Journal of Computational and Applied Mathematics 235 (2011) 2063–2076

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Fast high-dimensional approximation with sparse occupancy trees

Peter Binev [a], Wolfgang Dahmen [b], Philipp Lamby [a,*]

[a] Interdisciplinary Mathematics Institute, University of South Carolina, 1523 Greene Street, Columbia, SC 29208, USA

[b] Institut für Geometrie und Praktische Mathematik, RWTH Aachen, Templergraben 55, 52056 Aachen, Germany

ARTICLE INFO

Article history:
Received 7 April 2010
Received in revised form 17 September 2010

MSC:
41A15
41A63

Keywords:
High-dimensional approximation
Non-parametric regression
Non-linear approximation
Multiresolution tree

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 $y^i \in \mathbb{R}^{d'}$, $i = 1, \ldots, N$, 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 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 are primarily 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*.

1. Introduction

Methods for high-dimensional function approximation and statistical learning are commonly categorized into two classes. For an overview we refer to [1]. On the one hand we have parametric methods which try to fit the function globally, typically prescribing the structure of the approximant as the combination of a fixed set of trial functions and learning their coefficients by optimizing some error norm. Here one can think, for example, of generalized additive models, projection pursuit or artificial neural networks. Recently, low rank tensor-product approximations have attracted a lot of research; see [2] and the references therein. 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. This might be a demanding and time-consuming process, thereby effectively limiting the size of the data that can be handled. Furthermore these approximations cannot easily be adjusted to new data, as in the case of incremental online learning or in applications where the domain in which the function is to be evaluated changes over time.

On the other hand we have non-parametric methods which try to fit the function 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 much from its mean or median 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 and to use their average as the approximate function value. At first glance this kind of memory-based learning

* Corresponding author. Tel.: +1 803 5766397; fax: +1 803 7776527.
E-mail addresses: binev@math.sc.edu (P. Binev), dahmen@igpm.rwth-aachen.de (W. Dahmen), lamby@math.sc.edu, lamby@mailbox.sc.edu (P. Lamby).

0377-0427/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2010.10.005

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 "what 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 in which 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 that could be achieved much more efficiently. Here we refer to [3,4] which gives 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 area from climatology in which problems of this size arise will be described in a subsequent report [5].

Therefore, in situations where a fast answer to a query 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 are primarily 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 is new. Instead of using the full tree $\mathcal{T}^*$ (called the *master tree*), we consider only a finite subtree $\mathcal{T}(X)$ whose nodes correspond to the cubes that are occupied, i.e., contain at least one element from the set X. A special indexing and ordering of these cubes allows us to store all the information about the tree using only $\mathcal{O}(\hat{L}dN)$ bits where $\hat{L}$ is a chosen upper limit for the number of dyadic 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 variety of problems of the above type. In this context we have to mention that if d is very large and the x-data is distributed uniformly in $\mathbb{R}^d$ one cannot expect a good performance either from the nearest neighbor scheme or from our schemes. 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 that provides a piecewise constant approximation. For a given point $x \in X$, it finds the finest cube $K(x)$ from $\mathcal{T}(X)$ containing the point. Then the approximated value at x could be set to be the mean value of the points from $K(x) \cap X$. The time for a single query will be shown to be $\mathcal{O}(d\hat{L}\log N)$. 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 X close to x in neighboring cubes. The latter is subject to the way the partition is set. Of course, ignoring the function values in a neighboring cube, even if its position 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 basic strategy described above. 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 gives a significant improvement.

In Section 3 we extend this technology to construct piecewise linear approximations on simplex subdivisions. Working with simplices offers a number of advantages, as it commonly does 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 containing it. Hence, the query response becomes an average of all training points in the neighborhood of the query coordinates, even including the points in simplices that are far away in tree distance. 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 delivers similar and sometimes even better accuracy than the nearest neighbor approximation. The point is, of course, that the latter might be unavailable in real applications if d and N become 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 hierarchy of nested partitions of Ω:

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

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 $\Omega_{l,k} \in \mathcal{P}_l$ is the disjoint 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 $\mathcal{I}_{l,k} \subset \mathcal{I}_{l+1}$ have a fixed cardinality, which is 2 in the case of binary subdivision or 2^d in the case of dyadic subdivision.

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 $r \in \mathcal{I}_{l,k}$.

