Technische Universität München
Fakultät für Informatik

Diplomarbeit:

Numerical quadrature on sparse grids

Stefan Dirnstorfer

Contents

1 Introduction **3**

2 Univariate quadrature **4**
2.1 Gaussian and Gauss-Patterson Formulas 4
2.2 Hierarchical quadrature 5
2.2.1 Hierarchical basis functions 5
2.2.2 Polynomial Basis functions 7

3 Multivariate quadrature **8**
3.1 Monte Carlo and quasi-Monte Carlo 8
3.2 Full grid 9
3.3 Smolyak's Construction 9
3.4 Tensor product basis 10

4 Basis functions **12**
4.1 Gauss formulas 12
4.2 Piecewise Gauss polynomials 14
4.3 Polynomial degree of exactness 15
4.4 Integration error 16
4.5 Non adaptive numerical example 17

5 Adaptivity **19**
5.1 Construction 19
5.2 A good example 20
5.3 A bad example 22
5.4 Balanced adaptivity 23
5.5 A two dimensional example 25
5.6 Numerical results 27
5.7 Adaptive balance 29

6 Efficient data structure **31**
6.1 Integer vector 31
6.2 The bit string 32
6.3 Speed improvements 33

7 Numerical Results **35**
7.1 Test Function 35
7.2 Absorption Problem 37
7.3 CMO Problem 40
7.4 Asian option 43
7.5 Run times and complexities 46

8 Conclusion **48**

Bibliography **49**

1 Introduction

Multivariate integrals in up to several hundred dimensions often arise in statistical or physical computations. Many conventional quadrature algorithms suffer from the "curse of dimension", i.e. have computational costs growing exponentially with the dimension. Others have computing costs independent from the dimension but their accuracy is often low. Sparse grids are known to have a complexity independent of the number of dimensions up to a logarithmic factor. Different sparse grid implementations are based on integration rules suitable for certain function classes. Many integration rules lead to grids with only little room for adaptivity. The performance of such grids has been compared for a variety of integrands [4]. A sparse grid based algorithm which fully exploits adaptivity has been presented by Hans Bungartz [2]. Although this algorithm can adapt to a wide range of function classes, it exploits only little a priori knowledge about smooth integrands. This makes it inferior to non adaptive rules tuned for the class of smooth functions.
The scope of this publication is to provide some tuning to the adaptive integration rule introduced in [2]. Special emphasis was put on smooth functions, as this is the main disadvantage of this adaptive algorithm, compared to non adaptive sparse grids. The quadrature rule was improved such that polynomials with a higher degree can be integrated with the same amount of function evaluations. Then the adaptive strategy had to be adjusted in order to exploit the higher polynomial order of exactness in adaptive grids. Moreover, a new data structure was established, which considerably reduces computation time and storage requirements. Finally the performance of the new quadrature algorithm is compared to other quadrature rules in various examples.

2 Univariate quadrature

A sparse grid construction transforms a one-dimensional quadrature rule into a multivariate rule. The resulting quadrature formula generally keeps the properties of the underlying rule. It integrates the class of multivariate functions with a comparable performance as it does for the equivalent univariate class and provides similar options for adaptive refinement. It is therefore important to have a look at different quadrature rules.
Every quadrature formula Q evaluating the function f at the positions $x_1, .., x_n$ estimates the integral on a domain $[a..b]$ and can be written in the following way:

$$\int_a^b f(x)\,dx \approx Q\,f = \sum_{k=1}^{n} w_k\, f(x_k). \tag{2.1}$$

Whereas the w_k are called weights and the x_k nodes. A quadrature rule can be fully determined by the number of nodes n, the abscissas x_k and the weights w_k. The most important performance measure of these quadrature rules is the number of function evaluations that are necessary to approximate the integral up to a certain accuracy. This translates into computational costs, since evaluating the function is often one of the most time consuming tasks and exact run times heavily depend on implementation details.

2.1 Gaussian and Gauss-Patterson Formulas

With the Gauss rule it is possible to achieve the maximum possible polynomial degree of exactness. With n function evaluations polynomials of degree $\leq 2n-1$ can be integrated exactly [3].

Theorem: Let $x_1, .., x_n$ be the zeros of the n-th Legendre polynomial. Then there exist weights w_k, such that

$$\int_{-1}^{1} p(x)\,dx = \sum_{k=1}^{n} w_k\, p(x_k) \tag{2.2}$$

for all polynomials $p(x)$ with degree $\leq 2n-1$.

There are slight variations of this Gauss rule optimized for function classes different than polynomials. However Gauss based rules are generally assumed to be the optimal rule with points generated by a priori knowledge only. It is possible to adjust the Gauss rule towards more flexibility with a slight loss of accuracy.
The nodes resulting from the Gauss quadrature rule are generally not nested, meaning that a formula with a higher number of nodes does not necessarily

contain all the nodes of rules with lower numbers. A nested rule has two advantages. First, a more accurate rule can be applied and reuse all previous function evaluations, such that the number of points is increased until the error is within a given tolerance. And second, it will be shown that sparse grids based on nested quadrature rules are much more efficient.
The Gauss-Patterson rule can be built with $2n+1$ points containing all the points of an n point rule. The polynomial order of exactness with $2n+1$ points is $2n+\overline{n}+2$. Where $\overline{n}$ is n, if n is odd, and $n-1$, if n is even.
Further details about the Gauss-Patterson rule can be found in [4]. It is introduced here for the purpose of completeness and will be used for a comparison of two different sparse grid approaches.

2.2 Hierarchical quadrature

The first kind of hierarchical quadrature was established by Archimedes. He integrated the parabola $1-x^2$ by continuously adding triangles to the area. Figure 2.1 shows three triangles approximating a parabola.

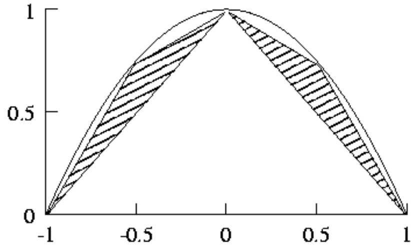


Figure 2.1: Archimedian integration

The biggest triangle has an area of 1. The two smaller triangles have an area of $\frac{1}{8}$. Adding these three and all the further triangles Archimedes derived the parabola's area:

$$1+2\times\frac{1}{8}+4\times\frac{1}{64}+\cdots=\frac{4}{3}. \tag{2.3}$$

The biggest advantage of this rule is that additional triangles can be added adaptively. There is no need to know their number and position before the first point is evaluated. The size of the added triangles indicates how far the area of all the triangles differs from the exact integral. Although this rule is still quite primitive it is a good basis for a fully adaptive quadrature algorithm.

2.2.1 Hierarchical basis functions

On the basis of the Archimedian quadrature rule, more sophisticated rules can be established. First, a formal mathematical representation of the existing rule is explained.
The mother of all triangles is the one dimensional hat function $\Phi(x)$ on the domain $[-1,1]$. It can be written as:

$$\Phi(x) := \begin{cases} 1-|x| & \text{for } x\in[-1,1] \\ 0 & \text{otherwise} \end{cases} . \tag{2.4}$$

Smaller functions can be defined by a level l and an index i, resulting in a number of translated and dilated hats.

$$\Phi_{l,i}(x) := \Phi\left(2^{l-1}(x+1) - 2i + 1\right) \qquad \text{with } 1 \leq i \leq 2^{l-1} \tag{2.5}$$

The supports of all functions on a single level are mutually exclusive and partition the domain $[-1, 1]$. On the support of a function on a level l, there live two functions on level $l+1$. These two smaller functions will be called left and right **son**, whereas the original function on level l will be called the **father**. Figure 2.2 shows the functions of level one to four

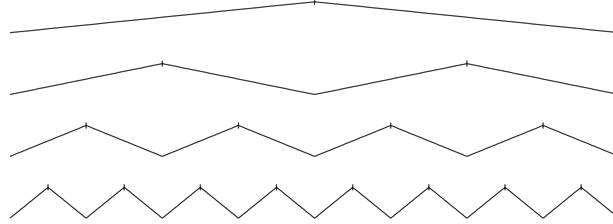


Figure 2.2: Hierarchical basis functions

Using functions up to a level L, a basis of piecewise linear functions with support on $[-1, 1]$ and $2^L - 1$ equidistant nodes can be built. The following formula shows a function approximation with hat functions.

$$f(x) \approx \sum_{l=1}^{L} \sum_{i=1}^{2^{l-1}} c_{l,i}\, \Phi_{l,i}(x) \tag{2.6}$$

The coefficients $c_{l,i}$ are called the **hierarchical surpluses**. They are the difference between the function $f(x)$ and the interpolation on the previous level. In the case of the Archimedian parabola these are $c_{l,i} = (1/4)^{l-1}$. This approximation can be easily integrated, by integrating the one-dimensional hat functions.

$$\int_{-1}^{1} f(x)\, dx \approx \int_{-1}^{1} \sum_{l=1}^{L} \sum_{i=1}^{2^{l-1}} c_{l,i}\, \Phi_{l,i}(x)\, dx = \sum_{l=1}^{L} \sum_{i=1}^{2^{l-1}} c_{l,i} \int_{-1}^{1} \Phi_{l,i}(x)\, dx \tag{2.7}$$

Nodal basis representation

In order to express the hierarchical quadrature formula with the general quadrature formula (2.1) the basis functions can be transformed into a nodal basis. For a given level l the nodal basis spans the same function space. The functions $\Phi_i(x)$ look like this:

$$\Phi_i(x) = \Phi\left(2^{l-1}(x+1) - i\right) \qquad \text{with } 1 \leq i < 2^l. \tag{2.8}$$

The nodes x_i of these functions are in the center of the basis function and the weights w_i are the volume of the basis function, i.e. $w_i = 2^{l-1}$. The hierarchical quadrature based on hat functions $\Phi(x)$ is equivalent to a trapezoidal rule integrating a function that is zero on -1 and 1. Its advantage however is that on higher levels basis functions can be inserted adaptively.

2.2.2 Polynomial Basis functions

Instead of simple hat functions, a hierarchical quadrature algorithm can be based on piecewise polynomials. A polynomal basis can be constructed such that on level l polynomials of degree l can be integrated exactly. The supports and nodes are the same as for the previous hat function basis. The polynomials are constructed such that they interpolate zero on all their fathers nodes and are normalized to be 1 at their own node. The first level starts with two linear functions having their nodes on the domains borders. Figure 2.3 shows the resulting piecewise polynomial basis.

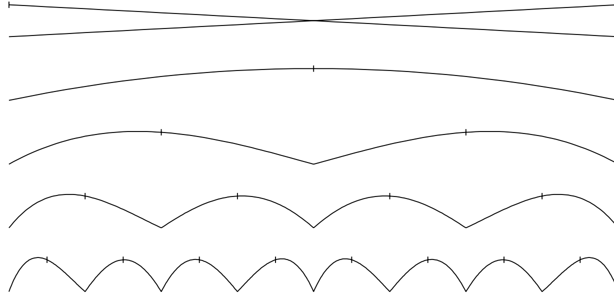


Figure 2.3: Polynomial basis functions

A major problem of these polynomials is that they have nodes on the borders. Moving to a multidimensional integration rule, it will be shown that the most simple grid would have 2 points in each direction, leading to a total of 2^{dim} points located at the domains corners. Evaluating a very high dimensional function at these positions can easily exceed today's computing capabilities.
A possible solution is to start with only one constant function on level one. For more than one dimension the minimum amount of nodes drops to 1^{dim}. The drawback is that on level l only polynomials of degree $l-1$ can be integrated exactly. Figure 2.4 shows the basis functions resulting from this advanced rule, as they were used by Hans Bungartz for adaptive high dimensional integration. In section 4 further improvements will be derived.

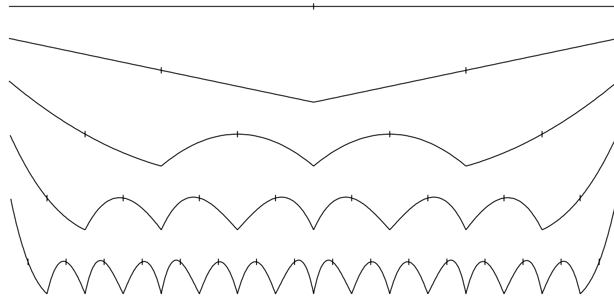


Figure 2.4: Polynomial basis functions

3 Multivariate quadrature

There are two main classes of multivariate quadrature formulas based on random or regular grids. Random grids, also known as Monte Carlo methods, spread their points on the functions domain according to a generated sequence of random or quasi-random numbers. Regular grids are constructed by a well defined algorithm, which is typically based on univariate rule. There are different ways to extend a univariate quadrature rule to higher dimensions. The main goals are to keep the number of points low and to preserve the convergence properties found in the first dimension. Sparse grids have been proven to satisfy these demands and are known by different names, like combination techniques [2] or Smolyak construction [4]. Depending on the univariate rule, sparse grids can appear in different forms. In this section the basic construction is introduced, but the main focus will be put on hierarchical quadrature based grids.

3.1 Monte Carlo and quasi-Monte Carlo

The standard Monte Carlo method evaluates a function on random points, distributed uniformly over the unit cube. With a sequence of random points (x_i), a d-dimensional integral is obtained by averaging the evaluated function values.

$$\int_{[0,1]^d} f(x)\,dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{3.1}$$

The integration error is $O(N^{-1/2})$. It is completely independent of the dimension and of the function's smoothness properties. The Monte Carlo method is the rule of choice for very high dimensional and discontinuous problems, but has poor performance otherwise.

Random vectors with independent random coordinates tend to point clumpings or holes, since there is a certain chance of some points to land very near to each other. Convergence can be improved by a more uniformly distributed sequence of quasi-random numbers, also refered as low discrepancy sequences. Halton, Sobol' and Faure sequences will be compared to sparse grids in the final section [7]. Quasi-Monte Carlo methods generally have error bounds of size $O((logN)^{dim}N^{-1})$. Due to the low algorithmic overhead and good performance for a wide range of functions, quasi-Monte Carlo methods are currently used for most higher dimensional integration problems.

3.2 Full grid

The most simple way to create a multivariate quadrature rule is the full grid, with equidistant points in every direction. For a given one dimensional quadrature rule $Q_l^{(1)}$ using n_l points, a d-dimensional rule $Q_l^{(d)}$ can be built recursively.

$$Q_l^{(d)} f = Q_l^{(1)} \otimes Q_l^{(d-1)} f \tag{3.2}$$

Whereas $\otimes$ is the tensor product for two univariate quadrature rules Q' and Q'' is defined as follows:

$$\begin{aligned} Q' f &= \sum_{i=1}^{n_1} w_{1,i}\, f(x_{1,i}) \\ Q'' f &= \sum_{i=1}^{n_2} w_{2,i}\, f(x_{2,i}) \\ (Q' \otimes Q'')\, f &= Q' \left(\sum_{i=1}^{n_2} w_{2,i}\, f(\cdot, x_{2,i}) \right) \\ &= \sum_{j=1}^{n_1} w_{1,j} \left(\sum_{i=1}^{n_2} w_{2,i}\, f(x_{1,j}, x_{2,i}) \right). \end{aligned} \tag{3.3}$$

The total amount of evaluated points is ${n_l}^d$, i.e. grows exponentially with the dimensionality, refering to the "curse of dimension". A quadrature rule with only two points in one direction will in hundred dimensions already use $2^{100} \approx 10^{30}$ points.

