Fast High-Dimensional Approximation with Sparse Occupancy Trees

Peter Binev, Wolfgang Dahmen, Philipp Lamby

April 8, 2010

Abstract

This paper is concerned with scattered data approximation in high dimensions: Given a data set $X \subset \mathbb{R}^d$ of N data points x^i along with values $y^i \in \mathbb{R}^{d'}$, $i = 1, \dots, N$, and viewing the y^i as values $y^i = f(x^i)$ of some unknown function f, we wish to return for any query point $x \in \mathbb{R}^d$ an approximation $\tilde{f}(x)$ to $y = f(x)$. Here the spatial dimension d should be thought of as large. We wish to emphasize that we do not seek a representation of $\tilde{f}$ in terms of a fixed set of trial functions but define $\tilde{f}$ through recovery schemes which, in the first place, are designed to be fast and to deal efficiently with large data sets. For this purpose we propose new methods based on what we call *sparse occupancy trees* and piecewise linear schemes based on *simplex subdivisions*.

AMS Subject Classification: 41A63, 62G08, 65D05

Key Words: high-dimensional approximation, non-parametric regression, non-linear approximation, multiresolution tree

1 Introduction

Methods for high-dimensional function approximation and statistical learning are commonly categorized into two classes. For an overview we refer to [HTF09]. On the one hand we have parametric methods which try to fit the function globally, typically by prescribing the structure of the approximant by defining a set of trial functions and learning the coefficients in this approach by optimizing some error norm. Here one can think, for example, of generalized additive models, projection pursuit or artificial neural networks. Although these methods have been applied successfully in a large number of applications they also have some drawbacks. First, the class of functions that are approximated well by such techniques is typically small and the right model has to be determined a priori. Second the training stage usually involves the solution of a non-linear optimization problem which might be a demanding and time-consuming process effectively limiting the size of the data that can be handled. Furthermore these approximations cannot easily be adjusted to new data, for example in the case of incremental online learning or applications where the domain in which the function is to be evaluated changes in time.

On the other side of the spectrum there are non-parametric methods which try to fit functions locally, usually by partitioning the input space and then using simple local models like piecewise constant approximations. The idea of being content with piecewise constants is supported by classical concentration of measure results according to which a well-behaved function (e.g. Lipschitz-continuous) in very high dimensions deviates from its mean or median by much only on sets of small measure.

A typical example for such a recovery strategy is to determine for any given query point its *k nearest neighbors* in the given data site and to use their average as the approximate function value. At first glance this kind of memory-based learning does not seem to require any training process except of reading and storing the incoming data. In practice, however, it is necessary to design a data structure that provides a fast solution to the question which are the nearest neighbors of a query point x. Unfortunately, the exact solution requires either a preprocessing time which is exponential in d or a single query time which is linear in N, the latter characterizing the brute force algorithm where the distance $\|x^i - x\|$ is computed for each training point. Actually, for function recovery purposes one would also be satisfied with an approximate solution which can be achieved much more efficiently. Here we refer to [Ind04, LMGY05] which give some of the basic ideas about the algorithms which could be useful in this context. However, none of the currently available methods seems to perform very well if d goes into the hundreds and N into the millions. An application background from climatology in which problems of this size arise will be described in a subsequent report [BBD$^+$on].

Therefore, in situations where a fast return to a query of a function evaluation matters, such as when approximate function values are required in discretizations of PDEs, say, alternative strategies may be preferable. In view of these considerations we develop and investigate in this paper some methods which in the first place are designed to be fast and to deal efficiently with large data sets and to provide fast algorithms for evaluation. The motivation behind our approach is to explore the potential of multiresolution ideas for high spatial dimension.

In the last two decades multilevel methods have proved to be essential for approximating functions with inhomogeneous local structural properties and have been successfully applied in different forms in several areas ranging from image processing to solutions of partial differential equations. A central ingredient of multiresolution analysis is the *tree* which describes the relations between levels of resolution. In a standard setting the initial domain Ω is related to a cube which is the root of the tree. Then, using consecutive dyadic partitions one can define different levels of resolution and build the corresponding tree structure level by level. To this end, as a key tool for data organization we propose the notion of *sparse occupancy trees*. The underlying concepts, perhaps with a different terminology, have been certainly used in somewhat different contexts such as nearest neighbor search. To our knowledge its use in recovery procedures seems to be new. Instead of using the full tree $\mathcal{T}$ (called the *master tree*), we consider only a subtree $\mathcal{T}(\mathcal{X})$ whose nodes correspond to the cubes that are occupied, i.e. contain at least one element from the set $\mathcal{X}$. A special indexing and ordering of these cubes allows us to store of all the information about the tree using only $\mathcal{O}(LdN)$ bits where L is a chosen upper limit for the number of levels in the tree. The sparse occupancy tree can then be used as a tool for constructing approximations to the function represented by the data.

We want to emphasize that we are not considering the sparse occupancy trees and

the problem of nearest neighbors as separate issues. We want to blend them in such a way that the resulting solution will give efficient and reliable recovery schemes for a for a variety of problems of the above type. In this context we have to mention that one can hardly expect a good performance of the nearest neighbor scheme or our schemes if d is very large and the x-data is distributed uniformly in $\mathbb{R}^d$. In this case the average distance between two data points would be large even for huge data sets, and *any* method based on localization strategies would be doomed to fail. However, in practical problems the input variables are often strongly correlated and the true intrinsic dimensionality of the data is much lower than the formal dimension of the problem. Multiresolution trees seem in particular suited to capture such coherent structures and, hence, to mitigate the curse of dimensionality.

In Section 2 we introduce a most simple algorithm providing a piecewise constant approximation. For a given point $x \in X$ it finds the finest cube $K(x)$ from $\mathcal{T}(\mathcal{X})$ to which it belongs. Then the approximated value at x could be set to be the mean value of the points from $K(x) \cap \mathcal{X}$. The time for a single query for the node of the sparse tree that corresponds to any point from the domain Ω will be shown to be $\mathcal{O}(\log N)$. A detailed study of this case and numerical tests provide useful insight concerning the following issue: the quality of this approximation depends significantly on the size of the cube $K(x)$ which could be large even if there are points from $\mathcal{X}$ close to x. The latter is subject to the way the partition is set. This is the case when the geodesic distance is much smaller than the distance in the tree. Of course, ignoring the function values in a neighboring cube which as a node in the tree is far from the cube holding the query point x, is likely to lead to highly inappropriate assignments of approximate function values. This observation will guide several attempts to improve upon the above described basic strategy. Since the evaluation process is very fast, a first idea is to generate several partitions of the same data by randomly shifting the partition boundaries and to use the weighted averages of the corresponding approximations. It will be shown that such an approach indeed has a significant effect.

In Section 3 we extend this technology to construct piecewise linear approximations on simplex subdivisions. Working with simplices offers a number of advantages which is a common experience in numerical grid generation even in lower dimensions. Therefore it is a little bit surprising that one hardly finds references discussing simplex partitioning methods in high dimensions. The elementary observation behind this is that a d-dimensional simplex has only $d + 1$ vertices compared to the 2^d vertices of a hyper-cube. Prescribing values for the vertices one can define piecewise linear approximations as demonstrated in Section 3. The values at the vertices are defined as weighted averages of the points in the surrounding simplices. The value of the query point is found by interpolating vertex values of the simplex the query falls into. Hence, the query response becomes an average of all training points in the neighborhood of the query coordinates, even including the points to which the tree distance is large. In fact, this was the main motivation for the development of the vertex scheme: to overcome the deficiencies of piecewise constant partitioning methods while retaining the efficiency of tree based algorithms. Indeed, in Section 4 we will show by numerical experiments that this scheme gives an accuracy like the nearest neighbor approximation, but much faster if the space dimension becomes large.