A *sparse occupancy tree* $\mathcal{T}(X)$ is a connected, finite subtree of $\mathcal{T}^*$ consisting only of cells which are occupied, i.e., that contain at least one sample from the data site X:

$$\mathcal{T}(X) \subset \left\{\Omega_{l,k} \in \mathcal{T}^* : \Omega_{l,k} \cap X \neq \emptyset\right\}.$$

There are several possible criteria for choosing the depth of such a tree. For instance one can fix a maximum depth L in advance, i.e., $l \leq L$, where L might be chosen to meet a certain spatial resolution reflected by $\operatorname{diam}(\Omega_{L,k})$. Another possibility is to choose the occupancy tree such that the data points separate, which means that each leaf of $\mathcal{T}(X)$ contains at most one data point from X. Note that in this case not all branches of the occupancy tree necessarily have the same depth.

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

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

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, finite tree $\mathcal{T}(X) \cap \mathcal{T}(x)$:

$$\tilde{f}(x) = \mathcal{A}(\{y^i | x^i \in \mathcal{L}(\mathcal{T}(X) \cap \mathcal{T}(x))\}), \tag{1}$$

where we use the notation

$$\mathcal{A}(Y) = \frac{1}{\operatorname{card}(Y)} \sum_{y \in Y} y \tag{2}$$

to denote the average of a finite subset $Y \subset \mathbb{R}^{d'}$ and $\mathcal{L}(\mathcal{T})$ to denote the set of leaves of a tree $\mathcal{T}$.

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 one training sample.

2.2. Occupancy trees as sorted lists

As it turns out in practice, the occupancy trees are typically *highly vertical*, i.e., compared with the number of leaves they contain a large number of interior nodes. This makes a standard implementation based on node elements with pointers to their children somewhat inefficient, especially with large data sets. Instead, we represent the occupancy tree by a pointer-free data structure, namely as a sorted list of strings. A similar storage scheme was originally proposed in [6] for quadtrees and is sometimes used in computer science to store multiscale volume data. In this context it is known as a *linear octree* and known to be highly efficient and useful for parallel data processing [7,8]. Here we generalize it to arbitrary dimensions and arbitrary subdivision geometries.

2.2.1. Data structures

We define a string $\mathbf{b}$ to be a finite sequence of integers: $\mathbf{b} = (b_1, b_2, \ldots, b_l)$, where l is the length of the string. The elements b_i of a string will also be called *characters*. We denote with $(\mathbf{b}, c) = (b_1, \ldots, b_l, c)$ the string that results from appending an additional character to the sequence. If $j < l$ 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 $\mathcal{U}_{l,k}$. 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 injective. 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 l and $j < l$ then $\Omega(\mathbf{b}) \subset \Omega(\mathbf{b}|_j)$. In particular, for two strings $\mathbf{b}^1$,$\mathbf{b}^2$ of length $l > j$, if $\mathbf{b}^1|_j = \mathbf{b}^2|_j$, 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 $\Omega(\mathbf{b}^2)$.

2.2.2. Algorithm

The approximation defined by Eq. (1) can now be realized by the following algorithm that consists of a training stage and an evaluation stage. Here we assume that a maximum refinement level L is fixed and all branches of the sparse occupancy tree $\mathcal{T} = \mathcal{T}(X)$ are refined to this depth. Hence, given $x \in \Omega$ we can denote the string that is assigned to the finest level cell $\Omega_{L,k}$ which contains x with $\mathbf{b}(x)$: $x \in \Omega(\mathbf{b}(x)) \in \mathcal{L}(\mathcal{T})$.

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 partial sums

$$Y^n = \sum_{i=1}^{n} y^i = Y^{n-1} + y^n \in \mathbb{R}^d$$

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. If we assume that a character can be represented by H bits, the sorting operation can be done in $\mathcal{O}(HLN \log(N))$ time, since two strings can be compared in $\mathcal{O}(HL)$ time. The storage is HLN bits for the strings plus N integers for the permutation vector and $\mathcal{O}(d'N)$ real values for the partial sums. Below we introduce binary and dyadic trees and we typically prescribe the dyadic refinement level $\hat{L}$. For binary trees $H = 1$ and $L = d\hat{L}$ and for dyadic trees $H = d$ and $L = \hat{L}$. In any case the computational time and the storage requirements are proportional to d.

