensor{j}.R,1);
      if (numMeas ~= 1),
        error('Code assumes numMeas == 1 always');
      end;
      Hlin(:,:,j,k,l) = powerMeasLin(x0(:,l),vSensorNodeLoc(:,j,k,l));
    end;
  end;
  HlinCurrent(:,:,j) = powerMeasLin(x0(:,1),model.sensor{j}.location);
  R(:,j) = model.sensor{j}.R;
  entR(:,j) = model.sensor{j}.entR;
end;
```

```
% Constant for use in entropy calculations
log2pie = log(2*pi*exp(1));

% Calculate covariance matrix if we take all measurements to get
% an upper bound
Pi = inv(P);
aPriInf = log(det(Pi));
for j = 1:numSensors,
  for k = 1:numBearings,
    for l = 1:N,
      H = Hlin(:,:,j,k,l);
      Ri = 1/R(j); % Assumes numMeas = 1, inv(.) otherwise
      Pi = Pi + H'*Ri*H;
    end;
  end;
end;
aPostInf = log(det(Pi));
maxMI = 0.5*(aPostInf-aPriInf);

curMI = 0;
curCost = 0;
config = false(numSensors,numBearings);

switch searchType
  case 'exhaustive',
    % Brute force search over all possible sensor node configurations
    [curCost,curMI,weightedCost,config,P] = ...
      exhaustiveSearch(numSensors,numBearings,Hlin,P,R,C_, ...
      entR,lambdai,lambdac,rat,config,0,0,0);

  case 'greedy',
    % Greedy single step approximation algorithm
    [curCost,curMI,weightedCost,config,P] = greedySearch(numSensors,...
      numBearings,Hlin,P,R,C_,entR,lambdai,lambdac,rat);
```

```
  case 'greedyNStep',
    % Greedy multistep approximation algorithm over time horizon of N
    pathLog = zeros(numSensors,numBearings,N);
    [curCost,curMI,weightedCost,config,P,pathLog] = ...
      greedySearchNStep(model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,N,...
      0,0,0,0,pathLog,config,0,0,0);

  case 'unconstrainedNStep',
    % Greedy multistep over time horizon of N without cost constraints
    pathLog = zeros(numSensors,numBearings,N);
    [curCost,curMI,weightedCost,config,P,pathLog] = ...
    greedySearchNStep(model,Hlin,P,R,zeros(size(C_)),entR,...
    lambdai,lambdac,rat,N,0,0,0,0,pathLog,config,0,0,0);

  case 'independent',
    % Decoupled single step algorithm
    [curCost,curMI,weightedCost,config,P] = independentSearch(...
      numSensors,numBearings,Hlin,P,R,C_,entR,lambdai,lambdac,rat);

  case 'independentNStep',
    % Decoupled multistep algorithm over time horizon N
    [curCost,curMI,weightedCost,config,P] = independentSearchNStep(...
      model,Hlin,P,R,C_,entR,lambdai,lambdac,rat,N,...
      0,0,0,0,config,0,0,0);
end

%selectedVNodes = repmat(config,1,N);
selectedVNodes = config;

% Calculate entropy
seqEnt = 0.5*(numState*log2pie + log(det(P)));

reward = curMI;

cost = 0;
```

```
for i = 1:numSensors
cost = cost + C_(i,selectedVNodes(i,:),1);
end

% update bearings and move sensors to the selected locations
for j = 1:numSensors,
    newLoc = vSensorNodeLoc(:,j,selectedVNodes(j,:,:)==true);
    % if no neighbors had viable moves then stay still
    if isempty(newLoc)
      newLoc = model.sensor{j}.location;
    end
    % if sensor moved then update bearing and location
    if newLoc ≠ model.sensor{j}.location,
        dx = newLoc(1) - model.sensor{j}.location(1);
        dy = newLoc(2) - model.sensor{j}.location(2);
        model.sensor{j}.bearing = acos(dx/sqrt(dx^2+dy^2));
        if dy < 0, % bearing is in 3rd or 4th quadrant
            model.sensor{j}.bearing = 2*pi - model.sensor{j}.bearing;
        end;
        % move the location only the distance that can be traveled
        % in one time step.
        scale = model.maxSpeed*model.T/sqrt(dx^2+dy^2);
        model.sensor{j}.location = ...
          model.sensor{j}.location + scale*[dx,dy]';
    end;
end;