2 Sparse Occupancy Trees

In this section we describe the general construction of *sparse occupancy trees* and propose efficient data structures for their practical realization. Then we will use this construction to design a very fast recovery scheme based on cube subdivision.

2.1 Basic Form and Piecewise Constant Approximation

Let us assume that the data set X is contained in a bounded domain $\Omega \subset \mathbb{R}^d$. Suppose, that we have a a hierarchy of nested partitions of Ω:

$$\{\Omega\} = \mathcal{P}_0 \prec \mathcal{P}_1 \prec \ldots \prec \mathcal{P}_j \prec \ldots \ ,$$

which means that for all $l \geq 0$ the sets $\mathcal{P}_l = \{\Omega_{l,k}, k \in \mathcal{I}_l\}$ are partitions of Ω and each cell $\Omega_{l,k} \in P_l$ is the disjunct union of cells on the next finer level $l+1$:

$$\Omega_{l,k} = \bigcup_{r \in \mathcal{I}_{l,k}} \Omega_{l+1,r} \ .$$

Typically, the partitions consist of cubes or simplices and the refinement sets $\mathcal{I}_{l,k}$ have a fixed cardinality.

The hierarchy of partitions induces an infinite *master tree* $\mathcal{T}^*$, whose root is Ω and whose other nodes are the cells $\Omega_{l,k}$. Each node $\Omega_{l,k}$ of this tree is connected by an edge to its children $\Omega_{l+1,r}$ where $r \in \mathcal{I}_{l,k}$.

In practice we only consider finite subtrees of some fixed depth L, i.e., $l \leq L$. L can be determined according to several possible criteria, such as spatial resolution reflected by $\operatorname{diam} \Omega_{L,k}$, or by separation, which means that each leaf $\Omega_{L,k}$ contains at most one data point from X.

Given $x \in \Omega$ we denote by $\mathcal{T}(x)$ the branchless subtree of $\mathcal{T}^*$, which only contains the cells containing x:

$$\mathcal{T}(x) = \{\Omega_{l,k} \in \mathcal{T}^* \ : \ x \in \Omega_{l,k}\} \ .$$

The *sparse occupancy tree* $\mathcal{T}(X)$ is the tree we get from the master tree by cutting all cells which do not contain a training point; equivalently, it is the largest subtree of $\mathcal{T}^*$ such that all its nodes contain an $x \in X$:

$$\mathcal{T}(X) := \bigcup \{\mathcal{T}(x^i) \ : \ x^i \in X\} \ . \tag{1}$$

A piecewise constant approximation can now be defined as follows: given a training set $X = \{x^1, \ldots, x^N\}$ and a test point x we average the values in the leaf of the branchless tree $\mathcal{T}(X) \cap \mathcal{T}(x)$:

$$\tilde{f}(x) = \mathcal{A}(\{y^i | x^i \in \mathcal{L}(\mathcal{T}(X) \cap \mathcal{T}(x))\}). \tag{2}$$

where we use the notation

$$\mathcal{A}(Y) = \frac{1}{\operatorname{card}(Y)} \sum_{y \in Y} y \tag{3}$$

to denote the average of a finite subset $Y \subset \mathbb{R}^{d'}$.

In other words, we identify the maximum level cell in the sparse occupancy tree that contains the test point and then average the values of the training points contained in this cell.

Remark 1. *Note that the scheme is interpolating (i.e., if a query coincides with a training point the algorithm returns the value of this training point), if each leaf of the tree contains only training sample.*

2.2 Occupancy Trees as Sorted Lists

It is possible to write computer codes, that follow word-by-word the above construction, i.e., one can implement a structure with node elements and pointers to their children to represent the occupancy tree. However, especially with large data sets, this typically does not result in the most efficient code. Instead, we represent the occupancy tree by a sorted list of strings, as described in the following.

2.2.1 Data Structures

We define a string $\mathbf{b}$ to be a finite sequence of integers: $\mathbf{b} = (b_1, b_2, \ldots, b_n)$, where n is the length of the key. The elements b_i of a string will also be called *characters*. We denote with $(\mathbf{b}, c) = (b_1, \ldots, b_n, c)$ the string that results from appending an additional number to the sequence. If $j < n$ we write $\mathbf{b}|_j$ for the substring that consists of the first j elements of $\mathbf{b}$: $\mathbf{b}|_j = (b_1, \ldots, b_j)$.

Our first aim is to construct an invertible map of the nodes in the master tree to the set of strings. We can do this recursively.

First we map the root node Ω to the empty string $\emptyset$. For each node $\Omega_{l,k}$ we prescribe an enumeration of its children $\Omega_{l+1,k_0}, \ldots, \Omega_{l+1,k_{r-1}}$ where $r = r(l,k)$ is the cardinality of $\mathcal{I}_{l,k}$. Then we assign for $i = 0, \ldots, r-1$ the strings $\mathbf{b}(\Omega_{l+1,k_i}) = (\mathbf{b}(\Omega_{l,k}), i)$ to the children of $\Omega_{l,k}$.

Clearly, a cell at level l is mapped to a string of length l and the mapping of all cells in the master tree into the set of strings of length l is injektive. Therefore, given a string $\mathbf{b}$ in the range of $\mathcal{T}^*$, we will use the notation $\Omega(\mathbf{b})$ to denote the cell that is mapped to the string $\mathbf{b}$.

Hence, we can make the following simple observations: if $\mathbf{b}$ has length n and $j < n$ then $\Omega(\mathbf{b}) \subset \Omega(\mathbf{b}|_j)$. In particular, if two strings $\mathbf{b}^1$,$\mathbf{b}^2$ of length $n > j$ have their first j bits to be identical, but $b^1_{j+1} \neq b^2_{j+1}$ then $\Omega(\mathbf{b}^1|_j) = \Omega(\mathbf{b}^2|_j)$ is the finest cell that contains both $\Omega(\mathbf{b}^1)$ and $\Omega(\mathbf{b}^2)$.

Finally, if $x \in \Omega$ and a maximum level L of the master tree is given, we denote with $\mathbf{b}(x)$ the string that is assigned to the the finest level cell Ω_{Lk} which contains x: $x \in \Omega(\mathbf{b}(x)) \in \mathcal{L}(\mathcal{T}^*)$.

2.2.2 Algorithm

The approximation defined by equation (2) can now be realized by the following algorithm that consists of a training stage and an evaluation stage. Again, we assume that the maximum level of the master tree L is prescribed. The *training stage* consists of the following steps:

1. For every training point x^i compute the string $\mathbf{b}(x^i)$.
2. Sort the $\mathbf{b}(x^i)$ lexicographically. Note that the lexicographical ordering of the nodes induces a new ordering of the points in X. Without loss of generality we will assume in what follows that the points x^i were already ordered in the same way. In the implementation one has to store the resulting permutation, of course.