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 the 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 m such that $\mathbf{b}(x^m) < \mathbf{b}(x) \leq \mathbf{b}(x^{m+1})$, where $<$ denotes the relation induced by the lexicographical ordering.
2. Compare $\mathbf{b}(x)$ with $\mathbf{b}(x^m)$ and $\mathbf{b}(x^{m+1})$. The maximum number of leading characters is

$$j := \max\{j : \mathbf{b}(x)|_j = \mathbf{b}(x^m)|_j \vee \mathbf{b}(x)|_j = \mathbf{b}(x^{m+1})|_j\}.$$

3. Find the position p such that $\mathbf{b}(x^{p-1}) < \mathbf{b}(x)|_j \leq \mathbf{b}(x^p)$. Obviously x^p 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 $\mathcal{U}_{l,k}$. This is the last possible string that begins with $\mathbf{b}(x)|_j$. Then search the position q such that $\mathbf{b}(x^q) \leq \tilde{b} < \mathbf{b}(x^{q+1})$. Hence, x^q 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 partial sums:

$$\tilde{f}(x) = \frac{1}{q-p+1}(Y^q - Y^{p-1}).$$

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

Remark 4. Especially if the data sets are very large, the computation of the partial sums might become a source of numerical inaccuracies caused by cancellation or overflow. The remedy for this is to break the partial sums 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 most efficiently 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 be fixed by rescaling the training data X component-wise.

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

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

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

2.3.2. Binary cube subdivision

In this variant the cubes are halved one dimension after another, i.e., the partitions are given by

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

Note that in this case (a) the cells of the partitions are generally not cubes but only rectangles, (b) 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), and (c) in contrast to dyadic subdivision the underlying master tree is a binary tree.

2.3.3. Bitstream generation

Cube subdivision is particularly attractive because generating the strings $\mathbf{b}(x)$ is very simple. Given an input $x = (x_1, \ldots, x_d) \in [0, 1]^d$ 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

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

In binary cube subdivisions we consider each single bit as a character, whereas in dyadic subdivision a character consists of d bits. That is, the jth character consists of the bits $b_{1j}, \ldots, b_{dj}$ which in the sense of Section 2.2.1 can be considered as a binary representation of an integer in the range $0, \ldots, 2^{d-1}$ 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 adverse 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, perhaps the 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:

$$\tilde{f}(x) = \frac{1}{\#(R(x))} \sum_{\rho \in R(x)} \tilde{f}_\rho(x) \tag{3}$$

or

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

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 it 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

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

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

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

and has the useful property that its Jacobian determinant $J_T(x) = \frac{1}{n!} = \text{const}$, see [9]. 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 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 postulate 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 $\mathbf{b} = \emptyset$, i.e., compared to Section 2.2.1 we replace the letter Ω by S to emphasize that we are working with simplices now. 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

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

To perform one bisection step we choose two vertices v^k, v^l and subdivide the edge that connects them 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 $\frac{1}{2}(v^k + v^l)$,

