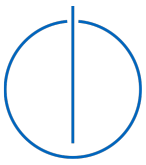
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Dimension-wise Spatial-adaptive Refinement with the Sparse Grid Combination Technique

Hendrik Möller



DEPARTMENT OF INFORMATICS

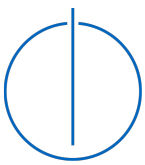
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Dimension-wise Spatial-adaptive Refinement with the Sparse Grid Combination Technique

Dimensionsweise räumlich-adaptive Verfeinerung mit der Sparse Grid Kombinationstechnik

Author:	Hendrik Möller
Supervisor:	Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Advisor:	M.Sc. Michael Obersteiner
Submission Date:	October 15, 2018



I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, October 15, 2018 Hendrik Möller

Acknowledgments

I want to acknowledge

- . . . my advisor, Michael Obersteiner, for his patience, availability and the time he put into our regular meetings,
- . . . my friends who are always entertaining and therefore keeping the mood up,
- . . . my brother for his support.

Abstract

Spatial adaptive refinement with the sparse grid combination technique is an effective way to adapt the combination technique to non-smooth functions. One problem of the standard combination technique is that all grids are always regular, making local refinement impossible. This paper tackles this problem by presenting the use of a dimension-wise spatial-adaptive refinement strategy. It also shows how error estimation can be tweaked in order to increase effectiveness and multiple variants tested with the presented algorithm and compared to the standard combination technique.

Räumlich adaptive Verfeinerung mit der Sparse Grid Kombinationstechnik is ein effektiver Weg um die Kombinationstechnik an ungleichmäßige Funktionen anzupassen. Ein Problem der Standard-Kombinationstechnik ist, das alle Gitter immer regulär sind, was lokale Verfeinerung unmöglich macht. Diese Arbeit beschäftigt sich mit diesem Problem und präsentiert eine dimensionsweise räumlich-adaptive Verfeinerungsstrategie. Es zeigt im Detail, wie eine Fehlerabschätzung effizienter genutzt werden kann sowie mehrere Varianten, die mit dem präsentierten Algorithmus getestet und mit der Standard Kombinationstechnik verglichen werden.

Contents

List of Notations

Notation	Description
$\mathbb{R}$	The real numbers including the zero.
$\boldsymbol{u}$	The vector u.
u_i	ith element of vector $\boldsymbol{u}$.
$\mathcal{O}$	Big O Notation.
$\lvert x \rvert_n$	The l_n norm of x.
$\Phi(x)$	The *hat function*.
x_i	Sampling point with index i.
$\Phi_i(x)$	The *hat function* for the point with index i.
h	The mesh size.
d	Number of dimensions.
ℓ	The discretization level.
w	The width of a basis function.
s_i	The support of the *hat function* from point with index i.
α_i	Function value of the gridpoint with index i.
N	Number of grid points.
$u_{\ell,i}$	The surplus of the basis function of the point $x_{\ell,i}$.
$\boldsymbol{\ell}$	A vector of levels.
$\boldsymbol{h_\ell}$	A vector of mesh sizes.
$\Omega_{\boldsymbol{\ell}}$	The grid with levels $\boldsymbol{\ell}$.
W_ℓ	Hierarchical increment space with level ℓ.
I	Index set.
V_ℓ	Nodal space of level ℓ.
ϵ_{tol}	The termination tolerance.
r_{tol}	The refinement tolerance.
v_ℓ	The volume of a basis function of level ℓ.
$\Omega_{\boldsymbol{\ell}}^{(c)}$	The combined *sparse grid* with level vector $\boldsymbol{\ell}$.
ℓ_{min}	The minimum level in the *combination technique*.
ℓ_{max}	The level of the *combination technique*.
$U \bigoplus V$	The direct sum of U and V, i.e. $u + v : \forall u \in U, v \in V$

1 Introduction

Calculating the integral of a function $f(x)$ is a typical problem in numerics. In many cases, the function $f(x)$ is not known but only the function values at certain, potentially arbitrary, points. In this scenario, a common approach is to approximate the integral using the finite set of sampling points. This is a non-trivial problem.

A naive approach would interpolate between sampling points and determine the surface under the interpolation function, using for example the trapezoidal rule. Doing so on a regular grid in higher dimensions would result in the *full grid* method. But the full grid method is affected by the so called *curse of dimensionality*, meaning that the degrees of freedom are exponentially dependent on d, being the dimensionality of the problem. So in a higher-dimensional problem, even modern devices would have a runtime way to high to be practical for integrating $f(x)$.

Since the full grid method is non-feasible for most practical usecases, Zenger (1991) [1] developed the *sparse grid* method, reducing the degrees of freedom and making the *sparse grid* algorithm executable in a reasonable amount of time.

While not solving the *curse of dimensionality*, this method reduces its impact on computation times and pushes the boundary of computational feasibility into higher-dimensional problems. One problem is that some conventional methods cannot be used with *sparse grids* anymore without adjusting them to the hierarchical basis, which might be complicated.

The *combination technique* solves that problem by dividing the *sparse grid* problem into smaller anisotropic grids and reconstructing the *sparse grid* solution from these smaller grids. It still has fewer degrees of freedom than the ordinary full grid method while having a slightly higher computation effort than the *sparse grid* approach.

One important topic in the *sparse grid combination technique* is the method on how to refine a current sparse grid in order to adapt to the problem at hand. One simple choice would be spatial adapting area-wise, dividing the smaller full grids into distinct areas and adding points by evaluating the error of one area. This thesis presents an alternative approach in which grids are not refined area-wise, but dimension-wise.

This paper is structured as follows: In chapter 2, the fundamental principles behind *full grids*, *sparse grids*, the *combination technique* as well as the hierarchical approach and spatial-adaptive refinement will be introduced. In chapter 3, the dimension-wise spatial-adaptive refinement method is presented and discussed in detail. The chapter 4

shows how the algorithm was implemented and how certain details in the algorithm were handled. Following are a series of tests and their results in chapter 5 with an analysis on how the adaptive strategy performs. A short review and outlook in chapter 6 concludes the paper.

2 Background

In this chapter, all necessary information about *full grids*, *sparse grids*, *combination technique* and *adaptive refinement* can be found as well as all mathematical notations used.

2.1 Full Grid

Reducing the *full grid* problem to one dimension, the solution is calculated by formalizing basis functions for every sampling point used. One famous linear basis function is the *hat function* (see Figure 2.1), defined as

$$\Phi(x) = \max(1 - |x|, 0) \tag{2.1}$$

on which the following methods are based on.

2.1.1 Nodal and Hierarchical Basis

Considering only the space $[0,1] \rightarrow \mathbb{R}$ and applying this onto any grid point $x_i = i \cdot h, 0 < i < 2^\ell$ with i being the index of the point and h the mesh width, the point-specific function is calculated by

$$\Phi_i(x) = \Phi\left(\frac{x - x_i}{h_i}\right) \quad . \tag{2.2}$$

With ℓ being the discretization level and therefore being the number of equidistant parts the mesh is divided into,

$$h_i = 2^{-\ell} \tag{2.3}$$

$$\ell = -\log_2(h_i) \tag{2.4}$$

the nodal basis V_ℓ is defined as

$$V_\ell : \left\{\Phi_{\ell,i} | \, 1 \leq i \leq 2^\ell - 1\right\} \quad . \tag{2.5}$$

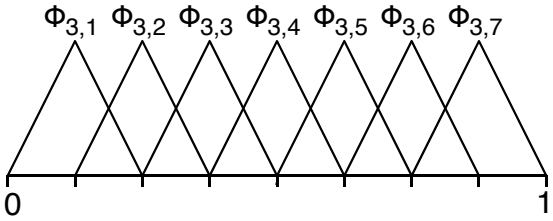


Figure 2.1: Example of nodal basis [taken from 2].

The width w of all *hat functions* describes the width of the "triangle". In the nodal basis, this would be

$$w = 2^{\ell-1} \quad . \tag{2.6}$$

The support s_i of a *hat function* is the interval from the left corner of the triangle to the right, containing all coordinates between them.

$$s_i = [x_i - w/2, x_i + w/2] \in \mathbb{R} \tag{2.7}$$

Note that both inherently only depend on the discretization level and every nodal basis function has the same width.

With α_i being the function value at grid point x_i and $N = 2^\ell + 1$ describing the amount of points, the integral of the function $f(x)$ can be calculated by computing the sum of the integral of all nodal basis functions:

$$\int_0^1 f(x)\mathrm{d}x \approx \sum_{i=0}^{N-1} \alpha_i \cdot \int_0^1 \Phi_i(x)\mathrm{d}x \tag{2.8}$$

Instead of using a nodal basis, one can use the hierarchical basis, giving each point $x_{\ell,i}$ a distinct level in a hierarchization tree. In this tree, points of a higher depth are children of the points with lower depth and every node can only have two parents. In Figure 2.2, $x_{1,1}$ is the parent of both $x_{2,1}$ and $x_{2,3}$, while $x_{3,1}$ and $x_{3,3}$ are children of $x_{2,1}$. With that, we can reconstruct the nodal base from Figure 2.1 by combining all hierarchical levels up to level ℓ (see Figure 2.2 and Figure 2.3) [3].

In the hierarchical basis, we get the specialized *hat-function*

$$\Phi_{\ell,i}(x) = \Phi(2^\ell \cdot x - i) \quad . \tag{2.9}$$

The width now gains an index as each point has a potentially different level:

$$w_{\ell,i} = 2^{\ell_i - 1} \tag{2.10}$$

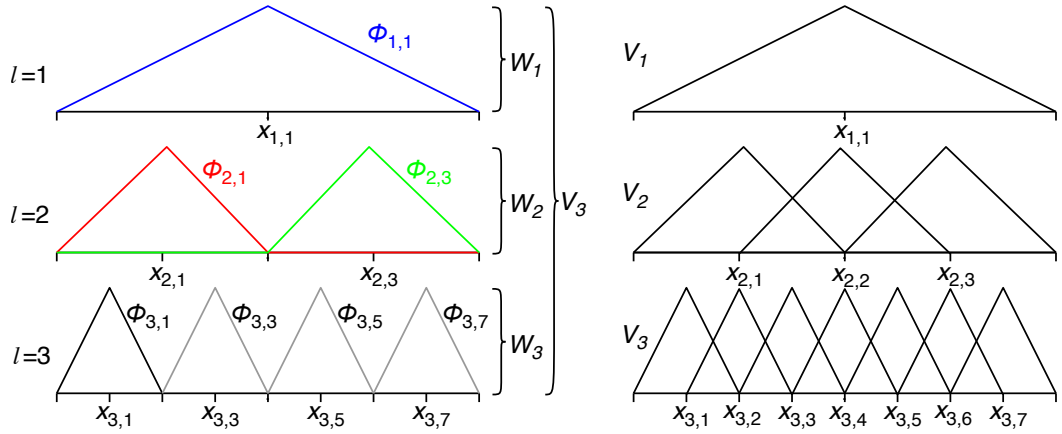


Figure 2.2: Comparison of the first three levels of hierarchical basis (left) and nodal basis (right) [taken from 2].

The support $s_{\ell,i}$ also just gains an index:

$$s_{\ell,i} = [x_i - w_{\ell,i}/2, x_i + w_{\ell,i}/2] \in \mathbb{R} \quad . \tag{2.11}$$

From that, the surpluses $u_{\ell,i}$ of the basis functions can be easily computed by the parents of the point:

$$u_{\ell,i} = \alpha_{\ell,i} - \frac{1}{2} \cdot (\alpha_{\ell,i-1} + \alpha_{\ell,i+1}); \quad i \text{ odd} \quad . \tag{2.12}$$

The calculation of the end result changes to:

$$I = \left\{ i \in \mathbb{N} : 0 \leq i \leq 2^{\ell}, \quad i \text{ odd} \right\} \tag{2.13}$$

$$\int_0^1 f(x)\mathrm{d}x \approx \int_0^1 \sum_{n=1}^{\ell} \sum_{i \in I} u_{n,i} \cdot \Phi_{n,i}(x)\mathrm{d}x = \sum_{n=1}^{\ell} \sum_{i \in I} u_{n,i} \cdot h_n \quad . \tag{2.14}$$

Figure 2.4 shows the difference of the nodal and hierarchical basis.

2.1.2 Multi dimensional grid

To be able to solve a *full grid* problem in higher dimensions, the tensor product can be utilized in a way that the methods above can be fully used (see Figure 2.5 and Figure 2.6). For simplicity purposes, only the domain $\Omega = [0,1]^d$ is considered.