3.3 Smolyak's Construction

Smolyak constructed a multidimensional grid with fewer points [8]. A univariate quadrature rule $Q_l^{(1)}$ on level l uses n_l points. Based on this rule, the d-dimensional sparse grid $Q_l^{(d)}$ can be built.

$$Q_l^{(d)} f = \left(\sum_{i=0}^{n_l} \left(Q_i^{(1)} - Q_{i-1}^{(1)} \right) \otimes Q_{l-i}^{(d-1)} \right) f \tag{3.4}$$

This construction is the basis of a wide range of sparse grid algorithms. Depending on the underlying rule $Q_l^{(1)}$ there are different complexities for the error and the number of points.
Since the function is integrated with integration rules of different levels, the number of points per level is considerably lower if the univariate rule is nested and evaluated points can be reused. Hierarchical rules are always nested and complexity bounds will be given below.
Another important feature of this grid is that the polynomial degree of exactness is preserved [4]. If $Q_l^{(1)}$ integrates a polynomial of degree n exactly, then $Q_l^{(d)}$ integrates a d-dimensional polynomial $\sum_{|i|_1 \le n} c_i\, x_1^{i_1} \cdots x_d^{i_d}$ exactly.

3.4 Tensor product basis

The Smolyak and the full grid construction create a higher dimensional quadrature rule by combining the univariate rules with the tensor product. Applying this tensor product construction to the hierarchical quadrature, the univariate basis functions are transformed to a set of multi-dimensional basis functions. A full and a sparse grid are distinguished by the subset of functions which are used on each level.
The multi-variate hierarchical quadrature uses a tensor product basis. Each basis function is a product of a univariate function Φ_{l_k,i_k} for each direction k. A multi dimensional function $\Phi_{\underline{l},\underline{i}}$ can be constructed as follows:

$$\Phi_{\underline{l},\underline{i}}(\underline{x}) = \prod_{k=1}^{dim} \Phi_{l_k,i_k}(x_k). \tag{3.5}$$

Figure 3.1 shows an example product of piecewise linear functions of level one and two.

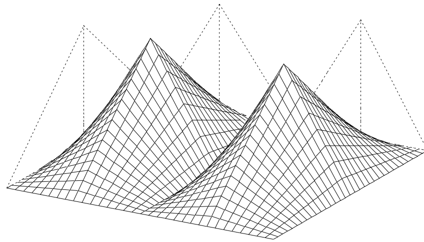


Figure 3.1: Tensor product basis

The Smolyak construction implies, which of these basis functions are used in a straight-forward implementation. The sparse grid is constructed for a certain level l. All basis functions in this grid have a sum of the univariate levels being less or equal to $l + dim - 1$. Each basis function has therefore a level, representing the sparse grid level on which it appears first. This function level basically sums up the levels l_k of the univariate functions.

$$level(\Phi_{\underline{l},\underline{i}}) = \left(\sum_{k=1}^{dim} l_k\right) - dim + 1 \tag{3.6}$$

Figure 3.2 demonstrates this fact more clearly for two-dimensional grids up to level three. As indicated by the one-dimensional basis functions plotted next to the coordinate axes, the grid is based on the hat functions described previously. The supports and the nodes shown here, are the same for the improved polynomial basis. On the first level the grid contains only the function on the top left. On level two there are five functions. All 17 functions build a grid of level three. An important property, which will influence the adaptive strategy, is that the supports of the functions on a new level can be constructed by subdividing the old supports along each direction.

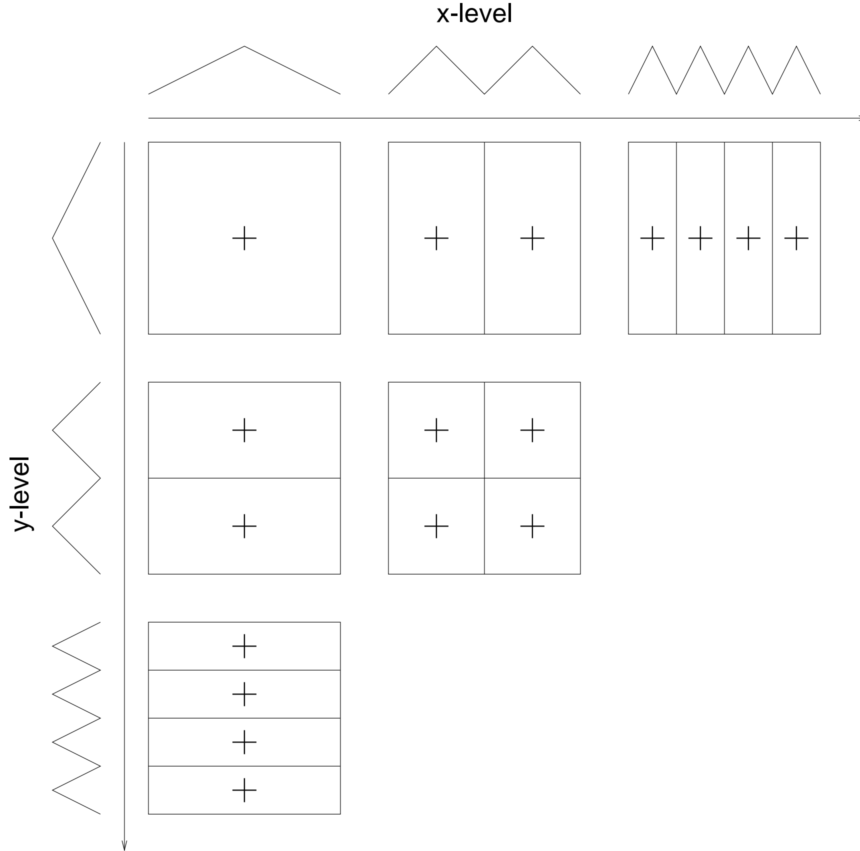


Figure 3.2: Sparse grid construction

Error bounds

For this hierarchical sparse grid several properties have been proven in [2]. The most interesting complexity is the number of necessary points N to achieve a certain accuracy. A simple relationship exists for the hat function based grid and integrands u with existing and continuous mixed derivatives

$$D^{\alpha}u := \frac{\delta^{|\alpha|_1}u}{\delta x_1^{\alpha_1}\cdots\delta x_d^{\alpha_d}} < \infty. \tag{3.7}$$

Measuring the interpolation error ϵ in the infinity norm, the following relation holds for the number of necessary grid points N:

$$N(\epsilon) = O(\epsilon^{-\frac{1}{2}}|log_2\epsilon|^{\frac{3}{2}(d-1)}). \tag{3.8}$$

Or, the other way around, expressing the error resulting of N function evaluations:

$$\epsilon(N) = O(N^{-2}|log_2 N|^{3(d-1)}). \tag{3.9}$$

In case of the piecewise polynomial basis, better error bounds exist [2]. The dimension only appears in the potency of a logarithmic term. Therefore the sparse grid is considered to solve the problem of the “curse of dimension” to some extent.

4 Basis functions

The multi-dimensional quadrature rule resulting from a Smolyak construction depends on the choice of the one-dimensional rule. Various integration rules differ in the position and weights of function evaluations. A static rule derives these values from an a priori knowledge of the integrated function. The Gauss rule determines positions and weights of a certain number of points to optimally integrate a certain type of functions. A more flexible rule can decide position and weights on the basis of function evaluations during the integration process. The Gauss-Patterson rule can introduce new points interlacing all the old ones. Local refinement is not possible. A third, most flexible hierarchical rule, based on piecewise polynomials, allows the introduction of new points between two existing points. The points can be focused on the interesting parts at the expense of a less than optimal choice for smooth functions.

Sparse grids based on the Gauss and the Gauss-Patterson rule have been presented and proven to perform well on a wide variety of smooth functions [4]. However, there is little room for further tuning, due to the low flexibility of the quadrature rule. A sparse grid algorithm based on a piecewise polynomial basis can place function evaluation following local features of a function. The basis functions introduced in [2] produce locally refined points that lie exactly in the middle of two existing points. When a smooth function is integrated, this leads to points lying equidistant over the whole domain, whereas Gauss proved them to integrate best with positions concentrated on the sides of the domain. Adapting the formulas found by Gauss to piecewise polynomials can increase their performance on smooth functions considerably.

4.1 Gauss formulas

Gauss quadrature rules are usually constructed to exactly integrate polynomials with the highest possible degree. Optimal positions can be computed for a certain number of points or can be determined to extend a certain number of existing points. The Gauss-Patterson rule exploits this kind of Gauss formula to extend an n point rule by $n+1$ new points. A piecewise polynomial quadrature rule must be able to insert new points individually with some control over their position.

A polynomial basis consists of a number of basis polynomials. Integration with piecewise basis polynomials is like using normal polynomials, when only a small interval $[a, b]$ is considered. Figure 4.1 shows the hierarchical polynomial pieces which are involved in the approximation of an integrand on the interval $[a, b]$.

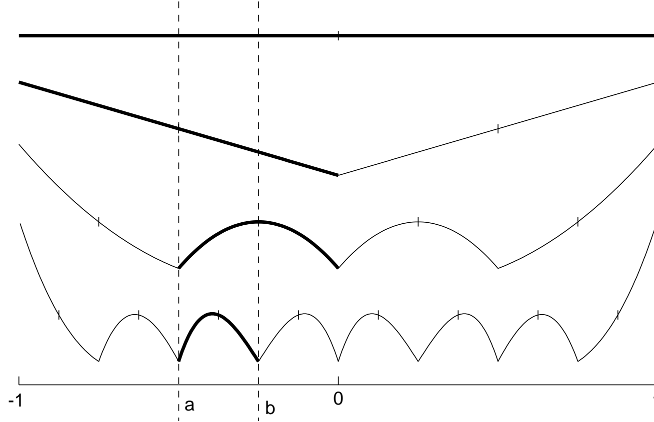


Figure 4.1: Hierarchical basis functions involved in the interval [a,b]

The polynomial basis functions $b_1(x), .., b_n(x)$ with $x_1, ..., x_n$ being the corresponding nodes should be constructed to integrate a polynomial $p(x)$ of highest possible degree on the domain $[a..b]$ exactly. A polynomial is integrated exactly, if there are coefficients $c_1, .., c_n$ depending only on evaluated points $p(x_1), .., p(x_n)$, such that:

$$\int_a^b \left(\sum_{i=1}^{n} c_i b_i(x) \right) dx = \int_a^b p(x) dx. \tag{4.1}$$

The c_i are the surpluses in a hierarchical basis and are $c_i = p(x_i)$ in nodal basis. The polynomial basis established in [2] allows exact integration of polynomials with degree $n-1$ or smaller with n basis functions. Polynomials of degree n can only be integrated with an additional basis function. As discussed previously, a new function is constructed, by interpolating the old nodes and normalizing it to one at the new node x_{n+1}.

$$b_{n+1}(x) = \begin{cases} \dfrac{\prod_{i=1}^{n}(x - x_i)}{\prod_{i=1}^{n}(x_{n+1} - x_i)} & \text{for x} \in [a..b] \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

If the basis functions $b_1, .., b_n$ integrate polynomials of degree $n-1$ exactly, the new basis functions $b_1, .., b_{n+1}$ do the same for degree n, for any arbitrary value x_{n+1}, as long as it differs from all the previous values of x_i. However, x_{n+1} can be chosen, such that polynomials of degree $n+1$ can be integrated.
The position x_{n+1} of this new point can be obtained from a Gauss type formula with preassigned abscissas [3]. The original formula can be used to add any arbitrary number of new points. With only a number of one new point, the formula can be written as:

Theorem: A set of basis function $b_1, .., b_{n+1}$ can integrate polynomials of degree $n+1$ exactly, if:

(a) for all polynomials p(x) of degree ≤ n:

$$\int_a^b \left(\sum_{i=1}^{n+1} c_i b_i(x) \right) dx = \int_a^b p(x) dx \tag{4.3}$$

(b)

$$\int_a^b (x - x_1)...(x - x_n)(x - x_{n+1}) dx = 0 \tag{4.4}$$

Solved by x_{n+1} this leads to

$$x_{n+1} = \frac{\displaystyle\int_a^b x \cdot (x - x_1) \cdots (x - x_n) dx}{\displaystyle\int_a^b (x - x_1) \cdots (x - x_n) dx} \tag{4.5}$$

Using this value in (4.2), the resulting basis functions can integrate polynomials of higher degree. This new set of basis functions will be called **piecewise Gauss** polynomials. This is a simple but yet powerful enhancement to the old piecewise polynomials.

4.2 Piecewise Gauss polynomials

The full set of piecewise Gauss basis functions can be constructed as discussed in section 2.2, but with the piecewise Gauss instead of centered nodes. Starting with the constant function on $[-1, 1]$, the successive son functions can be found by splitting the interval at the function's node (formula 4.5) and constructing a new functions on the two intervals (formula 4.2). The resulting basis functions are similar to the polynomial basis shown in section 2.2.2, but have their nodes spread to the sides. Figure 4.2 compares the old and the new basis functions.

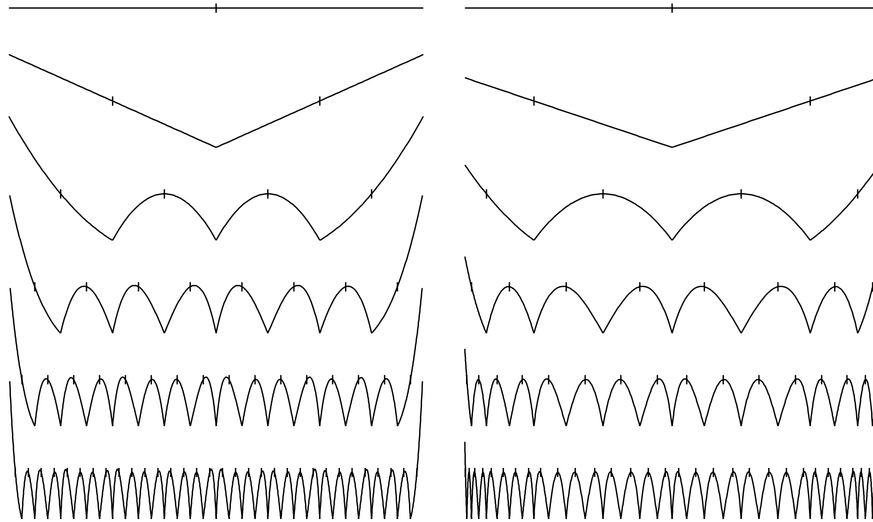


Figure 4.2: Basis functions constructed by the old rule (left) and piecewise Gauss rule (right).

Figure 4.3 shows a two dimensional sparse grid of level five. The piecewise Gauss rule spreads the points to the sides without affecting the original sparse grid structure.

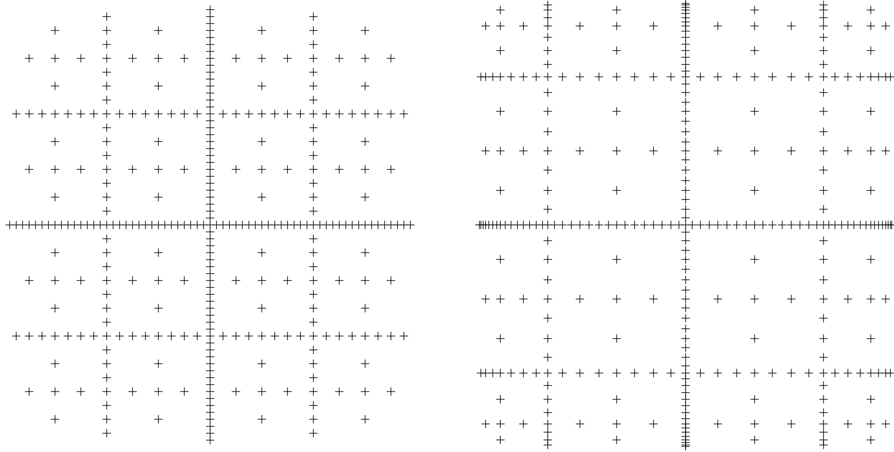


Figure 4.3: 2-dimensional grid generated by the old rule (left) and piecewise Gauss (right).