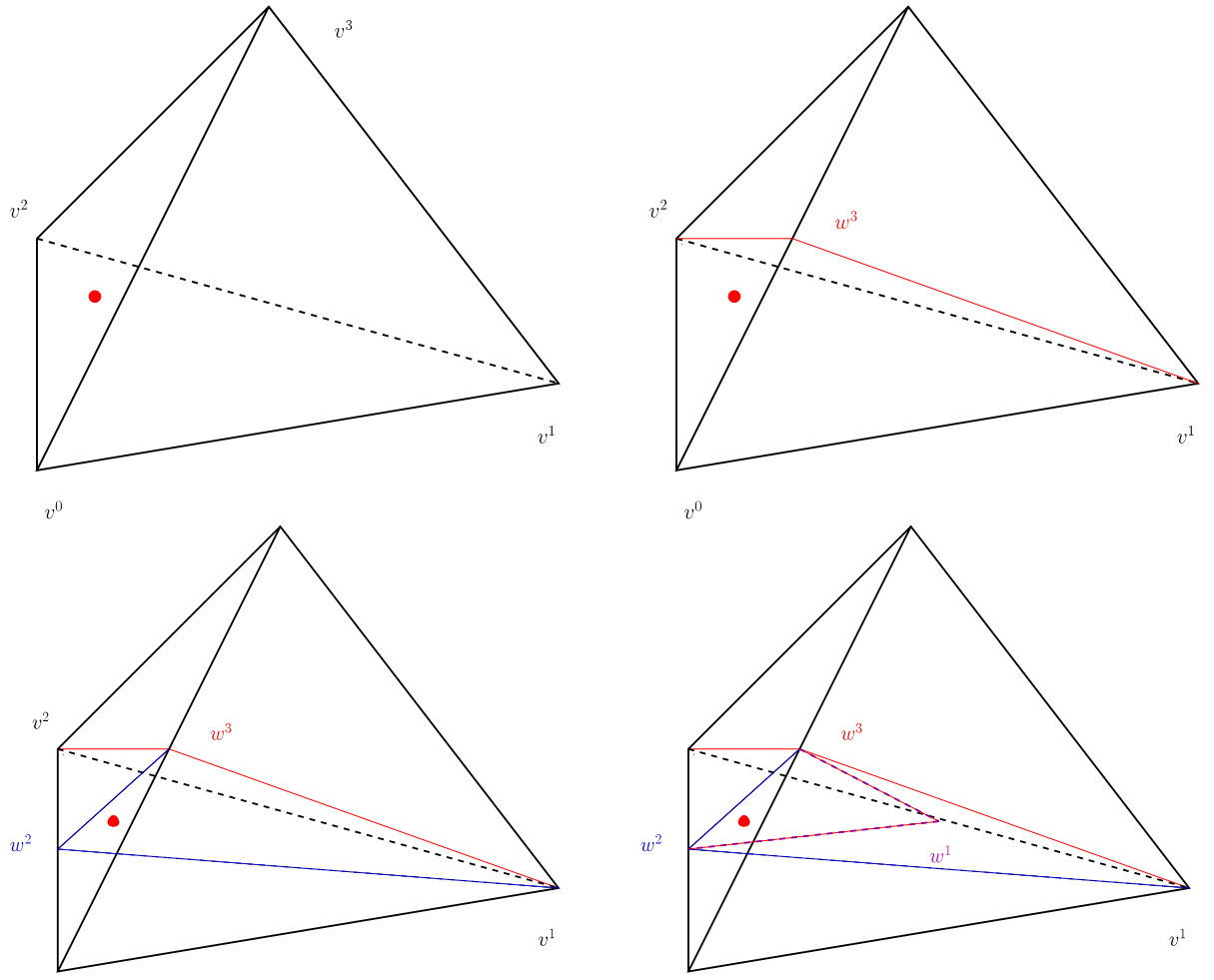

Fig. 1. Top left: initial configuration. $S_\emptyset = [v^0, v^1, v^2, v^3]$, $p = 0, q = d = 3$, $w^{q-p} = w^3 = (v^0 + v^3)/2$. Top right: first binary subdivision step. $S_0 = [v^0, v^1, v^2, w^3]$, $p = 0, q = 2$, $w^{q-p} = w^2 = (v^0 + v^2)/2$. Bottom left: second binary subdivision step. $S_{0,1}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

τ_k by $2\tau_k$, τ_l by $(\tau_l - \tau_k)$, and add 0 to the bitstream **b**. Otherwise, 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 **b**. (In case $\tau_l = \tau_k$ either case could be applied. In order to have uniqueness of the representation, the choice should be consistent, e.g., by always treating it as the first case.) Then we proceed with the next subsection. It is important to note that for each bisection, only two coordinates are changed and only one vertex is added.

The only task remaining is to determine what edge to subdivide in each step. For this purpose we use the following scheme, which with a different notation has been proposed in [10]. We organize the vertices into two groups. One consists of "old" vertices, which are denoted by v^j, as above, and the other consists of "new" vertices w^j. The general form of an intermediate simplex arising in this process is

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

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

$$w^{q-p} = \frac{1}{2}(v^p + v^q). \tag{5}$$

If v^p is replaced, then $p \to p + 1$, else if v^q is replaced, $q \to q - 1$.

