Computing 71, 65–87 (2003)
Digital Object Identifier (DOI) 10.1007/s00607-003-0015-5

Computing
Printed in Austria

Dimension–Adaptive Tensor–Product Quadrature

T. Gerstner and **M. Griebel,** Bonn

Received March 25, 2003; revised April 15, 2003
Published online: June 23, 2003

Abstract

We consider the numerical integration of multivariate functions defined over the unit hypercube. Here, we especially address the high–dimensional case, where in general the curse of dimension is encountered. Due to the concentration of measure phenomenon, such functions can often be well approximated by sums of lower–dimensional terms. The problem, however, is to find a good expansion given little knowledge of the integrand itself.
The dimension–adaptive quadrature method which is developed and presented in this paper aims to find such an expansion automatically. It is based on the sparse grid method which has been shown to give good results for low- and moderate–dimensional problems. The dimension–adaptive quadrature method tries to find important dimensions and adaptively refines in this respect guided by suitable error estimators. This leads to an approach which is based on generalized sparse grid index sets. We propose efficient data structures for the storage and traversal of the index sets and discuss an efficient implementation of the algorithm.
The performance of the method is illustrated by several numerical examples from computational physics and finance where dimension reduction is obtained from the Brownian bridge discretization of the underlying stochastic process.

AMS Subject Classifications: 65D30, 65C20, 65U05, 65Y20.

Keywords: multivariate numerical integration, adaptivity, curse of dimension.

1. Introduction

The computation of high–dimensional integrals is a central part of computer simulations in many application areas such as statistical mechanics, financial mathematics, and computational physics. Here, the arising integrals usually cannot be solved analytically, and thus, numerical approaches are required. Furthermore, often a high accuracy solution is needed and thus, such problems can be computationally quite challenging even for parallel supercomputers.

The main reason for this difficulty is the so–called *curse of dimension* [2], which can be understood in two ways. First, one observes that in classical numerical quadrature methods (e.g. based on product rules) the amount of work N required in order to achieve a prescribed accuracy ε grows exponentially with the dimension d,

$$\varepsilon(N) = O(N^{-r/d}),$$

for functions with bounded derivatives up to order r [7]. Thus, already for moderate dimensions the order of convergence is so slow that a high accuracy cannot be obtained in practice. The situation gets worse as the dimension increases.

The curse of dimension can also be approached from the point of numerical complexity theory. There it has been shown that for some integration problems (i.e. for integrands from certain function spaces) even the minimum amount of work in order to achieve a prescribed accuracy grows exponentially with the dimension [34]. These lower bounds hold for all algorithms from a specific algorithmic class (i.e. those using linear combinations of function evaluations). Such problems are therefore called intractable. However, application problems are often in a different (or smaller) problem class and thus may be tractable, although the correct classification can be difficult. In addition, there may exist (e.g. non–linear or quantum) algorithms which stem from a different algorithmic class and thus may be able to break the curse of dimension.

Randomized algorithms, whose probably best–known representative are Monte Carlo methods, is one such class of algorithms. Here, the integrand is evaluated at a set of (pseudo-)randomly chosen points and the integral is computed approximately as the average of these function values. Then, the average amount of work in order to reach an accuracy ε (for integrands with bounded variance) is

$$\varepsilon(N) = O(N^{-1/2})$$

and is thus independent of the dimension. Nevertheless, the convergence rate is quite low and a high accuracy is only achievable with a tremendous amount of work (that is function evaluations). Indeed, more than half of the computing time of today's supercomputers is used just for the generation of random numbers.

Therefore, so–called Quasi–Monte Carlo algorithms have attained much attention in the last years. Here, the integrand is evaluated not at random but at structurally determined points such that the discrepancy of these points is smaller than that for random points. Then, for functions with bounded (mixed) variation, the complexity is

$$\varepsilon(N) = O(N^{-1}(\log N)^d)$$

and is thus almost half an order better than the complexity of the Monte Carlo approach [25]. In addition, the bounds are deterministic. However, the dimension enters through a logarithmic term and this dependence on the dimension therefore causes problems for high dimensions.

Note that in both cases the convergence rate does not depend on the smoothness. Thus, smoother integrands are not computed more efficiently than non–smooth ones. The first method which makes use of the smoothness of the integrand and at the same time does not suffer from the curse of the dimension is the so–called

sparse grid method [41] which dates at least back to the Russian mathematician Smolyak [33]. In this approach, multivariate quadrature formulas are constructed by a combination of tensor products of univariate formulas. Of all possible combinations of one–dimensional quadrature formulas only those are considered whose corresponding indices are contained in the unit simplex. This way, the complexity becomes

$$\varepsilon(N) = O(N^{-r}(\log N)^{(d-1)(r+1)}),$$

for functions from spaces with bounded mixed derivatives up to order r. Thus, for $r > 1$ a better convergence rate than for Quasi–Monte Carlo can be expected. For very smooth integrands ($r \to \infty$) the convergence will even be exponential.

Despite the large improvements of the Quasi–Monte Carlo and sparse grid methods over the Monte Carlo method, their convergence rates will suffer more and more with rising dimension due to their respective dependence on the dimension in the logarithmic terms. Therefore, one aim of recent numerical approaches has been to reduce the dimension of the integration problem without (too great) affection of the accuracy.

In some applications, the different dimensions of the integration problem are not equally important. For example, in path integrals the number of dimensions corresponds to the number of time–steps in the time discretization. Typically the first steps in the discretization are more important than the last steps since they determine the outcome more substantially. In other applications, although the dimensions seem to be of the same importance at first sight, the problem can be transformed into an equivalent one where the dimensions are not. Examples are the Brownian bridge discretization or the Karhunen–Loeve decomposition of stochastic processes.

Intuitively, problems where the different dimensions are not of equal importance might be easier to solve. Numerical methods could concentrate on the more important dimensions and spend more work for these dimensions than for the unimportant ones. Interestingly, also complexity theory reveals that integration problems with weighted dimensions can become tractable even if the unweighted problem is not [39]. Unfortunately, classical adaptive numerical integration algorithms [13, 35] cannot be applied to high–dimensional problems since the work overhead in order to find and adaptively refine in important dimensions would be too large.

To this end, a variety of algorithms have been developed which try to find and quantify important dimensions. Often, the starting point of these algorithms is Kolmogorov's superposition theorem [22, 23]. Here, a high–dimensional function is approximated by sums of lower–dimensional functions. A survey of this approach from the point of approximation theory is given in [21]. Further results can be found in [29, 32]. Analogous ideas are followed in statistics for regression problems and density estimation. Here, examples are so–called additive models [16], multivariate adaptive regression splines (MARS) [11], and the ANOVA

decomposition [37, 40], see also [19]. Other interesting techniques for dimension reduction are presented in [17].

In case the importance of the dimensions is known a priori, techniques such as importance sampling can be applied in Monte Carlo methods [20]. For the Quasi–Monte Carlo method already a sorting of the dimensions according to their importance leads to a better convergence rate (yielding a reduction of the effective dimension). The reason for this is the better distributional behaviour of low discrepancy sequences in lower dimensions than in higher ones [6]. The sparse grid method, however, a priori treats all dimensions equally and thus gains no immediate advantage for problems where dimensions are of different importance.