4.3 Polynomial degree of exactness

Piecewise Gauss polynomials of level l have degree $l-1$ and integrate polynomials of degree l exactly. If l is odd, polynomials of degree $l+1$ can also be integrated. Due to the symmetry of the generated nodes, odd functions are always correctly integrated to zero. On level l there are $2^l - 1$ points. A classical non adaptive gauss scheme using this amount of nodes can integrate polynomials of degree $2 \cdot 2^l - 3$ exactly. This is the trade off that has to be accepted for a function basis that allows adaptivity.
Figure 4.4 compares the degree of polynomials which can be integrated exactly using different sets of basis functions. Piecewise polynomials in average increase their degree by one per level. Determining the function's node according to the Gauss formula (4.5) increases the polynomial degree by one, compared to nodes lying at any other position. With the Gauss rule, the polynomial degree of exactness is about twice the number of points.

	level	1	2	3	4	5	6
	points	1	3	7	15	31	63
Polynomial degree of exactness	piecewise polynom.	1	1	3	3	5	5
	piecewise Gauss	1	3	3	5	5	7
	Gauss	1	5	13	29	61	125

Figure 4.4: polynomial exactness of different basis functions

Although the polynomial degree of exactness was the main design goal of these new basis functions, it is not necessarily the most crucial point. Very smooth

functions or polynomials themselves profit most from a higher degree of exactness. Functions with singularities or discontinuous functions do generally not show any performance gains with different basis functions. Here the adaptive strategy is the most decisive point. The piecewise Gauss polynomials are superior to the old polynomials, as they can integrate higher order polynomials without constraining the adaptive strategy.

4.4 Integration error

The piecewise Gaussian basis can integrate polynomials of degree $n+1$ exactly using only polynomials of degree n. It is important to further investigate how functions can be integrated without interpolating them exactly. This will affect the adaptive strategy, discussed in the next section.
When a function is integrated with polynomial basis functions, the integrand is effectively interpolated by a polynomial, which is then integrated. In the piecewise Gauss basis the interpolation points are the nodes x_i. The quadrature error can be computed according to this theorem [2]:

Theorem: If a polynomial $p(x)$ of degree n interpolates a function $f(x)$ at the abscissas $x_1,..,x_{n+1}$, then there is a ξ for which the interpolation error is

$$|p(x) - f(x)| = \frac{1}{(n+1)!} \cdot D^{n+1} f(\xi) \cdot \prod_{i=1}^{n+1} (x_i - x). \tag{4.6}$$

When f(x) is a polynomial of degree $n+1$, then $D^{n+1}f(\xi)$ is constant and does therefore not depend on ξ. Integrating the error on a domain $[a..b]$ leads to

$$\int_a^b |p(x) - f(x)| dx = \frac{1}{(n+1)!} \cdot D^{n+1} u(\xi) \int_a^b \prod_{i=1}^{n+1} (x_i - x) dx. \tag{4.7}$$

The integral of the product is equal to zero, according to formula (4.4). Hence the error is zero:

$$\int |p(x) - f(x)| dx = 0 \tag{4.8}$$

Figure 4.5 shows three different basis functions generated by the piecewise Gauss rule, on the levels one, two and three. When basis functions up to these levels are used to interpolate a function, there occurs an interpolation error. The right column shows this error up to a constant factor depending on $D^{n+1}u(\xi)$. The error is zero at the interpolation points and integrates to zero on the whole domain. The Gaussian nodes are chosen such that the positive and the negative error is balanced, i.e. the area above and below the x-axis is the same. This is a crucial point and it will be shown that a straight forward adaptive strategy destroys this property.

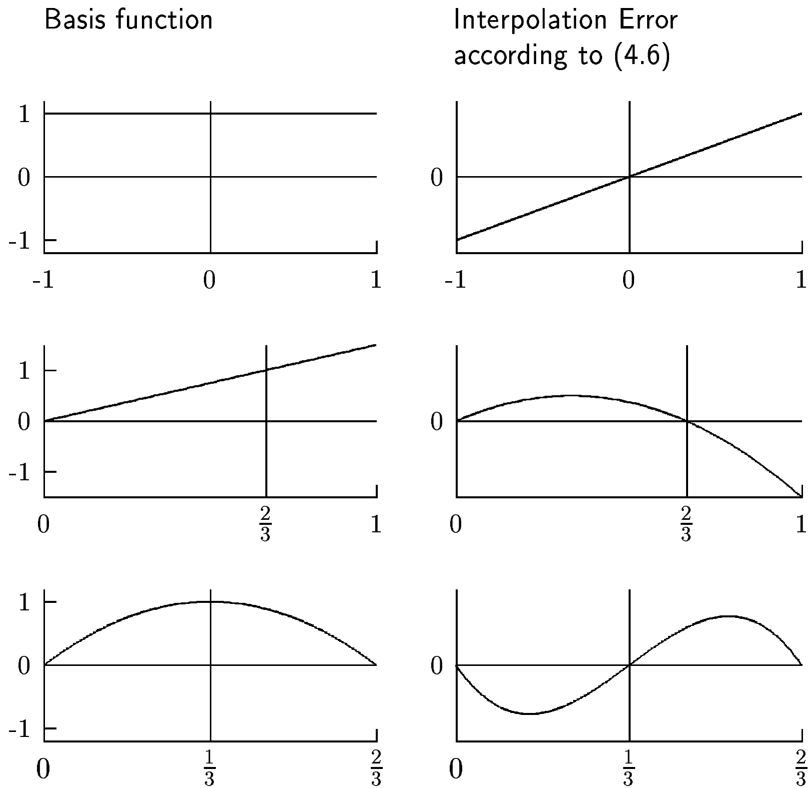


Figure 4.5: Basis functions and the error resulting from an interpolation of polynomials with higher degree.

4.5 Non adaptive numerical example

So far, different univariate quadrature rules and the non adaptive sparse grid construction have been introduced. In this section three quadrature rules are compared in a non adaptive computation. The Gauss-Patterson rule and the two hierarchical rules based on the old and the Gauss based piecewise polynomials are used to integrate the following five dimensional integral:

$$\left(1+\frac{1}{5}\right)^5 \cdot \int_{[0..1]^5} \prod x_i^{\frac{1}{5}}\, dx. \tag{4.9}$$

The plot 4.6 shows the numerical results. The exact value of this integral is 1 and the error is plotted on the y-axis. The x-axis represents the number of function evaluations. This example not representative, but gives an idea of the importance of the polynomial degree of exactness, which is the main difference between the quadrature rules. This relatively smooth integrand is still integrated well with piecewise Gauss polynomials. For even more smooth functions or polynomials themselves the difference between these curves would obviously be much greater. They can be nearly the same for extremely non smooth or discontinuous functions. A piecewise basis can often outweigh its disadvantage by the implementation of adaptivity, which is not taken into account in this plot. However, closer investigation in section 7.1 will show that the adaptive

algorithm introduced in the next section will not improve the performance considerably beyond the non adaptive results. This leaves the piecewise Gauss rule behind the Gauss-Patterson rule in this smooth example.

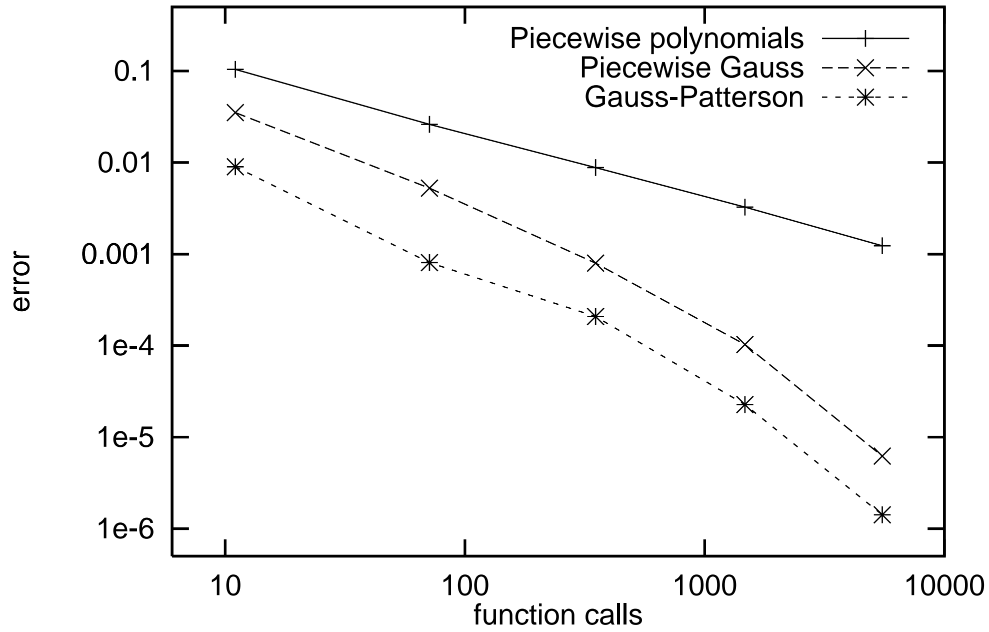


Figure 4.6: Nonadaptive numerical example

5 Adaptivity

Adaptive algorithms spend more effort on rough and less effort on smooth regions. The two crucial tasks of adaptive strategies are the detection of a function's roughness and an appropriate grid refinement. There are formulas which use a priori knowledge of the integrand's smoothness [4]. If this kind of information is not available the adaptive algorithm has to retrieve it during the integration process. For sparse grids several strategies have been published [2, 1]. After rough regions have been detected the grid has to be refined. Sparse grids do not allow arbitrary points to be inserted. Gauss or Gauss-Patterson based grids can only be refined along a whole direction at once. A hierarchical basis allows local subdivision of the function's support according to a well defined formula.

The adaptive scheme discussed in this section does not introduce radically new concepts. However, it was necessary to adjust the known algorithm in order to preserve the superiority of the new basis. The piecewise Gauss basis has been constructed with a regular grid in mind. A different grid construction voids the preliminaries of the gauss formula. A slightly modified construction process will resolve this problem as explained in section 5.4.

5.1 Construction

Each node contributes to the total integration result by a certain amount. This amount is determined by the hierarchical surplus and volume under the according basis function. The surplus measures the function's roughness, since it is the difference between the integrand and a smooth polynomial interpolation. Every node's volume contribution indicates how promising a grid refinement is at this point. A good adaptive algorithm based on this indication would be to find the node with the greatest contribution and then locally refine the grid at its position. After computing the surpluses of the new nodes the algorithm can choose the next node. The big disadvantage is that for the surplus calculation either all father nodes have to be revisited or huge amounts of extra information have to be stored with every node. This leads to a high time or memory complexity. It is not feasible to only refine the grid at only one point in each step. It is possible to refine the grid at more positions and insert many new points at once. Calculating all new surpluses can be done by one single tree traversal with a complexity depending on the total number of nodes in the tree and the number of dimensions.

An algorithm which combines several refinement steps can enqueue all the nodes of an initial grid in a priority queue with their volume contributions as their key. In each step the few largest nodes are then dequeued and the grid is refined at

their position. The newly generated nodes are then enqueued after their surplus is calculated. A good choice for the number of nodes taken from the queue was found to be about 10% of the queue's length in case of an up to 10-dimensional problem, 5% and 3% for more than 10 or more than 100 dimensions. However the numerical results don't vary a lot with this value. The new nodes have a smaller support and surpluses are usually smaller. They are therefore most likely queued at the rear of the queue and the same nodes are taken out of the queue no matter if less or a few more nodes are taken at once. Taking too many can jeopardize optimal adaptivity. Dequeuing 100% at once leads to a completely non-adaptive regular grid. Taking too few leads to unnecessary increase of computation time.
The grid can be easily refined at a certain position in a way that is already suggested by the tensor product basis. Each one-dimensional hierarchical basis function has two sons. The support of the son functions is created by dividing the father function's support. In a multi-dimensional tensor product basis function the support is split along each dimension, such that $2d$ new basis functions are created. Figure 5.1 shows a two-dimensional unrefined grid with one node and the new nodes and supports after the refinement.

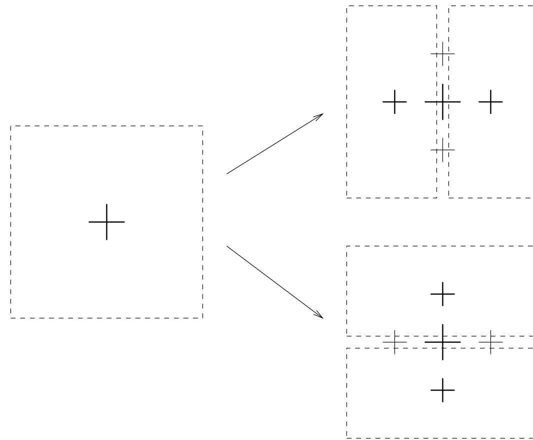


Figure 5.1: Nodes and supports of a refined grid

The algorithm has to stop when a desired accuracy is reached. A common criterion is the difference between the volumes of the current and the previously processed grid. This difference however depends on percentage of nodes being dequeued before the next grid is generated. In order to produce comparable data the sum of all volume contributions in the queue was used as an error estimate. Once this sum falls below a fixed epsilon a further refinement of all nodes is considered to change the integral by less than epsilon.

5.2 A good example

The superiority of an adaptive over a non-adaptive strategy depends on the nature of the integrated function. Adaptive schemes are especially suited if the function's rough parts are concentrated on a confined area of the domain. The

following three-dimensional integral is infinite on some of the domain's borders. Surpluses of points close to these borders will have surpluses tending to infinity and will therefore contribute much more to the total volume than other points.

$$\frac{1}{4^3} \int_{[0..1]^3} \prod_{i=1}^{3} \frac{1}{\sqrt{x_i}} \, dx \tag{5.1}$$

The integral was computed with the previously discussed adaptive scheme. The domain of the unit cube $[0..1]^3$ was transformed to the cube $[-1..1]^3$, such that the piecewise Gauss basis functions can be used.
The numerical result shows how the adaptivity improves the convergence process. The superiority of the adaptive grid grows with the number of points that have been evaluated. The reason is that the potential of adaptivity can only be developed after the function's nature can be derived from a number of evaluated points. The more points there are the better is the knowledge about the functions behavior, resulting in a curve with increasing downward slope.
A third curve was plotted to demonstrate the theoretical power of this refinement criterion. The volume contributions of all nodes were sorted according to their absolute value. The greatest values were then added to calculate an integral with a given amount of points. Obviously this convergence rate can not be reached in practice, since it uses a priory knowledge about the integrand. It rather shows the best possible results that can be achieved by searching for the greatest volume contribution. The next section shows these best possible results are not always that good.