When 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 $k = 1, \ldots, d$ and continue with the procedure. As proved in [10] this subdivision rule has the favorable property that simplices arising after the same number of binary subdivisions are congruent and that a simplex generated by one dyadic subdivision is geometrically similar to the original simplex. Fig. 1 shows one dyadic subdivision cycle 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 the vertices of the query-containing cell. We have to note however, that in high dimensions we do not aim to achieve a quadratic order of convergence—that would require prescribing 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 does not matter if two points separate early in the occupancy tree, because all training points spatially close to a query (i.e., in the same cell or in adjacent cells) will contribute to the result via the values of the vertices shared by their respective cells. As in kernel methods, the answer to a query is a weighted average of training samples, resulting in smoother approximants. Therefore, the variance of the approximation is reduced, which might be of interest, in particular, for regression problems. One can even think about constructing globally continuous approximants. The challenge in that 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.

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 at 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 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

$$x = \sum_{v\in\mathcal{V}(S)} \tau(S, v, x)v, \qquad \sum_{v\in\mathcal{V}(S)} \tau(S, v, x) = 1. \tag{6}$$

- 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

$$\mathcal{V}_l(\mathcal{T}) = \bigcup_{S\in\mathcal{S}_l(\mathcal{T})} \mathcal{V}(S) \tag{7}$$

  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 share v as a corner point.
- 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 we keep open the option that different branches may have different depths.

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

$$y_l(v) = \mathcal{A}\left(\left\{y^i \mid x^i \in \bigcup_{S\in\mathcal{S}_l(\mathcal{T},v)} S\right\}\right). \tag{8}$$

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

$$y_l(v) = \frac{1}{2}(y_{l-1}(p_1(v)) + y_{l-1}(p_2(v))). \tag{9}$$

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

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

$$\tilde{f}_l(x) = \sum_{v \in \mathcal{V}(S_l(x))} \tau(S_l(x), v, x) y_l(v). \tag{10}$$

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, these properties can be enforced by the right choice of $\mathcal{T}$ and l. Furthermore note that this algorithm is suited for online-learning purposes: to incorporate a new sample, just compute its place in the occupancy tree and add its value to all adjacent vertices. After that a query can immediately use the new information.

3.3.3. Special schemes

As mentioned before, our main aim is that the scheme be 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 by choosing 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 are rather deep, which may make them impractical depending on available memory.

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 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 (9). This 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 (Eq. (10)) do not sum up over all the vertices of the simplex, but only over the vertices that have been assigned values on level l:

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

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 in [11] 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 $M = 10^6$ points, and then various training data sets of N points, where $N = 10^e$ with $e = 3, 4, 5, 6$. This allows 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 $(\int_\Omega (\tilde{f}(x) - f(x))^2 \mathrm{d}\rho_X)^{1/2}$.

It is clear that in literature, for instance [12], one easily finds methods like CART, support vector machines, neural networks or low rank tensor product approximations [13], which achieve better accuracy for the Friedman problems than

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

N	10^3	10^4	10^5	10^6
k-nearest neighbors 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	2010,754	19,206,866

Table 2
Statistics of evaluation levels and corresponding errors.

Level	Single tree		100 random shifts	
	Queries	Average error	Queries	Average error
1	383,762	3.20158	0	
2	615,339	2.24789	0	
3	899	1.2386	743,241	1.08021
4	0		256,363	1.4529
5	0		395	0.7784
6	0		1	0.593573

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 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

$$y(x_1, \ldots, x_{10}) = 10 \sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5. \tag{11}$$

Here the $x_1, \ldots, x_{10}$ are uniformly distributed over the ranges $0 \le x_i \le 1$. 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 y-values of the test set have a variance of 4.87664^2.

Table 1 displays the residual mean square errors calculated with nearest neighbors, sparse occupancy trees, random shifts and the interpolating piecewise linear vertex algorithm, which leads to 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 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 $N = 10^6$.

Table 3
Results for 5-dimensional Friedman 1 example.

N	10^3	10^4	10^5	10^6
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	1041,575	10,042,198

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 at low levels. However, random shifts have a significant effect on this statistic. In this case the majority of evaluations occur on level 3 rather than level 2, and no evaluations are performed on level 1 or 2 anymore.

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

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 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.

4.2. Friedman 3 data set

Here

$$y(x_1, \ldots, x_4) = \tan^{-1}\left(\frac{x_2x_3 - (x_2x_4)^{-1}}{x_1}\right)$$

with $0 \le x_1 \le 100$, $40\pi \le x_2 \le 560\pi$, $0 \le x_3 \le 1$, and $1 \le x_4 \le 11$. The 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 4 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, the results indicate that the piecewise linear approximant improves the accuracy in regions of steep gradients.