For more than one dimension, the equations above have to be adapted. Introducing the dimension as another index, a multi-index $\boldsymbol{\ell}$ is defined:

$$\boldsymbol{\ell} = (\ell_1, \ldots, \ell_d) \in \mathbb{N}^d \tag{2.15}$$

$$\boldsymbol{h_\ell} = (h_{\ell_1}, \ldots, h_{\ell_d}) = 2^{-\boldsymbol{\ell}} = (2^{-\ell_1}, \ldots, 2^{-\ell_d}) \quad . \tag{2.16}$$

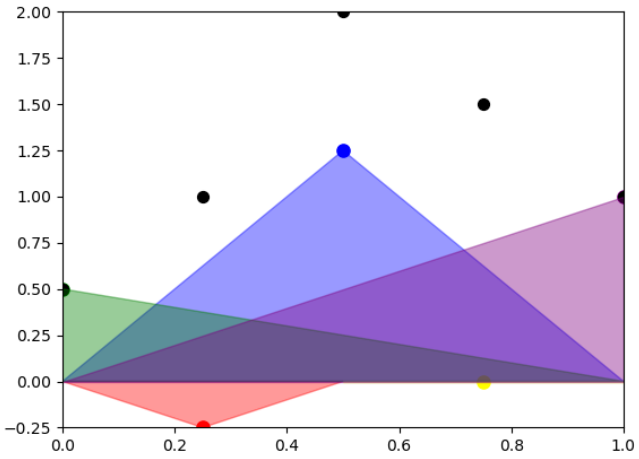


Figure 2.3: Construction of hierarchical basis functions and their calculated surpluses (colored dots) given sampling points (black dots).

Therefore, the grid $\Omega_{\boldsymbol{\ell}}$ consists of the grid points:

$$\boldsymbol{x}_{\boldsymbol{\ell},\boldsymbol{i}} = (x_{\ell_1,i_1}, \ldots, x_{\ell_d,i_d}) \tag{2.17}$$

$$x_{\ell_t,i_t} = i_t \cdot h_{\ell_t} = i_t \cdot 2^{-\ell_t}, \quad t = 1, \ldots, d \quad . \tag{2.18}$$

Then, a space V_ℓ of piecewise d-linear functions is associated for a grid $\Omega_{\boldsymbol{\ell}}$:

$$V_\ell = \operatorname{span}\left\{\Phi_{\ell,i} \middle| \, i_t \in [0, \ldots, 2^{\ell_t}], t \in [1, d]\right\} \quad . \tag{2.19}$$

The d-dimensional piecewise d-linear *hat-functions*, whose span is V_ℓ, are defined as following:

$$\Phi_{\ell,i}(x) = \prod_{t=1}^{d} \Phi_{\ell_t,i_t}(x_t) \quad . \tag{2.20}$$

The one-dimensional function Φ_{ℓ_t,i_t} changes to

$$\Phi_{\ell_t,i_t} = \begin{cases} 1 - |x/h_\ell - i_t| & , x \in [(i_t - 1)h_\ell, (i_t + 1)h_l] \cap [0, 1] \\ 0 & , \text{otherwise} \end{cases} \quad . \tag{2.21}$$

Now that *full grids* are well-defined, the next chapter is going to introduce the concept of *sparse grids*.

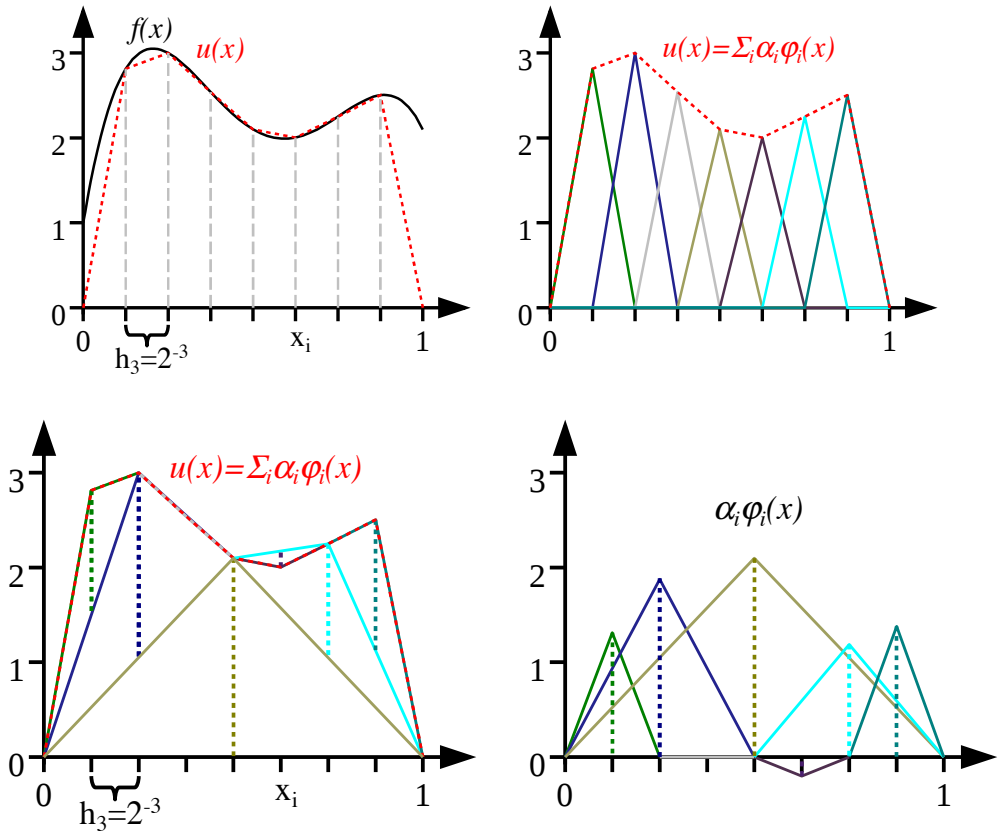


Figure 2.4: Side-by-side example of constructing the *hat-functions* (right) in one dimension with nodal basis (top) and hierarchical basis (bottom), approximating the integral of the function $f(x)$ by the sum of the constructed triangles (left) [taken from 2].

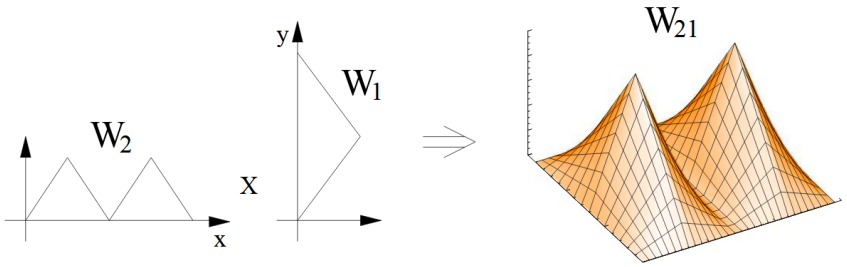


Figure 2.5: Generating the basis function $\Phi_{(2,1),(1,1)}$ from the one-dimensional basis functions $\Phi_{(2,1)}$, $\Phi_{(2,2)}$ and $\Phi_{(1,1)}$ using the tensor product [taken from 4].

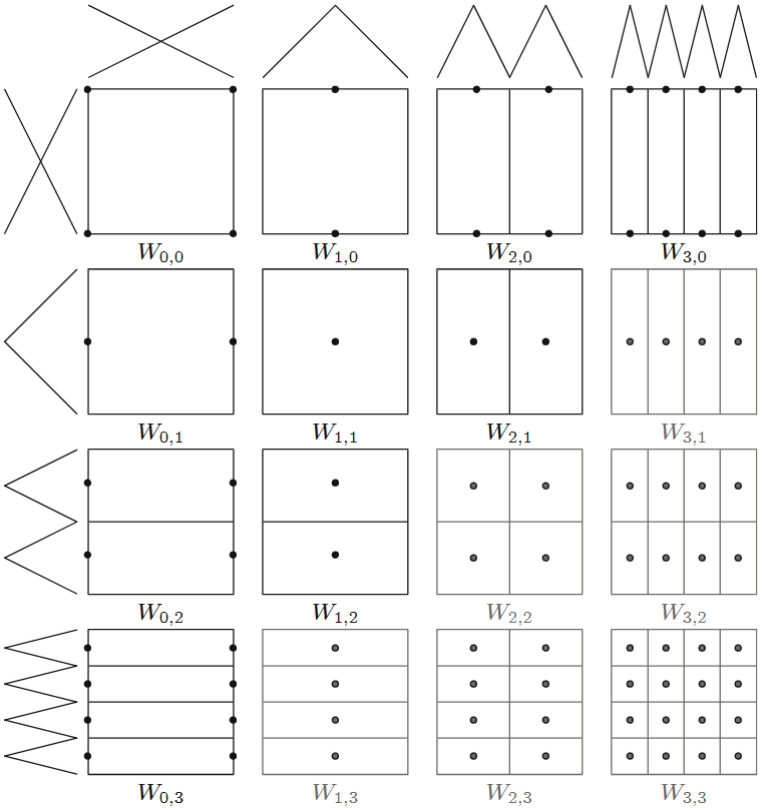


Figure 2.6: Example of a 2D tensor product usage with different levels on each axis and the corresponding grids [taken from 5].

	Full grid	Sparse grid
Degrees of Freedom	$\mathcal{O}(h^{-d})$	$\mathcal{O}(h^{-1}\log(h^{-1})^{d-1})$
Accuracy	$\mathcal{O}(h^2)$	$\mathcal{O}(h^2\log(h^{-1})^{d-1})$

Table 2.1: Comparison of *full grid* and *sparse grid* approach via their degrees of freedom and accuracy of approximation (assuming certain smoothness conditions)

2.2 Sparse Grid

To converge from a *full grid* to a *sparse grid*, the *full grid* space is split into subspaces W_ℓ with

$$W_\ell = \operatorname{span}\left\{\Phi_{\ell,i} : i \in I_\ell\right\} \tag{2.22}$$

and index set I_ℓ

$$I_\ell = \left\{i \in \mathbb{N}^d : 1 \leq i \leq 2^\ell - 1, \quad i_j \text{ odd for all } 1 \leq j \leq d\right\} \quad . \tag{2.23}$$

The hierarchical increment spaces W_ℓ are related to the nodal spaces V_ℓ, given by

$$V_\ell = \bigoplus_{k \leq \ell} W_k \tag{2.24}$$

with $\bigoplus$ representing the *direct sum* operator. The number of degrees of freedom of a subspace W_ℓ is bounded by

$$|W_\ell| = \mathcal{O}(2^{|\ell|_1}) \quad . \tag{2.25}$$

Resulting from an optimization corresponding to the number of degrees of freedom and the approximation accuracy [for details, see 4], the *sparse grid* spaces $V_{\text{sparse},n}$ of level n are defined as

$$V_{\text{sparse},n} = \bigoplus_{|\ell|_1 \leq n+d-1} W_\ell \quad . \tag{2.26}$$

An example can be seen in Figure 2.7

All in all, the *sparse grid* approach leads to fewer degrees of freedom while the accuracy isn't reduced that much, as Table 2.1 shows.

With *Sparse Grids* well established, the next chapter will introduce the concept of *adaptive refinement*.

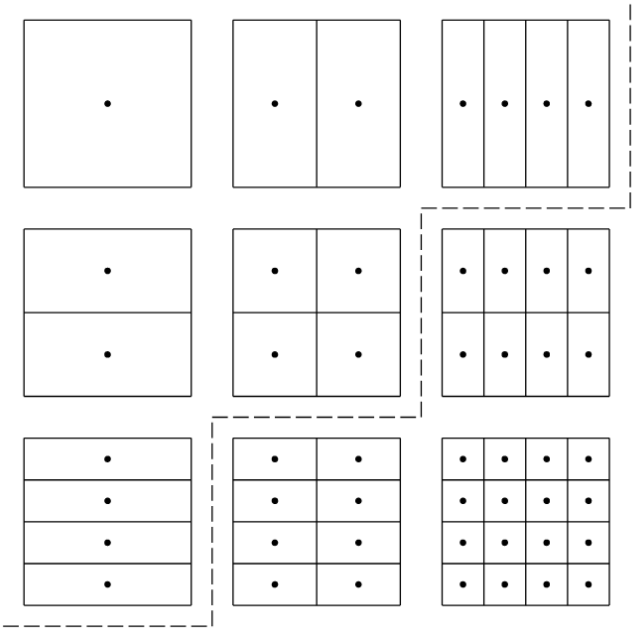


Figure 2.7: The subspaces W_ℓ for levels $|\ell|_1 \leq 4$ (above the dashed line) form the sparse grid space. The corresponding full grid space consists of all subaces W_ℓ for levels $|\ell|_\infty \leq 3$ [taken from 4].

