Multiple point evaluation on combined tensor product supports

Numer Algor (2013) 63:317–337
DOI 10.1007/s11075-012-9624-4

ORIGINAL PAPER

Multiple point evaluation on combined tensor product supports

R. Hiptmair · G. Phillips · G. Sinha

Received: 20 November 2011 / Accepted: 9 July 2012 / Published online: 26 July 2012

Abstract We consider the multiple point evaluation problem for an n-dimensional space of functions $[-1,1[^d \mapsto \mathbb{R}$ spanned by d-variate basis functions that are the restrictions of simple (say linear) functions to tensor product domains. For arbitrary evaluation points this task is faced in the context of (semi-)Lagrangian schemes using adaptive sparse tensor approximation spaces for boundary value problems in moderately high dimensions. We devise a fast algorithm for performing $m \geq n$ point evaluations of a function in this space with computational cost $O(m \log^d n)$. We resort to nested segment tree data structures built in a preprocessing stage with an asymptotic effort of $O(n \log^{d-1} n)$.

Keywords (Multilevel) segment tree · Adaptive sparse tensor product approximation

1 Introduction

We fix the dimension $d \in \mathbb{N}$ and denote by V a vector space of real valued functions $[-1,1[^d \mapsto \mathbb{R}$ on the d-dimensional hypercube. The main specimens

R. Hiptmair (✉)
SAM, ETH Zürich, 8092 Zürich, Switzerland
e-mail: hiptmair@sam.math.ethz.ch

G. Phillips
Neue Kantonsschule Aarau, 5000 Aarau, Switzerland
e-mail: gisela.phillips@nksa.ch

G. Sinha
Department of Mathematics, California Institute of Technology,
Pasadena, CA 91125, USA
e-mail: gauravsinha420@gmail.com

are provided by spaces of multivariate polynomials. We make the fundamental assumption that for any $\varphi \in V$ and $\boldsymbol{x} \in [-1, 1[^d$ the point evaluation $\varphi(\boldsymbol{x})$ can be accomplished with less than $W_x \in \mathbb{N}$ work units, where W_x may strongly depend on d, however. As a work unit we regard a single comparison, branching, or arithmetic operation. Counting work units will be our main gauge for computational effort. Yet, on modern computer architectures this may not be directly related to computing time.

For some (large) $n \in \mathbb{N}$ we are given *arbitrary* sequences of points $\boldsymbol{a}_k, \boldsymbol{b}_k \in [-1, 1[^d$, $k = 1, \ldots, n$, with $\boldsymbol{a}_k < \boldsymbol{b}_k$, where "$<$" is understood in a componentwise sense. The characteristic function of a non-degenerate tensor product box with corners $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d$, $\boldsymbol{a} < \boldsymbol{b}$, is defined according to

$$\chi_{\boldsymbol{a},\boldsymbol{b}}(\boldsymbol{x}) := \begin{cases} 1 & \text{, if } a_i \le x_i < b_i\,, \quad i = 1, \ldots, d\,, \\ 0 & \text{elsewhere,} \end{cases} \qquad \boldsymbol{x} \in [-1, 1[^d\,.$$

Based on another sequence $(\varphi_1, \ldots \varphi_n) \in V^n$ we introduce the linear combination of basis functions with tensor product supports

$$\Psi(\boldsymbol{x}) := \sum_{k=1}^{n} \varphi_k(\boldsymbol{x}) \cdot \chi_{\boldsymbol{a}_k,\boldsymbol{b}_k}(\boldsymbol{x})\,, \quad \boldsymbol{x} \in [-1, 1[^d\,. \tag{1}$$

The summands will be called box (supported) functions in the sequel.

The following computational task addresses the multiple evaluation of Ψ in many points.

Task 1 *Given $m \ge n$ points $\boldsymbol{x}_k \in [-1, 1[^d$, $k = 1 \ldots, m$, compute the m values $\Psi(\boldsymbol{x}_k)$.*

A naive implementation that relies on the straightforward summation of (1) requires an asymptotic computational effort of $O(W_x mn)$.[1]

In this article we propose data structures and an algorithm that allow to perform the evaluations of Task 1 with computational cost $O(m \log^d n)$ for $m, n \to \infty$, which means a considerable acceleration for large n, m. This can be achieved through a preprocessing step involving an effort of $O(n \log^{d-1} n)$, see Proposition 4.3. This reduces the cost of a single point evaluation to $O(W_x \log^d n)$, see Proposition 5.2. We acknowledge that the constants in the estimates may depend on d and will usually do so in an exponential fashion. This is acceptable, because storing a single function $\varphi \in V$ will usually take p^d bits for some $p > 1$. For example, if V is the space of multi-linear functions, we need 2^d coefficients to characterize any $\varphi \in V$.

A special case of functions represented as sums like (1) occurs in the context of sparse adaptive tensor discretizations also known as adaptive sparse grids.

[1] As usual $O(W_x mn)$ means a bound of the form $CW_x mn$ with the constant $C > 0$ independent of W_x, m, and n for all admissible values of W_x, m, and n.

There, V will be the $d+1$-dimensional space of multi-linear functions $\mathbb{R}^d \mapsto \mathbb{R}$, and the corner points are taken from a special set of nodes of hierarchical tensor product meshes. More precisely, we have for some maximal "level" $L \in \mathbb{N}$

$$\boldsymbol{a}_k, \boldsymbol{b}_k \in \left\{ \left(i_j 2^{-l_j}\right)_{j=1}^{d} : l_j = 0, \ldots, L,\ i_j \in \left\{-2^{l_j}, \ldots, 2^{l_j}\right\},\ i_j \text{odd} \right\} .$$

For more information about (adaptive) sparse tensor product spaces and their use to break the so-called "curse of dimensionality" in the approximation of solutions of moderately high-dimensional boundary value problems we refer to [1] and [8].

Usually, sparse grid functions need not be evaluated at arbitrary points. The exception are transport problems tackled by means of so-called semi-Lagrangian schemes, see, e.g., [6]. These methods follow the trajectories of a flow field over a short time to determine interpolation points. These can be located anywhere, if general flow fields are admitted. The semi-Lagrangian approach in combination with adaptive sparse grid spaces offers a promising numerical technique for moderately high-dimensional boundary value problems arising in areas as diverse as optimal control [4] and kinetic equations [2]. This has motivated the present article.

We point out that it is the very setting of *adaptive* sparse grids for which we developed the new algorithm. Then we may encounter the situation that the supports of all n basis functions may have an non-empty overlap. Then straightforward point evaluation inside this overlap region will incur $O(Ln)$ computational cost. Conversely, for a *regular sparse grid*, only $O(\log^d n)$ basis functions contribute to $\Psi(\boldsymbol{x})$ and naive summation of (1) becomes a competitive option.