4.3. 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 X that are uniformly distributed on a randomly chosen 5-dimensional sphere of radius $1/2$ in $[0, 1]^{20}$ (Table 5).

General standard estimates, which actually also apply to our schemes, predict that for a smooth function and uniformly distributed samples in $\mathbb{R}^d$ the L_2-approximation error on a non-adaptive partition is proportional to $N^{-1/d}$. Here, however, the data is given on a $D = 5$-dimensional submanifold and one would hope that the convergence is proportional to $N^{-1/D}$. To check this, we compute the quantity

Table 4
Results for Friedman 3 example.

N	10^3	10^4	10^5	10^6
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	8911	86,303	837,773	8225,310

Table 5
Results for data on submanifold example.

N	10^3	10^4	10^5	10^6
k-nearest neighbor k				
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	5085,728	50,918,786

$$\tilde{D} = \frac{\log(N_2/N_1)}{\log(RMSE_1/RMSE_2)} \tag{12}$$

where N_1, N_2 are the numbers of points in two training data sets and $RMSE_1, RMSE_2$ are the root mean square errors of the corresponding experiments with the same numerical scheme. 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 the promise from the introduction, namely that the multiresolution structure captures the structure of the input data, is fulfilled.

4.4. Comparison of CPU times

In this section we give an impression of the computational efficiency of the above-described schemes and their dependency on both the space dimension d as well as the distribution of the data. Here we compare our methods with the approximate nearest neighbor method, because, as mentioned in the introduction, this method is conceptually similar

Table 6
CPU times for uniformly distributed data in $[0, 1]^d$.

d	10 random shifts		Vertices		ANN ($\varepsilon = 2$)	
	Training	Evaluation	Training	Evaluation	Training	Evaluation
5	31	100	47	102	6	58
10	42	104	117	177	8	196
15	58	142	217	280	14	438
20	69	165	328	433	22	1273
25	86	170	382	496	26	2981
30	110	203	514	605	38	8617

Table 7
CPU Times for data distributed on a D-dimensional sphere in $\mathbb{R}^{20}$.

D	10 random shifts		Vertices		ANN ($\varepsilon = 2$)	
	Training	Evaluation	Training	Evaluation	Training	Evaluation
4	70	129	262	567	49	101
8	71	162	266	529	31	291
12	72	155	306	554	9	869
16	73	174	374	547	11	2073

and therefore a natural benchmark. The common idea of all these algorithms is to compute averages of nearby points in the input space, i.e., the algorithms do not make any use of the y-values of the training samples during the partitioning of the input space. Hence, the CPU-times depend only on the distribution of the training points in Ω. Furthermore, the previous experiments have shown that the relative accuracies of the different schemes depend on the specific function f to be approximated: in Section 4.1 the nearest neighbor method was preferable, whereas in Section 4.2 the vertex approximation was more accurate. Therefore, in the context of the current paper, it does not make much sense to consider how much CPU-time is needed to achieve a certain accuracy: such results heavily depend on the specific properties of the function to be approximated, which is application dependent. Here we want to demonstrate another point: namely, that considering CPU-time the sparse occupancy algorithms scale favorably compared to approximate nearest neighbors if the data is distributed on a high-dimensional manifold.

Therefore the following two tables show 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 approximate k-nearest neighbors algorithm, we use the implementation provided by the ANN-library [14]. In this case the distance between the query point and the ith point returned by the search may exceed the distance between the query point and the true ith nearest neighbor by a factor of $(1 + \varepsilon)$. Note that increasing ε makes the neighbor search easier and faster, but also decreases the accuracy of the approximation. All examples used $k = 20$ and $\varepsilon = 2$, which we found to be a good compromise in some applications.

Table 6 shows that both sparse occupancy tree methods, based on cube subdivision or vertex averages, show the approximately linear dependence on the space dimension predicted in Section 2.2.2. The approximate nearest neighbor method, however, shows some kind of super-linear complexity which confirms a known rule of thumb that approximate nearest neighbor search is efficient only for moderate dimensions up to, say, 20.

We should mention that uniformly distributed data is not the most important case in practice. Therefore it is of some interest to research situations like the one sketched in Section 4.3, where the data is concentrated on a low dimensional submanifold of the input space. In Table 7 we see that the efficiency of the approximate nearest neighbor mostly depends on the dimension D of the submanifold: if D is small, ANN is very efficient even though d might be large. On the other hand, the CPU-times of the sparse occupancy tree algorithms are almost constant, because they do not depend on D but only on d, which is evident from the description of the algorithm.