2.3 Adaptive Refinement

Basic principle: starting with fewer grid points and adapting the grid by adding new grid points in specific regions [6]. By doing so, one can try to maximize the effectiveness of grid points while ignoring points that only have a small impact on the solution. Another reason for using *adaptive refinement* is costly function evaluations, making the use of too many grid points infeasible. Optimally, one wants to add the points such that they reduce the error as much as possible.

In the *sparse grid* approach, one can adapt locally by refining single points. Whenever a point is refined, all children of that point can bed added. Any of the children can be omitted, if it doesn't contibute much to the result. If a metric measuring the importance of one dimension over the next isn't available, one conventionally adds all points. A child of a point is described as a point positioned at half the distance to an existing neighbor and having a level one higher than the parent. This increases the number of grid points by up to $2 \cdot d$ every time. But as the parents for each grid point have to exist in order to calculate the surpluses, refining could result in a higher increase than $2 \cdot d$ grid points, as shown in Figure 2.8.

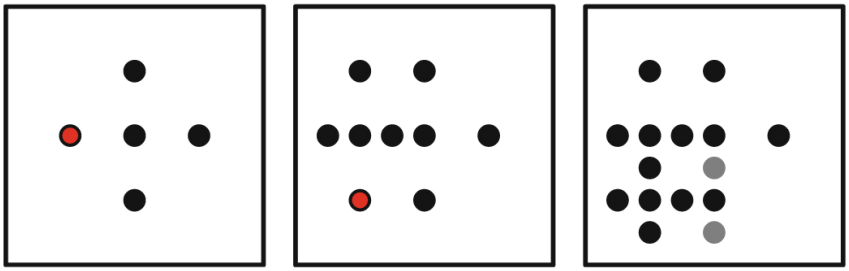


Figure 2.8: Starting with a sparse grid of level two (left) and wanting to refine the red marked point, we add all his children (middle). Doing that once more will leave us with more points (right). As we need the parents of each point, we have to add more, in this case gray depicted points (right) [taken from 6].

There are many methods on how to decide which points to refine [6]. All are based upon having an error estimation method, calculating an error value ϵ_i for each point. Then, a *refinement tolerance* r_{tol} is defined. In every step, all points that have an error value above r_{tol} are refined.

When using the hierarchical basis, a simple yet quite effective error estimator can be utilized. This *volume-guided* error estimator uses the *pagoda-volume* of each point as that points' error value. The *pagoda-volume* is basically the surplus $u_{\ell,i}$ of the point times the volume v_i of that points' basis function.

Interestingly, the volume of a basis function only depends on its level, not it's index or coordinates, as mesh-width and therefore height and form of the basis function also only depends on the level. One can therefore calculate the volume v_ℓ with

$$v_\ell = 2^{-1 \cdot |\boldsymbol{\ell}|_1} \quad . \tag{2.27}$$

In practice, many factors could make spatial refinement more difficult and even superfluous. For example, we could have a limited amount of grid points and no way of evaluating the function for more samples. In this case, we cannot refine any point we like, as some parents or the children might not be available, which makes the refinement process impossible. Refining many points in one step might lead to overshooting the goal while refining only few points per step might require multiple, potentially costly, refinement steps.

Many methods already have been developed for the full grid method, which cannot be directly used in a *sparse grid* problem. The next chapter presents a technique to remedy this problem.

2.4 Combination Technique

The *combination technique* solves a *sparse grid* problem by dividing the *sparse grid* into smaller anisotropic full grids and constructing the solution of the original problem as a linear combination of these smaller grids [7]. Another advantage is that the solution of each smaller grid does not depend on the other ones, meaning one can use the power of parallel computation to reduce runtime [8].

When discretizing a function $f(x)$ on a sequence of grids

$$\Omega_{\boldsymbol{\ell}} = \Omega_{\ell_1,\ldots,\ell_d} \tag{2.28}$$

we get uniform mesh sizes

$$h_t = 2^{-\ell_t} \quad , \tag{2.29}$$

with t being the t-th coordinate direction.

The *combination technique* of level $\ell_{\max}$ with certain minimal level $\boldsymbol{\ell}_{\min}$ considers all grids (see Figure 2.9) $\Omega_{\boldsymbol{\ell}}$ with

$$|\boldsymbol{\ell}|_1 = \ell_1 + \cdots + \ell_d = \ell_{\max} + (\ell_{\min} \cdot (d-1)) - q, \tag{2.30}$$

$$q = 0, \ldots, d-1, \quad \ell_t \geq \ell_{\min,t} \quad . \tag{2.31}$$

All $\Omega_{\boldsymbol{\ell}}$, from now on called *component grids*, are solved by calculating their corresponding integral approximation, giving a set of *component solutions* $v_{\boldsymbol{\ell}}$. The end result

$\Omega_{\ell}^{(c)}$ is calculated by combining all *component solutions* v_{ℓ} with a set of weights called *combination coefficients*

$$\Omega_{\ell}^{(c)} = \sum_{q=0}^{d-1} (-1)^q \cdot \binom{d-1}{q} \sum_{|\ell|_1 = \ell_{\max} + (\ell_{\min} \cdot (d-1)) - q} v_{\ell} \quad . \tag{2.32}$$

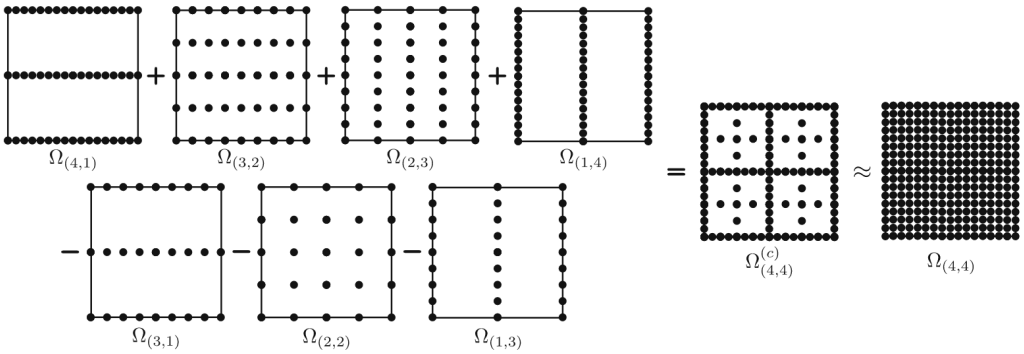


Figure 2.9: A *combination technique* construction with $\ell_{\max} = 4$ and $\boldsymbol{\ell}_{\min} = (1,1)$, with $\Omega_{4,4}$ representing the full grid of level $\ell_{\max}$ and $\Omega_{4,4}^{(c)}$ the resulting *sparse grid* [taken from 9].

As seen by the $(-1)^q$ in the equation (2.32), some component grids are subtracted. This is done because one don't want point doubling, so all duplicates have to be removed (see Figure 2.9).

The function values only depend on the position of the points and not on the actual grid configuration. Therefore, surplus values do not differ between a direct *sparse grid* or a *combination technique* implementation as exactly the same values are used during the hierarchization. Other *sparse grid* applications like *partial differential equations* are different, as the point values might depend on the discretization.

Having explained the fundamental principles, the next chapter presents the *dimension-wise spatial adaptive refinement* approach and discusses it in detail.

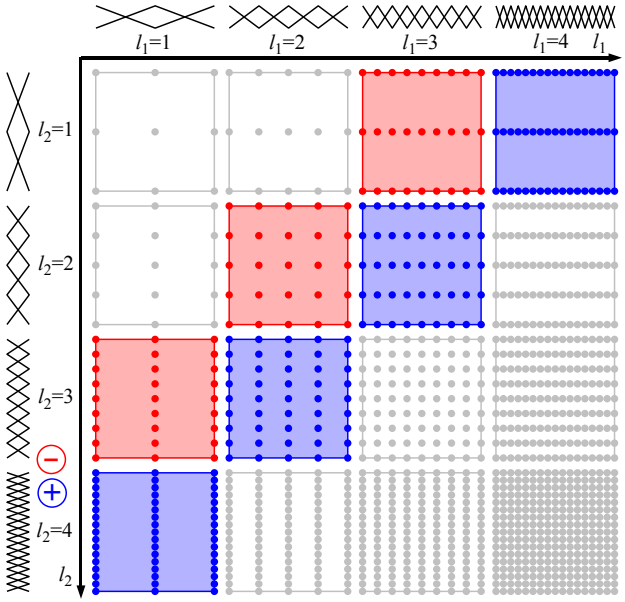


Figure 2.10: Example of a *combination technique* of level $\ell_{\max} = 4$ and $\boldsymbol{\ell}_{\min} = (1, 1)$ in two dimensions. The blue grids have a positive factor while the red ones are being subtracted from the solution [taken from 2].

3 Dimension-wise refinement

Dimension-wise spatial adaptive refinement is a method that defines refinement not on individual grid points, but on dimensionally coherent sets of them. We are, informally speaking, refining whole stripes in the grid (see Figure 3.1). One of the key problems: When refining, the grid does not have to stay regular, but block-adaptive. Many methods described in chapter 2 do not work that easily with non-regular grids.

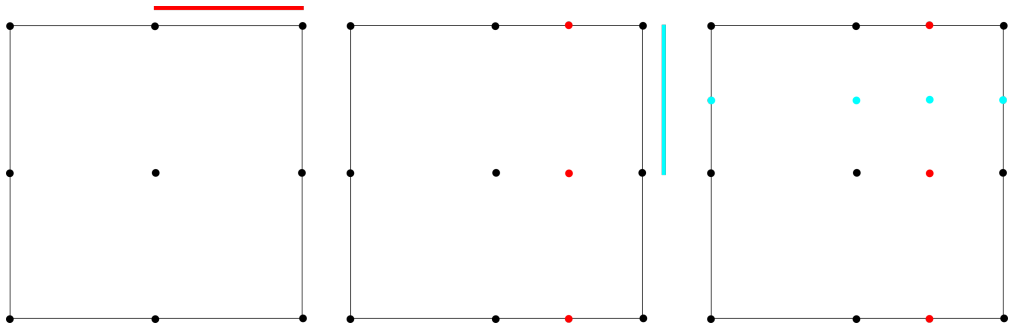


Figure 3.1: Starting on a grid with level 1 (left) and refining the second area of the first dimension (red depicted line), the red marked points are added (middle). Doing that again with the cyan marked area makes the top right area of the grid denser (right).

At first, this section is going to introduce the problem for one grid and one dimension. This will then be combined with the *combination technique* in chapter 3.2.

3.1 One grid

We need to construct our hierarchization tree, saving which node is a children of another, the coordinates and levels of all nodes. When evaluating this grid, the hierarchical basis can be built as described in chapter 2.1.1. The errors of each point can be calculated with the conventional *volume-guided* error estimator, as discussed in chapter 2.3.

Every grid point that is a leaf in the hierarchization tree (see Figure 3.2) is considered. Instead of refining individual points, this method considers areas as *refinement objects*. For example, when having three sampling points $x_{0,0} = 0,\ x_{1,1} = 0.5,\ x_{0,1} = 1$, there

are two *refinement objects*, one describing the area from $x_{0,0}$ to $x_{1,1}$ and the other one from $x_{1,1}$ to $x_{0,1}$.

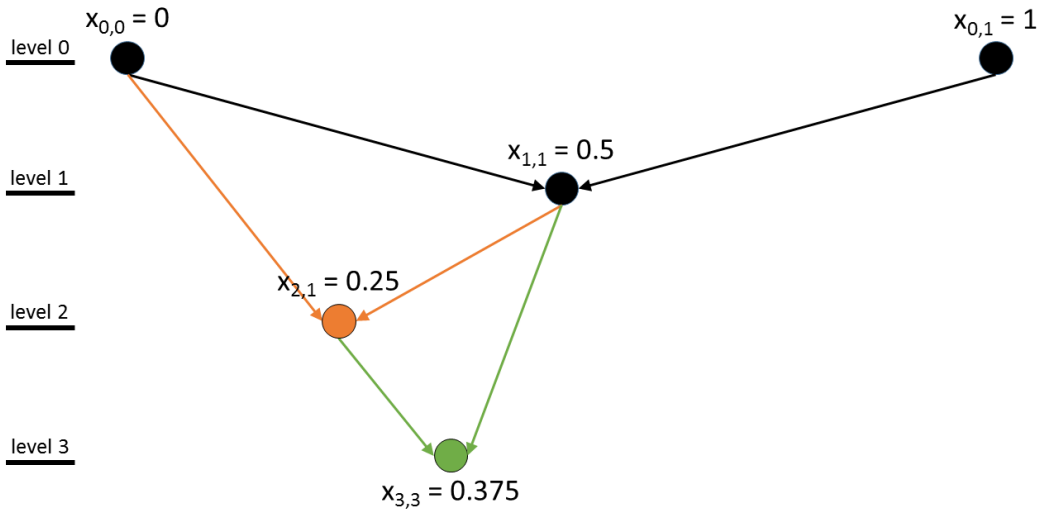


Figure 3.2: With nodes as individual grid points and edges being children links, the black nodes and edges represent our starting hierarchization tree of level 1. The orange node and edges are created when refining the first refinement object and the green ones are created by refining the second refinement object (respectively after the orange refinement).