The aim of this paper is to develop a generalization of the conventional sparse grid approach [33] which is able to adaptively assess the dimensions according to their importance and thus reduces the dependence of the computational complexity on the dimension. The dimension–adaptive algorithm tries to find important dimensions automatically and adapts (places more integration points) in those dimensions. To achieve this efficiently, a data structure for a fast bookkeeping and searching of generalized sparse grid index sets is proposed as well. We will show the performance of the new algorithm in a series of moderate and high–dimensional numerical examples from computational physics and finance. Thereby, the Brownian bridge discretization for the underlying stochastic processes is used advantageously.

The outline of this paper is as follows. In Section 2 we will shortly review the conventional sparse grid approach for multivariate integration. In Section 3 we will then illustrate the dimension–adaptive algorithm. Data structures and implementation details are the subject of Section 4. Numerical examples are presented in Section 5. The remarks of Section 6 conclude the paper.

2. Sparse Grids

Let us briefly review the conventional sparse grid method and indicate some basic properties of sparse grid quadrature formulas. For more information on the method itself and the previous literature see [14].

In the following, boldface letters indicate d–dimensional vectors or multi–indices. Let us consider the numerical integration of functions $f^{(d)}(\mathbf{x})$ from a function class $\mathcal{F}$ over the d–dimensional hypercube $\Omega := [-1,1]^d$,

$$If^{(d)} := \int_{\Omega} f^{(d)}(\mathbf{x})\, d\mathbf{x},$$

by a sequence of $n_l^{(d)}$–point quadrature formulas with level $l \in \mathbb{N}$ and $n_l^{(d)} < n_{l+1}^{(d)}$,

$$Q_l f^{(d)} := \sum_{i=1}^{n_l^{(d)}} w_{li} \cdot f^{(d)}(\mathbf{x}_{li}),$$

using weights w_{li} and abscissas $\mathbf{x}_{li}$. The quadrature error is defined by $E_l f^{(d)} := |If^{(d)} - Q_l f^{(d)}|$.

The sparse grid construction starts with a series of one–dimensional quadrature formulas for a univariate function $f^{(1)}$,

$$Q_l f^{(1)} := \sum_{i=1}^{n_l^{(1)}} w_{li} \cdot f^{(1)}(x_{li}).$$

Now, consider the difference formulas defined by

$$\begin{aligned} &\Delta_k f^{(1)} := (Q_k - Q_{k-1}) f^{(1)} \quad \text{with} \\ &Q_0 f^{(1)} := 0. \end{aligned}$$

Then, for $\mathbf{k} \in \mathbb{N}^d$, the conventional *sparse grid* quadrature method for d–dimensional functions $f^{(d)}$ is for a given level $l \in \mathbb{N}$

$$Q_l f^{(d)} := \sum_{|\mathbf{k}|_1 \le l+d-1} (\Delta_{k_1} \otimes \ldots \otimes \Delta_{k_d}) f^{(d)}. \tag{1}$$

Here, of all possible product combinations of one–dimensional quadrature formulas only those are considered whose indices are contained in the unit simplex $|\mathbf{k}|_1 \le l + d - 1$. The collection of all $n_l^{(d)}$ points $\mathbf{x}_{li} \in \Omega$ generated in this way is called a (conventional) sparse grid of level l. Note that if the univariate quadrature formulas are nested, then $\Delta_k f^{(1)}$ requires the same number of function evaluations as $Q_l f^{(1)}$ and also the resulting sparse grids are nested.

We will now take a look at the integration error of the sparse grid method. Let us consider the class of functions $\mathscr{W}_d^r$ with bounded mixed derivatives of order r,

$$\mathscr{W}_d^r := \left\{ g : \Omega \longrightarrow \mathbb{R}, \ \left\| \frac{\partial^{|\mathbf{s}|_1} g}{\partial x_1^{s_1} \ldots \partial x_d^{s_d}} \right\|_\infty < \infty, \ s_i \le r \right\}.$$

Let us further assume that the underlying one–dimensional quadrature formula satisfies the error bound

$$|E_l f^{(1)}| = O((n_l^{(1)})^{-r}),$$

for functions $f^{(1)} \in \mathscr{W}_1^r$. This bound holds, for example, for all interpolatory quadrature formulas with positive weights, such as the Clenshaw–Curtis, Gauß–Patterson and Gauß–Legendre formulas [7]. If such a quadrature formula is taken as the one–dimensional starting point and if $n_l^{(1)} = O(2^l)$, then the error of the conventional sparse grid quadrature formula is of the order

$$|E_l f^{(d)}| = O((n_l^{(d)})^{-r} (\log n_l^{(d)})^{(d-1)(r+1)}),$$

for $f \in \mathscr{W}_d^r$ [38].

We see that the convergence rate depends only weakly on the dimension but strongly on the smoothness r. However, the conventional sparse grid method treats all dimensions equally (because this is also true for the unit simplex) and thus the dependence of the quadrature error on the dimension in its logarithmic term will cause problems for high–dimensional integrands.

3. Dimension–Adaptive Quadrature

In order to be able to assess the dimensions differently, it is necessary to modify the original sparse grid construction. Note that conventional adaptive sparse grid approaches [3, 4, 9] merely tackle a locally non–smooth behaviour of the integrand function and usually cannot be applied to high–dimensional problems.

The most straightforward way to generalize the conventional sparse grid with respect to differently important dimensions is to consider a different index set than the unit simplex $|\mathbf{k}|_1 \leq l + d - 1$. For example, one could consider the class of general simplices $\mathbf{a} \cdot \mathbf{k} \leq l + d - 1$ where $\mathbf{a} \in \mathbb{R}_+^d$ is a weight vector for the different dimensions [12, 14, 30]. A static strategy would be to analyse the problem and then to choose a suitable vector $\mathbf{a}$. Such a strategy has two drawbacks, though. First, it is hard to a–priori choose the optimal (or, at least, a good) weight vector $\mathbf{a}$, and second, the class of general simplices itself may be inadequate for the problem at hand (e.g. more or less points in mixed directions may be required).

Instead, we will allow more general index sets [18, 28, 39] in the summation of (1) and try to choose them properly. To this end, we will consider the selection of the whole index set as an optimization problem, i.e. as a binary knapsack problem [5, 15], which is closely related to best N–term approximation [8]. A self–adaptive algorithm can try to find the optimum index set in an iterative procedure. However, not all index sets are admissible in the generalized sparse grid construction and special care has to be taken during the selection of indices, as we will see.

In the following, we will take a look at the general sparse grid construction and at the required conditions on the index set. After that, we will present the basic iterative algorithm for the selection of an appropriate index set. Then, we will address the important issue of error estimation.

3.1 Generalized sparse grids

We will start with the admissibility condition on the index set for the generalized sparse grid construction. An index set $\mathcal{I}$ is called *admissible* if for all $\mathbf{k} \in \mathcal{I}$,

$$\mathbf{k} - \mathbf{e}_j \in \mathcal{I} \text{ for } 1 \leq j \leq d, k_j > 1,$$

holds. Here, $\mathbf{e}_j$ is the j–th unit vector. In other words, an admissible index set contains for every index $\mathbf{k}$ all indices which have smaller entries than $\mathbf{k}$ in at least one dimension. Note that the admissibility condition on the index set ensures the

validity of the telescope sum expansion of the general sparse grid quadrature formulas using the difference formulas $\Delta^1_{k_j}$.