% mark all sensors as selected
selection = true(1,numSensors);
```

```
function [motionCostLog,leaderLog,estEntLog,posErrorLog,lambdaLog,...
  constrLog] = estimationPlanner(model,sim,searchType,N,...
  costConstr,fname,doPlots)

% Maintains a particle filter to represent the current estimate of the
% target state. Calls the dynamic program for planning the sensor
% node configuration for the next time step.

if (exist('mov','var')),
  mov = close(mov);
  clear mov;
end

lambdac = 2e-4;
lambdai = 1;
rat = 1;

lambdacIncrFact = 1.2;
lambdacDecrFact = 1.2;
lambdacMax = 5e-2;
lambdacMin = 1e-5;

disp(['Simulation commenced ' datestr(now,'HH:MM:SS dd mmm yy')]);

numParticle = 1000;

% Assign variables outside model
F = model.F;
numState = model.numState;
Qd = model.Qd;
Qdch = model.Qdch;
QdchInv = model.QdchInv;
numNoise = size(model.Qdch,2);
Pk = Qd;

% Particle distribution is centered around truth mean, with noise added
```

```
initCov = diag([10 1 10 1]); % Covariance of particles
initCovCh = chol(initCov)';
x = repmat(sim.xtlog(:,1),1,numParticle) + ...
  + initCovCh*randn(numState,numParticle);
w = ones(1,numParticle)/numParticle;
clear initCov initCovCh;

% Define masks for positions and velocities in state
posMask = logical([1 0 1 0]);
velMask = logical([0 1 0 1]);

% Extract sensor locations and bearings from model
numSensors = model.numSensors;
sensorLoc = zeros(2,numSensors);
sensorBear = zeros(1,numSensors);
for k = 1:numSensors,
  sensorLoc(:,k) = model.sensor{k}.location;
  sensorBear(:,k) = model.sensor{k}.bearing;
end;

numPlot = 200; % Plot the first numPlot particles

% Prepare figure for result visualization
if (get(0,'ScreenDepth') > 0 && doPlots),
  fig = figure(1);
  %fname = '';
  if (exist('fig','var')),
    clf;
    set(gcf,'Renderer','painters','DoubleBuffer','on',...
      'Position',[960 340 540 800]);
    movegui(gcf,'northeast');
    % Prepare movie
    if (exist('fname','var') && ¬isempty(fname)),
      if exist(fname,'file')==2
        delete(fname);
      end;
```

```
      palette=[0 0 1; 0 1 0; 0 1 1; 1 0 0;...
          1 0 1; 1 1 0; 0 0.5 0; gray(9)];
      fr.colormap = palette;
      mov=avifile(fname,'colormap',fr.colormap,'compression','RLE',...
        'fps',5,'keyframe',0.25,'quality',100);
      clear palette fname;
    end;
  end;
end;

FN = F;
for i = 1:N-1,
  FN = F*FN;
end;

Z = cell(numSensors,1);

numStep = sim.numStep;
motionCostLog = NaN + zeros(1,numStep);
estEntLog = motionCostLog;
posErrorLog = motionCostLog;
leaderLog = motionCostLog;

lambdaLog = motionCostLog;
constrLog = motionCostLog;


% Run simulation until time lapses or target leaves region populated
% with sensors
for t = 1:numStep,
  tic;

  xt = sim.xtlog(:,t);

  % Calculate mean and covariance at old time and propagate to new
  % time
```

```
mu = x*w';
P = (x.*repmat(w,numState,1))*x' - mu*mu';

% Generate measurements for each sensor
for k = 1:numSensors,
  sim.sensor{k}.z(:,t) = feval(model.sensor{k}.model,...
    sim.xtlog(:,t),model.sensor{k}.location) + ...
    + model.sensor{k}.Rch*randn(model.sensor{k}.numMeas,1);
end;