3. For all $n = 1, \ldots, N$, compute the running sums

$$Y^n = \sum_{i=1}^{n} y^i = Y^{n-1} + y^n \ ,$$

where for convenience we set $Y^0 = (0, \ldots, 0)$.

Remark 2. *The computation of the strings requires the generation of $\mathcal{O}(LN)$ characters. The sorting operation can be done in $\mathcal{O}(dN \log(N))$ time, if we assume that two strings can be compared in $\mathcal{O}(d)$ time. The storage is LN characters for the strings plus N integers for the permutation vector and $\mathcal{O}(d'N)$ real values for the running sums.*

Now let us assume that a query point x is given. Then, in the *evaluation stage*, we have to find the finest cell in the occupancy tree that contains x and to average the values of points in this cell. These points correspond to the strings which share the maximum number of leading characters with $\mathbf{b}(x)$ among all strings in the sorted list. These strings can be identified as follows:

1. Find the position n, such that $\mathbf{b}(x^n) < \mathbf{b}(x) \leq \mathbf{b}(x^{n+1})$, where $<$ denotes the relation induced by the lexicographical ordering.
2. Compare $\mathbf{b}(x)$ with $\mathbf{b}(x^n)$ and $\mathbf{b}(x^{n+1})$. The maximum number of leading characters is
$$j := \max\{j : \mathbf{b}(x)|_j = \mathbf{b}(x^n)|_j \vee \mathbf{b}(x)|_j = \mathbf{b}(x^{n+1})|_j\} \ .$$
3. Find the position m such that $\mathbf{b}(x^{m-1}) < \mathbf{b}(x)|_j \leq \mathbf{b}(x^m)$. Obviously x^m is the first point that shares j characters with $\mathbf{b}(x)$, because $\mathbf{b}(x)|_j$ is the smallest string that starts with $\mathbf{b}(x)|_j$.
4. Generate the string $\tilde{b} = (\mathbf{b}, R, \ldots, R)$ by appending $(L - j)$-times the maximum cardinality R of all sets $\mathcal{M}_{l,k}$. This is obviously the last possible string that begins with $\mathbf{b}(x)|_j$. Then search the position p such that $\mathbf{b}(x^p) \leq \tilde{b} < \mathbf{b}(x^{p+1})$. Clearly, x^p is the last point in the list that has to be considered for averaging.
5. The value of the approximation can then be computed by evaluation of the running sums:
$$\tilde{f}(x) = \frac{1}{p - m + 1}(Y^p - Y^{m-1})$$

Remark 3. *This evaluation algorithm essentially consists of three search algorithms, which, in the worst case, can be performed in $\mathcal{O}(d \log(N))$ time each by binary subdivision. In practice the effort is usually smaller, because n, m, and p are close together in the list, so that n can be used as good initial guess for the other two search operation. The evaluation of the running sum requires only constant time. Hence, the operation count for a single function evaluation is $\mathcal{O}(3d \log(N))$.*

Remark 4. *Especially if the data sets are very large, the method of running sums might become a source of numerical inaccuracies caused by cancellation or overflow. Remedies for this is to break the running sum into* chunks *or to use several* buckets *for values of different sign or magnitude.*

Remark 5. *It is obvious that in a computer program the above algorithm can be implemented in the most efficient manner, if the characters are bits (or sequences of bits); in this case the strings can be represented by* bitstreams. *This is the case for all binary trees and for the dyadic subdivision schemes which we will consider in this paper.*

2.3 Cube Subdivision

In this section we assume that $\Omega = [0,1]^d$ is the d-dimensional hypercube. For convenience we will omit the prefix "'hyper" most of the time and speak of cubes and cuboids, although in general $d \neq 3$. If the data is initially not contained in $[0,1]^d$, this can usually be achieved by rescaling the training data X component-wise, in particular when the training data is given beforehand instead of coming in incrementally.

2.3.1 Dyadic cube subdivision

In this variant each cube is immediately subdivided into its 2^d subcubes, i.e. the partitions are given by

$$\mathcal{P}_l = \{\prod_{i=1}^{d} [k_i 2^{-l}, (k_i+1)2^{-l}], \quad k_i \in \{0, \dots, 2^l - 1\}\} \ .$$

This means that each node in the master tree has 2^d children.

2.3.2 Binary cube subdivision

The cubes are halved one dimension after another. I.e., the partitions are given by

$$\mathcal{P}_l = \{\prod_{i=1}^{d} [k_i 2^{-l_i}, (k_i+1)2^{-l_i}], \quad k_i \in \{0, \dots, 2^{l_i} - 1\}, \quad l_i = \lfloor \frac{l+d-i}{d} \rfloor\}$$

Note that

- the cells of the partitions are generally not cubes but only rectangles,
- the ordering of the hyperplanes which are used for the subdivision is not determined adaptively but is prescribed (otherwise one would not have a well defined master tree),
- in contrast to dyadic subdivision the underlying master tree is now a binary tree.

2.3.3 Bitstream Generation

Cube subdivision is particularly attractive because the generation of the strings $\mathbf{b}(x)$ is very simple. Given an input $x = (x_1, \dots, x_d) \in [0,1]^d$ we can write its components in binary representation as

$$x_i = \sum_{k=1}^{\infty} b_{ik} 2^{-k} \ .$$

In the case $x_i \neq 1$ is binary rational, we assume that the sequence $\{b_{ik}\}_k$ ends with zeros, while for $x_i = 1$ we have $b_{ik} = 1, k \in \mathbb{N}$. In both, dyadic and binary subdivision, the bitstream assigned to a point x is then

$$(b_{11}, b_{21}, \dots, b_{d1}, b_{12}, b_{22}, \dots, b_{d2}, \dots, b_{1L}, b_{2L}, \dots, b_{dL}).$$

In binary cube subdivisions we consider each single bit as a character, whereas in dyadic subdivision a character consists of d bits. I.e., the j-th character consists of the bits $b_{1j}, \dots, b_{dj}$ which in the sense of Section 2.2.1 can be considered as a binary representation of an integer in the range $0, \dots, 2^{d-1}$ corresponding to a certain enumeration of the children of a cell in a dyadic subdivision scheme.

2.4 Random Shifts

The above recovery schemes suffer from the following fact. For any two points $x, x' \in X$ the *tree distance* $\mathrm{dist}_{\mathcal{T}}(x, x')$ is the shortest path in the tree $\mathcal{T}(X)$ connecting the nodes $\Omega_{L,k}(x) \ni x$ and $\Omega_{L,k'} \ni x'$. Of course, whereas $||x-x'||$ may be arbitrarily small for any fixed norm $||\cdot||$ on $\mathbb{R}^d$, the tree distance $\mathrm{dist}_{\mathcal{T}}(x, x')$ could be $2L$. The above recovery scheme takes local averages of function values whose tree distance is small possibly omitting values for arguments that are geometrically very close. In fact, an adversary effect on the quality of the reconstruction is reflected by numerical experiments that will be shown later below. There are several possible remedies. Since the recovery scheme is very fast, the perhaps simplest one is to perform several different recoveries with respect to randomly slightly shifted coordinate systems and then take the average of the outputs.