Then, the error value of every grid point is mapped onto the corresponding *refinement objects*. That is done by adding the error to every *refinement object* that lies within the area of a points hierarchical basis function support. In the example above (see Figure 3.2), the only leaf is $x_{3,3}$. Its support ranges from 0.25 to 0.5, making its error value count towards both neighboring areas and therefore *refinement objects*.

In result, every *refinement object* having an error value above a defined *refinement tolerance* r_{tol} is refined by splitting the area (and therefore *refinement objects*) in two, adding a new grid point in its middle with a level one higher than the maximal one out of the parents levels. In the example above, if we were to refine the first area (ranging from 0 to 0.5), a new point $x_{2,1} = 0.25$ would be added.

Whenever one or more areas have been refined, the grid is evaluated again. If after any evaluation, no area's error value is above a defined *termination tolerance* ϵ_{tol}, the algorithm terminates.

Everything just discussed can be applied to any amount of dimensions, as the

one-dimensional problem describes one stripe of points and therefore areas in one dimension. Having more dimensions, there are simply more stripes and areas. For example, a two-dimensional grid of level one has three stripes along every dimension that needs to be considered. Along these stripes, the hierarchical basis functions are built like previously described (see also Figure 3.3). The *in-place* principle is employed, meaning that the input is directly overwritten with the output. In this method, the input is represented by the function values α_i, the output are the surpluses $u_{\ell,i}$ of the grid points.

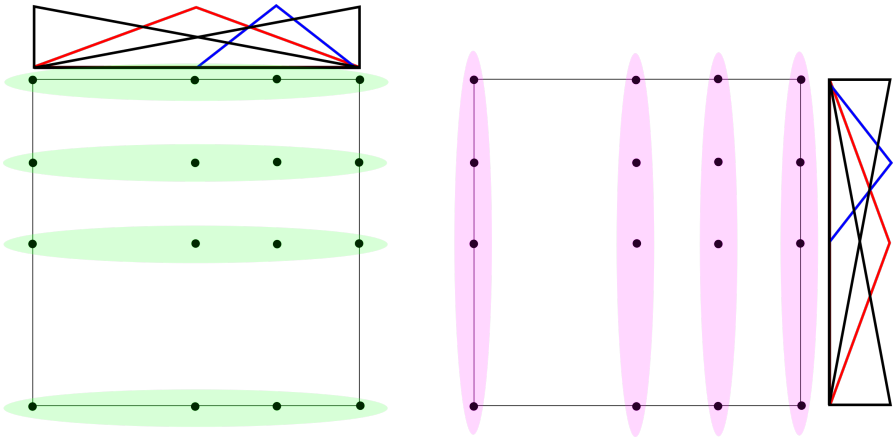


Figure 3.3: Hierarchization of the stripes in the first dimension (left), the green areas are the individual stripes. When all stripes in that dimension are done, the hierarchical basis functions are constructed for the stripes (pink areas) of the next dimension (right).

As the surplus calculation (see chapter 2.1.1) only depends on the parents values, the surpluses of no parent p_i of a point $x_{\ell,i}$ can be calculated before the surplus of the point $x_{\ell,i}$ itself. Otherwise, because of the *in-place* principle, the surplus instead of the function value of the parent would be taken into account. To calculate the surpluses in the correct order, the hierarchical basis functions are constructed for each stripe with the *bottom-up* principle, meaning that the surplus of the grid points having the highest level in that stripe is calculated first. After that, the levels are iterated through backwards from highest to the lowest level value.

When having multiple dimensions, the hierarchical basis functions have to be constructed for each grid point for each dimension. In order to do that properly, the functions are calculated of every stripe in one dimension before considering the next one.

3.2 Multiple Grids (Combination-Technique)

To be able to utilize the *combination technique*, the algorithm needs to be extended to support multiple grids.

As the refinement structure is the key point of this method, the refinement decisions should be taken into account as much as possible. To do that, the list of *refinement objects* and its information are globally used for all grids instead of having an individual list for each grid. Hence, whenever one or more areas are refined, it has an impact on the solution of multiple grids, not just one.

But the principles of the *combination technique* should function normally and right now, it would be possible to have grid points in our refinement construction that are not contained in all component grids of the *combination technique*. To sidestep this problem, before evaluating a grid, all points of a dimension for which the corresponding level exceeds the one of the component grid for the respective dimension are locally removed.

As an example, when evaluating the two-dimensional grid $V_{1,1}$, all points in both dimension with a level of two or higher are removed before the hierarchical basis functions are constructed. Another example is shown in Figure 3.4.

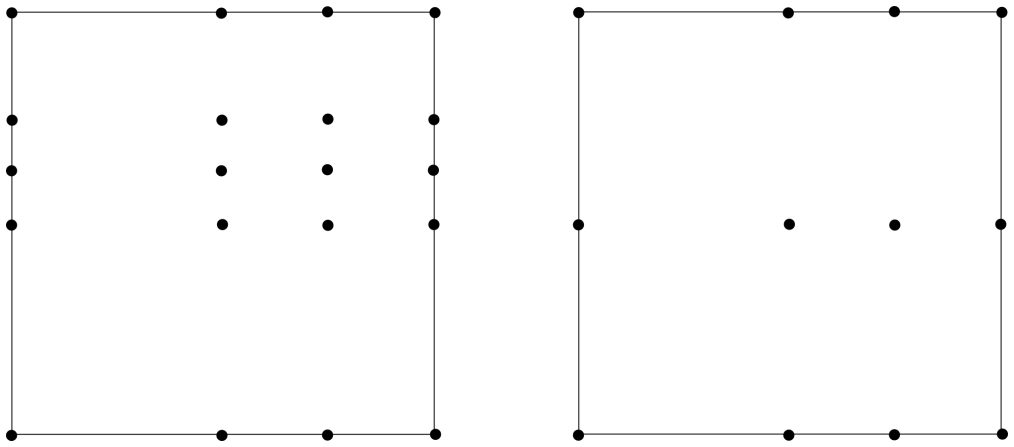


Figure 3.4: Example of a grid with all used points in current global refinement information (left). If the *combination technique* now needs the grid $V_{(2,1)}$, all points with levels greater than the corresponding level vector entries need to be removed (right).

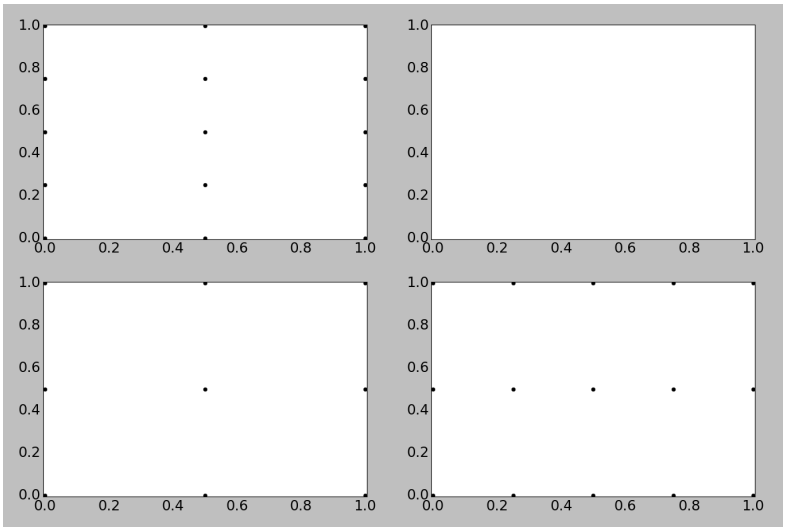
Therefore, the refinement data can be seen as a coarsening information which restricts the number of points in certain regions for the component grids, but doesn't add additional points.

If a point is refined and reaches a level greater than the one of the current *combination*

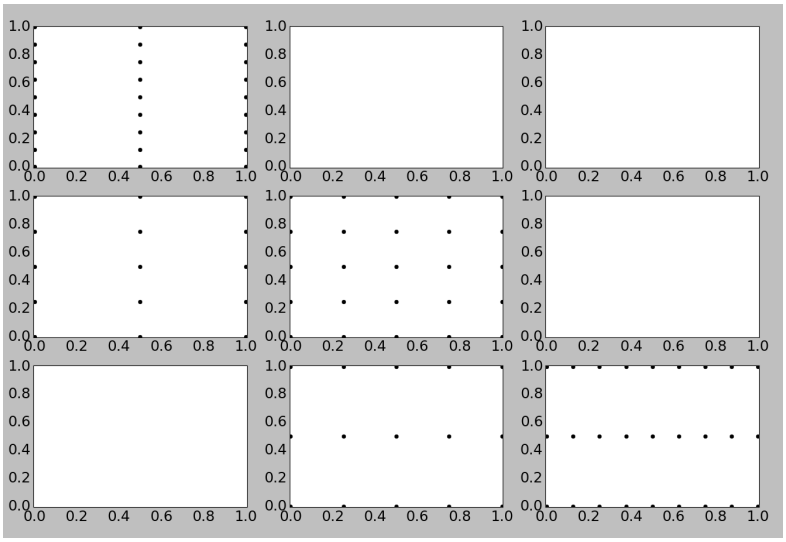
technique scheme, the entire scheme needs to be updated by increasing $\ell_{\max}$ of the *combination technique* by one, raising the number of grids and changing their level vectors (see Figure 3.5).

Again, if the tolerance ϵ_{tol} is reached after an evaluation of the whole *combination technique*, the program terminates.

Other techniques of spatial adapting the *combination technique* exist such as in [10], where the authors show how to split the domain into hierarchies of cells and refine them individually. In doing so, the algorithm only works on sub-domains that are spanned by single cells, not complete component grids. That's why the method is generally unsuited for an implementation that utilizes block adaptive full grid solvers, as they would need to be integrated and possibly even rewritten in the cell structure. On the other hand, the *dimension-wise spatial adaptive* method can output block adaptive component grids. These grids can be utilized by any existing solver that can handle those data structures.



(a) *Combination technique* with $\ell_{\max} = 2$.



(b) When (a) gets refined, the *combination technique* scheme is updated to this with $\ell_{\max} = 3$.

Figure 3.5: Example of increasing the $\ell_{\max}$ of a *combination technique*

4 Implementation

This chapter explains which specific strategies the program that was created within the scope of this thesis follows and goes into detail at which data structures are used.

4.1 Hierarchical Basis Function Construction

To construct the hierarchical basis functions, the program uses multiple data structures. First of which is the *gridPointValues*, storing the current grid point surpluses as an array in any arbitrary order. The *dictCoordinate* is a dictionary mapping the grid coordinates of every point to the corresponding index in *gridPointValues*. The *depthLevelVector* describes the depth of points in a stripe as an index for every dimension. For example, when looking at the grid $\Omega_{1,1}$, the *depthLevelVector* equals

$$[\,[0,2],[1]\,] \tag{4.1}$$

for both dimensions. That means that in a stripe, the first point (with index 0) and last point (with index 2) have the depth zero (as they are in the first bucket of the array) while the middle point has a depth of one. Therefore, when constructing the stripes and their hierarchical basis functions, the stripes are filled with the correct grid points in the correct order, which is from low to high coordinate value. In the example above, this would be the coordinates $[0, 0.5, 1]$ (in this order), as the program only considers the integral interval $[0, 1]$. As only with the *depthLevelVector*, it would be impossible to create children links for the hierarchization, another data structure, the *dictChildren* was used. This is a dictionary mapping grid points to their children as stripe indices to stripe indices. Using the example above with the grid $\Omega_{1,1}$, we get a *dictChildren* for each dimension as follows:

$$[\,0:[1],\ 1:[\,],\ 2:[1]\,] \tag{4.2}$$

Both the point at the left and right border have the center point at 0.5 as children and that point has no children, being a leaf in the hierarchization tree.

These four data structures are grid-specific and always constructed before evaluating a grid and deallocated after it.

4.2 Error Estimator

A *volume-guided* error estimator was employed. The program is implemented such that only error values of grid points that are leaves in every dimension (see Figure 4.1) are considered. Furthermore, only errors of the grids of level ℓ that do not have any forward neighbours, meaning for which no grid exists having a levelvector $\boldsymbol{\ell_2}$ with $\boldsymbol{\ell_2} \geq \boldsymbol{\ell}$, are used concerning the decision-making of the refinement.