Now we are able to define the *general sparse grid construction* [14]:

$$Q^{(d)}_{\mathscr{I}} f^{(d)} := \sum_{\mathbf{k}\in\mathscr{I}} (\Delta_{k_1} \otimes \cdots \otimes \Delta_{k_d}) f^{(d)},$$

for an admissible index set $\mathscr{I} \in \mathbb{N}^d$.

Note that this general sparse grid construction includes conventional sparse grids ($\mathscr{I} = \{\mathbf{k} : |\mathbf{k}|_1 \leq l + d - 1\}$) as well as classical product formulas ($\mathscr{I} = \{\mathbf{k} : \max\{k_1, \ldots, k_d\} \leq l\}$) as special cases. Unfortunately, little is known about error bounds of quadrature formulas associated to general index sets $\mathscr{I}$ (see [28,39]). However, by a careful construction of the index sets $\mathscr{I}$ we can hope that the error for generalized sparse grid quadrature formulas is at least as good as in the case of conventional sparse grids. Furthermore, the algorithm allows for an adaptive detection of the important dimensions.

3.2 Basic algorithm

Our goal is now to find an admissible index set such that the corresponding integration error ε is as small as possible for a given amount of work (function evaluations). The procedure starts with the one–element index set $\{\mathbf{1}\}, \mathbf{1} = (1, \ldots 1)$ and adds indices successively such that

- the resulting index sets remain admissible, and
- possibly a large error reduction is achieved.

To this end, an estimated error $g_{\mathbf{k}}$ called *error indicator* is assigned to each index $\mathbf{k}$ which is computed from the differential integral

$$\Delta_{\mathbf{k}} f^{(d)} = (\Delta_{k_1} \otimes \cdots \otimes \Delta_{k_d}) f^{(d)} \tag{2}$$

and from further values attributed to the index $\mathbf{k}$ like the work involved for the computation of $\Delta_{\mathbf{k}} f$. Let us remark here that the exact integration error is unknown since the integrand itself is unknown. We will address error estimation in the next section.

In our algorithm always the index with the largest error indicator is added to the index set. Once an index is added, its forward neighbourhood is scanned for new admissible indices and their error indicators are computed. Here, the *forward neighbourhood* of an index $\mathbf{k}$ is defined as the d indices $\{\mathbf{k} + \mathbf{e}_j,\ 1 \leq j \leq d\}$. Conversely, the *backward neighbourhood* is defined by $\{\mathbf{k} - \mathbf{e}_j,\ 1 \leq j \leq d\}$. Altogether, we hope to heuristically build up an optimal index set in the sense of [5, 15] or [8] this way.

Algorithm:

```
integrate(f)
    i := (1, ..., 1)
    O := ∅
    A := {i}
    r := Δ_i f
    η := g_i
    while (η > TOL) do
        select i from A with largest g_i
        A := A \ {i}
        O := O ∪ {i}
        η := η − g_i
        for k := 1, ..., d do
            j := i + e_k
            if j − e_q ∈ O for all q = 1, ..., d then
                A := A ∪ {j}
                s := Δ_j f
                r := r + s
                η := η + g_j
            endif
        endfor
    endwhile
    return r
```

Symbols:

$\mathcal{O}$	old index set
$\mathcal{A}$	active index set
$\Delta_{\mathbf{i}} f$	integral increment $\bigotimes_{k=1}^{d} \Delta_{i_k} f$
$g_{\mathbf{i}}$	local error indicator
η	global error estimate $\sum_{\mathbf{i} \in \mathcal{A}} g_{\mathbf{i}}$
$\mathbf{e}_k$	k–th unit vector
TOL	error tolerance
r	computed integral value $\sum_{\mathbf{i} \in \mathcal{O} \cup \mathcal{A}} \bigotimes_{k=1}^{d} \Delta_{i_k} f$

Fig. 1. The basic dimension–adaptive algorithm

Recall that the computed total integral is just the sum over all differential integrals within the actual index set $\mathcal{I}$. Now as soon as the error indicator for a new index is computed, the index can in fact already be added to the index set since it does not make sense to exclude the just computed differential integral from the total integral. Therefore, when the error indicator of an index is computed, the index is

put into the index set $\mathscr{I}$ (but its forward neighbours in turn are currently not considered).

To this end, we partition the current index set $\mathscr{I}$ into two disjoint sets, called *active* and *old* indices. The active index set $\mathscr{A}$ contains those indices of $\mathscr{I}$ whose error indicators have been computed but the error indicators of all their forward neighbours have not yet been considered. The old index set $\mathscr{O}$ contains all the other indices of the current index set $\mathscr{I}$. The error indicators associated with the indices in the set $\mathscr{A}$ act as an estimate $\eta = \sum_{\mathbf{i} \in \mathscr{A}} g_{\mathbf{i}}$ for the global error.

Now, in each iterative step of the dimension–adaptive algorithm the following actions are taken: The index with the largest associated error indicator is selected from the active index set and put into the old index set. Its associated error is subtracted from the global error estimate η. Also, the error indicators of the admissible forward neighbouring indices of this index are computed and their indices are put into the active index set. Accordingly, the corresponding values of the differential integral (2) are added to the current quadrature result and the corresponding values of the error indicators are added to the current global error estimate. If either the global error estimate falls below a given threshold or the work count exceeds a given maximal amount, the computation is stopped and the computed integral value is returned. Otherwise, the index with the now largest error is selected, and so on (see Figure 1).

A two–dimensional example for the operations of the algorithm is shown in Figure 2. Whenever an active index is selected and put into the old index set (in this example the indices $(2,2)$, $(1,4)$, and $(2,3)$) its two forward neighbours (indicated by arrows) are considered. If they are admissible, they are inserted in the active index set. In the example the forward neighbour $(2,4)$ of $(1,4)$ is not inserted since it is not admissible (its backward neighbour $(2,3)$ is in the active index set but not in the old index set).

3.3 Error estimation

Error estimation is a crucial part of the algorithm. If the estimated error for a given index $\mathbf{k}$ happens to be very small, then there may be no future adaptive refinement in its forward neighbourhood. Now, this behaviour can be good or bad. If the errors of the forward neighbours of $\mathbf{k}$ are smaller or of the same magnitude as the error of $\mathbf{k}$, then the algorithm has stopped the adaption properly. But, it might be that one or more forward neighbours have a significantly larger error and thus the algorithm should refine there. Unfortunately, there is usually no way to know the actual magnitude beforehand (besides by a close a–priori analysis of the integrand function, which is usually not available). The problem could of course be fixed by actually looking at the forward neighbours and the computation of their error indicators. But, this just puts the problem off since we encounter the same difficulty again with the neighbours of the neighbours.