In our implementation we scale the data to the interval $[0.3, 0.7]$, and then shift the data with random vectors in $[-0.3, 0.3]^d$. Let $\tilde{f}_\rho(x)$ denote the result of a query at x with the data shifted by the vector ρ and $X_\rho(x)$ be the corresponding set of training points in the leaf of the sparse occupancy tree containing x. Furthermore, let $R(x)$ be the set of shifts ρ for which the level of the evaluation is maximal. Then we have tested the following two schemes to compute a result from the random shifts:

$$\tilde{f}(x) = \frac{1}{\#(R(x))} \sum_{\rho \in R(x)} \tilde{f}_\rho(x) \tag{4}$$

or

$$\tilde{f}(x) = \frac{1}{\sum_{\rho \in R(x)} \#(X_\rho(x))} \sum_{\rho \in R(x)} \Big(\#(X_\rho(x)) \tilde{f}_\rho(x) \Big) \tag{5}$$

The idea of the second formula is to weight the points that occur in the sets $X_\rho(x)$ according to the number of their appearance in these sets. A third possibility is to choose just one of the $\rho \in R(x)$ randomly and to take this result. We usually prefer the first version.

3 Sparse Occupancy Trees using Simplices

As it has become clear from the above abstract description of a sparse occupancy tree, there are in principal no restrictions on the shape of the elements of the partitions. For the reasons that have been explained in the introduction we want to build trees based on simplices. This does not offer any advantages over cube subdivision, if we only consider piecewise constant approximations, but allows for extensions towards piecewise linear schemes, which will be the topic of Section 3.3. First, however, we have to go through some technicalities concerning data preparation and the computation of the bitstrings.

3.1 Data Preparation

To start a simplex subdivision scheme we have to map all the data points into a simplex. Here we choose the standard simplex

$$S = \{x \in \mathbb{R}^d : 0 \le x_1 \le x_2 \le \ldots \le x_d \le 1\}.$$

If one has data in the unit cube $[0,1]^d$, this can be achieved by the so-called *root transformation* $T : [0,1]^d \to S$:

$$x = (x_i)_{i=1}^d \longmapsto T(x) = \left(\prod_{j=i}^{d} x_j^{1/j} \right)_{i=1}^{d},$$

which can be computed recursively by

$$T(x)_d = x_d^{1/d}, \quad T(x)_i = T(x)_{i+1}\, x_i^{1/i}, \quad i = d-1, \ldots, 1,$$

and has the useful property that its Jacobian determinant $J_T(x) = \frac{1}{n!} = \text{const}$, see [FY94]. Furthermore, this transformation (and its inverse) are computationally cheap and numerically stable. However, since the transformation is singular on the boundary, it makes sense to scale the data initially to the cube $[0.125, 0.875]^d$.

Remark 6. *One should note, that this step is actually not without concern. Since the partial derivatives of the mapping T vary over a large range of magnitudes, the metric of the original data is effectively distorted. We assume that this might be the reason for the deterioration of the approximation quality we observe in some of our numerical experiments.*

3.2 Bitstream Generation

Next, we have to compute for any data point x its corresponding bitstream $\mathbf{b}(x)$. We start with the simplex $S = S(\emptyset)$ and initialize the bitstream $\mathbf{b} = \emptyset$. Then we proceed successively with the following bisection algorithm, where we assume that after some steps of subdivision x is contained in the simplex $S(\mathbf{b})$ with the vertices $v^j, j = 0, \ldots, d$. We assume that with respect to these vertices x has the barycentric coordinates $\tau(x, S(\mathbf{b})) = (\tau_0, \ldots, \tau_d)$ given by

$$x = \sum_{j=0}^{d} \tau_j v^j, \quad \sum_{j=0}^{d} \tau_j = 1 \ .$$

To perform one bisection step we choose two vertices v^k, v^l and subdivide the edge that connects these two vertices at its midpoint. Then we calculate the barycentric coordinates of x with respect to the two resulting subsimplices:

$$\begin{aligned} x = \sum_{j=0}^{d} \tau_j v^j &= \sum_{j=0, j\neq k,l}^{d} \tau_j v^j + \tau_k v^k + \tau_l v^l \\ &= \sum_{j=0, j\neq k,l}^{d} \tau_j v^j + 2\tau_k \left(\frac{v^k + v^l}{2} \right) + (\tau_l - \tau_k) v^l \\ &= \sum_{j=0, j\neq k,l}^{d} \tau_j v^j + (\tau_k - \tau_l) v^k + 2\tau_l \left(\frac{v^k + v^l}{2} \right) \ . \end{aligned}$$

If all barycentric coordinates of a point are in the range $[0,1]$, it can be concluded that x is in the interior of the simplex. Therefore, if $\tau_l < \tau_k$, then x is contained in

the subsimplex connected to the vertex v_l. In this case, we replace v^k by $\frac{1}{2}(v^k + v^l)$, τ_k by $2\tau_k$, τ_l by $(\tau_l - \tau_k)$, and add 0 to the bitstream $\mathbf{b}$. Else, if $\tau_l < \tau_k$, we replace v^l by $\frac{1}{2}(v^k + v^l)$, τ_l by $2\tau_l$, τ_k by $\tau_k - \tau_l$ and append 1 to $\mathbf{b}$. (In case $\tau_l = \tau_k$ both cases could be applied. In order to have uniqueness of the representation, the tie-break should be consistent and give always the same decision.) Then we proceed with the next subsection. It is important to note that only two coordinates are changed for each bisection and only one vertex is added to the total set of vertices produced in the course of the subdivision process.

The only task remaining is to determine what edge one subdivides in each step. For this purpose we use the following scheme that seems to be both efficient and to lead to favorable shapes of the intermediate simplices. We organize the vertices into two groups. One will consist of "old" vertices, which will be denoted by v^j, as above, and the other one will be comprised of "new" vertices w^j. The general form of an intermediate simplex arising in this process will be

$$S(\mathbf{b}) = \left[v^p, v^{p+1}, ..., v^q, w^{q-p+1}, w^{q-p+2}, ..., w^d\right],$$

where $0 \leq p \leq q \leq d$. The initial simplex $S(\emptyset)$ corresponds to $p = 0$, $q = d$ and has only old vertices. Every bisection will replace one of the old vertices v^p or v^q by the new vertex

$$w^{q-p} := \frac{1}{2}(v^p + v^q). \tag{6}$$

After each bisection the number of old vertices of the new simplex decrease by one. In analogy to dyadic subdivision, it takes d such bisections before ending up with an isotropic refinement of the initial simplex.

In case there is only one old vertex with index $p = q$, we declare the end of the level and start the next one by reassigning the names of the vertices as follows: $v^0 := v^p$ and $v^k := w^k$ for $k = 1, ..., d$ and continue with the procedure. In Figure 3.2 one dyadic subdivision cycle is graphically demonstrated for a three-dimensional configuration.

3.3 Piecewise Linear Approximation

On simplex partitions one can define piecewise linear approximations by prescribing values for all the vertices and interpolating the values of vertices connected to the cell a queries falls into. We have to note however, that in high dimensions it can hardly be the aim to achieve quadratic order of convergence - this would require that one could prescribe highly accurate values for the vertices which would be a very hard task. The real motivation for the development of such vertex schemes is that they provide a means to overcome the tree distance problem: it will not matter any more, if two points separate early in the occupancy tree because all training points spatially close to a query will contribute to the result via the vertices that connect neighboring cells. Similar to kernel methods the answer to a query will be a weighted average of training samples resulting in smoother approximants and therefore diminishing the variance of the approximation, a property that might be of interest, in particular, for regression problems. One can even think about constructing globally continuous approximants. The challenge hereby is how to deal with non-uniform, adaptive partitions and hanging nodes. However, we do not pursue this idea deeply in the current paper which is exclusively concerned with data interpolation.

