Fast Methods for Inference in Graphical Models

and Beat Tracking the Graphical Model Way

by

Dustin Lang

B.Sc., The University of British Columbia, 2002

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

.....

.....

THE UNIVERSITY OF BRITISH COLUMBIA

October 8, 2004

Abstract

This thesis presents two related bodies of work. The first is about methods for speeding up inference in graphical models, and the second is an application of the graphical model framework to the beat tracking problem in sampled music.

Graphical models have become ubiquitous modelling tools; they are commonly used in computer vision, bioinformatics, coding theory, and speech recognition, and are central to many machine learning techniques. Graphical models allow statistical independence relationships between random variables to be expressed in a flexible, powerful, and intuitive manner. Given observations, there are standard algorithms to compute probability distributions over unknown states (marginals) or to find the most likely configuration (maximum *a posteriori*, MAP, state). However, if each node in the graphical model has N states, then these computations cost $O\left(N^2\right)$. This is a particular concern when dealing with continuous (or large discrete) state spaces. In such state spaces, Monte Carlo methods are of great use; these methods typically produce better answers as N increases. There is therefore a need for algorithms to speed up these computations.

We review the graphical model framework and explain the algorithms that we consider: Belief Propagation, Maximum-Belief Propagation, and Particle Filtering and Smoothing. We review existing fast methods for speeding up these computations, and present a new fast method, some analysis, corrections, and improvements to existing methods, and an empirical evaluation of the fast methods with respect to a variety of parameters. Finally, we point out other applications to which this fast machinery can be applied.

The second body of work is an application of the graphical model framework to

a common problem in music analysis: beat tracking. The goal of beat tracking is to determine the tempo track of a piece of sampled music, and to determine the points in time at which beats occur. While fairly simple, our graphical model performs remarkably well on a difficult and varied set of songs.

Contents

List of Figures

Acknowledgements

Many people have been essential to making this thesis happen while keeping me more or less sane.

First of all, I want to thank my mom and dad and my brother. They have always supported me in whatever I have chosen to do. I know that no matter what happens they will be there to catch me if I fall and cheer me on when I cross the finish line, and that means a lot.

I have been fortunate to work in the Dark Side of the LCI. I count my labmates not as coworkers but as friends, and it's a pleasure to arrive at work and be surrounded by friends. In particular, I have greatly enjoyed working with Mike Klaas. He's not only very clever, but also has a great sense of humour and a very acceptable level of appreciation for the wonders of caffeine.

I also want to thank Nando de Freitas, who has been a brilliant supervisor. Sometimes we don't understand each other, but he is always ready to explain or be explained to (usually with math!).

I thank David Lowe for reading this thesis during a very busy week, and for providing many useful comments.

I am fortunate to live with a great group of guys at The Orphanage. After a rough day at school, it is a fantastically good thing to return to a warm house, a home-cooked meal, good companions, and a game of Settlers.

Finally, I want to thank Micheline for her amazing ability to brighten up a room, and my life.

Chapter 1

Welcome

Traditionally, the first chapter of a thesis is called the Introduction. I'm not a traditionalist, and my second chapter is an introduction to the topic area, so I've decided to make this chapter a welcome instead. Welcome to my thesis, dear reader.

In an introduction, I'm supposed to explain that I came up with a nice idea for a thesis and then pursued it in a logical fashion, without taking any side trips or stumbling down any dead-end alleys. Since this is not an introduction, I'm going to tell the real story, which should help explain how these somewhat disparate chapters are tied together.

I began my thesis work with the aim of exploring *statistical machine listening*: I planned to approach long-standing problems in music processing from a statistical machine learning perspective. The great advantage of probabilistic approaches is that they typically allow us to produce not only a single answer, but a whole distribution of potential answers with different weights (probabilities). One of the main difficulties in music processing problems is that there are often several reasonable answers; commiting to one and discarding the rest is generally not a good idea. Also, researchers typically break the problem into subproblems (beat detection, instrument identification, and so on) that are then tackled independently. Since the raw input data (sampled music) tends to be large, it is typically processed in different ways for each subproblem; different subsets of the total information are discarded. The problem with this approach is that ambiguities in one subproblem could potentially be resolved with information from other subproblems, but if the problems are solved independently and there is no representation of uncertainty, then doing this is impossible. I hoped that by using a probabilistic approach, I would be able to represent the uncertainty in each

subproblem. Then, by combining subproblems in a principled way, I could leverage different types of information to produce better results.

As often happens, this turned out to be a little bit more difficult than I thought it would. I started with the beat tracking subproblem; my solution is presented in Chapter 4. It took me a rather long time to arrive at my model for beat tracking, which is slightly embarrassing since it's nearly the simplest model one can imagine. Nevertheless, it works remarkably well on a wide variety of examples. I take comfort in the fact that it's often the case that good ideas are only obvious in hindsight.

- *Chapter 4: Beat Tracking*

I found that while my beat tracking solution often gave nice results, it was also quite computationally expensive. This led to an exploration of fast methods for inference in graphical models. This material is presented in Chapter 3. Chapter 3 is a bit of a monster. It contains a survey of existing fast methods, some new fast methods, some additional insights into existing fast methods, and an empirical evaluation. The empirical evaluation presents findings that challenge commonly-held beliefs about these methods.

- *Chapter 3: Fast Methods*

This thesis is not exactly two papers stapled together, but the beat tracking and fast methods chapters are mostly independent of each other. Chapter 2 presents an introduction to graphical models and also introduces some notation. It contains no new results. Readers familiar with the topic should probably just give it a brief skim, while those new to area might find it a useful introduction.

- *Chapter 2: Graphical Models*

When new phrases or ideas are introduced there is usually a note in the margin. There is also an index, which may help you find things that I think are important.

- *margin notes are great*

Enough philosophizing, on to the thesis!

Chapter 2

Graphical Models

2.1 Introduction to Graphical Models

We present a brief introduction to graphical models and several algorithms for inference in them. Good references for this material are Jordan [26] and Yedidia, Freeman and Weiss [41].

Probabilistic graphical models provide a framework for expressing statistical independence relationships between random variables. Each variable is represented by a node in a graph, and potential dependence relationships are represented by edges. Variables can be multidimensional, and can be discrete or continuous. Observed variables are represented by shaded nodes, and unknown or *hidden* variables are represented by unshaded nodes.

We focus on undirected graphical models. Directed models and factor graphs can easily be converted to undirected models, as shown in [41]. Several example undirected graphical models are shown in Figure 2.1.

In undirected graphical models, the relationships between variables are expressed in terms of *potential functions*. Consider nodes i and j, which represent the variables X_i and X_j. The potential function connecting these nodes is $\psi_{i,j}(X_i, X_j)$, and expresses the compatibility of the value X_i at node i with the value X_j at node j. See Figure 2.2. Large values of the potential function indicate that the values X_i and X_j are likely to co-occur, while small values indicate a statistically unlikely arrangement. Potential functions must be non-negative; unlike probabilities, they need not be normalised. According to the Hammersley-Clifford theorem [23], the joint probability of a graphical

- *potential functions*

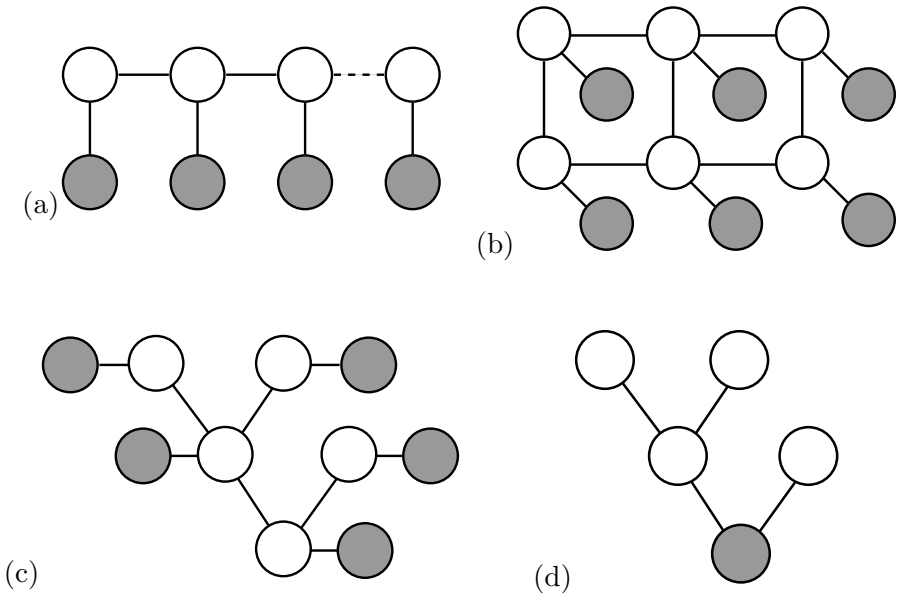


Figure 2.1: Examples of undirected graphical models. Shaded nodes represent observed variables, unshaded nodes represent hidden variables, and edges represent statistical dependence relationships between variables. (a) is a Markov chain. (b) is a Markov Random Field. (c) and (d) are arbitrary graphs.

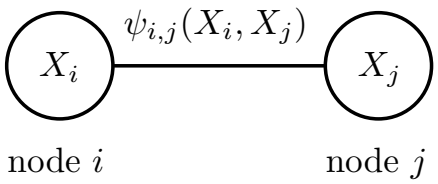


Figure 2.2: A potential function $\psi_{i,j}(X_i, X_j)$ between node i and node j describes the compatibility between the variable at node i having value X_i and the variable at node j having value X_j.

model is

$$p(X) = \frac{1}{Z} \prod_{i \in V} \psi_i(X_i) \prod_{(i,j) \in E} \psi_{i,j}(X_i, X_j)$$

where V are the vertices in the graph and E are the edges. The functions $\psi_i(X_i)$ describes how likely it is for the variable at node i to have the value X_i, and $\psi_{i,j}(X_i, X_j)$ are the potential functions connecting all pairs of variables that share edges. The partition function Z is a normalisation constant.

2.2 Inference Methods

There are two primary inference tasks in graphical models. Consider the set of variables X as being composed of the set of hidden variables X_H and the set of observed variables X_O. Given a set of observations X_O, the first inference task is to find the marginal probabilities of each hidden node X_H:

- *marginal probabilities*

$$p(X_i|X_O) \quad \text{for each} \quad i \in H \quad . \tag{2.1}$$

The second inference task is to find a maximum *a posteriori* (MAP) set of values for the hidden nodes X_H:

- *MAP set*

$$X_H^\star = \underset{X_H}{\operatorname{argmax}}\, p(X_H|X_O) \quad . \tag{2.2}$$

See Figure 2.3 for an example.

2.2.1 Belief Propagation

Belief Propagation [33] is an algorithm for computing the marginal probabilities in graphical models. The algorithm is exact for graphs that are trees. In some cases the same algorithm can also produce good results when used in cyclic ("loopy") graphs [32].

Belief Propagation (BP) proceeds by passing *messages* between nodes. A message passed to node j is simply a set of real values, one value per possible state at node j. The intuition is that the message passed from node i to node j tells node j which states are most preferred by node i; large values indicate favourable states.

- *message*

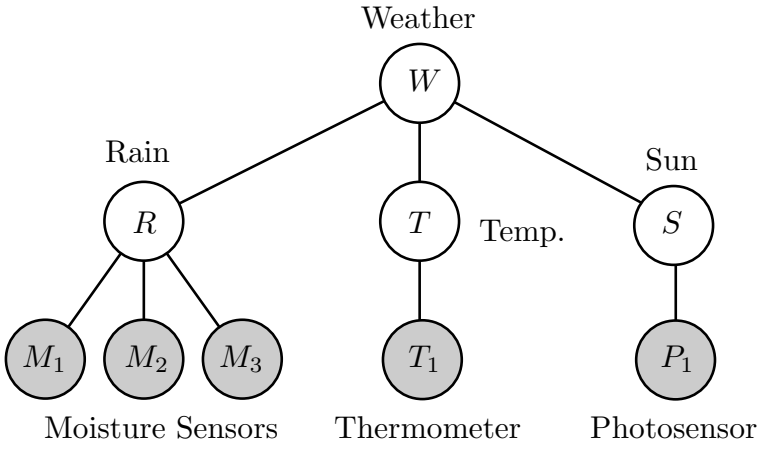


Figure 2.3: In the simple graph shown above, several noisy sensors are used to measure environmental conditions, and there are several hidden variables about which we would like to reason. Three binary moisture sensors M_i are connected to the binary hidden variable for rain, R. The potential functions (edges) connecting M_i and R model the noise properties of the sensors. For example, we expect $R = true$ when $M_i = true$, except that the sensors might have a false positive rate of 10% and a false negative rate of 30%. In a similar manner, the continuous temperature T and binary sun S variables are connected to their associated noisy sensors through potential functions that describe the noise processes of the sensors. Finally, these three hidden variables are each connected to the categorical weather variable W. The potential function between R and W should take a high value for $\{R = true, W = rainy\}$, and a low value for $\{R = true, W = sunny\}$.

In the terminology introduced above, the set of observed nodes is $X_O = \{M_1, M_2, M_3, T_1, P_1\}$ and the set of hidden nodes is $X_H = \{R, S, T, W\}$. By computing the marginal probabilities (equation 2.1), we could report the probabilities of each state at each node: $W = sunny$, $W = rainy$, $R = true$, $R = false$, $S = true$, $S = false$, and a probability distibution over the continuous variable T. By computing a MAP set (equation 2.2), we could report the most probable combination, such as $X_H^\star = \{W = sunny, R = false, S = true, T = 25.4°\text{C}\}$.

Formally, if the states at node i are x_i, then the probabilities (or *beliefs*) at node i are[1] • *belief*

$$b_i(x_i) = \frac{1}{Z}\ \psi_i(x_i) \prod_{j \in N(i)} m_{j,i}(x_i) \quad , \tag{2.3}$$

where $N(i)$ are the neighbours of node i (all nodes sharing edges with node i) and $m_{j,i}$ is the message passed from node j to node i. Z is a normalizing constant. Intuitively, $\psi_i(x_i)$ describes the local probability of each state, while the messages capture the compatibility of each state with the rest of the graph.

The message passed from node j to node i is

$$m_{j,i}(x_i) = \sum_{x_j} \psi_j(x_j)\ \psi_{j,i}(x_j, x_i) \prod_{k \in N(j) \text{ except } i} m_{k,j}(x_j) \quad , \tag{2.4}$$

which can be interpreted as the likelihood of arriving at state x_i, marginalized over all states x_j.

Another way of understanding the message is to substitute the expression for the belief $b_j(x_j)$ into the above expression:

$$m_{j,i}(x_i) \ \propto \ \sum_{x_j} \frac{b_j(x_j)\ \psi_{j,i}(x_j, x_i)}{m_{i,j}(x_j)} \quad .$$

Now, $b_j(x_j)$ can be seen as the probability of starting at state x_j, and $\psi_{j,i}(x_j, x_i)$ the probability of making the transition to state x_i given that we start in state x_j. We divide out the influence of the message passed in the opposite direction, $m_{i,j}(x_j)$. (An intuitive explanation for this is that the belief incorporates information from the whole graph, but during message-passing we want to move information in only one direction at a time, so we ignore edges that allow information to flow in the opposite direction.) The summand, therefore, gives the probability of starting at state x_j (at node j) and arriving at state x_i (at node i). By summing over all x_j, we enumerate all possible ways of arriving at x_i.

The messages must be passed in a specific order; this is called the message-passing *protocol*. In tree models, one valid protocol is that one round of messages is passed from the leaves to the root and then a second round is passed in the opposite direction. See

[1] This is the notation used in [41].

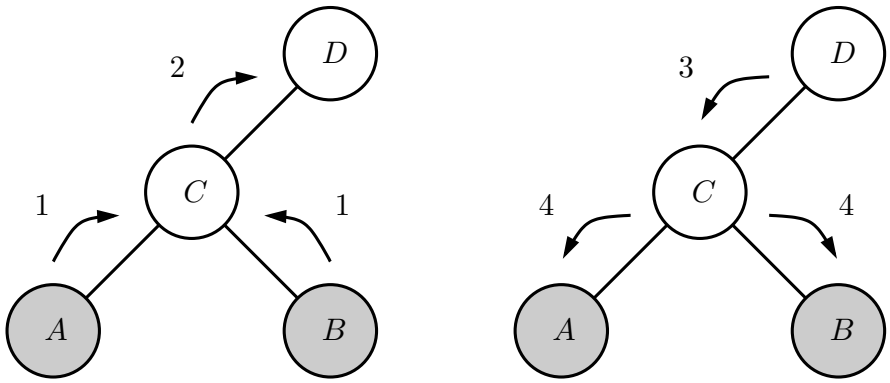


Figure 2.4: Message passing protocol in Belief Propagation. In the first round (left), messages are passed from leaves to the root. The messages labelled [1] are computed first (possibly in parallel), then message [2] is computed. In the second round (right), messages are passed back from the root to the leaves. Message [3] is computed first, and finally messages [4] are computed.

Figure 2.4. In the first round of messages, the message passed by node i summarizes the information from all nodes in the subtree rooted at i. Messages arriving at the root contain a summary of information from the whole tree. In the second round, the message passed to node i summarizes the information from all nodes between the root and i (including the information gathered in the first round). Messages arriving at the leaves thus incorporate information gathered from the whole tree.

2.2.2 Discrete Maximum-Belief Propagation

The Belief Propagation (BP) algorithm solves exactly the inference task of computing the marginal probabilities for graphical models that are trees. For the maximum *a posteriori* (MAP) task, a similar but distinct algorithm, Maximum-Belief Propagation (Max-BP) must be used. Max-BP is a type of dynamic programming, and for Hidden Markov Models is equivalent to the Viterbi algorithm.

As in standard BP, the Max-BP algorithm proceeds by passing messages between nodes. In the first round of message passing, we compute the costs of partial paths

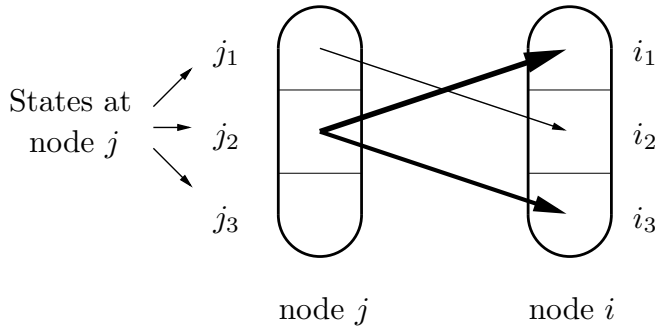


Figure 2.5: A Max-BP message. Node j, with states j_1, j_2, and j_3, is the bubble on the left; node i is on the right. The message $m_{j,i}(x_i)$ passed from node j to node i is indicated by the three arrows between the nodes; each arrow has associated with it a value, represented here by the width of the arrow. Note that one arrow arrives at each state x_i. The 'source' of each arrow is the corresponding value of $x^*_{j,i}(x_i)$ (see equation 2.6).

through the graph. In the second round, we trace backward through the graph, collecting states that are members of the cheapest total path.

The messages are the same as in BP (equation 2.4) except that the summation is replaced by a maximization. The message passed from node j to node i is

$$m_{j,i}(x_i) \;=\; \max_{x_j} \left\{ \psi_j(x_j)\;\; \psi_{j,i}(x_j, x_i) \prod_{k \in N(j) \text{ except } i} m_{k,j}(x_j) \right\} \quad . \tag{2.5}$$

See Figure 2.5 for an illustration.

The MAP problem can be interpreted as the problem of finding an optimal (highest-value) path through the graph. In this light, the message $m_{j,i}(x_i)$ is the value of the optimal way of arriving at each state x_i from node j. We must also record the state x_j that produces the best path:

$$x^*_{j,i}(x_i) \;=\; \operatorname*{argmax}_{x_j} \left\{ \psi_j(x_j)\; \psi_{j,i}(x_j, x_i) \prod_{k \in N(j) \text{ except } i} m_{k,j}(x_j) \right\} \quad . \tag{2.6}$$

After the first round of messages have been passed (from leaves to the root of the graph), the belief at the root node is computed (equation 2.3). The state with the

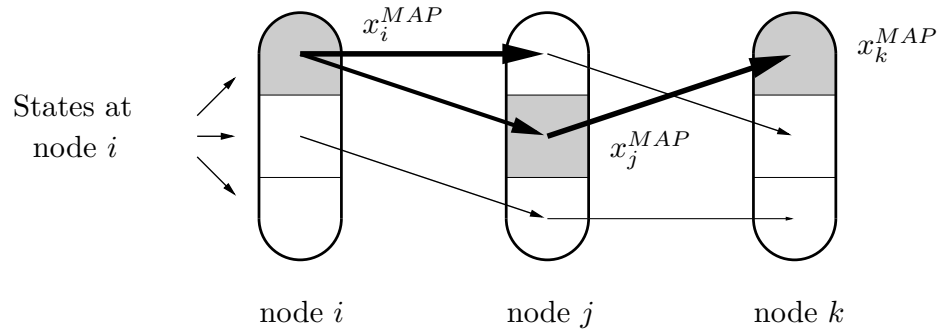


Figure 2.6: A Max-BP example. In this three-node graph, node i is a leaf node and node k is the root. (For this graph, we could arbitrarily make any of the nodes the root.) The messages from nodes i to j and j to k are shown by the arrows, where the width of the arrow shows the value of the message element. At state k, the top state has the highest belief, so is a member of the MAP set (which we have shown by shading it). We then trace backward through the graph, collecting the states $\{x^{MAP}\}$ that form the best path through the graph.

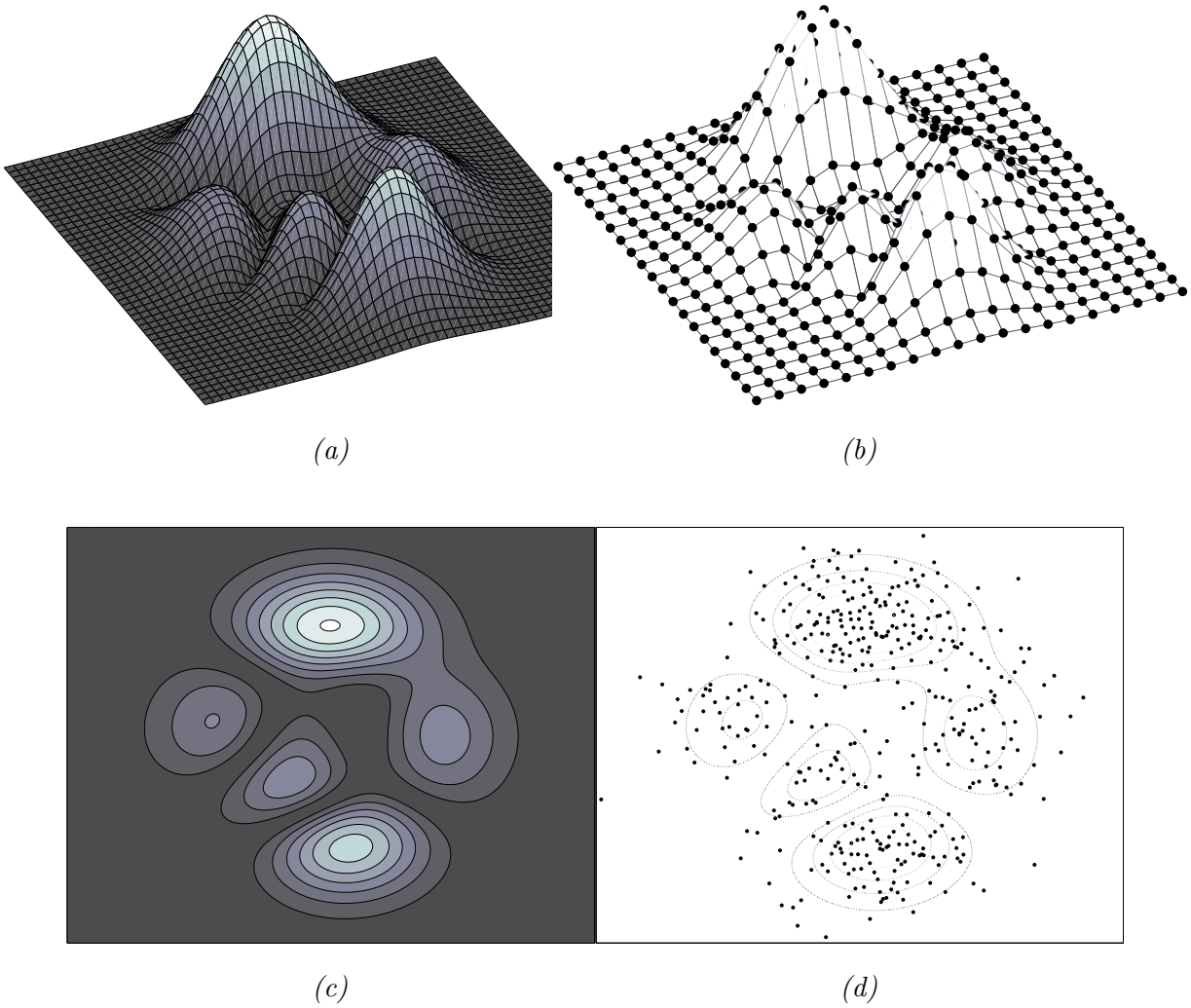
largest belief[2] is a member of the MAP set. We then trace backward through the graph (from the root to leaves), collecting, for each node, the state that led to that state; these form the MAP set. See Figure 2.6.

2.2.3 Continuous (Max-)BP

The BP and Max-BP algorithms described above solve the tasks of computing the marginals and MAP sets, respectively, for tree models with discrete state spaces. For tree models with continuous state spaces (or large discrete spaces), we can use (Max-)BP if we first use one of several 'gridding' methods to create a discrete approximation of the continuous space.

For bounded continuous state spaces, a simple gridding method is to place a uniform mesh over the state space. Each mesh point then becomes a discrete state. See

[2]In general there can be two or more states with beliefs equal to the maximum. It is a simple extension to collect all such sets of MAP sets. Here, we consider only the simple case where there is a single maximum.



(a) *(b)*

(c) *(d)*

Figure 2.7: Gridding methods. *(a)*: an example continuous state space. *(b)*: the uniform mesh method, with 20×20 grid. Each point becomes a discrete state. (Connecting lines have been added for ease of viewing.) *(c)*: a contour view of the same continuous state space. *(d)*: the Monte Carlo gridding method with 400 particles (the contour lines are overlaid for reference). Each particle becomes a discrete state. Note that areas of high probability have high particle density; more attention is paid to important regions of the state space.

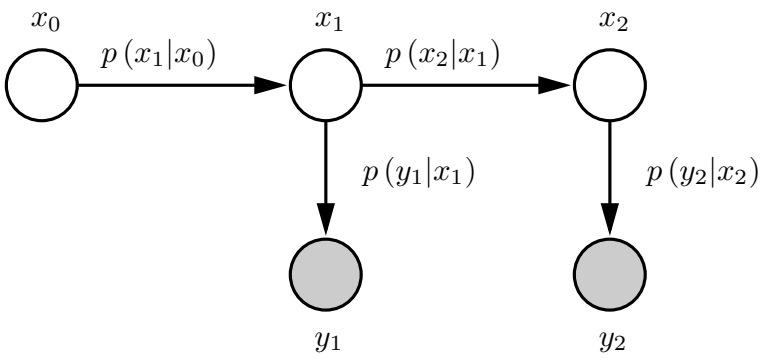


Figure 2.8: A Hidden Markov Model.

Figure 2.7. This method scales poorly to high-dimensional state spaces, since the number of mesh points grows exponentially with dimension.

Another gridding approach is to run a particle filter to obtain a Monte Carlo approximation of the state space. Such an "adaptive grid" method is presented in [16] for the Max-BP problem. This works particularly well for large or high-dimensional state spaces, particularly those with small regions of high density.

2.2.4 Particle Smoothing

The Belief Propagation algorithm presented above allows us to calculate the marginal probabilities of each state, given all the observations, for tree models with discrete state spaces. For continuous state spaces or very large discrete spaces, we must typically resort to approximate methods. For chain models, Monte Carlo particle methods are a standard approach. Good references are [8, 9].

The standard particle filter is typically applied to chain models (Hidden Markov Models, HMM), which are characterized by a set of observations $\{y_k\}$ and hidden variables $\{x_k\}$, for $k = 1 \ldots n$, plus an initial state probability distribution $\pi(x_0)$, transition model $p(x_k|x_{k-1})$, and observation model $p(y_k|x_k)$. See Figure 2.8.

For the remainder of this section we use the notation of [9], which has become standard in the literature. As explained in [26], an HMM can be converted into an

undirected model by setting

$$\psi(x_i) = p(x_i) \tag{2.7}$$

$$\psi(x_i, x_j) = p(x_j|x_i) \quad , \tag{2.8}$$

where x_j in the above can be either x_{i+1} (for transitions) or y_i (for observations).

Often, each pair of nodes in an HMM corresponds to an instant in time, and the task is to estimate the trajectory of the hidden variables given the observations. For each node, the particle filter generates an estimate of the *filtering distribution*: the distribution of the hidden variable given all the observations up to that point in time. This is accomplished by generating a number of weighted samples in the first node, and propagating the set of samples to subsequent nodes as time progresses. The samples are called *particles*; the particles at node k are $\left\{\tilde{x}_k^{(i)}\right\}$, $i = 1 \ldots N$, and have filtering weights $\tilde{w}_k^{(i)}$. The filtering distribution at node k is estimated by

- *filtering distribution*

$$\hat{p}(x_k \mid y_{0:k}) = \sum_{i=1}^{N} \tilde{w}_k^{(i)} \; \delta\left(\tilde{x}_k^{(i)} - x_k\right)$$

where δ is the Dirac delta function; the continuous distribution is approximated by a weighted set of discrete samples.

The particle filter alone does not solve the task of computing the marginal probabilities, however. Since we only move information forward in time, the filtering distribution at a given nodes takes into account only observations up to that node. To produce the marginal probabilities, we must move information in the opposite direction as well. For a chain of length n, the marginal probabilities are $p(x_k|y_{1:n})$, while the filtering distribution is $p(x_k|y_{1:k})$. The filtering distribution at state k takes into account only observations $y_{1:k}$, that is, observations up to time k; while the marginals take into account all observations $y_{1:n}$.

An estimate of the marginal probabilities can be computed by making a backward pass in which the samples generated during the forward (particle filter) pass are reweighted. This is called the forward filtering-backward smoothing method. We only state the result here; see [8] for details. The new particle weights are called the *smoothing weights*, $\tilde{w}_{k|n}^{(i)}$, and are computed as follows:

- *smoothing weight*

$$\tilde{w}_{n|n}^{(i)} = \tilde{w}_n^{(i)} \tag{2.9}$$

$$\tilde{w}_{k|n}^{(i)} = \tilde{w}_k^{(i)} \sum_{j=1}^{N} \frac{\tilde{w}_{k+1|n}^{(j)}\ p\left(\tilde{x}_{k+1}^{(j)} | \tilde{x}_k^{(i)}\right)}{\sum_{m=1}^{N} \tilde{w}_k^{(m)}\ p\left(\tilde{x}_{k+1}^{(j)} | \tilde{x}_k^{(m)}\right)} \tag{2.10}$$

The marginal probabilities, or *smoothing distributions*, $p(x_k|y_{1:n})$, are approximated by

- *smoothing distribution*

$$\hat{p}\left(x_k \mid y_{0:n}\right) = \sum_{i=1}^{N} \tilde{w}_{k|n}^{(i)}\ \delta\left(\tilde{x}_k^{(i)} - x_k\right) \quad .$$

Note that this distribution has the same particles as the filtering distribution, but different weights that incorporate information from all the observations.