Remark 1.1 In order to demonstrate the gist of the algorithms, we resort to pseudo-codes with a syntax borrowed from C++ and the standard template library (STL) [7]. Yet, we emphasize, that the code snippets enclosed in this article are "pseudo-code". They are bare bones and for the sake of lucidity were neither intended to be syntactically correct nor to comply with best practices of proper object oriented implementation.

2 Basic data structures

We rely on the class Interval that supports the usual operations (on one-dimensional bounded, half-open intervals $\subset \mathbb{R}$) like a point enclosure query method **bool** contains(**double**), an intersection test **bool** intersect(**const** Interval &I1,**const** Interval &I2) and an inclusion test **bool** contains(**const** Interval &subI). The function Interval merge(**const** Interval &I1,**const** Interval &I2) creates a new Interval object that combines two *adjacent* intervals into one.

A d-dimensional bounded tensor product domain, a "box", can be encoded by a sequence of d intervals, which suggests the data type

typedef vector<Interval> Box .

Thus a single term in (1) corresponds to an object of type BoxFunction, whose definition is given in Listing 1. The operator member function **operator** [](**int**), given an argument $i \in \{0, \ldots, d-1\}$ serves to access $[a_{i+1}, b_{i+1}[$, when the box data field stores $[a_1, b_1[\times \cdots \times [a_d, b_d[$. The phi data field of BoxFunction provides the φ_k component of a term in (1). It contains an object of type VFunction that stores a function $\in V$. It is supposed to provide the usual (real) vector space operations through overloaded arithmetic operators $+,-,*,/,+=,-=,*=,/=$. In the sequel, we write W_s for an upper bound on the computational cost of a binary operation of any two objects of type VFunction.

Listing 1 Class definition for a function with tensor product support

```
1  class BoxFunction {
2  public:
3    Box box;
4    VFunction phi;
5
6    BoxFunction(const Box &b,const VFunction &f);
7    int dim(void) const { return box.size(); }
8    const Interval &operator [](int) const;
9  };
```

A sum of the form (1) can be represented as an object of type

typedef list <BoxFunction> BoxFnSeq; .

Our algorithm expects an input of this type, but it could as well operate on suitable read-only iterator ranges.

3 Segment trees

A one-dimensional *segment tree* is a balanced binary search tree that can be used to answer the point enclosure query for a collection of intervals efficiently, see [3, Ch. 10] and [5, Section 2.2]. Here, we briefly review data structures, algorithms, and complexity issues connected with this fundamental concept from computational geometry.

The one-dimensional point enclosure problem reads as follows: given a collection of intervals

$$\{[a_k, b_k[:\ -1 \le a_k < b_k \le 1,\ k = 1, \ldots, n\}\,, \quad n \in \mathbb{N}\,,$$

and a point $\xi \in [-1, 1[$, find those intervals that contain ξ. A straightforward implementation will take $O(n)$ comparisons to arrive at an answer. However, once the corresponding segment tree has been constructed with $O(n \log n)$ cost, the point query can be answered with computational effort $O(\log n + K)$, where K is the number of intervals reported, see [3, Ch. 10].

The nodes of a segment tree possess a so-called *comparison interval* as key data field, see the class definition of SegTreeNode in Listing 2.

Listing 2 Data type for node of a segment tree

```
class SegTreeNode {
public:
  SegTreeNode *leftson,*rightson; // tree structure fields
  const Interval compintv;        // comparison interval
  list<const BoxFunction &> loclist;  // list of box functions
  // Data fields discussed in Section 4
  SegTreeNode *subtreeroot;
  WFunction locfun;

  SegTreeNode(const Interval &I,SegTreeNode
      *ls=NULL,SegTreeNode *rs=NULL);
};
```

Another important data field of SegTreeNode is the local list loclist of box functions. Its actual significance will be explained in the next section. For the time being we remark that, for a fixed coordinate direction $1 \le i \le d$, the i-th interval of the tensor product support of every function stored in loclist will contain the comparison interval of the node, cf. the discussion of the function **registerInterval** from Listing 4.

The first pass of the construction of segment trees for Task 1 is executed by the function **buildSegTree** of Listing 3, cf. [5, Algorithm 2.3].

Listing 3 Building a one-dimensional segment tree

```
SegTreeNode *buildSegTree(int i,const BoxFnSeq &fseq) {
  vector<double> bd(2); bd[0] = -1.0; bd[1] = 1.0;
  foreach f in fseq {
    const Interval &intv(f[i-1]);
    bd.push_back(intv.a); bd.push_back(intv.b);
  }
  // Sort vector bd and eliminate duplicate elements
  sort(bd); unique(bd);
  list<SegTreeNode *> t;
  for(j=0;j<bd.size()-1;j++)
    t.push_back(new
      SegTreeNode(Interval(bd[j],bd[j+1])));
  int n_sons;
  while ((n_sons = t.size())>1) {
```

```
14      int n_parents = n_sons/2;
15      for(int i=0;i<n_parents,i++) {
16          SegTreeNode *ls = t.front(); t.pop_front();
17          SegTreeNode *rs = t.front(); t.pop_front();
18          Interval parentintv =
                merge(ls->compintv,rs->compintv);
19          t.push_back(new SegTreeNode(parentintv,ls,rs));
20      }
21      if (n_sons > n_parents*2) {
22          // In case of an odd number of intervals
23          t.push_back(t.front()); t.pop_front();
24      }
25    return (t.front());
26    }
```

Definition 3.1 A balanced binary tree created by **buildSegTree** is called a *segment tree*.

The notions of root (node) and of depth($\mathbf{N}$) of a node $\mathbf{N}$ are borrowed from the standard terminology for binary trees. So is the depth depth($\mathcal{T}$) of a segment tree $\mathcal{T}$ and notions like "parent" and "child" of a node. All nodes with the same depth form a level of the tree

$$\mathcal{L}_l(\mathcal{T}) = \{\mathbf{N} \in \mathcal{T} : \text{depth}(\mathbf{N}) = l\}\,, \quad l = 0, 1, \ldots\,.$$

For an interval sequence of length n the function **buildSegTree** displayed in Listing 3 queries the sections of the support boxes in coordinate direction i and constructs a segment tree $\mathcal{T}$ with[2]

$$\text{depth}(\mathcal{T}) \le \lceil \log_2(2n+1) \rceil \le \Delta(n) := 1 + \log_2(n+1)\,, \tag{2}$$

and a bound on the number of nodes according to

$$\sharp\mathcal{T} \le 4n + 1\,. \tag{3}$$

Due to the sorting step, the computational effort involved is $O(n \log n)$ for $n \to \infty$. By construction the interval owned by each parent node is the union of the intervals of its children (see Line 18 of Listing 3),

$$\begin{matrix} \texttt{Nl = *N.leftson} \\ \texttt{Nr = *N.rightson} \end{matrix} \Rightarrow \texttt{N.compintv=Nl.compintv} \cup \texttt{Nr.compintv}\,, \tag{4}$$

see Fig. 1 for an example.