% Use constrained DP method
[reward,cost,selection,seqEnt,model] = ...
  dynamicProgram(P,mu,model,N,lambdai,lambdac,rat,searchType);

if (¬isnan(costConstr)),

  % Update dual variables
  if (cost > costConstr),
    lambdac = min(lambdac*lambdacIncrFact,lambdacMax);
  else
    lambdac = max(lambdac/lambdacDecrFact,lambdacMin);
  end;

  lambdaLog(t) = lambdac;
  constrLog(t) = cost - costConstr;

else

  minEnt = min(seqEnt);

  % Update dual variables
  if (minEnt > infConstr),
    lambdai = min(lambdai + lambdaiIncr,lambdaiMax);
  else
    lambdai = max(lambdai-lambdaiDecr,lambdaiMin);
  end;
```

```
  lambdaLog(t) = lambdai;
  constrLog(t) = minEnt - infConstr;
end;

% Add in cost of moving nodes
if (t == 1),
  motionCostLog(t) = cost;
else
  motionCostLog(t) = motionCostLog(t-1) + cost;
end;

% Retrieve measurement for the chosen sensor
for i = 1:numSensors,
  if (selection(i)),
    Z{i} = sim.sensor{i}.z(:,t);
  else
    Z{i} = [];
  end;
end;

% Update particles with sensor
if (sum(selection) == 0),
  % No update -- just propagate
  x = F*x + Qdch*randn(numNoise,numParticle);
else
  % Propagate and update
  [x,w] = particleUpdate(x,w,model,Z);
end;

% Update particle covariance
Pk = F*Pk*F' + Qd;

% Estimate entropy of position for performance measure
[partSubset,psw] = resample(x,w,numPlot);
[estEntLog(t),ml] = difEntropy(partSubset(posMask,:),psw);
```

```
posErrorLog(t) = sqrt(sum((ml-xt(posMask)).^2));

% Only resample after number of effective particles decreases
Neff = 1/sum(w.^2);
if (Neff < 0.25*numParticle),
  [x,w] = resample(x,w,numParticle);
  Pk = Qd; % Reset particle covariance
end;

graphicX = zeros(3,numSensors);
graphicY = zeros(3,numSensors);
scale = 1/15*(model.trackingRegion(2)-model.trackingRegion(1));
for k = 1:numSensors,
  % update sensor locations and bearings
  sensorLoc(:,k) = model.sensor{k}.location;
  sensorBear(:,k) = model.sensor{k}.bearing;

  % create a graphic for each sensor
  diffX = scale*[cos(sensorBear(k)), ...
    -1/2*cos(sensorBear(k)), ...
    1/2*cos(sensorBear(k)),]';
  diffY = scale*[sin(sensorBear(k)), ...
    1/2*sin(sensorBear(k)), ...
    -1/2*sin(sensorBear(k))]';
  graphicX(:,k) = sensorLoc(1,k)+diffX;
  graphicY(:,k) = sensorLoc(2,k)+diffY;
end;

% Plot sensor field and targets
if (exist('fig','var')),

  % Plot everything
  clf;
  subplot('position',[0.15 0.45 0.8 0.533]);
  handx = plot(partSubset(1,:),partSubset(3,:),'g.');hold on;
  hands = plot(sensorLoc(1,¬selection),sensorLoc(2,¬selection),'k.');
```

```
    handol = ellipse(3*ones(1,numSensors),1*ones(1,numSensors)...
      ,sensorBear,sensorLoc(1,:),sensorLoc(2,:),'r');
    handxe = plot(ml(1),ml(2),'rx','MarkerSize',12,'LineWidth',2);
    handxt = plot(xt(1),xt(3),'k+','MarkerSize',12,'LineWidth',2);
    hold off;
    axis square;
    axis(model.trackingRegion);
    set(handol,'MarkerSize',10,'MarkerFaceColor',[1 0 0],...
      'MarkerEdgeColor',[1 1 1]);

    subplot('position',[0.15 0.05 0.8 0.15]);
    plot((0:numStep-1),motionCostLog,'r-');
    ylabel('Motion Cost');
    axis([0 (numStep-1) 0 1000]);