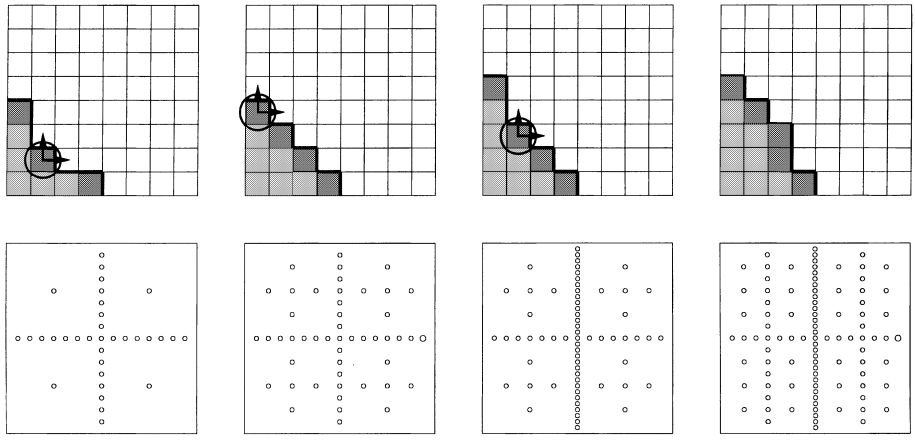

Fig. 2. A few snapshots of the evolution of the dimension–adaptive algorithm. Shown are the sparse grid index sets (upper row) together with the corresponding sparse grids using the midpoint rule (lower row). Active indices are dark–shaded, old indices are light–shaded. The encircled active indices have the largest error indicators and are thus selected for insertion into the old index set

We will here attack this problem through an additional consideration of the involved work. The number of function evaluations required for the computation of the differential integral (and thus also for the error estimation) for a given index $\mathbf{k}$ is known beforehand. If we assume that the univariate quadrature formulas are nested, then the number of function evaluations $n_{\mathbf{k}}$ related to an index $\mathbf{k}$ is given by

$$n_{\mathbf{k}} := n_{k_1}^{(1)} \cdot \ldots \cdot n_{k_d}^{(1)},$$

and thus can be computed directly from the index vector. Now, in order to avoid a too early stopping it makes sense to consider the forward neighbourhood of an index with a small error if the work involved is small – especially in comparison to the work for the index with the currently largest error. Let us therefore consider a generalized error indicator $g_{\mathbf{k}}$ which depends on both the differential integral and the number of function evaluations,

$$g_{\mathbf{k}} := q(|\Delta_{\mathbf{k}} f|, n_{\mathbf{k}}),$$

with a yet to be specified function q which relates these two numbers. Clearly, the function q should be increasing with the first and decreasing with the second argument.

As a possible choice for q we will consider the following class of generalized error estimators

$$g_{\mathbf{k}} = \max\left\{ w\frac{|\Delta_{\mathbf{k}} f|}{|\Delta_{\mathbf{1}} f|}, (1-w)\frac{n_{\mathbf{1}}}{n_{\mathbf{k}}} \right\}$$

where $w \in [0,1]$ relates the influence of the error in comparison to the work (we assume that $\Delta_{\mathbf{1}} f \neq 0$; this reference value can also be replaced by a suitable normalizing constant or the maximum of previously computed differential integrals). Let us remark that usually $n_{\mathbf{1}} = 1$.

By selection of $w = 1$ a greedy approach is taken which disregards the second argument i.e. when the function is known to be very smooth (e.g. strictly convex or concave) and thus the error estimates would decay with increasing indices anyway. Classical sparse grids are realized by $w = 0$ and in this case only the involved work is counted. Values of w in between will safeguard against both comparatively too high work and comparatively too small error.

Note that in general we have to assume that the integrand function fulfills a certain *saturation assumption*, compare also [1,10,36] for the case of adaptive finite elements. This means that the error indicators roughly decrease with the magnitude of their indices. This condition would not be true for example for functions with spikes on a very fine scale or large local discontinuities. Let us remark here that we believe it impossible to search such spikes or discontinuities in high–dimensional space unless the integrand function has special properties (for example, convexity). Note that such functions would practically not be integrable by Monte Carlo and Quasi–Monte Carlo methods as well.

Note furthermore that the global error estimate η typically underestimates the error. But, η and the true integration error ε are proportional to each other if the error indicators decrease with the magnitude of their indices. Therefore, the error tolerance `TOL` is only achieved up to a constant. In our experiments of Section 5 this constant was of moderate size (up to 10).

4. Data Structures

The number of indices in the index sets can become very large for difficult (high–dimensional) problems. For the performance of the overall dimension–adaptive algorithm it is necessary to store the indices in such a way that the operations required by the algorithm can be performed efficiently.

In view of Section 3.2 these operations are

- to insert and remove indices from the active index set $\mathscr{A}$,
- to insert indices into the old index set $\mathscr{O}$,
- to find the index in the active index set with the largest error,
- to check if an index is admissible.

In this section we will describe the data structures which allow a fast execution of these operations. We will use relative addressing for the storage of the indices, a heap for the active indices, a linear array for the old indices, and linked neighbour lists for the admissibility check.

4.1 Relative addressing

In contrast to classical numerical algorithms the dimension d of the problem at hand is highly variable and cannot be neglected in the space and time complexity of the algorithm. In application problems this dimension can readily range up to 1000 and, for example, already a cubic dependence on the dimension can render an algorithm impractible.

This easily overlooked problem becomes visible when for example a multi–index of dimension d has to be copied to a different memory location or when two indices have to be checked for identity. A straightforward approach would require $O(d)$ operations (to copy or compare all the single elements). If these operations are performed within an inner loop of the algorithm, complexities multiply and the total dependence on d is increased.

Therefore, we use relative addressing here. We allocate one two–dimensional array `I` for all (active and old) indices which contains the elements of the current index set $\mathscr{I} = \mathscr{A} \cup \mathscr{O}$. This array has dimension $m \times d$ where m is the maximum number of generated indices. The size m can be chosen statically as the maximum amount of memory available (or that one is willing to spend). Alternatively, m can be determined dynamically and the whole array is reallocated (e.g. with size $2m$) when the current number of elements denoted by `ni` exceeds the available space. One byte per index element is sufficient for the storage. Indices which are newly generated (i.e. as a forward neighbour of a previously generated active index) are inserted successively. Indices are never moved within the array or removed from the array.

For the description of the active and old index sets ($\mathscr{A}$ and $\mathscr{O}$) we use one–dimensional arrays `A` and `O` of maximum size m, respectively. Each entry in these arrays is the position of the corresponding index in the array `I`. In addition, the current number of indices in `A` and `O` denoted by `na` and `no` are stored (see Figure 3). Now, when an index is copied from `A` to `O`, only the entry to `I` has to be copied and not all its d elements. This way, the total dependence on d of the algorithm is reduced.

4.2 Active indices

So far we have not illustrated how the indices in `A` and `O` are stored. The required operations on the active and old index sets are quite different and therefore, we will arrange the two sets differently.

Let us first look at the set of active indices. The necessary operations are fast insertion and removal of indices. Furthermore, we have to be able to find the

`unsigned char I[m][d]`	entries of all indices
`int A[m]`	active indices
`int O[m]`	old indices
`double G[m]`	error estimates
`int N[m][2*d]`	neighbours
`int ni`	number of elements in I
`int na`	number of elements in A
`int no`	number of elements in O

Fig. 3. The data types and memory requirements for the dimension–adaptive algorithm

index with the largest associated error indicator. For the latter operation one clearly does not want to search through all the indices in order to find the current maximum every time (which would lead to a quadratic work complexity in the number of indices).

Let us first remark that we store the error indicators in an additional floating point array `G` of size m (with the same numbering as `I`, see Figure 3). We will here use a (at least in the computer science literature) well–known data structure called *heap* [31] which supports the required operations in the most efficient way. A heap is an ordering of the indices in `A` such that the error indicator for an index at position p is greater than (or equal to) the error indicators of the indices at positions $2p$ and $2p + 1$. This way, a binary tree hierarchy is formed on the set of indices where the index at the root (position 1) has the largest error indicator.