[2]We write $\lceil x \rceil$ for the smallest integer $\ge x$.

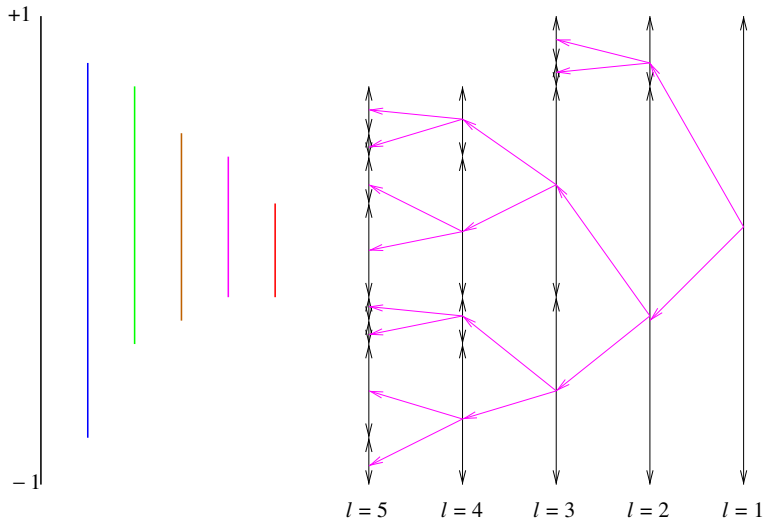

Fig. 1 Segment tree built from a collection of intervals represented by *colored vertical bars* on the *left*: nodes are represented by their comparison intervals, parent–child relationships are indicated by *arrows*

Proposition 3.2 *The comparison intervals of all nodes of a segment tree* $\mathcal{T}$ *on a particular level* $0 \le l \le \text{depth}(\mathcal{T})$ *form a* partition *of an interval* $[-1, \xi[$ *for some* $-1 < \xi \le 1$.

Proof On the leaf level the comparison intervals give a partition of $[-1, \xi[$ defined by the endpoints of the box cross sections in coordinate direction i. The whole interval $[-1, 1[$ may not be covered, because, in case the number of intervals is odd, the last one is moved to the next coarser level of the tree, see Line 23 of Listing 3.

The **while**-loop (Lines 13–24) in **buildSegTree** creates the levels of the tree. If `n_sons` is even, pairs of comparison intervals are merged into the comparison intervals of the next coarser level. If `n_sons` is odd, the last interval is promoted to the next coarser level, cf. Fig. 1. □

A box supported function is added to the loclist data member of a node, if the cross section of its support box in coordinate direction i

- contains the comparison interval of that node,
- but *fails* to contain the comparison interval of its parent node.

This rule is implemented in the recursive function **registerInterval** given in Listing 4. See also Fig. 2 for an example.

Listing 4 Registering box supported functions in local lists

```
void registerInterval(int i,SegTreeNode *n,const
    BoxFunction &f) {
  const Interval &intv(f[i-1]);
  if (intv.contains(n.compintv))
    n.loclist.push_back(f);
  else {
    if (((SegTreeNode *ls = n->leftson) != NULL) &&
          intersect(ls->compintv,intv))
        registerInterval(i,ls,l);
    if (((SegTreeNode *rs = n->rightson) != NULL) &&
          intersect(rs->compintv,intv))
        registerInterval(i,rs,l);
   }}
```

Proposition 3.3 *Assume that* **registerInterval** *(i, root, f) is invoked with* root *a pointer to the root of a segment tree $\mathcal{T}$ built from a list of n box functions with f being one of them. Then*

(i) *at most* $4 \cdot \mathrm{depth}(\mathcal{T})$ *recursive calls to* **registerInterval** *will be made,*
(ii) *f will be inserted into at most* $2\lceil \log_2(2n+1) \rceil$ *nodal lists,*

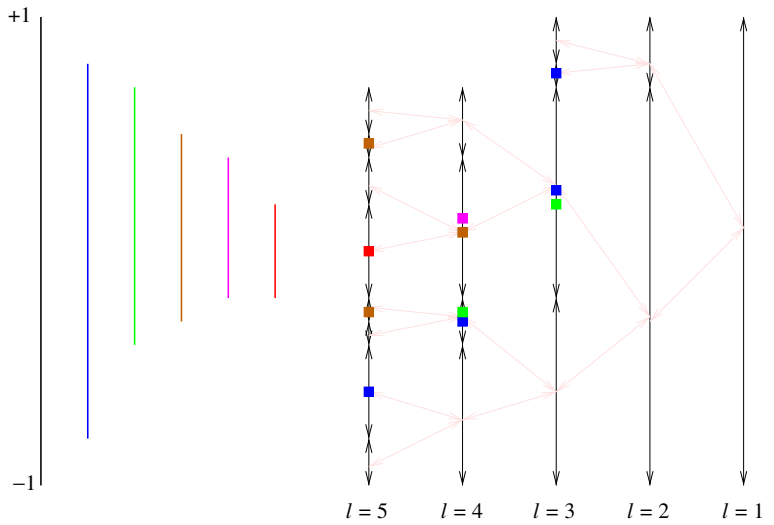

Fig. 2 Intervals (*left*) added to the loclist fields of nodes of the segment tree from Fig. 1 are represented by *squares* in the color of the interval

Proof Denote by $\mathcal{N}(f)$ the set of nodes of the segment tree, for which a recursive function call **registerInterval**(i, ... ,f) is made, see Lines 8, 11 in Listing 4. We first show

$$\sharp(\mathcal{N}(f) \cap \mathcal{L}_l(\mathcal{T})) \le 4 \quad \forall l = 0, \ldots, \text{depth}(\mathcal{T}) \,. \tag{5}$$

We adapt an argument from [5, Proof of Lemma 2.4]. Assume that there was a level l, for which (5) was not true. Note that **registerInterval**(i,n,f) is invoked, if the cross section I in coordinate direction i of the support box of f intersects the comparison interval of the node ∗n. Since, by Proposition 3.2 the comparison intervals of the nodes on level l are contiguous, there would be at least five nodes on level l with *contiguous* comparison intervals that have an overlap with I. Hence, the *adjacent* comparison intervals of three of them must be contained in I. As a consequence there is a parent node on level $l-1$, whose comparison interval is contained in I. In this case **registerInterval** is not invoked for any son node and interval I. This contradiction confirms (5).