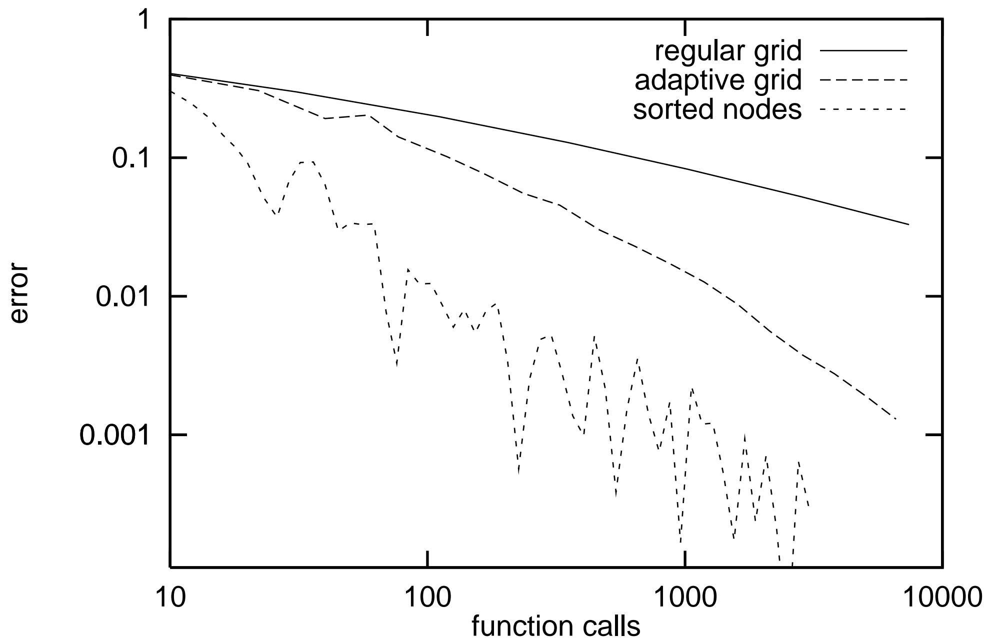


Figure 5.2: Adaptive quadrature of the "good example"

Figure 5.3 shows the points lying in the layer spanned by the x_1 and x_2 axis and $x_3 = 0$. The whole grid contains 922 points in all three dimensions and

258 points in the two dimensional subspace. Since the function is symmetric the grid is the same in the other directions. The points lie more densely close to the two borders where the integrand is infinite.

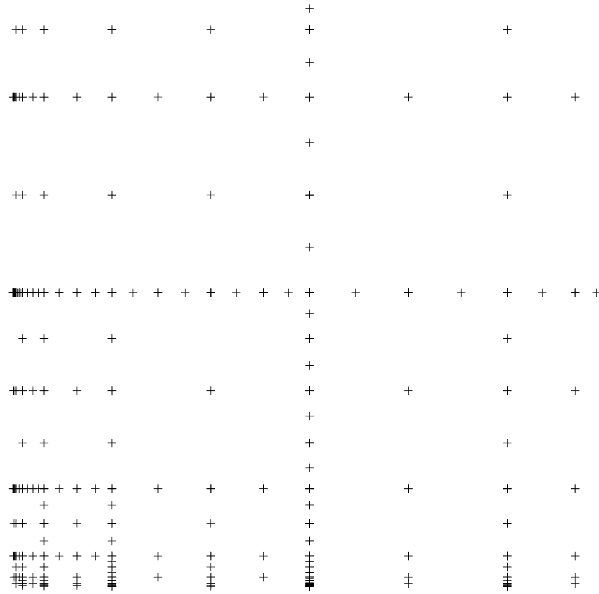


Figure 5.3: Adaptive grid

5.3 A bad example

An adaptive algorithm can obviously not considerably improve the integration of a very smooth function, because the resulting grids are always close to regular ones. Since many functions are smooth in some regions or even in whole directions it is absolutely essential that adaptivity does not lead to considerably worse results than a regular grid. With the adaptive refinement scheme established so far this can be the case, as the following example shows.

$$\frac{1}{(e-1)^5} \int_{[0..1]^5} \prod_{i=1}^{5} e^{x_i}\, dx \tag{5.2}$$

This is an exponential function integrated in five dimensions. The plot below shows again three curves representing the regular grid, the adaptive grid and the sum of the sorted nodes. The first remarkable thing is that adaptivity worsens the results by up to four digits. The second thing is that on most parts the sorted nodes performs even worse. This indicates some serious flaws in the choice of new points. The nodes with the greatest volume contributions do not always contribute most to the integrals accuracy.
In fact, there are only very few points responsible for this result. Only a small amount of nodes being refined or few points being deleted in regular grid already lead to this big decrease of accuracy. It is possible to determine these points and reconstruct the grid in an appropriate manner.

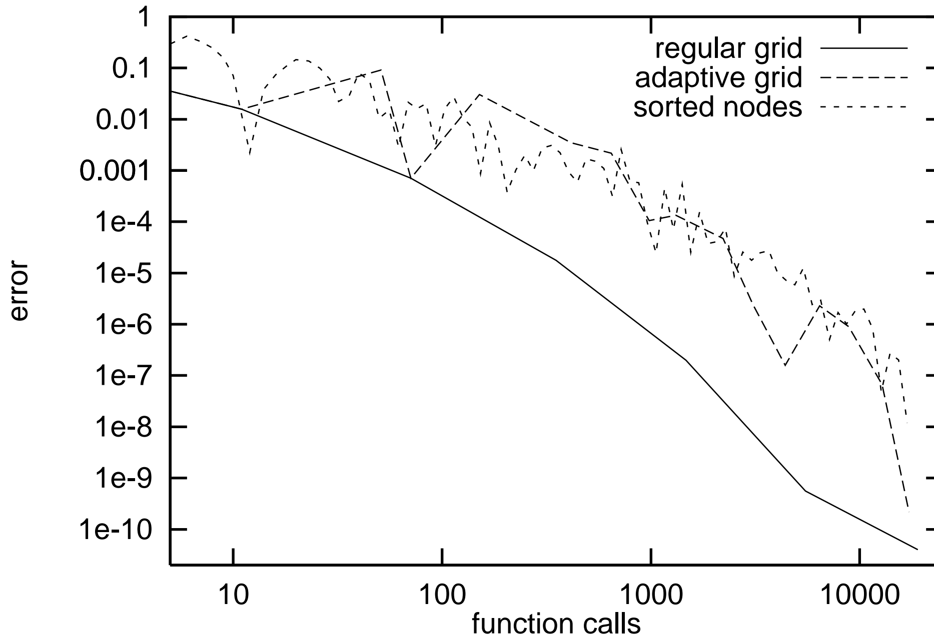


Figure 5.4: Adaptive integration of the "bad example"

5.4 Balanced adaptivity

As explained in section 4.4 the piecewise Gauss basis functions reach a higher polynomial degree of exactness by balancing the integration error on the function's domain. A basis function of polynomial degree n can integrate polynomials of degree $n+1$ exactly, because the integration error on the left and on the right side of the node sum up to zero. When both, the function left and right son are created, the volume contributions of both nodes sum up to zero. Creating only one son, however, leads to a new volume contribution, which is not balanced by another volume of the same size, but opposite sign.
Without taking any special care this situation occurs with nearly every refinement. Figure 5.5 shows how an unbalanced grid emerges after refining a five node grid at one point. The missing neighbor is the other son of the nodes father. It is the one that should balance the error.

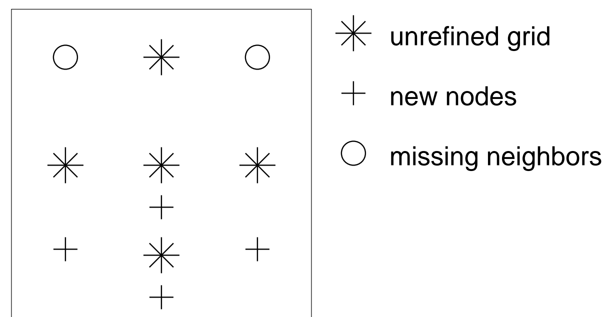
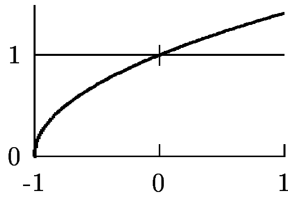
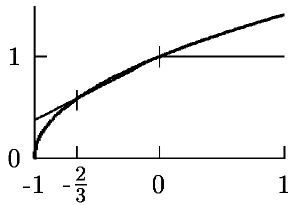


Figure 5.5: Unbalanced grid

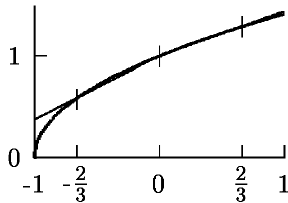
With a lower polynomial degree of exactness smooth functions can generally only be integrated less accurately. The following table demonstrates in three steps, how unbalanced refinement destroys integration accuracy for the function $\sqrt{x+1}$.



With one point in the middle of the domain the integration error on the left and the right side add to 0.11.



A refinement on the left reduces the error on the left side, while the right error lost its counterpart. The total error increases to 0.20.



A third point on the right corrects this situation. The error on the left as well as the error on the right are low. The total error is down to 0.016.

There are basically two possible methods to preserve the higher polynomial degree of exactness. One is to insert the missing neighbors and the other is not to insert the unbalanced node. Obviously only the first one allows the creation of new nodes. The second method is applied automatically, when fewer nodes are necessary to achieve the same accuracy. The previously discussed adaptive algorithm can therefore be enhanced by one new step, which balances all the newly created nodes. Figure 5.6 shows the nodes and their hierarchial position in an unbalanced and a balanced one-dimensional grid.

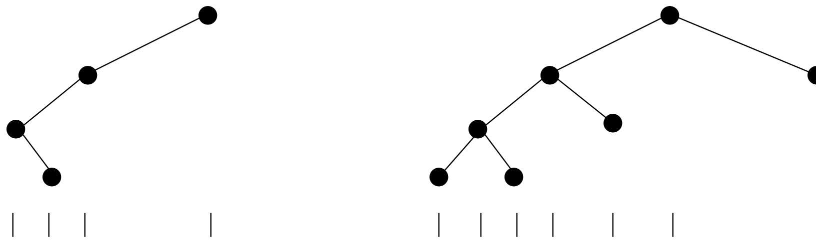


Figure 5.6: Balancing algorithm in one dimension

In multi-dimensional grids a node has to be balanced in every direction, and these new neighbors have to be balanced again. With every new direction the number of balancing nodes is doubled, unless the nodes lie in the center of that direction. All nodes lie on the 2^{dim} vertices of a hypercube with the original node in one corner. The surplus of the nodes depend on their father

nodes these, if not already existing and balanced, have to be balanced too. The number of new nodes can increase by a factor of up to 2^{dim}. This catastrophic number of new nodes will hardly be reached in practice for two reasons. First the balancing nodes are close to the original node and are therefore within a region which was chosen for refinement. Even without balancing, they would most likely be created anyway. Second, most of the nodes in a sparse grid have positions with many of the coordinates equal to zero, and have no neighbors in the corresponding directions. With n coordinates being different from zero there will be at most 2^n balancing nodes. This is the confining factor especially in extremely high dimensional grids, where only low levels can be reached.
Although the number of additional points is usually low, there are occasions where they do lead to unnecessary function evaluations. Optimally this balancing algorithm should only be applied in smooth regions. Section 5.7 describes an algorithm which adaptively applies balanced adaptivity.

5.5 A two dimensional example

The balancing algorithm can be visualized, when it is used to refine a two dimensional grid. The polynomial order of exactness is reduced by unbalanced refinement and is restored with the introduction of balancing points. The polynomial $p(x, y)$ has third degree in x and y. It can be integrated exactly by a sparse grid of second level.

$$p(x, y) = x^2 + (y - 0.2)^2 + x^2 y \tag{5.3}$$

A sparse grid of level two has a total of five points, which resulted from a refinement of a single point grid. The table below shows the volume contribution of each point in a domain ranging from -1 to 1 in each direction. The center point for instance has a surplus of p(0,0)=0.04. The volume of this constant basis function is $2 \times 2 = 4$ and the total volume contribution is 0.16. The sum of all five points is the exact integral value of $212/75 \approx 2.84$.
It is also worth noting that on this coarse grid volume contributions of neighboring nodes do not have opposite sign. The function has a minimum near the center. Therefore surpluses are positive in all directions. Balancing individual points in this grid would not be necessary. This grid is already balanced due to construction of the refinement.

	0.27	
0.67	**0.16**	**0.67**
	1.07	

The polynomial can be integrated exactly, but the basis functions are just strait line pieces, which balance the integration error. That is why further refinement

produces non-zero surpluses. The error estimate, the sum of the four points on the second level, produces an error limit of 2.68. Refining the grid at the largest node destroys this balance. The real error rises to $2 \times 0.17 = 0.34$, while the estimated error goes down to 1.6.

	0.27	
0.67	**0.16**	**0.67**
	–0.01	
+0.17	**1.07**	+0.17
	+0.01	

Once the new points are balanced in each direction the error is zero again. The only reason why the balanced points add exactly to zero is that the integrand is a polynomial with a degree one higher than the grid level. Other functions would only reduce their error if they are approximated better by a polynomial of higher degree.

–0.17	**0.27**	–0.17
0.67	**0.16**	**0.67**
	–0.01	
+0.17	**1.07**	+0.17
	+0.01	

The sparse grid on third level shows how neighboring surpluses add to zero and only the initial five points determine the integral value. The estimated error is now down to zero. It would have been reasonable not to refine any of the initial five points. There was however no way the decide this only on the basis of five evaluated points.

			+0.01			
	–0.17		**0.27**		–0.17	
			–0.01			
+0.01	**0.67**	–0.01	**0.16**	–0.01	**0.67**	+0.01
			–0.01			
	+0.17		**1.07**		+0.17	
			+0.01			

5.6 Numerical results

With a balanced grid it should now be possible to integrate a smooth function with the same accuracy as a regular grid, without a considerable increase of function evaluations. This property is verified using the two integrands seen in section 5.2 and 5.3. First, it will be demonstrated how convergence can be improved by the balanced adaptivity. The exponential function (5.2) from section 5.3 will be integrated on a balanced grid:

$$\frac{1}{(e-1)^5} \int_{[0..1]^5} \prod_{i=1}^{5} e^{x_i}\, dx.$$

The numerical results in figure 5.7 show that the balanced grid performs nearly as good as the regular grid. The error is worse by one digit at most instead of up to four digits. Since better performance than a regular grid could not be expected anyway, this result is almost optimal.
Figure 5.8 compares the balanced and the unbalanced grid with about 10000 points. Only the points lying in the layer spanned by the x_1 and x_2-axis are plotted. The unbalanced grid has 10183 points and computes the integral with an error of 6.3×10^{-8}. The balanced grid contains 10303 with an error of 2.7×10^{-10}. Both grids are plotted over each other, marking the points with a upright and a rotated cross respectively. Points which are present in both grids appear as a star. Although the positions of the points differ only slightly, the accuracy resulting from both grids differs by a factor more than 20. In the plotted subspace the unbalanced grid has eight unbalanced points. Four are balanced with additional points and four were not created, such that the total number of points in the grid remains constant.