When the root index is removed (i.e. by putting it into the old index set), then the one of the two sons (at positions 2 and 3) with the larger error indicator is promoted as the new root. The vacancy that arises this way is filled with the son which possesses the larger error indicator and this scheme is repeated recursively until the bottom of the tree is reached.

4.3 Old indices

Similarly, when a new index is inserted (i.e. as the forward neighbour of the just removed index) it is first placed at the last position in the tree (i.e. it is assigned the highest position `na+1`). Now, if the error indicator of the father of the new index is smaller than its own error indicator, then the two positions are swapped. This procedure is repeated recursively until the error indicator of the current father is larger than that of the new index. This way, insertion and removal are functions which can all be performed in $O(\log(\texttt{na}))$ operations.

The required operations on the old index set are the insertion of indices and the checking if an index is admissible. Since indices are never removed from the old index set, the indices are stored in order of occurrence and insertion is simply done at the end of `O` (at position `no+1`). The check for admissibility is more

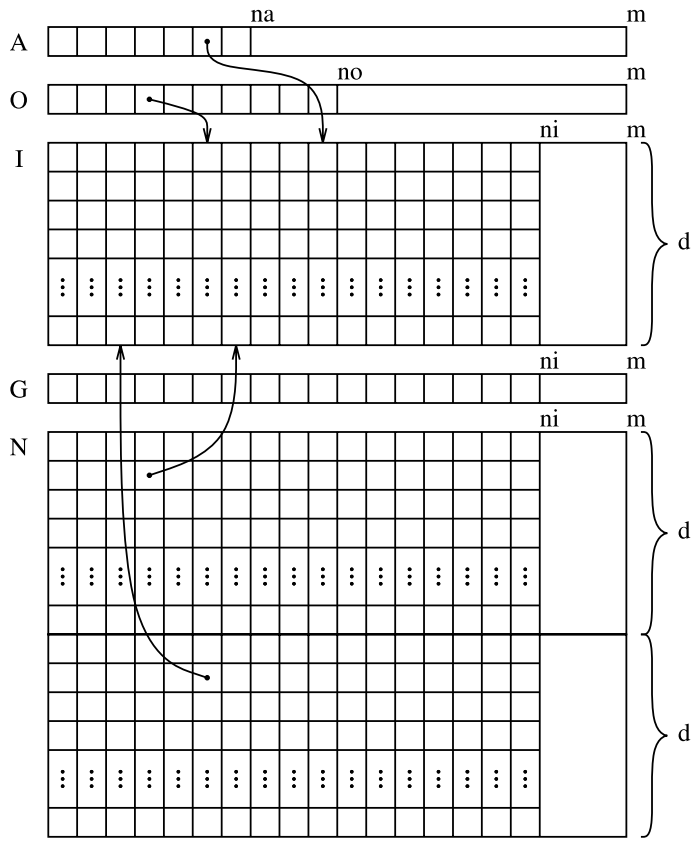

Fig. 4. A schematic representation of the data structures. Shown are the arrays for the active and old indices `A` and `O`, the index elements `I`, the error estimates `G`, and the neighbours `N`

difficult, though, since it involves the location of the whole backward neighbourhood of a given index. To this end, we explicitly store all the neighbours. For every index in both the active and old index sets the positions in `I` of the d forward neighbours and the d backward neighbours are stored. This requires an array `N` of size $m \times 2d$ where the first d entries are the forward and the second d entries the backward neighbours (see Figure 3). Note that the indices in I themselves already require $m \cdot d$ bytes. Thus, the overhead for the new array is not large. Note also that indices in the active index have only backward neighbours.

Now, let us discuss how the neighbour array is filled. Let us assume that a new index is generated as the forward neighbour of an active index $\mathbf{k}$ in direction i. The backward neighbour of the new index in direction i is known (the previously active index $\mathbf{k}$), but the $d-1$ other backward neighbours are unknown. Let us consider the backward neighbour in direction $j \neq i$. This backward neighbour can be found as the forward neighbour in direction i of the backward neighbour in direction j of the previously active index (see Figure 5). Put differently,

$$\bar{p}_j(p_i(\mathbf{k})) = p_i(\bar{p}_j(\mathbf{k})),$$

where p_i is the forward neighbour in direction i and $\bar{p}_j$ is the backward neighbour in direction j. In turn, when the backward neighbour in direction j is found, the new index is stored as its forward neighbour in direction j. This way, all required forward neighbours can be found and stored in the data structure. In summary, the construction of the neighbour array is done in constant additional time.

A new index is admissible if all backwards neighbours are in the array `O`. Indices in `O` can be distinguished from indices in `N` e.g. by looking at the first forward neighbour. Recall that indices in N do not have any forward neighbours and thus a marker (e.g. -1) may be used to identify them without additional storage. In summary, the admissibility check can now be performed in $O(d)$ operations.

4.4 Complexities

We will now discuss the space and time complexities of the algorithm. Concerning the time complexity we will distinguish between the work involved for the computation of the integral and the work overhead for the bookkeeping of the indices.

The memory requirement of the data types of Figure 3 is $(9d + 16)m + 12$ bytes. Additionally, the nodes and weights of the univariate quadrature formulas have to be stored (if they cannot be computed on–the–fly). This storage, however, can usually be neglected. In our experience, 257 quadrature nodes have proved to be more than enough for typical high–dimensional problems. In summary, the required memory is $O(d \cdot m)$ bytes (with a constant of about 9).

The amount of work required for $\Delta_{\mathbf{k}} f$ is $c \cdot n_{\mathbf{k}}$ where c is the cost of a function evaluation (which is at least $O(d)$). However, since the total cost depends on the size and structure of the index set, which is unknown beforehand, bounds for

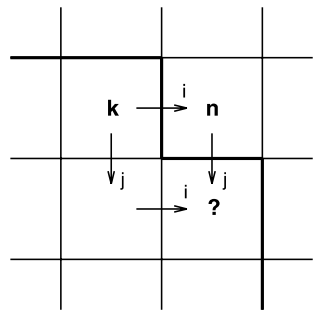

Fig. 5. Index **n** has just been generated as the forward neighbour in direction **i** of index **k**. The backward neighbour of **n** in direction $j \neq i$ can be found as the forward neighbour in direction **i** of the backward neighbour in direction j of **k**

the work required for the function evaluations can in general not be obtained. For the conventional sparse grid of level l, we know that this work is $O(2^l \cdot l^{d-1})$, but we hope that the work for a dimension–adapted grid is substantially smaller (especially concerning the dependence on d).

However, we can tell something about the work overhead for the bookkeeping of the indices. In view of Figure 1 we see that for each index which is put into `O` two *for* loops (over k and q) of size d are performed. In the outer loop, the new index is put into `A` which requires $O(\log \mathtt{na})$ operations. So, the worst case time complexity for bookkeeping is $O(d^2 + d \log \mathtt{na})$. Note that the average case complexity is smaller since the inner loop can be terminated early. In practice, the total overhead behaves like $O(d^2)$.