2.3 Summary

In this chapter, we have reviewed the graphical model framework, stated the primary inference tasks with which we are concerned, and presented several algorithms for solving these tasks. Belief Propagation (BP) can be used to solve the task of computing the marginal probabilities of the hidden states given all the observations. Maximum-Belief Propagation (Max-BP), similarly, allows us to compute a maximum *a posteriori* set of states for the hidden nodes. Both these algorithms are directly applicable to models with discrete state spaces, and give exact results for graphical models that are trees. For continuous state spaces or very large discrete spaces, we can apply a discretization strategy (such as deterministic or Monte Carlo gridding) and then apply (Max-)BP. Finally, for Hidden Markov Models with non-Gaussian continuous state spaces, Particle Smoothing can be used to compute the marginal probabilities.

Chapter 3

Fast Methods

3.1 Introduction

In the previous chapter, we reviewed several methods for inference in graphical models. These methods have strong theoretical bases, but in practice the resulting computation can be very expensive. When computed naively, the Particle Smoothing, Belief Propagation, and Max-Belief Propagation algorithms each cost $O\left(kN^2\right)$, where k is the number of nodes in the graphical model and N is the number of particles or discrete states at each node. This cost becomes prohibitive for many practical problems in which large numbers of states are required.

The $O\left(N^2\right)$ complexity of these algorithms arises when computing the smoothing weights in Particle Smoothing, and in computing the messages in Belief Propagation and Max-Belief Propagation (equations 2.10, 2.4, and 2.5, respectively). For Particle Smoothing and Belief Propagation, this cost arises because every particle in one state has some influence upon every particle in neighbouring nodes. This can be expressed as a Sum-Product problem (defined below). In Max-Belief Propagation we must find, for each particle in one state, the particle from a neighbouring state which exerts the strongest influence. This can be expressed as a Max-Product problem (also defined below).

In this chapter, we present fast methods for computing the Sum-Product and Max-Product problems. We show how these can be used to speed up Particle Smoothing, Belief Propagation and Max-Belief Propagation in Section 3.5. The Sum-Product and Max-Product problems arise in many other settings; these are discussed briefly in Section 3.6.

For both the Sum- and Max-Product problems, we are given a set of points $X = \{x_i\},\ i = 1 \ldots N$, each of which has a non-negative *weight*, *mass*, or *strength* w_i. We call these *source points*, *source particles*, or simply X *particles*. These names come from the analogy to physical particles with electric charge or mass. We are also given a set of points $Y = \{y_j\},\ j = 1 \ldots M$, which we call the *target points*, Y *points*, or *target particles*; these are the points at which we wish to measure the quantity of interest. In some cases, the X and Y points will coincide. It will often be the case that M and N are equal. We assume that the points live in a multidimensional vector space (with dimension D), though it is often the case that the methods and problems generalize to metric spaces in a straightforward manner.

- *sources*
- *targets*

The Sum-Product problem is to compute the values

$$f_j = \sum_{i=1}^{N} w_i \; K_i \left(\|x_i - y_j\| \right) \quad , \quad j = 1 \ldots M \quad . \tag{3.1}$$

- *Sum-Product problem*

We call f_j the *influence* at point y_j. K is called the *kernel function*. In many cases, $K_i(d) = K(d)$, meaning that the kernel is the same for each source. This problem is also commonly called weighted Kernel Density Estimation.

- *influence*
- *kernel function*

The Max-Product problem is simply the Sum-Product problem with the summation converted to a maximization:

- *Max-Product problem*

$$\begin{aligned} f_j^* &= \max_{i \in [1,N]} \left\{ w_i \; K_i \left(\|x_i - y_j\| \right) \right\} \\ x_j^* &= \operatorname*{argmax}_{i \in [1,N]} \left\{ w_i \; K_i \left(\|x_i - y_j\| \right) \right\} \quad , \quad j = 1 \ldots M \quad . \end{aligned}$$

This can be seen as a search for the particle in X that has the greatest influence upon each point in Y.

Here, we are interested in methods that will allow us to speed up the computation as N and M become large. We also want the methods to scale well as the dimension D becomes large. We consider the methods shown in Figure 3.1.

We mention the Fast Multipole Method (FMM) briefly; there is an extensive literature on the topic, including good tutorials. The FMM is based on a combination of space subdivision and series expansion, hence the series expansion must be rederived

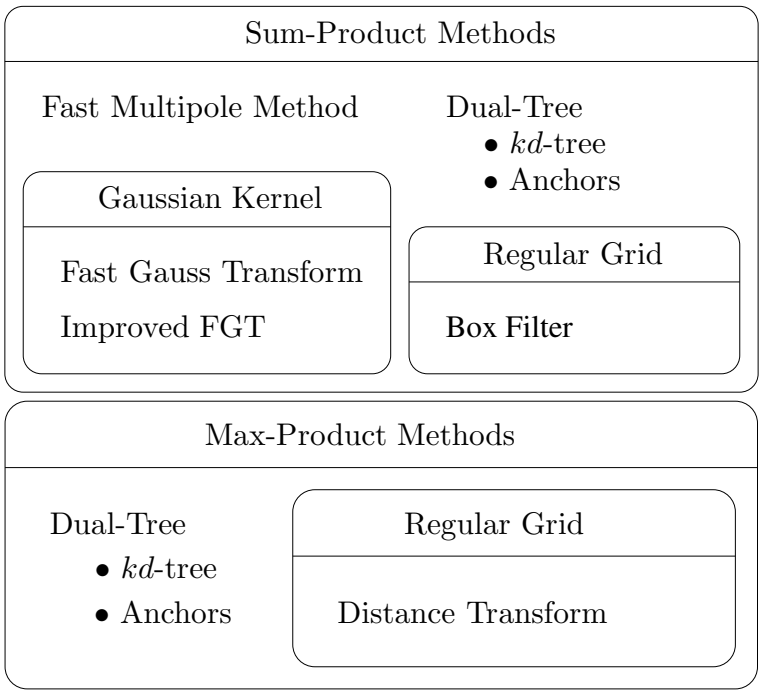


Figure 3.1: The fast methods we consider. See the text for discussion.

for each particular kernel function. The Dual-Tree method relies upon space partitioning and the ability to compute bounds on the kernel function, so can easily handle many different types of kernels. The Fast Gauss Transform (FGT) is a specialization of the FMM for Gaussian kernels. The Improved Fast Gauss Transform (IFGT) uses a different series expansion and a flat (non-hierarchical) subdivision of space. The box-filter method applies only to the special case where the source and target points are located on the vertices of a regular grid.

For the Max-Product problem, we again consider a Dual-Tree method, which can be applied to a very broad class of kernel functions and arbitrary distributions of points. For the particular case where the sources and targets are the vertices of a regular grid, and for a limited family of kernel functions, the Distance Transform can be applied.

Our specific contributions are:

- the Dual-Tree Max-Product algorithm;
- improved kernel bounds for the Dual-Tree Sum-Product algorithm;
- some implementation-level enhancements to the Dual-Tree Sum-Product algorithm;
- a correction to the error bound of the Improved Fast Gauss Transform;
- a protocol for choosing the parameters of the Improved Fast Gauss Transform;
- the application of the Sum-Product to Particle Smoothing, (Max-)Belief Propagation, and Gaussian Processes.

Of course, we also hope that our survey of existing fast methods will itself be helpful.

3.2 Related Work

Several researchers have presented methods for speeding up Belief Propagation (BP) and Maximum-Belief Propagation (Max-BP) in various settings. These can be split into two categories. The first are *structural* methods, which aim to reduce the total number of messages passed during Loopy Belief Propagation (where multiple rounds of message passing are required). The second type are methods that aim to reduce

the cost of computing a single message. We discuss only the former in this section; the latter are instances of Sum-Product and Max-Product methods and are discussed later. The fast Sum- and Max-Product methods we present can be used in conjunction with these structural methods.

Felzenszwalb and Huttenlocher [11] present two structural methods for speeding up Max-BP in Markov Random Fields (MRFs), particularly for computer vision applications. Since MRFs are cyclic, Max-BP is not guaranteed to produce the correct answer, but Loopy Max-BP can give good approximations in some problem domains [30, 13].

The first method uses a hierarchical (coarse-to-fine) strategy; a new MRF in created in which 2×2 blocks of nodes in the original MRF are treated as a single node. This process is repeated to create a pyramid. The smallest MRF at the top of the pyramid can be solved quickly by running several iterations of Max-BP. The results are then propagated downward to initialize the next-finer level in the pyramid. This process is repeated until the bottom of the pyramid (the original MRF) is solved. This strategy seems to work because it allows long-range movement of information through the graph in a small number of iterations. For many computer vision problems, strong image smoothness constraints allow blocks of nodes to be grouped together in such a manner. Such multiresolution strategies are common in many fields [38].

The second method aims to reduce the number of iterations of Max-BP required, by partitioning the graph into two halves and computing messages for only one half of the graph each iteration. Such a strategy is given a theoretical grounding in [24].

3.3 Sum-Product

In this section, we present five approximate methods for the Sum-Product problem. Recall that the Sum-Product problem is to compute

$$f_j = \sum_{i=1}^{N} w_i \; K_i \left(\|x_i - y_j\| \right) \quad , \quad j = 1 \ldots M \quad .$$

Given an absolute error tolerance ϵ, each of these methods compute approximate values of $\tilde{f}_j$ that satisfy $|\tilde{f}_j - f_j| \leq \epsilon$ for each j.

3.3.1 Fast Multipole Method

There is an extensive literature on the Fast Multipole Method (FMM). Good tutorials exist, so rather than reinvent the wheel we refer interested readers to the references of Beatson and Greengard [3] and Ihler [25]. In this section we will briefly discuss the important features of the FMM.

The FMM encompasses a large family of methods for solving Sum-Product problems. Most FMM formulations have been developed for applications in physics, where N-body problems commonly arise. In N-body problems, a set of objects are distributed in space; each object has a weighted distance-dependent influence upon each other object, and the influences are additive. For example, computing the gravitational or electrostatic interaction of N masses or charged particles is a typical N-body problem to which the FMM can be applied.

The FMM is typically applied to kernels which are not well-behaved near the origin, but become smooth at a sufficient distance. For example, the gravitational and electrostatic potentials[1] are of the form

$$K(x-y) = \frac{1}{\|x-y\|}$$

which is singular at $y = x$ but smooth elsewhere. The idea is then to develop series expansions of the kernel that are valid for some domain $\|x-y\| \geq R$. This is the *far-field expansion*. If the far-field expansion can be written as

$$K(x,y) \simeq \sum_{k=1}^{p} \phi_k(x)\,\psi_k(y) \quad , \quad \|x-y\| \in R \quad ,$$

then the problem can be split into two phases. In the first phase, the coefficients of

[1] The *potential* describes the energy stored in the field, which is the integral of force over distance. The force of the gravitational and potential fields go as $F \propto \frac{1}{\|x-y\|^2}$, so the potentials go as $E \propto \frac{1}{\|x-y\|}$.

the series expansion are computed:

$$C_k = \sum_{i=1}^{N} w_i \; \psi_k \left(x_i \right) \quad , \quad k = 1 \ldots p \quad ;$$

these depend only on the sources. In the second phase, the series expansion is evaluated at each target point:

$$f_j = \sum_{k=1}^{p} C_k \; \phi_k \left(y_j \right) \quad , \quad j = 1 \ldots N \quad ;$$

these depend only on the targets. In this way, the sources and targets interact only through the series expansion, and the cost becomes $O\left(Np\right)$, where p is the number of terms in the series expansion.

Since in general the far-field expansion is only valid on some domain, it is necessary to subdivide the space in which the particles live (typically using a hierarchical data structure such as the *kd*-tree), and construct different series expansions that are valid in different regions of the space. Evaluating the influence upon a single y particle is then a matter of recursing down the tree, evaluating at each level the series expansions that belong to branches of the tree that are sufficiently far away. This process stops at the leaf nodes, which contain some small number of particles whose influence can be evaluated directly.

Rather than evaluating the expansion separately for each y point, the FMM also creates a space-partitioning tree for the target points, and rather than evaluating the series expansion directly, constructs Taylor expansions that are valid for all target points within that branch of the target tree. The result of this process is that for each y point it is only necessary to evaluate a Taylor expansion.

This has been a very terse summary of a rich subject area. The key points are that the FMM is based on a hierarchical subdivision of space and series expansions that are valid on limited domains. The error bounds of the FMM are determined by the truncation error of the series expansions, so are determined *a priori*. The exact form of the series expansions depends on the form of the kernel; expansions for many kernels have been developed.

3.3.2 Fast Gauss Transform

The Fast Gauss Transform (FGT), presented by Greengard and Strain [21, 22, 2], is a specialization of the Fast Multipole Method. It solves the Sum-Product problem for the case when the kernel function is a Gaussian with scale parameter h:

$$K(d) = \exp\left\{-\frac{d^2}{h}\right\} \quad .$$

There is also a version for the case of variable scales:

$$K_i(d) = \exp\left\{-\frac{d^2}{h_i}\right\} \quad ,$$

[36], though we consider only the simple case here.

The FGT proceeds by dividing the space into a uniform grid of boxes with side length of order $\sqrt{h}$. For each source particle, a Hermite expansion of the Gaussian is performed. The expansion coefficients for the particles in each box can be summed, yielding a single expansion for each box. In a similar manner, each target box has a set of influential source boxes, and the Hermite expansion inside the box can be expressed as a Taylor expansion. Thus, for each target point in the box, we need only do a Taylor expansion. In this way, the source and target particles are decoupled through the series expansions, and the $O(MN)$ complexity becomes $O(M+N)$.

Formally, given a set of N_B particles in a single box B with center x_B, the Sum-Product can be approximated by a Hermite expansion about the center:

$$\begin{aligned} f_j(B) &= \sum_{i=1}^{N_B} w_i \ \exp\left\{-\frac{\|x_i - y_j\|^2}{h}\right\} \\ \tilde{f}_j(B) &= \sum_{\alpha \le p} A_\alpha(B)\ h_\alpha\left(\frac{y_j - x_B}{\sqrt{h}}\right) \end{aligned}$$

where α is a multi-index, h_α is the Hermite function of order α, and p is the order of the expansion. The coefficients $A_\alpha(B)$ are given by

$$A_\alpha(B) = \frac{1}{\alpha!}\sum_{j=1}^{N_B} w_j \left(\frac{x_j - x_B}{\sqrt{h}}\right)^\alpha \quad .$$

For each target box T, we compute the set of boxes B that are within range, then collect the Hermite expansions from these boxes and expand them in a Taylor expansion about

the box center y_T:

$$\begin{aligned} \tilde{f}_j &= \sum_B \tilde{f}_j(B) \\ &= \sum_{\beta \le p} C_\beta \left(\frac{y_j - y_T}{\sqrt{h}} \right)^\beta \end{aligned}$$

where β is a multi-index and the coefficients C_β are

$$C_\beta = \frac{(-1)^\beta}{\beta!} \sum_B \sum_{\alpha \le p} A_\alpha(B)\ h_{\alpha+\beta} \left(\frac{x_B - y_T}{\sqrt{h}} \right) \quad .$$

The set of boxes B that are within range of a target box T is determined by the error bound ϵ. We take the n closest boxes in each dimension (i.e., all boxes in a cube with sides of length $2n+1$ boxes centered at T) and ignore the influence of all other boxes. In the worst case we ignore a particle at distance nr (where r is the side length of the boxes) with weight w. Therefore, we choose n such that the influence of this worst-case particle is less than ϵ:

$$f \;=\; w \exp\left\{ -\frac{(nr)^2}{h} \right\} \;\le\; \epsilon$$

and w cannot be greater than the sum of weights W:

$$w \;\le\; W \;=\; \sum_{i=1}^{N} w_i \quad ,$$

hence the number of boxes within range of each target box does not depend on the number of source or target particles, but only on ϵ.

The number of terms p in the Hermite and Taylor expansions is also determined by the allowable error ϵ. The error bound in [2] is

$$|\tilde{f}_j - f_j| \;\le\; \frac{W}{(1-r)^d} \sum_{k=0}^{D-1} \binom{D}{k} (1 - r^p)^k \left(\frac{r^p}{\sqrt{p!}} \right)^{D-k}$$

where D is the dimension of the problem.

The amount of work required to perform the FGT is divided into two phases: computing the expansion coefficients for all the boxes takes $O(N)$; and evaluating the expansions for all the target points takes $O(M)$. The number of terms required in the series expansions is a function of the error bound ϵ.

Although the FGT yields linear performance with respect to the number of source and target points, there are several theoretical and practical shortcomings with respect to other problem parameters. First, the total number of Hermite expansions terms grows as p^D, where D is the dimension of the problem, and p is the number of expansion terms per dimension. In addition, the uniform division of space causes the number of boxes to become large as the scale parameter h becomes small or when dimension grows. The number of boxes grows as $\left(\frac{1}{h}\right)^{2D}$, so implementations of the algorithm that do not deal cleverly with empty boxes will become prohibitively expensive, both in the amount of memory required to store many empty boxes, and in computation time in examining empty boxes.

In practice, these shortcomings limit the FGT to problems up to dimension $D = 3$. Unless adaptive gridding methods are used, the memory requirements of the FGT can be become prohibitive if h is too small.

3.3.3 Improved Fast Gauss Transform

The Improved Fast Gauss Transform (IFGT) [39, 40] aims to improve upon the FGT by using a space-partitioning scheme and a series expansion that scale better with dimension.

The space-partitioning scheme used in the IFGT is adaptive, in that it is sensitive to the location of source points rather than being fixed. They simply run the farthest-point K-centers algorithm, which performs a rudimentary clustering of the data into K clusters. As in the FGT, the series expansion coefficients for all points in each cluster are summed.

The FGT treats the multidimensional Gaussian as a product of one-dimensional Gaussians, performing a series expansion for each dimension independently. The IFGT does a series expansion of the distance from sources to targets, which yields fewer terms in the series expansion, particularly for high-dimensional problems.

Formally, the algorithm proceeds as follows. First, the farthest-point clustering algorithm is run on the source points, producing K clusters. For each cluster B with

center x_B, the series expansion coefficients are computed:

$$C_\alpha^B \;=\; \frac{2^\alpha}{\alpha!}\sum_{i=1}^{N_B} w_i \;\exp\left\{-\frac{\|x_i - x_B\|^2}{h^2}\right\}\left(\frac{x_i - x_B}{h}\right)^\alpha, \quad |\alpha| < p \;\;.$$

Then, for each target cluster T, the set of clusters B that are within range are computed. Finally, the Taylor expansion for each target point is computed:

$$\tilde{f}_j = \sum_B \sum_{|\alpha|<p} C_\alpha^B \;\exp\left\{-\frac{\|y_j - x_B\|}{h^2}\right\}\left(\frac{y_j - x_B}{h}\right)^\alpha \quad .$$

The total number of terms in the multi-dimensional Taylor expansion of total order p (called $r_{p,d}$ in [39]) is

$$r_{p,D} = \binom{p+D}{D} = \frac{(p+D)!}{p!\;D!} \quad ,$$

where D is the dimension of the problem.

We must choose several parameters in order to satisfy a given error bound ϵ. There are two sources of error in the IFGT: truncation of the series expansion after order p, and ignoring source clusters that are out of range. The first error source is called E_T in equation 3.12 of [39]:

$$E_T \;\leq\; W \exp\left\{\frac{2\;r_x r_y - {r_x}^2 - {r_y}^2}{h^2}\right\}\frac{2^p}{p!}\left(\frac{r_x r_y}{h^2}\right)^p$$

where r_x is the maximum source cluster radius and r_y is the range: the largest distance between targets and source clusters considered to be 'within range'. p is the order of the series expansion. In [39] this bound is simplified (using the triangle inequality) to

$$E_T \;\leq\; W\frac{2^p}{p!}\left(\frac{r_x r_y}{h^2}\right)^p \quad ,$$

but we find that in typical cases this simplification introduces a factor of 10^3 or more to the error bound; we therefore use the first expression.

The second error source, as in the FGT, comes from ignoring the influence of clusters outside of range r_y. The error bound for this term in [39], equation 3.13, is wrong. See Figure 3.2 for an illustration. The equation given is

$$E_C \;\leq\; W \exp\left\{\frac{{r_y}^2}{h^2}\right\} \quad .$$

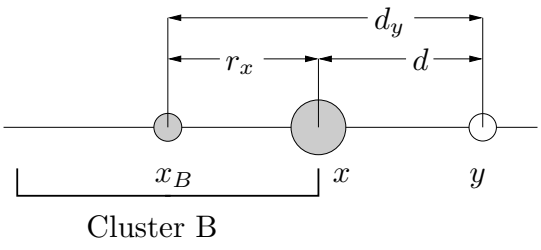


Figure 3.2: An illustration of the IFGT error bound due to ignoring clusters outside radius r_y. The cluster center is x_B and its radius is r_x; the cluster is ignored if the distance d_y from the center to y is greater than the range r_y. In the worst case, a particle x with large weight sits on the cluster boundary nearest to y. The distance between y and x can be as small as $d = r_y - r_x$.

(incorrect)

However, in the worst case a source particle x with weight $w = W$ and belonging to cluster B is located at the *nearest* possible point to target y. If the distance from y to the cluster center x_B is infinitesimally greater than r_y, then cluster B is ignored. If the particle x is located on the cluster boundary closest to y, then the distance between x and y is nearly $r_y - r_x$, *not* r_y. Hence, the true error bound is

$$E_C \;\leq\; W \exp\left\{ \frac{(r_y - r_x)^2}{h^2} \right\} \quad .$$

The IFGT papers do not suggest a method for choosing parameters K, r_y and p to meet a given error bound ϵ. We developed a protocol for choosing parameters that will satisfy a given error bound and also ensure that the complexity of the algorithm does not degrade. See Figure 3.3.3 for the protocol. It is based upon four constraints C:

$$\begin{aligned} C_1: &\quad E_C &\leq& \;\epsilon \\ C_2: &\quad E_T &\leq& \;\epsilon \\ C_3: &\quad K &\leq& \;K^* \\ C_4: &\quad \left(\frac{r_x r_y}{h^2}\right) &\leq& \;1 \end{aligned}$$

where C_1 and C_2 are hard constraints that guarantee the error bound, C_3 is a hard constraint that guarantees the algorithmic complexity, and C_4 is a soft constraint that

improves convergence. Note that each source cluster contributes to the error either through series expansion (E_T) or by being ignored (E_C), but not both. Therefore, it suffices to require $E_C \leq \epsilon$ and $E_T \leq \epsilon$, rather than $E_C + E_T \leq \epsilon$.

Note that the IFGT, unlike the FGT, does not cluster the target points y; the distance from each target to each source cluster is computed. The algorithm therefore has $O(KM)$ complexity, where K is the number of source clusters and M is the number of targets. To keep $O(M)$ complexity, K must be bounded above by a constant, K^*. This is constraint C_3. Note that r_x (the maximum cluster radius) decreases as K (the number of clusters) increases. Since K has an upper bound, r_x has a lower bound. Contrary to the claim in [39], r_x cannot be made as small as required by increasing K.

The main issue with the IFGT is that the error bound $E_T + E_C$ vastly overestimate the actual error. In [39], the error bound plots show that the bound typically overestimates the actual error by factors of 10^2 to more than 10^6. This is a major issue if we want guaranteed error bounds, because it means that we must choose p to be much larger than necessary.

Another problem is that in many cases we cannot choose parameters such that $\frac{r_x r_y}{h^2} \leq 1$, so the truncation error bound E_T goes to zero very slowly. As the number of terms is increased from $p-1$ to p, the error bound changes by a factor of $\frac{2r_x r_y}{ph^2}$. Note that this can be greater than one - the error bound can increase as more terms are added to the series expansion! Eventually, p becomes large enough that the error bound decreases, but by this time p can become very large. This occurs because the Taylor expansion error bound grows with the size of the domain of the expansion. The expansion is about zero (the cluster center), so the worst-case expansion error occurs at the farthest point from the cluster center. This is troubling, because we know that the Gaussian decays away from zero; the greatest distance from the cluster center has the largest expansion error bound, yet we know that the Gaussian has the smallest value here.

// Recall:
// $C_1 : E_C \leq \epsilon$
// $C_2 : E_T \leq \epsilon$
// $C_3 : K \leq K^*$
// $C_4 : \left(\frac{r_x r_y}{h^2}\right) \leq 1$
Input: r_y(ideal), ϵ
Output: k, r_y, p
Algorithm:
    for $k = 1$ to K^*:
        run k-centers algorithm.
        find largest cluster radius r_x.
        using $r_y = r_y$(ideal), compute C_1, C_4.
        if C_1 AND C_4:
            break

    if $k < K^*$:
        // C_4 can be satisfied.
        set $r_y = \min(r_y)$ such that C_1 AND C_4.
    else:
        // C_4 cannot be satisfied.
        set $r_y = \min(r_y)$ such that C_1.

    set $p = \min(p)$ such that C_2.

Figure 3.3: Protocol for choosing IFGT parameters. We first try to find $k < K^*$ that will allow all the constraints to be satisfied with the given $r_y(ideal)$. In many cases, this is not possible so we must set $k = K^*$ and increase r_y. Finally, we choose p to satisfy the truncation error bound C_2. In practice, the k-centers algorithm is run iteratively rather than being run anew each time through the loop.

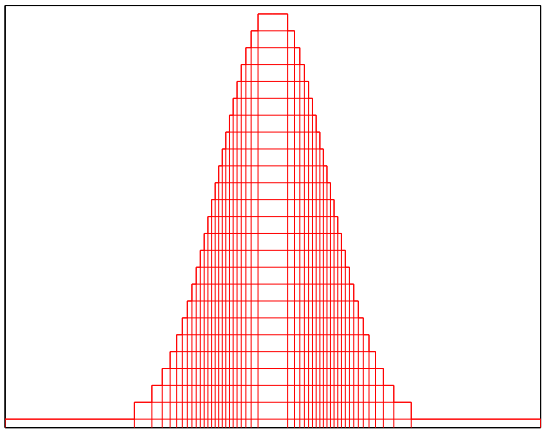


Figure 3.4: Box-Filter approximation of a Gaussian kernel. To achieve a relative error of 2%, 25 boxes are required.

3.3.4 Box Filter

Felzenszwalb, Huttenlocher and Kleinberg [10] note that when the source particles and target points are the vertices in a regular grid, the Sum-Product problem becomes a discrete convolution of the source weights by the kernel function. This is, if the kernel function $K(x_i, x_j)$ can be written as $K(i-j)$, then

$$f_j = \sum_{i=1}^{N} w_i K(i-j) \quad .$$

In general this takes $O(NM)$, where M is the support of the kernel. For certain kernels, however, this can be done more efficiently. The box filter, which has constant value inside some interval and zero elsewhere, allows the computation to be done in $O(N)$. By summing several box filters, we can build a piecewise constant approximation to any smooth separable kernel. See Figure 3.4.

To achieve an error bound of ϵ, we must make the absolute difference between the approximation and the real kernel small. This is best accomplished by composing boxes of equal height such that the vertical space between boxes is equal; see Figure 3.4. Such an arrangement is optimal because the error is bounded by the largest difference between the real value of the kernel and the box approximation. In order to

minimize this value globally, the vertical steps must be of equal size. It is necessary to use a number of boxes B,

$$B \simeq \frac{(K_{max} - K_{min})\, W}{2\epsilon} \quad ,$$

where K_{max} and K_{min} are the maximum and minimum values of the kernel, and W is the sum of source particle weights. The number of boxes can become large if small error bounds are required. In multiple dimensions, the convolution must be performed in each dimension. The cost of this method is $O(DBN)$, where D is the number of dimensions, B the number of box filters required to approximate the kernel, and N the number of points.

3.3.5 Dual-Tree Sum-Product

Alex Gray and Andrew Moore [18, 31, 19, 20] introduced the dual-tree recursion that we use, and have presented algorithms similar to our fast Sum-Product algorithm. Our contributions here are mostly implementation-level improvements, except for section 3.3.6, which presents better bounds on the influence function by using physically-inspired constraints.[2]

Our algorithm (in its simplest form) works for any non-increasing (monotonic decreasing or constant) kernel function of the distance between points, and non-negative particle weights. It is based on a dual-tree recursion, in which the frontier of a 'cross product' of trees is expanded in such a way that only areas of the trees that contain useful information are explored. Our algorithm allows the influence f_j on each target point y_j to be estimated to within a given error bound ϵ.

Alternatively, our algorithm could be adapted to become an any-time algorithm, in which an approximate answer can be returned at any time and the approximation becomes better as time progresses. This is a major difference between the dual-tree family of methods and the family of methods based on series expansions (Fast Multipole Method, Fast Gauss Transform, Improved FGT). The expansion-based methods

[2]Many of the ideas in this section were developed in collaboration with Mike Klaas. Both Mike and I thank the DSG Espresso Pool for supplying the brain fuel that allowed this work to proceed.

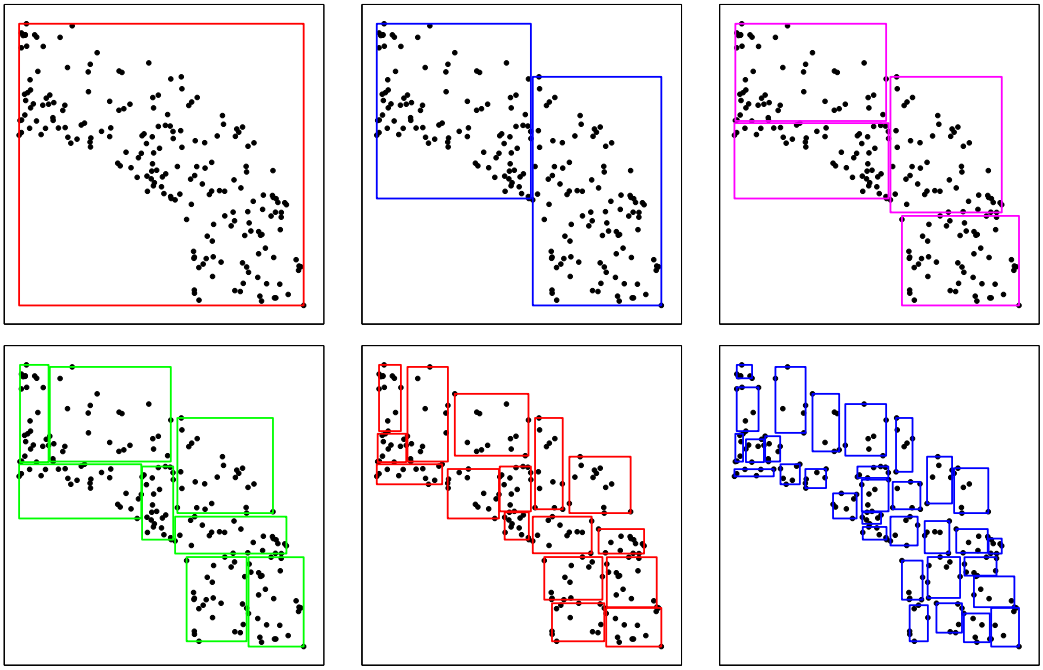


Figure 3.5: Illustration of a *kd*-tree partitioning of space. The points are two hundred samples from a ring with uniformly-distributed radius in $(0.6, 1)$ and uniformly-distributed angle in $\left(0, \frac{\pi}{2}\right)$. Nodes are split along the dimension with the largest range, and the dividing point is chosen so that the two children nodes contain an equal number of particles.

guarantee error bounds based on theoretical bounds on the series truncation error. These bounds are valid for all data distributions, but since they are based on worst-case behaviours, they are often quite loose for average-case problems. The amount of work that must be done is determined *a priori*, without reference to a particular data set. Dual-tree-based methods, on the other hand, are based solely on error bounds determined at run-time; they make decisions that are optimal for a single data set, rather than all possible data sets.