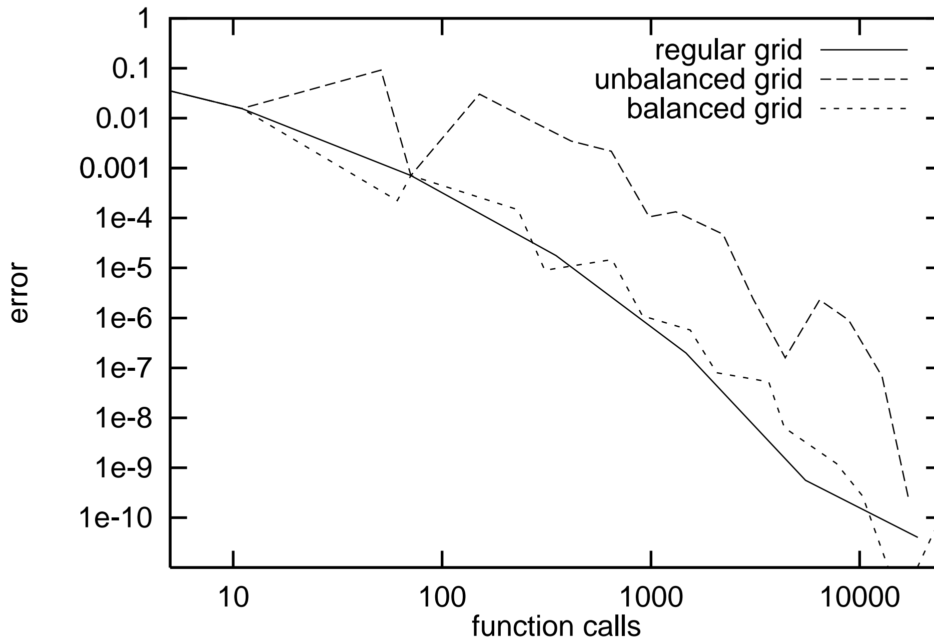


Figure 5.7: Balanced grid in the “bad example”

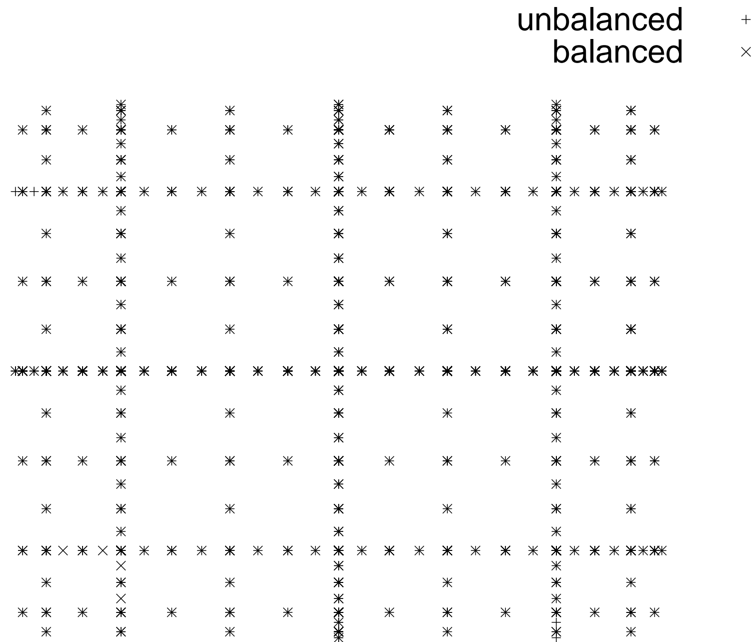


Figure 5.8: Balanced vs. unbalanced grid

The plot below shows how balancing increases the number of function evaluations. Whereas the ratio of error and points does not differ considerably, the number of points on a certain level is different. A cross marks the first creation of a point on level 10. Since every point is balanced in a balanced grid, it takes much more function calls to create points on a new layer.

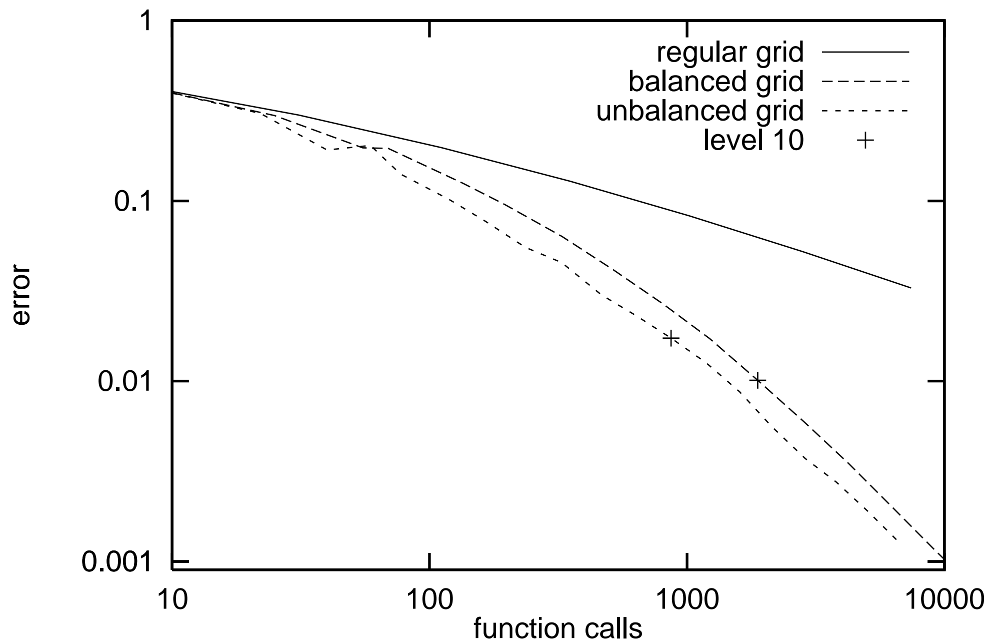


Figure 5.9: Computational results for the balanced grid in the "good example"

On level ten there are about twice as many points in the balanced than in the unbalanced grid. The error in the balanced grid is only lower by one third. Although the performance decrease is bearable in this example, it deserves some further attention in order to assure optimal convergence for all functions.

5.7 Adaptive balance

Balanced adaptivity only improves convergence where the integrand is smooth. Although the number of nodes evaluated in vain is generally not very high, performance can be increased when the functions smoothness is observed and balance only takes place in smooth regions and in smooth directions.
It can be verified, whether balanced nodes have improved accuracy on previous levels. If a point x is dequeued, it can be checked whether the sons should be balanced in direction i. Let y_i be the neighbor of x in direction i, as shown in figure 5.10.

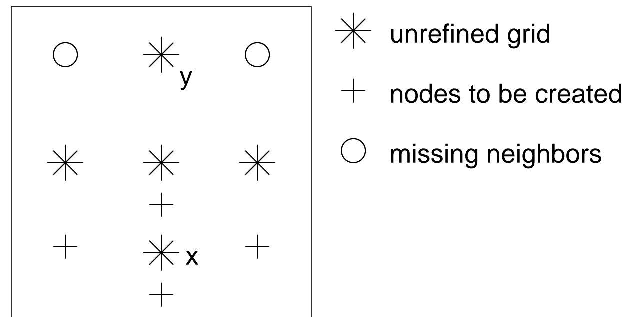


Figure 5.10: Unbalanced grid

The volume contributions of x and y_i are v and w_i. If y_i has not yet been created w_i is set to zero. In direction i a single node is considered to deteriorate the result, if the error estimate for the unrefined point, $|v + w_i|$, is better than the estimated error, $|v|$ and $|w_i|$, of an unbalanced refinement.

$$|v + w_i| < min\{|v|, |w_i|\} \tag{5.4}$$

Based on this result a new node is balanced in the appropriate directions. A further criterion for balancing would be the total amount of new points. Refinement could be reconsidered whenever balancing costs grow too high. This would however complicate the whole refinement strategy.
The result of this algorithm can be demonstrated when the smooth exponential function (5.2) is multiplied with the non-smooth root function (5.1).

$$\frac{1}{4(e-1)^9} \int_{[0..1]^{10}} \frac{1}{\sqrt{x_1}} \left(\prod_{i=2}^{10} e^{x_i} \right) dx \tag{5.5}$$

In the nine smooth directions balancing improves accuracy, whereas balancing in direction of x_1 does not. The adaptive algorithm can detect this and only balance in nine directions.

The numerical results show that adaptive balance is superior to a fully balanced as well as an unbalanced strategy. Balanced adaptivity can not always be expected to produce the optimal convergence rates. For example the polynomial in section 5.5 would not have been balanced automatically on level two. By and large, the results will mostly lie close to the better of the balanced and the unbalanced strategy.

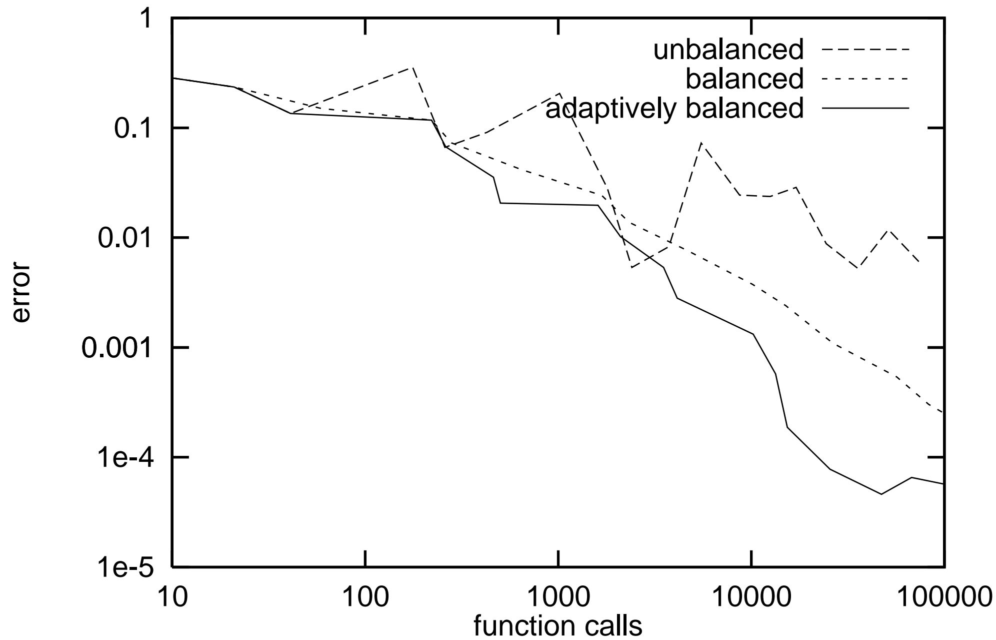


Figure 5.11: Results for the adaptively balanced grid

6 Efficient data structure

Adaptive sparse grids can be used to accurately integrate complex functions with a minimum of function evaluations. However maintaining and processing such a grid is quite costly compared to non adaptive schemes. Whereas mathematical operations to calculate surpluses and volumes are rather simple, accessing the data in the grid requires most of the computation time. To increase the performance of sparse grid algorithms it is therefore most important to find efficient data structures.
There have been various approaches to store sparse grids. A common data structure is a binary tree. Simple trees have an access complexity of $logn$. In order to increase efficiency additional information is needed to administrate the structure. This increases memory consumption and maintainance complexity. Another data structure for a sparse grid is a hash table. It allows complex and fast access required by sparse grid algorithms. Suitable hash functions and table implementations have been discussed [5]. The improvement addressed in this section is a more economical way to store the unique key, which describes the exact position in the grid.

6.1 Integer vector

A node in a multi-dimensional space can be represented by a vector containing all the coordinates. An economical way to store each coordinate is to encode the hierarchical position as an integer. The top node starts with number one. The lefts son of a node x gets the number $2x$ and the right son $2x+1$. Figure 6.1 shows the hierarchical tree.

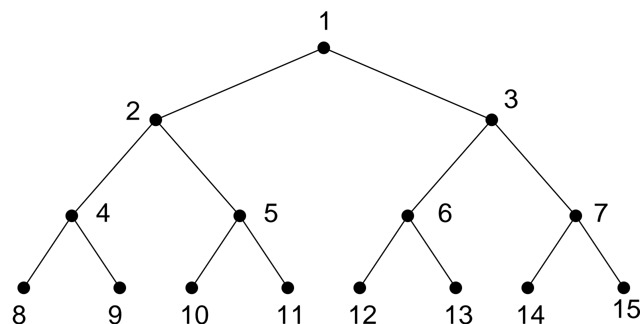


Figure 6.1: Hierarchical numbering scheme

Hierarchical position are often described with level l and index i. The corresponding integer can then easily be calculated as $2^{l-1} + i - 1$.
The top node, one, can be stored in one bit. In every new level the number of bits required grows by one. It takes at most l bits to store a coordinate of

level l. Without complex packing strategies, l bits have to be allocated for each coordinate. The total memory requirement of a d-dimensional integer vector storing multi-dimensional coordinates up to level l is:

$$dl\ Bits \tag{6.1}$$

Although accessing individual coordinates in this vector is very simple, memory consumption makes it an unsuitable hash key. Its size easily outgrows the size of the numerical data. With every hash table access the same amount of memory has to be read in order to check whether the right entry was found. This affects the execution speed, since main memory access is one of the most time consuming operations on modern computer systems.

6.2 The bit string

In a sparse grid of level l the basis functions can have up to level l in each direction. It is not necessary to allocate the required memory for each coordinate, because level l can not be reached in all dimensions at once. The sum of all one dimensional levels is $l+d-1$. Therefore a bit string can store a condensed version of each coordinate.

Each coordinate is encoded in the following way. The top node is encoded by an empty string. A node's left son is encoded by appending the bits 10 to the string. For the right son the bits 11 are appended. The coordinate strings are then joined together with a separating zero between them.

A coordinate of level l_i is stored in $2l_i-2$ bits. Summing up all the coordinates in a multidimensional position of a node of level l gives a total of $2l-2$ bits. The coordinates are separated by $d-1$ zero bits. The total storage requirement is:

$$d+2\,l-3\ Bits \tag{6.2}$$

Figure 6.2 shows how coordinates are stored in the bit structure. The integer representation of the coordinates is (2,6,1,4,7,...).

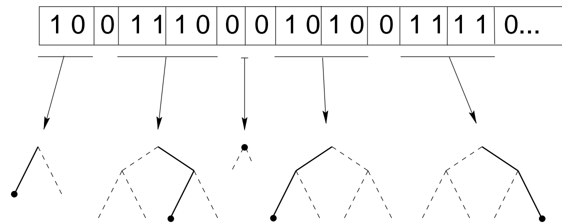


Figure 6.2: Bit string encoding a multidimensional position vector

There is another possible interpretation of the bit string. The bits representing the coordinate can be created by interlacing the bits of integer discussed in section 6.1 with one. Without the leading zeros, a hierarchical position is represented by the bits $1b_1b_2\cdots b_n$. These bits are stored as $1b_11b_2\cdots 1b_n$ in the bit string. In other words a one marks the next bit to contain a bit of the position and a zero separates the coordinates.

In terms of required bits this structure is not optimal, but memory was not the only design goal. Complicated bit arithmetic can outweigh the advantage of reduced memory access. A compromise between time and space has to be made.