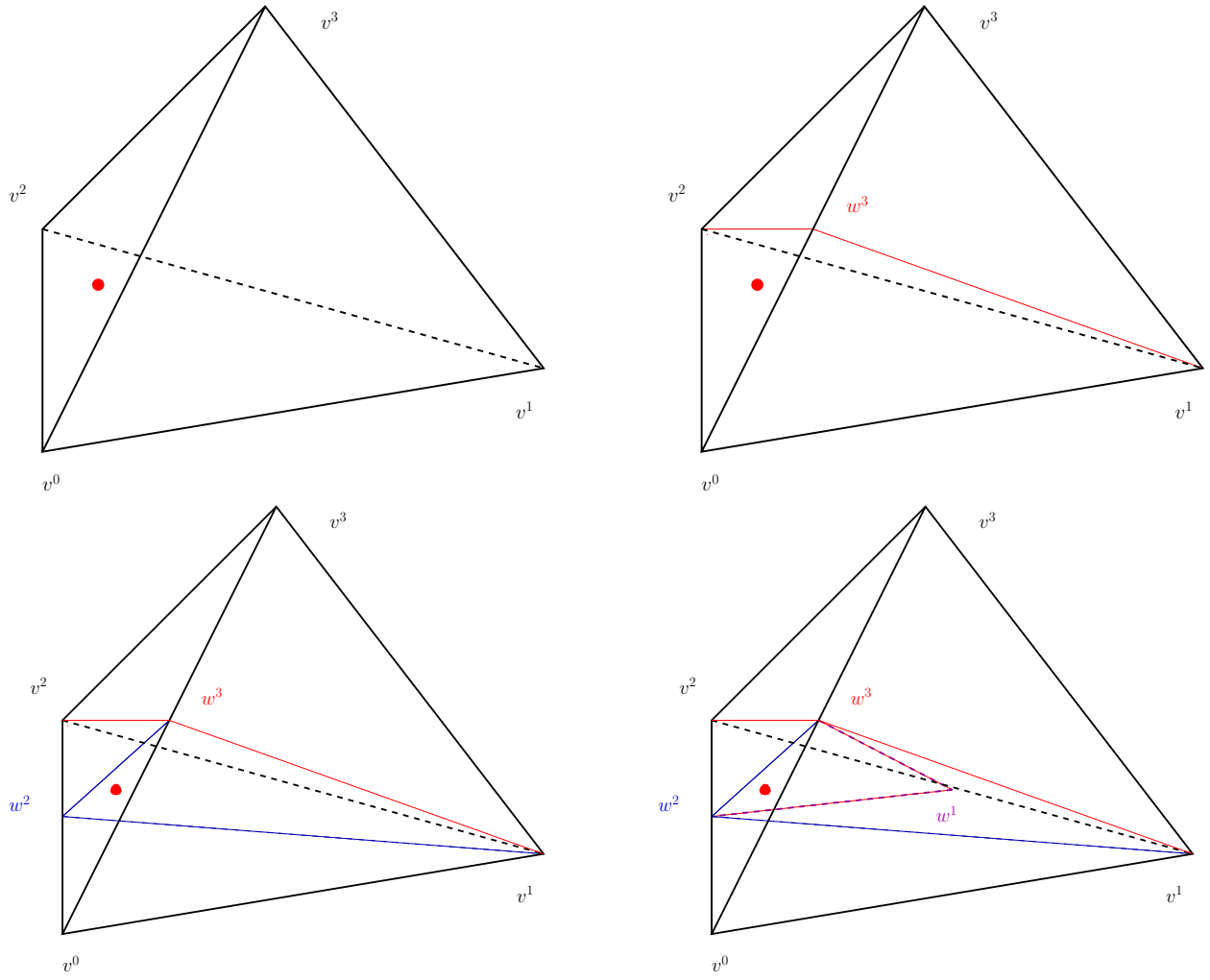

Figure 1: Top left: initial configuration. $S_\emptyset = [v^0, v^1, v^2, v^3]$, $p = 0, q = d = 3$, $w^{q-p} = w^3 = (v^0 + v^3)/2$. Top right: first binary subdivision step. $S_0 = [v^0, v^1, v^2, w^3]$, $p = 0$, $q = 2$, $w^{q-p} = w^2 = (v^0 + v^2)/2$. Bottom left: second binary subdivision step. $S_{0,1} = [v^1, v^2, w^2, w^3]$, $p = 1, q = 2$, $w^{q-p} = w^1 = (v^1 + v^2)/2$. Bottom right: last binary subdivion step. $S_{0,1,1} = [v^1, w^1, w^2, w^3]$, $p = 1, q = 1$, $w^{q-p} = w^0 = v^1$, reset.

In the following we will concentrate on the construction of a particular scheme which preserves the interpolation property of sparse occupancy algorithms because this method offers the best compromise between computational efficiency and accuracy in our numerical experiments. However, we will introduce some notations and ideas that allow for the development of other variants of piecewise linear schemes. The main choices in the construction are the design of the underlying occupancy tree, i.e. the depth of its branches which corresponds to the refinement of the underlying partition, and how to define values for the vertices, in particular, for the vertices that are not connected to occupied cells, but might be needed for the evaluation of a query.

3.3.1 Notation

To describe the vertex algorithms in detail we introduce some notation.

- For any d-dimensional simplex S we will denote the set of its $d+1$ vertices with $\mathcal{V}(S)$.
- If x is a point in S, and $v \in \mathcal{V}(S)$ then $\tau(S, v, x)$ is the barycentric weight of x with respect to v. These weights are defined by the equations

$$x = \sum_{v \in \mathcal{V}(S)} \tau(S, v, x) v, \qquad \sum_{v \in \mathcal{V}(S)} \tau(S, V, x) = 1 \ . \tag{7}$$

- We consider $S(\emptyset)$ to be the level 0 simplex. A level l simplex is a simplex that emerges from a level $l-1$ simplex by exactly d-binary subdivisions with the subdivision rule described in section 3.2, i.e., we base our linear approximation on a dyadic tree.
- With $S_l(x)$ we denote the level-l simplex of the master tree in which the data point x lies.
- Let $\mathcal{T}$ be an occupancy tree. Then the set of simplices on level l in this occupancy tree is denoted by $\mathcal{S}_l(\mathcal{T})$.
- A level-l vertex is a corner point of a level-l simplex. Note, that a vertex can belong to several levels. Furthermore

$$\mathcal{V}_l(\mathcal{T}) = \bigcup_{S \in \mathcal{S}_l(\mathcal{T})} \mathcal{V}(S) \tag{8}$$

  is the set of all level-l vertices connected to a level-l simplex of the tree $\mathcal{T}$.
- If $v \in \mathcal{V}_l(\mathcal{T})$, then $\mathcal{S}_l(\mathcal{T}, v) \subset \mathcal{S}_l(\mathcal{T})$ is the set of level-l simplices in the occupancy tree $\mathcal{T}$ which are adjacent to v.
- If in the subdivision process a vertex v emerges as average of the vertices w^1 and w^2 we write $w^1 = p_1(v)$ and $w^2 = p_2(v)$.