    subplot('position',[0.15 0.25 0.8 0.15]);
    plot((0:numStep-1),estEntLog,'r-');
    ylabel('Estimate Entropy');
    axis([0 (numStep-1) 0 10]);

    drawnow;
    if (exist('mov','var')),
      frrgb = getframe(gcf);
      fr.cdata = rgb2ind(frrgb.cdata,fr.colormap);
      size(fr.cdata);
      mov = addframe(mov,fr);
    end;
  end;
  if mod(t,numStep/10) == 0
      set(gcf,'PaperPositionMode','auto');
      print('-dpdf',['figures/sim-t-' num2str(t)]);
  end

end;

if (exist('mov','var')),
```

```
  mov = close(mov);
  clear mov;
end;

return;
```

```
function [model,sim] = makeModel(numSensors,regionSize,...
  targetSpeed,sensorSpeed,timeScale)
% Make the model for the sensor nodes and target object

numNeighbors = 7; %including self (ie staying still)
maxStep = 200;

model.maxSpeed = sensorSpeed;
model.T = timeScale;

% set up grid of virtual nodes
model.d = sensorSpeed*timeScale; % grid spacing
% Differential distance of neighboring virtual nodes in a hexagonal grid
model.neighborLocDiff = model.d*[0,0; 1,0; 1/2,sqrt(3)/2; ...
  -1/2,sqrt(3)/2; -1,0; -1/2,-sqrt(3)/2; 1/2,-sqrt(3)/2]';

% Define measurement model
model.numSensors = numSensors;
model.numNeighbors = numNeighbors;
model.sensor = cell(numSensors,1);
model.trackingRegion = [0 regionSize 0 regionSize];
regionSize = diag([model.trackingRegion(2) - model.trackingRegion(1);
  model.trackingRegion(4) - model.trackingRegion(3)]);
regionOffset = [model.trackingRegion(1); model.trackingRegion(3)];
sensorLoc = regionSize*rand(2,numSensors) ...
  + repmat(regionOffset,1,numSensors);
sensorBearing = 2*pi*rand(1,numSensors);

for k = 1:numSensors,
  model.sensor{k}.location = sensorLoc(:,k);
  model.sensor{k}.bearing = sensorBearing(k);
  model.sensor{k}.model = @powerMeas;
  model.sensor{k}.linearize = @powerMeasLin;
  model.sensor{k}.numMeas = 1;
  model.sensor{k}.R = 1;
  model.sensor{k}.Rch = chol(model.sensor{k}.R)';
```

```
  model.sensor{k}.RchInv = inv(model.sensor{k}.Rch);
  model.sensor{k}.entR=0.5*(model.sensor{k}.numMeas*log(2*pi*exp(1)) ...
    + log(det(model.sensor{k}.R)));
end

% Initial target position
numState = 4;
speed = targetSpeed;
sim.xtlog = zeros(numState,maxStep+1);

% start in at a random spot
sim.xtlog(1,1) = model.trackingRegion(1+round(rand(1)));
sim.xtlog(3,1) = model.trackingRegion(3+round(rand(1)));
% velocity towards the interior of the region
sim.xtlog(2,1) = speed*(-2*sim.xtlog(1,1)/regionSize(1,1)+1);
sim.xtlog(4,1) = speed*(-2*sim.xtlog(3,1)/regionSize(2,2)+1);

% Define continuous time dynamics model
Fs = [0 1; 0 0];
Z = zeros(2,2);
Fc = [Fs Z; Z Fs];

% Convert to equivalent discrete time model
model.F = expm(Fc*model.T);
model.numState = numState;
q = 0.01;

% Calculated by hand from Maybeck p171 eq 4-127b
Qs = q*[model.T^3/3 model.T^2/2; model.T^2/2 model.T];
model.Qd = [Qs Z; Z Qs];
model.Qdch = chol(model.Qd)';
model.QdchInv = inv(model.Qdch);
model.numNoise = size(model.Qdch,2);
clear Fs Z Fc q Qs;