5. Numerical Examples

We will now study the performance of the dimension–adaptive algorithm in numerical examples. First, we consider some two–dimensional test functions which allows us to show the resulting grids and index sets. In these cases, the exact value of the integral is known (or can be computed quickly). The second example is a path integral of 32 dimensions in which the integral value is also known beforehand. The third example is a 256–dimensional application problem from finance where the exact value is unknown. As univariate quadrature formulas we use in all examples the well–known Patterson formulas [27] which have shown to be a good choice for the sparse grid construction [14].

5.1 Test examples (2 dimensions)

Let us first consider simple combinations of exponential functions defined over $[0,1]^2$. In Figures 6 and 7 we depict the old and active index sets as well as the resulting dimension–adapted sparse grids for some isotropic and some anisotropic functions, respectively. In these examples, the selected error threshold is $\mathtt{TOL} = 10^{-15}$. Note that the integration error was of the same magnitude. As weighting parameter for error estimation we use $w = 1$ since the functions are smooth. In all examples, the computation took less than 0.01s on a Pentium II.

The first example is a sum of one–dimensional functions. The dimension–adaptive algorithm correctly selects no indices in joint dimensions. Also, more points are placed in x–direction than in y–direction in the anisotropic case. Clearly, here the conventional sparse grid would spend too many points in joint directions. Although the function has additive structure, the index (2,2) is selected as active by the algorithm (because (1,2) and (2,1) are put into the old index set). Note that for a d–dimensional additive function with non–constant directions always $d(d-1)$ indices of this type are selected.

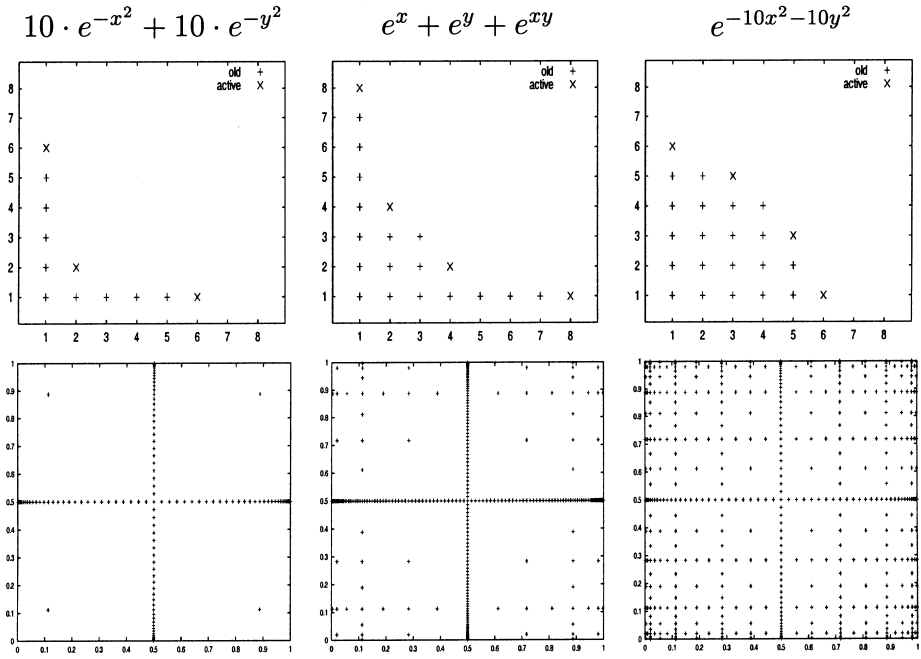

Fig. 6. The resulting index sets and corresponding sparse grids for $\texttt{TOL} = 10^{-15}$ for some isotropic test functions

The second example is not separable neither has product structure. The resulting index set is almost triangular, like the conventional sparse grid. However, the dimension–adaptive algorithm chooses to select more points on the axes while the conventional sparse grid would have spent too many points in the interior. In our experience, many application problems fall in this category which we would call nearly–additive.

The third example is the well known Gaussian hat function and has product structure. In this example, many points in joint dimensions are required. Here, the conventional sparse grid would have placed too few points there. This is an at first sight surprising result, since product functions should be easier integrable by a tensor product approach. However, the mixed derivatives of the Gaussian can get large even if they are bounded which reduces the efficiency of both the conventional sparse grid and the dimension–adaptive approaches.

5.2 Path integral (32 dimensions)

Let us now approach some higher–dimensional problems. We will first consider an initial value problem given by the linear parabolic differential equation

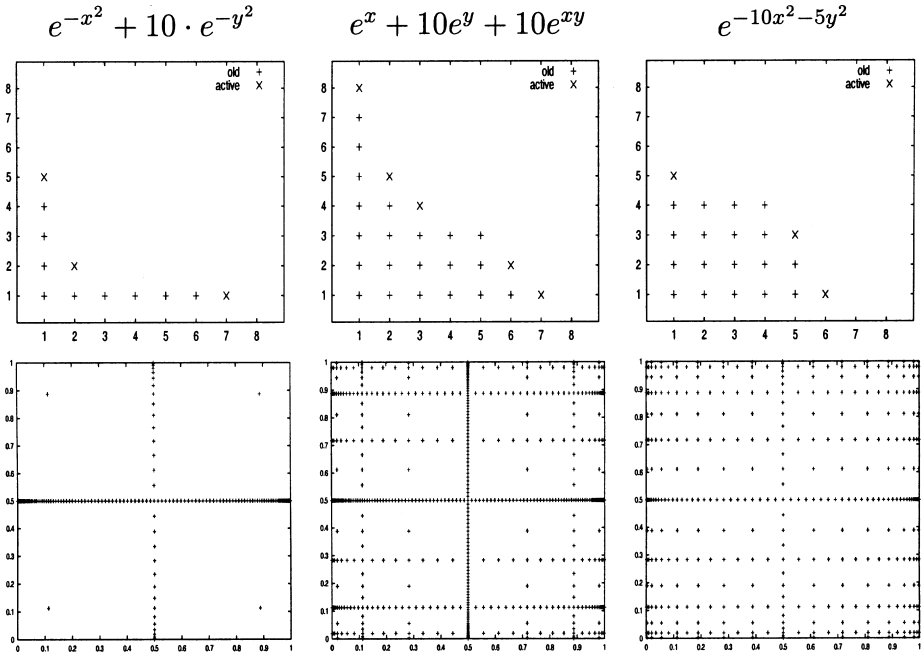

Fig. 7. The resulting index sets and corresponding sparse grids for $\texttt{TOL} = 10^{-15}$ for some anisotropic test functions

$$\frac{\partial u}{\partial t} = \frac{1}{2} \cdot \frac{\partial^2 u}{\partial x^2}(x,t) + v(x,t) \cdot u(x,t),$$

with initial condition $u(x,0) = f(x)$. The solution of this problem can be obtained with the Feynman–Kac formula as

$$u(x,t) = E_{x,0}\left(f(\xi(t)) \cdot e^{\int_0^t v(\xi(r),t-r)\, dr} \right),$$

where ξ represents a Wiener path starting at $\xi(0) = x$. The expectation $E_{x,0}$ can be approximated by a discretization of time using a finite number of time steps $t_i = i \cdot \Delta t$ with $\Delta t = t/d$. The integral in the exponent is approximated by a one–dimensional quadrature formula such as a sufficiently accurate trapezoidal rule.