The same arguments bear out that[3]

$$\sharp\{\mathbf{N} \in \mathcal{L}_l(\mathcal{T}) : \mathbf{N}.\texttt{compintv} \subset I \ \wedge \ \text{parent}(\mathbf{N}).\texttt{compintv} \not\subset I\} \le 2 \,, \quad l \ge 1 \,.$$

Only for these nodes the box function f is appended to loclist , cf. Lines 3, 4 of Listing 4. In light of the bound (2) the assertion follows. □

Corollary 3.4 *The accumulated length of all local lists stored in the* loclist *data fields of the nodes of a segment tree $\mathcal{T}$ built from a list of n box supported functions by* **buildFullSegTree** (*Listing* 5) *is bounded by*

$$\sum_{\mathbf{N} \in \mathcal{T}} \mathbf{N}.\texttt{loclist.size}() \le \Phi(n) := 2n(2 + \log_2(n+1)) \,.$$

Corollary 3.5 *The computational effort for* **buildFullSegTree**, *see Listing* 5, *when invoked for a sequence of n box supported functions is less than $W_f \Phi(n)$ work units for some $W_f > 0$ independent of n.*

Listing 5 Building a one-dimensional segment tree complete with interval lists

```
1 SegTreeNode *buildFullSegTree(int i,BoxFnSeq &fseq) {
2   SegTreeNode *root = buildSegTree(i,fseq);
3   foreach f in fseq { registerInterval(i,root,f); }
4   return root; }
```

[3]The operator ♯ tells the cardinality of a set.

Lemma 3.6 *Let the segment tree $\mathcal{T}$ be built by* **buildFullSegTree** *(i, fseq) and f be a box supported function contained in the list* fseq *with support $[a_1, b_1[\times\cdots\times[a_d, b_d[$. Then*

$$[a_i, b_i[=\bigcup\{\mathbf{N}.\texttt{compintv}:\ \mathbf{N}\in\mathcal{T},\ f\in\mathbf{N}.\texttt{loclist}\}\ .$$

provides a partition *of* $[a_i, b_i[$.

Proof The assertion of the lemma is an immediate consequence of the fact that all support intervals are the union of comparison intervals, because both **buildSegTree** and **registerInterval** operate on the same list of functions. In addition the partition property stated in the lemma is a consequence of Proposition 3.2. □

4 Box function tree

Now we discuss how to handle multidimensional tensor product supports in order to facilitate the fast point evaluation sought in Task 1. This will be done by means of nested segment trees, each of which belongs to a particular ccoordinate direction i, $1 \le i \le d$. In short, we refer to this number $i \in \{1, \ldots, d\}$ as the *direction* of the tree and its nodes.

In computational geometry nested segment trees are known as *multilevel segment trees*, see [5, Section 2.3]. They are used for efficient point enclosure queries for d-dimensional boxes; more precisely, the data structure allows to access all boxes containing a given point with effort $O(K + \log_2^d n)$ after a preprocessing stage that costs $O(n \log_2^d n)$. Here, K is the number of enclosing boxes found. This is not a useful estimate for our purpose, because all the supports of the terms in (1) may have non-empty intersection. In case the point $\boldsymbol{x}$ lies in this intersection, we encounter $K = n$ and, consequently, $O(n)$ cost for evaluating $\Psi(\boldsymbol{x})$. On the other hand, we do not care about which boxes contain $\boldsymbol{x}$. This suggests that we modify the standard algorithms and augment it by an extra *accumulation step* in the preprocessing stage. This section gives the details.

In a nested segment tree, each node of direction $i > 1$ may hold another segment tree of direction $i - 1$; through the subtreeroot data field of SegTreeNode the node can access the root of this segment tree (subtree), which may be empty. The subtrees are built recursively as segment trees spawned by the local box function lists (loclist field, see Line 5 of the class definition of SegTreeNode in Listing 2) attached to the nodes of the current tree, see the routine **buildSubTrees** given in Listing 6 and [5, Algorithm 2.7]. We start from the d-th coordinate direction in the function **initBoxTree**, see Listing 7, and work our way down to coordinate direction 1. Thus, the level of the recursion in **buildSubTrees** will determine the direction of a subtree and its nodes.

Listing 6 Recursive construction of multidimensional segment tree (box tree)

```
1  void buildSubTrees(int i,SegTreeNode *root) {
2    if (root != NULL) {
3      list<const BoxFunction &> &loclst = root->loclist;
4      if (!loclst.empty()) {
5        if (i>1) {
6          root->subtreeroot =
               buildFullSegTree(i-1,loclst);
7          buildSubTrees(i-1,root->subtreeroot);
8          }
9        else { sumLocFn(loclst,root->locfun);  }
10       (root->loclist).clear(); // Clear local lists, optional
11        }
12     buildSubTrees(i,root->leftson);
13     buildSubTrees(i,root->rightson);
14   }}
```

Listing 7 Initialization of a box tree

```
1  SegTreeNode *initBoxTree(int d,BoxFnSeq fseq) {
2    SegTreeNode *root = buildFullSegTree(d,fseq);
3    buildSubTrees(d,root);
4    return root; }
```

Definition 4.1 We dub multilevel segment trees created by **initBoxTree** from Listing 7 *box function trees*.

Let us single out a node of direction $i = 1$ of a box function tree. It owns a comparison interval I_1. The corresponding subtree is attached to a node of direction 2, which holds a comparison interval I_2, and so forth. Thus we can associate a unique d-dimensional box $I_1 \times I_2 \times \ldots I_d$ to each node of direction 1. All these boxes form an overlapping tiling of $[-1, 1[^d$ and we refer to them as *comparison boxes*.

Listing 8 Summation of basis functions that are uniform on a d-dimensional box

```
1  void sumLocFn(BoxFnSeq &fseq,VFunction &func) {
2    foreach f in fseq { func += f.phi; } }
```

An explanation for the invocation of the function **sumLocFn**, see Listing 8, in Line 9 of **buildSubTrees** is postponed until Section 5.

The function **initBoxTree** performs the preprocessing step of our algorithm. Now we analyze its complexity, starting with auxiliary identities.

Lemma 4.2 *Let* $f : \mathbb{R}_0^+ \mapsto \mathbb{R}_0^+$ *satisfy* $f(0) = 0$. *Then, for any* $n \in \mathbb{N}$, $x > 0$,

$$\max\left\{\sum_{k=1}^{n} f(\xi_k),\ \sum_{k=1}^{n}\xi_k = x,\ \xi_k \geq 0\right\} = \begin{cases} f(x) & \text{, if } f \text{ is convex,} \\ nf\left(\frac{x}{n}\right) & \text{, if } f \text{ is concave.} \end{cases}$$

Proof A convex f with the stated properties satisfies $f(\xi) + f(\eta) \leq f(\xi + \eta)$ for all $\xi, \eta \geq 0$, and, therefore, the sum becomes maximal, when only one of the ξ_k does not vanish. As f is non-decreasing, that ξ_k should attain the maximal value x.