% Add noise to true target location
```

```
% xt = xt + initCovCh*randn(numState,1);

% Create structure for storing measurements
sim.sensor = cell(numSensors,1);
for k = 1:numSensors,
  sim.sensor{k}.z = zeros(model.sensor{k}.numMeas,maxStep+1);
end;

% Create target trajectory
t = 1;
while (t ≤ maxStep),
  t = t + 1;

  % Propagate truth model
  sim.xtlog(:,t) = model.F*sim.xtlog(:,t-1) + ....
    model.Qdch*randn(model.numNoise,1);
end

% Trim unused measurement positions
sim.xtlog = sim.xtlog(:,2:t);
for k = 1:numSensors,
  sim.sensor{k}.z = sim.sensor{k}.z(:,2:t);
end;

sim.numStep = t-1;
```

```
function z = powerMeas(x,loc)
% Produce simulated range measurement between x and loc

b = 100;
a = 20*b;

posMask = [1 3];
numParticle = size(x,2);

dif = x(posMask,:) - repmat(loc,1,numParticle);
z = a./(b + sum(dif.^2,1));
```

```
function H = powerMeasLin(x,loc)
% Create the linearized version of the measurement model

% Get 'a' and 'b' from their current values in powerMeas
alpha = powerMeas([0 0 0 0]',[0 0]');
gamma = powerMeas([30 0 0 0]',[0 0]');

b = 900*gamma/(alpha-gamma);
a = b*alpha;

posMask = [1 3];
numParticle = size(x,2);
numState = size(x,1);

H = zeros(1,numState,numParticle);

dif = x(posMask,:) - repmat(loc,1,numParticle);

% This is the derivative of h(x) with respect to the interim result
% c = dif'.dif
dhdc = -a./(b + sum(dif.^2,1)).^2;

for i = 1:numParticle,
  H(:,posMask,i) = dhdc(i)*2*dif(:,i)';
end;
```

```
function [x,w] = particleUpdate(x,w,model,Z)
% Update the each of the particles in the particle filter for the next
% time step.

F = model.F;
Qd = model.Qd;
QdchInv = model.QdchInv;
numState = model.numState;
numParticle = size(x,2);
numSensors = length(Z);

% Propagate particles noise-free, and set kernel size
x = F*x;
Pk = Qd;

% Linearize measurement model for each particle
lengths = zeros(1,numSensors);
for (i = 1:numSensors),
  if (¬isempty(Z{i})),
    lengths(i) = length(Z{i});
  end;
end;
numMeas = sum(lengths);

% Calculate model linearizations and predicted (mean) measurements for
% each particle
Hlin = zeros(numMeas,numState,numParticle);
zp = zeros(numMeas,numParticle);
R = zeros(numMeas,numMeas);
z = zeros(numMeas,1);
j = 1;
for (i = 1:numSensors),
  if (¬isempty(Z{i})),
    Hlin(j:j+lengths(i)-1,:,:) = ...
      feval(model.sensor{i}.linearize,x,model.sensor{i}.location);
    zp(j:j+lengths(i)-1,:) = ...
```

```
      feval(model.sensor{i}.model,x,model.sensor{i}.location);
    R(j:j+lengths(i)-1,j:j+lengths(i)-1) = model.sensor{i}.R;
    z(j:j+lengths(i)-1) = Z{i};
    j = j + lengths(i);
  end;
end;

% Perform KF update for each linearization
xn = zeros(numState,numParticle);

for (i = 1:numParticle),
  % Perform kalman filter update using the linearization
  H = Hlin(:,:,i);
  K = Pk*H'*inv(H*Pk*H' + R);
  xupd = x(:,i) + K*(z - zp(:,i));
  Pupd = Pk - K*H*Pk;
  Pupdch = chol(Pupd)';

  % Generate a new particle location drawn from the distribution
  % given by the kalman filter update
  wd = randn(numState,1); % noise
  xn(:,i) = xupd + Pupdch*wd;

  wn(i) = w(i)*exp(0.5*sum(wd.^2))*prod(diag(Pupdch));
end;