Our dual-tree algorithm begins by creating space-partitioning trees for both the X and Y particles. See Figure 3.5 for an example of such a partitioning. We call these the X tree and Y tree, respectively. The leaves of each tree contain single particles.

Different types of space-partitioning tree, such as the *kd*-tree or ball-tree (anchors hierarchy [31]), can be used. One of the features of these trees is that it is a simple and inexpensive operation to compute the distance bounds between a pair of nodes.

The basis of the algorithm is that, given a node from each of the X and Y trees, we can easily compute upper and lower bounds on the influence of all particles in the X node upon all points in the Y node. We can tighten the influence bounds by *expanding* nodes X and Y; that is, examining their children nodes. See Figure 3.6. The influence upon each Y child is the sum of the influences of the X children. Since the children nodes are more compact, the distance bounds are tighter, which in turn leads to tighter influence bounds. If the difference between the upper and lower influence bounds is much smaller than ϵ, then there is almost nothing to be gained from expanding the nodes. This is where the computational savings of tree-based methods come from. If the nodes contain N_X and N_Y particles, the cost of computing the Sum-Product naively is $O(N_X N_Y)$, while the dual-tree method performs only one operation.

At the beginning of the algorithm, we cache the sum of weights within each node of the X tree. Then, we compute the distance bounds between the root nodes of the X and Y trees. This allows us to compute a bound on the sum of the influence of all particles in the X tree upon all particles in the Y tree. For each Y particle, we store this bound. Finally, we enqueue the X and Y root nodes on the list discussed below.

As the algorithm proceeds, we keep a list of node-node pairs (one node from each of the X and Y trees, henceforth called "node-pairs" or "pairs") that we could expand. Expanding a node-pair involves considering all pairs of the nodes' children. For each child pair, we compute the new distance bound, and along with the total weight of particles in the X child this allows us to compute the influence bound of all points in the X child upon all points in the Y child. We enqueue each child pair, and for each particle in the Y node, update the influence bounds.

- *node-pairs*

The key point is that as long as the nodes of the trees become smaller as we move from the root to the leaves, then the influence bounds become tighter each time we process a pair from the queue. Intuitively, this occurs because the worst-case distribution of mass in the X tree becomes less bad; the distance bounds between the

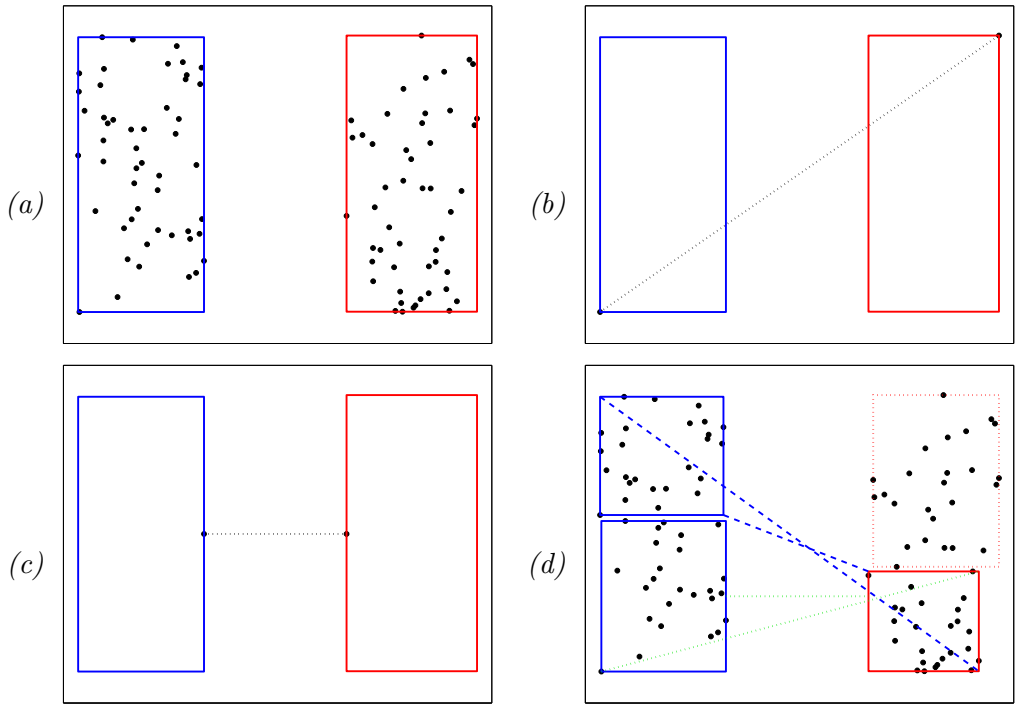


Figure 3.6: Illustration of dual-tree influence bounds.

(a): X (left) and Y (right) nodes containing 50 particles each.

(b): the arrangement of particles that yields the lower-bound influence: all particles are in the farthest corners of the nodes, hence the distance is as large as possible, the value of the kernel function is as small as possible, and therefore the influence is as small as possible. The dotted line shows the upper bound on distance.

(c): an arrangement that leads to the upper-bound influence: all particles are as close together as possible, hence all distances are equal to the lower bound on distance, the kernel function has the largest value possible, and the influence is as large as possible. The dotted line shows the lower bound on distance.

(d): the effect of examining the children of the X and Y nodes. For clarity, the details are shown for only the lower Y child node. The upper distance bounds become smaller, and the lower distance bounds become larger.

child nodes is tighter than that of the parents. If we had exact arithmetic, we could compute the exact answer by expanding all pairs until we reached leaf nodes in both trees.[3]

We know that the error bounds will decrease each time we process a node-pair from the queue. In practice, we would like to know that we are moving toward satisfying the error bound for each Y particle as quickly as possible. To achieve this, we attach a *value* to each node-pair and use a priority queue. The value function is the size of the error bound multiplied by the number of particles in the Y node:

- *value*

$$v(X,Y) = N_Y \times \left(f^{(upper)}(X,Y) - f^{(lower)}(X,Y) \right)$$

where N_Y is the number of points in Y. $f^{(upper)}$ and $f^{(lower)}$ are the upper and lower bounds on the influence of all particle in X upon all particle in Y, defined as:

$$\begin{aligned} f^{(upper)}(X,Y) &= K\left(d^{(lower)}(X,Y)\right) \sum_{i \in X} w_i \\ f^{(lower)}(X,Y) &= K\left(d^{(upper)}(X,Y)\right) \sum_{i \in X} w_i \end{aligned} \tag{3.2}$$

where K is the kernel function and $d^{(lower)}$ and $d^{(upper)}$ are the lower and upper bounds on the distance between X and Y. Note that we cache the sum of weights in X.

Our value function assigns highest priority to node-pairs that are currently contributing most to the total error bound. Note that we cannot guarantee that processing the highest-priority node will lead to the largest possible decrease in total error, since we cannot know in advance how much tighter the child bounds will be compared to the parent bounds. Thus, we cannot guarantee that we are always reducing the error by the largest possible amount per unit of time, but we are guaranteed to be expending effort where it is most needed.

Given upper and lower influence bounds $f^{(upper)}$ and $f^{(lower)}$, if we wish to return an estimate with the smallest guaranteed symmetric error ($|f - \tilde{f}| \leq e$), then

$$\tilde{f}(X,Y) = \frac{1}{2}\left(f^{(upper)}(X,Y) + f^{(lower)}(X,Y) \right)$$

[3] Though this would cost at least $O(MN)$ so would hardly be worthwhile!

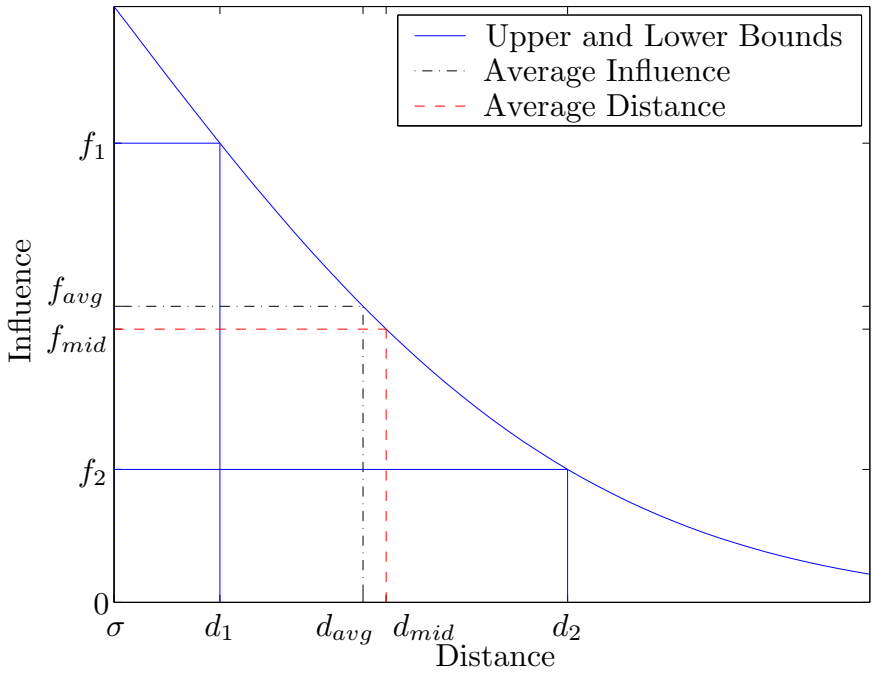


Figure 3.7: A possible explanation for bias in the influence estimates for some problems. A Gaussian kernel from σ to approximately 3σ is shown. The solid lines show upper and lower bounds on the distance and influence. The dash-dotted line shows the average of the upper and lower influences; this is the influence estimate with the smallest guaranteed symmetric error bound. However, a more accurate estimate could be the dashed line, which corresponds to the influence at the average of the upper and lower distance bounds. For concave-up kernels, our estimate will always be biased in this manner.

which has error bound

$$e(X,Y) \;=\; \frac{1}{2}\;\left(f^{(upper)}\left(X,Y\right) - f^{(lower)}\left(X,Y\right)\right) \quad .$$

However, in our experiments we have find that for some problems this estimate will systematically overestimate the true influence f_j. While we have not explored this effect in detail, one explanation could be that rather than the average of the upper and lower influence bounds, a better estimate would be the influence and the average of the upper and lower *distance* bounds. See Figure 3.7. For concave-up kernels, the estimate $\tilde{f}$ given above will always be an overestimate if this explanation is correct. For fairly small ϵ, most of the source particles within one σ (where the Gaussian is concave-down) will be expanded to the leaves, while outside one σ (where the Gaussian is concave-up) many nodes will be unexpanded and hence prone to this effect.

Implementation Details

In developing our algorithm, we discovered several enhancements that may seem unintuitive, yet give considerable performance increases. Only readers interested in such details need read this section.

One might suppose that we need to keep the queue sorted by order of priority. However, all we really need to do[4] is expand the subset of nodes necessary to bring the error bounds low enough; the order in which the work is performed is irrelevant.[5] We found that for Gaussian kernels with reasonable scales, the priorities span a very large range of values: many orders of magnitude. For reasonable error bounds ϵ, we must expand nodes with values ranging over several orders of magnitude. Rather than keep a fully-sorted queue, we create a queue for each order of magnitude (henceforth, "order"). When dequeuing, we choose the first element from the highest-order queue.

By doing this, we avoid expending effort maintaining a fully sorted queue. The consequence is that as we approach the target error (as we are expanding pairs of the final order) we may expand nodes with values smaller than the optimal; we may do

[4]This only applies for the non-anytime version of the algorithm.

[5]Except in so far as we have to process parents before we can process their children.

```
       Inputs: root nodes of X and Y trees, X_r, Y_r; ε
       // Initialization.
1      {U_r, L_r} = w(X_r) K (d^{lower,upper}(X_r, Y_r))
2      for j = 1 to N(Y_r)
3          f_j^{upper,lower} = {U_r, L_r}
4      enqueue(X_r, Y_r, U_r, L_r)
       // Main loop.
5      while NOT (bounds_satisfied ())
6          {X_i, Y_i, U_i, L_i} = dequeue()
           // Expand nodes X_i and Y_i.
7          for yc = 1 to children*(Y_i)
8              Y_c = child*(Y_i, yc)
               // Δ is change in bounds.
9              Δ^{upper,lower} = {−U_i, −L_i}
10             for xc = 1 to children*(X_i)
11                 X_c = child*(X_i, xc)
                   // Compute influence bounds for children.
12                 {U_c, L_c} = w(X_c) K (d^{upper,lower}(X_c, Y_c))
13                 Δ^{upper,lower} += {U_c, L_c}
14                 if NOT (leaf (X_c) AND leaf (Y_c))
15                     enqueue(X_c, Y_c, U_c, L_c)
               // Update bounds on f_j for each particle in Y_c
16             for j = 1 to N(Y_c)
17                 f_j^{upper,lower} += Δ^{upper,lower}
       // Return estimates f̃_j and error bounds e_j.
18     for j = 1 to N(Y_r)
19         f̃_j = (1/2) (f_j^{(upper)} + f_j^{(lower)})
20         e_j = (1/2) (f_j^{(upper)} − f_j^{(lower)})
```

Figure 3.8: Pseudocode for the Dual-Tree Sum-Product algorithm. The notation $\{A, B\} = f(\{X_A, X_B\})$ means $A = f(X_A)$ and $B = f(X_B)$. The enqueue(X, Y, L, U) function places a new element on the priority queue. We use the value function $(U - L) \times N(Y)$. The function *children**() returns the number of children for non-leaf nodes and 1 for leaf nodes. The *child*$^*(X, i)$ function returns the i-th child for non-leaf nodes and the node itself for leaf nodes.

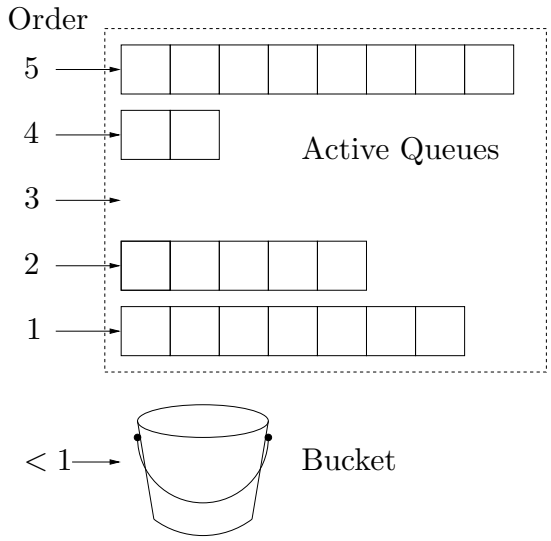


Figure 3.9: Order-of-magnitude queues. There is a set of *active queues*, containing (unsorted) pairs with values of the same order of magnitude. There is also a *bucket* containing (unsorted) pairs with small values.

some unnecessary work. In practice, we find that the amount of unnecessary work we perform is small, and is more than paid for by avoiding the cost of maintaining a fully sorted queue. In our experiments, we found that simply maintaining the fully-sorted property of the queue took a significant portion of the computational effort, even when we used a Fibonacci heap as the underlying data structure,

For kernels whose values do not span such a large range of values, we could use smaller increments than orders of magnitude, such as powers of two. Alternatively, we could use a finer discretization as the error bound approaches ϵ.

As a further enhancement, rather than maintaining a queue for each order, we maintain a fixed number of active queues and a catch-all *bucket*[6] for pairs with small value; see Figure 3.9. Many pairs that are placed in the bucket will never be expanded. Therefore, we would like to minimize the amount of work performed on these pairs, while also allowing the possibility that they *will* be expanded. When expanding a

• *bucket*

[6]We find that the bucket must be made of a hard material, since a hole in the bucket can result in a memory leak.

node-pair, we first compute the upper bound of the influence. From this we can compute an upper bound on the pair's value, by assuming that the lower influence bound is zero:

$$\begin{aligned} f^{(upper)}(X,Y) &= K\left(d^{(lower)}(X,Y)\right) \\ v(X,Y) &= N(Y)\ \left(f^{(upper)} - f^{(lower)}\right) \\ &\leq N(Y)\ f^{(upper)} \end{aligned}$$

If the upper bound on value is small enough that the pair will be put in the bucket, then we don't have to compute the lower influence or value bounds. If at any point the active queues become empty, we create a new set of active queues (with smaller orders) and refill them from the bucket; at this time, we must compute the lower influence bound $f^{(lower)}$ and recompute the value v of pairs that are removed from the bucket. This optimization allows us to avoid a significant amount of work (an upper distance and lower influence bound calculation) for 'hopeless' pairs that are never removed from the bucket. We find this strategy to be faster than the Fibonacci heap approach described by Gray and Moore [20].

As another optimization, we found that checking if we have achieved the error bound is a relatively expensive operation compared to expanding a node-pair. Therefore, we process a batch of node-pairs before checking if we have reached the target error level. We found that batches of one thousand node-pairs gave good results.

Finally, we note that in order to satisfy the error bound ϵ for each point in Y, we must first satisfy the bound that the sum of error bounds is less than $\epsilon N(Y)$. This condition is much cheaper to check, since we need only maintain a total error bound, rather than $N(Y)$ individual bounds. During the first phase (before we have satisfied the sum bound), we can use the "deferred asynchronous propagation" trick of Gray and Moore [20]: rather than adjusting the bounds for each point in Y, we propagate the $\Delta^{\{upper,lower\}}$ values to the immediate children of Y. In the pseudocode of Figure 3.8, lines 16 and 17 are replaced by a simple operation that does not cost $O\left(N\left(Y_c\right)\right)$. When the first phase completes, these deferred Δ values must be propagated to the leaves. A significant portion of the running time of the algorithm is spent in the first

phase, so this can lead to considerable savings.

Extensions

We briefly mention a few simple extensions to our dual-tree Sum-Product algorithm.

First, if would be straightforward to loosen some of the restrictions we have made on the form of the kernel function. At present we consider monotonic kernels $K(X,Y) = K(\|X-Y\|)$, and in our algorithm we explicitly calculate the distance bounds between nodes. The bounds are then simply

$$K^{\{upper,lower\}} = K\left(d^{\{lower,upper\}}\right) \quad .$$

Instead, we could simply demand that bounds on the influence be calculable (in $O(1)$) given two nodes:

$$\begin{aligned} K^{upper} &= U(X,Y) \\ K^{lower} &= L(X,Y) \end{aligned}$$

where U and L are the kernel-bounds functions. This would allow, for example, non-stationary (translationally variant) kernels.

Second, we could allow different kernels for each source point, for certain families of kernels. At present, we demand a uniform kernel:

$$K_i(X_i,Y) = K(X_i,Y)$$

but it would be easy to allow certain families of parametrized kernels:

$$K_i(X_i,Y) = K(X_i,Y,P_i)$$

where P_i is some set of parameters. For example, with Gaussian kernels we could allow each source particle to have a unique variance. At present, the influence bounds are defined by the value of the kernel at the largest and smallest distances. With this change, the upper bound would become the value of the kernel with greatest value at the smallest distance (and vice versa for the lower bound). This is simple in the (unnormalized) Gaussian case: the kernel with greatest value is simply the kernel with

largest variance, and vice versa. We could simply keep track of the variance limits within each subtree of the source tree, in the same way that we track the sum of weights.

Third, we have required non-negative particle weights, since this is typically the case for Sum-Product problems that arise in probabilistic inference. However, for more general Sum-Product settings, relaxing this restriction would be useful. This could be done quite easily: rather than caching the sum of weights within each X subtree, we would cache the sums of positive and negative weights separately. The bounds computations would also change: the upper influence bound would arise when all the particles with positive weights were as near as possible to the target point and all particles with negative weights were as far away as possible. The lower influence bound would be the opposite situation.

3.3.6 Better Bounds for Dual-Tree Methods

In this section we present new bounds that improve the Dual-Tree Sum-Product algorithm. Our bounds on the kernel influence are always as good or better than the bounds currently used. Recall that the bounds (equation 3.2) are central to the Dual-Tree approach. Given a source node X and target node Y, we wish to bound the influence of all particles in the source node on all points in the target node.

As the algorithm proceeds, nodes are expanded in order to tighten the bounds. The estimate for the influence at each target point is bounded by the sum of bounds of all the source nodes. Once the sum of bounds has been tightened to less than the allowable error ϵ, the algorithm completes.

The original bounds, which are used by Gray and Moore and others, are the result of finding the distributions of particles (mass) that maximize and minimize the influence, subject to the constraint that the mass is equal to the masses of the actual particles in the node X. This can be written in continuous form as

$$\begin{aligned} w^*(x) &= \underset{w(x)}{\text{extrema}} \int_X K\left(x', Y\right) w\left(x'\right) \mathrm{d}x' \\ &\quad s.t. \quad \int_X w\left(x'\right) \mathrm{d}x' = W \quad , \end{aligned}$$

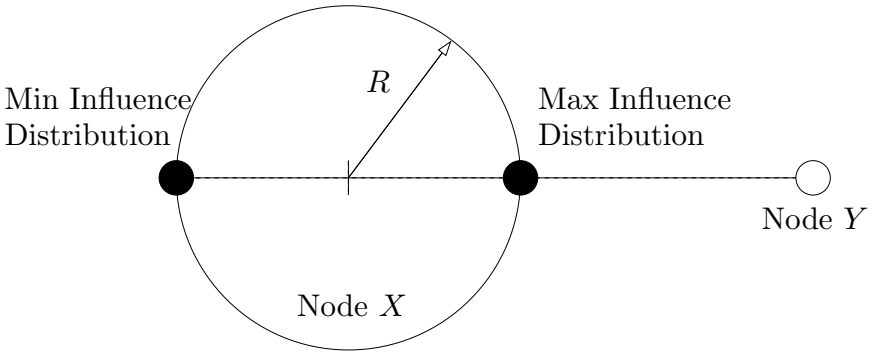


Figure 3.10: Original influence bounds. The maximum-influence distribution is a single particle with mass W at the closest point to node Y, while the minimum-influence distribution is a particle with mass W at the farthest point from Y.

where W is the sum of the weights of particles in node X:

$$W = \sum_{i \in X} w_i \quad .$$

For monotonic decreasing kernels, the extrema distributions collapse to delta functions at the nearest and farthest points, respectively:

$$\begin{aligned} w^*_{(min)}(x) &= W\delta\left(x - \underset{x' \in X}{\operatorname{argmax}} \left|x' - Y\right|\right) \\ w^*_{(max)}(x) &= W\delta\left(x - \underset{x' \in X}{\operatorname{argmin}} \left|x' - Y\right|\right) \quad , \end{aligned}$$

where the argmax and argmin are the points in X that are farthest and nearest from Y. See Figure 3.10.

We now make an analogy between the particles in X and physical particles. The constraint of the original bounds is that the zeroth raw moment of inertia of the function $w^*(x)$ matches that of the particles in X. That is, the total masses must be equal. We add the constraints that the first and second raw moments[7] of inertia must also match those of the particles in X. That is, the original bounds satisfy the

[7] The *raw moments* are defined in terms of distance from the origin of the coordinate system, while the (more intuitive) *central moments* are defined in terms of the center of mass of the object.

constraint C_0, defined as

$$C_0 \quad : \qquad \int_X w\left(x\right)\, \mathrm{d}x \quad = \quad \sum_{i \in X} w_i \quad ,$$

and we add the constraints C_1 and C_2:

$$\begin{aligned} C_1 \quad : \qquad \int_X w\left(x\right) x\, \mathrm{d}x \quad &= \quad \sum_{i \in X} w_i\, x_i \\ C_2 \quad : \quad \int_X w\left(x\right) \left\|x\right\|^2\, \mathrm{d}x \quad &= \quad \sum_{i \in X} w_i\, \left\|x_i\right\|^2 \quad . \end{aligned}$$

The constraint C_1 requires that the center of mass (first moment) of the extrema distributions match that of the particles in X. Constraint C_2 requires that the second moment of inertia match that of X.

Finding the bounds now becomes the problem of finding the minimum and maximum of the constrained optimization

$$\begin{aligned} w^*\left(x\right) \quad = \quad & \underset{w(x)}{\text{extrema}} \int_X K\left(x', Y\right) w\left(x'\right) \mathrm{d}x' \\ & s.t. \;\; C_0,\; C_1,\; C_2 \quad . \end{aligned} \tag{3.3}$$

We make a series of proofs and conjectures as shown in Figure 3.11. First, we show that the optimal solutions must be composed of a finite number of Dirac delta functions (ie, point masses or particles). We conjecture that any such solution can be decomposed into a sum of $D+1$-particle solutions, where D is the dimension of the problem.

• *proof roadmap*

For the upper bound, we conjecture that if we are given a $D+1$-particle solution, it is always possible to find a two-particle solution that satisfies the same constraints and is better. Hence, the maximum solution must be a two-particle solution. Next, we show that the optimal two-particle solution must lie on the line connecting the centers of the source and target nodes. Finally, for some cases we show that the endpoint solution is the optimal two-particle solution.

For the lower bound, if we assume that a two-particle solution is optimal, we find that the particles are not constrained to lie on points but rather to lie on a $D-1$-dimensional spherical surface. We conjecture that this solution is the optimal not only for two-particle solution, but for all solutions.

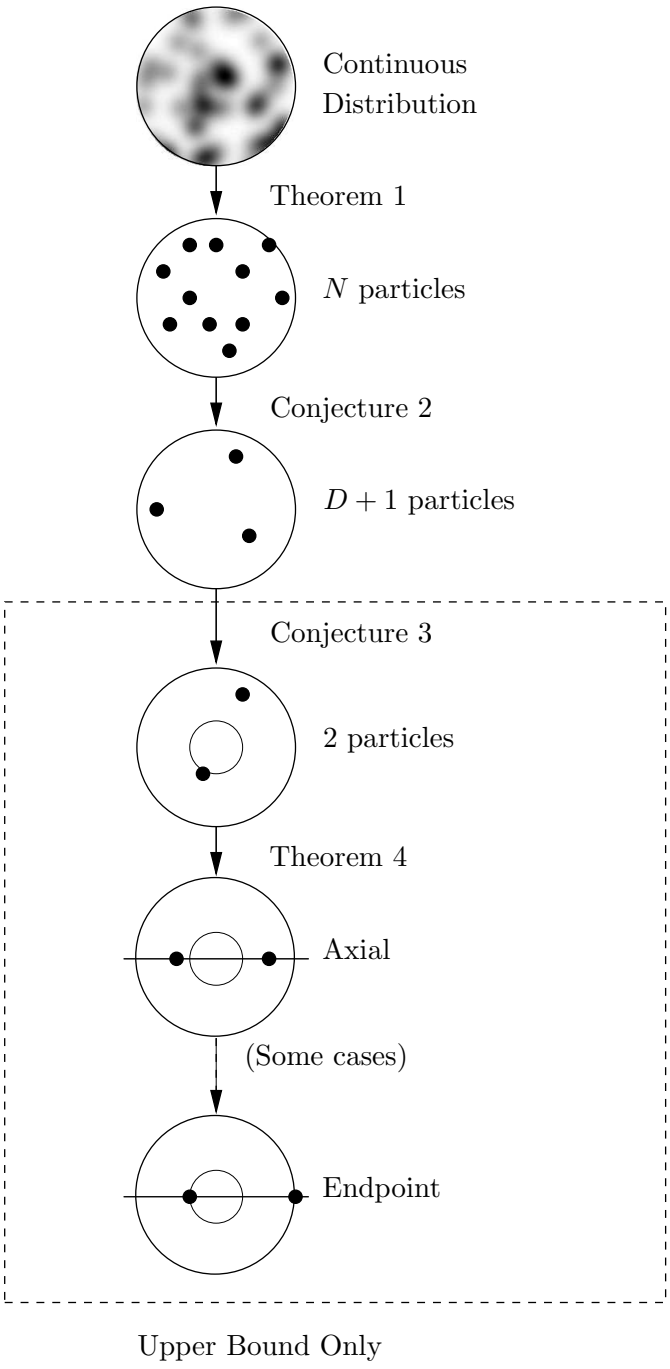


Figure 3.11: A roadmap of the better upper bound proof. See the text (on page 53) for discussion.

The first theorem, which states that the optimal solutions $w^*(x)$ cannot be continuous, follows.[8]

Theorem 1. *For the optimization problem above (equation 3.3), the existence of a continuous optimum $w^*(x)$ implies that*

$$\frac{\partial^3 K(x,y)}{\partial x^3} = 0 \quad .$$

Proof. Equation 3.3 can be solved as an infinite linear programming problem. Assume that $w^*(x)$ is continuous. We set up a Lagrangian with multipliers $\Theta = \{\theta_0, \theta_1, \theta_2\}$:

$$\mathcal{L}(w(x), \Theta) = \int_X w(x) K(x,y)\, \mathrm{d}x - \sum_{i=0}^{2} \theta_i \left[\int_X x^i \; w(x)\, \mathrm{d}x - \alpha_i \right] \quad ,$$

where α_i are the raw moments of the particles in node X:

$$\alpha_i = \sum_{j \in X} x^i \; w_j \quad .$$

Differentiating the Lagrangian (3.4) by $w(x)$ and setting to zero yields

$$\mathbb{I}_X K(x,y) = \mathbb{I}_X \left(\sum_{i=0}^{2} \theta_i \; x^i \right) \quad , \tag{3.4}$$

where $\mathbb{I}_X$ is the set membership function. Hence, $K(x,y)$ is a polynomial of degree ≤ 2 when $x \in X$. Therefore, the third derivative of $K(x,y)$ is zero. □

Corollary. *Given a kernel $K(x,y)$ with 3 or more non-zero derivatives (the Gaussian, for example), the extrema distributions $w^*(x)$ are composed of a finite sum of Dirac masses.*

Next, we conjecture that rather than dealing with a finite number of Dirac masses (henceforth called *particles*), we must deal with only $D+1$ particles, where D is the dimension of the problem.

Conjecture 2. *Any sum of N particles can be written as a sum of sets of up to $D+1$ particles:*

$$w(x) = \sum_{i=1}^{N} w_i \delta(x - x_i) \rightarrow w(x) = \sum \sum_{j=1}^{D+1} w_j \delta(x - x_j) \quad .$$

[8] This theorem is the result of discussions with Nando de Freitas, Joel Friedman, and Mike Klaas.