The most natural way to discretize the Wiener path is by a random walk, i.e. by the recursive formula

$$\xi_k = \xi_{k-1} + \sqrt{\Delta t}\, z_k,$$

where $\xi_0 = x$ and z_k are normally distributed random variables with mean zero and variance one. The dimensions in the random walk discretization are all of the same importance since all the variances are identical to Δt.

In the Brownian bridge discretization [6], however, the path is discretized using a future and a past value

$$\xi_k = \frac{1}{2}(\xi_{k-h} + \xi_{k+h}) + \sqrt{\frac{h \cdot \Delta t}{2}} \cdot z_k.$$

Starting with $\xi_0 := x$ and $\xi_d := x + \sqrt{t}\, z_d$, the subsequent values to be computed are $\xi_{d/2}, \xi_{d/4}, \xi_{3d/4}, \xi_{d/8}, \xi_{3d/8}, \xi_{5d/8}, \xi_{7d/8}, \xi_{1d/16}, \xi_{3d/16}, \ldots$ with corresponding $h = 1/2, 1/4, 1/4, 1/8, 1/8, 1/8, 1/8, 1/16, 1/16, \ldots$. The Brownian bridge leads to a concentration of the total variance in the first few steps of the discretization and thus to a weighting of the dimensions.

Let us now consider the concrete example [24]

$$v(x,t) = \left(\frac{1}{t+1} + \frac{1}{x^2+1} - \frac{4x^2}{(x^2+1)^2}\right),$$

with initial condition $u(x,0) = \frac{1}{x^2+1}$. The exact solution is then

$$u(x,t) = \frac{t+1}{x^2+1}.$$

The results for $d = 32$, $t = 0.02$ and $x = 0$ are shown in Figure 8 (left). We see the integration error plotted against the number of function evaluations in a log–log scale. Here, the conventional sparse grid method is compared with the dimension–adaptive algorithm for the random walk and Brownian bridge discretizations. In this example, the conventional sparse grid is for the random walk discretization obviously close to the optimum since the dimension–adaptive method cannot improve on the performance. The conventional sparse grid gains no advantage from the Brownian bridge discretization, but the convergence rate of the dimension–adaptive algorithm is dramatically improved. Note that the convergence rate of the Quasi–Monte Carlo method (with Brownian bridge) is

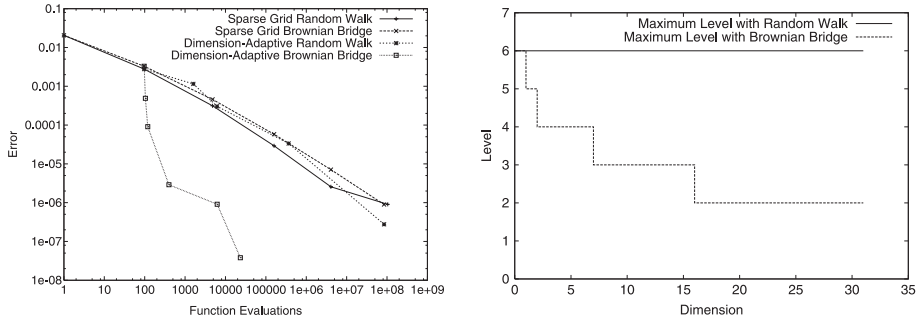

Fig. 8. Computational results for the path integral ($d = 32$): integration error vs. number of function evaluations (left) and maximum level over all dimensions (sorted) for the dimension–adaptive algorithm with Brownian bridge discretization (right)

comparable to that of the conventional sparse grid approach [24, 14]. In Figure 8 (right) we plot the maximum level per dimension of the final index set of the dimension–adaptive method without and with the Brownian bridge discretization. Here, the dimensions are sorted according to this quantity. For the Brownian bridge discretization, the maximum level decays with the dimension. This shows that only a few dimensions are important and thus contribute substantially to the total integral while the other dimensions add significantly less.

5.3 CMO problem (256 dimensions)

Let us now consider a typical collateralized mortgage obligation problem, which involves several tranches which in turn derive their cash flows from an underlying pool of mortgages [6, 26]. The problem is to estimate the expected value of the sum of present values of future cash flows for each tranche. Let us assume that the pool of mortgages has a 21 1/3 year maturity and cash flows are obtained monthly. Then, the expected value requires the evaluation of an integral of dimension $d = 256$ for each tranche,

$$\int_{\mathbb{R}^d} v(\xi_1, \ldots, \xi_d) \cdot g(\xi_1) \cdot \ldots \cdot g(\xi_d) \quad d\xi_1 \ldots d\xi_d,$$

with Gaussian weights $g(\xi_i) = (2\pi\sigma^2)^{-1/2} e^{-\xi_i^2/2\sigma^2}$. The sum of the future cash flows v is basically a function of the interest rates i_k (for month k),

$$i_k := K_0 e^{\xi_1 + \cdots + \xi_k} i_0$$

with a certain normalizing constant K_0 and an initial interest rate i_0 (for details see [6], first example, and [14, 26]). Again the interest rates can either be discretized using a random walk or the Brownian bridge construction. For the numerical computation, the integral over $\mathbb{R}^d$ is transformed to an unweighted integral on $[0, 1]^d$ with the help of the inverse normal distribution.

In Figure 9 we again compare the conventional sparse grid method with the dimension–adaptive method for the random walk and the Brownian bridge discretization. The error is computed against an independent Quasi–Monte Carlo calculation. Note that also in this example the convergence rate of the conventional sparse grid approach is comparable to the Quasi–Monte Carlo method [14].

We see that again a weighting of the dimensions does not influence the convergence of the conventional sparse grid method. But for the dimension–adaptive method the amount of work is again substantially reduced (by several orders of magnitude) for the same accuracy when the Brownian bridge discretization is used and thus higher accuracies can be obtained. In this example the dimension–adaptive method also gives better results than the conventional sparse grid method for the random walk discretization. This implies that the conventional sparse grid spends too many points in mixed dimensions for this problem. The problem seems to be intrinsically lower–dimensional and nearly additive [6].

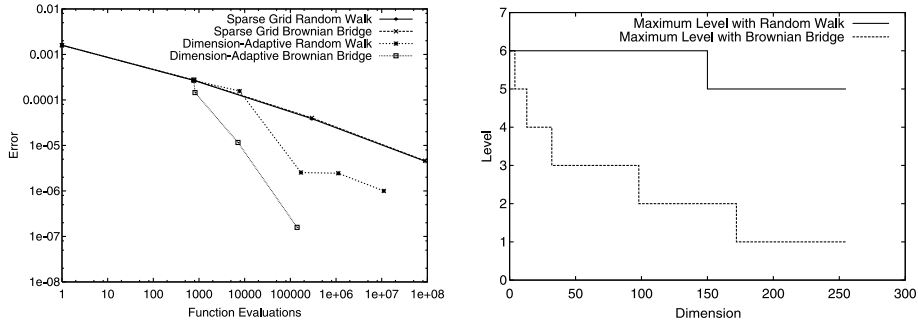

Fig. 9. Computational results for the CMO problem ($d = 256$): integration error vs. number of function evaluations (left) and maximum level over all dimensions (sorted) for the dimension–adaptive algorithm with and without Brownian bridge discretization (right)

6. Concluding Remarks

In this paper, we have presented a dimension–adaptive algorithm for the numerical integration of multivariate functions. The method which can be seen as a generalization of the sparse grid method tries to find important dimensions automatically and places more integration points there. We have also discussed the implementation of the algorithm and proposed data structures which allow for the efficient bookkeeping of the sparse grid index sets.