RchInv = inv(chol(R)');

% Recalculate predicted measurements for updated particle values
zp = zeros(numMeas,numParticle);
j = 1;
for i = 1:numSensors,
  if (¬isempty(Z{i})),
    zp(j:j+lengths(i)-1,:) = ...
      feval(model.sensor{i}.model,xn,model.sensor{i}.location);
    j = j + lengths(i);
```

```
  end;
end;

% Perform the reweighting
err1 = RchInv*(repmat(z,1,numParticle) - zp);
err2 = QdchInv*(xn - x);
wn = wn .* exp(-0.5*(sum(err1.^2,1) + sum(err2.^2,1)));

% Renormalize
wns = sum(wn);
if (wns > 0),
  w = wn/wns;
else
  w(:) = 1/numParticle;
  warning('All particle weights truncated to zero; reset uniformly.');
end;
x = xn;
```

```
function [xnew,wnew] = resample(x,w,numParticle)
%RESAMPLE  Resample a selection of particles
% Format: [xnew,wnew] = resample(x,w,numParticle)
% columns of x contain particles
% w contains weights
% numParticle indicates the number of particles to resample from x
% xnew and wnew output resampled particles and uniform weights
%   respectively
% Uses semi-deterministic method

numState = size(x,1);
xnew = zeros(numState,numParticle);
cum = cumsum(w);
% Make sure cumulative distribution sums to at least one (may be less
% due to numerical round-off)
if (cum(end) < 1),
  cum(end) = 1;
end;

% Use semi-deterministic resampling method -- pick particle locations
% to be regularly spaced
loc = (rand + (0:(numParticle-1)))/numParticle;

i = 1;
for (j = 1:numParticle),
  % Move along until we reach new particle location
  while (loc(j) > cum(i)),
    i = i + 1;
  end;

  % Generate the particle
  xnew(:,j) = x(:,i);
end;

% Assign new weights equally
wnew = ones(1,numParticle)/numParticle;
```

```
function [h,ml] = difEntropy(x,w)
%Differential entropy estimate for particle representation
% Columns of x contain samples of variable
% w contains sample weights
% h returns entropy estimate
% ml returns coarse maximum likelihood estimate
% Rule of thumb kernel size is used

numParticle = size(x,2);
dimx = size(x,1);

% Calculate kernel sizes using rule of thumb
ex = x*w';
ex2 = x.^2*w';
sigx = sqrt(numParticle/(numParticle-1)*(ex2 - ex.^2));
hx = sigx*numParticle^(-1/(4+dimx));

% Calculate kernel matrix for x
Kx = zeros(numParticle,numParticle);
for (i = 1:dimx),
  rm = repmat(x(i,:)/hx(i),numParticle,1);
  Kx = Kx + (rm - rm').^2;
end;
Kx = exp(-0.5*Kx)/sqrt(prod(2*pi*hx.^2));

% Calculate entropy estimate
den = (Kx*w');
mask = w > 0;
h = -sum(w(mask)' .* log(den(mask)));