6.3 Speed improvements

A typical task done by a sparse grid algorithm is a full or a partial grid traversal. A full traversal is necessary to compute the surpluses and volume contributions of all the nodes. Partial traversals are necessary when the grid is refined and to check a nodes integrity, i.e. the presence of all the father nodes its surplus depends on. In fact most hash table accesses are part of traversals in which nodes are accessed in a well defined order. It is possible to improve this process if the properties of the traversal algorithms are taken into account.
In a sparse grid traversal there is a variable maintaining the current position in the grid. When the traversal proceeds, a new position is written into this variable. This variable is then used to find the corresponding entry in the hash table. The optimal structure to store the current position is an integer vector. It has to be changed often and memory requirement is irrelevant, since it is only stored once in the memory. The ideal structure to store the key in the hash table is a bit string, because it does not need to be changed and is stored many times. It would be possible to convert the integer vector into the bit string every time the hash table is accessed. Since most of the time the coordinates of two consecutive positions only differ slightly, better performance can be achieved, when the converted bit string is maintained parallel to the integer vector.
The bit string can be accessed and updated quickly, if it is enhanced by two additional components. One is an integer vector containing the integer representation of each coordinate. The second new component is an offset pointer, which points to all zeros representing a new dimension.

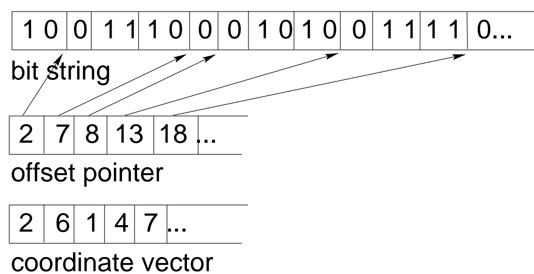


Figure 6.3: Improved bit string for faster access

A coordinate can be read directly from the position vector. Whenever a coordinate is changed, the bit string has to be updated. If the old and the new representation do not have the same length, the following bits have to shifted to make enough space or fill the gap and the offset vector has to be updated.
Despite these new components main memory access stays low, since on modern processors this enhanced bit string structure is small enough to reside in the

processor cache, where all the costly updating operations occur. Figure 6.4 illustrates the memory access using the enhanced bit string.

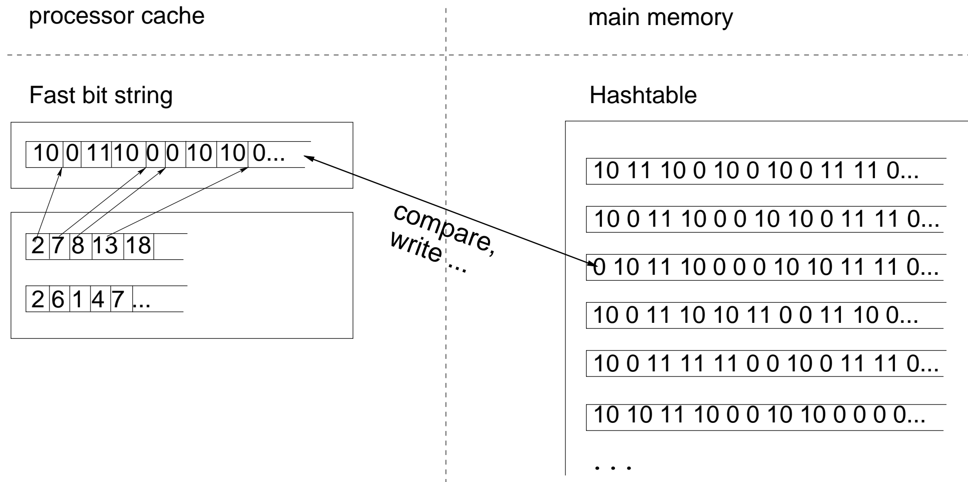


Figure 6.4: Efficient hash table access using bit strings

Hierarchical operations

When traversing the sparse grid data structure the most common operations are finding a left or right son, finding the father and finding the neighbor, i.e. the father's other son.
For the neighbor operation only one bit has to be changed to convert the last 10 into 11 or vice versa. Using the offset vector the bit can be accessed directly and the complexity is $O(1)$.
For a son or father operation two bits have to be inserted or deleted. The remaining bits have to be shifted. The total number of shifted bits is at most $d + 2l - 3$. However a modern computer processor can shift up 64 bits at once, which makes this extremely fast. The offset pointer is updated by adding or subtracting two in up to d components. The total complexity is $O(d + 2l)$.
Figure 6.5 shows how finding the right son in the second dimension affects the data structure.

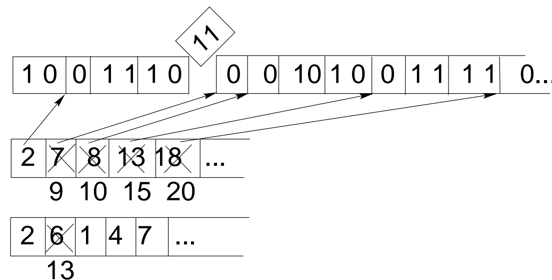


Figure 6.5: Son operation using a fast bit string

7 Numerical Results

In the following the strengths and weaknesses of adaptive sparse grids are evaluated. Several integration problems have been collected by Morokoff, Caflisch and Owen [6, 7]. These examples have been integrated with quasi-Monte Carlo methods, using different quasi-random number generators. Gerstner and Griebel [4] cited some of these integrals, where Gauss-Patterson quadrature outperforms the quasi-Monte Carlo methods on these or, at least, on slightly modified versions of the integrands. All their results were obtained without adaptive strategies, except for one example, where the regular construction was altered to focus on the important directions. Sparse grids based on the Gauss-Patterson rule also performed better than all the other non-adaptive rules investigated and is therefore the only one the piecewise Gauss rule is compared with. The quasi-Monte Carlo methods were based on quasi-random numbers generated after Faure, Sobol and Halton. The results of all the three are given, if they deviate considerably, or if not stated otherwise. Monte Carlo and quasi-Monte Carlo methods typically show convergence rates proportional to cN^{α}. Whereas α is called the fitted convergence rate. It is obtained by a least square fit of the average errors of several runs. In a log-log-plot these error plots appear as straight lines with slope α. Only these regression lines are taken from the original sources, since the error points vary with the random numbers generated. Sparse grid algorithms often show convergence rates that increase with time. The coefficient α then grows with the total number of investigated function evaluations.

7.1 Test Function

The first test function has already been introduced with five dimensions in section 4.5. This rather academic function is now integrated with an adaptive strategy in different dimensions and compared with the quasi-Monte Carlo and the Gauss-Patterson method, where data was available.

$$\left(1+\frac{1}{d}\right)^d \int_{[0..1]^d} \prod x_i{}^{\frac{1}{d}}\, dx \tag{7.1}$$

Since this integral is quite smooth, adaptivity does not increase accuracy. Figure 7.1 shows that in five dimensions the piecewise Gauss rule performs somewhere between quasi-Monte Carlo and Gauss-Patterson. The regression line of quasi-Monte Carlo the methods is plotted for more than 1000 points, where it is backed by numerical data.
The same function can also be integrated with a higher number of dimensions. Figure 7.2 shows the results of adaptive sparse grid quadrature in different

dimensions and the quasi-Monte Carlo method which showed next to no performance change with different numbers of dimensions. This plot demonstrates the negative effects of dimensionality on the accuracy of sparse grids. Unfortunately no higher dimensional data was published for the Gauss-Patterson based grid. Since decreasing performance in higher dimensions is an inherent property of sparse grids, Gauss-Patterson quadrature would eventually also fall behind quasi-Monte Carlo.

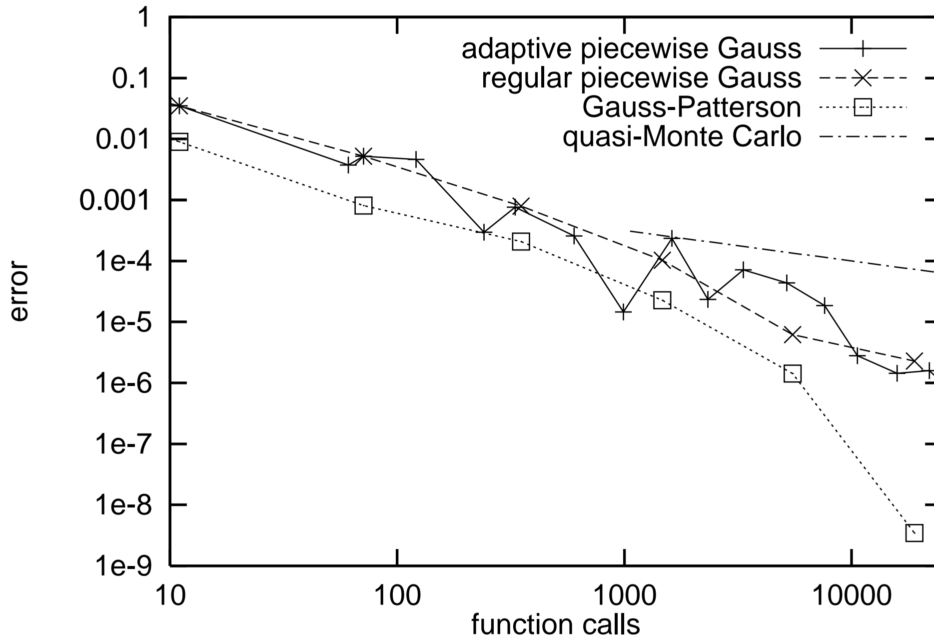


Figure 7.1: Computational results for the test function in 5 dimensions

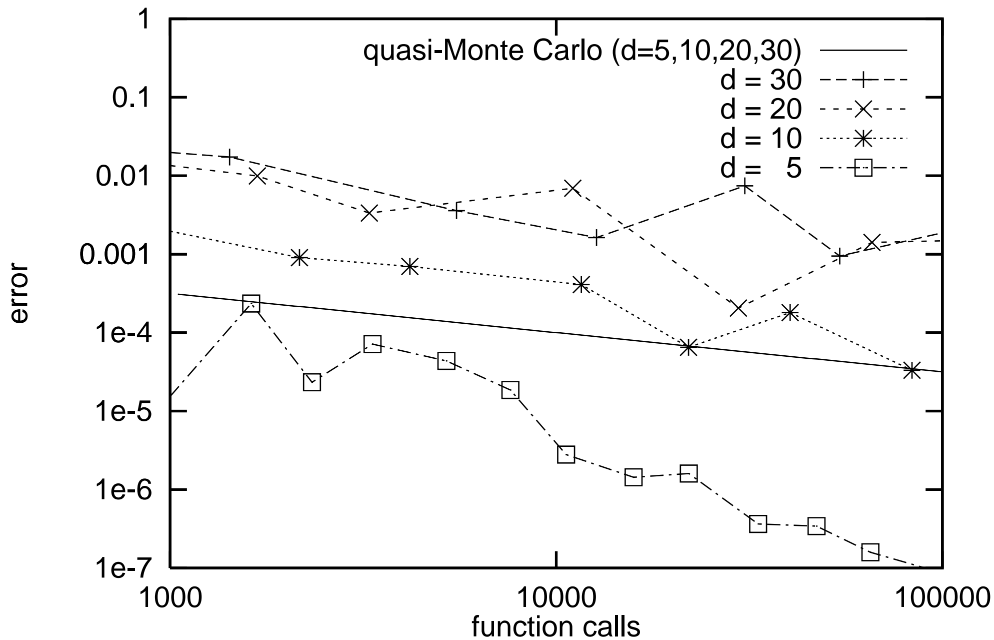


Figure 7.2: Computational results for different dimensions

7.2 Absorption Problem

The next example is simple transport problem described by the following integral equation [7]:

$$y(x) = x + \int_x^1 \gamma y(z)dz \tag{7.2}$$

A particle travels through a one dimensional slab of length one. In each step the particle travels a distance which is uniformly distributed on [0,1]. This may cause it to exit the slab or, otherwise it is absorbed with probability $1 - \gamma$. A particle at position x eventually leaves the slab with probability $y(x)$. The exact solution of this problem is given by:

$$y(x) = \frac{1}{\gamma} - \frac{1-\gamma}{\gamma}\, e^{\gamma(1-x)} \tag{7.3}$$

The same problem can also be represented as an infinite dimensional integral.

$$y(x) = \int_{[0,1]^\infty} \sum_{n=0}^{\infty} F_n(x,z)dz \tag{7.4}$$

whereas F_n is the probability for the particle to leave the slab after exactly n steps and the vector z contains the leap lengths.

$$F_n(x,z) = \gamma^n \theta\left((1-x) - \sum_{j=1}^{n} z_j\right)\theta\left(\sum_{j=1}^{n+1} z_j - (1-x)\right) \tag{7.5}$$

with the Heaviside function $\theta(s)$ defined as:

$$\theta(s) = \begin{cases} 1 & \text{for } s \geq 0 \\ 0 & \text{for } s < 0 \end{cases} \tag{7.6}$$

This discontinuous integrand is a kind of worst case function for a sparse grid, which was tuned for functions with bounded mixed derivatives. Discontinuities that are parallel to the coordinate axes can be integrated without great problems. As Figure 7.3 shows, this integrand has a step running diagonal, which leads to infinite mixed derivatives.
Since higher dimensions do not contribute considerably to the integral, they can be truncated to a finite number. Computational results of $y(0)$ with $\gamma = 0.5$ in 20 dimensions are shown in figure 7.4. The results for the adaptive sparse grid lie in the range of the standard Monte Carlo method, but can not compete with quasi-Monte Carlo. The same computation with a regular grid did not produce relative errors below 50% and would thus not even appear in this picture. Although the results might look bad for adaptive sparse grids, at least adaptivity rescues Monte Carlo-performance in a worst case.
Another interesting property is, that of the points in the adaptive grid 90 percent had surpluses equal to zero. Since the integrand is constant, once the particle left the slab, this is no big surprise. Quasi-Monte Carlo methods are not affected by additional constant directions, since it means effectively integrating the non-constant part with quasi-random vectors projected into the

non-constant subspace. In this example it turns out to be a great disadvantage, that a grid refinement generates points, which differ in only one coordinate from the existing points and have no volume contribution if the corresponding direction is constant.