For a concave f we find $f(\xi) + f(\eta) \geq f(\xi + \eta)$ for all $\xi, \eta \geq 0$, which means that the sum becomes maximal in the case $\xi_1 = \xi_2 = \cdots = \xi_n = \frac{x}{n}$. □

Proposition 4.3 (cf. [5, Theorem 2.7]) *The computational effort involved in executing* **initBoxTree** *for a list of* $n \in \mathbb{N}$ *functions with d-dimensional tensor product supports is bounded by* $C \max\{W_f, W_S\} n \log^d n$, *where* $C > 0$ *depends on d only, and* W_s, W_f *were introduced in Section 2 and Corollary* 3.5, *respectively.*

Proof To begin with, note that for $i > 1$ **buildSubTrees**(i,root) involves the following two passes

(I) For each node of the tree invoke **buildFullSegTree** for direction $i - 1$ and on the local list loclist of box functions.
(II) For each node of the tree do a recursive call to **buildSubTrees** passing direction $i - 1$ and the local list.

Write $\omega(i, n)$ for a bound for the computational effort (in work units) it takes to execute **buildSubTrees** for direction $i \in \{1, \ldots, d-1\}$ and on a subtree created from a box function list of length $n \in \mathbb{N}_0$. According to (3) this tree comprises at most $4n + 1$ nodes, which we number consecutively. We denote by m_k the length of the local box function list of node k, $k \in \{1, \ldots, 4n + 1\}$. In case there are fewer nodes, the excess m_k are simply set to zero. Corollary 3.4 gives us the bound

$$\sum_{k=1}^{4n+1} m_k \leq \Phi(n) := 2n(2 + \log_2(n + 1)) \ . \tag{6}$$

If $i = 1$, we merely invoke **sumLocFn**, see Listing 8, on all the local lists, with cost proportional to $\sum_k m_k$, which leads to the estimate

$$\omega(1, n) \leq W_s \Phi(n) \ , \tag{7}$$

with $W_s > 0$ independent of n reflecting the cost of adding two objects of type VFunction, cf. Section 2.

If $i > 1$ we add the effort required by the two passes in **buildSubTrees** to obtain the recursion formula

$$\omega(i,n) \le \max\left\{ \sum_{k=1}^{4n+1} \underbrace{W_f \Phi(m_k)}_{\text{Pass (I)}} + \underbrace{\omega(i-1,m_k)}_{\text{Pass (II)}},\ \sum_{l=1}^{4n+1} m_l \le \Phi(n) \right\}, \tag{8}$$

where, thanks to Corollary 3.5, the cost of a call to **buildFullSegTree** from the k-th node (Pass (I) above) has been bounded by $W_f \Phi(m_k)$.

We continue by induction with respect to i, where, according to the assertion of the theorem, the induction hypothesis is

$$\omega(i,n) \le C_i \Phi(n) \log_2^{i-1}(n+1)\ , \tag{9}$$

which, by (7), is clearly satisfied for $i = 1$ with $C_1 = W_s$. Both $t \mapsto \omega(i-1,t)$ and $t \mapsto \Phi(t)$ are convex, non-negative, and vanish for $t = 0$. Thus we can apply Lemma 4.2 to (8), which yields the recursive estimate

$$\omega(i,n) \le W_f \Phi(\Phi(n)) + \omega(i-1,\Phi(n))\ , \quad i \ge 2\ . \tag{10}$$

The resulting bounds for special values of W_s and W_f are plotted in Fig. 3. Next, plugging (9) for $\omega(i-1,n)$ into (10) yields

$$\omega(i,n) \le W_f \Phi(\Phi(n)) + C_{i-1} \Phi(\Phi(n)) \log_2^{i-2}(\Phi(n)+1)\ , \quad i \ge 2\ . \tag{11}$$

Tedious, but elementary computations establish that for $n \ge 2$

$$\log_2(\Phi(n)+1) \le 5 + \log_2(n+1) + \log_2(\log_2 n)\ , \tag{12}$$

$$\Phi(\Phi(n)) \le 8\Phi(n)\log_2(n+1)\ . \tag{13}$$

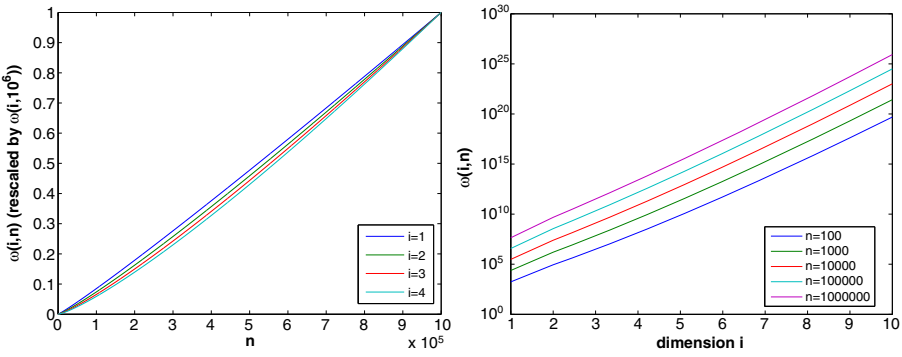

Fig. 3 Bounds $\omega(i,n)$ from (10) and (7) for $W_f = 1$ and $W_s = 1$

Combining these estimates with (11) we conclude that for $n \geq 4$

$$\omega(i,n) \leq 8W_f\Phi(n)\log_2(n+1) + 2^{2i-1}C_{i-1}\Phi(n)\log_2(n+1)\log_2^{i-2}(n+1)\ . \tag{14}$$

This amounts to the induction hypothesis for i with $C_i = W_f + 2^{2i-1}C_{i-1}$. □

5 Point evaluation

After the discussion of the preprocessing stage, we now turn to the actual evaluation requested in Task 1. This is tackled by the function **eval**, see Listing 9, invoked for node=**initBoxTree**(d,fseq) and i=d, where the sequence fseq encodes the sum (1) as explained in Section 2. The argument x must pass a vector with d floating point coefficients: **typedef** vector<**double**> Point. The main difference to the usual point query task discussed in [5, Section 2.3] is that the boxes containing $\boldsymbol{x}$ are not of interest.

Listing 9 Point evaluation function for box function tree

```
1  double eval(const SegTreeNode *node, int i, const
       Point &x) {
2    double val = 0.0;
3    if (node != NULL) {
4      if ((node->compintv).contains(x[i-1])) {
5        if (i == 1)
6          val = node->locfun(x);
7        else
8          val = eval(node->subtreeroot,i-1,x);
9        val += eval(node->leftson,i,x) +
             eval(node->rightson,i,x);
10       }}
11   return val; }
```

Listing 10 Building box function tree combined with point evaluation

```
1  const SegTreeNode *root = initBoxTree(d,fseq);
2  double psi = eval(root,d,x);
```

Proposition 5.1 *If* fseq *is an object of type* BoxFnSeq *encoding the sum* (1)*, and* $\boldsymbol{x} \in \mathbb{R}^d$ *is stored in* x*, then the code given in Listing* 10 *stores the value* $\Psi(\boldsymbol{x})$ *in* psi*.*

Proof of Proposition 5.1

(i) We first establish that a specific box supported function will be taken into account at most once in **eval**. Examining the function **registerInterval** we

note that due to the partition property stated in Lemma 3.6 the nodes to whose loclist data field a fixed box function is added must have disjoint comparison intervals.
Thus, the comparison boxes associated with the nodes of direction 1 of a box function tree that hold a particular box supported function in their loclist fields have to be disjoint, too. Since at most one of a set of disjoint comparison boxes is visited during the execution of **eval**, the same box function will never be summed twice.