% Return ML estimate for x as the one with the largest sample point
% value
[v,i] = max(den);
ml = x(:,i);
return;
```

Bibliography

[1] M.S. Arulampalam, S. Maskell, N. Gordon, T. Clapp, D. Sci, T. Organ, and S.A. Adelaide. A tutorial on particle filters for online nonlinear/non-GaussianBayesian tracking. *IEEE Transactions on signal processing*, 50(2):174–188, 2002.

[2] M.R. Benjamin, J.J. Leonard, H. Schmidt, and P.M. Newman. An overview of MOOS-IvP and a brief users guide to the IvP Helm autonomy software. *Massachusetts Institute of Technology, MIT CSAIL, Tech. Rep. TR-2009-28-07*, 2009.

[3] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 2000.

[4] W.F. Caselton and J.V. Zidek. Optimal monitoring network designs. *Statistics & Probability Letters*, 2(4):223–227, 1984.

[5] C. Chekuri and M. Pal. A recursive greedy algorithm for walks in directed graphs. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 245–253, 2005.

[6] A.S. Chhetri, D. Morrell, and A. Papandreou-Suppappola. Scheduling multiple sensors using particle filters in target tracking. In *2003 IEEE Workshop on Statistical Signal Processing*, pages 549–552, 2003.

[7] T.M. Cover and J.A. Thomas. *Elements of information*. John Wiley & Sons, NY, USA, 1991.

[8] Noel A. C. Cressie. *Statistics for Spatial Data*. Wiley, 1991.

[9] J. Curcio, J. Leonard, and A. Patrikalakis. SCOUT-a low cost autonomous surface platform for research in cooperative autonomy. In *OCEANS, 2005. Proceedings of MTS/IEEE*, pages 725–729, 2005.

[10] E. Ertin, J. Fisher, and L. Potter. Maximum mutual information principle for dynamic sensor query problems. In *Information Processing in Sensor Networks*, pages 558–558. Springer, 2003.

[11] C. Guestrin, A. Krause, and A.P. Singh. Near-optimal sensor placements in gaussian processes. In *Proceedings of the 22nd international conference on Machine learning*, pages 265–272. ACM, 2005.

[12] G.M. Hoffmann and C.J. Tomlin. Mobile sensor network control using mutual information methods and particle filters. *IEEE Transactions on Automatic Control*, 2009.

[13] C.W. Ko, J. Lee, and M. Queyranne. An exact algorithm for maximum entropy sampling. *Operations Research*, 43(4):684–691, 1995.

[14] A. Krause and C. Guestrin. Near-optimal nonmyopic value of information in graphical models. In *Proc. of Uncertainty in AI*. Citeseer, 2005.

[15] A. Krause, C. Guestrin, A. Gupta, and J. Kleinberg. Near-optimal sensor placements: Maximizing information while minimizing communication cost. In *Proceedings of the 5th international conference on Information processing in sensor networks*, page 10. ACM, 2006.

[16] A. Krause, A. Singh, and C. Guestrin. Near-optimal sensor placements in Gaussian processes: Theory, efficient algorithms and empirical studies. *The Journal of Machine Learning Research*, 9:235–284, 2008.

[17] P.S. Maybeck. *Stochastic models, estimation, and control (volume 1 & 2)*, volume 2. Navtech Book, 1994.

[18] M.D. McKay, R.J. Beckman, and W.J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, May 1979.

[19] G.L. Nemhauser, L.A. Wolsey, and M.L. Fisher. An analysis of approximations for maximizing submodular set functionsI. *Mathematical Programming*, 14(1):265–294, 1978.

[20] K. Pister, J. Kahn, and B. Boser. Smart dust. *keynote address, IPSN*, 3, 2003.

[21] G.J. Pottie and W.J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, 2000.

[22] N. Ramakrishnan, C. Bailey-Kellogg, S. Tadepalli, and V.N. Pandey. Gaussian processes for active data mining of spatial aggregates. In *Proceedings of the SIAM International Conference on Data Mining*, 2005.

[23] N. Roy and C. Earnest. Dynamic action spaces for information gain maximization in search and exploration. In *American Control Conference, 2006*, page 6, 2006.

[24] A. Singh, A. Krause, C. Guestrin, and W. Kaiser. Efficient informative sensing using multiple robots. *Journal of Artificial Intelligence Research*, 34(1):707–755, 2009.

[25] B. Warneke, M. Last, B. Liebowitz, and K.S.J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, 2001.

[26] J.L. Williams. *Information theoretic sensor management*. PhD thesis, Massachusetts Institute of Technology, 2007.

[27] J.L. Williams, J.W. Fisher, and A.S. Willsky. Approximate dynamic programming for communication-constrained sensor network management. *IEEE Transactions on signal Processing*, 55(8):4300–4311, 2007.

[28] J.L. Williams, J.W. Fisher III, and A.S. Willsky. Performance guarantees for information theoretic active inference. *AI & Statistics (AISTATS)*, 2007.

[29] F. Zhao, J. Shin, and J. Reich. Information-driven dynamic sensor collaboration for tracking applications. *IEEE Signal Processing Magazine*, 19(2):61–72, 2002.