3.3.2 Principle of the Approximation

Let $\mathcal{T} = \mathcal{T}(X)$ be a finite simplex-based occupancy tree. This means that each node $\Omega_{k,l}$ in the tree contains a training point. For the moment we do not prescribe a certain maximum depth L for the tree and keep open the option that different branches may have different depth.

In the training stage of the piecewise linear approximation we compute for all levels l and all vertices $v \in \mathcal{V}_l(\mathcal{T})$ the value

$$y_l(v) = \mathcal{A}(\{y^i \,|\, x^i \in \bigcup_{S \in \mathcal{S}_l(\mathcal{T},v)} S\}). \tag{9}$$

If $v \notin \mathcal{V}_l(T)$ we define its (unweighted) value recursively by averaging the values of its parents:

$$y_l(v) = \frac{1}{2}(y_{l-1}(p_1(v)) + y_{l-1}(p_2(v))). \tag{10}$$

This recursion terminates, because the level-0 values of the the vertices of $S(\emptyset)$ are defined, if the training data set is not empty.

In the evaluation stage the level-l value of a query point x is then determined by piecewise linear interpolation of the vertex values of the level-l cell in *the master tree* the query falls into, concretely

$$\tilde{f}_l(x) = \sum_{v \in \mathcal{V}(S_l(x))} \tau(S_l(x), v, x)\; y_l(v). \tag{11}$$

Note, that the description of this algorithm only becomes complete, when we define how exactly we construct $\mathcal{T}$, i.e., how deep we refine the branches of the occupancy tree. Second, we have to decide, which of the various level-values $\tilde{f}_l(x)$ shall become the result $\tilde{f}(x)$ of the query. In particular, the above approximation is not necessarily continuous and not necessarily interpolating. However, this can be enforced by the right choice of $\mathcal{T}$ and l. Furthermore note, that this algorithm is suited for online-learning purposes: if one gets a new sample, one just computes its place in the occupancy tree and adds its value to all adjacent vertices. After that a query can immediately use the new information.

3.3.3 Special Schemes

As mentioned before we mainly aim at an interpolating scheme. This property can be enforced by choosing the underlying occupancy tree such that no two leaves of $\mathcal{T}$ join at a common vertex and an evaluation level l which is equal or larger than the level of the last occupied cell the query falls into. In the experiments of Section 4 we use a version that chooses l as the level of the smallest cell in the master tree still connected to at least one vertex of a cell in the occupancy tree. With these choices it is obviously guaranteed that a query coincident with a training point returns the value of this training point, because all vertices of the evaluation simplex are influenced by this training sample only. Typically, vertex separating trees become rather deep, which might become unpractical, if memory limitations have to be observed.

Therefore we also experimented with minimal separating trees, i.e., we chose $\mathcal{T}$ to be the smallest tree, such that each leaf of $\mathcal{T}$ contains only one training point. However, we observed a severe decrease of accuracy in this case, so that we disregarded this non-interpolating approach.

The easiest (but perhaps not best) method to enforce global continuity of the approximation is to perform all evaluations at the same level l. This would just mean to define a piecewise linear function on a uniform partition. The disadvantage of this simplistic approach is, that for highly non-uniformly distributed data, it requires frequent

use of vertex values that are not defined by training points nearby, but by the recursion (10). This, however, decreases the accuracy because it increases the probability that information is taken from data points which are far away from the query location.

3.3.4 Variants

Furthermore we have tested the following modification of the algorithm:

- Weighted vertex values: compute the vertex values not just by averaging but take the distances (or the barycentric weights) of the data points to the vertices into account.
- Best vertices: in the evaluation stage (Equation 11) do not sum up over all the vertices of the simplex, but only over the vertices that have been assigned values on level l:

$$g(x) = \frac{\sum_{v \in \mathcal{V}(S_l(x)) \cap \mathcal{V}_l(T)} b(S_l(x), v, x) g_l(v)}{\sum_{v \in \mathcal{V}(S_l(x)) \cap \mathcal{V}_l(T)} b(S_l(x), v, x).}$$

None of these modifications delivers a significant improvement in the numerical experiments we performed; therefore we do not present results for them.

4 Numerical Results

In this Section we demonstrate the performance of the above described schemes with some numerical results. As test cases we have chosen the examples designed by Friedman in [Fri91] since they are relatively well-known. A limitation of these examples is that the x-data is always supposed to be uniformly distributed. Since in many practical situations the input data is correlated or otherwise restricted to some submanifold of the formal input space, we have designed one test case of our own in order to cover this situation, too.

In all examples the setup is as follows: First, we generate a test data set of $M = 1,000,000$ points, and then various training data sets of N points, where $N = 10^e$ with $e = 3, 4, 5, 6$. This allows us to get some insight into the convergence behavior of the schemes. We measure the accuracy of the approximation using the root mean square error

$$RMSE = \sqrt{\frac{1}{M} \sum_{i=1}^{M} (\tilde{f}(x^i) - f(x^i))^2}$$

of the test set. Assuming that the x^i are independently drawn from a distribution ρ_X on $\mathbb{R}^d$, this is essentially a Monte-Carlo approximation of the weighted L_2-error $(\int_\Omega (\tilde{f}(x) - f(x))^2 \, d\rho_X)^{1/2}$.

It is clear that in literature (for instance [MLH03]) one easily finds methods like CART, support vector machines, or neural networks, which achieve better accuracy for the Friedman problems than the methods analyzed here. But these schemes are outside the scope of the current work because they use the distribution of the y-data in their training processes. The nearest neighbor and sparse occupancy tree recovery schemes described in this paper can be characterized as semi-adaptive schemes since they all use only the x-data in order to decide how to partition the input space. It

N	1,000	10,000,	100,000	1,000,000
k-nearest neighbor				
k				
1	3.45642	2.72654	2.16177	1.70666
5	2.5381	1.87911	1.41927	1.07411
10	2.51983	1.81674	1.34074	0.990133
20	2.6283	1.86574	1.35252	0.98068
opt-k	8	10	12	16
opt	2.50911	1.81674	1.33621	0.976907
Sparse Occupancy Trees				
Dyadic Cubes	4.63352	3.36706	3.14852	2.65411
Binary Cubes	4.41309	3.2352	2.30208	2.24182
Simplices	5.13561	4.04371	3.55441	3.22269
Random Shifts (Dyadic Cubes)				
10	3.27477	2.56207	1.74117	1.33433
50	3.18321	2.49552	1.62769	1.1639
100	3.11513	2.42343	1.67022	1.18687
Vertex Algorithm				
	3.18713	2.49601	1.9143	1.48756
No. Vertices	21,086	195,418	2,010,754	19,206,866

Table 1: Results for the ten-dimensional Friedman 1 example.

therefore seems appropriate to restrict the comparison to the relative performance of such related schemes.

4.1 Friedman 1 Data Set

In this test case we approximate the function

$$y(x_1, \ldots, x_{10}) = 10 \sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5. \tag{12}$$

Hereby the $x_1, \ldots, x_{10}$ are uniformly distributed over the ranges $0 \leq x_i \leq 1$. The variables $x_6, \ldots x_{10}$ clearly do not contribute to the y-values which causes a deterioration of the convergence, since the semi-adaptive schemes have no means to detect that these inputs are irrelevant. In order to quantify these effects, we repeat the experiment without the extra dimensions. In both cases the test set has a variance of 4.87664^2.