Sketch of proof. We show this by construction. See Figure 3.12 for an example in two dimensions. Note that the weights are non-negative. This implies that it is possible to find a set of up to $D+1$ particles whose convex hull contains the center of mass. The construction proceeds as follows. Start with the original distribution of N particles with arbitrary (positive) masses. We will iteratively remove mass from this distribution while adding it to the equivalent distribution that we are constructing. At each step, select $D+1$ points whose convex hull contains the center of mass. Choose portions of each of these particles' mass such that the center of mass coincides with the total center of mass. It is always possible to select some positive amount of mass in this way. Add these masses to the distribution under construction, and remove them from the original distribution. Repeat until all mass has been removed from the original distribution. □

Corollary. *Assume we are given a set of N particles that comprise an optimal solution to the optimization problem (equation 3.3). Any such solution can be written as a sum of solutions comprised of sets of $D+1$ particles. Since the optimization and constraints are linear in weight, the solutions are independent. It thus suffices to find the optimal solutions of equation 3.3 comprised of $D+1$ particles.*

Now, consider the upper bound (the maximum) of equation 3.3. We move from $D+1$ particles to two particles. We do not yet have a proof for this.

Conjecture 3. *Given any set of $D+1$ particles (with moments $\alpha_i, i = \{0,1,2\}$), it is possible to find two particles that have the same moments and equal or greater influence. This is, there exist points x_1 and x_2 and weights w_1 and w_2 such that*

$$w_1 K(x_1, y) + w_2 K(x_2, y) \;\geq\; \sum_{i=1}^{D+1} w_i K(x_i, y) \quad ,$$

and the moments of the two particles match that of the $D+1$ particles.

Next we show that the maximum two-particle solution must have both particles on the line connecting the centers of the source and target nodes.

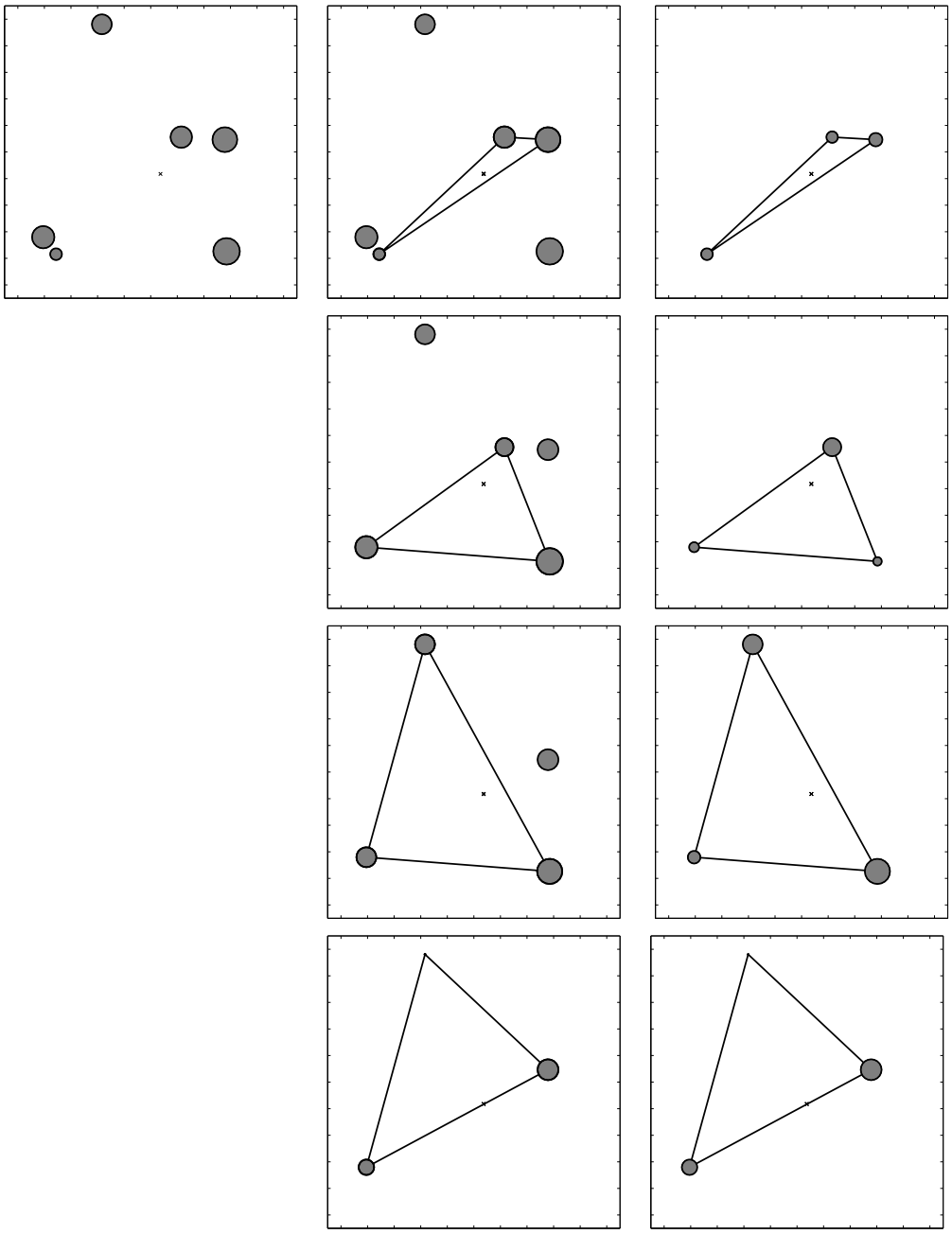


Figure 3.12: Example of the proof by construction. The original distribution is the top-left box. The left column shows the remainder of the original distribution. The right column shows the constructed triangle. Each row is one step in the decomposition process. The area of each circle represents its mass. In the final step, all the remaining mass is removed from the original distribution.

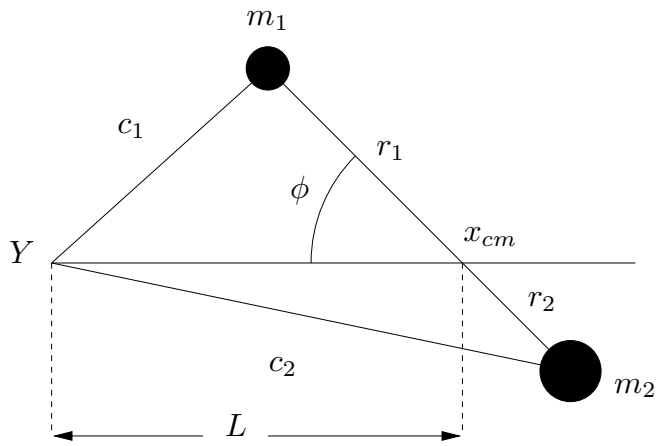


Figure 3.13: Rotation of a two-particle system. The target point is Y (on the left). The particles are m_1 and m_2; their center of mass is x_{cm}, which is also the center of mass of the node X. $r_1 \geq r_2$ so $m_1 \leq m_2$. L is the distance from the nearest point in node Y to x_{cm}. The distance from Y to m_1 is c_1 and the distance from Y to m_2 is c_2.

Theorem 4. *Assume that the center of mass is the center of the source node, and that the kernel is Gaussian. The maximum two-particle solution to equation 3.3 occurs when both particles are located on the line connecting the centers of the source and target nodes.*

Proof. We call the line connecting the centers of the source and target nodes XY. Assume we are given an optimal solution S that satisfies the constraints (C_0, C_1, C_2), and has particles not on XY. We will show that it is possible to rotate the particles about their center of mass, giving solution S'. By doing this, we can improve the solution; the optimal rotation is to place the particles on the line XY. This contradicts the assumption that S is an optimal solution.

First, note that rigid rotation about the center of mass preserves the moments, so if S satisfies the constraints then so does S'.

The geometry is shown in Figure 3.13. The distances c_1 and c_2 can be calculated using the cosine law, which allows us to write the influence as a function of the angle

of rotation ϕ. Setting the first derivative to zero, we find that potential solutions are

$$\begin{aligned} \phi_1 &= 0 \\ \phi_2 &= \pi \\ \cos\phi_3 &= \frac{r1 - r2}{2L} \quad . \end{aligned}$$

Taking the second derivative, we find that ϕ_1 and ϕ_2 are local maxima and ϕ_3 is a local minima. Since ϕ is an angular variable, the global maximum is the maximum of the local maxima. Both ϕ_1 and ϕ_2 correspond to solution in which the particles are on the line XY. □

This theorem (which relies on a number of conjectures which remain to be proven) is very useful. For a two-particle solution in which the particles lie on the line XY, the moment constraints allow us to write the masses of the two particles (w_1 and w_2) and the position of the second particle (x_2) in terms of the position of the first particle (x_1). See Figure 3.14. The constraint that both particles be inside node X (ie, have distance to the center of the node not greater than R) constrains x_1 to lie in a range of radii $[R_{min}, R]$. We are thus left with a one-dimensional constrained optimization in one variable, which is much simpler to solve than the original multi-variable, multi-dimensional problem.

Assume that the target is at position $y > R$. As the first particle moves from R_{min} to R, its distance to y decreases so the value of the kernel function increases. Meanwhile, its mass w_1 decreases. The second particle, meanwhile, is constrained to move from $-R$ to $-R_{min}$ and its mass increases. Clearly, the influence of the second particle increases as this occurs, while the influence of the first particle may increase or decrease, depending on the relative importance of mass relative to kernel value.

While it is possible to solve this optimization numerically at run time, the added computational expense of doing this is likely to be greater than the gain yielded by the better bounds. Therefore, we would like to find analytically conditions where the maximum is one of the endpoints ($x_1 = R_{min}$ or $x_1 = R$).

In the following, we consider the Gaussian kernel. Given a bandwidth σ, node

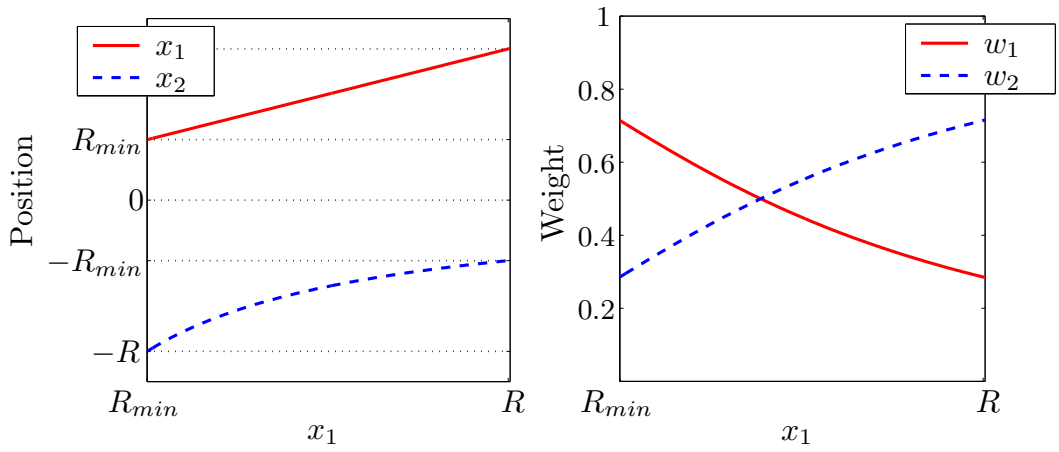


Figure 3.14: Positions and weights in the two-particle system. Left: positions of the two particles as x_1 moves from R_{min} to R. Right: weights of the two particles as x_1 moves from R_{min} to R.

radius R, and moment constraints, we want to show that for targets outside a particular range, the optimal solution occurs at the endpoint $x_1 = R$. If this condition holds, then we can compute the better upper bound at only marginally greater cost than the original bound.

We have thus far found it impossible to derive closed-form expressions for the conditions that must hold for $x_1 = R$ to be the solution. Running numerical tests allows us to get a sense of the structure of the solution space: see Figure 3.15.

If we try to find analytic expressions for the boundaries of the regions where the endpoint solutions hold, we encounter equations of the form

$$p_1 \exp(t_1) = p_2 \exp(t_2) \quad ,$$

where p_1 and p_2 are polynomials and the terms t_1 and t_2 are not equal. It may be impossible to derive closed-form solutions to these equations. We may be forced to resort to numerical methods or weaker bounds.

It has been difficult to prove even weaker bounds. For example, we can focus on the first particle and determine when the derivative of the influence of the first particle is positive. If we can show it is positive in $[R_{min}, R]$, then $x_1 = R$ is the maximum.

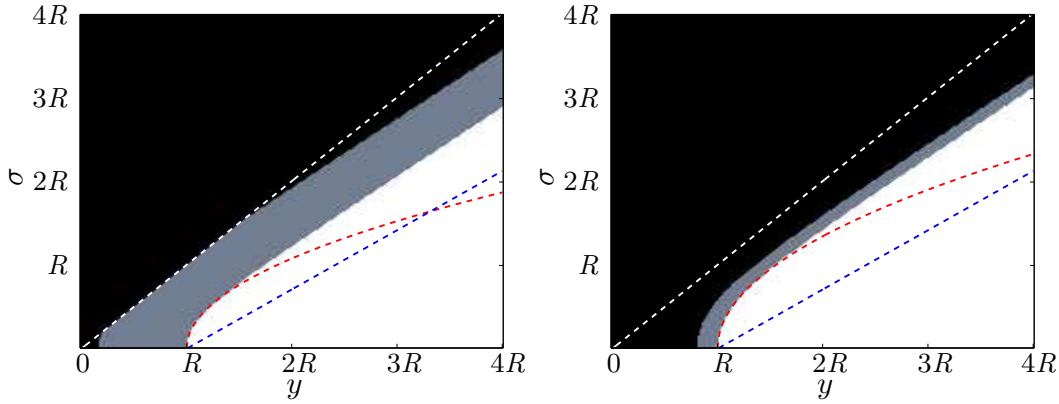


Figure 3.15: Solution phases of the two-particle system. The black area shows where the endpoint $x_1 = R_{min}$ is the maximum solution; the white area shows where $x_1 = R$ is the solution. In the gray area, the maximum is in the interior of the range. The left plot shows the results for a source node with small second moment, while the right plot shows a large second moment. The lines show the bounds we would like to prove. The white line separates the $x_1 = R_{min}$ region. The red line shows where $x_1 = R$ is a local maximum of the influence of the first particle. The blue line shows where the point of inflection of the Gaussian occurs at R.

We can ignore the second particle because the derivative of its influence with respect to x_1 is always positive; this weakens the bound.

Trying this, we find that the minimum of the derivative occurs either at $x_1 = R$ or at some point in the interior that is the solution to a cubic equation. It would be straightforward to solve this numerically, but it is difficult to handle analytically. If we ignore the potential interior solution and assume that one of the endpoints is the solution, then we arrive at the first bound shown in Figure 3.15. This bound is

$$\sigma \leq (y - R)(R + R_{min})$$

Unfortunately, as y (the distance to the target) becomes large, this bound becomes weak: it requires $\sigma \leq \Theta\left(\sqrt{y}\right)$, while the true bound appears to go as $\sigma \leq \Theta\left(y\right)$.

Another bound that may be provable and has better asymptotic behaviour is

$$\sigma \leq \frac{1}{\sqrt{2}}(y - R) \quad ,$$

which demands that the Gaussian be concave-up at R. Being concave-up is not in itself a sufficient condition for the solution to be $x_1 = R$, but the bound seems to hold empirically. This bound is shown in Figure 3.15.

Better Lower Bound

Our better lower bound also requires several unproven conjectures. If we assume that a two-particle solution is optimal, then we find that the minimum solution to equation 3.3 is not a point but rather a curve. In this optimal solution, both particles sit on the radius of a circle centered at y_f. Recall that y_f is the point in the target node Y that is farthest from the center of source node X. This can be shown by writing the total influence in terms of the position of one of the particles, setting the derivatives to zero, and solving the resulting simultaneous equations.

The result is the lower bound

$$f(X, y) \geq W \exp\left(-\frac{W y_f^2 + M}{W \sigma^2}\right) \quad ,$$

where M is the (central) second moment,[9] and W is the sum of masses in X.

Since the two-particle solution is not a point but rather a surface, we conjecture that this surface is also the solution to the general problem.

Testing of Better Bounds

We tested our better bounds by running the Dual-Tree algorithm with the Anchors Hierarchy, computing at each step both the standard bounds and the better bounds described above. The results are shown in Figure 3.16. The better upper bound is at most half as large as the original bound. The better lower bound is considerably larger than the original bound.

3.4 Max-Product

The Max-Product problem is simply the Sum-Product problem with the summation converted to a maximization:

$$\begin{aligned} f_j^* &= \max_{i \in [1,N]} \left\{ w_i \; K_i \left(\|x_i - y_j\| \right) \right\} \\ x_j^* &= \operatorname*{argmax}_{i \in [1,N]} \left\{ w_i \; K_i \left(\|x_i - y_j\| \right) \right\} \quad , \quad j = 1 \ldots M \quad . \end{aligned}$$

where, as before, w_i is the weight of source particle x_i and $K_i(d)$ is a kernel function of the distance between sources x and targets y. We focus on the case where the kernel function is uniform: $K_i(d) = K(d)$.

The Max-Product problem can be seen as a search for the source particle x_j^* among the N particles in X that has the largest influence upon target point y_j.

3.4.1 Distance Transform

Felzenszwalb and Huttenlocher [11] note that for cases where the source and target points are the vertices of a regular grid, the Distance Transform [4, 12] can be used to solve the Max-Product problem.

[9] Although we discussed the raw moments earlier, it is simpler to consider the central moments here; it is simple to convert between the two.

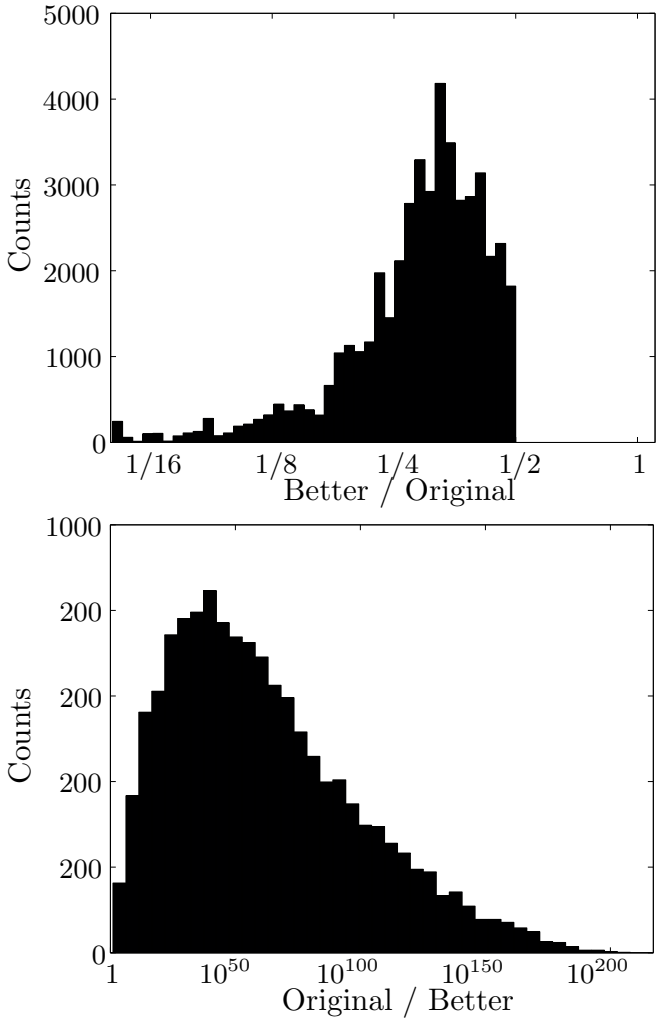


Figure 3.16: Better bounds results. Top: Upper bound. Bottom: Lower bound.

For example, the truncated linear and quadratic cost functions

$$\begin{aligned} c_1(x_i, x_j) &= \min(|x_i - x_j|, d) \\ c_2(x_i, x_j) &= \min\left(|x_i - x_j|^2, d\right) \quad , \end{aligned}$$

which correspond to the kernel functions

$$\begin{aligned} K_1(x_i, x_j) &= \exp\{-\min(|x_i - x_j|, d)\} \\ K_2(x_i, x_j) &= \exp\left\{-\min\left(|x_i - x_j|^2, d\right)\right\} \end{aligned}$$

can be solved with distance transforms, if the state space is a regular grid.

The Distance Transform takes advantage of the fact that on a regular grid the kernel function changes in a predictable way. We reproduce the example of the distance transform in one dimension from [10], except with weighted kernel functions. See Figures 3.17 and 3.18. This example uses the kernel function K_1 above. We work in negative log space so we use cost function c_1. Assume that the truncation term d is very large so has no effect in this example. The problem of finding the kernel with maximum influence now becomes the problem of finding the lower envelope of the cost function c_1. Note that in one dimension, the cost function of each source particle is an inverted one-dimensional cone (a "V"); the weight of the particle determines the cost of the base of the cone.

In one dimension, the sources and targets do not have to be embedded in a regular grid. We can simply sort the sources and proceed as described in the example, except that instead of simply adding one, we must keep track of the distances between points. In higher dimensions, however, this becomes infeasible, since the multi-dimensional Distance Transform depends upon many points sharing coordinate values.

The distance transform is fast: it costs $O(DN)$, where D is the dimension of the problem and N is the number of grid points. While distance transform methods are very fast, their domain is somewhat limited: except for one-dimensional problems, they cannot be used in conjunction with Monte Carlo gridding methods (sections 2.2.3 and 2.2.4, for example).

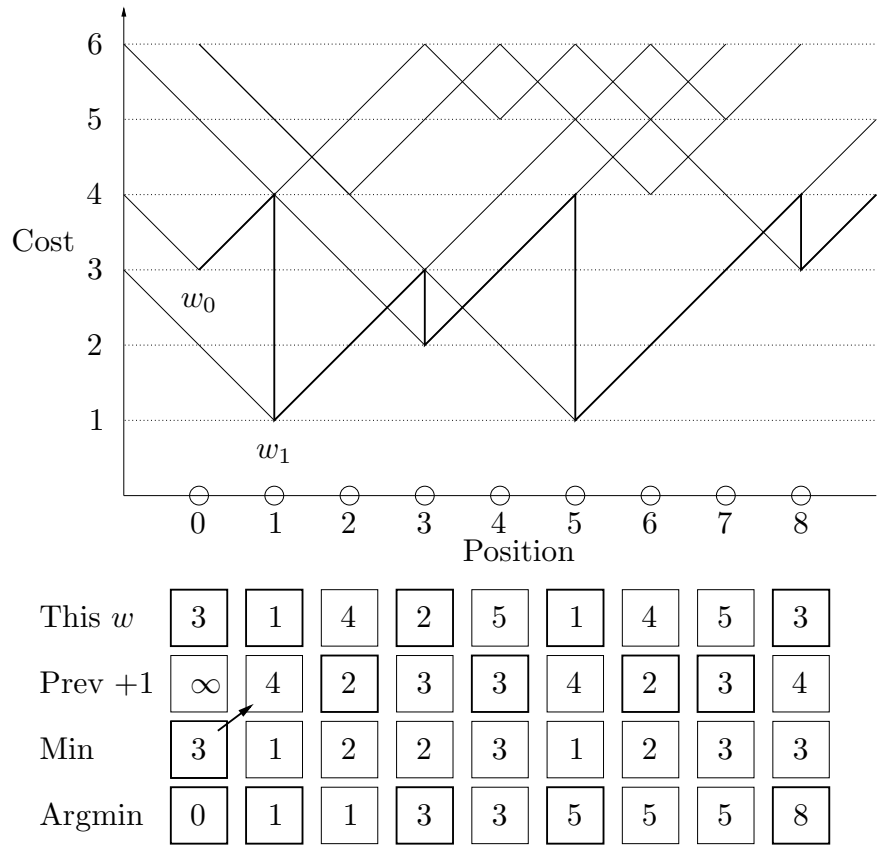
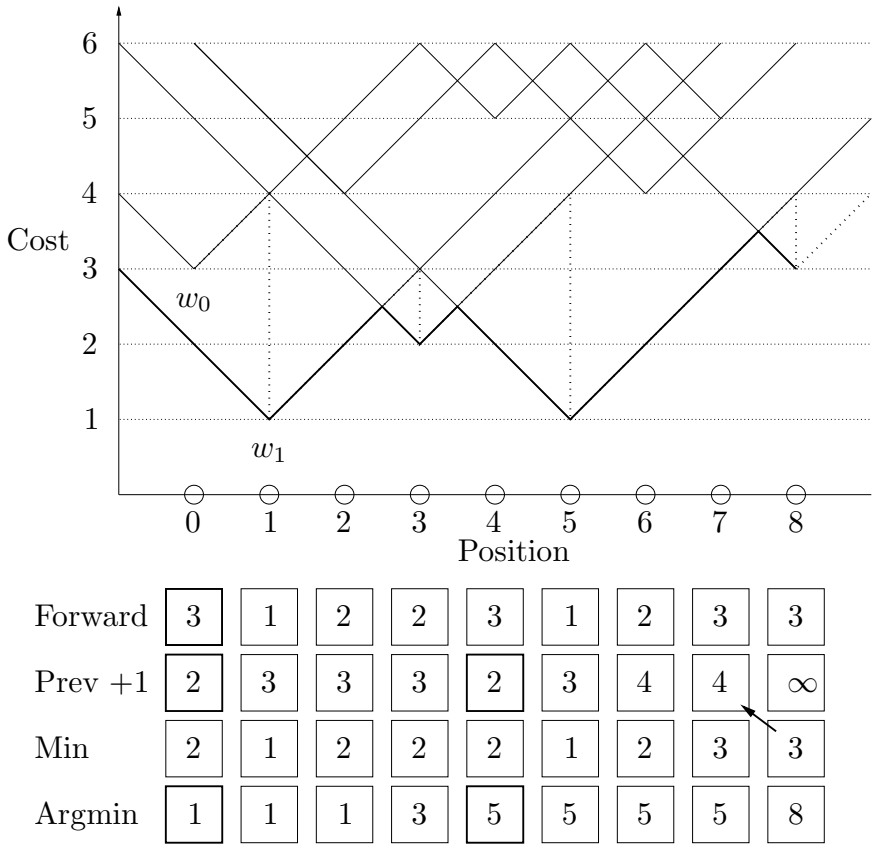


Figure 3.17: Distance Transform example: part 1: forward pass.

The X and Y points are the regular grid $\{0 \ldots 8\}$. Each X particle is represented by a cone; the weight of the particle determines the height of the cone (ie, the minimum cost). The find the Max-Product, we find the lower envelope of all the cones (the min), and the cones to which the lower envelope belongs (the argmin).

We first make a forward pass through the points. At each position i, we compute $\text{Min}\,(i) = \min\,(\text{Min}\,(i-1)+1\ ,\ w_i)$. That is, we either take the previous answer plus one (labelled "Prev + 1"), or we take the cone anchored at i, whose value is w_i (labelled "This w").

If the previous answer was chosen, the argmin is unchanged; if cone i was chosen, the argmin becomes i. The boxes where this occurs are highlighted. This process projects the influence of the cones to the right. In the reverse pass we gather the influence to the left; see the following Figure.



Forward	3	1	2	2	3	1	2	3	3
Prev +1	2	3	3	3	2	3	4	4	∞
Min	2	1	2	2	2	1	2	3	3
Argmin	1	1	1	3	5	5	5	5	8

Figure 3.18: Distance Transform example: part 2: reverse pass.

In the reverse pass of the Distance Transform algorithm, we start from the right and update the values computed in the forward pass. At each point i, we consider the value computed in the forward pass (labelled "Forward"), and the value of the previous answer plus one.

This is, the minimum at position i is:

$$\text{Min}\,(i) = \min\left(\text{Min}\,(i+1)+1\ ,\ \text{Forward}\,(i)\right) \quad ,$$

where $\text{Forward}\,(i)$ is the $\text{Min}\,(i)$ value computed in the forward pass.

If the previous value is chosen, the argmin is updated. The boxes where this occurs are highlighted; these correspond to the locations where the dotted line is different from the solid line.

3.4.2 Dual-Tree Max-Product

As in the Sum-Product case, we develop a dual-tree strategy. The algorithm is based on bounding the distance and weight, hence the influence, of subtrees of X particles upon subtrees of Y particles. Unlike the Sum-Product case, the Max-Product algorithm gives the exact result.

We begin by constructing space-partitioning trees for the X particles and Y points. The leaf nodes of these trees can contain multiple points.[10] We also cache at each node in the X tree the maximum particle weight contained by the node. At leaf nodes, we sort the particles in order of decreasing weight.

The algorithm proceeds by doing a depth-first recursion down the Y tree. For each node, we maintain a list of X nodes that could contain the best particle. For each X node we compute the lower and upper bounds of the maximum influence of particles in the node on all points in the Y node. The largest lower bound on influence is the *pruning threshold*: any candidate node whose upper bound is less than this threshold cannot possibly contain the best particle, and hence need not be considered. See Figures 3.19 and 3.20.

• *pruning threshold*

In each recursive step, we choose one Y child on which to recurse. Initially, the set of X candidates is the set of candidates of the parent. We sort the candidates by lower bound, which allows us to explore the most promising nodes first. For each of the candidates' children, we compute the lower bound on distance and hence the upper bound on influence. Any candidates that have upper bound less than the pruning threshold are pruned. For those that are kept, the lower influence bound is computed; these nodes have the potential to become the new best candidate.

As we descend the trees, the influence bounds become tighter so we are able to prune more and more nodes, until at the leaf level we are, ideally, left with only a few nodes. Once we reach the leaf nodes, we begin looking at individual particles. The candidate nodes are sorted by lower influence bound, and the particles are sorted by

[10] The number of particles in the leaf nodes influences the tradeoff between pruning based on distance and pruning based on weight. We have not investigated this effect in practice. In our experiments we allow up to 25 particles in leaf nodes.