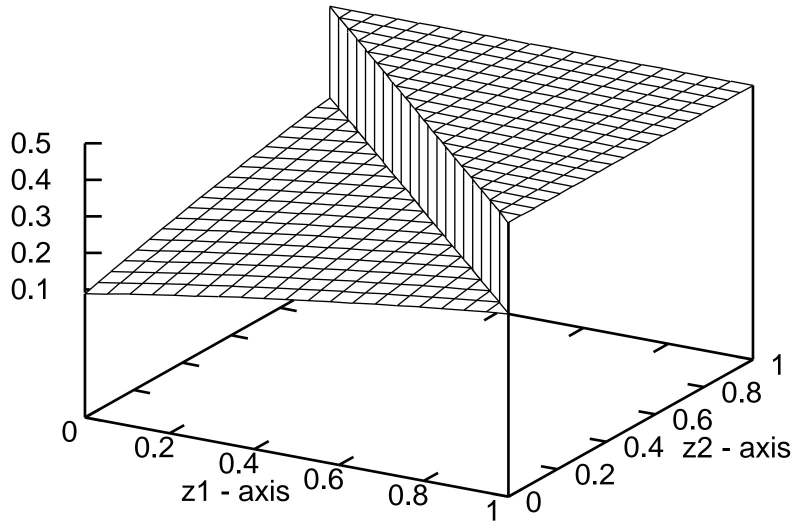


Figure 7.3: The integrand $\sum_{n=0}^{\infty} F_n(0, z)$ in the first two dimensions

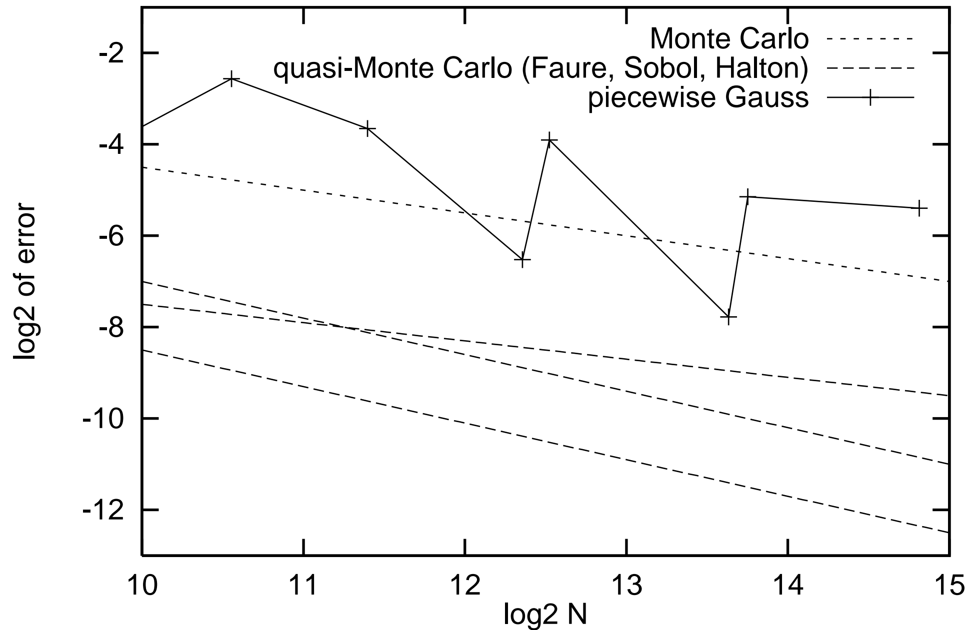


Figure 7.4: Results for the discontinuous integrand in 20 dimensions

Much better results can be obtained by describing the same problem with a smooth formulation. $F_n(x,z)$ in formula (7.4) can be replaced by $F_n^*(x,z)$ describing the contribution of each jump.

$$F_n^*(x,z) = \gamma^n (1-x)^n \left(\prod_{i=1}^{n-1} z_i^{n-i} \right) \left(1 - (1-x) \prod_{i=1}^{n} z_i \right) \tag{7.7}$$

With this smooth integrand the sparse grid properties can be fully exploited. Computational results for 20 dimensions are shown in figure 7.5. While the quasi-Monte Carlo Methods gain about two digits of accuracy, the sparse grid gains eight. With 30000 evaluated points the adaptive sparse grid outperforms quasi-Monte Carlo by about four digits.

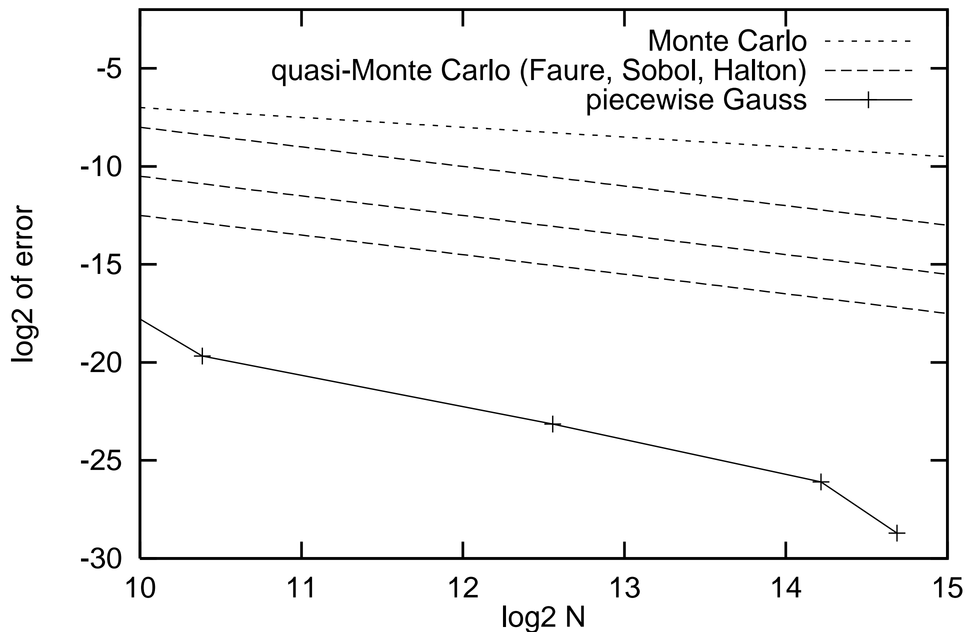


Figure 7.5: Results for the smooth integrand in 20 dimensions

Although this integrand is perfectly smooth, adaptivity is still needed, since higher dimensions still contribute only little to the total result. Unfortunately no Gauss-Patterson results were published for one of the twenty dimensional problems. It was however possible to further reduce the number of dimensions until the dimension independent quasi-Monte Carlo method could be outperformed. Figure 7.6 shows Gauss-Patterson beats the best of the quasi-Monte Carlo methods in eight dimensions but not the adaptive grid. The quadrature stops with an error of 10^{-8}, since this is the error, done by truncating the infinite dimensional integral to eight instead of twenty dimensions.

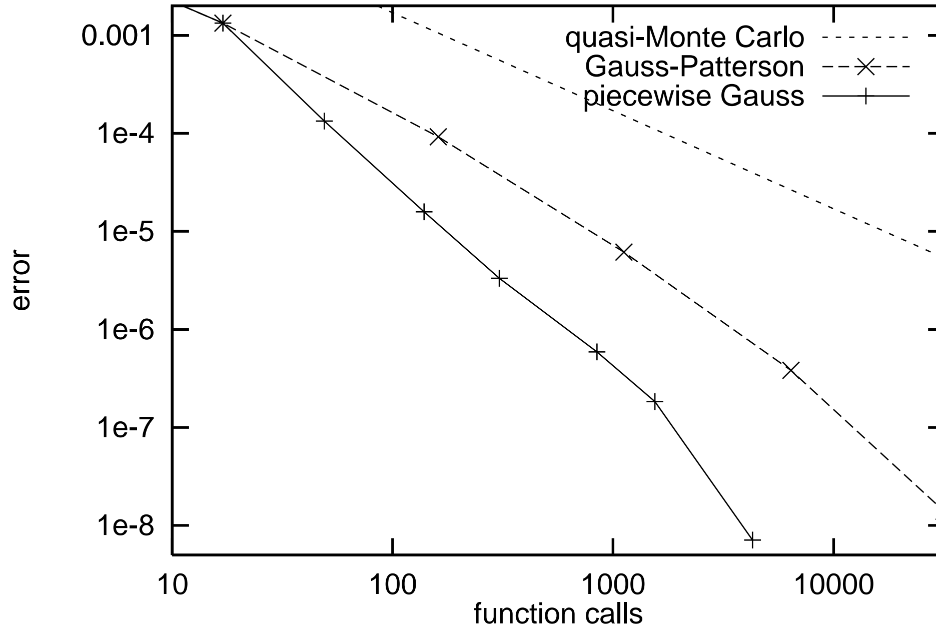


Figure 7.6: Results for the smooth integrand in 8 dimensions

7.3 CMO Problem

A typical collateralized mortgage obligation problem has been described in [6]. The present value of a security is the expectation value over the random variables involved in the interest rate fluctuations with each dimension representing one month.

$$\begin{aligned} PV &= E(v) \\ &= E(\sum_{k=1}^{d} u_k m_k) \end{aligned} \tag{7.8}$$

The variables of this problem are:

$$\begin{aligned} u_k &= \text{discount factor for month k} \\ m_k &= \text{cash flow for month k} \\ i_k &= \text{interest rate for month k} \\ w_k &= \text{fraction of remaining mortgages prepaying in month k} \\ r_k &= \text{fraction of remaining mortgages of month k} \\ c_k &= \text{(remaining annuity at month k)/c} \\ c &= \text{monthly payment} \\ \xi_k &= \text{an } \mathcal{N}(0,\sigma^2) \text{ random variable} \end{aligned}$$

Several of these variables can be computed as follows:

$$
\begin{aligned}
u_k &= \prod_{j=0}^{k-1} (1 + i_j)^{-1} \\
m_k &= c\, r_k((1 - w_k) + w_k c_k) \\
r_k &= \prod_{j=1}^{k-1} (1 - w_j) \\
c_k &= \sum_{j=0}^{d-k} (1 + i_0)^{-j}
\end{aligned}
\tag{7.9}
$$

The interest and prepayment rate can be computed from the random variable ξ_k and the initial interest rate i_0. The constant $K_0 = e^{-\sigma^2/2}$ normalizes the log-normal distribution, such that $E(i_k) = i_0$. K_1, K_2, K_3 and K_4 are parameters of the system.

$$
\begin{aligned}
i_k &= K_0\, e^{\xi_k}\, i_{k-1} \\
&= K_0^k\, e^{\xi_1 + \cdots + \xi_k}\, i_0 \\
w_k &= K_1 + K_2\, arctan(K_3 i_k + K4)
\end{aligned}
\tag{7.10}
$$

With the density of the normal distribution

$$
g(\xi) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{\xi^2}{2\sigma^2}} \tag{7.11}
$$

the expectation value can be written as an integral over R^d. Using the inverse distribution function $G(x)$ defined as $G^{-1}(x) = \int_{-\infty}^{x} g(\xi) d\xi$, the integral can then be transformed into an unweighted integral over the unit cube.

$$
\begin{aligned}
PV &= \int_{R^d} v(\xi_1, \cdots, \xi_d)\, g(\xi_1) \cdots g(\xi_d) d\xi_1 \cdots d\xi_d \\
&= \int_{[0,1]^d} v(G(x_1), \cdots, G(x_d))\, dx_1 \cdots dx_d
\end{aligned}
\tag{7.12}
$$

Brownian Bridge A more efficient way to compute this integral was suggested in [6]. They reduced the effective number of dimensions by representing the interest rate fluctuations as a Brownian motion $b(t)$. The future value $b(t+\Delta t_1)$ can be generated by a random jump from a past value:

$$
b(t_0 + \Delta t_1) = b(t) + \sqrt{\Delta t_1}\, \xi_{t_0 + \Delta t_1}. \tag{7.13}
$$

Using the Brownian bridge formula, $b(t + \Delta t_1)$ can also be computed from a future and a past value:

$$
b(t_0 + \Delta t_1) = (1 - a) b(t_0) + a b(t + \Delta t_1 + \Delta t_2) + c\, \xi_{t_0 + \Delta t_1} \tag{7.14}
$$

in which

$$
\begin{aligned}
a &= \frac{\Delta t_1}{\Delta t_1 + \Delta t_2} \\
c &= \sqrt{a \Delta t_2}.
\end{aligned}
\tag{7.15}
$$

Starting with $b(0) = 0$ and $b(d) = \sqrt{d}\,\xi_d$, the subsequent values can be found as the successive mid-points $b(d/2)$, $b(d/4)$, $b(3d/4)$, $b(d/8)$, $b(3d/8)$ and so on. Most of the functions variation is put into the dimensions defining the first few values of b. This increases the potential of adapativity and the performance of quasi-Monte Carlo quadrature. The interest rate fluctuations can now be computed from the Brownian bridge formula.

$$i_k = K_0^k \, e^{b(k)} \, i_0 \tag{7.16}$$

Numerical example in 256 dimensions In the first example, the piecewise Gauss and the Gauss-Patterson rule are compared. The parameters of the system are defined as follows:

$$(i_0, c, K_1, K_2, K_3, K_4, \sigma^2) := (0.007, 1.0, 0.01, -0.005, 10, 0.5, 0.0004)$$

Figure 7.7 shows, how Brownian bridges (BB) improve the power of adaptive schemes. In order to make this advantage accessible to Gauss-Patterson quadrature, the standard sparse grid construction was modified, such that a higher level formula was used in the 30 most important directions. These directions were found by a previous run up to a level of two. The piecewise Gauss rule finds these important directions itself, but shows a disadvantage where no use of adaptivity can be made, i.e. at the very beginning and at the non-Brownian bridge construction. Although the piecewise Gauss grid performs best for more than 10^6 points, there is no doubt that more sophisticated adaptivity applied to the Gauss-Patterson grid would lead to even better results.

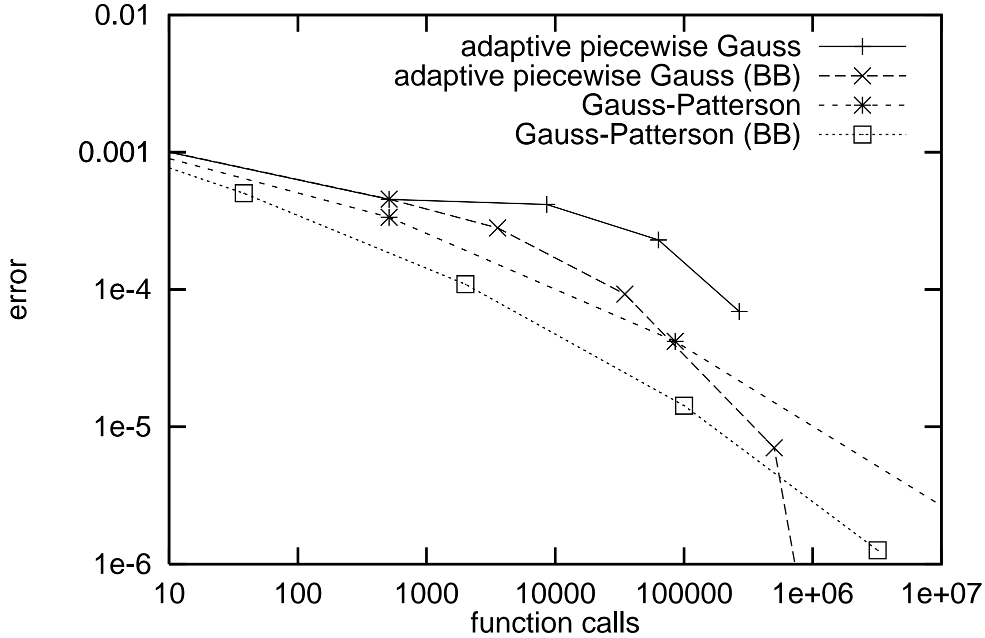


Figure 7.7: CMO-Problem in 256 dimensions

Numerical example in 360 dimensions Results for the Sobol' quasi-Monte Carlo method have been published for a 360 dimensional version of this example. The problem was found to be nearly anti-symmetric. Antithetic sampling can improve the performance of the quasi-Monte Carlo method, by integrating the function $(f(x) + f(-x))/2$. Since anti-symmetric functions produce symmetric adaptive grids, this property is automatically exploited by the sparse grid, except for the fact that the same results could be achieved by storing only half the number of grid points.
Figure 7.9 shows, that the adaptive sparse grid can not compete with the standard quasi-Monte Carlo method within the investigated sample size. An increasing downward slope can be expected with even more function evaluations. Because of the rather large algorithmic overhead and the exploding memory consumption, sparse grid are probably not applicable in these high dimensions.