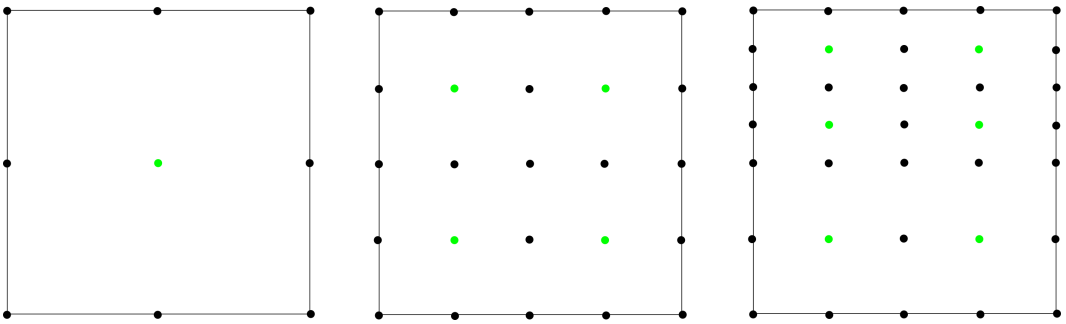


Figure 4.1: Starting with the grid $W_{1,1}$ (left), only the center point is a leaf in every dimension. When refining, we get a total of four error-relevant points (middle). If the upper two points errors are greater than the threshold, another two stripes are added (right). The green marked points are leaves in every dimension of that grid.

As described in chapter 3.2, points sometimes have to be locally removed. The refinement objects of these smaller areas still exist. Therefore, when applying the error value of a point to the corresponding refinement objects, the support of a basis function could reach more than two refinement objects. In this case, the error value is distributed over the whole range. So the percentage of the error value applied is based on the width of the corresponding refinement object.

4.3 Refinement

The current coarsening information is written into refinement objects. These are handled by *refinement containers*, which contain all refinement objects of one dimension. So there are d refinement containers, d being the dimensionality of the problem. As the grid-specific data structures (see chapter 4.1) are always constructed anew, the refinement process becomes relatively easy. Whenever one refinement object is refined, it is divided into two new refinement objects with half the width.

For the refinement decision-making process, two approaches were considered. The

first one uses the $\epsilon_{\text{tol}} = r_{\text{tol}}$, refining all refinement objects that are below the termination tolerance. The second one computes r_{tol} by using the maximal error ϵ_{max} of all refinement objects and a factor m called *margin*

$$r_{\text{tol}} = \epsilon_{\text{max}} \cdot m \quad . \tag{4.3}$$

Therefore it refines refinement objects starting with those with the largest error.

For the termination tolerance ϵ_{tol}, a user defined value is simply used and always checks

$$\epsilon_{\text{max}} < \epsilon_{\text{tol}} \quad , \tag{4.4}$$

meaning that if the maximal error of all refinement objects should be less than ϵ_{tol}, the program terminates.

4.4 Program Structure

In this section, python pseudocode can be found, showing the rough structure of the program. While Figure 4.2 shows how the multidimensional hierarchization was implemented, Figure 4.3 explains the main loop of the program.

```
# multi-dimensional construction of hierarchical basis functions
#  for one grid
def construct_hierarchical_basis_functions:
    for d in dimensions:
        # builds stripes for one dimension
        stripe = build_next_stripe_in_dim_d

        # construct the hierarchical basis functions for a stripe
        #  returning its leaves and the partial_integral
        for level from second_deepest to highest depth:
            for point with level:
                for children in point:
                    update_surplus_for(children)_with_parent(point)
                if point.has_no_children:
                    add_to_leaves(point)
    # looping over all leaves
    for leaf in leaves:
        calculate_support_of_basis_function_of(leaf)
        add_error_to_corresponding_refinement_objects
    return grid_integral
```

Figure 4.2: Pseudo code showing the multi-dimensional hierarchization

```
def run:
    # this is the main loop
    while (max_error > termination_tolerance):
        integral = 0
        # one step of our program means evaluating all grids
        #  in our combination technique scheme
        for grid in combi_scheme:
            partial_integral = evaluateGrid(grid)
            # calculates the result of the combination technique
            coeff = combi_technique_coefficient_for_grid(grid)
            integral += partial_integral * coeff

        # after evaluation, we have to refine
        do_refinements

# the evaluation of one grid
def evaluateGrid:
    # initialization of dictChildren, gridPoints,
    #  dictCoordinate and depthLevelVector
    init_grid_specific_data

    # construct the hierarchical basis functions for all stripes
    #  returning its partial_integral
    grid_integral = construct_hierarchical_basis_functions
    return grid_integral

# the refinement process
def do_refinements:
    # we loop over every refinement object, refining all
    #  that have an error above our refinement tolerance
    for refine_object in all_refinement_objects:
        if refine_object.error > refinement_tolerance:
            lmax_change = refine(refine_object)
            if lmax_change is not zero:
                update_combi_scheme
```

Figure 4.3: Pseudo code showing the structure of the program.

5 Results

To verify the approach, experiments with different functions have been conducted. These test cases were chosen to give a wide range of data, covering many potentially problematic cases. In order to do that, the test cases were run with the standard *combination technique* as well as with the *dimension-wise spatial-adaptive* refinement strategy, always one with a starting $\ell_{\text{min}} = 1$ and one with $\ell_{\text{min}} = 2$.

For all result graphs, the blue line belongs to the standard *combination technique* with $\ell_{\text{min}} = 1$; $\ell_{\text{max}} = 2$, while the green one is the algorithm discussed in this paper (abbreviated with SDH) using $\ell_{\text{min}} = 1$; $\ell_{\text{max}} = 2$ as starting level. Last but not least, the red line is also the adaptive algorithm but with a starting level of $\ell_{\text{min}} = 2$; $\ell_{\text{max}} = 3$. As a metric, the number of distinct function evaluations is used (horizontal axis in the plots). For comparison, the relative analytical error is given (vertical axis in the plots). The tests were started with the $r_{\text{tol}} = \epsilon_{\text{max}} \cdot 0.1$ approach to make the refinement decisions. The ϵ_{tol} were always chosen so that the tests were computable within the scope of this thesis.

5.1 Genz Gaussian

With a midpoint value m, coefficient factor c and d being the dimensionality of the problem, the *genz gaussian* function [from 11] is defined as:

$$x = x_0, \ldots, x_{d-1} \tag{5.1}$$

$$f(x) = \exp\left(-\sum_{i=0}^{d}(10^c \cdot (i+1) \cdot (x_i - m)^2)\right) \quad . \tag{5.2}$$

This function was used with two different setups explained below.

5.1.1 Corner case

The setup with $m = 0.99$ and $c = 2$ (see Figure 5.1) is called the *corner case*. When looking at the function, it is to be expected, that the area near 1 in every dimension should be denser. To check that, tests with only two dimension were run, so that the resulting *sparse grid* can be plotted well. Figure 5.2 clearly shows that this is the case.

The test results from the corner case can be seen in Figure 5.3, 5.4 and 5.5, running it with dimensionality two to four.

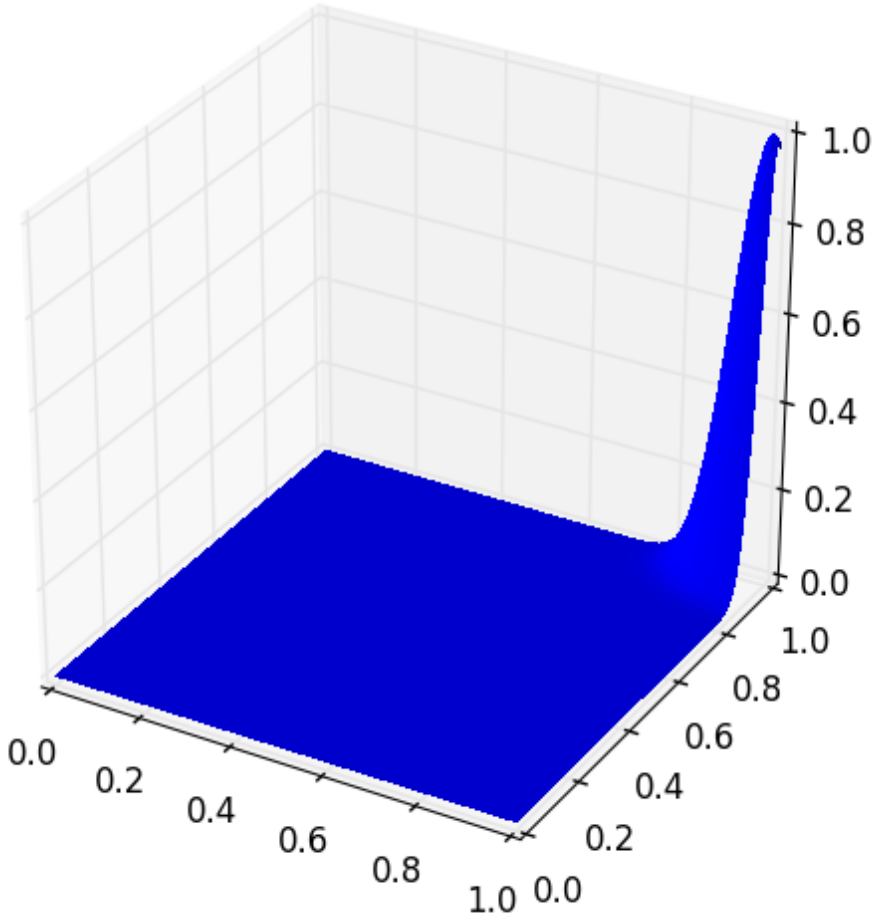


Figure 5.1: Two-dimensional print of the *Genz "Gaussian"* function with $m = 0.99$ and $c = 2$.

As seen in Figure 5.3, the red and green line are nearly always below the blue line up to a certain amount of points. Interestingly, in higher dimensions (see Figure 5.4 and 5.5), while the adaptive method with $\ell_{\text{min}} = 1$ stays near the standard *combination technique* (blue line), the red line (adaptive method with $\ell_{\text{min}} = 2$) seems to be better the higher d is. This can be explained with that by functions like the corner case, a higher starting $\ell_{\text{min}} > 1$ could be always straight up better. To verify that, the same test is run with a standard *combination technique* starting at $\ell_{\text{min}} = 2$.

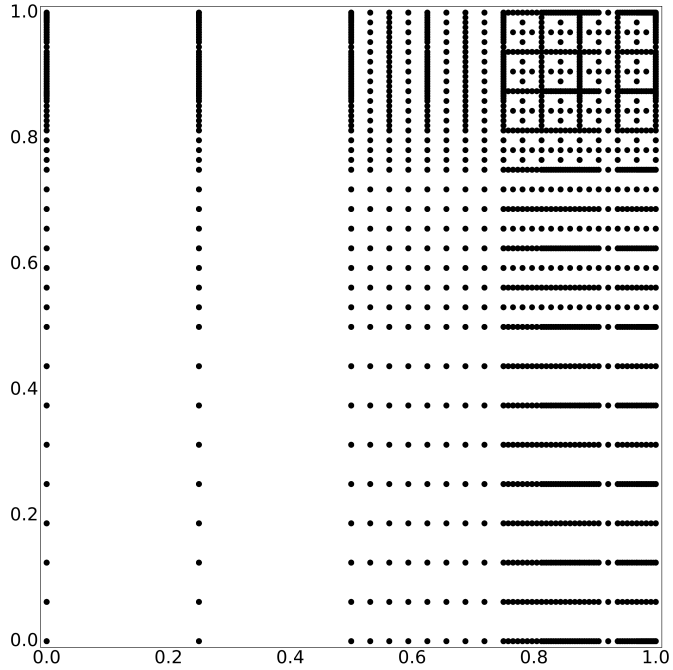


Figure 5.2: Resulting sparse grid when using the presented algorithm with $d = 2$ and $\epsilon_{\text{tol}} = 10^{-8}$

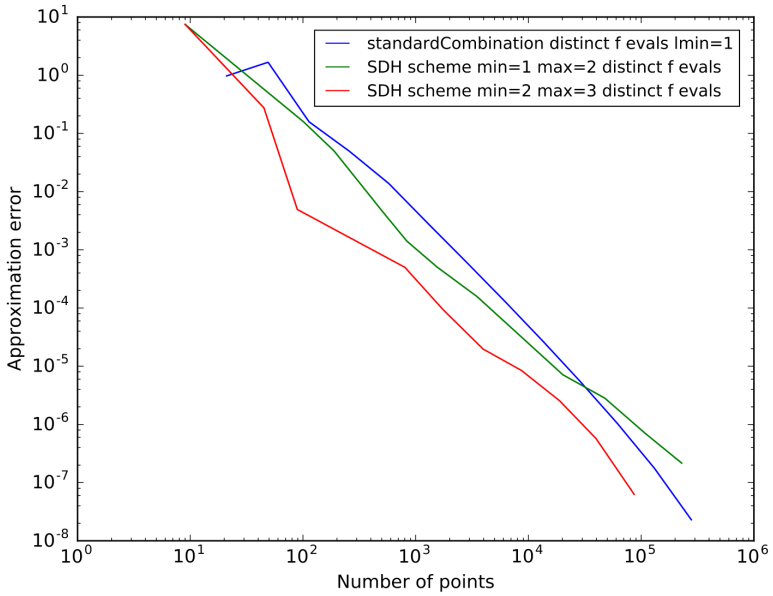


Figure 5.3: The corner test case when running with $d = 2$ and $\epsilon_{\text{tol}} = 10^{-12}$.