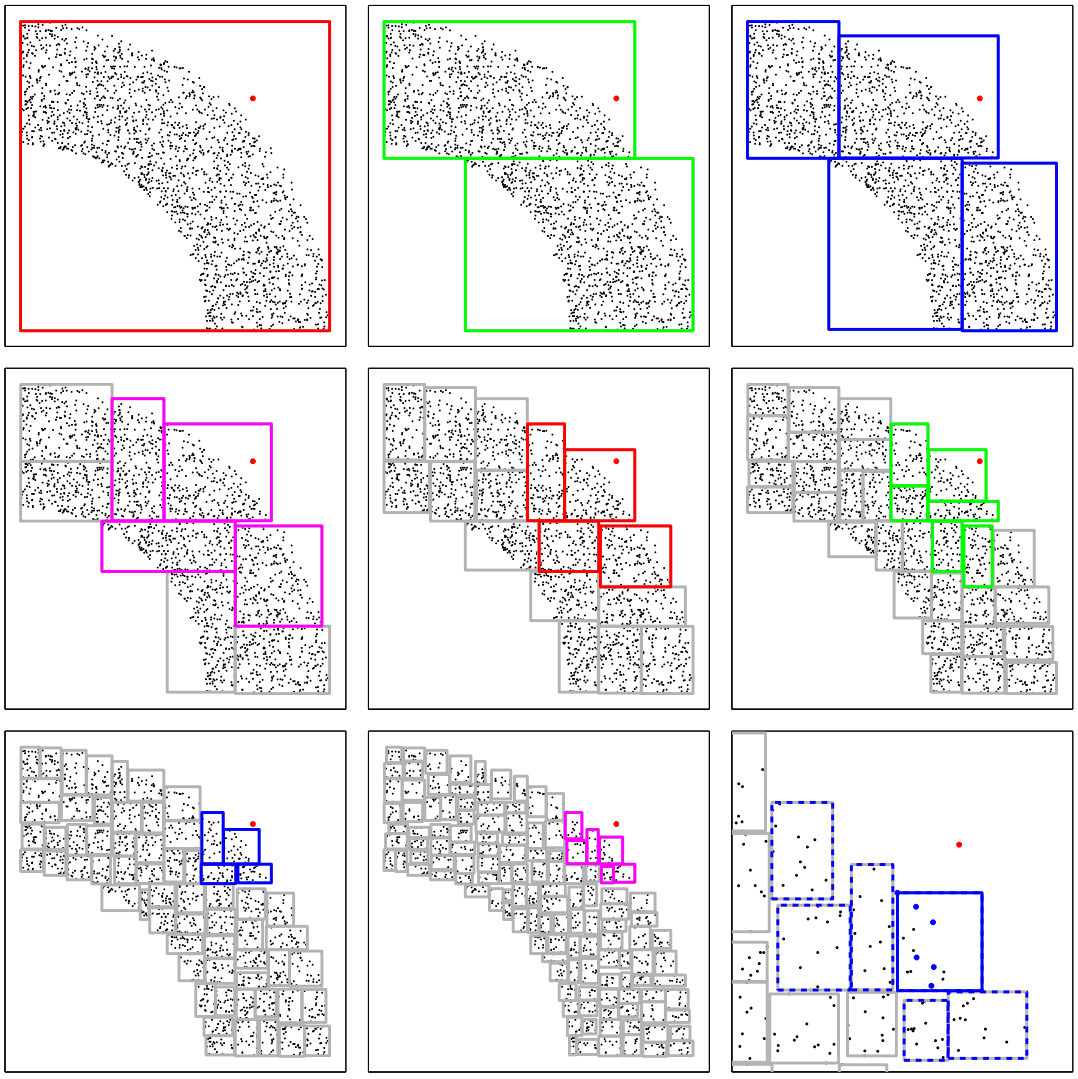


Figure 3.19: Dual-tree Max-Product pruning. The X particles are 2000 samples from the ring with uniformly distributed radius in $(0.6, 1)$ and uniformly distributed angle in $\left(0, \frac{\pi}{2}\right)$. The weights are uniformly distributed in $(0, 1)$. The kernel is a Gaussian with scale $\sigma = 0.1$. The pruning steps for a single Y point (the red dot) are shown. Each plot shows one level in the X tree. The candidate (coloured) and non-candidate (gray) nodes are shown. In the bottom-right plot, a close-up of the six final candidate nodes is shown (dashed blue). The single box whose particles are examined is shown (solid blue). The subset of the particles that are examined individually is shown (blue).

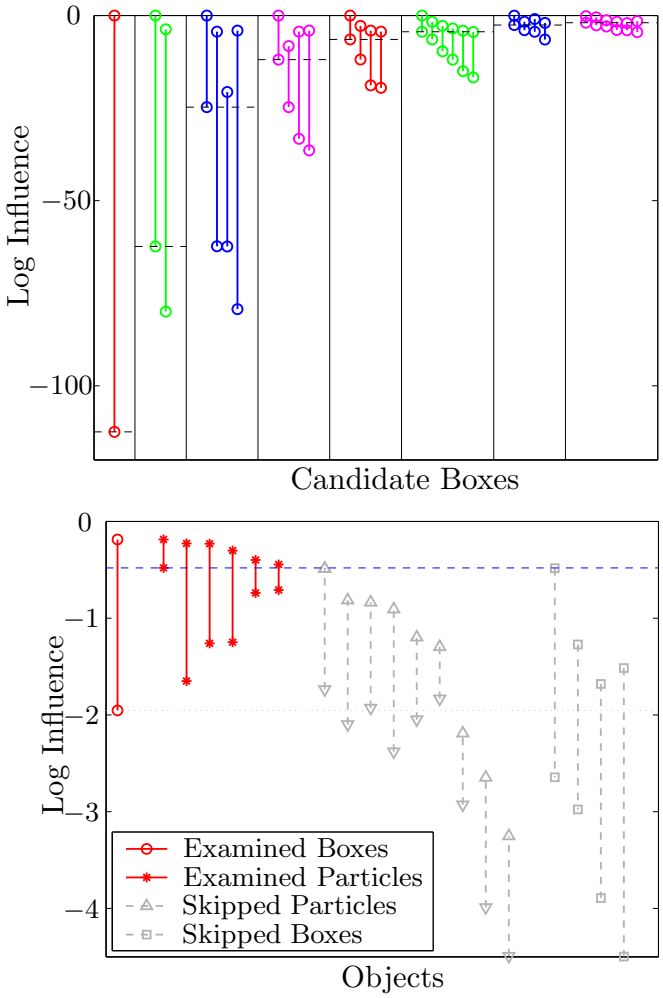


Figure 3.20: Dual-tree Max-Product example.

Top: the influence bounds for the nodes shown in the previous figure. The pruning threshold at each level is shown (dashed line), along with the bounds for each candidate node.

Bottom: pruning at the leaf level: in the example, six leaf nodes are candidates. We begin examining particles in the first box. As it happens, the first particle we examine is the best particle (the correct answer). Pruning by particle weight (the upper marker) allows us to ignore all but the first six particles. The pruning threshold is then sufficiently high that we can prune the remaining candidate nodes without having to examine any of their particles. There were 2000 particles in X, of which six nodes (containing 94 particles total) were candidate leaf nodes. Of these, only six particles from the first node were examined individually.

weight, so we examine the most promising particles first and minimize the number of individual particles examined. In many cases, we have to examine only the first few particles in the first node, since often the pruning threshold increases sufficiently that the remaining candidate nodes can be pruned.

The dual-tree strategy is effective because, for a given node in the Y tree, the list of candidate nodes in the X tree is valid for all the points within the Y node. In this way, pruning decisions are made once rather than being 'rediscovered' for each Y point.

3.5 Application to Inference in Graphical Models

In the section we describe how to apply our fast Sum-Product and Max-Product algorithms to Belief Propagation, Maximum-Belief Propagation, and Particle Smoothing.

3.5.1 Belief Propagation

The expensive operation in the Belief Propagation algorithm is computing the messages. Recall that the message from node j to node i is (equation 2.4):

$$m_{j,i}(x_i) = \sum_{j=1}^{N_j} \psi_j(x_j)\ \psi_{j,i}(x_j, x_i) \prod_{k \in N(j) \text{ except } i} m_{k,j}(x_j) \quad ,$$

which costs $O(N_i N_j)$, where N_j is the number of states at node j[11] and N_i is the number of states at node i. For each of the N_i states, we compute a sum of N_j terms.

If the potential function $\psi_{j,i}(x_j, x_i)$ can be written as a kernel function $K(d(x_j, x_i))$, (where d is a distance metric) then our fast Sum-Product methods can be applied easily. We specialize to the case where $d(x_j, x_j) = \|x_j - x_i\|$, though as discussed in section 3.3.5, this restriction could be relaxed.

First, note that $\psi_j(x_j)$ and the incoming mesasges $m_{k,j}(x_j)$ do not depend on x_i. These terms are combined to become the weight $w(x_j)$. Meanwhile, the potential

[11] For Discrete BP, this is simply the number of discrete states, while for Continuous BP (section 2.2.3) it is the number of particles (discretized states).

```
     INPUTS: root nodes of X and Y trees, X_r, Y_r.
     ALGORITHM:
1      leaves = {}
2      candidates = {X_r}
3      max_recursive(Y_r, leaves, candidates, −∞)
     FUNCTION  max_recursive(Y, leaves, candidates, τ)
4      if (leaf(Y)  AND  candidates = {})
         // Base Case: reached leaves.
5        max_base_case(Y, leaves)
6      else
         // Recursive case: recurse on each Y child.
7        foreach y ∈ children*(Y)
8          τ_y = τ
9          valid = {}
10         foreach p ∈ candidates
             // Check if we can prune parent node p.
11           if (w(p) K (d^lower(p, y)) < τ_y)
12             continue
13           foreach x ∈ children(p)
               // Compute child bounds.
14             f^{upper,lower}(x) = w(x) K (d^{lower,upper}(x, y))
             // Set pruning threshold.
15           τ_y = max(τ_y , max_x (f^lower(x)))
16           valid = valid ∪ {x ∈ children(p) : f^upper(x) ≥ τ_y}
17         valid = {x ∈ valid : f^upper(x) ≥ τ_y}
18         leaves_y = {x ∈ valid : leaf(x)}
19         candidates_y = {x ∈ valid : NOT (leaf(x))}
20         sort(leaves_y by f^lower)
21         max_recursive(y, leaves_y, candidates_y, τ_y)
```

Figure 3.21: Dual-tree Max-Product algorithm, part 1. In the recursive step, we recurse on each child of the Y node. We examine the children of each candidate node, and collect the set of children above the pruning threshold τ_y, as we raise the threshold (lines 10–16). Finally, we select the nodes above threshold, sort the non-leaves, and recurse (lines 17–21).

```
     FUNCTION  max_base_case(Y, leaves)
1      foreach x ∈ leaves
2        f^{upper,lower}(x) = w(x) K(d^{lower,upper}(x, Y))
3      τ = max_x (f^{lower}(x))
4      leaves = {x ∈ leaves : f^{upper}(x) ≥ τ}
5      sort (leaves by f^{lower})
       // Examine individual y points.
6      foreach y ∈ Y
7        τ_y = τ
8        foreach x ∈ leaves
           // Prune nodes by Y (cached), then by y.
9          if ((f^{upper}(x) < τ_y)  OR  (w(x) K(d^{lower}(x, y)) < τ_y))
10           continue
           // Examine individual x particles.
11         foreach i ∈ x
             // Prune by weight.
12           if (f^{upper}(x) w(i)/w(x) < τ_y)
13             break
14           f(i) = w(i) K(d(i, y))
15           if (f(i) > τ_y)
                 // i is the new best particle.
16               τ_y = f(i)
17               x*(y) = i
```

Figure 3.22: Dual-tree Max-Product algorithm, part 2 (base case). We first calculate the influence bounds of the candidates, find the pruning threshold, and prune (lines 1–5). We then examine individual y points. We prune based on the cached bound (for the parent node Y) and then on y (lines 6–10). Next, we look at individual particles in the node x. The particles are sorted by weight, so when a particle is reached whose weight leads to an influence below threshold, we can skip the remaining particles (lines 13–14). Finally, we compute the influence of the particle and decide if it is the new best (lines 14–17).

function becomes the kernel function:

$$\begin{aligned}\left(\psi_j\left(x_j\right)\prod_{k\in N(j)\text{ except }i} m_{k,j}\left(x_j\right)\right) &\Longrightarrow w_j \\ \psi_{j,i}\left(x_j,x_i\right) &\Longrightarrow K\left(\|x_j-x_i\|\right) \\ m_{j,i}\left(x_i\right) &\Longrightarrow \sum_{j=1}^{N_j} w_j\ K\left(\|x_j-x_i\|\right)\quad .\end{aligned}$$

Now, computing the message is clearly a Sum-Product problem, and the applicable fast method can be plugged in.

3.5.2 Maximum-Belief Propagation

Our fast Max-Product algorithms can be applied to the Maximum-Belief Propagation algorithm in the same way the Sum-Product is used for normal Belief Propagation. The expensive operation arises when computing the messages (equation 2.5):

$$m^*_{j,i}\left(x_i\right) \;=\; \max_{x_j}\left\{\ \psi_j\left(x_j\right)\ \psi_{j,i}\left(x_j,x_i\right)\prod_{k\in N(j)\text{ except }i} m_{k,j}\left(x_j\right)\ \right\}\quad .$$

This costs $O\left(N_iN_j\right)$ if the number of states at nodes i and j are N_i and N_j, respectively; we perform a maximization over N_j terms for each of N_i sites.

As in the BP case, our fast methods can be applied in cases where the potential function $\psi_{j,i}\left(x_j,x_i\right)$ can be written as a kernel function:

$$\begin{aligned}\left(\psi_j\left(x_j\right)\prod_{k\in N(j)\text{ except }i} m_{k,j}\left(x_j\right)\right) &\Longrightarrow w_j \\ \psi_{j,i}\left(x_j,x_i\right) &\Longrightarrow K\left(\|x_j-x_i\|\right) \\ m^*_{j,i}\left(x_i\right) &\Longrightarrow \max_{x_j}\Big\{w_j\ K\left(\|x_j-x_i\|\right)\Big\}\quad .\end{aligned}$$

Now, the Max-BP messages is clearly an instance of the Max-Product problem, and for problems with suitable potential functions, our fast method can be applied simply.

3.5.3 Particle Smoothing

The expensive operation in particle smoothing is computing the smoothing weights (equation 2.10):

$$\begin{aligned} \tilde{w}_{n|n}^{(i)} &= \tilde{w}_n^{(i)} \\ \tilde{w}_{k|n}^{(i)} &= \tilde{w}_k^{(i)} \sum_{j=1}^{N} \frac{\tilde{w}_{k+1|n}^{(j)}\ p\left(\tilde{x}_{k+1}^{(j)}|\tilde{x}_k^{(i)}\right)}{\sum_{m=1}^{N} \tilde{w}_k^{(m)}\ p\left(\tilde{x}_{k+1}^{(j)}|\tilde{x}_k^{(m)}\right)} \quad . \end{aligned}$$

This costs $O\left(N^2\right)$, where N is the number of particles per node. Note that particle filtering costs only $O\left(N\right)$; it is only in calculating the smoothing weights that quadratic complexity is incurred.

The smoothing weights can be computed efficiently by applying a fast Sum-Product method twice. First, compute the denominator:

$$d_j = \sum_{m=1}^{N} \tilde{w}_k^{(m)}\ p\left(\tilde{x}_{k+1}^{(j)}|\tilde{x}_k^{(m)}\right)$$

by making the transformations

$$\begin{aligned} \tilde{w}_k^{(m)} &\implies w_m \\ \tilde{x}_k^{(m)} &\implies x_m \\ \tilde{x}_{k+1}^{(j)} &\implies x_j \\ p\left(\tilde{x}_{k+1}^{(j)}|\tilde{x}_k^{(m)}\right) &\implies \psi\left(x_m, x_j\right) \implies K\left(\|x_m - x_j\|\right) \\ d_j &\implies \sum_{m=1}^{N} w_m\ K\left(\|x_m - x_j\|\right) \quad , \end{aligned}$$

where the fourth line is justified by equation 2.8. In this way, computing d_j is a Sum-Product problem which can be solved with one of our fast methods.

Next, compute the outer sum, replacing the denominator by d_j:

$$s_i = \sum_{j=1}^{N} \frac{\tilde{w}_{k+1|n}^{(j)}\ p\left(\tilde{x}_{k+1}^{(j)}|\tilde{x}_k^{(i)}\right)}{d_j} \quad .$$

Make the following transformations:

$$\begin{aligned}
\left(\frac{\tilde{w}_{k+1|n}^{(j)}}{d_j}\right) &\implies w_j \\
\tilde{x}_{k+1}^{(j)} &\implies x_j \\
\tilde{x}_k^{(i)} &\implies x_i \\
p\left(\tilde{x}_{k+1}^{(j)}|\tilde{x}_k^{(i)}\right) &\implies \psi\left(x_j, x_i\right) \implies K\left(\|x_j - x_i\|\right) \\
s_i &\implies \sum_{j=1}^{N} w_j \; K\left(\|x_j - x_i\|\right) \quad .
\end{aligned}$$

Clearly, s_i is also a Sum-Product problem which can be solved with one of the fast methods.

Finally, the smoothing weights are given by

$$\tilde{w}_{k|n}^{(i)} = \tilde{w}_k^{(i)} s_i \quad .$$

In this way, we can perform particle smoothing efficiently for Hidden Markov Models whose transition model $p\left(x_k|x_{k-1}\right)$ can be written as the kernel function $K\left(\|x_k - x_{k-1}\|\right)$.

3.6 Other Applications

Here we mention briefly another setting in which the Sum-Product problem arises.

3.6.1 Gaussian Processes: Fast Conjugate Gradients

We do not review the Gaussian Processes literature here, since good introductory papers are available, including [14, 29]. We use their notation here.

A Gaussian Process is a distribution

$$p\left(\mathbf{t}|\mathbf{C}, \{x_n\}\right) = \frac{1}{Z} \exp\left(-\frac{1}{2}\left(\mathbf{t} - \mu\right)^\top \mathbf{C}^{-1}\left(\mathbf{t} - \mu\right)\right)$$

where $\mathbf{t}$ is a set of random variables $\mathbf{t} = \{t(x_1), t(x_2), \ldots\}$ that are related through the covariance matrix $\mathbf{C}$, which has elements derived from a covariance function: $\mathbf{C}_{m,n} = C\left(x_m, x_n\right)$.

A typical task to be performed with a Gaussian Process model is to predict the value of t_{N+1} given x_{N+1} and a set of observations $\mathcal{D} = \{x_1, \ldots, x_N\}$. The prediction is a Gaussian distribution:

$$p\left(t_{N+1}|x_{N+1}, \mathcal{D}, C\left(x_m, x_n\right)\right) = \frac{1}{Z}\exp\left(-\frac{\left(t_{N+1} - \hat{t}_{N+1}\right)^2}{2\hat{\sigma}^2_{N+1}}\right) \quad .$$

where the mean and variance are given by

$$\begin{aligned} \hat{t}_{N+1} &= k_{N+1}^\top \mathbf{C}_N^{-1} t_N \\ \hat{\sigma}_{N+1} &= \kappa - k_{N+1}^\top \mathbf{C}_N^{-1} k_{N+1}^\top \end{aligned}$$

and $t_N = \{t_1, t_2, \ldots, t_N\}$. The vector k and scalar κ are

$$\begin{aligned} k_{N+1} &= \left(C\left(x_1, x_{N+1}\right),\ C\left(x_2, x_{N+1}\right),\ \ldots,\ C\left(x_N, x_{N+1}\right)\right) \\ \kappa &= C\left(x_{N+1}, x_{N+1}\right) \quad . \end{aligned}$$

Given N observations, the cost of making predictions is $O\left(N^3\right)$ if we take the naive approach of inverting the matrix C_N. This is prohibitively expensive for most datasets of practical interest. Gibbs [15] notes that we need not compute C_N^{-1}; we only need to compute $C_N^{-1}t_N$ once and then $C_N^{-1}k_{N+1}^\top$ for each prediction. By using a conjugate gradient approach, these can be computed approximately in $O\left(N^2\right)$. Given a matrix C_N and a vector u, the method derives a series of estimates of $C_N^{-1}u$. We begin with the estimate $y_1 = \mathbf{0}$, and at each step compute an update:

$$y_{k+1} = y_k + \lambda_k h_k$$

where λ_k and h_k are computed as follows:

$$\begin{aligned} g_1 &= u \\ h_1 &= u \\ \lambda_k &= \frac{g_k^\top g_k}{g_k^\top C_N h_k} \\ g_{k+1} &= g_k - \lambda_k C_N h_k \\ \gamma_k &= \frac{g_{k+1}^\top g_{k+1}}{g_k^\top g_k} \\ h_{k+1} &= g_{k+1} + \gamma_k h_k \quad . \end{aligned}$$

After N iterations we are guaranteed to reach $y_N = C_N^{-1}u$, but good approximations can be had with $K < N$. For a subset of matrices C_N it is also possible to compute an error bound on the estimate y_K; see [15] for details.

Note that the most expensive operation in the above set of equations is the matrix-vector multiplication $C_N h_k$. We introduce our fast methods to this computation by recalling that the elements of C_N are derived from the correlation function $C(x_m, x_n)$. Hence, we can make the following transformation:

$$f = C_N h_k \quad \Longrightarrow \quad f_i = \sum_{j=1}^{N} C\left(x_i, x_j\right) h_k^{(j)} \ , \quad i = 1 \ldots N \ ,$$

where $h_k^{(j)}$ is the j-th element of the vector h_k. In this way, for a subset of correlation functions $C(x_m, x_n)$, the matrix-vector multiply becomes a Sum-Product problem, and our fast methods can be applied.[12]

As it happens, Gibbs [14] suggests that the Gaussian correlation function captures the main properties that we want for regression problems, and notes that it is often used in practice. That is,

$$C\left(x_m, x_n\right) = \theta_1 \exp\left\{ -\frac{1}{2} \sum_{1}^{L} \frac{\left(x_m^{(l)} - x_n^{(l)}\right)^2}{r(l)^2} \right\} + \theta_2 \ ,$$

where $r(l)$ are scale parameters for each of the L dimensions, and θ_1 and θ_2 control the overall noise model properties.

With the extensions to our Dual-tree Sum-Product algorithm discussed in sections 3.3.5, many of the other correlation functions presented by Gibbs (including periodic correlation and input-dependent noise) can be solved.

In summary, making predictions with a Gaussian Process model costs $O\left(N^3\right)$ if performed naively (by matrix inversion), or $O\left(KN^2\right)$ if approximated with the conjugate gradient method. For some correlation functions, our fast methods can be applied, yielding $O\left(KN\right)$ complexity.

[12] Note that the vector h_k can contain negative elements, so our Dual-tree Sum-Product algorithm would need to be extended to the negative weights case, as discussed in section 3.3.5. The Fast Gauss Transform and Improved FGT work with negative weights.

3.7 Empirical Testing of Fast Sum-Product Methods

The section presents the results of testing the Fast Gauss Transform, Improved Fast Gauss Transform, and Dual-Tree methods (using *kd*-tree and Anchors Hierarchy data structures) for fast Sum-Product computation. We examine the performance of these methods with respect to data set size, dimension, allowable error, and data set structure ("clumpiness"), measured in terms of CPU time and memory usage. The results are striking, challenging several claims that are commonly made about these methods.

For simplicity, we consider a subset of the Sum-Product problem. Some of these simplifying assumptions can be easily relaxed for some of the fast methods, but we assume that these will not change the essential properties that we examine. Specifically, we assume:

$$K_i(x_i, y_j) = \exp\left(-\frac{\|x_i - y_j\|^2}{h^2}\right)$$

$$w_i \geq 0$$

$$M = N \quad .$$

That is, we consider the points to live in a vector space, and use a Gaussian kernel, non-negative weights, and equal numbers of sources and targets.

To be useful to a broad audience, we feel that fast Sum-Product algorithms must allow the user to set a guaranteed absolute error level. That is, given an allowable error ϵ, the method must return approximations $\hat{f}_j$ such that

$$\left|\hat{f}_j - f_j\right| \leq \epsilon \quad .$$

Ideally, the methods should work well even with small ϵ, so that one can simply "plug it in and forget about it."

3.7.1 Implementations

The FGT implementation we test was generously provided by Firas Hamze, and is written in C with a MATLAB wrapper. It uses a fairly simple space-subdivision scheme, so

has memory requirements of $O\left(\left(\frac{1}{h}\right)^3\right)$. Adaptive space subdivision or sparse memory allocation could perhaps improve the performance, particularly for small-bandwidth problems.

The implementation of the IFGT that we test was generously provided by Changjiang Yang and Ramani Duraiswami. It is written in C++ with MATLAB bindings.

We use our own implementation of the Dual-Tree algorithm. The Dual-Tree recursion is independent of the space-partitioning data structure. We implemented the *kd*-tree and the Anchors Hierarchy [31]. It is written in C with MATLAB bindings.

3.7.2 Results

All tests were run on our Xeon 2.4 GHz, 1 GB memory, compute servers. We ran the tests within MATLAB; all the fast algorithms are written in C or C++ and have MATLAB bindings. We stopped testing a method once its memory requirements rose above 1 GB in order to avoid swapping. In all cases we repeated the tests with several data sets. In some of the plots the error bars are omitted for clarity. The error bars are typically very small. Most of the plots have log-log scales. The curve labelled "Naive" is the straightforward $O\left(N^2\right)$ summation.

For the IFGT, we set the upper bound on the number of clusters to be $K^* = \sqrt{N}$. In practice, K^* should be set to a constant, but since we are testing over several orders of magnitude this seems more reasonable.

Many of the tests below can be seen as one-dimensional probes in parameter space about the point $N = 10,000$, Gaussian bandwidth $h = 0.01$, dimension $D = 3$, allowable error $\epsilon = 10^{-6}$, clumpiness $C = 1$ (ie, uniform) point distribution, with weights drawn uniformly from $[0, 1]$. In all cases the points are confined to the unit D-cube. We occasionally choose other parameters in order to illustrate a particular point. We use $D = 3$ to allow the FGT to be tested.

Test A: N

Researchers have focused attention on the performance of fast algorithms with respect to N (the number of source and target points). Figure 3.23 shows that it is crucially important to consider other factors, since these strongly influence the empirical performance.

In this test, the scale of the Gaussians is $h = 0.1$, so a large proportion of the space has a significant contribution to the total influence. An important observation in Figure 3.23 is that the dual-tree methods (KDtree and Anchors) are doing about $O\left(N^2\right)$ work. Empirically, they are never faster than Naive for this problem. Indeed, only the FGT is ever faster, and then only for a small range of N. The IFGT appears to be demonstrating better asymptotic performance, but the crossover point (if the trend continues) occurs at about 1.5 hours of compute time.

Another important thing to note in Figure 3.23 is that the dual-tree methods run out of memory before reaching $N = 50,000$ points; this happens after a modest amount of compute time. Also of interest is the fact that the IFGT has *decreasing* memory requirements. We presume this is because the number of clusters increases, so the cluster radii decrease and the error bounds converge toward zero more quickly, meaning that fewer expansion terms are required.

Test B: N

In this test, we repeat test A but use a smaller Gaussian scale parameter, $h = 0.01$. The behaviour of the algorithms is strikingly different. We can no longer run the IFGT, since the number of expansion terms required is more than 10^{10} for $N = 100$. The dual-tree methods perform well, though memory usage is still a concern.

Test C: Dimension D

In this test, we fix $N = 10,000$ and vary the dimension. We set $\epsilon = 10^{-3}$ to allow the IFGT to run in a reasonable amount of time. Surprisingly, the IFGT and Anchors, both of which are supposed to work well in high dimension, do not perform particularly

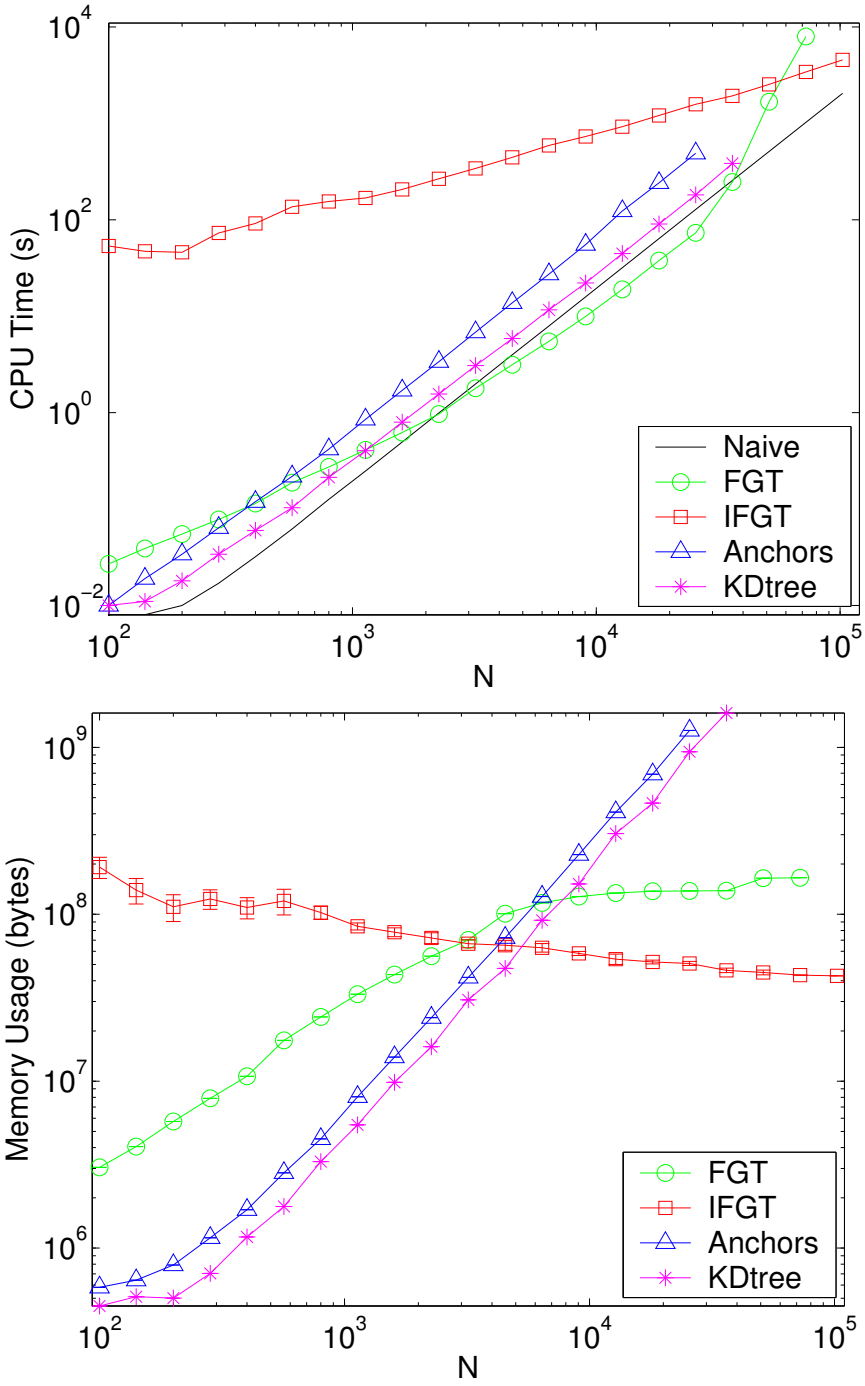


Figure 3.23: Test A: $D = 3$, $h = 0.1$, uniform data, $\epsilon = 10^{-6}$. Top: CPU Time. Bottom: Memory usage.