Finally, we emphasize that the tables do not reflect that favorable values for the number k of nearest neighbors and the relaxation parameter ε are not known beforehand, but have to be determined by some learning technique like cross validation. The sparse occupancy tree algorithms, on the other hand, do not have any tuning parameters and do not require prior information about the data.

5. Conclusion

The aim of this paper is 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 are 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 $d \log N$. 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.

Comparing the approximate nearest neighbor search with the piecewise linear vertex scheme, one can formulate a rough rule of thumb as follows: the former works very well in situations where the target function has relatively low variance and actually depends only on a subset of the input variables (since it delivers accuracy nearly as good as exact nearest neighbors in these cases) or if the data is concentrated around a low dimensional subset of the input space (because in this case the efficiency of the neighbor search is good). In these cases the difficulty of the approximation problem is reduced by some favorable properties of the data. However, the new vertex scheme outperforms the nearest neighbor methods in the more demanding, truly high-dimensional applications. In the case of strongly varying target functions f, the vertex scheme often delivers better accuracy. Furthermore, if the dimension of the manifold containing the data is large, the vertex scheme is preferable since it will be computationally more efficient than the approximate nearest neighbor search. The latter might even be unfeasible in such situations.

In our opinion, the piecewise linear vertex approximation scheme has the most 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.

Acknowledgements

We are very indebted to Ron DeVore for his continuous support of this research through numerous valuable discussions concerning approximation and mathematical learning theory. In particular, he has introduced us to applications in meteorology which have been a major motivation for this work.

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

[1] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd ed., Springer, 2009.
[2] T. Kolda, B. Bader, Tensor decompositions and applications, SIAM Review 51 (3) (2009) 455–500.
[3] P. Indyk, Nearest neighbor in high-dimensional spaces, in: Goodman O'Rourke (Ed.), CRC Handbook of Discrete and Computational Geometry, 2nd ed., CRC Press, 2004, pp. 877–892.
[4] T. Liu, A.W. Moore, A. Gray, K. Yang, An investigation of practical approximate nearest neighbor algorithms, in: L.K. Saul, Y. Weiss, L. Bottou (Eds.), Advances in Neural Information Processing Systems 17, MIT Press, Cambridge, MA, 2005, pp. 825–832.
[5] A. Belochitski, P. Binev, R. DeVore, M. Fox-Rabinovitz, V. Krasnopolsky, P. Lamby, Tree-approximation of the long wave radiation parameterization in the NCAR CAM global climate model, Tech. rep. 10:08, Interdisciplinary Mathematics Institute, University of South Carolina, Columbia, SC (2010).
[6] I. Gargantini, An effective way to represent quadtrees, Communications of the ACM 25 (12) (1982) 905–910.
[7] H. Sundar, R. Sampath, G. Biros, Bottom-up construction and 2:1 balance refinement of linear octrees in parallel, SIAM Journal on Scientific Computing 30 (5) (2008) 2675–2708.
[8] C. Burstedde, L.C. Wilcox, O. Ghattas, `p4est`: scalable algorithms for parallel adaptive mesh refinement on forests of octrees, SIAM Journal on Scientific Computing (2010) (submitted for publication).
[9] K. Fang, Y. Wang, Number–theoretic Method in Statistics, Chapman & Hall, 1994.
[10] J. Maubach, Local bisection refinement for N-simplicial grids generated by reflection, SIAM Journal of Scientific Computing 16 (1) (1995) 210–227.
[11] J.H. Friedman, Multivariate adaptive regression splines, The Annals of Statistics 19 (1) (1991) 1–141.
[12] D. Meyer, F. Leisch, K. Hornik, The support vector machine under test, Neurocomputing 55 (2003) 169–186.
[13] J. Beylkin, Gregory Garcke, M.J. Mohlenkamp, Multivariate regression and machine learning with sums of separable functions, SIAM J. Sci. Comp. 31 (3) (2009) 1840–1857.
[14] D. Mount, S. Arya, ANN: Approximate Nearest Neighbors, Version 1.1.2, University of Maryland, http://www.sc.umd.edu/~mount/ANN (1997–2010).