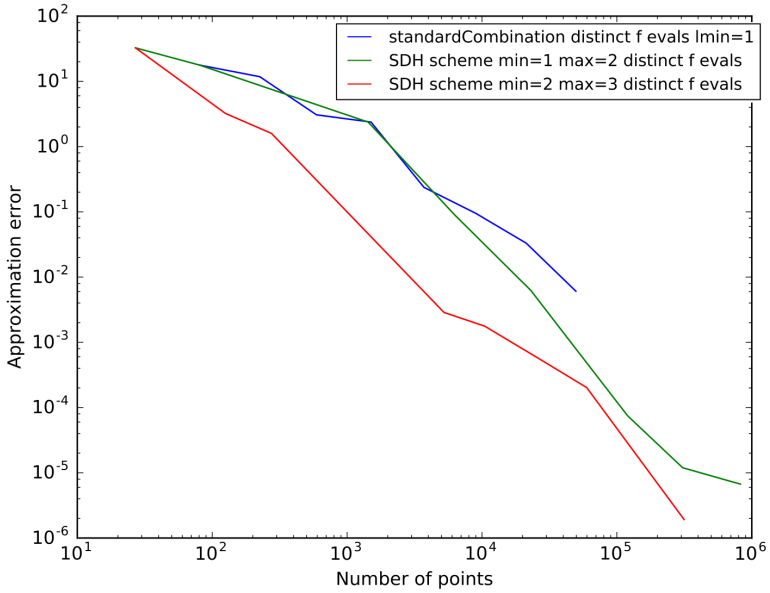


Figure 5.4: The corner test case when running with $d = 3$ and $\epsilon_{\text{tol}} = 10^{-10}$.

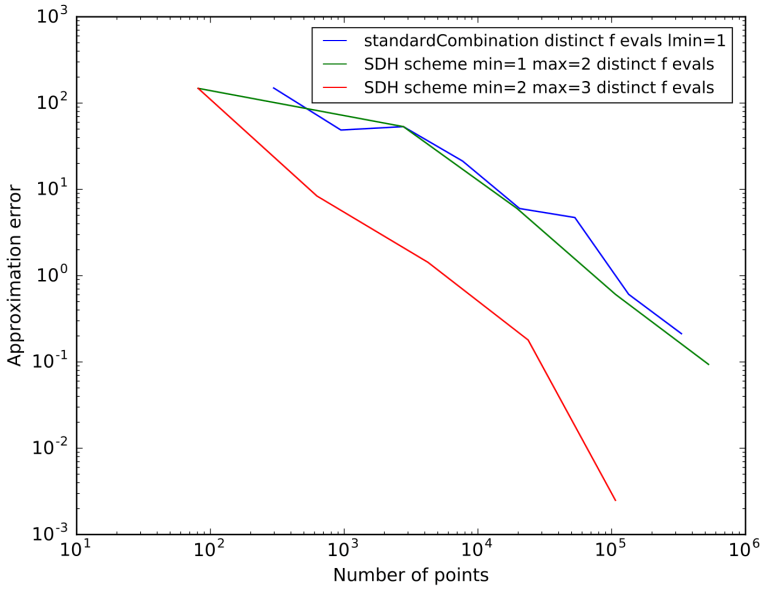


Figure 5.5: The corner test case when running with $d = 4$ and $\epsilon_{\text{tol}} = 10^{-7}$.

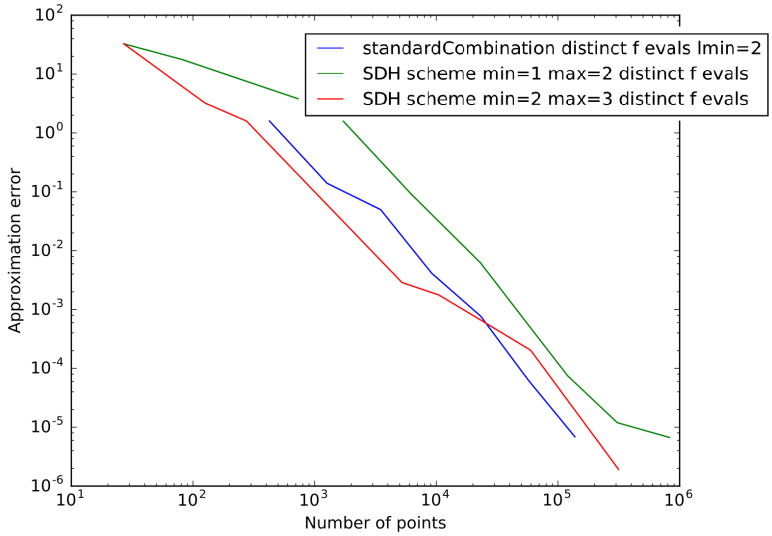


Figure 5.6: The corner test case when running with $d = 3$ and $\epsilon_{\text{tol}} = 10^{-10}$, but instead of using the standard *combination technique* with $\ell_{\text{min}} = 1$, it is used with $\ell_{\text{min}} = 2$.

As one can see from Figure 5.6, the adaptive algorithm with $\ell_{\text{min}} = 2$ follows a similar curve than the one representing the standard *combination technique* with $\ell_{\text{min}} = 2$. As expected, the $\ell_{\text{min}} = 2$ approach seems to be generally better in that test case. But still, the adaptive algorithm, especially with fewer than 10^4 points, seems to perform better than the standard *combination technique*.

The first idea to improve the algorithm was instead of using $r_{\text{tol}} = \epsilon_{\text{max}} \cdot 0.1$, all objects are refined that are below the termination tolerance, in short using $r_{\text{tol}} = \epsilon_{\text{tol}}$.

It can be clearly seen that the adaptive strategy $\ell_{\text{min}} = 1$ curve (green line) already becomes worse (in relation to the standard *combination technique*) after around 10^3 points while with respective $r_{\text{tol}} = \epsilon_{\text{max}} \cdot 0.1$ test case (see Figure 5.3), the lim $= 1$ curve crosses the blue line (standard *combination technique*) approximately at $3 \cdot 10^4$. Furthermore, the $\ell_{\text{min}} = 2$ seems to become less consistent with more ups and downs. So from now on, only the $r_{\text{tol}} = \epsilon_{\text{max}} \cdot 0.1$ approach will be considered as it seems more promising.

Another idea was to make already dense regions less probable for the refinement decision-making process to refine. In order to do that, the error values of each refinement object are multiplied with the size of the region they are representing. This is called the *volume-guided-punished-depth* error estimator. To be able to compare that, the corner case test was run with the same dimensionalities.

Comparing the results using the *volume-guided-punished-depth* error estimator (see

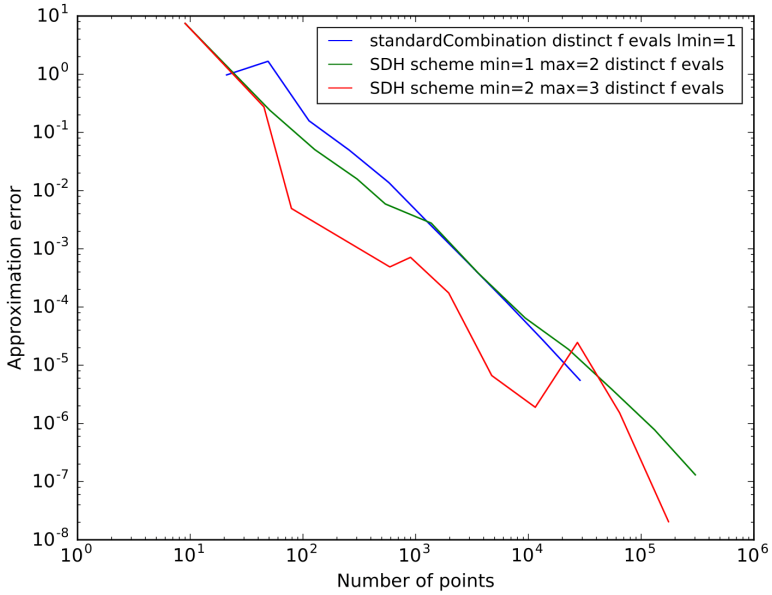


Figure 5.7: Result of the corner test case using $d = 2$ and $r_{\text{tol}} = \epsilon_{\text{tol}} = 10^{-13}$.

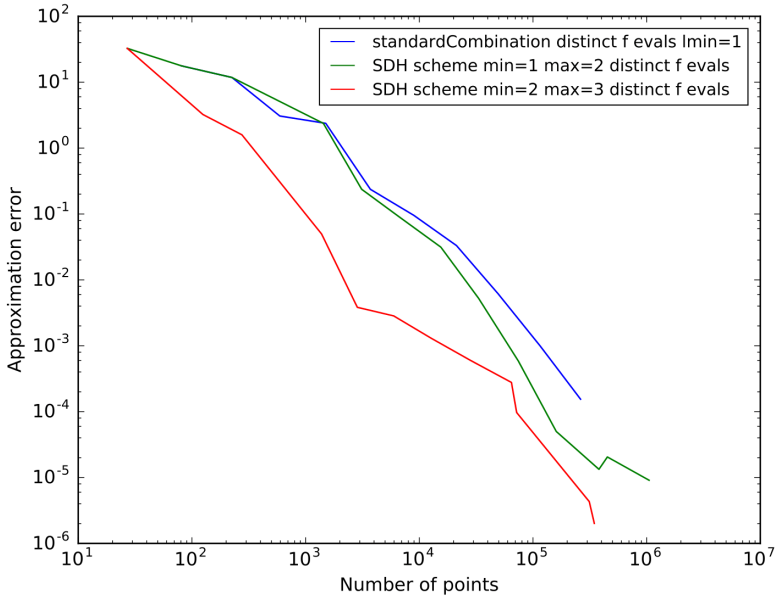


Figure 5.8: Result of the corner test case using the *volume-guided-punished-depth* error estimator and with $d = 3$ and $\epsilon_{\text{tol}} = 10^{-14}$.

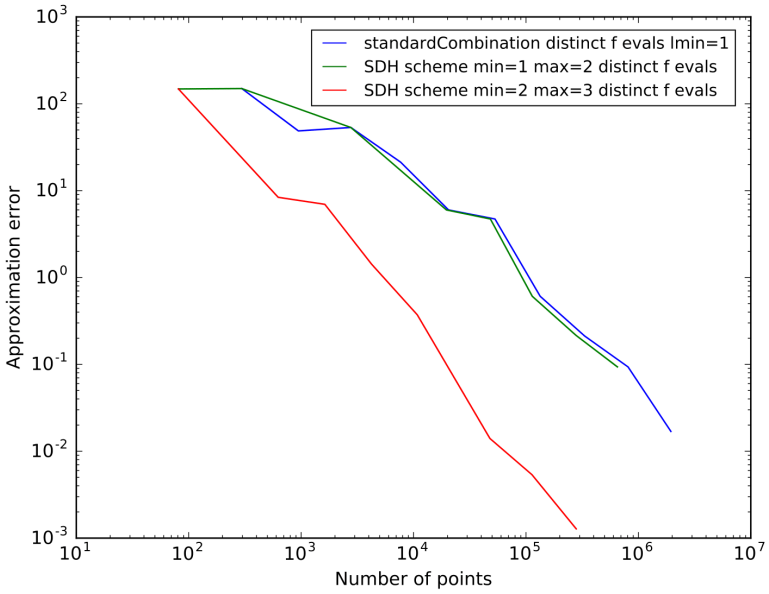


Figure 5.9: Result of the corner test case using the *volume-guided-punished-depth* error estimator and with $d = 4$ and $\epsilon_{\text{tol}} = 10^{-10}$.

Figure 5.8 and 5.9) to their corresponding test cases using the convential *volume-guided* error estimator (see Figure 5.4 and 5.5), there are slight differences. In the $d = 3$ cases, the number of points, from which on the adaptive algorithm consistently performs better, is lower. Interestingly, in the $d = 4$ case, there is not that kind of an improvement. While both $\ell_{\text{min}} = 1$ and $\ell_{\text{min}} = 2$ of the adaptive algorithm (green and red line) seem to perform better with more points than their corresponding test using the *volume-guided* error estimator, they perform a little bit worse in the area of low amounts of grid points.