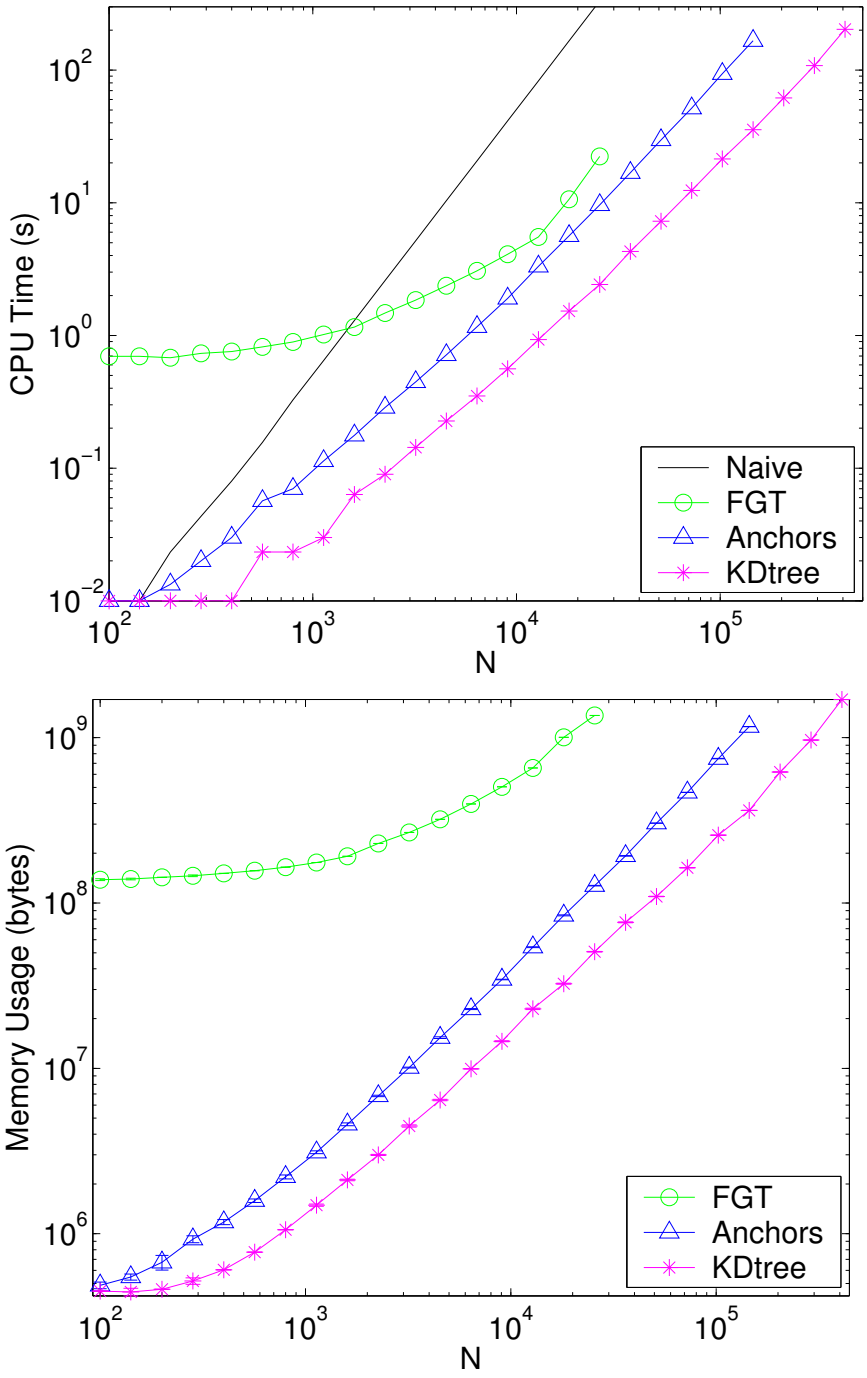


Figure 3.24: Test B: $D = 3$, $h = 0.01$, uniform data, $\epsilon = 10^{-6}$. Top: CPU Time. Bottom: Memory usage.

well. The IFGT's computational requirements become infeasibly large above $D = 2$, while Anchors never does better than KDtree. This continues to be true even when we subtract the time required to build the Anchors Hierarchy.

Also surprising is the fact that the KDtree method *improves* up to dimension 4; it is typically assumed that its performance always decreases with dimension. Further, the *curse of dimensionality* does not seem to rear its head here; KDtree works well up to at least $D = 100$.

Test D: Allowable Error ϵ

In this test, we examine the cost of decreasing ϵ. The dual-tree methods have slowly-increasing costs as the accuracy is increased. The FGT has a more quickly increasing cost, but for this problem it is still competitive at $\epsilon = 10^{-11}$.

We find that the dual-tree methods begin to have problems when $\epsilon < 10^{-11}$; while these methods can give arbitrarily accurate approximations given exact arithmetic, in practice they are prone to cancellation error.[13]

The bottom plot in Figure 3.26 shows the maximum error in the estimates. The dual-tree methods produce results whose maximum errors are almost exactly equal to the error tolerance ϵ. One way of interpreting this is that these methods do as little work as possible to produce an estimate that satisfies the required bounds. The FGT, on the other hand, produces results that have real error well below the requirements. Notice the 'steps'; we believe these occur as the algorithm either adds terms to the series expansion, or chooses to increase the number of boxes that are considered to be within range.

Data Set Clumpiness

Next, we explore the behaviour of the fast methods on data sets drawn from non-uniform distributions.

[13]This is a problem with the algorithm in general, not just our implementation of it; other published versions of the algorithm share this problem.

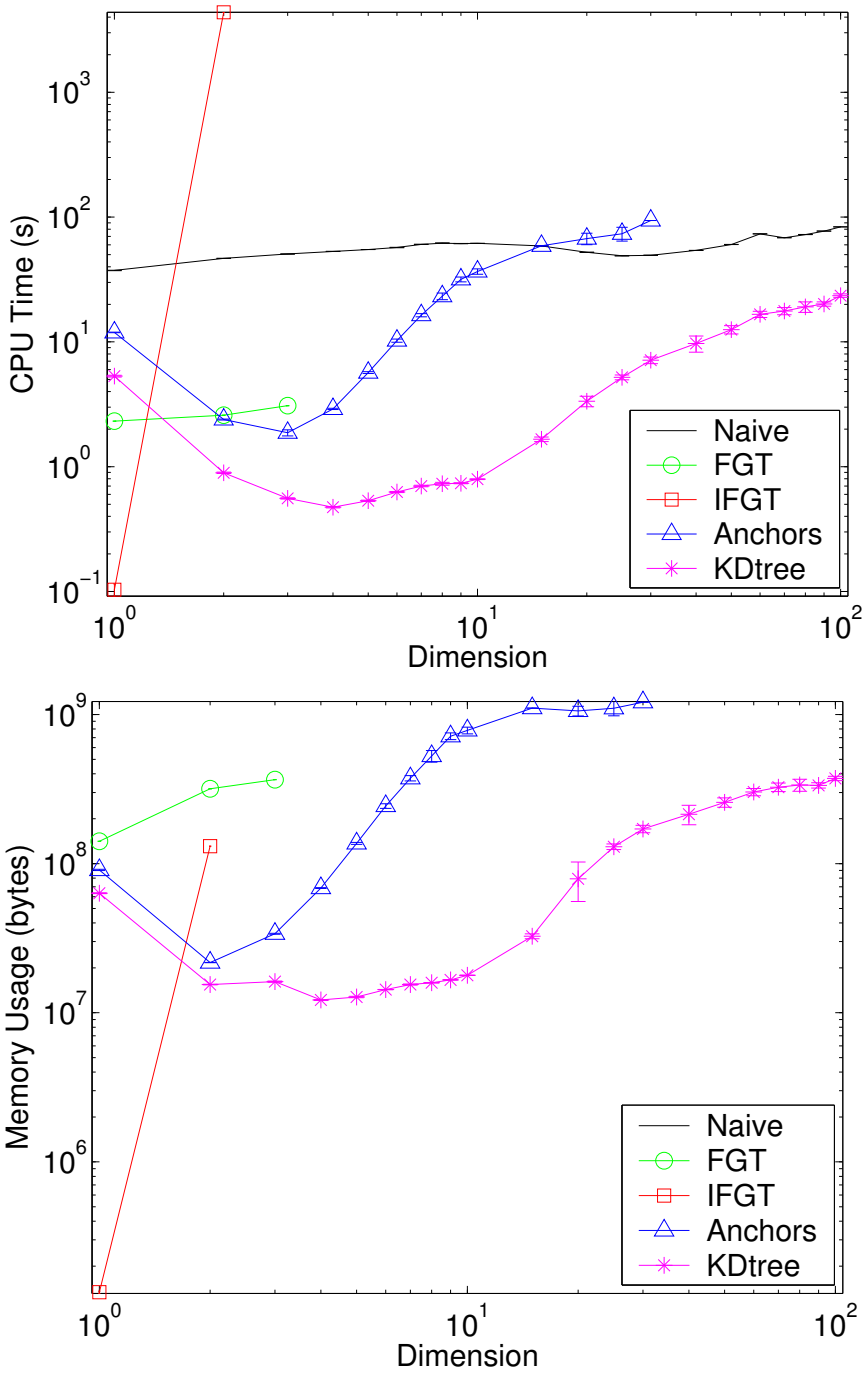


Figure 3.25: Test C: $h = 0.01$, $\epsilon = 10^{-3}$, $N = 10,000$, uniform data.

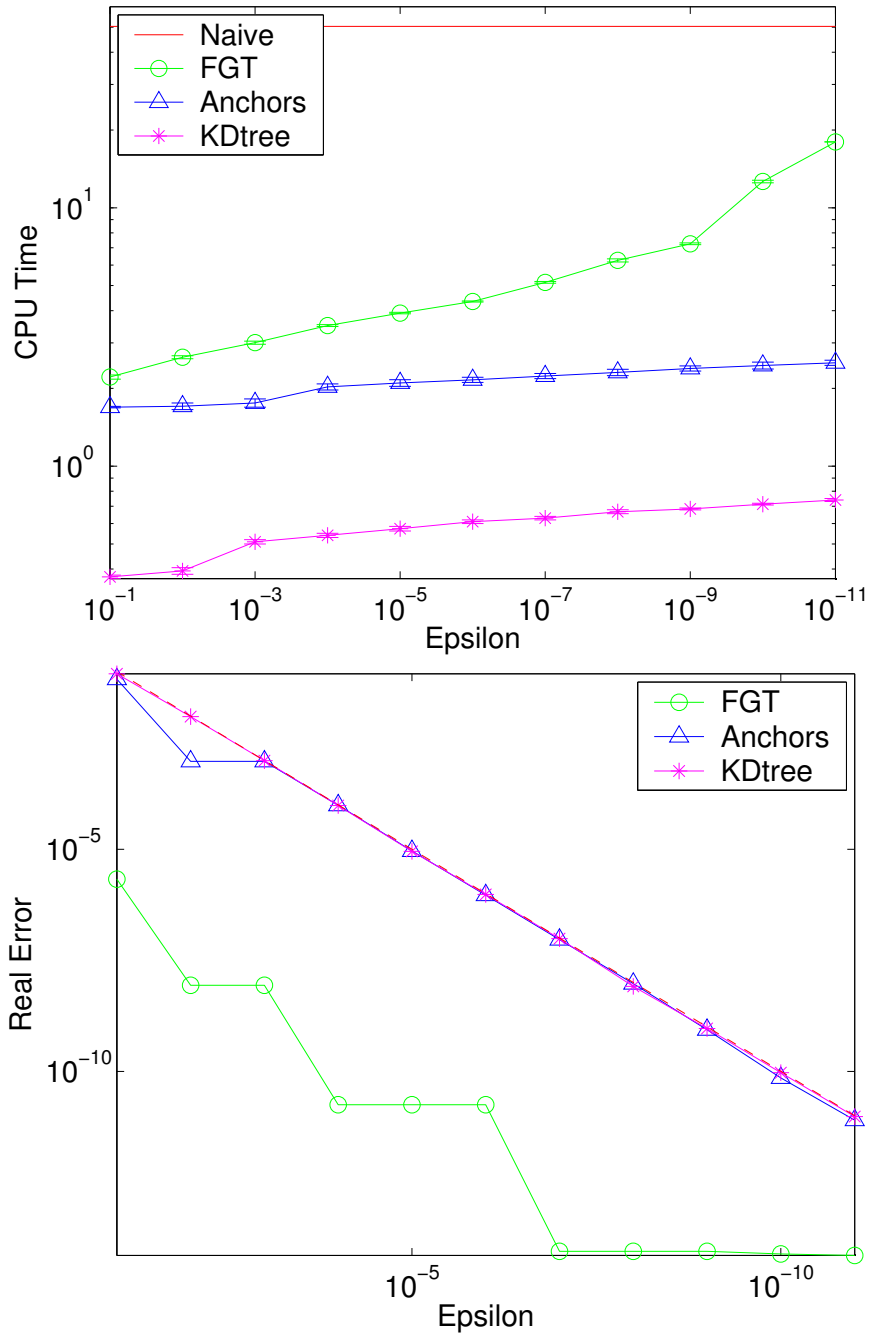


Figure 3.26: Test D: $D = 3$, $h = 0.01$, $N = 10,000$, uniform data. Top: CPU Time. Bottom: Real error.

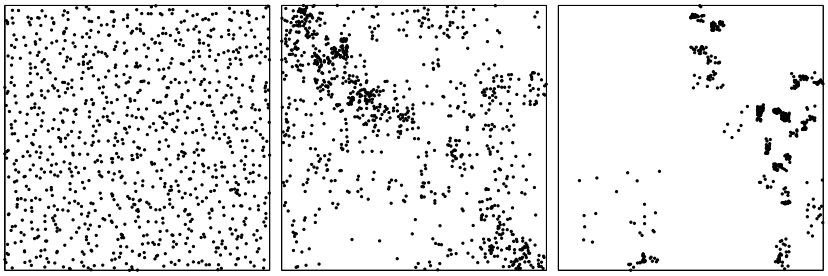


Figure 3.27: Example clumpy data sets. The clumpinesses are $C = 1$ (left), $C = 1.5$ (middle), and $C = 3$ (right). Each data set contains 1000 points.

We use a method for generating clumpy data that draws on the concept of lacunarity. Lacunarity [1, 34] measures the texture or 'difference from uniformity' of a set of points, and is distinct from fractal dimension. It is a scale-dependent quantity that measures the width of the distribution of point density. Lacunarity at a given scale can be measured by covering the set with boxes of that scale; the distribution of point densities in the boxes is measured, and the lacunarity is defined as the second moment of the distribution divided by the first moment squared.

We adapt the notion of the ratio of variance to squared mean. Given a number of samples N and a clumpiness C, our clumpy data generator recursively divides the space into 2^D sub-boxes, and distributes the N samples among the sub-boxes such than

$$\sum_{i=1}^{2^D} N_i = N$$
$$C - 1 = \frac{\text{var}\left(\{N_i\}\right)}{\text{mean}\big(\,\{N_i\}\,\big)^2} \quad .$$

This process continues until N is below some threshold (we use 10). Some example clumpy data sets are shown in Figure 3.27.

Test E: Source Clumpiness

In this test, we draw the source particles X from a clumpy distribution, while the targets are drawn from a uniform distribution.

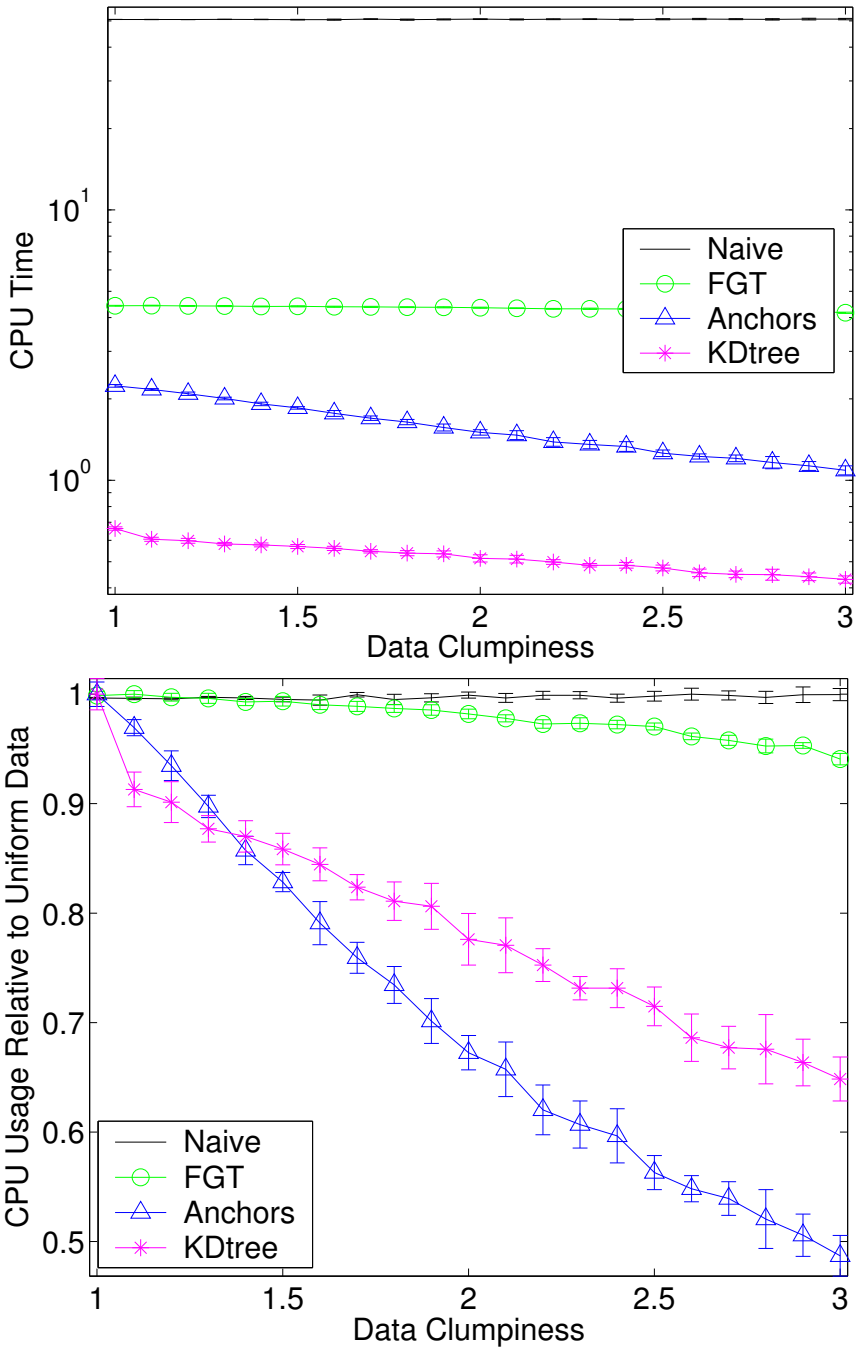


Figure 3.28: Test E: $D = 3$, $h = 0.01$, $\epsilon = 10^{-6}$, $N = 10,000$, clumpy X, uniform Y. Top: absolute CPU time. Bottom: relative CPU time as clumpiness increases.

Figure 3.28 shows the relative CPU time as clumpiness increases. The dual-tree methods show significant improvements as the source points become clumpy. Anchors improves more than KDtree, although KDtree is still faster in absolute terms. The FGT shows minor improvement as clumpiness increases.

Test F: Source and Target Clumpiness

In this test, we draw both the sources and targets from clumpy distributions. The dual-tree methods show even more marked improvement as clumpiness increases. The FGT also shows greater improvement than in the previous test.

Test G: Clumpy Data, Dimension D

In this test, we test the performance with dimension, given clumpy data sets. The results are not very different than the uniform case (Test C). This is surprising, since neither Anchors nor IFGT does particularly well, even given clumpy data.

3.7.3 Conclusions

We presented a comparison between the most widely used fast methods for the Sum-Product problem. In our comparison, we varied not only the number of points N, but also the structure in the data, the required precision and the dimension of the state space. The results indicate that the fast methods can only work well when there is structure in the kernel matrix. They also indicate that dual tree methods are preferable in high dimensions. Surprisingly, the results show that the KDtree works better than the Anchors Hierarchy in our particular experiments. This seems to contradict common beliefs about these methods. Yet, there is a lack of methodological comparisons between these methods in the literature. This makes it clear that further investigation is warranted.

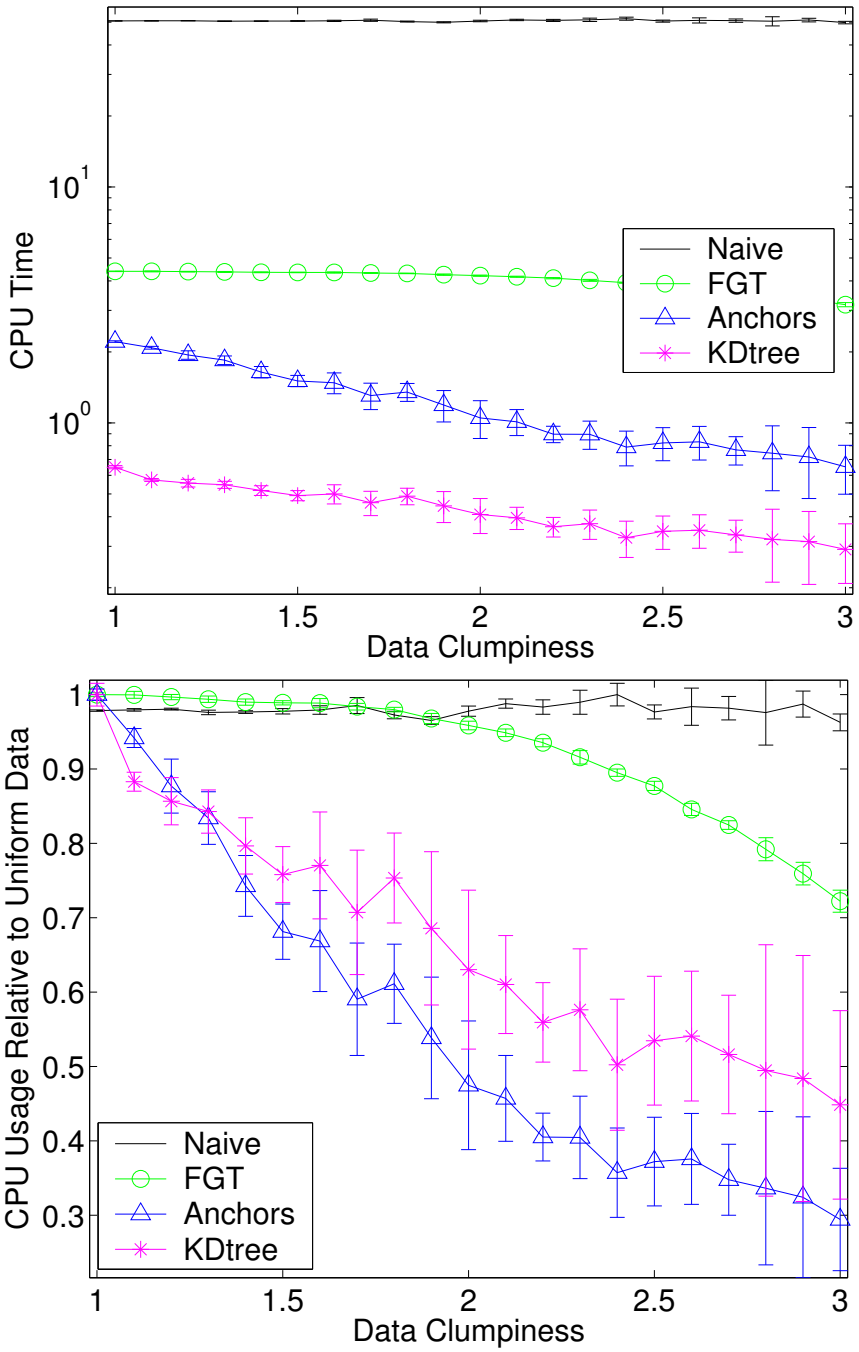


Figure 3.29: Test F results: $D = 3$, $h = 0.01$, $\epsilon = 10^{-6}$, $N = 10,000$, clumpy X, clumpy Y. Top: absolute CPU time. Bottom: relative CPU time as clumpiness increases.

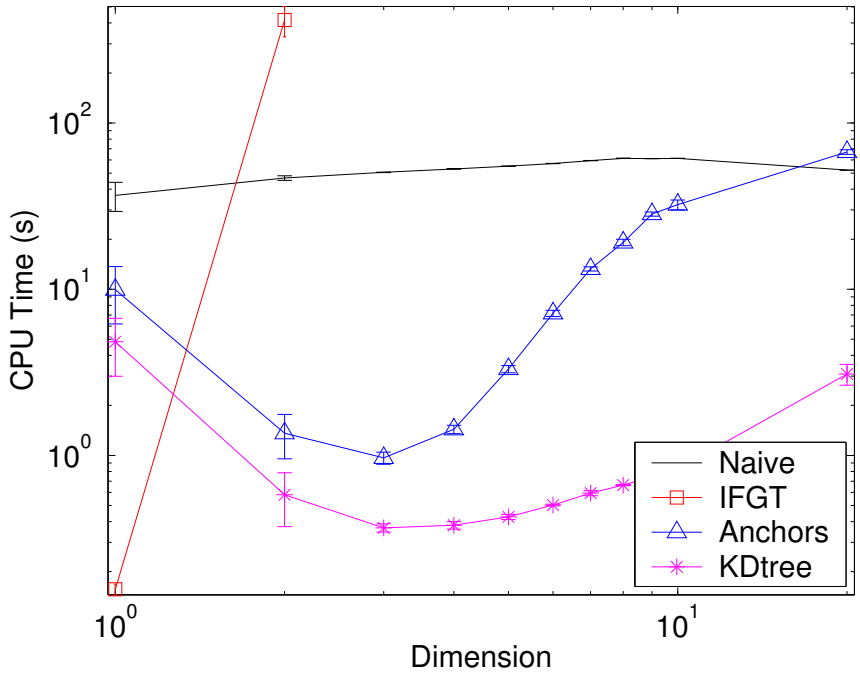


Figure 3.30: Test G results: $h = 0.01$, $\epsilon = 10^{-3}$, $N = 10,000$, clumpy X, clumpy Y.

3.8 Empirical Testing of Fast Max-Product Methods

The tests presented in this section were performed by Mike Klaas as part of a paper that we co-authored along with Nando de Freitas. I implemented the Dual-Tree Max-Product algorithm, while Mike implemented the Distance Transform (and also tracked down a few bugs in the Dual-Tree). The text is largely due to Mike. I thank Mike for allowing me to include the material here.

3.8.1 Performance in N

Here, we focus on performance in synthetic and real-world settings as N grows; comparisons are made both in settings where the Distance Transform is applicable and where it is not. We present results in terms of both CPU time and number of distance computations (kernel evaluations) performed. This is important as in some applications the kernel evaluation is extremely expensive and thus dominates the runtime of

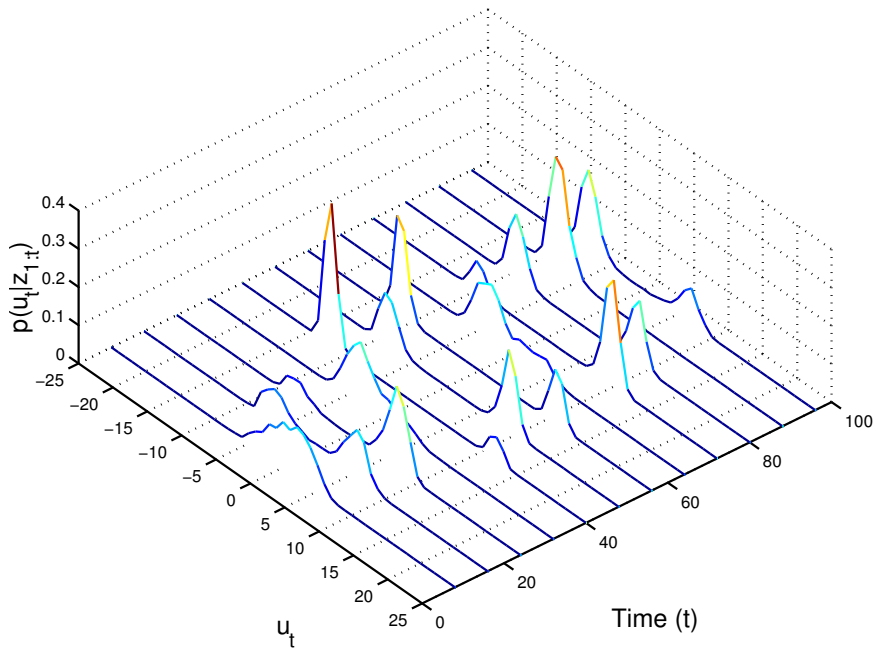


Figure 3.31: Filtered distribution $p(\mathbf{u}_t|\mathbf{z}_{1:t})$

the algorithm.

Multi-modal non-linear time series

We consider the following standard reference model

$$u_{t+1} = \frac{1}{2}u_t + 25\frac{u_t}{1+u_t^2} + 8\cos 1.2t + v_{t+1} \tag{3.5}$$

$$z_{t+1} = \frac{u_{t+1}^2}{20} + w_{t+1} \tag{3.6}$$

Where $v_t \sim \mathcal{N}(0, \sigma_v)$ and $w_t \sim \mathcal{N}(0, \sigma_w)$. The filtered distribution is bimodal and highly non-linear (see figure 3.31), meaning the standard particle filter produces significant error even with a high particle count.

After running a standard SIR particle filter, we implemented the MAP sequence estimation described in [16]. Figure 3.32 demonstrates the accuracy gained by calculating the MAP solution. We chose a one-dimensional setting so that the Dual-Tree algorithm could be directly compared against the Distance Transform. (We used a modified version of the Distance Transform that allows it to be applied to one-dimensional problems

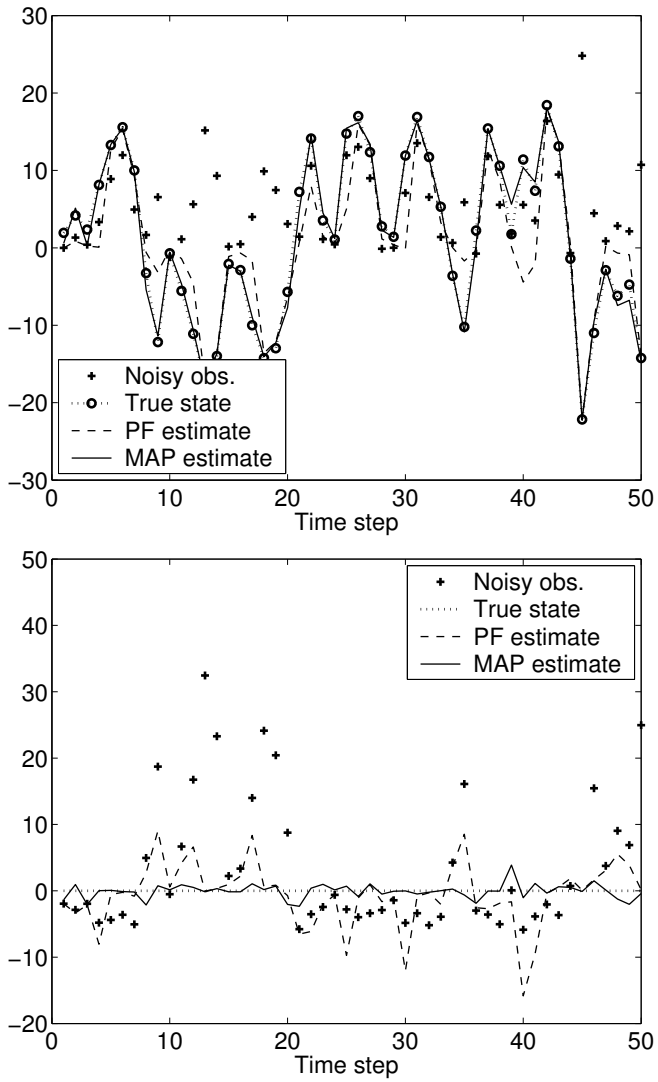


Figure 3.32: Top: Particle filter and MAP estimates of the latent state in the 1-D time series experiment. Bottom: the same plot, with the true state subtracted from each line. Mean error for the particle filter was 4.05, while the MAP solution achieved a mean error of 1.74.