(ii) Secondly, we show that each box supported function, whose support contains $\boldsymbol{x} = (x_1, \ldots, x_d)^T$ actually contributes to the sum. Pick such a function f and denote by $B = [a_1, b_1[\times \cdots \times [a_d, b_d[$ its tensor product support, satisfying $\boldsymbol{x} \in B$.
At direction d, since $x_d \in [a_d, b_d[$, thanks to Lemma 3.6, there is a node $\mathbf{N}_d$ of direction d such that

$$x_d \in (\mathbf{N}_d).\texttt{compintv} \quad \wedge \quad f \in (\mathbf{N}_d).\texttt{loclist}\ .$$

From Listing 9 we see that **eval** will be called for the sub-tree attached to $\mathbf{N}_d$, which has been built from a function list containing f. Hence, there is a node $\mathbf{N}_{d-1}$ of direction $d-1$ such that

$$x_{d-1} \in (\mathbf{N}_{d-1}).\texttt{compintv} \quad \wedge \quad f \in (\mathbf{N}_{d-1}).\texttt{loclist}\ .$$

Applying this argument recursively verifies the existence of a node $\mathbf{N}_1$ of direction 1, whose comparison box contains $\boldsymbol{x}$ and whose loclist includes f. Thus, the locfun field of $\mathbf{N}_1$ has been initialized in **sumLocFn**, see Listing 8, from a sum comprising the V-part of f. Consequently, $(\mathbf{N}_1).\texttt{locfun}(\boldsymbol{x})$ involves a term $f(\boldsymbol{x})$. □

Now we study the computational cost of **eval**. We make the natural assumption that the evaluation of a function $\in W$ stored in an object of type VFunction at a single point takes a constant amount of work, that may depend on the dimension d, however.

Proposition 5.2 *If, in Listing* 9*, the box function tree has been created from a function list of length n, then the evaluation in Line* 2 *of Listing* 10 *requires an asymptotic computational effort* $O(W_x \log^d n)$ *for large n.*

Proof At a particular direction i, $1 < i \leq d$, the recursive execution of **eval** for a point $\boldsymbol{x} = (x_1, \ldots, x_d)^T \in [-1, 1[^d$ boils down to

(I) visiting all nodes of direction i, whose comparison interval contains x_i, and
(II) executing **eval** for the sub-trees of direction $i-1$ of those nodes.

Denote by $\zeta(i,n)$ a bound for the cost of executing **eval** for a box function tree of dimension i built from a box function list of length n. For $i=1$ exactly one VFunction object is evaluated at $\boldsymbol{x}$ on each level of the tree, which, by (2), permits us to set

$$\zeta(1,n) \le W_x \Delta(n) \quad , \quad \Delta(n) := \log_2(n+1) \; . \tag{15}$$

The cost of evaluating a single VFunction is incorporated through the constant W_x, which may strongly depend on d, however.

For direction $i>1$, a recursive call to **eval** is made for exactly one node on each level. The local function lists of these nodes do not contain shared box functions, as explained in the proof of Proposition 5.1. Hence, based on (2), we find the recursive estimate

$$\zeta(i,n) \le \max \left\{ \sum_{\ell=1}^{\Delta(n)} 1 + \zeta(i-1,n_k), \; \sum_{\ell=1}^{\Delta(n)} n_\ell \le n \right\} \; ,$$

where n_k is the length of the local box function list attached to the k-th node (of direction i) for which **eval** is called. Since $t \mapsto \zeta(i,t)$ is concave with $\zeta(i,0)=0$, from Lemma 4.2 we infer

$$\zeta(i,n) \le \Delta(n) \left(1 + \zeta \left(i-1, \frac{n}{\Delta(n)} \right) \right) \; . \tag{16}$$

Then a simple induction confirms that the choice $\zeta(i,n) := W_x \log_2^i(n+1)$ complies with (15) and (16) for large n. □

6 Empirical complexity

The theoretical complexity bounds of Propositions 4.3 and 5.2 hold for worst case scenarios concerning the arrangement of support boxes. In this section, we study the actual effort for the preprocessing and evaluation stages for concrete examples of functions Ψ and sets of evaluation points. Throughout, V is the space of multi-linear functions $\mathbb{R}^d \mapsto \mathbb{R}$.

To gauge the cost, we measure certain operation counts for the setup phase and evaluation stage of our algorithm in different typical situations. In partiular, for setup we tracked

- the execution count **Nadd** for the operation += for objects of type VFunction, which is needed in the function **sumLocFn**, see Listing 8.
- the number **Ncontains** of enclosure tests for two intervals, as needed in Line 3 of **registerInterval**, see Listing 4.
- the number **Nintscts** of intersection queries for two invervals, as used in Lines 7, 10 of **registerInterval**, see Listing 4.

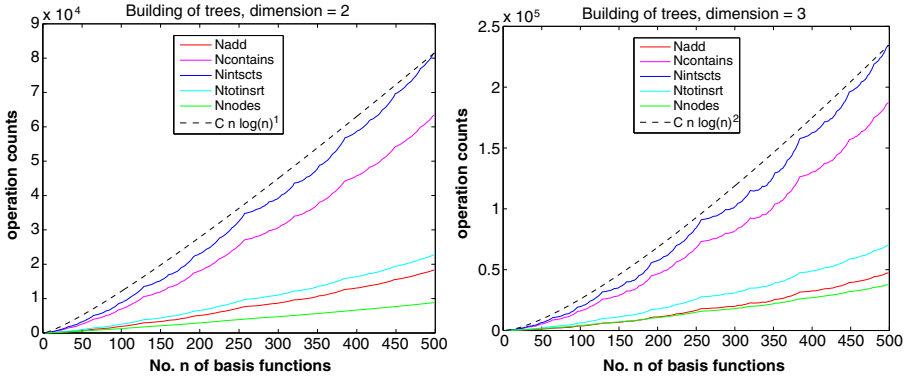

Fig. 4 Experiment 1: invocation counts during initialization of the multidimensional segment tree for $d = 2$ (*left*), $d = 3$ (*right*)

- a counter **Ntotinsrt** for appending box function objects to local lists, as done in Line 4 of **registerInterval**, see Listing 4.
- the total number **Nnodes** of nodes of segment trees of various directions created during setup by the **new** statements in Lines 11, 19 of **buildSegTree**, see Listing 3.