5.1.2 Center case

The *genz gaussian* function was also used with $m = 0.5$ and $c = 2$ (see Figure 5.10)

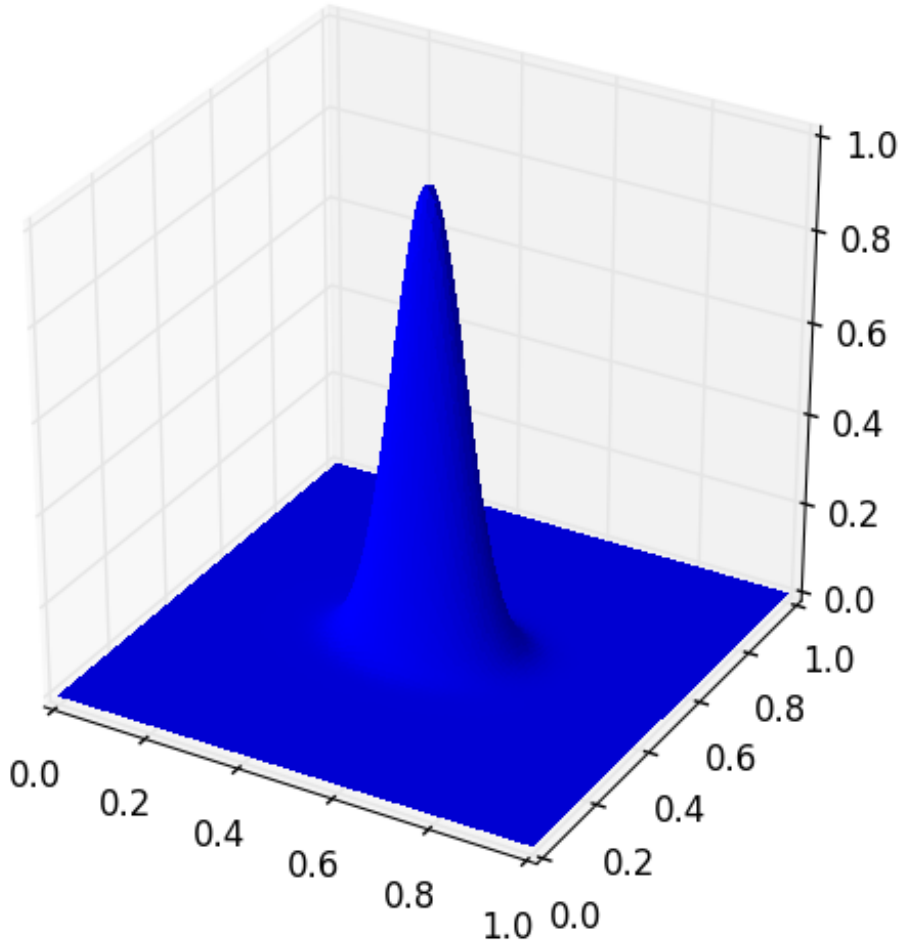


Figure 5.10: Two-dimensional print of the *Genz "Gaussian"* function with $m = 0.5$ and $c = 2$.

The test results from the center case can be seen in Figure 5.11, 5.12 and 5.13.

With $d = 2$ (see Figure 5.11) and higher amounts of grid points, while the standard *combination technique* is at a relative error of 10^{-13}, the dimension-wise spatial adaptive refinement strategy flattens and stays around a relative error term of 10^{-3}. But similar to the corner case, the adaptive algorithm performs better with fewer amounts of points, in this case up to around $3 \cdot 10^2$ (only comparing the $\ell_{\text{min}} = 1$ setups). With $d = 3$

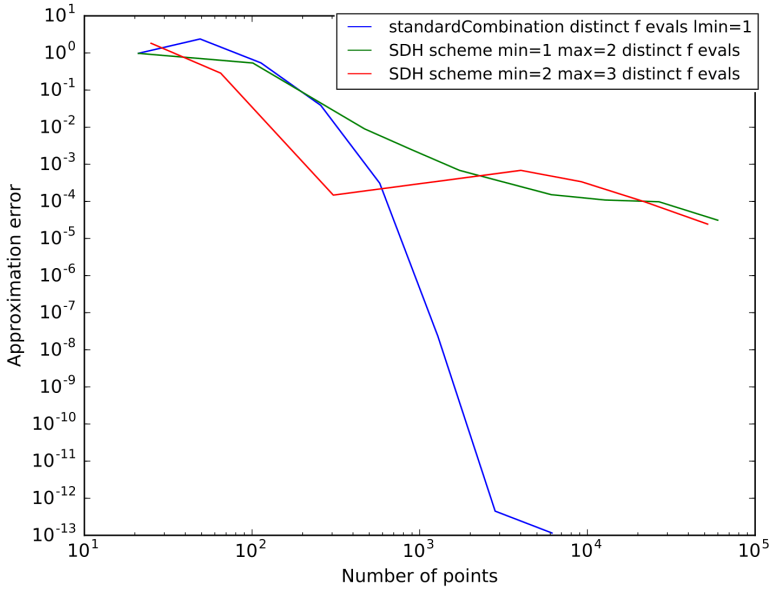


Figure 5.11: The center test case when running with $d = 2$ and $\epsilon_{\text{tol}} = 10^{-10}$.

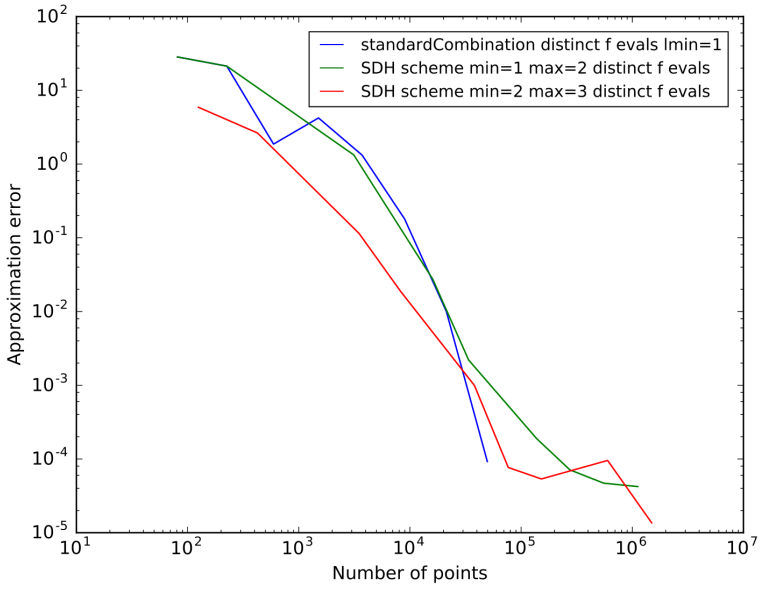


Figure 5.12: The center test case when running with $d = 3$ and $\epsilon_{\text{tol}} = 10^{-10}$.

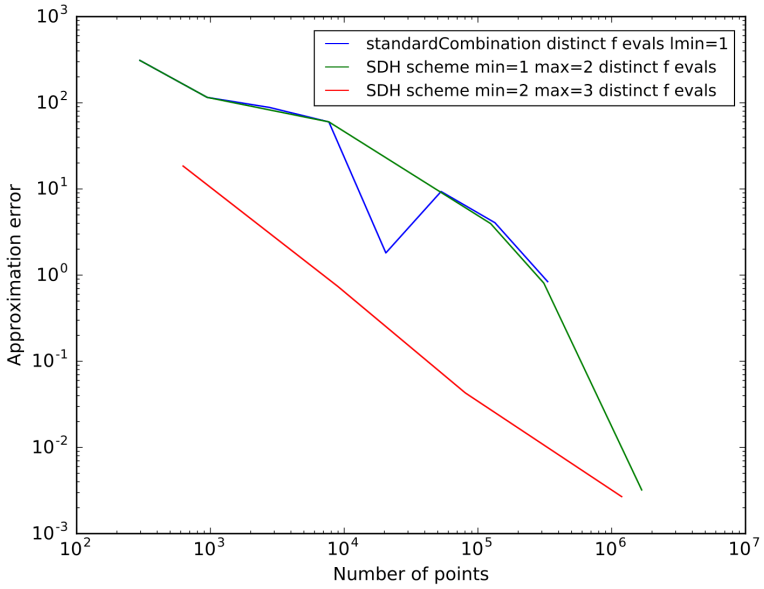


Figure 5.13: The center test case when running with $d = 4$ and $\epsilon_{\text{tol}} = 10^{-7}$.

(see Figure 5.12), the results are better, as now, the standard *combination technique* only consistently performs better with more than approximately 10^4 grid points. When having a dimensionality of four, the results differ. While the adaptive algorithm with $\ell_{\min} = 1$ seems to be nearly equally good in relation the standard *combination technique* with $\ell_{\min} = 1$, the red line (adaptive algorithm with $\ell_{\min} = 2$) seems to perform better. A test case with standard *combination technique* with $\ell_{\min} = 2$ verifies the same reason as in the corner case.

5.2 Griebel

Another function, whose variation grows exponentially, was used for testing purposes. The from now on called *Griebel* function [taken from 12] is defined as

$$x = x_0, \ldots, x_{d-1} \tag{5.3}$$

$$f(x) = 1 + \left(\frac{1}{d}\right)^d \cdot \prod_{i=0}^{d-1} x_i^{1/d} \tag{5.4}$$

For a three-dimensional plot of the *Griebel* function, see Figure 5.14.

As done before, the Griebel function was tested with the dimensions two to four, as seen in Figure 5.15, 5.16 and 5.17.

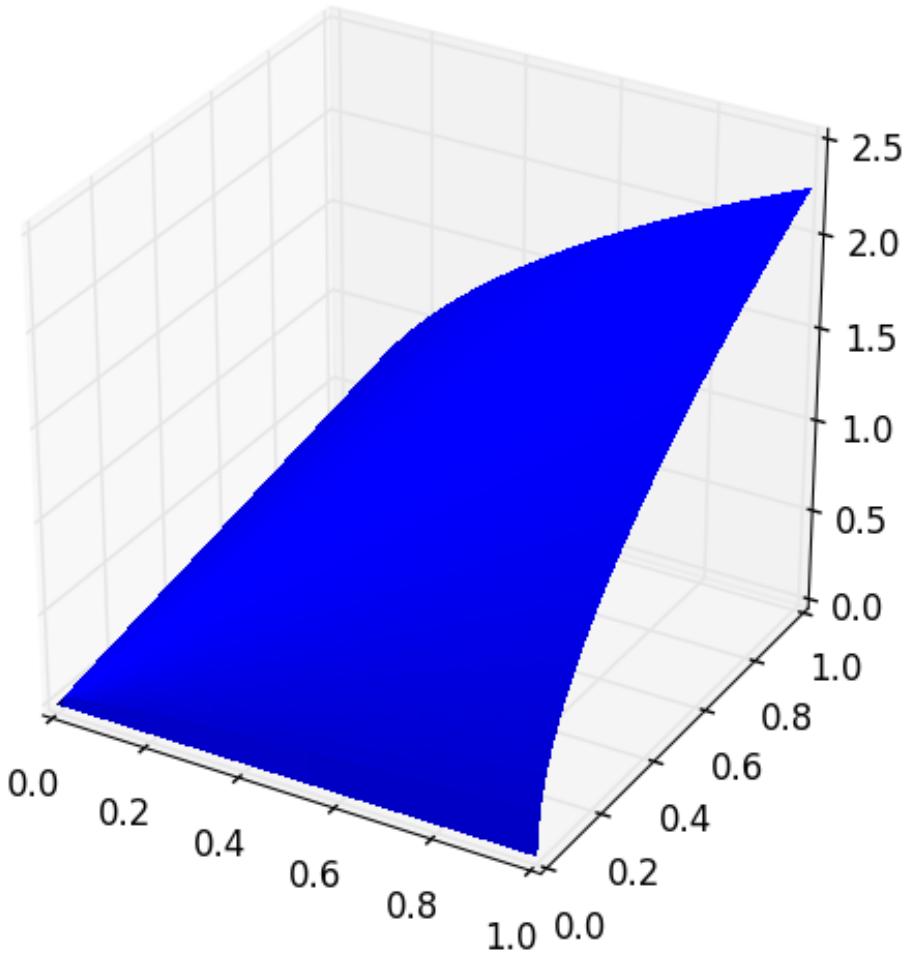


Figure 5.14: Two-dimensional print of the *Griebel* function.