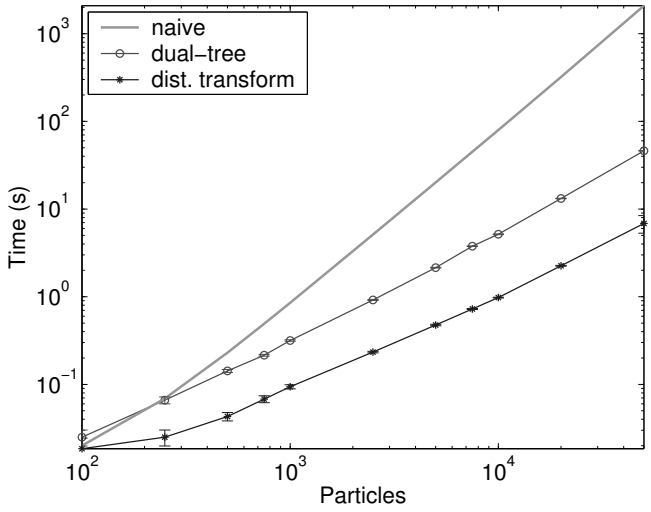


Figure 3.33: 1-D time series results. The dual-tree algorithm became more efficient than naïve computation after approximately 70ms of compute time. Both Dual-Tree and Distance Transform methods show similar asymptotic growth, although the constants in the Distance Transform are approximately three times smaller.

that do not lie on a regular grid.) Figures 3.33 and 3.34 summarize the results. It is clear that the Distance Transform is superior in this setting, although the Dual-Tree algorithm is still quite usable, being several orders of magnitude faster than the naïve method.

Beat-tracking

We tested the fast methods for the beat-tracking problem described in chapter 4. The algorithm used was the forward pass particle filter followed by Max-Belief Propagation. Since the state space of this model is a three-dimensional Monte Carlo grid, the Distance Transform cannot be used.

Figures 3.35 and 3.36 summarize the results. Using the fast method allows more particles to be used, which results in a better solution. This can be quantified by looking at the probability of the MAP solution, which is computed during Max-Belief Propagation. The probability of the MAP sequence with 50,000 particles was $p = 0.87$, while using 1000 particles resulted in a MAP sequence of probability $p = 0.53$.

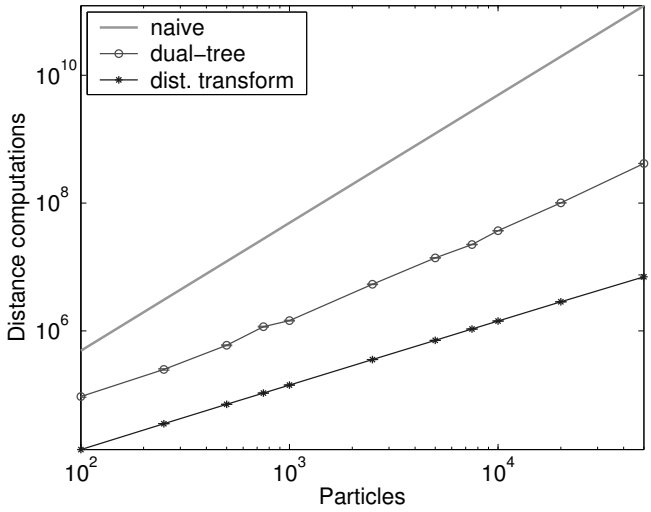


Figure 3.34: 1-D time series results: distance computations v. particle count.

3.8.2 The effect of other parameters

Distribution and dimension

To examine the effects of other parameters on the behaviour of the dual-tree algorithm, we ran several experiments varying dimensionality, distribution, and spatial index while keeping N constant. We used two spatial indices: kd-trees and metric trees (built using the Anchors hierarchy). We generated synthetic data by drawing points from a mixture of Gaussians distributed evenly in the space. Figure 3.37 shows a typical clustered data distribution. In all runs the number of particles was held constant at $N = 20,000$, and the dimension was varied to a maximum of $D = 40$. Figures 3.38 and 3.39 show the results for CPU time and distance computations, respectively. In these figures, the solid line represents a uniform distribution of particles, the dashed line represents a 4-cluster distribution, the dash-dot line has 20 clusters, and the dotted line has 100. We expect methods based on spatial indexing to fare poorly given uniform data since the average distance between points quickly becomes a very peaked distribution in high dimension, reducing the value of distance as a measure of contrast. The results are consistent with this expectation: for uniform data, both the kd-tree and Anchors methods exceeded $O\left(N^2\right)$ distance computations by $D = 12$.

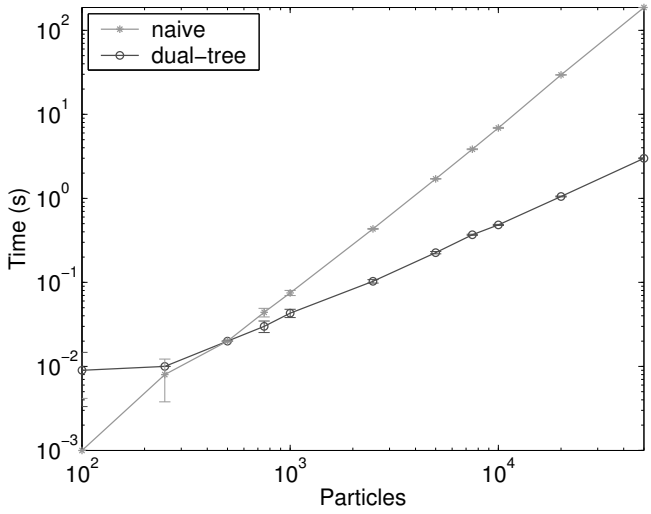


Figure 3.35: Beat tracking results: time v. particle count. The dual-tree method becomes more efficient at $t = 10$ms, and thereafter dominates the naïve method.

Perhaps more surprising is that the kd-tree method consistently outperformed the Anchors method on uniform data even up to $D = 40$. The depth of a balanced binary kd-tree of $20,000$ particles and leaf size 25 is ten, so for $D > 10$ there are many dimensions that are not split even a single time!

Of more practical interest are the results for clustered data. It is clear that the distribution vastly affects the runtime of Dual-Tree algorithms; at $D = 20$, solving the Max-Product with the Anchors method was six times faster on clustered data compared to uniform. We expect this effect to be even greater on real data sets, as the clustering should exist on many scales rather than simply on the top level as is the case with our synthetic data. It is also interesting to note the different effect that clustering had on kd-trees compared to metric trees. For the Anchors method, clustering always improved the runtime, albeit by a marginal amount in low dimensions. For kd-trees, clustering *hurt* performance in low dimensions, only providing gains after about $D = 8$. The difference in the two methods is shown in figure 3.40.

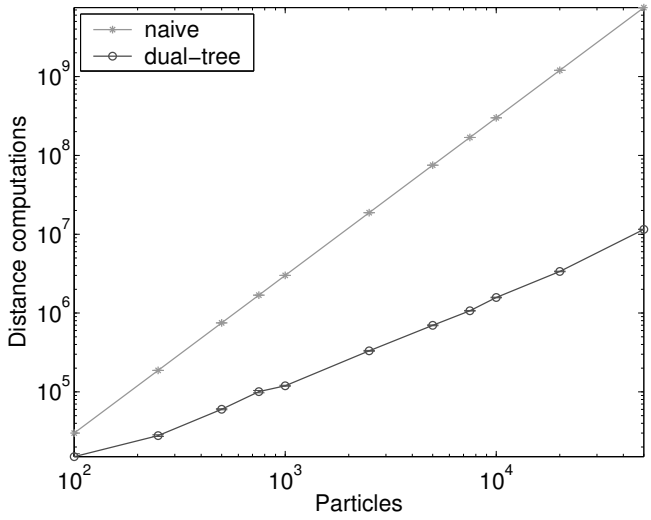


Figure 3.36: Beat tracking results: distance computations vs. particle count.

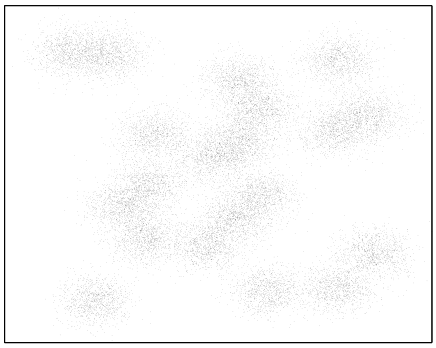


Figure 3.37: Synthetic data set with $c = 20$ clusters.

Effect of kernel width

To measure the effect of different kernels, we test both methods on a 1-D uniform distribution of $200,000$ points, and use a Gaussian kernel with bandwidth (σ) varying over several orders of magnitude. The number of the distance computations required was reduced by an order of magnitude over this range (see Figure 3.41). It seems that with very wide kernels, the values that the kernel function takes on are in a smaller range, so the weights become more important.

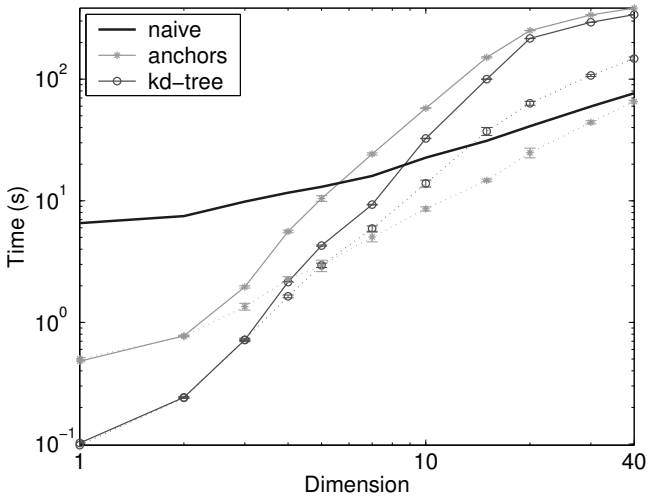


Figure 3.38: Time vs. dimensionality. For clarity, only the uniform distribution and one level of clustered data are shown. This experiment demonstrates that some structure is required to accelerate the Max-Product problem in high dimensions.

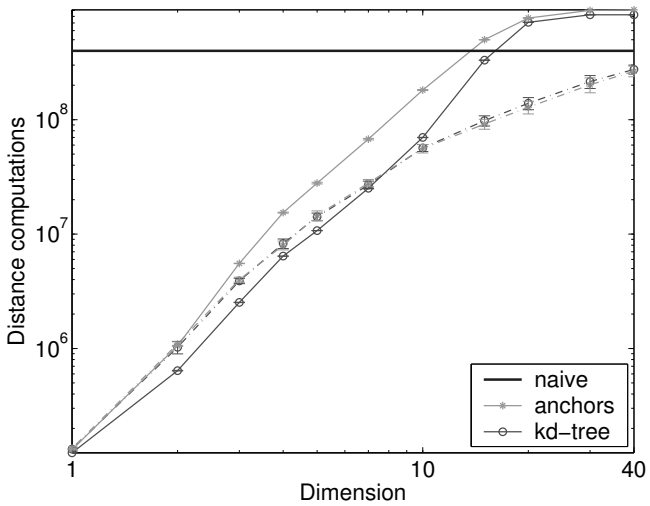


Figure 3.39: Distance computations vs. dimensionality. The level of clustering shown is less than in Figure 3.38. For kd-trees, clustering hurts performance when $D \leq 8$.

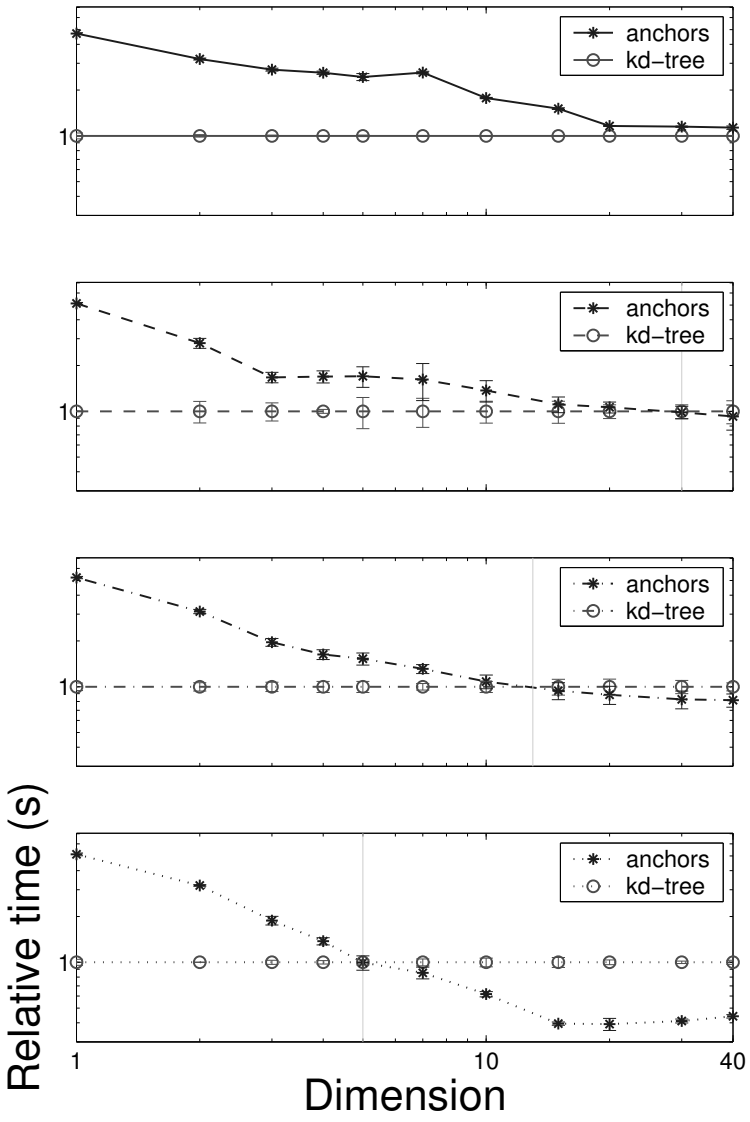


Figure 3.40: Time vs. dimensionality; relative to kd-tree = 1. Top: uniform data. Second: four clusters. Third: 20 clusters. Bottom: 100 clusters. Metric trees are better able to properly index clusters: the more clustered the data, the smaller dimensionality required for the Anchors method to outperform kd-trees ($D = 30$ for somewhat-clustered data, $D = 15$ for moderately-clustered data, and $D = 6$ for significantly-clustered data).

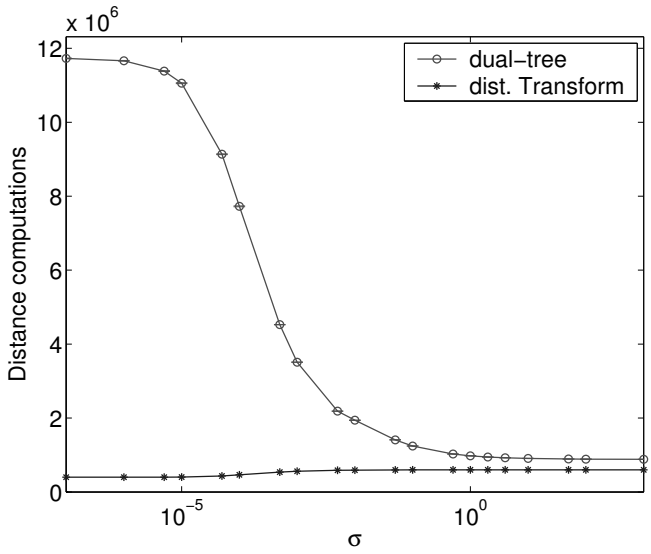


Figure 3.41: Effect of kernel bandwidth: distance computations vs. bandwidth of a Gaussian kernel.

3.8.3 Conclusion

We examine some of the variables that affect the performance of Dual-Tree recursion, such as dimensionality, data distribution, spatial index, and kernel. These parameters have dramatic effects on the runtime of the algorithm, and our results suggest that more exploration is warranted into these effects—behaviour as N varies is only a small part of the story.

Chapter 4

Application: Beat Tracking

4.1 Introduction

The material in this chapter borrows heavily from the paper written by myself and Nando de Freitas for the Neural Information Processing Systems 2004 (NIPS*17) conference [28]. The comments from our anonymous reviewers were very helpful.

In this chapter, we present an application of graphical models, and our fast inference methods, to a popular problem in music processing: beat tracking.

Human listeners have little difficulty tracking the beat of most popular music; tapping a foot in time to music is an example of beat tracking. Dixon describes *beats* as follows: "much music has as its rhythmic basis a series of pulses, spaced approximately equally in time, relative to which the timing of all musical events can be described. This phenomenon is called the *beat*, and the individual pulses are also called beats" [7]. Given a piece of recorded music (an MP3 file, for example), we wish to produce a set of beats that correspond to the beats perceived by human listeners. For a listener tapping a foot or clapping in time to music, the beats are the taps or claps.

- *beat*

The set of beats of a song can be characterized by the trajectories through time of the *tempo* and *phase offset*. Tempo is typically measured in beats per minute (BPM), and describes the frequency of beats. The phase offset determines the time offset of the beat. When tapping a foot in time to music, tempo is the rate of foot tapping and phase offset is the time at which the tap occurs. Phase zero is called the 'down beat' and phase π is the 'up beat'; 'down' and 'up' refer to the position of the foot when tapping in time to the beat.

- *tempo*
- *phase offset*

The beat tracking problem is somewhat ill-posed. Music is often ambiguous; different human listeners can perceive the beat differently. There are often several beat tracks that could be considered reasonable or correct. For example, when listening to music with a fast tempo, many human listeners will tap every second beat (2/2 time rather than 4/4 time).

We see the beat tracking problem as not only an interesting problem in its own right, but as one aspect of the larger problem of machine analysis of music. Given beat tracks for a number of songs, we could extract descriptions of the rhythm and use these features for clustering or searching in music collections. We could also use the rhythm information to do structural analysis of songs - for example, to find repeating sections. In addition, we note that beat tracking produces a description of the time scale of a song; knowledge of the tempo of a song would be one way to achieve time-invariance in a symbolic description. Finally, we note that beat tracking tells us where the important parts of a song are; the beats (and major divisions of the beats) are good sampling points for other music-analysis problems such as note detection. Conversely, analyses of other aspects of music could help to choose between multiple interpretations of the beat.

4.2 Related Work

Many researchers have investigated the beat tracking problem; we present only a brief overview here. Scheirer [35] presents a system, based on psychoacoustical observations, in which a bank of resonators compete to explain the processed audio input. The system is tested on a difficult set of examples, and considerable success is reported. The most common problem is a lack of global consistency in the results - the system switches between locally optimal solutions.

Goto [17] has described several systems for beat tracking. He takes a very pragmatic view of the problem, and introduces a number of assumptions that allow good results in a limited domain - pop music in 4/4 time with roughly constant tempo, where bass or snare drums keep the beat according to drum patterns known *a priori*,

or where chord changes occur at particular times within the measure.

Cemgil and Kappen [6] phrase the beat tracking problem in probabilistic terms, and we adapt their model as our local observation model. They use MIDI-like (event-based) input rather than audio, so the results are not easily comparable to our system.

4.3 Graphical Model

In formulating our model for beat tracking, we assume that the tempo is nearly constant over short periods of time, and usually varies smoothly. We expect the phase to be continuous. This allows us to use the simple graphical model shown in Figure 4.1. We break the song into a set of *frames* of two seconds; each frame is composed of a pair of nodes in the graphical model. We expect the tempo to be constant within each frame, and the tempo and phase offset parameters to vary smoothly between frames.

• *frame*

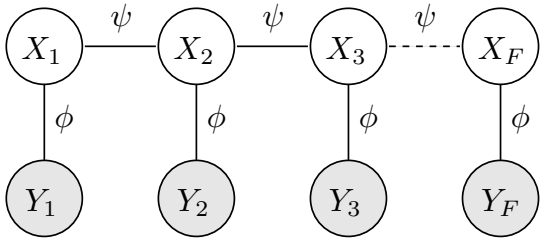


Figure 4.1: Our graphical model for beat tracking. The *hidden state* X is composed of the state variables tempo and phase offset. The *observations* Y are the features extracted by our audio signal processing. The potential function ϕ describes the compatibility of the observations with the state, while the potential function ψ describes the smoothness between neighbouring states. Each pair of X and Y nodes comprises one frame.

In this undirected probabilistic graphical model, the potential function $\phi(Y_i, X_i)$ describes the compatibility of the state variable $X = \{T, P\}$, composed of tempo T and phase offset P, with the local observations Y. The potential function $\psi(X_i, X_{i+1})$ describes the smoothness constraints between neighbouring frames. The observations

Y comes from processing the audio signal, which is described in Section 4.6. The ϕ function comes from domain knowledge and is described in Section 4.5. This model allows us to trade off local fit and global smoothness in a principled manner. Ambiguity in one section of the song can be resolved by using contextual information from more certain regions. By using an undirected model, we allow contextual information to flow both forward and backward in time.

In such models, belief propagation (BP) allows us to compute the marginal probabilities of the state variables in each frame. Alternatively, maximum belief propagation (Max-BP) allows a joint maximum *a posteriori* (MAP) set of state variables to be determined. That is, given a song, we generate the observations Y_i, $i = 1 \dots F$, (where F is the number of frames in the song) and seek a set of states X_i that maximize the joint product

$$P(X, Y) = \frac{1}{Z} \prod_{i=1}^{F} \phi(Y_i, X_i) \prod_{i=1}^{F-1} \psi(X_i, X_{i+1}) \quad .$$

4.4 Smoothness Model

The potential function $\psi(X_i, X_{i+1})$ expresses our belief that the tempo typically varies smoothly and phase is typically continuous. The ψ we use is the product of tempo and phase smoothness components:

$$\psi = \psi_T \; \psi_P \quad .$$

We expect that relative changes in tempo are important, so for the tempo smoothness component ψ_T, we place a Gaussian transition model on the log of tempo change:

$$\log(T_{i+1}) = \log(T_i) + \mathcal{N}\left(0, \; {\sigma_T}^2\right)$$

so we can write the tempo smoothness term as the Gaussian

$$\psi_T(T_i, T_{i+1}) = \mathcal{N}\left(\log(T_{i+1}) - \log(T_i), \; {\sigma_T}^2\right) \quad .$$

We set the variance $\sigma_T = 0.1$.

The phase smoothness component ψ_P allows us to enforce continuity of phase between frames. We want beats to be spaced approximately evenly in time, even

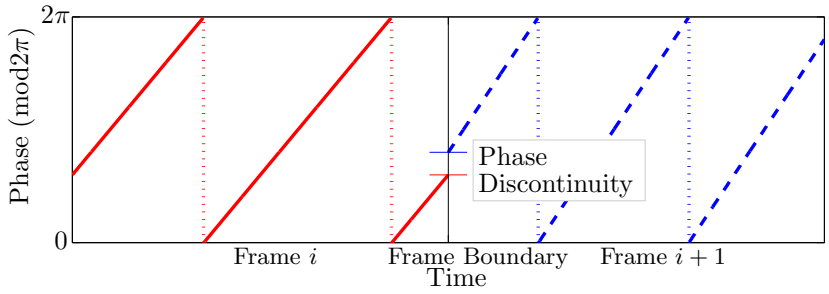


Figure 4.2: A demonstration of the phase continuity constraint. The plot shows two frames with different tempos and phase offsets. There is a linear relationship between phase and time in each frame. Beats occur at points where phase (mod 2π) is zero. We would like the phase to be continuous at the boundary between frames.

across frame boundaries; see Figure 4.2. Given tempo and phase offsets values in a frame, there is a linear relationship of phase with time:

$$\theta\left(t \mid T, P\right) = 2\pi T t - P \quad .$$

We want the phases of neighbouring frames to be continuous at the frame boundary. If the boundary separating frames i and $i+1$ is at time t_b, we want

$$\theta(t_b \mid T_i, P_i) \;\simeq\; \theta(t_b \mid T_{i+1}, P_{i+1}) \quad .$$

Since phase values are cyclic, we project the phase values onto the unit circle and place a Gaussian on the distance between points on the circle:

$$\begin{aligned} \theta_i &= \theta(t_b \mid T_i, P_i) \\ \theta_{i+1} &= \theta(t_b \mid T_{i+1}, P_{i+1}) \\ \psi_P &= \mathcal{N}\Big(\left(\cos\theta_i - \cos\theta_{i+1} \;,\; \sin\theta_i - \sin\theta_{i+1}\right),\; {\sigma_P}^2 \Big) \quad , \end{aligned}$$

where $\mathcal{N}$ is a two-dimensional Gaussian. We set the variance $\sigma_P = 0.1\pi$.

4.5 Local Probability Model

In this section, we describe the derivation of our local potential function (also known as the observation model) $\phi\left(Y_i, X_i\right)$.

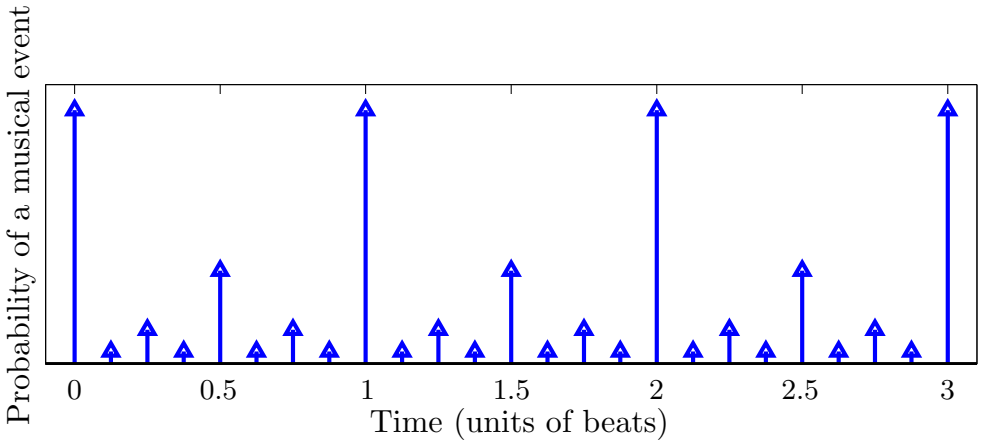


Figure 4.3: Cemgil and Kappen probability of a note occurring at different times, in units of beats, for $D = 8$ and $\lambda = 1$. The function is periodic and infinite. Musical events (notes) are most likely to occur on the beat (that is, at an integer number of beats). The probability of a note occurring at finer and finer divisions of the beat drops away exponentially.

Our model is an adaptation of the work of [6], which was developed for use with MIDI input. Their model is designed so that it "prefers simpler [musical] notations". The beat is divided into a fixed number of bins (some power of two), D, and each note is assigned to the nearest bin. The probability of observing a note at a coarse subdivision of the beat is greater than at a finer subdivision. More precisely, a note that is quantized to the bin at beat number k has probability

$$p(k) \;\propto\; \exp\Big(-\lambda\, d\,(k) \Big)$$

where $d(k)$ is the number of digits in the binary representation of the number (k mod 1); that is, the number of bits required to represent the fractional portion of the number k, and λ is a parameter. See Figure 4.3.

Since we use recorded music rather than MIDI, we must perform signal processing to extract features from the raw data. This process produces a signal that has considerably more uncertainty than the discrete events of MIDI data, so we adjust the model. We add the constraint that features should be observed near *some* quantization point,

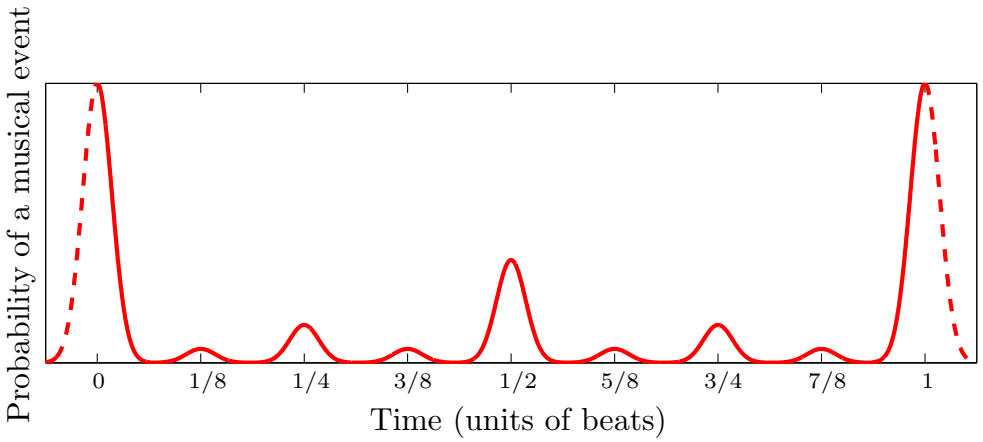


Figure 4.4: One period of our template function $b(t)$, which gives the expected distribution of notes within a beat. Given tempo and phase offset values, we stretch and shift this function to get the expected distribution of notes in time.

which we express by centering a Gaussian around each of the quantization points. The variance of this Gaussian, σ_Q^2 is in units of beats, so we arrive at the periodic template function $b(t)$, shown in Figure 4.4. We have set the number of bins to 8, λ to one, and $\sigma_Q = \frac{1}{40}$.

The template function $b(t)$ expresses our belief about the distribution of musical events within the beat. By shifting and scaling $b(t)$, we can describe the expected distribution of notes in time for different tempos and phase offsets:

$$b\left(t \mid T, P\right) = b\left(Tt - \frac{P}{2\pi}\right) \quad .$$

Our signal processing (described below) yields a discrete set of events that are meant to correspond to musical events. In each frame, we produce a set of discrete events $Y = \{t_i, E_i\}, \;\; i = 1 \ldots M$, where t_i is the time at which the event occurs and E_i is the 'strength' or 'energy' of the event. Given state variables $X = \{T, P\}$, the expected distribution of events in time is $b(t \mid T, P)$. We can treat this as a multinomial distribution in the continuous limit (as bin size approaches zero), from which a set of

events Y are drawn. This leads to the potential function

$$\phi\left(Y, X\right) = \phi\Big(\left\{t, E\right\}, \left\{T, P\right\}\Big) = \prod_{i=1}^{M} \left[\ b\left(\ t_i \mid T, P\ \right)\ \right]^{E_i} \quad .$$

4.6 Audio Feature Extraction

Our signal processing stage is meant to extract features from the raw audio signal that approximate musical events (drum beats, piano notes, guitar strums, etc.). As discussed above, we produce a set of events composed of time and 'strength' values, where the strength describes our certainty that an event occurred. We assume that musical events are characterized by brief, rapid increases in energy in the audio signal. This is certainly the case for percussive instruments such as drums and piano, and will often be the case for string and woodwind instruments and for voices. This assumption breaks for sounds that fade in smoothly rather than 'spikily'.

The audio signal processing is shown in Figure 4.5. We begin by taking the short-time Fourier transform (STFT) of the signal: we slide a 50 millisecond Hann window over the signal in steps of 10 milliseconds, take the Fourier transform, and extract the energy spectrum. Following a suggestion by [35], we pass the energy spectrum through a bank of five filters that sum the energy in different portions of the spectrum. We take the logarithm of the summed energies to get a 'loudness' signal. Next, we convolve each of the five resulting loudness signals with a filter that detects positive-going edges. This can be considered a 'loudness gain' signal. Finally, we find positive maxima of the loudness gain signals within 50 ms neighbourhoods. The result is a set of points, one per 50 ms for each of the five frequency bands, that describe the energy gain in the band, with emphasis on the maxima. These are the features Y that we use in our local probability model ϕ.