In Table 1 which displays the residual mean square errors calculated with nearest neighbors, sparse occupancy trees, random shifts and the interpolating piecewise linear vertex algorithm, one can make the following observations. The piecewise constant approximation with sparse occupancy trees clearly is not competitive with regard to approximation accuracy. However, random shifts significantly improve the performance. Applying a moderate number of random shifts clearly outperforms the 1-nearest neighbor method, although it is still clearly worse than the k-nearest neighbor method with an optimally chosen k. In this particular example binary splitting of cubes works

	single tree		100 random shifts	
level	queries	average error	queries	average error
1	383762	3.20158	0	
2	615339	2.24789	0	
3	899	1.2386	743241	1.08021
4	0		256363	1.4529
5	0		395	0.7784
6	0		1	0.593573

Table 2: Statistics of evaluation levels and corresponding errors.

significantly better than dyadic splitting. This is explained by the fact, that the superfluous variables come last in the splitting. That means that the superfluous splits in these directions have less influence on the tree structure and the averaging procedure. The approximation with piecewise constant approximation on simplices is significantly worse than the cube version confirming the suspicions we formulated about the data preparation step in Section 3.1.

It is clear that the accuracy of the answer to a single query depends significantly on the level on which the query is evaluated. This is also confirmed by Table 2 which shows on which level the queries were evaluated in the case $N = 10^6$. As already explained in the introduction, one cannot expect too much from any partitioning scheme in this particular example, because the data is uniformly distributed so that the training samples separate already on low levels. However, random shifts have a significant effect on this statistics. In this case the majority of evaluations is pushed from level 2 to level 3 and no evaluations are performed on level 1 or 2 any more.

For comparison Table 3 lists the results, if one removes $x_6, \dots x_{10}$ from the input data. As expected the difference between the dyadic and the binary tree algorithm becomes insignificant and the vertex algorithm achieves comparable accuracy as the nearest neighbor algorithm. Furthermore, in both Tables 1 and 3 one can observe that choosing a too large number of random shifts can lead to a slight deterioration of the residual. This behavior is similar to what one can observe when one lets k increase above the optimum in the k-nearest neighbor approximation.

4.2 Friedman 2 Data Set

In this case we approximate the function

$$y(x_1, \dots, x_4) = \sqrt{x_1^2 + \left(x_2 x_3 - \frac{1}{x_2 x_4}\right)^2}$$

with the four variables uniformly distributed over the ranges $0 \le x_1 \le 100$, $40\pi \le x_2 \le 560\pi$, $0 \le x_3 \le 1$, and $1 \le x_4 \le 11$. The function has variance 375^2.

This test case merely confirms the conclusions outlined in the previous sections. The last line of the table lists how many vertices have been assigned values in the training stage of the vertex algorithm. This information is relevant because the extensive memory consumption of the vertex algorithm seems to be its major disadvantage

N	1,000	10,000,	100,000	1,000,000
k-nearest neighbors				
k				
1	1.88274	1.17671	0.734954	0.460371
5	1.38843	0.809425	0.47469	0.28492
10	1.41282	0.792543	0.442179	0.253858
20	1.56399	0.848696	0.452497	0.24638
opt-k	6	8	12	17
opt	1.38235	0.789593	0.440825	0.245935
Sparse Occupancy Trees				
Dyadic Cubes	2.72086	1.67133	1.04174	0.708435
Binary Cubes	2.54749	1.66814	1.04311	0.644786
Binary Simplices	3.29683	2.14322	1.43075	0.958765
Random Shifts (Dyadic Cubes)				
10	1.54714	0.95336	0.626991	0.615749
50	1.49284	0.829795	0.571043	0.534478
100	1.52529	0.826601	0.5509	0.512936
Vertex Algorithm				
	1.54153	0.88528	0.511434	0.289492
No. Vertices	10,782	103,758	1,041,575	10,042,198

Table 3: Results for 5-dimensional Friedman 1 example.

N	1,000	10,000,	100,000	1,000,000
k-nearest neighbor				
k				
1	83.1134	45.6397	18.9724	13.9701
5	62.5134	30.4696	15.925	8.63483
10	65.5309	28.6937	14.2602	7.41368
20	76.067	30.1256	13.748	6.71658
opt-k	5	11	18	31
opt	62.5134	28.683	13.7265	6.58647
Sparse Occupancy Trees				
Dyadic Cubes	115.894	65.4261	37.8324	21.219
Binary Cubes	119.475	65.1913	35.9613	20.854
Simplices	142.638	84.5237	47.9805	27.8929
Random Shifts (Dyadic Cubes)				
10	75.4826	38.8928	19.7915	14.5742
50	66.2124	35.9743	19.5236	10.5921
100	65.304	35.257	19.6445	10.4952
Vertex Algorithm				
	65.244	29.7608	14.0904	7.02889
No. Vertices	8911	86303	837,773	8,225,310

Table 4: Results for Friedman 2 Example

N	1,000	10,000,	100,000	1,000,000
k-nearest neighbor				
k				
1	0.155238	0.103645	0.0692366	0.0439895
5	0.135107	0.0868966	0.0556544	0.0348372
10	0.140372	0.0905709	0.0570974	0.0355295
20	0.153082	0.0978829	0.0625607	0.0386026
opt-k	5	5	5	6
opt	0.135107	0.0868966	0.0556544	0.0347881
Sparse Occupancy Trees				
Dyadic Cubes	0.202478	0.116544	0.0811854	0.0602853
Binary Cubes	0.215195	0.118146	0.0843254	0.064841
Simplices	0.176661	0.11912	0.0887659	0.0491299
Random Shifts (Dyadic Cubes)				
10	0.140396	0.094879	0.0585261	0.0362686
50	0.139053	0.0922174	0.060494	0.0368731
100	0.140396	0.09131	0.0608876	0.0375257
Vertex Algorithm				
	0.0960549	0.0601811	0.0336853	0.0199746
No. Vertices	8,911	86,303	837,773	8,225,310

Table 5: Results for Friedman 3 Example

for its practical application. In this case the trained vertex tree needs about $8-9$ times more memory than the incoming data. However, this seems still more economical than storing, say, 50 or 100 randomly shifted occupancy trees, so that the vertex algorithm is surely more memory efficient than the random shift algorithm.

4.3 Friedman 3 Data Set

Here

$$y(x_1, \ldots, x_4) = \tan^{-1}\left(\frac{x_2 x_3 - (x_2 x_4)^{-1}}{x_1}\right)$$

with $0 \leq x_1 \leq 100$, $40\pi \leq x_2 \leq 560\pi$, $0 \leq x_3 \leq 1$, and $1 \leq x_4 \leq 11$. The variance of the test data set is 0.316525^2. Note that this function has a very steep gradient if $x_1 \to 0$ and almost jumps from $-\pi/2$ to $\pi/2$ when the numerator changes sign.

In Table 5 we see that the optimal number of nearest neighbors hardly (if at all) increases when N grows, indicating that the target function indeed is not very smooth. In this case the vertex algorithm performs even better than the optimal nearest neighbor algorithm. This can be explained by its interpolation property, which is very helpful here due to the nearly discontinuous behavior of the function. Furthermore, one is led to assume that the piecewise linear approximant improves the accuracy in regions of steep gradients.