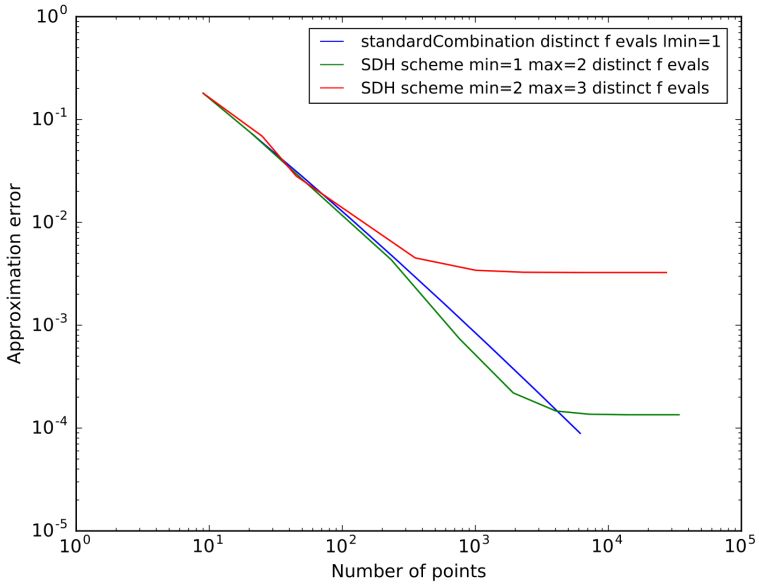


Figure 5.15: A Griebel test case when running with $d = 2$ and $\epsilon_{\text{tol}} = 10^{-12}$.

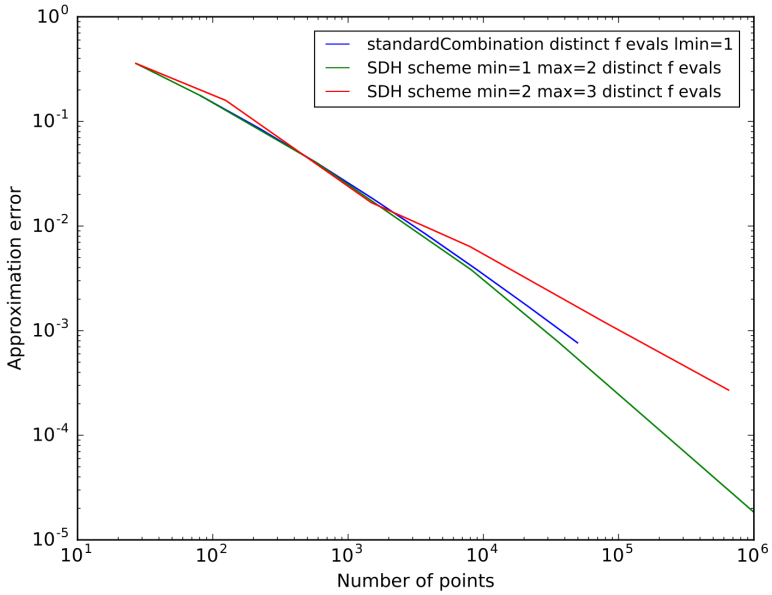


Figure 5.16: Result of a Griebel test case when running with $d = 3$ and $\epsilon_{\text{tol}} = 10^{-8}$.

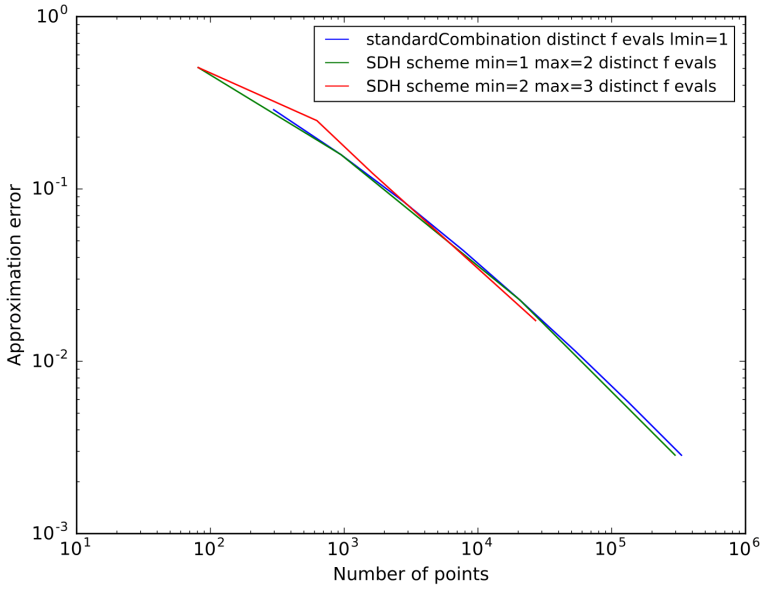


Figure 5.17: Griebel test case with $d = 4$ and $\epsilon_{\text{tol}} = 10^{-6}$.

With $d = 2$, while the standard *combination technique* does not flatten at all, the adaptive algorithm seems to have an error limit. Interestingly, in both the center- and corner case, the $\ell_{\text{min}} = 2$ approach with the adaptive algorithm performed better than the corresponding $\ell_{\text{min}} = 1$ algorithm. In the *Griebel* test case, the adaptive algorithm with $\ell_{\text{min}} = 2$ (red line) performes worse. Furthermore, with higher dimensionality, both $\ell_{\text{min}} = 1$ and $\ell_{\text{min}} = 2$ setup of the adaptive algorithm performs better. It is worth pointing out that in both $d = 2$ and $d = 3$, the adaptive algorithm with $\ell_{\text{min}} = 1$ at every number of points is at least as good as the standard *combination technique* with $\ell_{\text{min}} = 1$.

5.3 Miscellaneous

To test whether numerical issues occur while building the hierarchical basis functions or evaulating them, a test with $r_{\text{tol}} = 0$ is run. The algorithm refines every area in every dimension every step and should therefore have the same results as the standard *combination technique*, as they have the exact same resulting *sparse grid*.

As can be seen in Figure 5.18, the green line representing the adaptive algorithm with $\ell_{\text{min}} = 1$ is exactly on the blue line (standard *combination technique* with $\ell_{\text{min}} = 1$) for a while, but drifts apart from it at around 10^3 points. A numerical issue is therefore guaranteed from at least that number of points forward.

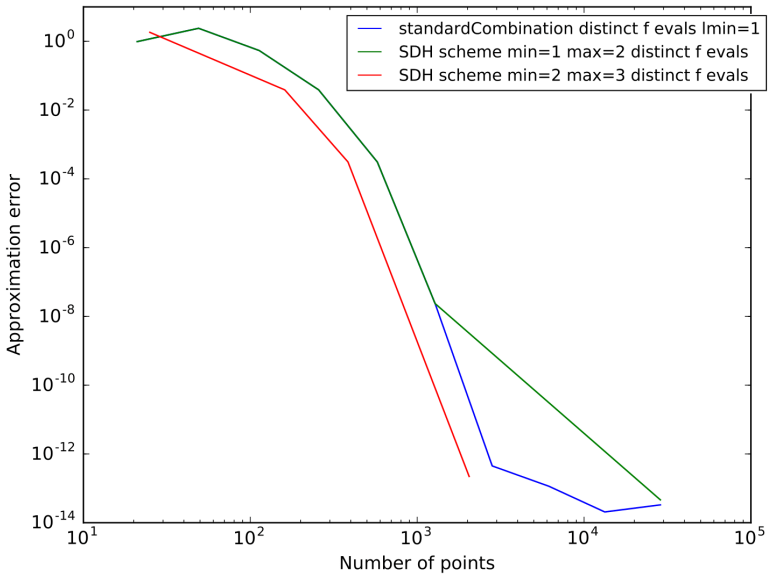


Figure 5.18: A center case test result using $dim = 2$, $\epsilon_{\text{tol}} = 10^{-12}$ and $r_{\text{tol}} = 0$.

When printing the partial integrals of the individual grids and the whole integral solution while running the adaptive algorithm, it can be seen that the solution of the *combination technique* using the adaptive algorithm often were worse than many partial integrals.

6 Conclusion

This chapter summarizes the important lessons that can be learned from this and gives an outlook on how to potentially improve the concept.

6.1 Summary

A *dimension-wise spatial-adaptive* algorithm seems to perform better in relation to the standard *combination technique* with higher dimensional problems and fewer grid points. That makes this approach promising, as a dimensionality of $d \leq 4$ is too low for many *sparse grid* problems, making $d > 4$ cases very probable of performing better than the presented tests. Furthermore, as many practical applications cannot afford a high amount of points, performing better with fewer grid points can be more significant than with many grid points.

There are still multiple open issues that have to be tackled in order to optimize the strategy. By testing further variations and tweaking the algorithm, there is probably more potential that can be utilized from a *dimension-wise spatial-adaptive refinement* strategy.

6.2 Outlook

There are some direct problems encountered. The numerical issues proven to exist have to be tackled for this algorithm to improve. As the adaptive algorithm results in a higher relative error than the standard *combination technique* at many fixed number of grid points, the adaptive algorithm did not refine optimally. Therefore, other techniques or variants to potentially improve the *dimension-wise spatial-adaptive refinement* decision-making must be considered. For example, it might improve the method to use the maximum error value of a stripe of a dimension of all grids rather than adding up all error values of a stripe for each grid individually.

Another aspect that probably can be improved is the efficiency of the program code written for this thesis. As an example, the program calculates the whole hierarchization anew after every refinement which can be optimized. With better runtime performance, more detailed tests could be run more easily.

Even with the presented setup of an *dimension-wise spatial-adaptive refinement* algorithm, one might consider using it with higher dimensional *sparse grid* problems and rather low amounts of function evaluations available. As of right now, the adaptive method performes weaker than the standard *combination technique* when having access to many points. Should the strategy be more improved, it could end up being better than the standard *combination technique* in every higher dimensional *sparse grid* case.

List of Figures

List of Tables

Bibliography

[1] H.-J. Bungartz and M. Griebel. "Sparse grids". In: *Acta Numerica* 13 (2004), pp. 147–269. DOI: 10.1017/S0962492904000182.

[2] M. Bader. *Algorithms for Scientific Computing – 1D Hierarchical Basis*. 2017. URL: https://www5.in.tum.de/lehre/vorlesungen/asc/ss17/hierbas_1D.pdf.

[3] M. Molzer. "Implementation of a Parallel Sparse Grid Combination Technique for Variable Process Group Sizes". Bachelor's thesis. MSE, Technische Universität München, Jan. 2018. URL: https://www5.in.tum.de/pub/BA_Molzer.pdf.

[4] T. Gerstner and M. Griebel. "Sparse Grids". In: *Encyclopedia of Quantitative Finance* (2008).

[5] J. Garcke. "Sparse Grids in a Nutshell". In: *Sparse Grids and Applications*. Ed. by J. Garcke and M. Griebel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 57–80. ISBN: 978-3-642-31703-3.

[6] J. Garcke and M. Griebel. "Spatially Adaptive Refinement". In: *Sparse Grids and Applications of Lecture Notes in Computational Science and Engineering* (2012), p. 243262.

[7] M. Griebel, M. Schneider, and C. Zenger. "A combination technique for the solution of sparse grid problems". In: *Iterative Methods in Linear Algebra*. IMACS, Elsevier, North Holland, 1992, pp. 263–281.

[8] M. Hegland, J. Garcke, and V. Challis. "The combination technique and some generalisations". In: *Linear Algebra and its applications* (2006), pp. 249–275.

[9] M. Heene, A. Parra Hinojosa, M. Obersteiner, H.-J. Bungartz, and D. Pflüger. "High Performance Computing in Science and Engineering ' 17". In: ed. by W. Nagel, D. Kröner, and M. Resch. Springer-Verlag, Mar. 2018. Chap. EXAHD: An Exa-Scalable Two-Level Sparse Grid Approach for Higher-Dimensional Problems in Plasma Physics and Beyond, pp. 513–529. ISBN: 9783319683935.

[10] J. Noordmans and P. W. Hemker. "Application of an Adaptive Sparse-Grid Technique to a Model Singular Perturbation Problem". In: *Computing*. Vol. 65. Springer, 2000, pp. 357–378.

[11] J. D. Jakeman and S. G. Roberts. "Local and Dimension Adaptive Sparse Grid Interpolation and Quadrature". In: *CoRR* abs/1110.0010 (2011).

[12] T. Gerstner and M. Griebel. "Numerical integration using sparse grids". In: *Numerical Algorithms* 18.3 (Jan. 1998), p. 209. ISSN: 1572-9265. DOI: 10.1023/A:1019129717644. URL: https://doi.org/10.1023/A:1019129717644.