4.7 Inference

To find a maximum *a posteriori* (MAP) set of state variables that best explain a set of observations, we need to optimize a $2F$-dimensional, continuous, non-linear, non-

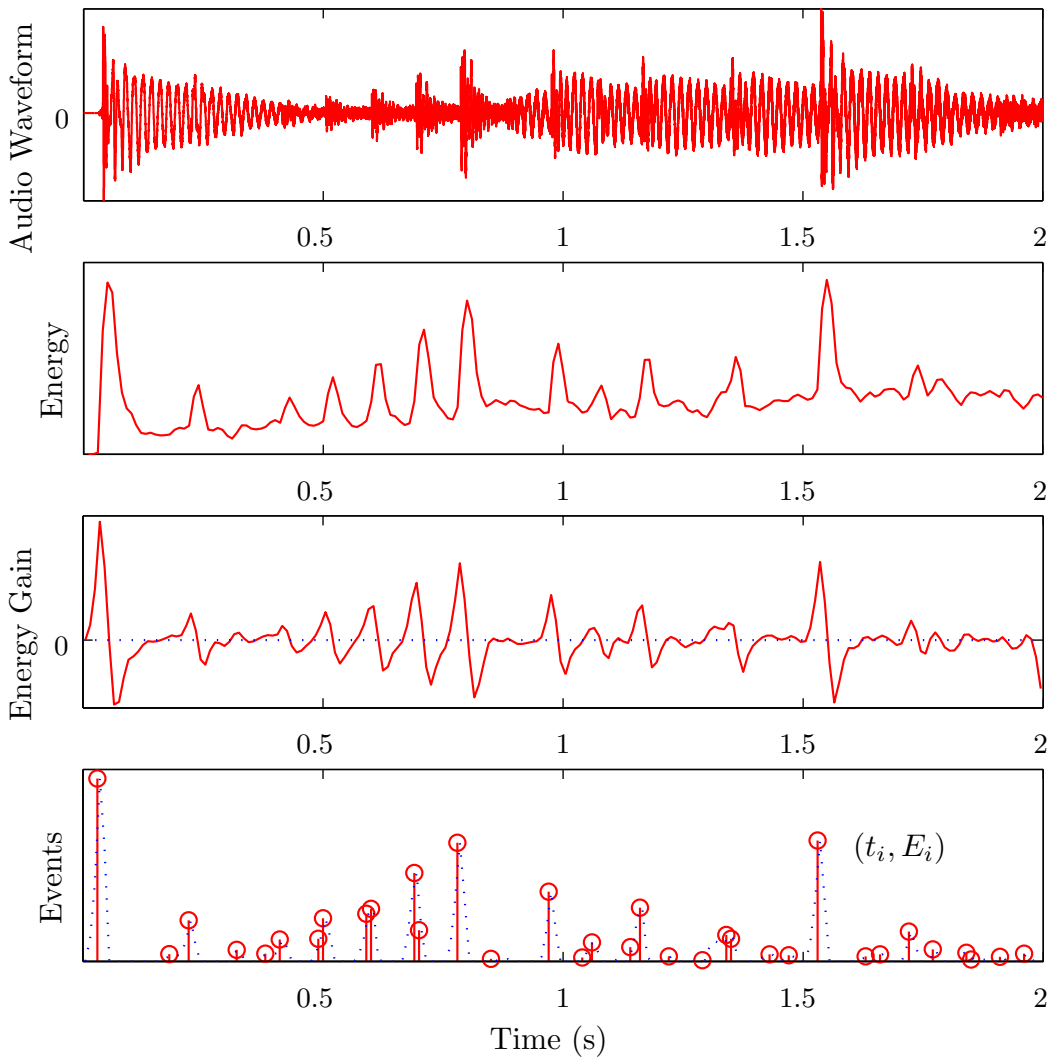


Figure 4.5: Our audio signal processing (feature extraction). We are given the raw audio waveform (top plot). We take the short-time Fourier transform and pass the spectra through a set of five bandpass filters. The log of energy from the third filters is shown (second plot). Next, we convolve the energy signal with a soft-derivative filter to find the increase in energy (third plot). Finally, we find positive maxima in the energy gain signal (bottom plot). These discrete peaks are the features (or observations) Y. The dotted line in the bottom plot is the positive portion of the energy gain signal. In summary, the signal processing detects the locations of rapid increases in energy in various frequency bands.

Gaussian function that has many local extrema. F is the number of frames in the song, so is on the order of the length of the song in seconds - typically in the hundreds. This is clearly difficult. We present two approximation strategies. In the first strategy, we convert the continuous state space into a uniform discrete grid and run discrete maximum-belief propagation (Max-BP). In the second strategy, we run a particle filter in the forward direction, then use the particles as 'grid' points and run Max-BP as per [16].

Since the landscape we are optimizing has many local maxima, we must use a fine discretization grid (for the first strategy) or a large number of particles (for the second strategy). Max-BP costs $O\left(N^2\right)$ when performed naively, where N is the number of discretized states (or particles) per frame. We use the methods of Chapter 3 to reduce the cost of the computation.

For the results presented here, we discretize the state space into $N_T = 90$ tempo values and $N_P = 50$ phase offset values for the fixed-grid method. We distribute the tempo values uniformly on a log scale between 40 and 150 BPM, and distribute the phase offsets on a uniform grid. For the particle filter version, we use $N_T \times N_P = 4500$ particles. With these values, our *C* and *Matlab* implementation runs at faster than real time (the duration of the song) on a standard desktop computer.

4.8 Results

A standard corpus of labelled ground truth data for the beat-tracking problem does not exist. To evaluate our algorithm, we manually labelled a relatively small number of songs by listening to each song and marking each perceived beat. We sought out examples that we thought would be difficult, and we attempted to avoid the methods of [27]. Ideally, we would have several human listeners label each song, since this would help to capture the ambiguity inherent in the problem. However, this would be quite time-consuming.

One can imagine several methods for speeding up the process of generating ground truth labellings and of cleaning up the noisy results generated by humans. For example,

a human labelling of a short segment of the song could be automatically extrapolated to the remainder of the song, using energy spikes in the audio signal to fine-tune the placement of beats. However, by generating ground truth using assumptions similar to those embodied in the models we intend to test, we risk invalidating the results. We instead opted to use 'raw' human-labelled songs.

There is no standard evaluation metric for beat tracking. We use the ρ function presented by Cemgil *et al* [5] and used by Dixon [7] in his analysis:

$$\rho(S,T) = \frac{100}{(N_S + N_T)/2} \sum_{i=1}^{N_S} \max_{j \in T} \exp\left\{ -\frac{(S_i - T_j)^2}{2\sigma^2} \right\} ,$$

where S and T are the ground-truth and proposed beat times, and σ is set to 40 milliseconds. A ρ value near 100 means that each predicted beat is close to a true beat, while a value near zero means that each predicted beat is far from a true beat.

We have focused on finding a globally-optimum beat track rather than precisely locating each beat. We could likely improve the ρ values of our results by fine-tuning each predicted beat, for example by finding nearby energy peaks, though we have not done this in the results presented here.

Table 4.1 shows a summary of our results. Note the wide range of genres and the choice of songs with features that we thought would make beat tracking difficult. This includes all our results (not just the ones that look good).

The first two columns list the name of the song and its genre or the reason we included it. An [(e)] indicates that we edited the song to remove talking on long periods of silence. An [(x)] indicates that only an excerpt of the song was available. The third column lists the qualitative performance of the fixed grid (FG) version: [$\times 2$] means our algorithm produced a beat track twice as fast as ground truth, [$\times 1/2$] means we tracked at half speed, and [π] means we produced a syncopated (π phase error) beat track. A blank entry means our algorithm produced the correct beat track. A star [$\star$] means that our result incorrectly switches phase or tempo. In one case, we tracked at 3/2 the true tempo; this is marked as [$\diamondsuit$]. The ρ values are after compensating for the qualitative error (if any). The fifth column shows a histogram of the absolute phase error (0 to π); this is also after correcting for qualitative error. The remaining columns

discussion of Table 4.1.

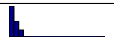
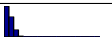
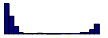
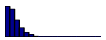
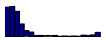
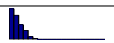
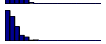
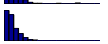
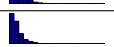
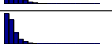
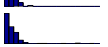
Song	Comment	FG	ρ	Phase Err	PF	ρ	Phase Err
Glenn Gould / Bach Goldberg Var'ns 1982 / Var'n 1	Classical piano		88			86	
Jeno Jandó / Bach WTC / Fuga 2 (C Minor)	Piano; rubato at end		77			77	
Kronos Quartet / Caravan / Aaj Ki Raat	Modern string quartet		75			71	
Maurice Ravel / Piano Concertos / G Major - Presto	Classical orchestra	$\star$ π	44		$\star$ π	50	
Miles Davis / Kind Of Blue / So What (e)	Jazz instrumental	$\times 1/2$	61		$\times 1/2$	59	
Miles Davis / Kind Of Blue / Blue In Green	Jazz instrumental		57			59	
Holly Cole / Temptation / Jersey Girl	Jazz vocal		78			77	
Don Ross / Passion Session / Michael Michael Michael	Solo guitar	$\star$ $\diamondsuit$	40		$\star$ $\diamondsuit$	42	
Don Ross / Huron Street / Luci Watusi	Solo guitar		70			69	
Tracy Chapman / For You	Guitar and voice	$\times 2$	59		$\times 2$	61	
Ben Harper / Fight For Your Mind / Oppression	Acoustic		70			68	
Great Big Sea / Up / Chemical Worker's Song	Newfoundland folk		79			78	
Buena Vista Social Club / Chan Chan	Cuban		72			72	
Beatles / 1967-1970 / Lucy In The Sky ...	Changes time signature	$\star$	42		$\star$	41	
U2 / Joshua Tree / Where The Streets... (e)	Rock		82			82	
Cake / Fashion Nugget / I Will Survive	Rock	π	81		π	80	
Sublime / Second-Hand Smoke / Thanx Dub (x)	Reggae		79			79	
Rancid / ... And Out Come The Wolves / Old Friend	Punk	$\times 1/2$	82		$\times 1/2$	79	
Green Day / Dookie / When I Come Around	Pop-punk		75			74	
Tortoise / TNT / A Simple Way ...	Organic electronica	$\times 2$	79		$\times 2$	79	
Pole / 2 / Stadt	Ambient electronica		71			71	
Underworld / A Hundred Days Off / MoMove	Electronica		79			79	
Ravi Shankar / The Sounds Of India / Bhimpalsi (e)	Solo sitar		71			67	
Pitamaha: Music From Bali / Puri Bagus, Bamboo (x)	Indonesian gamelan		86		π	89	
Gamelan Sekar Jaya / Byomantara (x)	Indonesian gamelan		89			88	

Table 4.1: The songs used in our evaluation. See the text on page 111 for explanation.

contain the same items for the particle filter (PF) version. Note that while the FG and PF versions are often nearly identical, they occasionally choose qualitatively different answers.

Out of 25 examples, the fixed grid version produces the correct answer in 17 cases, tracks at double speed in two cases, half speed in two cases, syncopated in one case, and in three cases produces a track that (incorrectly) switches tempo or phase. The particle filter version produces 16 correct answers, two double-speed, two half-speed, two syncopated, and the same three 'switching' tracks.

An example of a successful tempo track is shown in Figure 4.6.

The result for *Lucy In The Sky With Diamonds* (one of the 'switching' results) is worth examination; see figure 4.7. It is also interesting to examine the final message passed during Max-BP: we can see several local maxima corresponding to qualitatively different solutions; see Figure 4.8.

4.9 Future Work

We would like to investigate several modifications of our model and inference methods. Longer-range tempo smoothness constraints as suggested by [5] could be useful. This would allow us to express that a tempo change to a new value and back to the original value is more likely than two arbitrary tempo changes.

It would be useful to extract MAP sets of parameters for several qualitatively different solutions, as this would help to express the ambiguity of the problem. As we saw in the *Lucy In The Sky With Diamonds* example, there can be multiple plausible beat tracks for many pieces of music. This is a harder inference problem than finding k-best MAP sets, since we want the solutions to belong to different local maxima.

The particle filter could also be changed. At present, we first perform a full particle filtering sweep and then run Max-BP. Taking into account the quality of the partial MAP solutions during particle filtering might allow superior results by directing more particles toward regions of the state space that are likely to contain the final MAP solution. Since we know that our probability terrain is multi-modal, a mixture particle

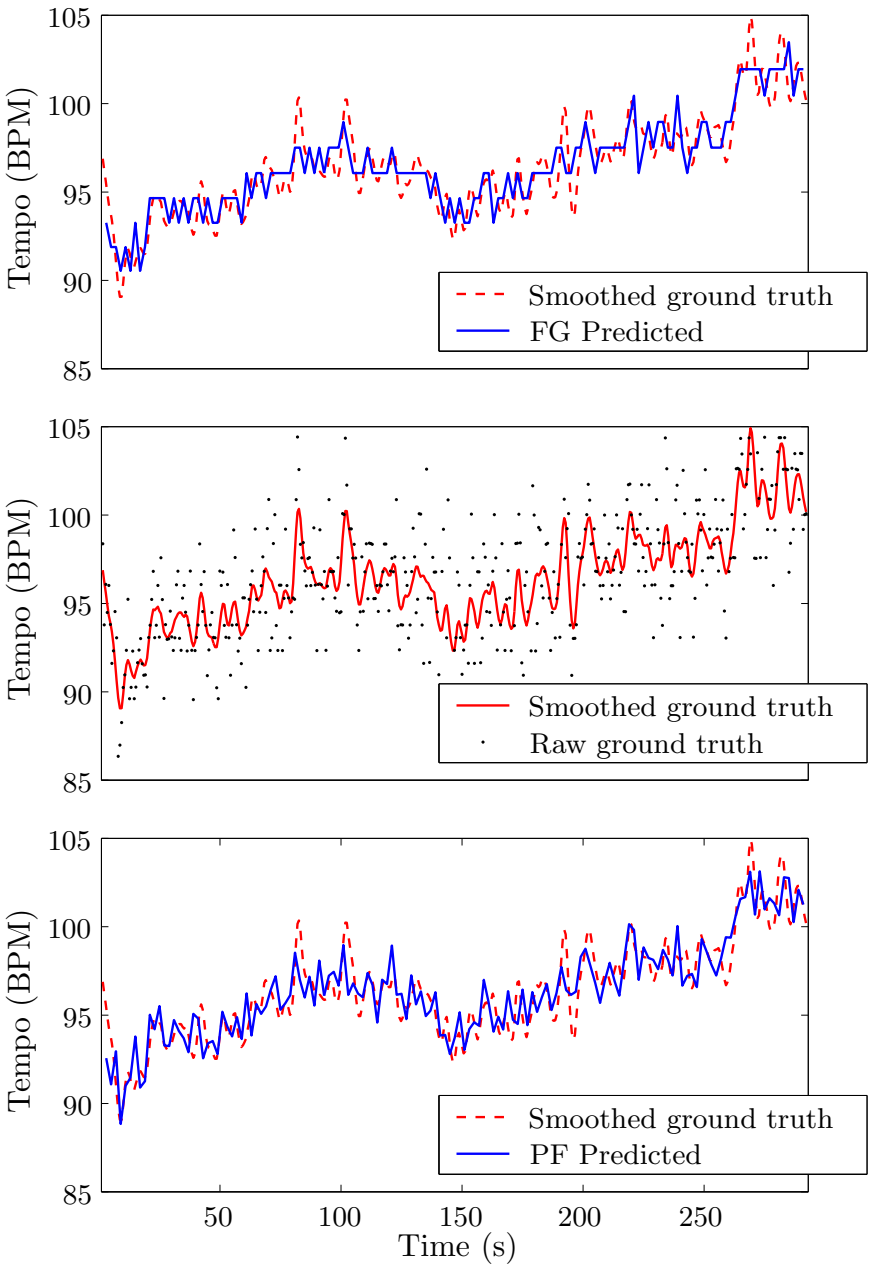


Figure 4.6: Tempo tracks for *Cake / I Will Survive*. Middle: 'raw' ground-truth tempo (instantaneous tempo estimate based on the time between adjacent beats) and smoothed ground truth (by averaging). Top: tracking results for the fixed grid version. Bottom: results for the particle filter version.

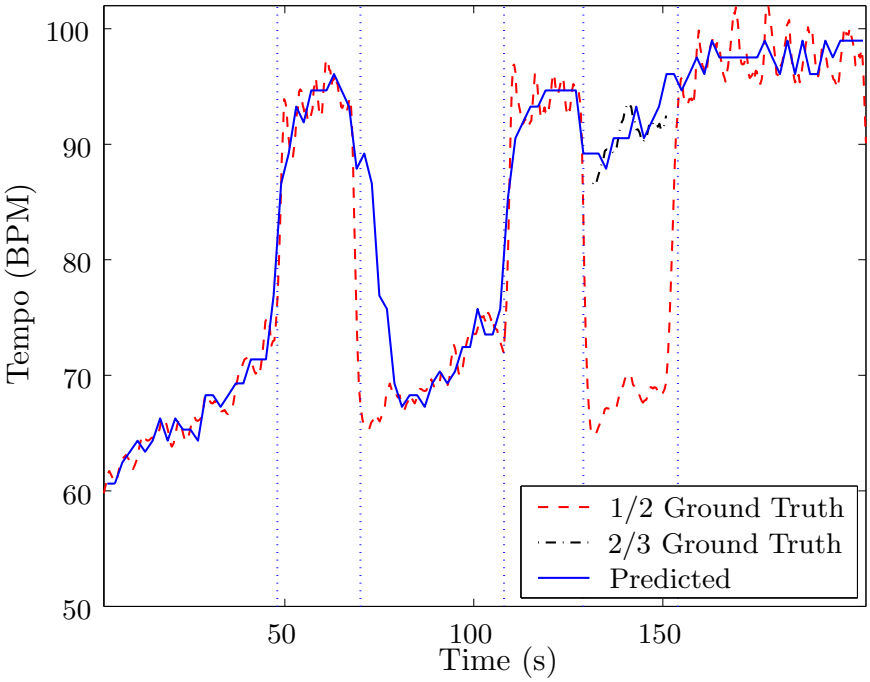


Figure 4.7: Tempo tracks for *Lucy In The Sky With Diamonds*. The vertical lines mark times at which the time signature changes between 3/4 and 4/4. We track correctly (at half speed), and follow the first three time signature changes (with a slight glitch at the second change). On the fourth change, however, we track at 2/3 the true tempo. On the fifth change we return to the correct beat.

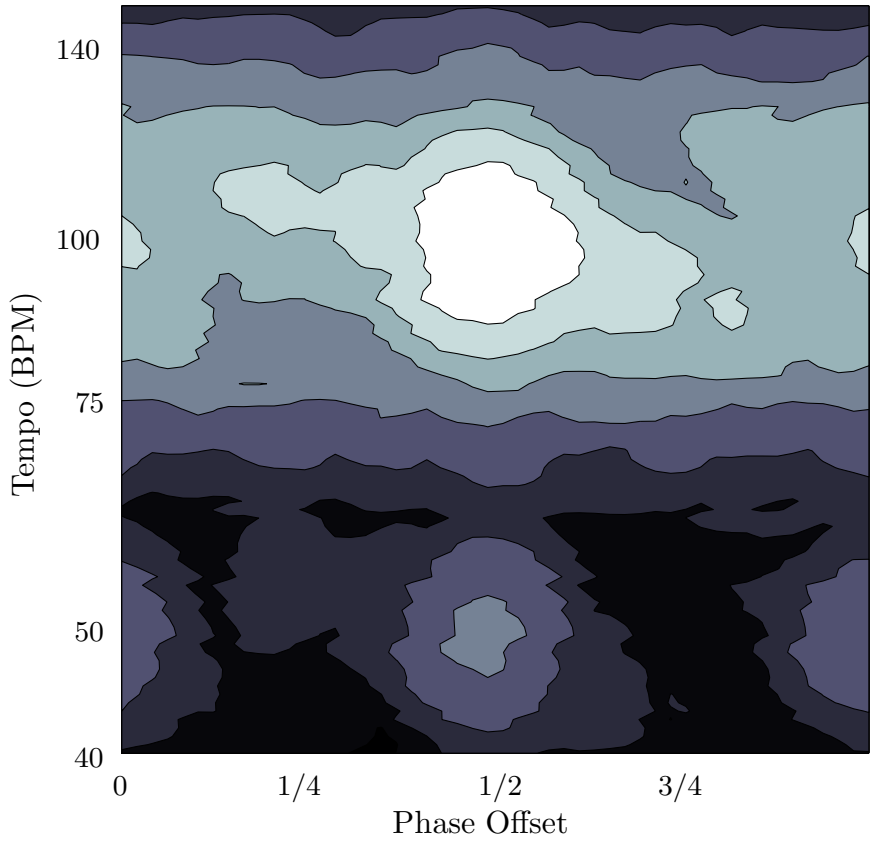


Figure 4.8: The last message passed during Max-BP. Bright means high probability. This message tells us the probability of a sequence that ends with each state. The global maximum (near 100 BPM, phase offset 1/2) corresponds to the beat track shown in the Figure 4.7. The local maximum near 50 BPM corresponds to an alternate solution in which, rather than tracking the quarter notes, we track one beat per measure. This alternate solution is quite plausible; some human listeners will produce it. Note also the local maximum near 100 BPM but phase-shifted by a half beat from the global maximum. This corresponds to a syncopated beat track.

filter would be useful [37].

Our current model of tempo change (ψ_T) does not take into account the fact that abrupt tempo changes to related tempos (half, double, etc.) do occur.

An issue that we have not considered is parameter estimation: we have set σ_Q, σ_P, and σ_T arbitrarily. It would be preferable to learn optimal values for these hyperparameters from the inputs. Some songs, including most electronic, rock, and pop music, have very steady tempos and rarely have phase discontinuities. Others, particularly solo performances of classical music, have large tempo variance (musically, this is known as *rubato*). Having a single set of hyperparameters for all types of music is unsatisfying. Allowing these values to vary would likely improve our results, and, indeed, the optimal values of the hyperparameters could perhaps be used for tasks such as genre identification.[1]

4.10 Conclusions

We present a graphical model for beat tracking and evaluate it on a set of varied and difficult examples. We achieve good results that are comparable with those reported by other researchers, although direct comparisons are impossible without a shared data set and evaluation metric.

There are several advantages to formulating the problem in a probabilistic setting. The beat tracking problem has inherent ambiguity and multiple interpretations are often plausible. With a probabilistic model, we can produce several candidate solutions with different probabilities. This is particularly useful for situations in which beat tracking is one element in a larger machine listening application. Probabilistic graphical models allow flexible and powerful handling of uncertainty, and allow local and contextual information to interact in a principled manner. Additional domain knowledge and constraints can be added in a clean and principled way.

[1] One of our anonymous reviewers pointed out the previous two issues.

Chapter 5

Farewell

Since I started my thesis with a welcome, it seems fitting to end it with a farewell.

Let us review briefly what we've discussed. Chapter 2 got us up and running, thinking about graphical models and how we compute with them. We reviewed the most common techniques: Belief Propagation, in both standard and Max flavours, plus Particle Filtering and Particle Smoothing.

Chapter 3 presented a lot of material. First, we presented the problems we want to solve, which we call the Sum-Product and Max-Product. Next, we reviewed some of the existing fast methods for the Sum-Product, including the Fast Multipole Method, Fast Gauss Transform, Improved FGT, the Box Filter trick, and the Dual-Tree method. We then took a brief digression to present some new ideas for tightening the bounds that are central to the Dual-Tree method.

Next, we switched to the Max-Product problem. We presented the Distance Transform, which can be used to solve some Max-Product problems in cases where the state space is a regular grid. We then presented a novel algorithm for the Max-Product problem using a Dual-Tree approach.

We then returned to graphical models and stated explicitly how the fast methods can be used to solve Belief Propagation, Max-Belief Propagation, and Particle Smoothing. We also presented the idea of using the Sum-Product algorithms to speed up Gaussian Processes; one of the key computations there involves computing the product of an inverse kernel matrix and a vector. This can be solved with the fast conjugate gradient method, which requires many Sum-Product computations. In this setting the weights can be negative, so some of the fast methods would need to be extended. Doing this would open the door to solving many problems that can be

expressed as the multiplication of a structured kernel with an arbitrary vector.

To complete the chapter, we presented some empirical tests that compare the fast methods head-to-head. Some of the results challenge claims in the literature about the performance of the fast methods. For example, the Dual-Tree method using the Anchors Hierarchy, and the Improved Fast Gauss Transform are both supposed to perform well with high-dimensional, clustered data. However, we find that this is not the case: the Dual-Tree *kd*-treeis faster. Our empirical evaluation highlights the importance of considering factors other than N (the number of particles). The degree of structure in the data and the properties of the kernel function have a much greater effect on the empirical performance.

In chapter 4, we presented an application of graphical models to the beat tracking problem. Given a piece of sampled music, we want to produce an estimate of the tempo track of the song, and an estimate of the location of beats. The results are quite good, and by using a probabilistic approach we leave open the possibility of combining the beat tracker with other machine listening applications.

I thank you for your attention, dear reader, and bid you farewell.

Bibliography

[1] C Allain and M Cloitre. Characterizing the lacunarity of random and deterministic fractal sets. *Physical Review A*, 44(6):3552–3558, Sep 1991.

[2] B J C Baxter and G Roussos. A New Error Estimate of the Fast Gauss Transform. *SIAM Journal of Scientific Computing*, 24(1):257–259, 2002.

[3] R K Beatson and L Greengard. A short course on fast multipole methods. In M Ainsworth, J Levesley, W Light, and M Marletta, editors, *Wavelets, Multilevel Methods and Elliptic PDEs*, pages 1–37. Oxford University Press, 1997.

[4] G Borgefors. Distance Transformations in digital images. *Computer Vision, Graphics, and Image Processing*, 34:344–371, 1986.

[5] A T Cemgil, B Kappen, P Desain, and H Honing. On Tempo Tracking: Tempogram Representation and Kalman Filtering. *Journal of New Music Research*, 28(4):259–273, 2001.

[6] A T Cemgil and H J Kappen. Monte Carlo Methods for Tempo Tracking and Rhythm Quantization. *Journal of Artificial Intelligence Research*, 18(1):45–81, 2003.

[7] S Dixon. An Empirical Comparison of Tempo Trackers. Technical Report TR-2001-21, Austrian Research Institute for Artificial Intelligence, Vienna, Austria, 2001.

[8] A Doucet. On sequential simulation-based methods for Bayesian filtering. Technical Report CUED/F-INFENG/TR 310, Department of Engineering, Cambridge University, 1998.

[9] A Doucet, N de Freitas, and N J Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, 2001.

[10] P F Felzenswalb, D P Huttenlocher, and J M Kleinberg. Fast Algorithms for Large-State-Space HMMs with Application to Web Usage Analysis. In *Advances in Neural Information Processing Systems 16*, 2003.

[11] P F Felzenszwalb and D P Huttenlocher. Efficient Belief Propagation for Early Vision. In *CVPR*, 2004.

[12] P F Felzenszwalb and D P Huttenlocher. Distance Transforms of Sampled Functions. Technical Report TR2004-1963, Cornell Computing and Information Science, September 2004.

[13] B J Frey, R Koetter, and N Petrovic. Very loopy belief propagation for unwrapping phase images. In *Advances in Neural Information Processing Systems 14*, pages 737–743, 2002.

[14] M N Gibbs. *Bayesian Gaussian Processes for Regression and Classification*. PhD thesis, University of Cambridge, 1997.

[15] M N Gibbs and D J MacKay. Efficient implementation of Gaussian Processes. `http://www.inference.phy.cam.ac.uk/mng10/GP/gpros.ps.gz` or `http://www.cs.toronto.edu/~mackay/gpros.ps.gz`, 1997.

[16] S J Godsill, A Doucet, and M West. Maximum a posteriori sequence estimation using Monte Carlo particle filters. *Ann. Inst. Stat. Math.*, 53(1):82–96, March 2001.

[17] M Goto. An Audio-based Real-time Beat Tracking System for Music With or Without Drum-sounds. *Journal of New Music Research*, 30(2):159–171, 2001.

[18] A G Gray and A W Moore. 'N-Body' Problems in Statistical Learning. In *Advances in Neural Information Processing Systems 4*, pages 521–527, 2000.

[19] A G Gray and A W Moore. Rapid Evaluation of Multiple Density Models. In *Artificial Iintelligence and Statistics*, 2003.

[20] A G Gray and A W Moore. Nonparametric Density Estimation: Toward Computational Tractability. In *SIAM International Conference on Data Mining*, 2003.

[21] L Greengard and J Strain. The Fast Gauss Transform. *SIAM Journal of Scientific Statistical Computing*, 12(1):79–94, 1991.

[22] L Greengard and X Sun. A New Version of the Fast Gauss Transform. *Documenta Mathematica*, ICM(3):575–584, 1998.

[23] J M Hammersley and P Clifford. Markov fields on finite graphs and lattices. Unpublished manuscript, 1971.

[24] Firas Hamze and Nando de Freitas. From Fields To Trees. In *UAI*, 2004.

[25] A Ihler. An overview of fast multipole methods. Area Exam; `http://ssg.mit.edu/~ihler/papers/ihler_area.pdf`.

[26] Michael I Jordan. *Introduction to Probabilistic Graphical Models*. In Press.

[27] D LaLoudouana and M B Tarare. Data Set Selection. Presented at NIPS Workshop, 2002.

[28] Dustin Lang and Nando de Freitas. Beat Tracking the Graphical Model Way. In *Advances in Neural Information Processing Systems 17*, 2004.

[29] D J MacKay. Introduction to Gaussian Processes. `http://www.cs.toronto.edu/~mackay/gpB.ps.gz`, 1997.

[30] R J McEliece, D J C MacKay, and J-F Cheng. Turbo Decoding as an Instance of Pearl's 'Belief Propagation' Algorithm. *IEEE Journal on Selected Areas in Communication*, 16(2):140–152, 1998.

[31] A W Moore. The Anchors Hierarchy: Using the triangle inequality to survive high dimensional data. Technical Report CMU-RI-TR-00-05, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, February 2000.

[32] Kevin P Murphy, Y Weiss, and Michael I Jordan. Loopy-belief Propagation for Approximate Inference: An Empirical Study. In *Uncertainty in AI (UAI)*, 1999.

[33] J Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan-Kaufmann, 1988.

[34] R E Plotnick, R H Gardner, W W Hargrove, K Prestegaard, and M Perlmutter. Lacunarity analysis: a general technique for the analysis of spatial patterns. *Physical Review E*, 53(5 B):5461–5468, May 1996.

[35] E D Scheirer. Tempo and Beat Analysis of Acoustic Musical Signals. *J. Acoust. Soc. Am.*, 103(1):588–601, Jan 1998.

[36] J Strain. The Fast Gauss Transform with variable scales. *SIAM Journal of Scientific and Statistical Computing*, 12:1131–1139, 1991.

[37] J Vermaak, A Doucet, and Patrick Pérez. Maintaining Multi-Modality through Mixture Tracking. In *ICCV*, 2003.

[38] A S Willsky. Multiresolution Markov models for signal and image processing. *Proceedings of the IEEE*, 90(8):1396–1458, August 2002.

[39] C Yang, R Duraiswami, and N A Gumerov. Improved fast gauss transform. Technical Report CS–TR–4495, UMIACS, University of Maryland, College Park, 2003.

[40] C Yang, R Duraiswami, N A Gumerov, and L S Davis. Improved Fast Gauss Transform and Efficient Kernel Density Estimation. In *International Conference on Computer Vision*, Nice, 2003.

[41] J S Yedidia, W T Freeman, and Y Weiss. Understanding Belief Propagation and Its Generalizations. Technical Report TR2001-22, Mitsubishi Electric Research Labs, 2001.

Index