N	1,000	10,000,	100,000	1,000,000
k-nearest neighbor				
1	0.629136	0.376805	0.218324	0.123483
5	0.547673	0.280663	0.143379	0.0765174
10	0.590996	0.293347	0.136615	0.0671632
20	0.648708	0.348594	0.147633	0.0644286
opt-k	3	5	9	18
opt	0.539988	0.280663	0.136575	0.0643436
Sparse Occupancy Trees				
Dyadic Cubes	0.66076	0.461315	0.284637	0.165377
Binary Cubes	0.699489	0.474227	0.28751	0.166394
Simplices	0.667239	0.502764	0.354782	0.230808
Random Shifts (Dyadic Cubes)				
10	0.638401	0.408558	0.233137	0.116529
50	0.584682	0.327386	0.186034	0.108537
100	0.583817	0.314226	0.174883	0.103498
Vertex Algorithm				
	0.501392	0.294926	0.150229	0.0667254
No. Vertices	53,271	516,167	5,085,728	50,918,786

Table 6: Results for Data on Submanifold Example

4.4 Data on Submanifold

In this example we consider samples of the smooth function $f(x) = \sin(2\pi(x_1+\cdots+x_d))$ for $d = 20$ and sample sites $\mathcal{X}$ that are uniformly distributed on a randomly chosen 5-dimensional sphere in $[0,1]^{20}$.

Standard estimates for piecewise constant approximation predict that for a smooth function $y = f(x)$ the L_2-error on a non-adaptive partition is proportional to $N^{-1/D}$, where D is the dimension of the manifold from which the samples are drawn. Therefore we compute the quantity

$$\tilde{D} = \frac{\log(N_2/N_1)}{\log(RMSE_1/RMSE_2)} \tag{13}$$

where N_1, N_2 are the numbers of points in two training data sets and $RMSE_1, RMSE_2$ are the calculated root mean square errors of the corresponding experiments with the same numerical scheme. In the current example we except, of course, values for $\tilde{D}$ around 5. Indeed, we observe values between 4.5 and 6.5 for the sparse occupancy trees, about 3.5 to 4.5 for the nearest neighbor algorithm and between 3 and 4.5 for the vertex algorithm. Even if these numbers are not too reliable, they clearly indicate that all the schemes indeed capture the submanifold and do not converge with the rate one would expect for a real 20-dimensional problem.

	50 Rand. Shifts	Vertices	k-NN	ANN
10-dim Friedman 1 Example				
Training	285s	112s	8s	8s
Evaluation	688s	157s	2439s	86s
RMSE	1.1639	1.48756	0.976907	1.16764
20-dim Sinus Wave				
Training	463s	437s	18s	18s
Evaluation	951s	354s	159,929s	6043s
RMSE	0.371329	0.221245	0.276657	0.277038

Table 7: Computational times (in seconds)

4.5 Comparison of CPU Times

To give an impression of computational efficiency of the above schemes and of its dependency on the space dimension, we reconsider the ten-dimensional Friedman 1 example from Section 4.1 and another artificial example where the the input data points x^i are uniformly distributed in 20-dimensional space and the function to be approximated is $f(x) = \sin(\sum_{k=1}^{d} x_k)$. Table 7 shows the computational times (in seconds) needed to process a training and a test data set of 10^6 points each on a computer with 2.3 GHz AMD Opteron processor. For the comparison with the k-nearest neighbors algorithm we use the kd-tree implementation provided by the ANN-library [MA10]. We used the same package to make a comparison with the approximate nearest neighbors method. In this case the distance between the query point and the i-th point returned by the search may exceed the distance between the query point and the true i-th nearest neighbor by a factor of $(1 + \varepsilon)$. In order to get some kind of fair comparison we tried to find parameters k and ε such that the approximation accuracy is about the same as for the random shift or the vertex method. In the 10-dim problem we finally used $k = 9$ and $\varepsilon = 4$, in the 20-dimensional example $k = 17$ and $\varepsilon = 1$.

In either case exact nearest neighbor search is not a practical option. The higher the dimension of input space becomes the less efficient nearest neighbor search becomes. The kd-tree obviously still perform well for 10 dimensions and in this case the approximate nearest neighbor method is preferable to the schemes presented here, but in the given 20-dimensional example the vertex algorithm produces better accuracy in less time. Moreover, Table 7 does not reflect that it is not clear beforehand what favorable values for the number of nearest neighbors and the relaxation parameter ε are; they have to be determined by some learning technique like cross validation. The vertex algorithm, on the other hand, does not have any tuning parameters and does not require prior knowledge.

5 Conclusion

The aim of this paper was to investigate several algorithms that might serve as efficient alternatives to the k-nearest neighbors approximation in high dimensions. These

algorithms are based on sparse occupancy trees and suited for large data sets and online learning. The algorithms scale well in high dimensions because the preprocessing and storage costs are at most proportional to d and $N \log N$ and evaluation costs are proportional to $\log N$. Hence, the computational costs do not depend exponentially on d as one can typically observe for (approximate) nearest neighbor methods. Simultaneously the approximation quality of the k-nearest neighbors method is preserved. In particular, the piecewise linear vertex approximation scheme seems to have potential for further improvements because already in its current, rather simple, implementation it outperforms the piecewise constant methods in various examples and there are several directions which one can search for improvements, notably with regard to the construction of globally continuous approximants.

Acknowledgments

This work has been supported in part by the National Science Foundation grant DMS-0721621, the Office of Naval Research/DEPSCoR contract N00014-07-1-0978, the Office of Naval Research/DURIP contract N00014-08-1-0996, the Army Research Office/MURI contract W911NF-07-1-0185, the Bulgarian Scientific Foundation grant contract #DO 02-102/23.04.2009, the TMR network "Wavelets in Numerical Simulation", and the Special Priority Program SPP 1324 funded by the German Research Foundation.

References

[BBD+on] Alexei Belochitski, Peter Binev, Ronald DeVore, Michael Fox-Rabinovitz, Vladimir Krasnopolsky, and Philipp Lamby. Tree-approximation of the long wave radiation parameterization in the NCAR CAM global climate model. Technical report, Interdisciplinary Mathematics Institut, University of South Carolina, Columbia, SC, 2010, in preparation.

[Fri91] Jerome H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–141, 1991.

[FY94] K.T. Fang and Y.Wang. *Number–theoretic method in statistics*. Chapman & Hall, 1994.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2nd edition, 2009.

[Ind04] Piotr Indyk. Nearest neighbor in high-dimensional spaces. In Goodman and O'Rourke, editors, *CRC Handbook of Discrete and Computational Geometry*, pages 877–892. CRC Press, 2nd edition, 2004.

[LMGY05] Ting Liu, Andrew W. Moore, Alexander Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 825–832. MIT Press, Cambridge, MA, 2005.

[MA10] David Mount and Sunil Arya. ANN: Approximate Nearest Neighbors, Version 1.1.2. University of Maryland, `http://www.sc.umd.edu/~mount/ANN`, 1997-2010.

[MLH03] David Meyer, Friedrich Leisch, and Kurt Hornik. The support vector machine under test. *Neurocomputing*, 55:169–186, 2003.