We have shown that this algorithm can substantially improve the convergence rate of the conventional sparse grid method through a reduction of the dependence on the dimension. This behaviour has been confirmed in numerical experiments and in application problems from computational physics and finance. In these examples, the dimension–adaptive algorithm was clearly superior to the Monte Carlo and Quasi–Monte Carlo methods.

Let us finally remark that the whole approach is not restricted to integration problems but can also be used for interpolation, the solution of partial differential and integral equations, or eigenvalue problems for the high–dimensional case. There, the possible application areas include computer simulations in statistical physics and chemistry, queueing theory, and data mining.

References

[1] Bank, R.: Hierarchical bases and the finite element method. Acta Numerica *5*, 1–43 (1996).
[2] Bellman, R.: Dynamic Programming. Princeton: University Press 1957.
[3] Bonk, T.: A new algorithm for multi–dimensional adaptive numerical quadrature. In: Hackbusch, W., Wittum, G., (eds.) Adaptive methods: algorithms, theory and applications, volume 46 of Notes on Numerical Fluid Mechanics. Braunschweig: Vieweg 1993.
[4] Bungartz, H.-J.: Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson–Gleichung. PhD thesis, Institut für Informatik, TU München, 1992.
[5] Bungartz, H.-J., Griebel, M.: A note on the complexity of solving Poisson's equation for spaces of bounded mixed derivatives. J. Complexity *15*, 167–199 (1999).
[6] Caflisch, R., Morokoff, W., Owen, A.: Valuation of mortgage backed securities using Brownian bridges to reduce effective dimension. J. Comput. Finance 1, (1997).

[7] Davis, P., Rabinowitz, P.: Methods of numerical integration. Academic Press, 1975.
[8] DeVore, R.: Nonlinear approximation. Acta Numerica *7*, 51–150 (1998).
[9] Dirnstorfer, S.: Adaptive numerische Quadratur höherer Ordnung auf dünnen Gittern. Master's thesis, Institut für Informatik, TU München, 2000.
[10] Dörfler, W.: A robust adaptive strategy for the nonlinear Poisson equation. Computing *55*, 289–304 (1995).
[11] Friedman, J.: Multivariate adaptive regression splines. Annals of Statistics *19*, 1–141 (1991).
[12] Garcke, J., Griebel, M.: Classification with anisotropic sparse grids using simplicial basis functions. Intelligent Data Analysis *6*(6), 483–502 (2002).
[13] Genz, A., Malik, A.: An adaptive algorithm for numerical integration over an n-dimensional rectangular region. J. Comp. Appl. Math. *6*, 295–302 (1980).
[14] Gerstner, T., Griebel, M.: Numerical integration using sparse grids. Numer. Algorithms *18*, 209–232 (1998).
[15] Griebel, M., Knapek, S.: Optimized tensor-product approximation spaces. Constructive Approximation *16*(4), 525–540 (2000).
[16] Hastie, T., Tibshirani, R.: Generalized additive models. London: Chapman and Hall 1990.
[17] He, T.-X.: Dimensionality reducing expansion of multivariate integration. Birkhäuser 2001.
[18] Hegland, M.: Adaptive sparse grids. In: Proceedings of CTAC, Brisbane, July 16–18, 2001, 2001.
[19] Hegland, M., Pestov, V.: Additive models in high dimensions. Technical Report 99–33, School of mathematical and computing sciences, Victoria University of Wellington, 1999.
[20] Kalos, M., Whitlock, P.: Monte Carlo Methods. Wiley & Sons 1986.
[21] Khavinson, S.: Best approximation by linear superposition (approximate nomography). AMS Translations of mathematical monographs vol. 159. Providence: AMS 1997.
[22] Kolmogoroff, A.: On the representation of continuous functions of several variables by superpositions of continuous functions of fewer variables. Dokl. Akad. Nauk SSSR *108*, 179–182 (1956). (in Russian, Engl. Transl.: Amer. Math. Soc. Transl. (2) *17*, 369–373 (1961)).
[23] Kolmogoroff, A.: On the representation of continuous functions of several variables by superpositions of continuous functions of one variable and addition. Dokl. Akad. Nauk SSSR *114*, 953–956 (1957). (in Russian, Engl. Transl.: Amer. Math. Soc. Transl. (2) *28*, 55–59 (1963)).
[24] Morokoff, W., Caflisch, R.: Quasi–monte carlo integration. J. Comp. Phys. *122*, 218–230 (1995).
[25] Niederreiter, H.: Random number generation and quasi–Monte Carlo methods. Philadelphia: SIAM 1992.
[26] Paskov, S., Traub, J.: Faster valuation of financial derivatives. J. Portfolio Management *22*, 113–120 (1995).
[27] Patterson, T.: The optimum addition of points to quadrature formulae. Math. Comp. *22*, 847–856, 1968.
[28] Plaskota, L.: The exponent of discrepancy of sparse grids is at least 2.1933. Adv. Comp. Math. *12*, 3–24 (2000).
[29] Rassias, T., Simsa, J.: Finite sums decompositions in mathematical analysis. Chichester: Wiley & Sons 1995.
[30] Röschke, D.: Über eine Kombinationstechnik zur Lösung partieller Differentialgleichungen. Master's thesis, Institut für Informatik, TU München, 1991.
[31] Sedgewick, R.: Algorithms in C. Addison Wesley, 1990.
[32] Simsa, J.: The best L^2-approximation by finite sums of functions with separable variables. Aequationes Mathematicae *43*, 284–263 (1992).
[33] Smolyak, S. A.: Interpolation and quadrature formulas for the classes W_s^a and E_s^a. Dokl. Akad. Nauk SSSR 131, 1028–1031 (1960). (in Russian, Engl. Transl.: Soviet Math. Dokl. *4*, 240–243 (1963)).
[34] Traub, J., Wasilkowski, G., Woźniakowski, H.: Information–based complexity. New York: Academic Press 1988.
[35] Van Dooren, P., De Ridder, L.: An adaptive algorithm for numerical integration over an *n*–dimensional cube. J. Comp. Appl. Math *2*, 207–217 (1976).
[36] Verführt, R.: A review of a posteriori error estimation and adaptive mesh–refinement techniques. Teubner, 1996.
[37] Wahba, G.: Spline models for observational data. Philadelphia: SIAM 1990.
[38] Wasilkowski, G. W., Woźniakowski, H.: Explicit cost bounds of algorithms for multivariate tensor product problems. J. Complexity *11*, 1–56 (1995).
[39] Wasilkowski, G. W., Woźniakowski, H.: Weighted tensor product algorithms for linear multivariate problems. J. Complexity *15*, 402–447 (1999).
[40] Yue, R., Hickernell, F.: Robust designs for smoothing spline ANOVA models. Metrika *55*, 161–176 (2002).

[41] Zenger, C.: Sparse grids. In: Hackbusch, W., (ed.) Parallel algorithms for partial differential equations, volume 31 of Notes on Numerical Fluid Mechanics. Braunschweig: Vieweg 1991.

Thomas Gerstner
Michael Griebel
Department for Applied Mathematics
University of Bonn Wegelerstr. 6 D-53115 Bonn
Germany
e-mail: {gerstner,griebel}@iam.uni-bonn.de