For the point evaluation as implemented in the **eval** function of Listing 9 we monitored the *average* number

- **Neval** of point evaluations of VFunction-objects as done in Line 6 of the **eval** function.
- **NPcont** of queries issued in Line 4 whether an interval contains a point.

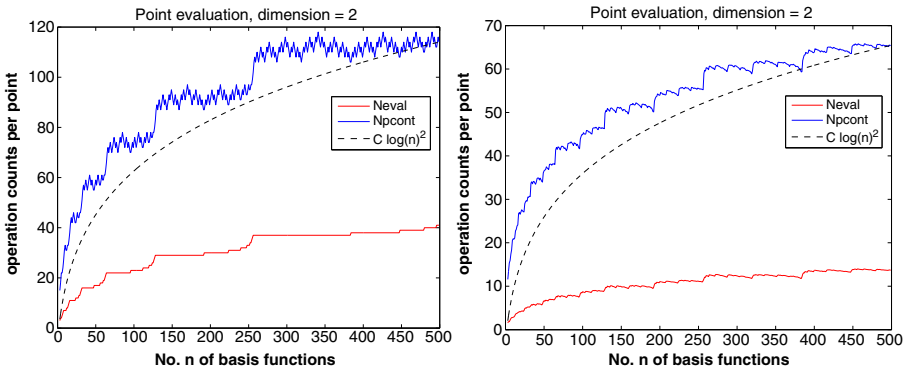

Fig. 5 Experiment 1: (average) invocation counts during point evaluation for $d = 2$: single point $x = 0$ (*left*), randomly chosen points (*right*)

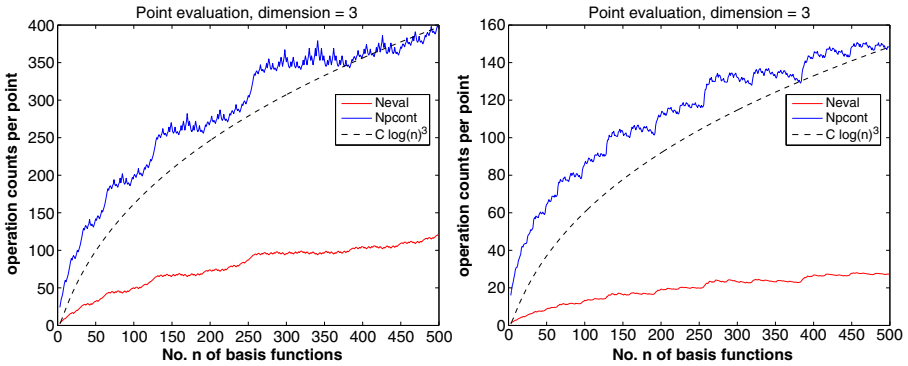

Fig. 6 Experiment 1: (average) invocation counts during point evaluation for $d = 3$: single point $x = 0$ (*left*), randomly chosen points (*right*)

6.1 Experiment 1

We pick $n \in \mathbb{N}$ basis functions $\varphi_1, \ldots, \varphi_n$ with nested supports

$$\operatorname{supp}(\varphi_i) = \left[-1 + \frac{i}{n+1}, 1 - \frac{i}{n+1}\right]^d , \quad i = 1, \ldots, n , \tag{17}$$

centered around zero. Their linear combination was evaluated for

1. the central point 0 contained in the intersection of all supports,
2. and 10^4 points randomly chosen in $[-1, 1]^d$ (uniform distribution).

The various operation counts for $d = 2, 3$ are depicted in the graphs of Figs. 4, 5, and 6. They very well match the theoretical predictions of asymptotic complexity given in Propositions 4.3 and 5.2. Small wonder, since the situation that the evaluation point belongs to all support boxes should really represent the worst possible arrangement for our algorithm.

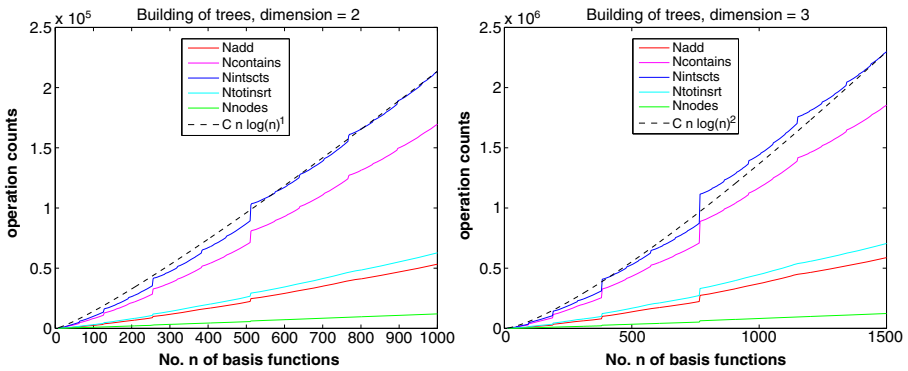

Fig. 7 Experiment 2: invocation counts during initialization of the multidimensional segment tree for $d = 2$ (*left*), $d = 3$ (*right*)

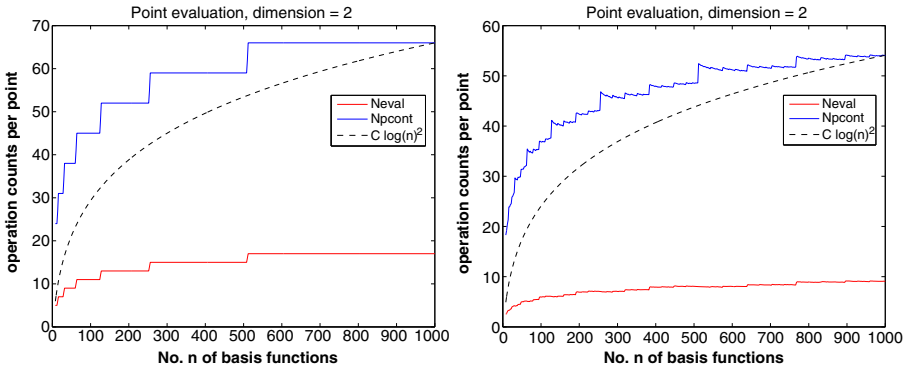

Fig. 8 Experiment 2: (average) invocation counts during point evaluation for $d = 2$: single point $x = 0$ (*left*), randomly chosen points (*right*)