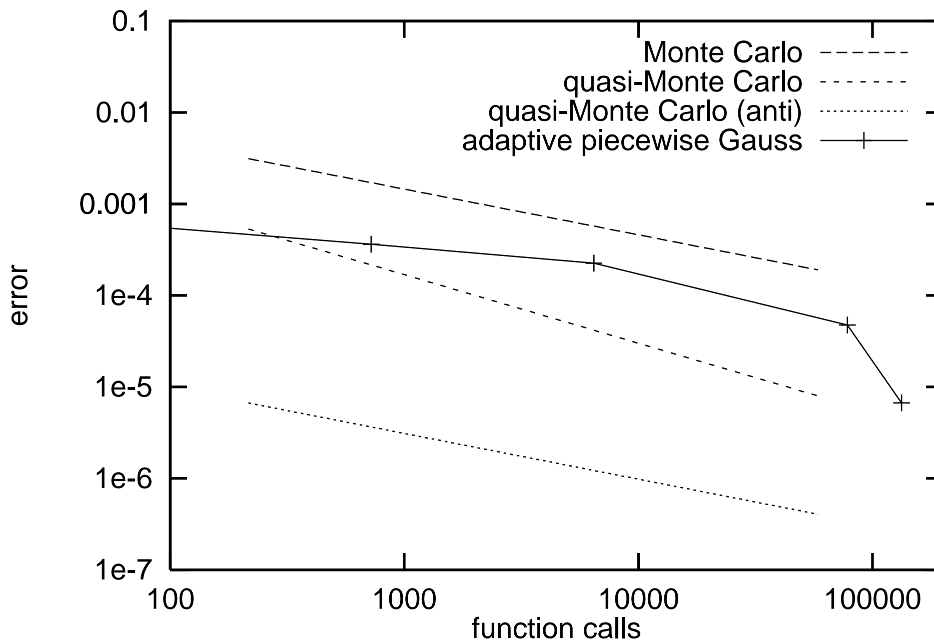


Figure 7.8: CMO-Problem in 360 dimensions (BB)

7.4 Asian option

This final example computes the exercise price of path-dependent options, known as Asian options. These options have values depending on the average of the underlying asset price at certain sampling times $t_1, .., t_d$ and can be computed as the expectation value of v.

$$v = u \cdot \left[(-1)^s \left(\frac{1}{d} \sum_{k=1}^{d} a_k - c \right) \right]^+ \tag{7.17}$$

with

$$[x]^+ = \begin{cases} x & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{7.18}$$

The variables of this problem are:

$$\begin{aligned} t_k &= \text{sampling times} \\ a_k &= \text{asset price at time } t_k \\ c &= \text{strike price} \\ s &= \text{call or put} \\ u &= \text{discount factor} \\ \sigma &= \text{volatility} \\ \xi_k &= \text{a } g_k = \mathcal{N}(0,\ \sigma^2(t_k - t_{k-1})) \text{ random variable} \\ \mu_k &= \text{drift for time } t_k. \end{aligned}$$

Following the notation of the previous example $K_0 = e^{-\sigma^2/2}$ normalizes the log-normal distribution and a_k's can be computed as jumps from past to future values

$$a_k = K_0^{t_k}\, e^{\xi_1+\cdots+\xi_k+\mu_k}\, a_0 \tag{7.19}$$

or with the Brownian bridge formula, which was used to obtain the numerical results in this section:

$$a_k = K_0^{t_k}\, e^{b(k)+\mu_k}\, a_0. \tag{7.20}$$

The expectation value can be written as an integral over R^d. Using the inverse distribution function $G_k(x)$ defined as $G_k^{-1}(x) = \int_{-\infty}^{x} g_k(\xi)d\xi$, the integral can then be transformed into an unweighted integral over the unit cube.

$$\begin{aligned} E(v) &= \int_{R^d} v(\xi_1,\cdots,\xi_d)\, g_1(\xi_1)\cdots g_d(\xi_d)d\xi_1\cdots d\xi_d \\ &= \int_{[0,1]^d} v(G_1(x_1),\cdots,G_d(x_d))\, dx_1\cdots dx_d \end{aligned} \tag{7.21}$$

The following two real world examples describe a call option with a different volatility and a different strike price. The integrand evaluates the average asset price and evaluates to zero where the average falls below the strike price. The unweighted integrand tends to infinity, since an unbounded asset price development can always increase the option's return. The common variables were set to:

$$\begin{aligned} d &= 13 \\ a_0 &= 125.83 \\ u &= 0.8 \\ s &= 0 \\ (t_0,..,d_d) &= (0, 3.89863, 3.91781, 3.93699, 3.95616, 3.97534, 3.99452 \\ &\quad 4.0137, 4.03288, 4.05205, 4.07123, 4.09041, 4.10959, 4.12877) \\ (\mu_1,..,\mu_d) &= (0.164786, 0.16581, 0.166836, 0.167864, 0.168894, 0.169927, \\ &\quad 0.170941, 0.171904, 0.172869, 0.173835, 0.174803, 0.175772, 0.176743). \end{aligned}$$

Example 1: In this first example the strike price is well below the spot price, thus making the exercise of this option rather likely. The exercising probability of $\approx 75\%$ leads to a support of the unweighted integrand on 75% of the unit cubes area. The smooth part of the function dominates in this examples and only a relatively small part is cut away by the $[\cdot]^+$ function.

Figure 7.9 shows the numerical results for the values $\sigma = 0.225$ and $c = 100$. The adaptive grid outperforms the Monte Carlo method and the accuracy of about 0.1%, which is typically desired in practice, can be achieved with only 1000 function evaluations.

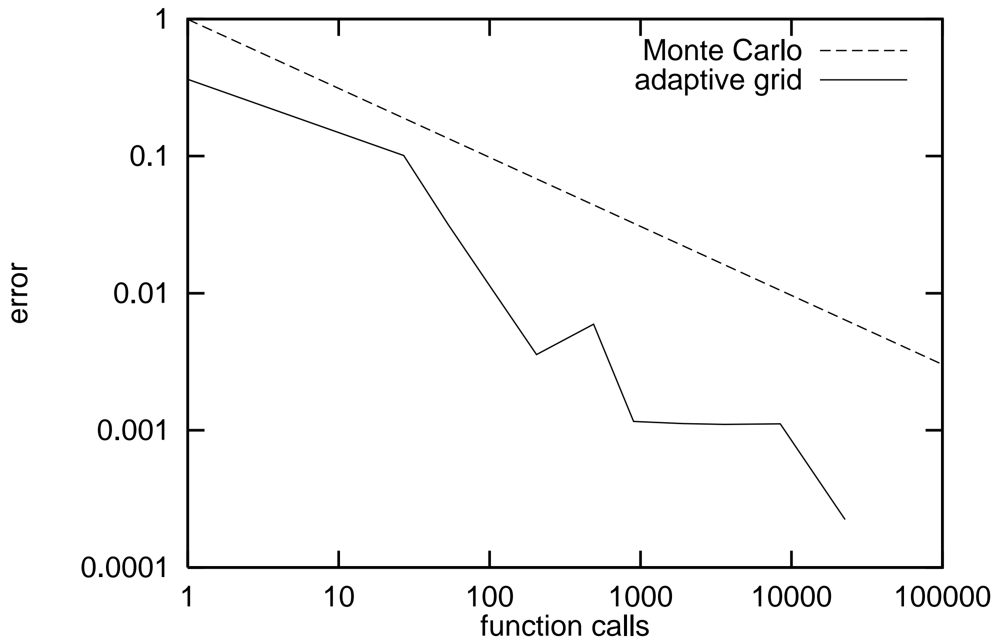


Figure 7.9: Asian option with exercise probability 75%

Example 2: In this example the values were set to $\sigma = 0.09$ and $c = 233$. A high strike price and a low volatility make this option rather unlikely to be exercised. The function's support covers 0.5% of the area and is squeezed tightly to the domain's border. By default the sparse grid would run in all directions and only find points which evaluate to zero. With the Brownian Bridge there is one most important direction which can be used to find at least one point of the support. It can be found on level 6 with the coordinates $(0, \cdots, 0, 0.996969)$. With only a total of 6 evaluated points the grid determines the integral already up to 10% accurately. Figure 7.10 shows that further refinement of this start grid does not considerably improve accuracy. Despite big head start the integral's approximation oscillates heavily and does not converge.

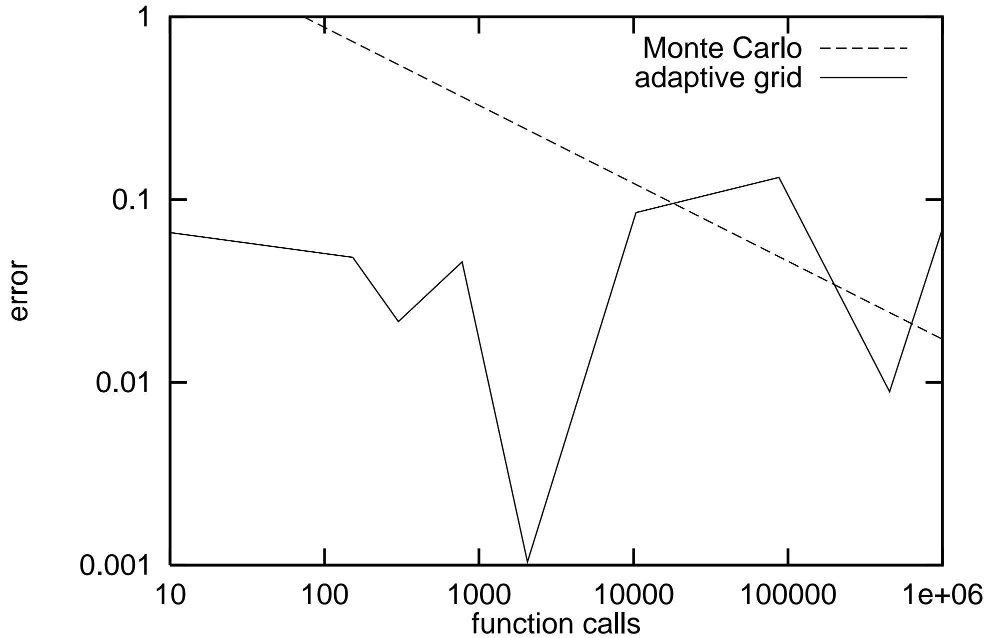


Figure 7.10: Asian option with exercise probability 0.5%

7.5 Run times and complexities

Although evaluating the function is the most time consuming factor in many application fields, for practical considerations it is useful to have an idea of run times and complexities of the investigated examples. The sparse grid algorithm consists of two main parts. First, evaluating the function at the desired points and second, computing the surpluses. All the example integrands had evaluation complexities $O(d)$ and contribution to the total run time was rather small. The most time consuming task was computing the surpluses. Computing a node's hierarchical surplus using basis functions of polynomial degree p was theoretically proven to have complexity $O(p^2d)$ [2]. The deepest computation in this publication reached a level of 17 in section 5.2 with a maximum polynomial degree of 16. Computational results showed that the time spent on functions involved in polynomial operations still had only minor influence. For all practical considerations the time complexity can be assumed to be $O(d)$. The adaptive algorithm has to recompute all surpluses every time new nodes are added to the grid. The new nodes typically outnumber the existing nodes by a large factor such that recomputation costs can be ignored. The total time complexity for creating a d-dimensional grid with N points is $O(dN)$. This is the same complexity as for the Monte Carlo method where d random numbers have to be created for every node. This suggests a constant asymptotic ratio of time needed by Monte Carlo and the sparse grid to handle a certain amount of nodes. Since Monte Carlo only depends on Processor speed while sparse grids heavily depend on main memory access, this ratio depends on the employed hardware. On a Sun SPARC with 300 MHz, the CMO-Problem (section 7.3)

was used to compare the run times for generating a certain amount of nodes with the current sparse grid implementation and a simple Monte Carlo method.

grid	dim	level	nodes	sparse grid	Monte Carlo	ratio
regular	10	7	397825	84s	17s	4.9
adaptive	10	15	543062	106s	25s	4.2
"	50	9	87025	64s	19s	3.4
"	100	7	48049	65s	20s	3.3
"	200	6	48705	235s	43s	5.5

In terms of generated points per time the Monte Carlo Method outperforms the sparse grid by a factor ranging from 3.3 to 5.5. The 10-dimensional example is computed with a regular and an adaptive grid. Despite a minor performance difference there are no serious slow downs resulting from a higher polynomial degree. Total integration speed ups for the published examples can be expected for the smooth integrands, where performance gains where already encountered in terms of function evaluations, i.e. the smooth absorption problem or the first example of the Asian option.

8 Conclusion

In this publication a powerful adaptive sparse grid algorithm based on piecewise polynomials has been outlined. Its performance has been demonstrated in various examples and was compared to quasi-Monte Carlo and a non-adaptive version of a Gauss-Patterson based sparse grid. The final results can be summarized in these propositions:

Sparse grids require smooth integrands. In the discontinuous example the quasi-Monte Carlo method greatly outperformed the sparse grid. Although adaptivity avoided the fiasco experienced with a regular grid, the adaptive strategy could not place the points more efficiently than quasi-random sequences.

Higher polynomial degree of exactness increases performance. The Gauss-Patterson rule has a polynomial exactness of $O(2^l)$ on level l, compared to $O(l)$ in the piecewise case. The Gauss-Patterson rule can therefore produce more accurate results on one level l and its accuracy accelerates quicker with increasing levels. On the other hand, piecewise basis functions can reach higher levels with much fewer points, whenever adaptivity can be applied.

Adaptivity is an essential ingredient to sparse grids. Despite claims that non-adaptive sparse grids can "outperform Quasi-Monte Carlo methods by about a factor of 2 in the fitted convergence rate α"[4], this turned out to hold only for a very limited set of carefully chosen examples. Many practical examples do have nearly constant directions and can not be represented with the same accuracy in fewer dimensions, as section 7.2 shows. This does not despise Gauss-Patterson quadrature in general, since it would theoretically be possible to implement efficient directed adaptivity. The point-wise adaptivity of piecewise Gauss quadrature might be overkill in many application fields.

Sparse grids decrease performance with dimensions and increase performance with time. Sparse grids are much more sensitive to dimensionality than quasi-Monte Carlo methods, especially when the integrand is nearly constant in additional directions. On the other hand, convergence rates grow with the number of function evaluations. This can mainly be accounted to the growing polynomial degree of exactness and the growing adaptive efficiency. By and large sparse grids are applicable in low to moderate dimensional problems with moderate to high accuracy requirements.

Acknowledgement: I would like to thank Hans-Joachim Bungartz for supervising this project.

Bibliography

[1] Thomas Bonk. *Ein rekursiver Algorithmus zur adaptiven numerischen Quadratur mehrdimensionaler Funktionen.* Fakultät für Informatik, Technische Universität München, 1994.

[2] Hans-Joachim Bungartz. *Finite Elements of Higher Order.* Shaker Verlag, 1998.

[3] Philip J. Davis and Philip Rabinowitz. *Numerical Integration.* Blaisdell Publishing Company, 1967.

[4] Thomas Gerstner and Michael Griebel. *Numerical Integration using Sparse Grids.* Institut für Angewandte Mathematik, Universität Bonn, 1997.

[5] M. Griebel. *Adaptive sparse grid multilevel methods for elliptic PDEs based on finite differences.* Institut für Angewandte Mathematik, Universität Bonn, 1997.

[6] Art Owen Russel E.Caflisch, William Morokoff. Valuation of mortgage backed securities using brownian bridges to reduce effective dimension, 1997.

[7] William Morokoff Russel E.Caflisch. Quasi-monte carlo integration, 1995.

[8] S.A. Smolyak. *Quadrature and interopaltion formulas for tensor products of certain classes of functions*, pages 240–243. Dokl.Akad.Nauk SSSR 4, 1963.