6.2 Experiment 2

We pick $n = dk$ basis functions, $k \in \mathbb{N}$, $\varphi_1, \ldots, \varphi_{dm}$ with anisotropic supports

$$\begin{aligned}
\operatorname{supp}(\varphi_i) &= \left[-1 + \frac{i}{k+1}, 1 - \frac{i}{k+1}\right] \times [-1,1]^{d-1}\,, \quad i = 1, \ldots, k\,,\\
\operatorname{supp}(\varphi_i) &= [-1,1] \times \left[-1 + \frac{i-k}{k+1}, 1 - \frac{i-k}{k+1}\right] \times [-1,1]^{d-2}\,,\\
&\qquad\qquad i = k+1, \ldots, 2k\,,\\
&\vdots\\
\operatorname{supp}(\varphi_i) &= [-1,1]^{d-1} \times \left[-1 + \frac{i-(d-1)k}{k+1}, 1 - \frac{i-(d-1)k}{k+1}\right]\,,\\
&\qquad\qquad i = (d-1)k+1, \ldots, dk\,.
\end{aligned}$$

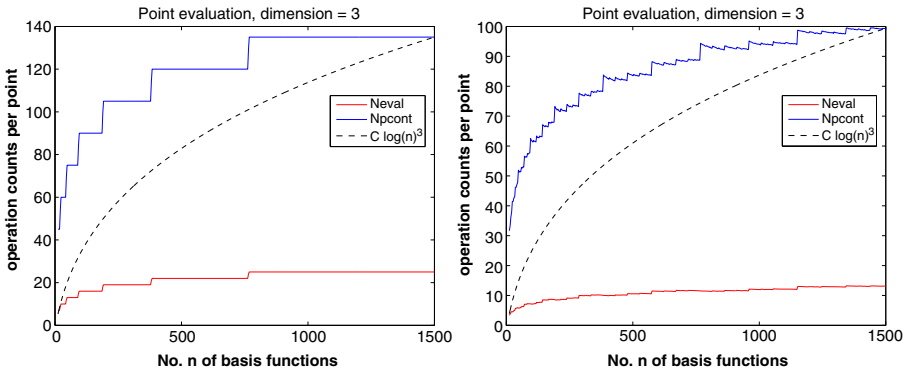

Fig. 9 Experiment 2: (average) invocation counts during point evaluation for $d = 3$: single point $x = 0$ (*left*), randomly chosen points (*right*)

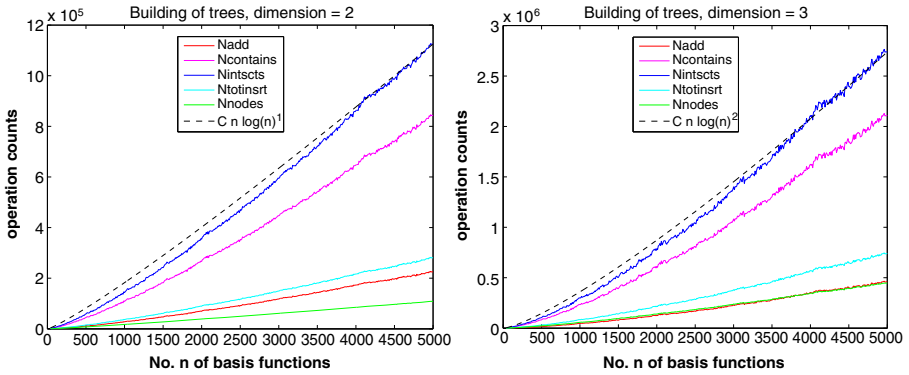

Fig. 10 Experiment 3: invocation counts during initialization of the multidimensional segment tree for $d = 2$ (*left*), $d = 3$ (*right*)

Again their linear combination was evaluated for

1. the central point 0 contained in the intersection of all supports,
2. and 10^4 randomly chosen in $[-1, 1]^d$.

Refer to Figs. 7, 8, and 9 for the measured number of operations for $d = 2, 3$. Apparently **eval** runs faster than predicted by Proposition 5.2, which is not surprising, because each evaluation point is contained in only a small number of support boxes.

6.3 Experiment 3

We choose n basis functions with random supports, that is, the endpoints of the intervals forming their support boxes were randomly sampled from a uniform distribution in $[-1, 1]$ and swapped, if necessary. For $d = 2, 3$ the operations

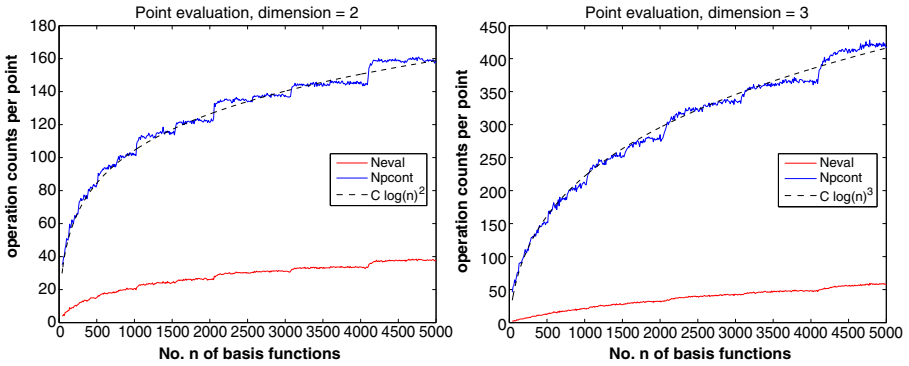

Fig. 11 Experiment 3: average invocation counts during point evaluation at randomly chosen points, $d = 2$ (*left*), $d = 3$ (*right*)

counts are plotted against n in Figs. 10 and 11. It seems that this random placement of support boxes makes our algorithm operate close to the worst case complexity bounds (as in Experiment 1).

Acknowledgements The authors would like to thank the two anonymous referees for their valuable suggestions.

References

1. Bungartz, H.-J., Griebel, M.: Sparse grids. Acta Numer. **13**, 147–269 (2004)
2. Campos Pinto, M.: A direct and accurate adaptive semi-Lagrangian scheme for the Vlasov–Poisson equation. Int. J. Appl. Math. Comput. Sci. **17**, 351–359 (2007)
3. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry. Algorithms and Applications, 2nd edn. Springer, Berlin (2000)
4. Klompmaker, I.: A semi-lagrangian scheme using adaptive sparse grids for front propagation. In: Slides for Workshop on Advancing Numerical Methods for Viscosity Solutions and Applications, Banff, 13–18 Feb 2011. http://temple.birs.ca/∼11w5086/Klompmaker.pdf (2011)
5. Langetepe, E., Zachmann, G.: Geometric Data structures for Computer Graphics. A K Peters, Wellesley, MA (2006)
6. Staniforth, A., Cote, J.: Semi-Lagrangian integration scheme for atmospheric models: a review. Mon. Weather Rev. **119**, 2206–2223 (1991)
7. Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison Wesley Longman, Reading, MA (1997)
8. Widmer, G., Hiptmair, R., Schwab, C.: Sparse adaptive finite elements for radiative transfer. J. Comput. Phys. **227**, 6071–6105 (2008)